Pro*k*rustean graph: An efficient data structure used to extract sequence information for arbitrary *k*-mer size ranges

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. Despite the widespread adoption of k-mer-based methods in bioinformatics, a fundamental question persists: How to elucidate the structural transition of a k-mer set when the order switches to k'? Attaining a generalized answer has significant implications to k-mer-based methods where the influence of k have been empirically analyzed (eg. in areas of assembly, genome comparison, etc.).

We unravel the problem with a principle: k-mers and k'-mers can be grouped by their co-occurrences, and those in the same group behave similarly regardless of applications. This concept is embodied in a model, the Pro*k*rustean graph, which embraces all similarity information of a given sequence set and has a channel to access k-mers of any k. This gives us a theoretical framework in which we can understand the presence and frequency of k-mers as k changes.

Practically, a Pro*k*rustean graph is a space efficient data structure that can quickly be queried to extract essentially arbitrary information regarding k-mers with time complexity independent of k-mer size range. We provide a series of examples that perform in competitive time and space when compared to purpose-built tools that operate on a single k, such as KMC and GGCAT. For example, with large read sets, we can count all k-mers for $k = 30 \ldots 150$ in $\simeq 30$ seconds in comparison to KMC requiring $\simeq 30$ seconds for a single k. Similarly, on long read sets, we can count all unitigs in $\simeq 30$ seconds for the entire range $k = 30 \ldots 50000$, a task that is prohibitively burdensome for GGCAT.

Our construction algorithm of Pro*k*rustean graph utilizes an (extended) Burrows-Wheeler transform as input, is easily parallelizable, and operates in $O(N)$ time where N is the cumulative input sequence length. We provide theoretical justification that the size of a Pro*k*rustean graph is $O(N)$, and in practice, is significantly smaller than N.

All code and algorithms are publicly accessible at: `https://github.com/KoslickiLab/prokrustean`.

1 Introduction

How to formulate the transition of k-mers to k'-mers of a sequence set? The fundamental question in bioinformatics has been surprisingly elusive. Despite the huge impact of the primary parameter k in k-mer-based methods, the decision of k is mostly left to practitioners. The challenges are (1) choosing the best k or (2) utilizing multiple k's. Both are not rigorously achievable until the structure of k-mers against k'-mers is modeled. For example, optimal parameters for fixed k has been described in alignment [25], but finding the optimal k in assembly and other areas mostly relies on heuristics [7]. With the theoretical void, the recent theme of constructing de Bruijn graphs with multiple k's is intrinsically challenging [11, 24]. Similarly, in the field of metagenomics, empirical methods have attempted to find 'optimal' k values (eg. by phylogenetic references [21]), while others have called for utilizing multiple k sizes to characterize organisms absent from reference databases [17]. In the area of k-mer counting, most tools focus on a specific k [18], but recent tools argue for additional insight extracted from a range of k-mer sizes [5].

We aim to address precisely how changing k-mer size affects k-mer sets of sequences; *Figure* 1 summarizes the theme of this work. After constructing a data structure, named the Pro*k*rustean graph, we demonstrate that one can implement methods agnostic to k-mer size and with performance independent of the k range chosen; examples include counting k-mers and counting maximal unitigs of de bruijn graphs for any/all k. We show that our approach is not only competitive in terms of speed/memory to purpose-built tools when considering a single k, but outperforms them when considering a range of k's. We also show how it can be used to create new k-mer-based functionalities (eg. find k that maximizes the number of unitigs).

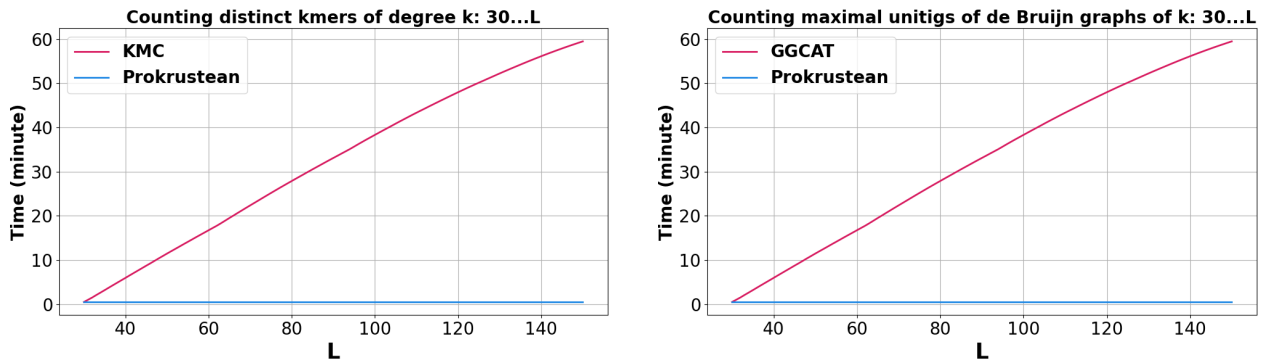

Fig. 1: The performance of Pro*k*rustean graphs for two representative functionalities were compared with state-of-arts: KMC [16] is a k-mer counting tool and GGCAT [10] constructs compacted de Bruijn graphs . Both were iterative called to extract k-mer/unitig counts of $k = 30 \ldots L$ of the sequences (4.5gb accumulated length, ERR3450203), while our algorithms (Section 4) using Pro*k*rustean graphs extract the whole information in a single run.

The name Pro*k*rustean is motivated from a tale of Greek mythology, the bed of Procrustes. Procrustes fitted guests to his bed either by stretching them or cutting their legs. The brutal k have been even worse that guests (sequences) are unaware of how much they lose their parts. Now, the guests are blessed that a new bed (Pro*k*rustean graph) is arrived to quantify the loss in advance, allowing them small comforts and reliefs. Well, of course, only until their fate arrives anyway and cut them into mers.

2 Approach

This section contains a high level description of the key concepts and intuition behind the approach. A principle of k is described with two main ingredients: stratification, and reflection.

2.1 Principle of the transition of k

Consider the behaviors of two k-mer sets generated from same sequences: 30-mers and 31-mers. Do two de Bruijn graphs made from each k-mer set show notable difference in structure? Or, do two distributions of pairwise Jaccard similarities between sequences' k-mer sets show significantly different variation? The answer is no, supported by empirical evidence and insights the behaviors of 30-mers and 31-mers are largely similar. Then, in the opposite perspective, gather $(30, 31)$-mers that contribute to disparities in the two applications, de Bruijn graphs and Jaccard similarities. Do the $(30, 31)$-mers exhibit quite random patterns depending on

the applications, or share some common characteristics independent from applications? This one is difficult to answer, and underscores the gap in the understanding of k-mers.

We approach the problem by recognizing the inherent trade-off in choosing the parameter k. The **Desiderata** listed below represent the key qualities that practitioners commonly value in k-mer based methods, applicable across a range of contexts.

Desideratum I. Each sequence is reflected by distinct k-mers.

This first desired property favors larger k's because longer k-mers are more likely to be unique. For example, in Jaccard similarities, which is a set-based measure, too small of k's can lead to duplications of k-mers and dilute the accuracy. Similarly, in de Bruijn graphs used for assemblies, smaller k values may result in more cycles due to the ambiguity in the location represented by each k-mer.

Desideratum II. Each common substring between sequences is reflected by some k-mers.

This second desired property favors smaller k's. In a Jaccard index between two sequences, too large k values can drive the index to zero, as common k-mers are dominated by unique ones. In de Bruijn graphs, a larger k value may lead to an increase in disconnected components due to fewer similarities supporting the edges.

While these **Desiderata** make intuitive sense, they do not explain the specific behavior of 30-mers and 31-mers, especially in a scenario where two almost equivalent (string-wise) 30-mers and 31-mers turn out to act differently in applications. The hint is in the progression: $k = 31 \to 30 \to \cdots \to 1$. Start with a 31-mer K and remove its last letter at each stage and see how its role evolves. Let K appear at positions p_X and p_Y in sequences X and Y. The occurrences of K may persist until a certain critical point, say $k = 25$, when K may appear at p_Z of a different sequence Z. We can easily imagine that the role of K diverges significantly between $k = 26$ and $k = 25$. The event can be summarized as "The existence of K implies the existence of some common substring of just X and Y until $k = 26$, but the implication breaks from $k = 25$."

Principle. If two *mers* share a superstring and have the same frequency of occurrence as the superstring (i.e. co-occur with the superstring), the *mers* behave similarly in any k-mer-based method. That is, the meaning of <u>reflect</u> in **I** and **II** is limited: The sequence in **I** or the common substring in **II** are <u>reflected</u> by k-mers that co-occur with them.

This **Principle** tells that all we need is to group co-occuring *mers* to explain the transition of k's. Also, the principle implies that once the groups are constructed, a few representative strings - sequences and common substrings - explain the behaviors of all *mers*. Therefore, the subsequent sections are devoted to construct a compact structure that defines the set of representative strings and their connection to all *mers* of all k's.

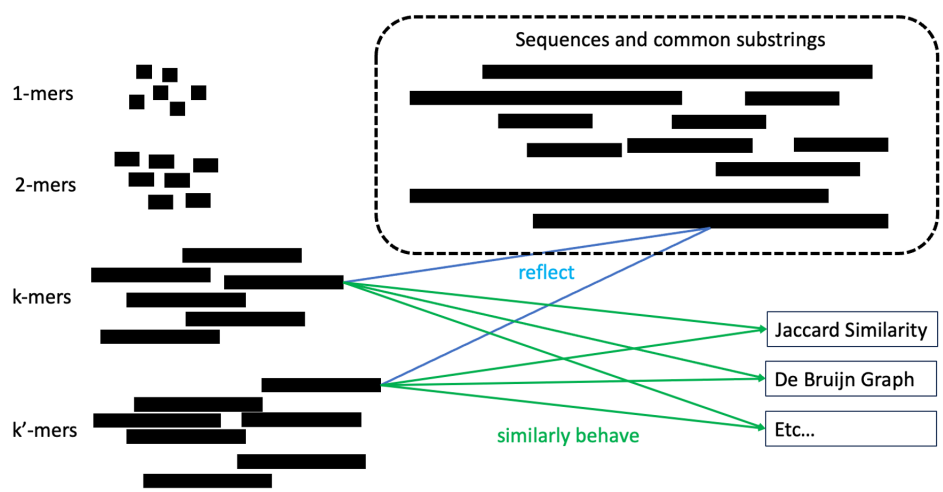

Fig. 2: Sketch of the **Principle**. All *mers*, regardless of k's and applications, are grouped by strings which they reflect. <u>reflect</u>, represented by blue arrows, is not limited to a substring relationships, but also co-occurrences. The roles of the *mers* reflecting the same string then similarly behave in any applications as denoted by green arrows.

2.2 Occurrences Of Maximal Repeat

Maximal repeats play an important role in achieving the principle in *section* 2.1. Consider organizing all patterns found in genetic sequences without any biological knowledge. Then maximal repeats are indeed compact representations of similarities because any substring similarity is included in some maximal repeat. One promising property introduced earlier is that the number of maximal repeats is less than N for a sequence of length N([15]). However, their occurrences can be as many as $O(N^2)$ presenting a significant challenge, and the occurrences have not been properly appreciated.

A frustrating part in previous work about maximal repeats is that the occurrences are claimed to be stored in a linear space, but the approaches still struggle with the intrinsic quadratic outputs, leaving practitioners to manage this complexity before utilizing them. For example, [8] and [1] achieve maximal repeat occurrences in linear time but with quadratic output. [2] and [13] achieve linear space representation but lack practical algorithms for its utilization. [9] introduces an $O(N)$ computation method via recursive formulae, but its implementation proves challenging.

The aim has to be specific: How to properly use all occurrences of maximal repeats to improve k-mer-based methods? This is because many methods have already achieved the goal to some limited extent. The concept introduced in the next section, stratification, accomplishes the goal.

2.3 Stratification

Setting aside k-mers for the moment, the key idea of stratification is to recursively capture locations of sequence similarities in a compact, hierarchical representation. For instance, consider two sequences: ACCCT and GACCC. Their similarity can be concisely described with: "Both sequences have ACCC because other similarities (e.g. CC) are covered by ACCC". When introducing another sequence TCCCG, it quickly becomes challenging to make such compact statements that captures all similarities in these strings. Consider two possible cases:

1. ACCC is in 2 sequences, and CCC is in all 3 sequences.
2. ACCC is in 2 sequences, and CCC is in 1 sequence TCCCG and in 1 string ACCC.

The first case uses 5 units of "information" (here, given in terms of number of substrings) but the second case uses 4 units while preserving the same meaning. This is the intuition behind stratification as depicted in Figure 3.

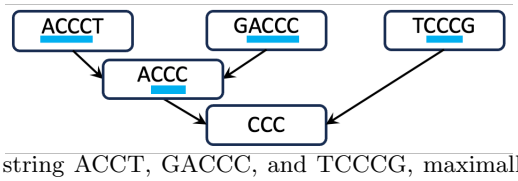

Fig. 3: Stratification. Given three string ACCT, GACCC, and TCCCG, maximally matching subregions (indicated with blue underlining) are used to label terminal nodes of each edge. This process repeats hierarchically, hence the name *stratification*.

We can visualize the hierarchy of a string set using a graph, with strings as node labels and regions of maximal length shared with adjacent nodes (i.e. *covered* regions) highlighted (in blue in Figure 3). In this example, ACCCT is covered by the region of ACCC. Similarly, the region corresponding to CCC in the node ACCC is covered by the terminal node CCC.

The stratification of a set of strings is then the graph containing all possible covered regions, starting with the initial set of strings and proceeding through all subsequently introduced nodes/substrings. Naïvely storing all similarities/overlaps in a set of strings would result in requiring $O(N^2)$ storage. However, by keeping track of the maximal overlaps and using this hierarchical structure, we can reduce this to $O(|\Sigma|N)$ where Σ denotes the alphabet.

2.4 Reflection

Reflection characterizes how the similarities identified in stratification are observed in terms of k-mers and hence affected by k. Consider the example given in Figure 4, recalling that each label is *covered* with blue regions when similarities are found, and that there are nodes in lower levels labeled with the strings captured by blue regions. For a given k-mer size, an *uncovered* region is a region of length at least k in the label

of a node that does not appear in any descendants' node labels (see the red underlining in Figure 4 where two different k-mer sizes are used to show uncovered regions). These red, uncovered regions contain distinct, unique k-mers. Observe that the strings in red regions are distinct from each other. The key observation is that every k-mer appears multiple times in blue/covered regions but only once in some red/uncovered region. Note, appearing once in a red region does not mean the k-mer appears once, but appears the same as the superstring in the box.

The figure also reveals the rule of reflection behind increasing the k-mer size from $k = 3$ to $k = 4$. In the right graph labeled $k = 4$, some information is lost because the node CCC is now ignored since it is shorter than 4 so by definition cannot contain a $k = 4$ uncovered region. Also, the blue regions of CCC that exist in the ACCC and TCCCG in the graph $k = 3$, are absent in the graph $k = 4$. Instead, ACCC and TCCCG are filled with red regions. Note that TCCCG includes two 4-mers TCCC and CCCG, meaning reflection merges 4-mers together, which becomes an important feature in the context of sequence assembly later. In conclusion, collecting all distinct 3-mers or 4-mers, one would simply enumerate all nodes and read the red regions.

Fig. 4: The principle of reflection. Reflection which is dependent on the size of k is depicted for $k = 3$ with the graph on the left, and $k = 4$ with the graph on the right. Blue underlining depicts covered regions, and red underlining depicts uncovered regions.

We chose the terminology "reflection" since when a particular k-mer size is chosen, this choice is reflected in how similarities are viewed in the underlying stratification graph.

3 Method

Here, we present the technical treatment of the ideas presented in the approach Section 2.

Important note: all proofs are included in the supplement.

3.1 Basic Notations

Let $\Sigma = \{A, C, T, G\}$ be our alphabet, $\Sigma^n = \{w_1 w_2 \cdots w_n | w_i \in \Sigma\}$ the set of all n-length strings, and $\Sigma^* = \bigcup_{n=1}^{\infty} \Sigma^n$ all finite length strings on the alphabet. We use $S \in \Sigma^*$ to represent a finite-length string, and $\mathcal{U} \subset \Sigma^*$ for a set of strings. Consider the following preliminary definitions:

Definition 1. – *A region in the string S is a triple (S, i, j) such that $1 \leq i \leq j \leq |S|$.*
- *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:*

$$\mathrm{occ}_{\mathcal{U}}(S) := \{(S', i, j) \mid S' \in \mathcal{U},\ str(S', i, j) = S\}.$$

- *The k-mers of S is a set of substrings of length k of S: $\mathrm{mers}_k(S) = \{str(S, i, j) \mid |(S, i, j)| = k\}$.*
- *S is a maximal repeat in $\mathcal{U}$ if it occurs at least twice and more than any of its extensions: $|\mathrm{occ}_{\mathcal{U}}(S)| > 1$, and for all S' a superstring of S, $|\mathrm{occ}_{\mathcal{U}}(S)| > |\mathrm{occ}_{\mathcal{U}}(S')|$.*

I.e., given a fixed set $\mathcal{U}$, 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 the original string S.

