Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

vMUS-dBG: A Novel De Bruijn Graph Model for De Novo Genome Assembly Using Variable-Length Minimum Unique Substrings

Andrews Frimpong Adu

afadu1@st.knust.edu.gh

Kwame Nkrumah University of Science and Technology

Elliot Sarpong Menkah

Kwame Nkrumah University of Science and Technology

Peter Amoako-Yirenkyi

Kwame Nkrumah University of Science and Technology

Samson Pandam Salifu

Kwame Nkrumah University of Science and Technology

Research Article

Keywords: minimum unique substring, repeats, suffix tree, sequence uniqueness, de bruijn graphs, vMUS

Posted Date: April 27th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-8997819/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

*v*MUS-dBG: A Novel De Bruijn Graph Model for De Novo Genome Assembly Using Variable-Length Minimum Unique Substrings

Andrews Frimpong Adu[1*], Elliot Sarpong Menkah[2], Peter Amoako-Yirenkyi[3], Samson Pandam Salifu[1]

[1*]Biochemistry and Biotechnology, Kwame Nkrumah University of Science and Technology, KNUST, Kumasi Ghana.
[2]Mathematics, Kwame Nkrumah University of Science and Technology, KNUST, Kumasi Ghana.
[3]Chemistry, Kwame Nkrumah University of Science and Technology, KNUST, Kumasi Ghana.

*Corresponding author(s). E-mail(s): afadu1@st.knust.edu.gh;
Contributing authors: esmenkah@knust.edu.gh;
amoakoyirenkyi@knust.edu.gh; sspandam@knust.edu.gh;

Abstract

De novo genome assembly using de Bruijn graphs (DBGs) typically relies on fixed-length $\boldsymbol{k}$-mers as the nodes of the graph. While this approach is effective, it presents a fundamental trade-off: smaller $\boldsymbol{k}$ values tend to collapse repeats, whereas larger $\boldsymbol{k}$ values can result in fragmentation, particularly in low-coverage regions. Although multi-$\boldsymbol{k}$ and variable-order methods help mitigate these issues, they still rely on fixed-length topology or heuristic parameter selection. In this work, we introduce a de Bruijn graph constructed from Minimum Unique Substrings (MUSs), substrings that appear exactly once within the genome. This new graph is referred to as the variable-length MUS de Bruijn graph ($\boldsymbol{v}$MUS-dBG). In the $\boldsymbol{v}$MUS-dBG, the nodes are defined by read-extracted MUS anchors, and directed edges represent read-supported transitions between successive occurrences of MUSs. Each edge is also enhanced with instance-level metadata to preserve positional weights (repeats) and support counts. This innovative design eliminates the necessity for a global $\boldsymbol{k}$-mer selection, while producing a concrete, repeat-aware graph construction that operates differently from the abstract manifold-style DBG model. Our experiments using real 24x *E. coli K12* HiFi data

demonstrate that a prototype implementation of our approach achieves contiguity and accuracy comparable to that of a fixed-k method. These results establish MUS-based variable-length graph construction as a principled and biologically grounded alternative to fixed-k De Bruijn graph assembly to explore.

Keywords: minimum unique substring, repeats, suffix tree, sequence uniqueness, de bruijn graphs, vMUS

1 Introduction

Genome assembly algorithms have historically used graph-based models to represent and reconstruct DNA sequences from fragmented reads produced by sequencing technologies. Early methods based on overlap graphs [1], depict reads as nodes and represent the suffix-prefix overlaps as edges. This approach reduces assembly to the problem of finding a Hamiltonian path (path that visits every node exactly once), which is conceptually straightforward but leads to an NP-complete problem and does not scale well with modern high-throughput sequencing datasets. The introduction of the de Bruijn graph (DBG) model significantly transformed this field by rephrasing genome assembly as an Eulerian path problem [2–4]. In a DBG, input reads are first broken into substrings of fixed length k, called k-mers, which represent the graph nodes, and edges are represented by $(k+1)$-mers. The genome is then reconstructed from the Eulerian path in the graph. This redefinition enabled more efficient and scalable assembly methods and remains the basis for many contemporary assemblers.

Although highly effective and efficient, the de Bruijn strategy is limited by the requirement to use a global fixed value for k-mer length. This value is determined by the user and introduces a parameter bias that lacks a universally optimal solution. The choice of k therefore, has a significant impact on the quality of the assembly output [5]. Selecting smaller k values may improve coverage in low-depth regions but can lead to the collapse of repetitive sequences. This results in a tangled graph that limits the length of contiguous sequences (contigs). On the contrary, larger k values help resolve these repeats but are more susceptible to fragmentation. Even a slight drop in coverage can break the graph into disconnected components. Both scenarios lead to suboptimal and fragmented assemblies. This instability means that the graph?s topology can change significantly based on the user?s choice of k.

To address this trade-off, many modern assemblers employ multi-k and adaptive strategies, including iterative graph construction [6–8] and variable-order graphs [9–12]. The iterative approach increases the value of k in each iteration, while variable-order graphs represent all orders of k in a single data structure. Although these methods enhance robustness, they still rely on fixed-length k-mers as the primary graph components, necessitating heuristic decisions about parameter selection and graph simplification. Moreover, [9] introduced the manifold de Bruijn graph as an attempt to avoid the need to settle on a single k-mer length at any stage, but it has been of theoretical interest [8, 12]. These graphs generalize DBGs by allowing substrings of arbitrary length as nodes of the graph instead of fixed-length k-mers.

Technically, these approaches tend to address the symptoms of the problem rather than its underlying structural limitations, resulting in poor performance with complex datasets, such as polyploid or cancer genomes [13].

In this study, we introduce the variable-length Minimum Unique Substring de Bruijn graph (vMUS-dBG). This is a practical assembly graph model where graph nodes consist of variable-length minimum unique substrings (MUSs) and edges represent read-supported transitions between successive MUS occurrences. A MUS is the shortest substring that uniquely identifies a region in the genome, with its length determined adaptively based on the local sequence context [14, 15]. This property is transformative because, by definition, a MUS must lie at the boundary between a repetitive region [14, 15]. Therefore, any graph constructed from MUSs will automatically have these repeat boundaries encoded in its structure, reducing redundancy and complexity of genome reconstruction.

This model, therefore, eliminates the necessity for a global k-mer value and makes the assembly process dependent solely on the inherent information present in the genome. The vMUS-dBG is motivated by [9