3.2 Stratification

With these preliminaries, we can now rigorously define stratification as in Section 2.3.

Definition 2. *A region (S, i, j) of a string S is* <u>*stratifying in $\mathcal{U}$*</u> *if:*

$$|\operatorname{occ}_{\mathcal{U}}(str(S, i, j))| > |\operatorname{occ}_{\mathcal{U}}(S)| > 0 \tag{1}$$

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

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

We call the string of a stratifying region $str(S, i, j)$ a <u>*stratum*</u> *in $\mathcal{U}$.* <u>*strata*</u> *is the plural form.*

Note: We often omit 'in $\mathcal{U}$' and simply use 'stratifying' when the context specifies the string set $\mathcal{U}$. In words, the string of a stratifying region appears more frequently in the string set $\mathcal{U}$ than its 'parent' string S, but each extension appears at most as frequently as S (noting that a substring/region cannot appear less frequently than the string in which it is contained).

Note that the inequality in (1) leads to the recursive structure of strata. After calculating the strata of $\mathcal{U}$, we can add these to $\mathcal{U}$ and then proceed recursively. At the top level (just calculating the strata of $\mathcal{U}$, i.e. the top level of the stratification graph) this roughly captures supermaximal repeats: these are maximal repeats not included in any other maximal repeat. This feature reveals the primary properties comprising the foundation of this project.

In example Figure 3, we have that $(\text{ACCCT}, 1, 4)$ is stratifying in $\mathcal{U} = \{\text{ACCCT, GACCC, TCCCG}\}$ but not stratifying in $\mathcal{U}' = \{\text{ACCCT, GACCC, TCCCG, ACCC}\}$. Similarly, $(\text{ACCC}, 2, 4)$ is stratifying in $\mathcal{U}' = \{\text{ACCCT}, \text{GACCC}, \text{TCCCG}, \text{ACCC}\}$ but not in $\mathcal{U} = \{\text{ACCCT}, \text{GACCC}, \text{TCCCG}\}$.

Definition 3. *$\mathcal{S}_{\mathcal{U}}$ is the set strata of $\mathcal{U}$ if:*

1. *for each region (S, i, j) stratifying in $\mathcal{U}$ where $S \in \mathcal{U}$, $str(S, i, j) \in \mathcal{S}_{\mathcal{U}}$ and*
2. *for each region (S, i, j) stratifying in $\mathcal{U}$ where $S \in \mathcal{S}_{\mathcal{U}}$, $str(S, i, j) \in \mathcal{S}_{\mathcal{U}}$.*

We collect the strata with the original sequences and define $\text{StraStr}(\mathcal{U}) := \mathcal{U} \cup \mathcal{S}_{\mathcal{U}}$.
We define the corresponding regions as: $\text{StraRgn}(\mathcal{U}) := \bigcup_{S \in \text{StraStr}(\mathcal{U})} \{(S, i, j) | (S, i, j) \text{ is stratifying in } \mathcal{U}\}$.

The graph-like structure mentioned in Section 2 starts to be made clear: the set of nodes of the stratification graph corresponds to that of $\text{StraStr}(\mathcal{U})$, while edges connect two nodes if the the string of one corresponds to a region that is stratifying in the string of the other, which are $\text{StraRgn}(\mathcal{U})$.

The following three theorems together imply that sequence similarities in $\mathcal{U}$ are exhaustively and compactly identified by stratification.

Theorem 1 (Complete1). *A string R is a maximal repeat if and only if R is a stratum.*

Theorem 2 (Complete2). *A region (S, i, j) ($S \in \mathcal{U}$) is an occurrence of a maximal repeat if and only if*

1. *(S, i, j) is the whole region of S and $occ_{\mathcal{U}}(S) > 1$ or*
2. *(S, i, j) is stratifying or*
3. *There exists a stratum R such that (S, i, j) is stratifying relatively in R. That is, there exists an extension $(S, i', j') \in \operatorname{occ}_{\mathcal{U}}(R)$ such that $(R, i - i', |R| - (j' - j))$ is stratifying.*

Theorem 1 says each maximal repeat is identified as a stratum and Theorem 2 implies their each occurrence is accessible in the recursion structure, as desired in Section 2.2. Consider an arbitrary region. Whether or not the region is an occurrence of a maximal repeat is determined by stratification. If the region is an occurrence of a maximal repeat, it must be either stratifying or is stratifying in some other stratum. So, the region is discovered in the chain of the recursion derived by stratification.

Theorem 3 (Compact). *For $\mathcal{U}$ a set of sequences of total length N on the alphabet Σ,*

$$|\mathcal{S}_{\mathcal{U}}| < N \text{ and } |\text{StraRgn}(\mathcal{U})| < 2|\Sigma|N.$$

Theorem 3 indicates that stratification is a compact $O(N)$ way to express all similarities by representing all occurrences of maximal repeats in significantly less space than the usual $O(N^2)$ way to represent them.

3.3 Reflection

Reflection is complementary to stratification, but takes into consideration k-mer size.

Definition 4. *A region (S, i, j) of a string S is* <u>*k-reflecting in $\mathcal{U}$*</u> *if:*
for each k-mer K in $str(S, i, j)$,

$$|\operatorname{occ}_{\mathcal{U}}(K)| = |\operatorname{occ}_{\mathcal{U}}(S)| > 0, \tag{3}$$

and for each extension (S, i', j'), there exists a k-mer K in $str(S, i', j')$ that,

$$|\operatorname{occ}_{\mathcal{U}}(K)| > |\operatorname{occ}_{\mathcal{U}}(S)|. \tag{4}$$

So the string of a k-reflecting region maximally contains k-mers that co-occur with S. Similar to a stratum, the string of a k-reflecting region $str(S, i, j)$ is a <u>k-reflectum</u> and multiple such strings are <u>k-reflecta</u>.

Assuming stratification is already constructed, k-reflecting regions are efficiently obtained as 'leftovers' of stratifying regions. See *Figure* 5 for a concrete example. We provide an instructive definition to describe the decompositions in the figure that find all k-reflecting regions of a string S. Supplementary Algorithm 3 contains a formal algorithm.

Definition 5. *The* <u>*k-decomposed regions of S in $\mathcal{U}$*</u> *containing k-reflecting regions are collected as following:*

1. *Collect stratifying regions of length k or above in a loop, sequentially.*
2. *Collect k-reflecting regions in the same loop by covering any length k region not covered by stratifying regions: The regions are maximally extended but do not intersect with any region more than $k-1$.*

The two lemmas below show the decomposition efficiently covers areas of k-mers.

Lemma 1 (Mer Cover). *Any length k region in S is included in a k-decomposed region of S.*

Lemma 2 (Computation). *Decomposing regions in S takes $O(|\{\text{stratifying regions in } S\}|)$ time.*

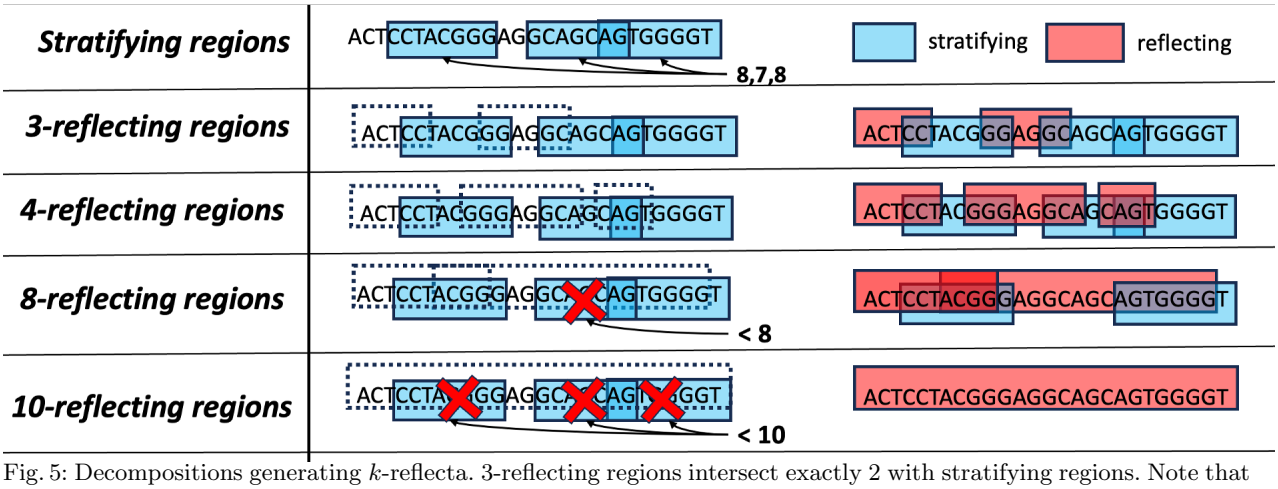

Fig. 5: Decompositions generating k-reflecta. 3-reflecting regions intersect exactly 2 with stratifying regions. Note that not only 'gaps' between stratifying regions are considered because the 4-reflecting region of CAGT is placed above an intersection AG. The dotted rectangles represent such captured areas. As k grows, reflecting regions dominate. 10-reflecting region occupies the whole area.

3.4 Pro*k*rustean graph and Pro*k*rustean decomposition

We finally build models of stratification and reflection. The graph is a mere replication of stratification.

Definition 6. *The* <u>*Prokrustean graph*</u> *of $\mathcal{U}$ is a directed multigraph (V, E) with sizes and intervals labeling vertices and edges, respectively:*

- *$v_S \in V$ if and only if $S \in \mathrm{StraStr}(\mathcal{U})$,*

– $e_{S,i,j} \in E$ *if and only if* $(S, i, j) \in \mathrm{StraRgn}(\mathcal{U})$, *which directs from* v_S *to* $v_{str(S,i,j)}$.
– $size(v_S) = |S|$, *and* $interval(e_{S,i,j}) = (i, j)$.

The <u>*Prokrustean decomposition of S*</u> *in* $\mathcal{U}$ *is a process of generating a local graph* $G_S := (V_S, E_S)$ *of a Prokrustean graph* $G := (V, E)$ *of* $\mathcal{U}$*:*

$$Pdecompose(v_S, k; G) \to G_S \tag{5}$$

- $V_S = \{v_S\} \cup \{\ v_{str(S,i,j)} |\ (S, i, j)$ *is a k-decomposed region* $\}$
- $E_S = \{(v_S, v_{(S,i,j)}, i)\ |\ (S, i, j)$ *is a k-decomposed region* $\}$
- $size(v)$ *and* $interval(e)$ *are equally defined as in* G.
- $type(v) :=$ *'stratum' if* $v \in V$ *and 'reflectum' if* $v \in V_S \setminus V$
- $type(e) :=$ *'stratifying' if* $e \in E$ *and 'reflecting' if* $e \in E_S \setminus E$.

A Pro*k*rustean graph describes relationships in stratification. Its vertices represent sequences and strata, and its edges denote stratifying regions. The labels *size* and *interval* imply that an implementation of the graph need not contain strings which are not required in algorithms before printing.

Next, the region decomposition of S is described as a local transformation of a Pro*k*rustean graph. The figure *fig.* 6 shows a concrete example of how the local graph transformation looks like for varying vertices and k's.

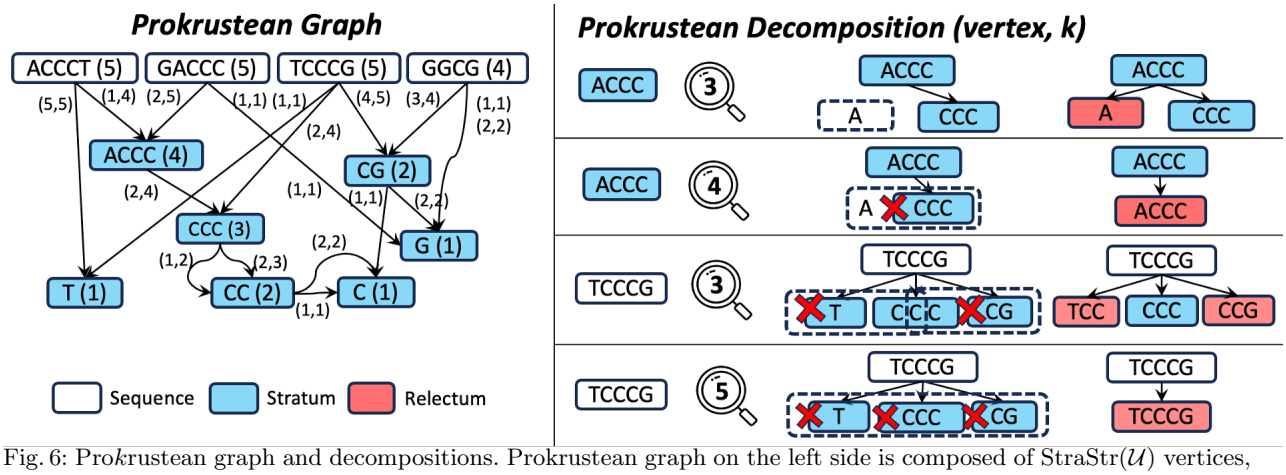

Fig. 6: Pro*k*rustean graph and decompositions. Prokrustean graph on the left side is composed of $\mathrm{StraStr}(\mathcal{U})$ vertices, meaning white sequences and blue strata vertices. Decompositions on the right are done on two vertices ACCC and TCCCG with varying k's generating red vertices. As k grows, red vertices dominate and swallow blue vertices, i.e. reflecta occupy the strings more and more. The dotted rectangles describe how reflecting regions are identified and sometimes 'cancel' stratifying regions. For example, the edge from ACCC to CCC survives only until $k = 3$, at the first line (ACCC, 3) and then a single red vertex comprises the whole decomposed region in the second line (ACCC,4), meaning the 4-reflectum ACCC occupies the entire string.

3.5 Principle revisited: similarly behaving k-mers and k'-mers.

Principle in section 2.1 is stated concretely.

Definition 7. *A string* S <u>*reflects*</u> *a string* S' *in* $\mathcal{U}$ *if:*

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

For example, a globally unique k-mer reflects a single sequence in $\mathcal{U}$. Such relationship is extended universally in the theorem below.

Theorem 4 (Universal reflection). *For any string* S *such that* $occ_{\mathcal{U}}(S) > 0$,

$$S \text{ reflects exactly one string in } \mathrm{StraStr}(\mathcal{U}).$$

Every string occurring in $\mathcal{U}$, say *mer*, regardless of its length, reflects exactly one sequence or maximal repeat. So, all possible *mers* are partitioned into disjoint groups centering on StraStr($\mathcal{U}$). The key in section 2.1 was to efficiently identify similarly behaving *mers*, because they are overwhelmingly many when all k's are considered. So, they are implicit in stratification and accessible in reflection. Concretely, algorithms work with StraStr($\mathcal{U}$), and locally access k-mers through k-reflectum if necessary. Below Lemma 3 and Lemma 4 describe how reflection contributes to the identification.

Lemma 3. *A k-reflectum and its substring k-mers together reflect the same string in* StraStr($\mathcal{U}$).

Lemma 4. *A k-mer K occurs in $\mathcal{U}$ if and only if K occurs uniquely in a k-reflectum in* StraStr($\mathcal{U}$).

4 Application

We demonstrate applications of the Pro*k*rustean graph by a variety of algorithms that efficiently compute some common (and some uncommon) tasks with k-mers. For time and space estimations, we use $|prokrustean| := max(|V|, |E|) = max(|\text{StraStr}(\mathcal{U})|, |\text{StraRgn}(\mathcal{U})|)$ to denote the cardinality of the prokrustean graph. $O(|prokrustean|) = O(|N|)$ holds in theory and experiments show $|prokrustean| < 0.1|N|$ with a proper *Lmin*: the minimum size of strata to be collected by the construction algorithm. We use three datasets: two short read datasets SRR20044276 & ERR3450203, and a long read metagenome SRR18495451. We compare to mature, purpose built tools KMC [16] (for counting k-mers) and GGCAT [10] (for building compacted de Bruijn graphs).

Pro*k*rustean graph construction

The construction algorithm of a Pro*k*rustean graphs in section 4 uses an (extended) Burrows-Wheeler transform (eBWT) of $\mathcal{U}$ [19] along with a wavelet tree [20]. The algorithm runs in $O(N)$ time and space. The memory usage measured is around $5N$ bits and $2 * |output|$ bytes.

dataset	sequences(N)	Lmin	time	memory	output	thread	time (wavelet)	time (build)	$\|StraStr\|$ & $\|StraRgn\|$
Short reads(SRR)	0.94M (72M)	1	29s	559mb	184mb	8	8s	21s	8M/21M
Short reads(SRR)	0.94M (72M)	20	13s	200mb	40mb	8	8s	5s	2.7M/5.4M
Short reads(ERR)	55M (4.5B)	30	28min	11gb	3.3gb	8	12min	16min	221M/407M
Long reads (SRRL)	0.83M (11.3B)	30	90min	43gb	14gb	8	20min	70min	890M/1.7B

Table 1: The results of construction algorithms. M and B means millions and billions. Note that the choice of *Lmin* makes outputs vary in lower k sizes, as shown in the two results of SRR20044276. Observe that $|StraStr|$ is much smaller than N, which are 'reflected' building blocks of algorithms.

The construction for future experiments is sufficiently efficient, but the wavelet tree's creation and rank function are time-intensive in the (wavelet) and (build) processes. We utilized the single-threaded wavelet tree from the SDSL project [14], but employing more advanced methods like [12] could notably enhance performance. The information in outputs ($|prokrustean|$) is almost always much smaller than N with a reasonable *Lmin*, allowing the efficiencies of algorithms in subsequent sections.

4.1 Collect distinct *k*-mers

We start from a fundamental application of k-mers. *Lemma* 4 proved that k-reflecta in $\mathcal{U}$ cover all of the k-mers found in $\mathcal{U}$, disjointly. So, collecting distinct k-mers is equivalent to collecting k-reflecta.

The time complexity is $O(|prokrustean|)$ because Lemma 2 indicates that *Pdecompose* takes time linear in outgoing edges for each v_S (i.e. stratifying regions of S), meaning each edge in E is considered only once. If printing k-mers is asked as in line 7, then the running time is linear to the accumulated length of distinct k-mers. We distinguish the functionalities to separate lower-level computations.

	dataset	time	memory	thread	time (compute)	time (load/save)	k
Prokrustean	Short reads(SRR)	7.7s	163mb	8	0.02ms	7.7s	20
	Short reads(ERR)	140s	12.2gb	8	4s	136s	30
	Long reads (SRRL)	5.5m	43gb	8	11s	5.4m	30
KMC	Short reads(SRR)	2.2s	297mb	8	2s	0.2	20
	Short reads(ERR)	39s	12gb	8	30s	9s	30
	Long reads (SRRL)	73s	12gb	8	64s	9s	30

Table 2: **Comparison with fixed k:** Distinct kmers were collected with Prokrustean graph and KMC3. Note that even though the total times of Pro$k$rustean graphs are longer than the mature tool, collecting k-reflecta (bolded time(compute)) is negligible that the information amount to be processed is very low. The two commands used in time columns of KMC are ./kmc and ./kmc_tools transform.

Algorithm 1 Collect Distinct k-reflecta

Input: Pro$k$rustean graph $G_{\mathcal{U}} = (V, E)$, $k \in \mathbb{N}$
Output: Distinct k-reflecta in $\mathcal{U}$

```
for v_S ∈ V do
    (V_S, E_S) = Pdecompose(v_S, k; G_U)
    for e ∈ E_S do
        if type(e) = 'reflecting' then (i, j) := interval(e)
            Report (v_S, i, j) # the string S[i..j] is a k-reflectum.
            If print requested then access str(S, i, j) and print k-mers in S[i..j].
        end if
    end for
end for
```

4.2 Count distinct k-mers for range $k = 1 \ldots N$

Normally, counting k-mers is equivalent to finding the string k-mers, imposing applications to multiply the burdens when multiple k's are queried. However, a Pro$k$rustean graph can be used to fulfill this task in $O(|\text{Pro}k\text{rustean}|)$ time, regardless of the range of k.

	dataset	time	memory	thread	time (compute)	time (load/save)	k
Prokrustean	Short reads(SRR)	**0.36s**	80mb	8	0.06s	0.3s	1...150
	Short reads(ERR)	**30s**	5.8gb	8	5s	25s	30...150
	Long reads (SRRL)	**88s**	24gb	8	13s	65s	30...50000
KMC	Short reads(SRR)	**5m**	837mb	8	5m		1...150
	Short reads(ERR)	**3613s**	12gb	8	3613s		30...150
	Long reads (SRRL)	**days**	-	-	-		30...50000

Table 3: **Comparison with a range of k:** Distinct kmers were counted for the given range of k. The time(compute) of Pro$k$rustean does not increase a lot from collecting k-mers of a fixed k. This is expected because both functions run a single time through a Pro$k$rustean graph. KMC was called iteratively for each k, so it cannot avoid the steady increase of total amount of time, with inability to complete on the long read data in reasonable time. The efficiency of Pro$k$rustean is due to the invariant allowing one scan on the graph to compute the whole information.

Recall that each k-mer in k-reflecta is distinct globally, and k-reflecta are idenfitied locally in each $S \in \text{StraStr}(\mathcal{U})$. So, through the progress of $k = 1 \to 2 \to \cdots \to |S|$, carefully observing k-reflecta reveals an invariant.

(1) If a k-reflecting region is the whole region of S, $k \to k+1$ decreases k-mers reflecting S by 1.
(2) If a k-reflecting region is in between stratifying regions, $k-1 \to k$ increases k-mers reflecting S by 1.

Theorem 5. *Fix $\mathcal{U}$ and let $k > 1$. An invariant holds for k-decomposed regions of S:*
for $C^k(S) :=$ the number of k-mers that reflect S,

$$C^k(S) = C^{k-1}(S) - \mathbf{1}_{Cond_Single_Relectum(S,k)} + \sum_{\substack{\forall(S,i,j) \\ stratifying}} \mathbf{1}_{Cond_Reflectum_Between_Stra(S,i,j,k)} \tag{6}$$

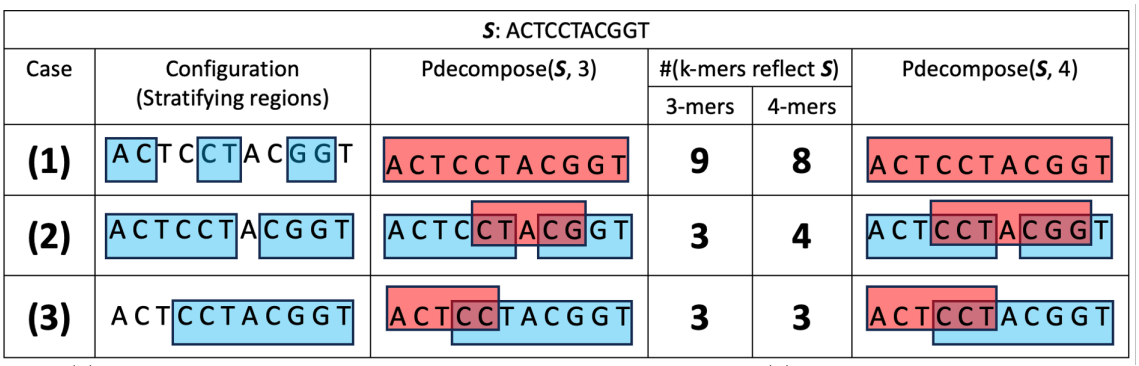

Fig. 7: Case(1) describes a k-reflectum occupying the whole string, and Case(2) describes a k-reflectum between two stratifying regions. Case(3) does not subject to any condition, meaning the number of k-reflecta and k-mers in them do not change as k increase.

where

1. $Cond_Single_Reflectum(S, k) :=$ *at k, a single k-reflecting region occupies the whole region* $(S, 1, |S|)$.
2. $Cond_Reflectum_Between_Stra(S, i, j, k) :=$ *at k, for a stratifying region (S, i, j), there exists a stratifying region at its right such that a k-reflecting region is defined between them.*

or equivalently,

1. $Cond_Single_Reflectum(S, k) = |$*largest stratifying region in* $S\ |< k \leq |S|$
2. $Cond_Reflectum_Between_Stra(S, i, j, k) = |$*intersection with the closest stratifying region on right of* $(S, i, j)\ | \leq k - 1\ \ \&\ \ k \leq min(|(S, i, j)|, |$*largest stratifying region on right of* $(S, i, j)\ |)$

Note that the conditions are expressed solely with stratifying regions. The conditions are used in indicator functions, which can be realized with arrays indexed by k's that represent each C^k and its changing rate. Then each interval in the condition contributes constant time to *Algorithm* 4. So, the running time is $O(|prokrustean|)$.

Sequence Cardinality δ

Delta (δ, [5]) is a metric suggested recently to measure the sequence growth and similarity. It is computed in $O(|\text{Pro}k\text{rustean}|)$ time as evident in this section. $\delta(S) := \max_k \left(\frac{d_k(S)}{k}\right)$, where $d_k(S)$ is the number of distinct k-mers in S.

4.3 Support universal assembly

By universal assembly, we denote the same computation power as a brute-force approach of extending of a given contig. In this section, we show universal assembly can be simulated above a Pro$k$rustean graph in $O(|\text{Pro}k\text{rustean}|)$. Then as practical proof of concepts, we introduce applications that extracts a compacted de Bruijn graph of any k (*section* 4.4), and counts maximal unitigs of de Bruijn graphs for a range of k (*section* 4.5). Lastly, the connections to variable-order de Bruijn graph of all k's and overlap-graphs are described (*section* 4.6).

Theorem 6. *(right extension) Let a string M reflects $S \in \text{StraStr}(\mathcal{U})$.*
If M is a suffix of S,

$$\forall c \in \Sigma,\ occ_{\mathcal{U}}(Mc) = occ_{\mathcal{U}}(Sc) \tag{7}$$

else,

$$\exists ! c \in \Sigma\ s.t.\ occ_{\mathcal{U}}(Mc) > 0 \tag{8}$$

Above theorem explains any mer which is a suffix of its reflected string in $\text{StraStr}(\mathcal{U})$ behaves the same as the string. Also, if the mer is in the middle of the reflected string, then no extension but just one scenario is possible. Then extending inside of a stratum or sequence, the next mer can 'fall' into a stratifying region and reflect another stratum. If the length of the mer is short, there is more possibility to fall, and otherwise the extension will more likely reach the end of the string. This matches the intuition of **Desiderata** discussed in section 2. Therefore, using this information, we define two components that represent 'flexible' extensions above a Pro$k$rustean graph.

Definition 8. *For a stratifying region (S, i, j), let $str(S, i, j)$ has a suffix R, which is a stratum, and $S[j+1]$ is the letter c. Then:*

$$ancestor_G(v_R, c) := e_{S,i,j}$$

Let a prefix R' of a string R is the largest stratum such that a k-reflectum is a prefix. Then:

$$descendant_G(v_R, k) := v_{R'}$$

$ancestor_G$ and $descendant_G$ show clear distinction between branching and extension. $ancestor_G(v_R, c)$ is called at the end of a stratum. It finds a larger string in which the extension c is allowed. $descendant_G$ is called at the front of a stratum. It finds a stratum with a k-reflectum being a prefix. These two ingredients can be implemented in varying forms.

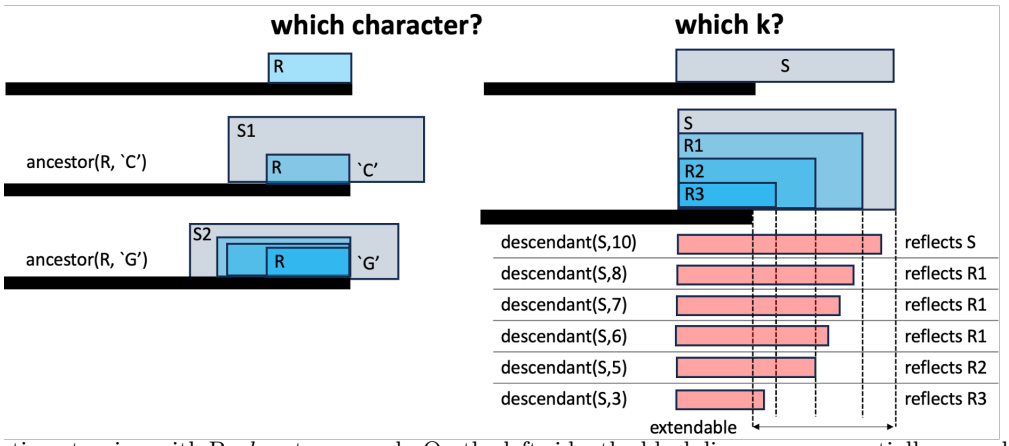

Fig. 8: A contig extension with Pro$k$rustean graph. On the left side, the black line means a partially assembled contig. An extension always ends with a sequence or a stratum. $ancestor_G(R, c)$ returns an extendable stratifying region with $c \in \Sigma$ (or $\varnothing$ if not found), represented in grey box. At the right side of the figure, $descendant_G(S, k)$ indicates how much the contig will be extended. Since k is fixed in a de Bruijn graph of k-mers, the choice among the red reflecting boxes (descendant) is predetermined. $ancestor_G(R, c)$ and $descendant_G(S, k)$ are used in turn to extend a contig until they meet a dead end with each $ancestor_G(R, c)$.

Algorithm 2 Universal right extension

Input: Pro$k$rustean graph $G_{\mathcal{U}} = (V, E)$, an l-reflecting edge $e_{S,i,j}$ from $Pdecompose(v_S, l)$
Output: k^*-reflecting edges whose strings overlap with $str(S, i, j)$ by length $k^* - 1$.

Set $focusedEdges, nextEdges = \{\}$
if $j = size(v_S)$ **then** # last edge of decomposed regions of S
  for $c \in \Sigma$, **if** $ancestor_G(v_S, c) \neq \varnothing$ **then** $focusedEdges.add(ancestor_G(v_S, c))$
else
  $focusedEdges.add(e_{S,i,j})$
end if
for $e_{S,p,q} \in focusedEdges$ **do**
  $k^* = Strategy(e_{S,p,q}; G_{\mathcal{U}})$
  $e_{S,p',q'}$ = next of $e_{S,p,q}$ in $Pdecompose(v_S, k^*)$
  if $type(e_{S,p',q'})$ is 'stratifying' **then**
    $v_{descendant} = descendant_G(v_{str(S,p',q')}, k^*)$
    $nextEdges.add$(first of $Pdecompose(v_{descendant}, k^*)$)
  else
    $nextEdges.add(e_{S,p',q'})$
  end if
end for
Return $nextEdges$

Algorithm 2 abstracts how two ingredients described in fig. 8 are used to obtain all possible extensions of a given l-reflectum. Consider a contig that is being assembled. For a short suffix S of the contig, we always

have some vertex $v_S \in V$ of the Pro*k*rustean graph. Then suppose we want to overlap a suffix M of S at the next extension. If M is the suffix of S, by theorem 6, $ancestor_G$ suggests the options. Otherwise, only one extension is possible. $Strategy(e_{S,i,j})$ abstracts the decision of $descendant_G$ that how much the contig is extended. There will be varying implementations of *Strategy* in the scheme of variable-order de Bruijn graphs.

4.4 Extract a compacted de Bruijn graph of *k*

	dataset	CPU-time	memory	thread	time (compute)	time (load/save)	k
Prokrustean	Short reads(SRR)	8s	342mb	1	4.8s	3.2s	20
	Short reads(ERR)	10m	20gb	1	8m	2m	30
	Long reads (SRRL)	51m	67gb	1	47m	4m	30
GGCAT	Short reads(SRR)	35s	283mb	1	35s		20
	Short reads(ERR)	9m	1.1gb	1	9m		30
	Long reads (SRRL)	9m	1.1gb	1	9m		30

Table 4: **Comparison with fixed *k*:** The Pro*k*rustean graph performs slightly slower in CPU-time than the purpose-built, optimized GGCAT in the task of building compacted de Bruijn graphs. The CPU-time was measured as (threads*total time) because the Pro*k*rustean approach is single threaded. Since Pro*k*rustean graph can simulate de Bruijn graphs directly on itself, this task was designed for showing proof of concept and the reasonable running time.

We show the connection of a Pro*k*rustean graph with the union of de Bruijn graphs of all k's. The performance is comparable to a state-of-art tool due to the compact representation of a Pro*k*rustean graph. But the Pro*k*rustean graph of long reads took much more time and space to generate a de Bruijn graph because the cardinality of the Pro*k*rustean graph is much larger than that of short reads. Indeed, we observed maximal unitigs of de Bruijn graphs built with long reads increase as k increases from 30. This implies that the de Bruijn graph of $k = 30$ loses a more significant amount of information than in short reads.

The de Bruijn graph of a fixed k is extracted by setting *Strategy* in algorithm 2 to always return $k - 1$. That is, algorithm 2 will always return $(k - 1)$-reflecting edges, and a contig can be assembled by running the algorithm iteratively and obtain a series of edges. Here, a small complication is revealed that de Bruijn graphs of degree k is about k-mers and $k - 1$ similarities. So, the actual implementation constructs k-reflecta in the context of $(k - 1)$ similarities.

Note that a k-reflectum is, in the context of assembly, a new type of a unitig that includes a same colored k-mers(Lemma 3). A unitig is a simple path in a de Bruijn graph, and colors of a k-mer denote reads in which the k-mer occurs, and sometimes include the frequencies of the k-mer in each read. Therefore, if a k-reflectum is split into k-mers, then de Bruijn graphs are generated, and if k-reflecta comprising a simple path are merged maximally, then compacted de Bruijn graphs are generated.

4.5 Count maximal unitigs of de Bruijn graphs of *k*-mers for range $k = 1...N$

	dataset	time	memory	thread	time (compute)	time (load/save)	k
Prokrustean	Short reads(SRR)	**0.36s**	70mb	8	0.05s	0.31s	20..150
	Short reads(ERR)	**31s**	5.8gb	8	5s	26s	30..150
	Long reads (SRRL)	**127s**	26gb	8	18s	109s	30..50000
GGCAT	Short reads(SRR)	**757s**	310mb	8	757s		20..150
	Short reads(ERR)	**5,917s**	1.1gb	8	5917s		30..150
	Long reads (SRRL)	**days**	-	-	-		30..50000

Table 5: **Comparison with a range of *k*:** On the task of counting the number of maximal unitigs in a de Bruijn graph, the Pro*k*rustean approach is orders of magnitude faster than GGCAT which was not designed for this task so needs to compute at each k separately. The efficiency shown in time(compute) of Pro*k*rustean is due to the structure of the algorithm that considers only the first/last decomposed edges of each vertex and that captures from and until when they represent divergence/convergence/tip. So, the performance is linear to the vertices. For example, the Pro*k*rustean graph of ERR has 221 million vertices, so a task scaning 221 million units of information for a simple integer lookup is worth only 5 seconds.

Maximal unitigs are key to understand the topology of a de Bruijn graph. They are computed in $O(|\text{Pro}k\text{rustean}|)$ time for arbitrary range of k. The intuition is that knowing k-reflecta are unitigs, it is enough to analyze the existence of a k-reflectum at each end of a stratum. For example, having a k-reflectum at back and having multiple $ancestor_{\mathcal{U}}(v_S, *)$ implies a divergence populated by stratum S at k. Then, changing k, analyzing decomposed regions at front and back of S is enough, which takes constant time per S, so the algorithm takes $O(|\text{StraStr}(\mathcal{U})|)$. We leave the algorithm in Supplementary *Algorithm* 7.

4.6 Simulate variable-order de Bruijn graph & overlap graph

Another potential contribution of a Pro$k$rustean graph in assembly is in advancing existing assemblers with the so-called variable-order scheme. Choosing a proper k in *Strategy* in Algorithm 2 allows a multi-k approach, extending the functionality of variable-order de Bruijn graphs in [6] and [4]. These previous implementations have trade-offs around space-efficiency, usable k range, and the utilization of k-mer frequencies. We claim Pro$k$rustean graph has access to frequencies, and is space-efficient, and uses all k's. Consider following implementation:

- Enhance Pro$k$rustean graph with a vertex label $frequency(v_S) := |occ_{\mathcal{U}}(S)|$ to help the decision in $Strategy(e_{S,i,j})$. The information is either freely accessible during the construction or directly computed from the graph.
- Utilizing the label $frequency$, choose the 'best' parameter k^* in $Strategy(e_{S,i,j})$ in Algorithm 2.

Above approach clearly implements a variable-order de Bruijn graph. The noteworthy part is it may use only $O(|\text{Pro}k\text{rustean}|)$ time and space. That is, the number of the returning values of *Strategy* can be limited, instead of all possible k's. Consider the right part of *Figure* 8. Choices of $k = 8, 7, 6$ have the same effect as they reflect the same string R1 and hence choosing the longest $k = 8$ is always desirable. Likewise, for each $e_{S,p,q}$ and the stratum $str(S, p, q)$, it is enough to consider strata sharing their prefix, at considering options in $Strategy(e_{S,p,q})$, so time complexity does not pump above $O(|\text{Pro}k\text{rustean}|)$. Also, $O(|\text{Pro}k\text{rustean}|)$ bytes are required for above implementation as we are working with a Pro$k$rustean graph only. All the labels of the graph takes constant amount of bytes per each vertex and edge.

Lastly, we provide an answer to the open problem 5 in [23] which comments that a variable-order de Bruijn graph "corresponds" to an overlap graph. As the word correspondence (not 'equivalence') implies, read overlaps are not automatically considered in variable-order approaches, but require extra effort. Overlaps can be built into the Pro$k$rustean construction by:

- Modify the vertex label $frequency(v_S)$ to allow prepending/appending characters: for $c_l, c_r \in \Sigma \cup \{\$\}$, $frequency(v_S, c_l, c_r) := |occ_{\mathcal{U}}(c_l S c_r)|$.

Again, the label can either be built during the construction or from the graph. With the extended frequency label, decisions of c or k of $ancestor_G$, $descendant_G$ and $Pdecompose$ can prioritize returning v_S such that S has thick overlapping occurrences. For example, $C := \Sigma_{c\in\Sigma} frequency(v_S, c, \$) = \Sigma_{c\in\Sigma}|occ_{\mathcal{U}}(cS\$)|$ means S represented by a vertex v_S has C reads that ends with S. Let some $ancestor_{\mathcal{U}}(v_S, c')$ has the source vertex $v_{S'}$, and $C' := \Sigma_{c\in\Sigma} frequency(v_{S'}, \$, c) = \Sigma_{c\in\Sigma}|occ_{\mathcal{U}}(\$Sc)|$. If C and C' are both large enough, we can believe the choice of c' for extending S is reliable in the context of an overlap graph.

5 Pro$k$rustean graph construction

A Pro$k$rustean graph can be built with two generalized suffix trees - one built with $\mathcal{U}$ and the other with reverse strings of $\mathcal{U}$ - or an affix tree of $\mathcal{U}$. We briefly introduce how stratification is connected to the suffix tree structure, which clearly reveals the straightforward way of obtaining a Pro$k$rustean graph. However, the straightforward method is less practical because building a suffix tree requires about $20N$ bytes. The consequential algorithms of Pro$k$rustean graph construction use ebwt that implicitly navigate the suffix tree structure of $\mathcal{U}$. We leave all details of the engineering in Supplementary section 4.

Let $\mathcal{U}^r$ be the set of reversed strings of $\mathcal{U}$. Let $T_{\mathcal{U}}$ and $T_{\mathcal{U}^r}$ be each the generalized suffix tree built with $\mathcal{U}$ and $\mathcal{U}^r$, respectively. Also, let $node(S) \in T_{\mathcal{U}}$ the node representing S and $node^r(S^r) \in T^r_{\mathcal{U}}$ and the node representing S^r. Below fact is well-known in the context of an affix tree.

Fact. If both $node(S)$ and $node^r(S^r)$ exist, then R is a maximal repeat.

Now, let $(node(S), node(S'))$ be an edge in $T_{\mathcal{U}}$. Let an edge $e_{S,i,j}$ be in a Pro*k*rustean graph of $\mathcal{U}$. Then three stratification scenarios can be described by $T_{\mathcal{U}}$ and $T^r_{\mathcal{U}}$.

1. If $i = 1$, then $node(str(S, i, j))$ in $T_{\mathcal{U}}$ has an edge representing $S[j + 1..|S|]$.
2. If $j = |S|$, then $node^r(str^r(S, i, j))$ in $T^r_{\mathcal{U}}$ has an edge representing the reverse of $S[1..i - 1]$.
3. If $i \neq 1$ & $j \neq |S|$, then both scenarios above hold.

This is an intuitive observation describing information-wise inclusion between the suffix tress and a Pro*k*rustean graph. That is, every stratifying relationship is depicted on edge(s) of the suffix trees. For example, the scenario 1 can be briefly proved by the definition of stratification. Since (S, i, j) can be extended on right and hence its string is a maximal repeat(stratum), there exists a $node(str(S, i, j))$ having an edge branching with $S[j + 1]$. Now, the terminal node is $node(S)$ because otherwise, say $node(S')$ is the terminal node, S' is a substring of S. It contradicts to the fact that (S, i, j) is stratifying because there must be an extension (S, i', j') of (S, i, j) whose string is S', meaning (S, i, j) is relatively stratifying in (S, i', j').

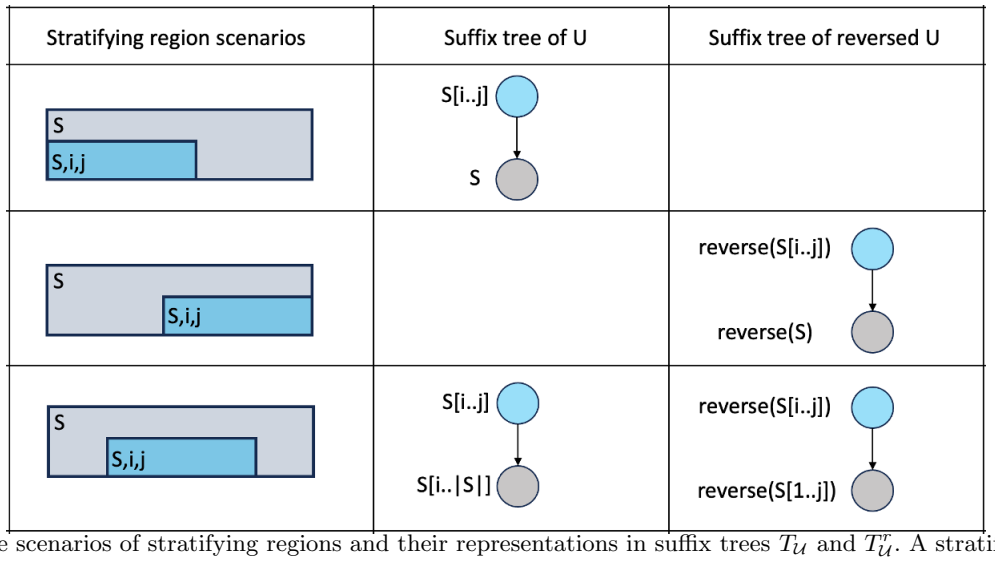

Fig. 9: Three scenarios of stratifying regions and their representations in suffix trees $T_{\mathcal{U}}$ and $T^r_{\mathcal{U}}$. A stratifying region (S, i, j) can be placed at each end of S or in the middle of S. Since the existence of a node in $T_{\mathcal{U}}$ implies right-maximality and the existence of a node in $T^r_{\mathcal{U}}$ implies left-maximality, starting from the string of (S, i, j), the left over substring of $S[j + 1..|S|]$ or $S[1..i - 1]$ is represented by some edge in $T_{\mathcal{U}}$ or $T^r_{\mathcal{U}}$.

However, the reverse of the observations do not hold, meaning an additional condition is required to decide which edges of the suffix trees imply stratifying relationships. For example, even if the first scenario in fig. 9 holds that $(node(S), node(str(S, i, j)))$ is an edge in $T_{\mathcal{U}}$, an edge of $e_{S,1,*}$ may not exist in the Pro*k*rustean graph. The further development is introduced in Supplementary theorem 9.

Discussion

To our knowledge, this is the first approach that clearly captures how changing k-mer size affects sets of k-mers. For each sequence or maximal repeat, a set of k-mers uniquely co-occur with it. When k changes to k', the composition of co-occurring mers may or may not change. If the composition does not change, each co-occurring k-mer can be mapped to a co-occurring k'-mer of the string and they act exactly same in k-mer-based methods. Therefore, to explain the difference in the behavior of any application utilizing k-mers and $k + 1$-mers, vertices of maximal repeats of length k in the Pro*k*rustean graph and their incident edges are necessary and sufficient information to describe the behavior of such methods. We posit that the concept of stratification may provide an entry-point by which practitioners can better understand the structure and similarities in genomic sequences.

This efficient Pro*k*rustean data structure marks an advancement in k-mer-based bioinformatic analyses. Unlike traditional methods that often require a trade-off between computational time and the breadth of k-mer sizes analyzed, the Pro*k*rustean graph approach is shown to extract sequence information across arbitrary k-mer size ranges with time complexity largely independent of k-mer size. However, the Pro*k*rustean approach

is not merely computational, but also conceptual, providing a new lens through which we can view k-mer-based sequence analysis. We intend this work to allow bioinformaticians efficient ways to answer questions such as: "which k-mer size would maximize the N5G0 of this assembly?", "what biological insight could be determined from considering all k-mer-sizes simultaneously for measures such as the Jaccard index, cosine similarity, or average nucleotide identity?", and "how might variable-order de Bruijn graphs represent overlap graphs to improve assembly accuracy and efficiency?", etc.

With regard to practicalities, the construction algorithm has room for improvement. Both the storage and construction are affected by the integer representations of sizes and locations of regions. The constant factor of the $O(N)$ performance can be large depending on the cardinality of the output size, which can vary depending on the characteristics of the sequences analyzed. While the eBWT and Wavelet tree data structures are used to further optimize the Pro$k$rustean construct, both of these areas are under active development and further advancements can improve the efficiency of the Pro$k$rustean graph construction.

Bibliography

[1] G. Badkobeh, M. Crochemore, and C. Toopsuwan. Computing the maximal-exponent repeats of an overlap-free string in linear time. In *String Processing and Information Retrieval: 19th International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings 19*, pages 61–72. Springer, 2012.

[2] P. Barenbaum, V. Becher, A. Deymonnaz, M. Halsband, and P. A. Heiber. Efficient repeat finding in sets of strings via suffix arrays. *Discrete Mathematics & Theoretical Computer Science*, 15(Discrete Algorithms), 2013.

[3] 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.

[4] 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.

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

[6] 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.

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

[8] S. P. Chockalingam, S. V. Thankachan, and S. Aluru. A parallel algorithm for finding all pairs κ-mismatch maximal common substrings. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 784–794. IEEE, 2016.

[9] A. Colosimo and A. De Luca. Special factors in biological strings. *Journal of theoretical biology*, 204(1):29–46, 2000.

[10] 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.

[11] 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.

[12] J. Fischer, F. Kurpicz, and M. Löbel. Simple, fast and lightweight parallel wavelet tree construction. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 9–20. SIAM, 2018.

[13] M. Gallé and M. Tealdi. xkcd-repeats: A new taxonomy of repeats defined by their context diversity. *Journal of Discrete Algorithms*, 48:1–16, 2018.

[14] 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.

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

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

[17] D. Koslicki and D. Falush. Metapalette: a k-mer painting approach for metagenomic taxonomic profiling and quantification of novel strain variation. *MSystems*, 1(3):e00020–16, 2016.

[18] S. C. Manekar and S. R. Sathe. A benchmark study of k-mer counting methods for high-throughput sequencing. *GigaScience*, 7(12):giy125, 2018.

[19] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the burrows–wheeler transform. *Theoretical Computer Science*, 387(3):298–312, 2007.

[20] G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[21] N. Pornputtapong, D. A. Acheampong, P. Patumcharoenpol, P. Jenjaroenpun, T. Wongsurawat, S.-R. Jun, S. Yongkiettrakul, N. Chokesajjawatee, and I. Nookaew. Kitsune: A tool for identifying empirically optimal k-mer length for alignment-free phylogenomic analysis. *Frontiers in Bioengineering and Biotechnology*, 8:556413, 2020.

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

[23] 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.

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

[25] J. Shaw and Y. W. Yu. Theory of local k-mer selection with applications to long-read alignment. *Bioinformatics*, 38(20):4659–4669, 2022.

1 Supplemental Material

2 Proofs of theorems

We first prove a simple lemma that is useful to make further discussions concise. If a region is more abundant than its 'parent' string, than it is stratifying or some extension of it is stratifying.

Lemma S5. *For S a string such that $occ_{\mathcal{U}}(S) > 0$, let the string of a region (S, i, j) be more abundant than S. Then (S, i, j) is stratifying in $\mathcal{U}$ or has an extension that is stratifying in $\mathcal{U}$.*

Proof. First, since $|occ_{\mathcal{U}}(str(S, i, j))| > |occ_{\mathcal{U}}(S)|$, by the definition of a stratifying region, (S, i, j) may be stratifying if the additional condition holds: $\forall(S, i', j') \supset (S, i, j)$, $occ_{\mathcal{U}}(str(S, i', j')) = occ_{\mathcal{U}}(S)$. Otherwise, some other extension of (S, i, j) must be stratifying as following. Consider extensions $(S, i-\delta, j+\gamma)$, $0 < \delta+\gamma$. Since the maximum extension is the whole region of S, $occ_{\mathcal{U}}(str(S, i - \delta, j + \gamma)) = occ_{\mathcal{U}}(S)$ eventually holds with some δ and γ. Then there must be some choice of $\delta = \delta^*$ and $\gamma = \gamma^*$ such that increasing each either makes its string occur less or is not valid, i.e. if $i-\delta^* \neq 1$, $|occ_{\mathcal{U}}(S, i-\delta^*, j+\gamma^*)| > |occ_{\mathcal{U}}(S, i-\delta^*-1, j+\gamma^*)| = |occ_{\mathcal{U}}(S)|$ or if $j + \gamma^* \neq |S|$, $|occ_{\mathcal{U}}(S, i - \delta^*, j + \gamma^*)| > |occ_{\mathcal{U}}(S, i - \delta^*, j + \gamma^* + 1)| = |occ_{\mathcal{U}}(S)|$. Therefore, $occ_{\mathcal{U}}(S, i - \delta^*, j + \gamma^*)$ is the stratifying extension of (S, i, j).

□

Theorem 1 (Complete1). *A string R is a maximal repeat if and only if R is a stratum.*

Proof. This theorem is dependent on the next theorem (Theorem 2).

The forward direction: It is immediate from Theorem 2 that any occurrence (S, i, j) of R satisfies: (1) it is of the whole string that is abundant or (2) it is stratifying or (3) relatively stratifying in some other maximal repeats. (2) is already the condition to be found. For (3) case, the forward proof of Theorem 2 describes there exists some extension (S, i_n, j_n) of a string R_n that includes (S, i, j) as relatively stratifying. Also, R_n is a stratum as described in the proof, so R is a stratum. Lastly for (1) case, its string R must occur at some other place in $\mathcal{U}$ and it must be a proper substring of it, otherwise the string itself is the R. Therefore, the such other occurrence is stratifying or at least relatively stratifying in another stratum and hence R is a stratum, by the logic above.

The backward direction: Any stratum is trivially a maximal repeat, because the definition of stratifying region precisely describes the condition of maximal repeats (and more). □

Theorem 2 (Complete2). *A region (S, i, j) $(S \in \mathcal{U})$ is an occurrence of a maximal repeat if and only if*

1. *(S, i, j) is the whole region of S and $occ_{\mathcal{U}}(S) > 1$ or*
2. *(S, i, j) is stratifying or*
3. *There exists a stratum R such that (S, i, j) is stratifying relatively in R. That is, there exists an extension $(S, i', j') \in \mathrm{occ}_{\mathcal{U}}(R)$ such that $(R, i - i', |R| - (j' - j))$ is stratifying.*

Proof. The forward direction: The first condition is just a possible scenario of an occurrence of a maximal repeat. Other than the first condition, (S, i, j) is more abundant than its extension because its string is a proper substring of S. Then Lemma S5 tells that (S, i, j) is stratifying or an extension (S, i_1, j_1) is stratifying. Since the former case satisfies 1., let (S, i, j) not stratifying and (S, i_1, j_1) is stratifying. Then some relative location of (S, i, j) can be defined: let $R_1 := str(S, i_1, j_1)$, which is obviously a stratum by its construction, then $(R_1, i - i_1 + 1, |R_1| - (j_1 - j))$ is the relative position of (S, i, j) in (S, i_1, j_1) that has the same string as (S, i, j). The string of $(R_1, i - i_1 + 1, |R_1| - (j_1 - j))$ must be more abundant than R_1 because the string of (S, i, j) is a maximal repeat and $(R_1, i - i_1 + 1, |R_1| - (j_1 - j))$ cannot be the whole $(R_1, 1, |R_1|)$ considering the construction.

Now we are at the starting point a recursion that Lemma S5 tells that $(R_1, i - i_1 + 1, |R_1| - (j_1 - j))$ is stratifying or an extension is stratifying. If $(R_1, i - i_1 + 1, |R_1| - (j_1 - j))$ is not stratifying the same process above will resulting in finding some R_2 and a relative region of (S, i, j) in it. The process ends at some R_n where the relative region of (S, i, j) being stratifying in it, because $|R_1| > ... > |R_n|$ by the construction and there cannot be an extension of the relative position of (S, i, j) forever. Therefore, picking the R_n and the relative position of (S, i, j) in R_n, the condition 2 satisfies. **Note that what the process does is to recursively capturing regions of strata consequently.

The backward direction: If (S, i, j) is stratifying, trivially its string is a stratum by definition.

Otherwise, let (S, i, j) be relatively stratifying in an occurrence of a stratum (S, i', j'). Then (S, i, j) is more abundant than S. We show left extension of one unit will make the occurrence decrease, and the right case is symmetric. The left extension $(S, i-1, j)$ is either in (S, i', j') or not, where the latter case happen when $i = i'$.

If $(S, i-1, j) \subset (S, i', j')$, then obviously the relatively stratifying region (S, i, j) in (S, i', j') satisfies $occ_{\mathcal{U}}(S, i-1, j) < occ_{\mathcal{U}}(S, i, j)$. Even if $(S, i-1, j) \not\subset (S, i', j')$, then $i = i'$ that the left extension goes out of the region (S, i', j'). Such case also decreases the occurrence. Let $R := (S, i', j')$ be the stratum relatively including (S, i, j). Then a string of shape cR is less abundant than R for any c. This means for any prefix P of R, cP is less abundant than P. Therefore, the string of (S, i, j), which is also a prefix of R is more abundant than any left extension, and hence the region is an occurrence of a maximal repeat.

Therefore, applying symmetrically to the right case, extending (S, i, j) either left or right makes the occurrence decreased and hence $str(S, i, j)$ is a maximal repeat.

□

Theorem 3 (Compact). *For $\mathcal{U}$ a set of sequences of total length N on the alphabet Σ,*

$$|\mathcal{S}_{\mathcal{U}}| < N \text{ and } |\mathrm{StraRgn}(\mathcal{U})| < 2|\Sigma|N.$$

Proof. From Theorem 1, we have that strata correspond to maximal repeats, and so the first inequality follows from the well-known fact that the number of maximal repeats (i.e. strata) is less than N ([15]). The second inequality will follow from our showing that for each letter in the alphabet, at most one stratifying region can have the letter at its immediate left among all stratifying regions of a stratum. That is, $\forall c \in \Sigma$, $\forall R \in \mathcal{S}_{\mathcal{U}}$, $1 \geq |\{(S, i, j) | str(S, i, j) = R \ \& \ (S, i, j) \in \mathrm{StraRgn}(\mathcal{U}) \ \& \ i > 1 \ \& \ S[i-1] = c\}|$. Indeed, if this holds, the case of each letter at the immediate right of a region holds symmetrically, implying that $|\mathrm{StraRgn}(\mathcal{U})| \leq 2|\Sigma||\mathcal{S}_{\mathcal{U}}| < 2|\Sigma|N$ as desired.

To show this, we argue by contradiction. For a given $c \in \Sigma$, let $(S, i, j) \in \mathrm{StraRgn}(\mathcal{U})$ has c at its immediate left: $S[i-1] = c$. Assume for contradiction that there exists another stratifying region $(S', i', j') \in \mathrm{StraRgn}(\mathcal{U})$ sharing the same string and has the same letter at its left: that is, $str(S, i, j) = str(S', i', j')$ and $S[i-1] = S'[i'-1] = c$. Consider extending two regions together that $(S, i-\delta_l, j+\delta_r)$ and $(S', i'-\delta_l, j'+\delta_r)$ where $\delta_l, \delta_r > 0$. And let δ_l^* and δ_r^* be the largest as possible while preserving string equality: $str(S, i-\delta_l^*, j+\delta_r^*) = str(S', i'-\delta_l^*, j'+\delta_r^*)$. Note that two base regions share c at their immediate left, so $\delta_l^* \geq 1$ is guaranteed and $(S, i, j) \subsetneq (S, i-\delta_l^*, j+\delta_r^*)$. Now, by definition, increasing δ_r^* or δ_l^* will cause the two regions to have different strings or makes at least one region invalid as the end of the string was reached. However, if both extended regions reach both ends of their string, this implies that $(S, i-\delta_l^*, j+\delta_r^*) = (S', i'-\delta_l^*, j'+\delta_r^*) = (S, 1, |S|) = (S', 1, |S'|)$ that S and S' were the same string and hence $(S, i, j) = (S', i', j')$ which is a contradiction. In the other case, at least one region can extend to left or right and get an extended string but the other cannot extend that direction or at least corresponds to a different string. Without loss of generality, say that $(S, i-\delta_l^*, j+\delta_r^*)$ can be extended left. This means that $|occ_{\mathcal{U}}(str(S, i-\delta_l^*-1, j+\delta_r^*))| < |occ_{\mathcal{U}}(str(S, i-\delta_l^*, j+\delta_r^*))|$ and hence $|occ_{\mathcal{U}}(S)| < |occ_{\mathcal{U}}(str(S, i-\delta_l^*, j+\delta_r^*))|$, and since $(S, i, j) \subsetneq (S, i-\delta_l^*, j+\delta_r^*)$, extending (S, i, j) does not always decrease the occurrence number of corresponding strings, (S, i, j) is not stratifying, which is contradiction.

Theorem 4 (Universal reflection). *For any string S such that $occ_{\mathcal{U}}(S) > 0$,*

$$S \text{ reflects exactly one string in } \mathrm{StraStr}(\mathcal{U}).$$

Proof. First, we show S occurring in $\mathcal{U}$ reflects at least one string in $\mathrm{StraStr}(\mathcal{U})$. Recall the definition of "S reflects S'" is "S is a substring of S' and occurs the same as S'". Truely S is a substring of some string in $\mathrm{StraStr}(\mathcal{U})$ because it occurs in some sequence in $\mathcal{U}$. Next, check if a superstring of S occurs the same as S.

Assume for contradiction for every superstring of S in $\mathrm{StraStr}(\mathcal{U})$, S occurs more than each of them. Then consider an occurrence of S in $S^1 \in \mathrm{StraStr}(\mathcal{U})$. Since it is assumed to be more abundant than S^1, the occurrence is either stratifying or can be extended to be a stratifying region (*Lemma S*5). If it is stratifying, then it is contradiction because S is a stratum and it occurs the same as itself ($\in \mathrm{StraStr}(\mathcal{U})$). Therefore, the occurrence must be in another stratifying region of a string S^2 and S is a substring of S^2 by its construction. Consider the occurrence of S in S^2. By the assumption, again, S is more abundant than S^2

too, so an occurrence of S is either stratifying or extendable to be a stratifying region in S^2. This process stops because the sizes of captured strata $S^1, S^2, \ldots, S^n$ decrease by the definition of stratifying regions. But the termination is contradictory because S is assumed to be abundant than any superstring in $\mathrm{StraStr}(\mathcal{U})$ meaning always a stratifying region including its occurrence can be found inside of the last stratum S^n.

Second, we show S reflects exactly one string in $\mathrm{StraStr}(\mathcal{U})$. Assume for contradiction S reflects two strings $S', S'' \in \mathrm{StraStr}(\mathcal{U})$. Since S is a substring of S' and S'', any occurrence of S' or S'' includes an occurrence of S. Conversely, for any occurrence of S, S' and S'' should have occurrences including the region, or otherwise S occurs more than S' or S''. That means for any occurrence of S, it is in an overlapped region of two occurrences of S' and S''. Any occurrence of S being an overlapped region of S' and S'', while those three co-occur, is contradiction because S is a stratum that is defined in the overlapped region. That is, extending the overlapped region on both sides, which are each end of S' or S'', makes the number of occurrence surely decreased. That means S is more abundant than either S' or S'', which is contradiction.

Therefore, S always reflects a string in $\mathrm{StraStr}(\mathcal{U})$ and such string is unique.

□

2.1 *Pdecompose*

The algorithm below gets decomposed regions. We briefly discuss the correctness to the connected concepts.

Algorithm 3 correctly follows the definition of decomposed regions. Only stratifying regions of length k are involved (line 2..6), and whenever a length k region is not covered, a region that covers it. Assume a length k region at the front or end of S is not covered. Then line 12 or line 27 catches the case and then generates a region to cover it. If a length k region is not covered between stratifying regions, line 15 or 18 catches it. If there is no stratifying region, then a region cover all(line 7). Lastly, any a region interval ($interval^r$) is always intersecting with stratifying region by length k.

Then this algorithm finds all k-reflecting regions. First, any length k region is now covered either by stratifying regions or the generated regions as described above. Second, no two regions cover the same length k region unless both are stratifying regions. If it is the case, there has to be an intersection of length at least k between regions but Algorithm 3 does not allow the case generating k-reflecting regions. Therefore, the generated region satisfies each length k region (S, i, j) to not included in a stratifying region meaning $occ_{\mathcal{U}}(S, i, j) = occ_{\mathcal{U}}(S)$. Also, since the regions are intersecting with stratifying regions by exactly $k-1$, or otherwise one region spans to the whole region, extending the region results in another length k region (S, i', j') be included in a stratifying region such that $occ_{\mathcal{U}}(S, i', j') > occ_{\mathcal{U}}(S)$.

The algorithm runs in $O(|\text{stratifying regions in } S|)$ time. Two loops in line 2 and line 11 are bounded by the number of outgoing edges of v_S. And all operations in the loop are constant.

Lemma 1 (Mer Cover). *Any length k region in S is included in a k-decomposed region of S.*

Proof. The construction of *Pdecompose* naturally satisfies the lemma. Assume a length k region (S, i, j) is not included in any k-decomposed region of S. Check both sides of the region and find a stratifying region or k-reflecting region that intersects by $k-1$ if both are not found, then expand until such region is found. The region cannot be the whole region $(S, 1, |S|)$ by the construction rule of k-decomposed region. If eventually the region is intersecting with a stratifying region, it is contradiction because a k-reflectum must be defined to cover the $k-1$ intersection with a stratifying region. Otherwise, if a k-reflectum is found, it is also contradictory because the k-reflectum is not maximal that it was not extended even though the length k region intersecting with the k-reflectum is as abundant as S, not included in a stratifying region.

□

Lemma 2 (Computation). *Decomposing regions in S takes $O(|\{\text{stratifying regions in } S\}|)$ time.*

Proof. The decomposition algorithm Algorithm 3 is the evidence. Two loops are all governed by the stratifying regions (=the outgoing edges of a vertex of a Pro$k$rustean graph). The rest are all constant time assignments.

□

Lemma 3. *A k-reflectum and its substring k-mers together reflect the same string in* $\mathrm{StraStr}(\mathcal{U})$.

Proof. This lemma is immediate from the definition of k-reflectum. A k-reflectum defined from S is composed of k-mers that each is as abundant as S. Since every substring is the case, k-reflectum is also as abundant as S.

□

Lemma 4. *A k-mer K occurs in $\mathcal{U}$ if and only if K occurs uniquely in a k-reflectum in* $\mathrm{StraStr}(\mathcal{U})$.

Proof. This lemma is naturally derived from Theorem 4 and Lemma 3. Any k-mer occuring in $\mathcal{U}$, along with any string, reflects exactly one string S in $\mathrm{StraStr}(\mathcal{U})$. Two k-mers cannot occur either in the same k-reflectum or different k-reflecta defined from S, because the fact will make the k-mer more abundant than S.

□

Algorithm 3 *Pdecompose*

Input: Pro$k$rustean graph $G_{\mathcal{U}} = (V, E)$, $k \in \mathbb{N}$, $v_S \in V$
Output: A graph (V_S, E_S) representing decomposed regions of S.

Set $V_S = \{v_S\}$, $E_S = \varnothing$, $E_{stra} = \varnothing$
for $e = (v_S, *, *) \in E$ **do**
  if $|interval(e)| \geq k$ **then**
    E_{stra}.add(e)
  end if
end for
if E_{stra} is empty **then**
  $V_S.add(v_{new})$ s.t. $size(v_{new}) = |size(v_S)|$
  $E_S.add(e_{new})$ s.t. $e_{new} := (v_S, v_{new}, 1)$, $interval(e_{new}) = (1, size(v_S))$
end if
for $e = (v_S, v_{S'}, p) \in E_{stra}$ **do**
  if e is first in E_{stra} and $p > 1$ **then**
    # reflecting region includes 1st position.
    $interval^r := (1, interval(e).from + (k-1) - 1)$
  else if e':=prev in E_{stra}, $interval(e) \cap interval(e') = \varnothing$ **then**
    # reflecting region occurs in the gap of two stratifying region.
    $interval^r := (interval(e).to - (k-1) + 1, interval(e').from + (k-1) - 1)$
  else if e':=prev in E_{stra}, $|interval(e) \cap interval(e')| < k - 1$ **then**
    # reflecting region occurs between the intersection of two stratifying region.
    $interval^r := (interval(e').from - (k-1), interval(e).to + (k-1))$
  end if
  if $interval^r$ set **then**
    $V_S.add(v_{new})$ s.t. $size(v_{new}) = |interval^r|$
    $E_S.add(e_{new})$ s.t. $e_{new} := (v_S, v_{new}, interval^r.from)$, $interval(e_{new}) = interval^r$
  end if
  $V_S.add(v_{S'})$, $E_S.add(e)$ # existing stratifying region.
  if e is last in E_{stra} and $p < size(v_S)$ **then**
    # reflecting region includes the last position.
    $interval^r := (interval(e).to - (k-1) + 1, size(v_S))$
    $V_S.add(v_{new})$ s.t. $size(v_{new}) = |interval^r|$
    $E_S.add(e_{new})$ s.t. $e_{new} := (v_S, v_{new}, interval^r.from)$, $interval(e_{new}) = interval^r$
  end if
end for

3 Application Explanations

Four implementations are introduced: collecting distinct k-mers, counting k-mers for a range of k, building compacted de Bruijn graph of degree k, and counting maximal unitigs of degrees for a range of k. Each implementation was tested by independently built correspondence. For example, distinct k-mers were collected naively in a set and then compared with the result from Pro$k$rustean graph string by string.

3.1 Collect distinct k-mers

There is no more information to be added from the algorithm but the actual implementation requires some consideration about text access for printing. The constant access to a sequence can be safely assumed in general. There must be a identifier based fetching function. However, for getting the string of a stratum in constant time requires a map. One occurrence (S, i, j) per each stratum should be sampled. It can be either stored with a sequence or extracted at an application running. In this project, collecting distinct k-mers or building de Bruijn graphs use printing so requires the map. The memory occupation of the map is not trivial, so should be part of consideration for further development. In the experiment results in the main paper did the occurrence sampling on memory.

3.2 Count distinct k-mers of range $k = 1 \ldots N$

Recall the observation below intorduced in the main text. The figure below exemplifies why the observation is correct.

(1) If a k-reflectum is the whole S, $k \to k+1$ decreases k-mers representing S by 1.
(2) If a k-reflectum is placed between two stratifying regions, $k-1 \to k$ increases k-mers representing S by 1.

S: ACTCCTACGGT					
Case	Configuration (Stratifying regions)	Pdecompose(***S***, 3)	#(k-mers reflect ***S***)		Pdecompose(***S***, 4)
			3-mers	4-mers	
(1)	A C T C C T A C G G T	A C T C C T A C G G T	**9**	**8**	A C T C C T A C G G T
(2)	A C T C C T A C G G T	A C T C C T A C G G T	**3**	**4**	A C T C C T A C G G T
(3)	A C T C C T A C G G T	A C T C C T A C G G T	**3**	**3**	A C T C C T A C G G T

Fig. 10: Case(1) describes a k-reflectum occupying the whole string, and Case(2) describes a k-reflectum between two stratifying regions. Case(3) does not subject to any condition, meaning the number of k-reflecta and k-mers in them do not change as k increase.

Theorem 5. *Fix $\mathcal{U}$ and let $k > 1$. An invariant holds for k-decomposed regions of S:*
for $C^k(S) :=$ the number of k-mers that reflect S,

$$C^k(S) = C^{k-1}(S) - \mathbf{1}_{Cond_Single_Relectum(S,k)} + \sum_{\substack{\forall(S,i,j) \\ stratifying}} \mathbf{1}_{Cond_Reflectum_Between_Stra(S,i,j,k)} \tag{6}$$

where

1. *$Cond_Single_Reflectum(S,k) :=$ at k, a single k-reflecting region occupies the whole region $(S, 1, |S|)$.*
2. *$Cond_Reflectum_Between_Stra(S,i,j,k) :=$ at k, for a stratifying region (S,i,j), there exists a stratifying region at its right such that a k-reflecting region is defined between them.*

or equivalently,

1. *$Cond_Single_Reflectum(S,k) = |$largest stratifying region in S $| < k \leq |S|$*
2. *$Cond_Reflectum_Between_Stra(S,i,j,k) = |$intersection with the closest stratifying region on right of (S,i,j) $| \leq k-1$ & $k \leq min(|(S,i,j)|, |$largest stratifying region on right of (S,i,j) $|)$*

Proof. We first recall that for each k, it is enough to count the possible k-mers in each k-reflectum of each $S \in \text{StraStr}(\mathcal{U})$. Then in the transition of $k \to k+1$, only considering the scenarios in which a k-reflectum contributes to increasing or decreasing k-mers is enough. Therefore, above observations on how decomposed regions of degree k and $k+1$ build the invariant of the theorem.

First, if no stratifying region exists, the whole string S is a k-reflectum. Then unless $k+1 > |S|$, increasing k decreases the number of k-mers by exactly one. The condition $Cond_Single_Reflectum(S,k)$ clearly formulates it.

Second, if stratifying region exists, 'each existence of a paired two stratifying regions' is counted as a contribution. Three scenarios exist between k-reflectum and stratifying regions: a k-reflectum is either at the front/end of S and intersect with a stratifying region or defined between two stratifying regions. The former two cases do not affect the k-mer count at the transition $k \to k+1$ because anchoring on one edge of S, such k-reflectum increases exactly by one because the intersection with a stratifying region increases exactly one. However, the latter case increase the k-reflectum by one because the intersections on both sides with two stratifying regions increase the area by two, so one more $k+1$-mer is contributed at $k \to k+1$. That is,

for a stratifying region, if it is paired with some other stratifying region after $k \to k+1$, with a small or no intersection, then it contributes to increasing k-reflectum.

Note that the pair is not the only condition for 2. the intersection with the closest stratifying region on right has to be small enough or none. If the intersection is at least $k-2$, then any k-mer including the string in the intersection as a substring will be included in a stratifying region.

The first inequality in the condition $Cond_Reflectum_Between_Stra(S, i, j, k)$ denotes if two close stratifying region exists, then k has to be at least $k-1$ so that a k-reflectum occurs between them. The second inequality in the condition $Cond_Reflectum_Between_Stra(S, i, j, k)$ says any kind of such pair should exist. If the k is too large that it exceeds $\min(|(S, i, j)|, |\text{largest stratifying region at right}|)$, then at least for (S, i, j), there is no contribution to make.

□

The Algorithm 4 utilizes the invariant Theorem 5. First, for the lowest k value L, calculate the k-mer count. It takes $|\text{Pro}k\text{rustean}|$ time. Next, for each $k = L+1 \ldots N$, calculate the contribution to the increase of k-mers. The $\partial\partial C$ denotes "the contribution to the contribution of increasing count." For example, if $k-1$ was the size of the largest stratifying region and exceeds it at k, then from that moment the k-mers in the string decreases by 1 as k increases by 1. Therefore, instead of setting $\partial C[x]$ for all the range $k \leq x \leq |S|$, set $\partial\partial C[k] -= 1$ for the starting position and then 'cancel' it at $|S|+2$ as $\partial\partial C[|S|+2] += 1$. The reason of $|S|+2$ is that until $|S|+1$ the k-mer count will decrease. Line 7,8 explain it. The rest parts express the second condition in Theorem 5. Concrete details are in the codebase.

The algorithm runs in $O(|\text{Pro}k\text{rustean}| + N)$ time. Note that the loop at line 3 includes two nested loop which are all looping stratifying outgoing edges of v_S. Also, the other parts are constant time operations. So, the practical time complexity is $O(|\text{Pro}k\text{rustean}|)$. However, a theoretically worst case would be that a very long string is the only element of $\mathcal{U}$ and distinct k-mers are asked for the whole range. Then line 15 requires But again in practice, $|\text{Pro}k\text{rustean}|$ will normally outweigh the one time scan of $k = L+1 \ldots N$.

Algorithm 4 Count Distinct k-mers of a range

Input: Pro$k$rustean graph $G_{\mathcal{U}} = (V, E)$, $L \in \mathbb{N}$
Output: Count distinct k-mers for $k = L..N$, $N :=$ maximum sequence length

1: Set $C, \partial C, \partial\partial C :=$ each list of length N initialized with 0's.
2: $C[L] =$ Use *Algorithm* 1 for gathering L-reflecta and count L-mers in them.
3: **for** $v_S \in V$ **do**
4: **for** $e = (v_S, *, *) \in E$ **do**
5: Find $e_{max} :=$ the edge of the largest stratifying region
6: **end for**
7: $\partial\partial C[M] -= 1$, $M := |interval(e_{max})|$
8: $\partial\partial C[M] += 1$, $M := |S| + 2$
9: **for** reverse order of 'stratifying' $e \in E_S$ **do**
10: Set the $e_{rmax} := argmax(|interval(e_{rmax})|, |interval(e)|)$ the edge of the largest stratifying region at right.
11: $\partial\partial C[M] += 1$, $M := |interval(e_{rclose}) \cap interval(e)|$ or $L+1$, the smaller.
12: $\partial\partial C[M] -= 1$ $M := |interval(e_{rmax})| + 1$ or $|(S, i, j) + 1|$, the smaller.
13: Set $e_{rclose} := e$ # used as closest region at the next iteration.
14: **end for**
15: **for** $k = L+1..N$ **do**
16: $\partial C[k] = \partial C[k-1] + \partial\partial C[k]$
17: $C[k] = \partial C[k-1] + \partial C[k]$
18: **end for**
19: **end for**

3.3 Basics of universal assembly

Theorem 6. *(right extension) Let a string M reflects $S \in \mathrm{StraStr}(\mathcal{U})$.*

If M is a suffix of S,

$$\forall c \in \Sigma,\ occ_{\mathcal{U}}(Mc) = occ_{\mathcal{U}}(Sc) \tag{7}$$

else,

$$\exists! c \in \Sigma \text{ s.t. } occ_{\mathcal{U}}(Mc) > 0 \tag{8}$$

Proof. For eq. (7), since M reflects S, the $occ_{\mathcal{U}}(M) = occ_{\mathcal{U}}(S)$, so $occ_{\mathcal{U}}(Mc) \geq occ_{\mathcal{U}}(Sc)$ for any $c \in \Sigma \cup \{\$\}$. If $occ_{\mathcal{U}}(Mc) > occ_{\mathcal{U}}(Sc)$ for any c, then $\Sigma_{c \in \Sigma \cup \{\$\}} occ_{\mathcal{U}}(Mc) > \Sigma_{c \in \Sigma \cup \{\$\}} occ_{\mathcal{U}}(Sc)$ which is contradiction.

For eq. (8), since M is not a suffix, there must be at least one character c that can extend M. Such character cannot be more than one because if $occ_{\mathcal{U}}(Mc') > 0$ for some $c' \in \Sigma \cup \{\$\}$, then $occ_{\mathcal{U}}(M) > occ_{\mathcal{U}}(S)$.

3.4 Extracting compacted de Bruijn graphs of degree *k*

There are two considerations required before build de Bruijn graphs with a Pro*k*rustean graph.

Complication of $k-1$ similarity usage Given a parameter k, a de Bruijn graph defines vertices corresponding to k-mers and edges corresponding to prefix-suffix match of length $k-1$. That means de Bruijn graphs utilize similarities of length $k-1$. Therefore, Pro*k*rustean should use parameter $k-1$ instead of k to analyze or extract properties about de Bruijn graphs of degree k. This yields some consideration at stratums of size $k-1$ that behave a little peculiar. And it might be one of the reasons why the relationship between maximal repeats and compacted de Bruijn graphs has not been clarified against ample intuitions.

Pro*k*rustean graph edge direction For any graph, storing bidirectional edges are expensive. The natural direction of Pro*k*rustean graph is from a vertex v_S to edges where the edges represent the stratifying regions of S. Then the opposite direction that the occurrences of v_S are not stored. Then the $ancestor_{\mathcal{U}}(v_S, k)$ function introduced in *Section* 4 is difficult to design. In this implementation, we have tried using the Pro*k*rustean graph as is, so $ancestor_{\mathcal{U}}(v_S, k)$ is indirectly used. That is, starting from a k-reflectum, instead of navigating the straightforward right extensions, the k-reflectum is extended by both directions as much as possible following the Pro*k*rustean graph structure down to the very 'bottom'.

Definition 9. *$branchByAncestor_G(v_R, c) := e_{S,i,j} \in E$ such that $S[j+1] = c$, $str(S, j-|R|, j) = R$. That is, $e_{S,i,j}$ is an edge representing a stratifying region that is extendable with the letter c and a suffix of its string is R. If no such case exists, $branchByAncestor_G(v_R, c) = \varnothing$.*

Definition 10. *$extendByDescendant_G(e_{S,i,j}, k) := e_{R,i',j'} \in E_R$ such that $str(S, i-(k-1), j) = str(R, i, k-1)$. That is, $e_{R,i',j'}$ is an edge representing a k-reflecting region that its string shares length $k-1$ prefix-suffix with the string $str(S, i, j)$.*

As the names imply, $branchByAncestor_G$ and $extendByDescendant_G$ each guide branching and extension. The basic mechanism is to extend with k-reflecta that reflect strings in $\mathrm{StraStr}(\mathcal{U})$. Start from an edge $e_{S,i,j} \in E_S$ that is generated by *Pdecompose*. If K is not a suffix of S, then use $extendByDescendant_G(e_{S,i,j}, k) = e_{R,i',j'}$ to find next k-reflectum. $R \neq S$ will always be the case here that R is a descendant of S, meaning a path from S to R exists in G.

If K is a suffix of S, then $branchByAncestor_G(v_S, c) := e_{S',i,j}$ is used to find a stratifying region that connects K and c. Then again $extendByDescendant_G(e_{S',i,j}, k) = e_{R,i',j'}$ is used to determine the next k-reflectum. S' can either be same as R or not. The former means extension is done within S' and the latter means R is a descendant of S. This happens when because the next k or can be If $(S, j-(k-1), j+1)$ is stratifying, the extension is derived by $ascendByC_G(v_R, c) = e_{S,i,j}$. This means Kc occurs in $\mathcal{U}$ and reflects S. If K is not a suffix of R, then $descendByK_G(e_{S,i,j}, k)$ finds the next k-reflectum K' such that they share $k-1$ suffix and prefix.

Algorithm 5 Build de Bruijn graph of degree k

Input: The Pro$k$rustean graph $G_{\mathcal{U}}$, $k \in \mathbb{N}$

Output: The compacted de Bruijn graph of degree k built with $\mathcal{U}$.

Filter out vertices of V of size less than $k-1$
for $v_S \in V$ **do**
    If $left_cnt(v_S)$ implies convergence or tip, make a 'front' $Unitig$ of v_S
    If $right_cnt(v_S)$ implies divergence, make a 'back' $Unitig$ of v_S
end for
Initialize $visited(v_S) = false$ for all $v_S \in V$
for $v_S \in V$ **do**
    Skip if $visited(v_S)$, otherwise $visited(v_S) = true$
    Call a subprocedure to fill $setDescendants$ as below
  – If a first edge, which is stratifying, is less than If $left_cnt(v_S)$ implies convergence or tip, make a 'front' $Unitig$ of v_S

    If $right_cnt(v_S)$ implies divergence, make a 'back' $Unitig$ of v_S
end for
For each $v_S \in V$
Preprocess $branchByAncestor_G$, $extendByDescendant_G$.
Set $Next, Prev, Paths$ # Maps of edge to edges.
for $v_S \in V$ with $size(v_S) \geq k$ **do**
    Run $Pdecompose(v_S, k-1, G_{\mathcal{U}}) = (V_S, E_S)$
    for $e \in E_S$, $type(e) =$'reflecting' **do**
        if e not last in E_S **then** # not a suffix k-reflectum
            $Next[e].add(extendByDescendant_G(e,k))$
            $Prev[extendByDescendant_G(e,k)] = e$
        else# a suffix k-reflectum
            for $c \in \Sigma$ such that $branchByAncestor_G(v_S, c) = e'$ found **do**
                $Next[e].add(extendByDescendant_G(e',k))$
                $Prev[extendByDescendant_G(e',k)] = e$
            end for
        end if
    end for
end for
for $e_1 \in Next$ such that $Prev[e_1]$ is not single **do**
    Identify a path $P := e_1, e_2, ..., e_n$ such that $Next(e_i) = e_{i+1}$, $Next[e_i]$ is single, $Prev[e_{i+1}]$ is single for $i = 1, .., n-1$.
    $Paths[e_1] = P$
end for
Fill $CDBG$ a path to paths. For $e \in Paths$, $CDBG[Paths[e]].add(Paths[e'])$ for $e' \in Next[e_{last}]$ where e_{last} is the last edge of $Path[e]$.

3.5 Count maximal unitigs of de Bruijn graphs of a k range

As in the case of counting distinct k-mers, this task requires deeper understanding of Pro$k$rustean graph in order to consider the transition of k with respect to the specific application: counting maximal unitigs. The intuitive example, Algorithm 6, counts maximal unitigs of a fixed k and helps closing the gap of familiarity. The basic insight is that "each string (k-reflectum or k-mer) reflecting a stratum or a sequence is explained by the reflected string." And then in order to analyze the transition of $k \to k+1$, the insight is flipped: "whether each stratum or sequence is 'active' in a specific k is determined by k-reflectum or k-mer reflecting it." For example, the divergence in a de Bruijn graph can be explained by the stratum (or sequence) reflected by a k-mer or k-reflectum that is positioned at the last of the stratum. Flipping the notion, if a stratum has no k-reflectum at its last position, then it has no effect at k. Instead, there must be a stratifying region at the last position and probably the stratum of the stratifying region would have some k-reflectum at its last, and so on.

Once we are used to flipping the concept, then the algorithm can be easily designed. For counting maximal unitigs, the strategy is to count 3 components in the graph. (1) the start of a tip detected from a stratum or sequence vertex, (2) the start after a convergence detected from a stratum vertex, and (3) the start after a divergence detected from a stratum vertex. The sequence vertex is not subject to (2) or (3) because convergence and divergence implies some similarity event represented by strata.

We define $PdecomposeFirst(v_S, k; G)$ and $PdecomposeLast(v_S, k; G)$ meaning a constant time operation to make the first and last edge of decompoased regions of S of degree k. This is used to check the divergence and convergence topology of the graph efficiently. Also, as already defined, $left_cnt(v)$ and $right_cnt(v)$ are assumed to be supported, which are freely obtainable from Pro$k$rustean graph and require 1 byte per stratum to store, for small alphabet of sequences.

Algorithm 7 describes all cases of maximal unitig start. As the case in Algorithm 4, the variable ∂C describing the change minimize the required assignment. The corresponding implementation is in the codebase, and was tested with independently implemented de Buijn graphs. Sufficiently large samples ($N > 10M$) was used to check the correctness.

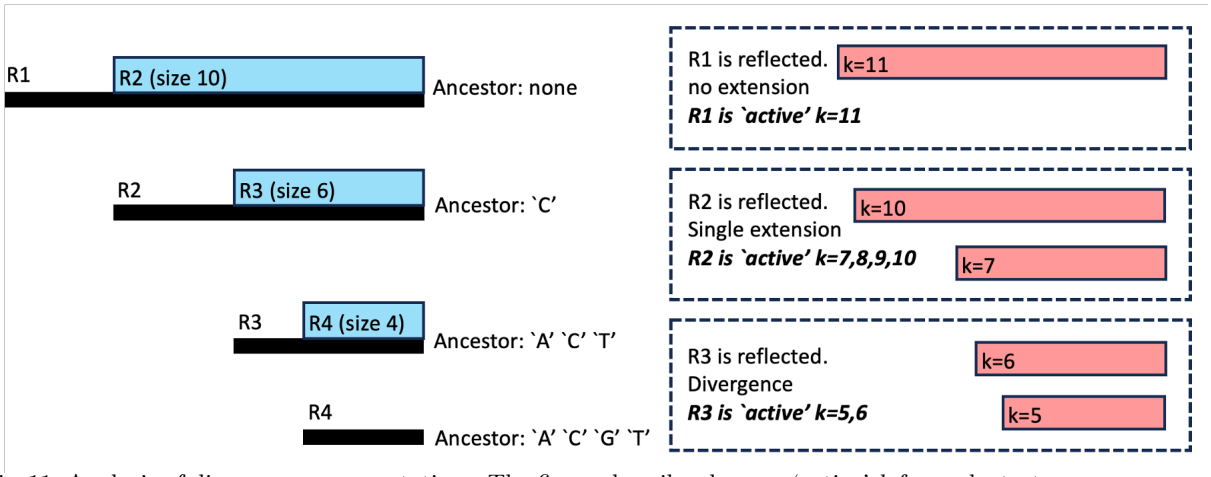

Fig. 11: Analysis of divergence representations. The figure describes how an 'active' k for each stratum or sequence is decided. The left figure shows some hierarchy of strata occurring as a suffix recursively. Ancestor means how many $ancestor_{\mathcal{U}}$ defined before are valid by some letters. That is, R1 has no ancestor, so no extension is possible if an extension meets that end. R4 has many ancestors, meaning multiple choices are possible. The sufficient and necessary condition for each Ancestor information being 'active' is analyzed in the dotted boxes. For example, focusing on the R2 case, since a stratifying region of length 6 is positioned at the last of the region. A 6-reflectum cannot exist at the last. Instead, 6-reflectum resides in R3, so at $k = 6$, a divergence is implied by R3. If $k = 6 \to 7$ is the transition, then R2 has a reflectum at the last position, and hence represents the connectivity. It has one extension, so it does not imply a divergence but is not a dead end too, suggesting possibly a simple path can be generated.

Algorithm 6 Count Maximal Unitigs of a de Bruijn graph of degree k

Input: The Pro$k$rustean graph $G = (V, E)$, $k \in \mathbb{N}$

Output: The number of maximal unitigs of the de Bruijn graph of $\mathcal{U}$ of degree k

```
Set count := 0
Set V' = {v ∈ V | the first of Pdecompose(v, k) is 'reflecting'}
for v ∈ V' do
    if v represents a sequence (not a stratum) and size(v) ≥ k then
        count++ and go to next iteration.
    end if
    if size(v) ≥ k then
        if left_cnt(v) = 0 then count++
        else if left_cnt(v) > 1 then count++
        else if right_cnt(v) > 1 then count += right_cnt(v)
        end if
    else if size(v) = k − 1 then
        if left_cnt(v) > 1 and right_cnt(v) = 1 then count++
        else if right_cnt(v) > 1 then count += right_cnt(v)
        end if
    end if
end for
```

Algorithm 7 Count maximal unitigs of de Bruijn graphs of k-mers of a k range

Input: Pro$k$rustean graph $G_{\mathcal{U}} = (V, E)$, $L \in \mathbb{N}$
Output: Count distinct k-mers for $k = L..N$, $N :=$ maximum sequence length

```
Set C, ∂C := each list of length N initialized with 0's.
C[L] = run Algorithm 6 to count maximal unitigs at L.
for v_S ∈ V do
    if S is sequence then
        if e := PdecomposeFirst(v_S, L − 1; G) is 'stratifying' then
            ∂C[|interval(e)| + 2] += 1
        end if
        ∂C[size(v_S) + 2] −= 1
    else
        if left_cnt(v_S) == 0 then
            if e := PdecomposeFirst(v_S, L − 1; G) is 'stratifying' then
                ∂C[|interval(e)| + 2] += 1
            end if
            ∂C[size(v_S) + 2] −= 1
        else if left_cnt(v_S) > 1 then
            if e := PdecomposeFirst(v_S, L − 1; G) is 'stratifying' then
                ∂C[|interval(e)| + 2] += 1
            end if
            if right_cnt(v_S) == 1 then ∂C[size(v_S) + 2] −= 1
            else ∂C[size(v_S) + 1] −= 1
            end if
        end if
        if right_cnt(v_S) > 1 then
            if e := PdecomposeLast(v_S, L − 1; G) is 'stratifying' then
                ∂C[|interval(e)| + 2] += right_cnt(v_S)
            end if
            ∂C[size(v_S) + 2] −= right_cnt(v_S)
        end if
    end if
    for k = L + 1..N do
        C[k] = ∂C[k − 1] + ∂C[k]
    end for
end for
```

4 Pro*k*rustean Graph Construction

In this section, the construction algorithm of Pro$k$rustean graph is introduced.

4.1 Additional Definitions and Theorems

Definition 11. *Consider two regions (S,i,j) and (S',i',j').*
if $S = S'$ & $max(i, i') \leq min(j, j')$,

$$(S, i, j) \cap (S', i', j') := (S, max(i, i'), min(j, j')) \tag{9}$$

otherwise

$$(S, i, j) \cap (S', i', j') = \varnothing \tag{10}$$

Definition 12. *For a string S occurring in $\mathcal{U}$, $\overline{S}$ is the string in* $\mathrm{StraStr}(\mathcal{U})$ *reflected by S.*

Theorem 4 says $\overline{S}$ for any string is unique in $\mathrm{StraStr}(\mathcal{U})$.

Definition 13. *Let $\sigma_l, \sigma_r \in \Sigma$ and $R \in \mathcal{S}_{\mathcal{U}}$ such that $occ_{\mathcal{U}}(\sigma_l R), occ_{\mathcal{U}}(R\sigma_r) > 0$. Attaching the letters to R, eg. $\sigma_l R$ we mean a new concatenated string that σ_l is on the immediate left of R.*

– σ_l *and* σ_r *are* <u>*exclusive*</u> *over* R *if and only if* $|occ_{\mathcal{U}}(\sigma_l R)| = |occ_{\mathcal{U}}(\sigma_l R\sigma_r)| = |occ_{\mathcal{U}}(R\sigma_r)|$.
– σ_r *is* <u>*right-exclusive*</u> *to* σ_l *if and only if* $|occ_{\mathcal{U}}(\sigma_l R)| = |occ_{\mathcal{U}}(\sigma_l R\sigma_r)|$.
– σ_l *is* <u>*left-exclusive*</u> *to* σ_r *if and only if* $|occ_{\mathcal{U}}(\sigma_l R\sigma_r)| = |occ_{\mathcal{U}}(R\sigma_r)|$.

Theorem 7. *Let* $R \in \mathcal{S}_{\mathcal{U}}$.
Let $\sigma_l, \sigma_r \in \Sigma$ *such that* $|occ_{\mathcal{U}}(\sigma_l R)|, |occ_{\mathcal{U}}(R\sigma_r)| > 0$.

1. σ_r *is not right-exclusive to some* $\sigma \in \Sigma$ *over* R *if and only if* $occ_{\{\overline{R\sigma_r}\}}(R) \subset \mathrm{StraRgn}(\mathcal{U})$ *and the shape of* $\overline{R\sigma_r}$ *is* $R\sigma_r\Sigma^*$.

2. σ_l *is not left-exclusive to some* $\sigma \in \Sigma$ *over* R *if and only if* $occ_{\{\overline{\sigma_l R}\}}(R) \subset \mathrm{StraRgn}(\mathcal{U})$ *and the shape of* $\overline{\sigma_l R}$ *is* $\Sigma^*\sigma_l R$.

3. σ_l *and* σ_r *are exclusive over* R *if and only if* $occ_{\{\overline{\sigma_l R\sigma_r}\}}(R) \subset \mathrm{StraRgn}(\mathcal{U})$ *and the shape of* $\overline{\sigma_l R}$ *and* $\overline{R\sigma_r}$ *is* $\Sigma^*\sigma_l R\sigma_r\Sigma^*$.

Proof. We first draw the information to be extracted from the theorem. The theorem is intended to search stratifying regions in which each R is the string. Since a stratum is a maximal repeat that extending it makes the occurrence smaller, normally an occurrence of R in $\overline{R\sigma_r}$ and $\overline{\sigma_l R}$ is stratifying. It is because $R\sigma_r$ has lower occurrences than R, so a string having the same occurrence (reflected) as $R\sigma_r$ is by definition $\overline{R\sigma_r}$. So, any method that can find $\overline{R\sigma_r}$, such as suffix trees or suffix arrays or bwt, can lead R the stratifying information of their occurrences.

The problem is that not all $\overline{R\sigma_r}$ and $\overline{\sigma_l R}$ have an occurrence of R as stratifying. For example, consider some σ_r is opposed to both 1. and 3. that σ_r is right-exclusive to some letter σ_l but σ_l is not left exclusive to σ_r. That is, whenever σ_l appears on left of R, σ_r always appears on right of R, but not the opposite. That means extending $\sigma_l R$ to right does not decrease the occurrence number. Then it is easy to verify that there exists a string in $\mathrm{StraStr}(\mathcal{U})$ that is sharing right edge with R and R is extended by σ_l on left. That means an occurrence of R is stratifying on $\overline{\sigma_l R}$ but not stratifying on $\overline{R\sigma_r}$.

With the insight, we show the theorem is correct. First, 1. and 2. are symmetrical, and each backward direction is straightforward to prove once the logic of forward is constructed. So we prove the forward of 1. Let σ_r is not right-exclusive to any letter. That means for any $\sigma \in \Sigma$, $\sigma R\sigma_r$ is less abundant than $R\sigma_r$. Now, consider $\overline{R\sigma_r}$ which we know always exists, by Theorem 4. Then R is a prefix of $\overline{R\sigma_r}$, but assume R is not a prefix of k to show it, i.e. $\sigma_l R\sigma_r$ is a substring of $\overline{R\sigma_r}$. Since $|occ_{\mathcal{U}}(R\sigma_r)| = |occ_{\mathcal{U}}(\overline{R\sigma_r})| = |occ_{\mathcal{U}}(\sigma_l R\sigma_r)|$ holds where the first equation is by definition of reflection and the second equality is because $\sigma_l R\sigma_r$ is assumed a substring of $\overline{R\sigma_r}$. This is contradiction because we started from $|occ_{\mathcal{U}}(R\sigma_r)| > |occ_{\mathcal{U}}(\sigma_l R\sigma_r)|$ above.

Therefore, R is a prefix of $\overline{R\sigma_r}$ and its occurrence is a stratifying region, and since R is a prefix the shape of $\overline{R\sigma_r}$ is $R\sigma_r\Sigma^*$.

The forward of case 3. is simpler that since σ_l, σ_r are exclusive each other over R, $R\sigma_r$ always has σ_l on left and vice versa. Then with the similar logic above, $\overline{\sigma_l R}$ reflects what $\overline{R\sigma_r}$ reflects, which is what $\overline{\sigma_l R\sigma_r}$ reflects. Therefore, R is stratifying in $\overline{\sigma_l R\sigma_r}$ and the string form is $\Sigma^*\sigma_l R\sigma_r\Sigma^*$.

The backwards are straightforward with counterpositives.

This theorem is required to find stratifying regions by a bottom-up approach, not a top-down approach. That is, finding all stratifiying regions $\mathrm{StraRgn}(\mathcal{U})$ is impossible in linear time $O(N)$ where N is the total length of sequences in $\mathcal{U}$. This is because just reading all $\mathrm{StraStr}(\mathcal{U})$ requires quadratic time in a worst case, and really is expensive in practice. Therefore, first, maximal repeats (strata) should be identified, and then for each maximal repeat R, stratifying regions such that the string is R should be found. This bottom-up approach can be implemented by suffix trees or suffix arrays or BWTs, as known that they can capture maximal repeats.

4.2 Straightforward implementation: Two suffix trees

We work with two suffix trees. Let $\mathcal{U}^r$ be the set of reversed strings of $\mathcal{U}$. Let $T_{\mathcal{U}}$ and $T_{\mathcal{U}^r}$ be each the generalized suffix tree built with $\mathcal{U}$ and $\mathcal{U}^r$, respectively.

Distinct Extensions	Suffix index
`A'R`C'	1111
`C'R`G'	2222
`C'R`T'	3333

Extension Characteristics	Conclusion
`A' is exclusive to `C' over R	Exists:
`C' is left exclusive to `G' and `T' over R	**Not** exists
`G' and `T' are not right exclusive to `C'	Exists:

Fig. 12: An example of Theorem 7. Depending on the distinct extensions on each stratum, whether the specific form of stratifying region exists or not is decided. For example, since A and C are exclusive to each other, some 'A'R'C' substring exists such that R is stratifying inside of it. The 'C' on left of R is not subject to any cases in Theorem 7. Then there must be no stratifying region of the string R such that C is at immediate left.

Theorem 8. *Let R be a string occurring in $\mathcal{U}$.*

$$R \text{ is a maximal repeat} \tag{11}$$

$$\text{if and only if} \tag{12}$$

$$\text{a node representing } R \text{ and its reversed string } R^r \text{ exists in each } T_{\mathcal{U}} \text{ and } T_{\mathcal{U}^r}\text{, repectively.} \tag{13}$$

Proof. By the definition of suffix tree, R is right-maximal if it is a node in $T_{\mathcal{U}}$ and is left-maximal if it is a node in $T_{\mathcal{U}^r}$.

Theorem 9. *Let R be a maximal repeat in $\mathcal{U}$ and $\sigma_l, \sigma_r \in \Sigma$ such that $|occ_{\mathcal{U}}(\sigma_l R)|, |occ_{\mathcal{U}}(R\sigma_r)| > 0$. Also, let $node(S)$ denotes the node representing S, when the tree of the node is clear.*

1. *If no $\sigma \in \Sigma$ is left-exclusive to σ_r over R, $node(R_r)$, the child node of $node(R)$ of $T_{\mathcal{U}}$ found from the branch σ_r, represents $\overline{R\sigma_r}$. Therefore, $(R_r, 1, |R|)$ is stratifying.*
2. *If no $\sigma \in \Sigma$ is right-exclusive to σ_l over R, $node(R_l^r)$, the child node of $node(R_l^r)$ of $T_{\mathcal{U}^r}$ found from the branch σ_l, represents the reverse string of $\overline{\sigma_l R^r}$. Therefore, $((R_l^r)^r, |R|, |(R_l^r)^r| - |R|)$ is stratifying.*
3. *If σ_l and σ_r are exclusive over R, consider $node(R_r)$, the child node of $node(R)$ of $T_{\mathcal{U}}$ found from the branch σ_r, and $node(R_l^r)$, the child node of $node(R^r)$ of $T_{\mathcal{U}^r}$ found from the branch σ_l, meaning R_r and the reverse of R_l^r matches the prefix and suffix by R. Merge R_r and the reverse of R_l^r by overlapping the R and generate S. Then $(S, |R_l^r| - |R|, |R_l^r|)$ is stratifying.*

Proof. 1. $node(R_r)$ found from the branch σ_r of $node(R)$ satisfies $R_r = \overline{R\sigma_r}$. First, R_r is a maximal repeat because it is represented by a node (right maximal), and no $\sigma \in \Sigma$ is left-exclusive to σ_r over R, meaning extending on the left of R, which is a prefix of R_r makes the occurrence decreased. Second, $(R_r, 1, |R|)$ is stratifying, because R_r is reflected by $R\sigma_r$. For contradiction, if a larger region is an occurrence of a stratum, then the stratum should be reflected by $R\sigma_r$ instead of R_r.

2. This is symmetric to 1., only carefully considering the reverse strings.
3. This is also similar to 1., but the exception is that R_r and $(R_l^r)^r$ are not maximal repeats. If R_r and $(R_l^r)^r$ are merged by overlapping the R, meaning concatenating $(R_l^r)^r[1..|R|]$, R, $R + R_r[|R| + 1..|R_r|]$ and obtain S, this is reflected by both $R\sigma_r$ and $\sigma_l R$. It is because σ_l and σ_r are exclusive, so extending R to S will not decrease the occurrence. Therefore, $(S, |R_l^r| - |R|, |R_l^r|)$ is stratifying.

With any library supporting generalized suffix tree constructions, implementing Pro*k*rustean is straightforward. The algorithm uses $O(N)$ time and space. However, the algorithm has a huge constant factor because building a suffix tree takes usually around $20N$ bytes. With two suffix trees and the affix link information, the space occupation is burdensome. Therefore, we use EBWT of $\mathcal{U}$ for the construction to further decrease the amount of time and space.

Algorithm 8 Stratification with two suffix trees

Input: Two suffix trees $T_{\mathcal{U}}$ and $T_{\mathcal{U}^r}$.
Output: StraStr($\mathcal{U}$) and StraRgn($\mathcal{U}$)

1: Set $affix(node(S)) = \varnothing$ for each node $node(S) \in T_{\mathcal{U}}$.
2: Set $StraStr = \mathcal{U}$.
3: Set $StraRgn = \varnothing$.
4: Construct $affix(node(S)) = node^r(S^r)$ for every $node(S)$, when $node^r(S) \in T_{\mathcal{U}^r}$ exists.
5: For each $node(S)$ with $affix(node(S)) \neq \varnothing$, add S to $StraStr$.
6: For each $node(S)$ with $affix(node(S)) \neq \varnothing$, consider branches of $node(S)$ and $affix(node(S))$. Collect each letter pair (σ_l, σ_r) from each branch of $node(S)$ and $affix(node(S))$ such that $\sigma_l R \sigma_r$ occurs. Also, $(\varnothing, \sigma_r)$ and $(\sigma_l, \varnothing)$ are considered too. Follow *Theorem* 9 and find stratifying regions, and add them to $StraRgn$.
7: return $StraStr$ and $StraRgn$

**Suffix trees are assumed to support finding the 'first occurrence' of the string of each node in constant time. The first occurrence of S is the one located on the leftmost of the concatenated string of $\mathcal{U}$ used to build $T_{\mathcal{U}}$, among occurrences of S in $\mathcal{U}$. The information can be constructed while building a suffix tree and usually libraries building suffix trees support it.
**In line 4, $affix(node(S))$ can be constructed by checking if for the first occurrence (S', i, j) of $node(S)$, a node in $T_{\mathcal{U}^r}$ has the last occurrence (S'^r, j, i), where the last occurrence can be implemented symmetrically as the first occurrence. Indeed, the structure is naturally accessible if we are working with the affix tree of $\mathcal{U}$. There exists $O(N)$ algorithms for constructing an affix tree.
**In line 6, suffix trees are assumed to be able to sample 'if (σ_l, σ_r) satisfies $\sigma_l R \sigma_r$ occurs in $\mathcal{U}$. This is because for each node the left extension is accessible as is the case in most suffix tree implementations. Go to a child node of the branch σ_r and then check valid left letters. It collects all information. $(\varnothing, \sigma_r)$ means $R\sigma_r$ is in interest, which is also accessible in suffix trees.

4.3 Space-efficient implementation: EBWT

We got some sense of how the two suffix trees easily find stratifying relationships. Using an EBWT or only one suffix tree/array theoretical means the same: We lose the information of the reversed suffix tree, and hence cannot catch when the left extension (cR) will diverge ($\overline{cR}$). Therefore, another strategy is used in EBWT that all 'representative' occurrences of strata are projected to sequences so that they can be assembled. Below definition prepares for it.

Definition 14. *Fix $\mathcal{U}$ and its generalized suffix array. $1stSuff_{\mathcal{U}}(c_l, S, c_r)$ is the lowest suffix index among suffices that start with Sc_r and has c_l on left.*

$1stSuff(c_l, S, c_r)$ is a number but since the context is a generalized suffix array, we can theoretically think it as (S, i) that the suffix of S at i. Note that this tuple is not obtained for free. $(S, i) = 1stSuff(c_l, S, c_r)$ is used when the implied information is required.

The theorem below catches another interesting intuition around stratification. Let $1stSuffCollection_{\mathcal{U}}$ the set of all possible $1stSuff_{\mathcal{U}}$ in $\mathcal{U}$, i.e. $1stSuffCollection_{\mathcal{U}} = \{1stSuff_{\mathcal{U}}(c_l, S, c_r) | S$ is a stratum, $c_l, c_r \in \}$. $1stSuffCollection_{\mathcal{U}}$ is going to be collected by EBWT in $O(N)$ time.

Theorem 10. *Let (R, p, q) be stratifying in $\mathcal{U}$ and $R[p-1] = c_l$ for some letter $c_l \in \Sigma^{\varnothing}$. Then:*

$$\textit{for any } c_r \in \Sigma^{\varnothing}, \textit{ if } (S, i) = 1stSuff_{\mathcal{U}}(c_l, str(R, p, q), c_r)$$

$$\textit{then } (S, i-(p-1)) = 1stSuff_{\mathcal{U}}(*, R, *) \textit{ for some letter in } *$$

Proof. The theorem says if R and $str(R, p, q)$ are in stratifying relationship at the region (R, p, q), then the suffix $1stSuff_{\mathcal{U}}(c_l, str(R, p, q), c_r)$ is always included in some $1stSuff_{\mathcal{U}}(*, R, *)$ if $R[p-1] = c_l$. For example, if (R, p, q) =('AAACCC',3,5) then $str(R, p, q)$ ='ACC' and $R[p-1]$ ='A'. Now, $(S, i) = 1stSuff_{\mathcal{U}}(c_l, str(R, p, q), c_r)$ should be in form $S[i...]$ ='ACC...'. Then theorem says $(S, i-(p-1))$ should be 'AAACCC...' that includes 'ACC....' and it must be some $1stSuff_{\mathcal{U}}(*, R, *)$, meaning $1stSuffCollection_{\mathcal{U}}$ includes it.

For the proof, assume that $(S, i-(p-1))$ is not occurring in any form of $1stSuff_{\mathcal{U}}(*, R, *)$. But still the suffix $(S, i-(p-1))$ starts with R because (S, i) starts with $str(R, p, q)$. Since (R, p, q) is stratifying

and $R[p-1] = S[i-1] = c_l$, extending (R, p, q) to $(R, 1, q)$ the occurrence of the strings of the regions do not change. Likewise, extending (S, i) to $(S, i-(p-1))$, a prefix of (S, i) is $str(R, p, q)$ and it extends left to $(S, i-(p-1))$ to have R as a prefix. If R is not a prefix, that means some divergence happend in the extension, which is contradiction for (R, p, q) being stratifying.

We know $(S, i-(p-1))$ starts with R now, and assumed it is not in $1stSuff_{\mathcal{U}}(*, R, *)$. Then let the left/right characters of the R part in $(S, i-(p-1))$ is $c'_l, c'_r \in \Sigma^{\varnothing}$. There must be another suffix (S', i') that is in $1stSuff_{\mathcal{U}}(c'_l, R, c'_r)$. Since (S', i') comes earlier than $(S, i-(p-1))$ in the generalized suffix array, and a prefix of them are both R, the order is decided by the rest. From the example above, $(S, i-(p-1))$ =‘AAACCC...’ comes later than (S, i') =‘AAACCC,,,’. This shows the contradiction that ‘ACCC...’ must be also not earliest because ‘ACCC,,,’ will come earlier than that. That is, the suffix $(S', i' + |R| - (p-1))$ comes earlier than (S, i) and they both starts with $str(R, p, q)$. Therefore, (S, i) cannot be in $1stSuff_{\mathcal{U}}(*, str(R, p, q), *)$ which is contradiction. □

The theorem implies that $1stSuffCollection_{\mathcal{U}}$ contains all information of stratification. Since they are all represented as suffices in the sequences, they are ‘projected’ in the sequence area. Then using all elements of $1stSuffCollection_{\mathcal{U}}$ in a specific sequence, they can be assembled by checking their relationships. The algorithm *Algorithm* 9 does the task in time of $O(|\text{projected suffices in a sequence}|)$, which is efficient because the number of projected suffices is equal to $|\text{StraRgn}(\mathcal{U})|$.

One last caution: collecting $1stSuffCollection_{\mathcal{U}}$ blindly for every distinct (c_l, c_r) can complicate the algorithm *Algorithm* 9. This is because all stratification relationships are in $1stSuffCollection_{\mathcal{U}}$ but not all projected suffices in $1stSuffCollection_{\mathcal{U}}$ are projected from stratifying regions. This is evident from *Theorem* 7 that extensions not subject to 1.,2. and 3. are not direct immediate left or right letters of a stratifying region. Therefore, we slightly modify the definition of $1stSuffCollection_{\mathcal{U}}$ that for each $1stSuff_{\mathcal{U}}(c_l, S, c_r)$, either c_l or c_r should satisfy the condition in *Theorem* 7, and the suffix index has to be lowest for the appreciated extension.

Definition 15. *Fix $\mathcal{U}$ and its generalized suffix array. $\underline{1stSuffCollection_{\mathcal{U}}}$ is a set of $1stSuff_{\mathcal{U}}(c_l, R, c_r)$ for all maximal repeat R and all $c_l, c_r \in \Sigma$ satisfying the relationship over R in one of the cases in Theorem 7.*

Stage1: Gather $1stSuffCollection_{\mathcal{U}}$ The concept ‘first suffix in an interval’ is comprehensible in suffix tree/array and EBWT. Those techniques hold the information of intervals of suffix array for each string S. That is, at node of S, the suffices starting with S are identified as an suffix index interval.

The same information is accessible for EBWT. [22] described how to compute LCP from bwt in $O(N)$ time where $1stSuffix$ is indirectly passed through the implied suffix tree structure traveled. The implicit suffix tree traversing algorithm with EBWT was originally introduced by [3].

The suffix tree navigation in [22] holds the information of right extended child intervals every time. For example, when string S is implicitly represented, the algorithm holds suffix index intervals of suffices starting with Sa for all $a \in \Sigma^{\varnothing}$. Then, the algorithm splits the intervals for left extensions, i.e. cS for all $c \in \Sigma^{\varnothing}$ now has each child intervals of suffix indices for each cSa form. Therefore, we can collect distinct (c, a) per each implicit node, meaning cSa occurs in $\mathcal{U}$, if the cSa interval in the implicit node cS is not empty.

We can collect $1stSuffCollection_{\mathcal{U}}$ now. Note again that the suffix of form (S, i) cannot be directly collected unless the suffix array is sampled. We travel the implicit suffix tree as [22] describe. First, a maximal repeat S can be easily inferred by its definition, with the distinct extensions of S obtained above. Second, by the *Theorem* 7, get valid distinct extensions. Note that exclusiveness described in the theorem is exactly talking with distinct extensions. Lastly, once the distinct extension letter is determined, we find the $1stSuff_{\mathcal{U}}(c_l, S, c_r)$. For the first suffix of a Sc_r form, it is already there that the first index of the suffix interval of Sc_r is it. For the first suffix of c_lS form, since the suffix index interval for c_lS is literally about starting c_lS, the c_lS has to be reversed to S using the ‘select’ function of fm-index.

The collected information is in form $(suffIdx, isLowest(R), id(R))$, $suffIdx := 1stSuff_{\mathcal{U}}(c_l, R, c_r)$ and $isLowest(R) :=$ among all suffices found from $1stSuff_{\mathcal{U}}(*, R, *)$, the $suffIdx$ is the lowest for R, and $id(R)$ is the identifier of a maximal repeat R. Please do not be confused about ‘lowest’ in $isLowest$ against the concept of $1stSuff_{\mathcal{U}}(c_l, R, c_r)$. It means among all suffices of a prefix R, the index is lowest. So, it is lowest of $1stSuff_{\mathcal{U}}(*, R, *)$ with any input in $*$. Lastly, R is not stored as a string but only $id(R) : |R|$ pair is stored.

Stage2: Build stratification We have all projected suffices and maximal repeat information annotated, that can be $(suffIdx, isLowest(R), id(R))$ and access to $size(R)$ is implied. The implementation allows a $fetch(suffIdx)$ that returns all list of $(isLowest(R), id(R))$ in constant time.

Now, for each sequence, use fm-index to check each suffix index at each sequence position and fetch the annotated suffices above. For example, start from the terminal symbol $ at id(sequence), run fm-index to read the sequence from backward and run $fetch(suffIdx)$ at each position. The consequence is $ProjRgn_{\mathcal{U}}(S) :=$ annotated positions of form $(pos, [(isLowest(R_1), id(R_1), size(R_1)), , ..., (isLowest(R_n), id(R_n), size(R_n))]$. The list is composed of regions each annotated with maximal repeat because $(pos, pos + size(R_i)$ is a valid interval in the sequence. The positions in $ProjRgn_{\mathcal{U}}(S)$ are ordered, and regions annotated at each position are ordered by the $size(R_i)$. The ordering can be done in $O(|S|)$ time utilizing $O(|S|)$ space because the projected regions cannot exceed $O(|S|)$ which is derived from the theorem *Theorem* 3. In practice, the number of regions is negligible so we directly sorted them.

Once the projected intervals and their annotations are ready, then *Algorithm* 9 finds stratifying relationships. The algorithm has three nested while loop that every loop contribute to moving the $rgnPointers$. Since |annotations in $rgnPointers$| is exactly the whole annotation of projected stratifying regions, and the while loop ends when the pointers are consumed, meaning largest annotation of the earliest position is reached, the algorithm runs in $O(|\text{annotations in } rgnPointers|)$. The other parts are all constant operations.

The algorithm is very efficient and parallelizable. Distributing sequence indices to threads, they can independently work in sequence-based.

Algorithm 9 Build Stratification

Input: id(sequence), $ProjRgn_{\mathcal{U}}(S)$.

Output:: the pair (id(sequence):stratifying regions) and a set of (id(R):Stratifying regions) for R's that are marked $isLowest(R) = true$ in the input.

Set $rgnPointers$ = Consuming pointer map position:region. Each region is one in the list of $ProjRgn_{\mathcal{U}}(S)$[pos]. initialized to smallest element in $ProjRgn_{\mathcal{U}}(S)[pos]$.
Set $nextPosPointers$ = Pointer map position:nextPos where nextPos is the next position of pos in $ProjRgn_{\mathcal{U}}(S)$.
Set $focusPos$ = last pos in $ProjRgn_{\mathcal{U}}(S)$
Set $focusRgn = rgnPointers$[focusPos]
##

Use Algorithm 10 to collect stratifying regions per each stratum, that is represented by each region in $ProjRgn_{\mathcal{U}}(S)$ such that $isLowest(R) = true$.

##
for pos:region in $rgnPointers$ **do**
    if region $\neq \varnothing$ **then**
        Report (last region in $ProjRgn_{\mathcal{U}}(S)[pos]$) is stratifying in id(sequence)
    end if
end for

Algorithm 10 Build Stratification - Find Stratifying regions per each stratum projected in a sequence

```
All contexts in Algorithm 9 is used
while focusRgn ≠ ∅ do
    Set regions
    if focusRgn.isLowest = True then # Add stratifying region that is at position 1.
        if focusRgn is not first in ProjRgn_U(S)[focusPos] then
            regions.Add(direct child(smaller) region in ProjRgn_U(S)[focusPos]).
        end if
    end if
    focusPostPos= nextPosPointers[focusPos]
    while focusPostPos ≠ ∅ do
        focusPostRgn= rgnPointers[focusPostPos]
        if focusRgn does not include focusPostRgn then
            break
        end if
        # Find the largest included post region.
        Set parentRgnOfPostRgn = parent (larger) region in ProjRgn_U(S)[focusPostPos]
        while parentRgnOfPostRgn ≠ ∅ and focusRgn includes parentRgnOfPostRgn do
            focusPostRgn = parentRgnOfPostRgn
            parentRgnOfPostRgn = parent (larger) region in ProjRgn_U(S)[focusPostPos]
        end while
        if focusRgn.isLowest = True then
            regions.Add(focusPostRgn)
        end if# move consuming pointer
        if parentRgnOfPostRgn ≠ ∅ then
            rgnPointers[focusPostPos] = parentRgnOfPostRgn
        else
            rgnPointers[focusPostPos] = ∅
            nextPosPointers[focusPos] = nextPosPointers[focusPostPos]
        end if
        focusPostPos = nextPosPointers[focusPos]
    end while
    if focusRgn.isLowest = True then
        Report regions as stratifying in the maximal repeat of focusRgn
    end if
    if focusRgn has larger region in ProjRgn_U(S)[focusPos] then
        focusRgn = next (larger) region of focusRgn in ProjRgn_U(S)[focusPos]
    else if focusPos is not first in ProjRgn_U(S)[focusPos] then
        focusPos= previous position of focusPos in ProjRgn_U(S)
        focusRgn= rgnPointers[focusPos]
    else
        focusRgn= ∅
    end if
end while
```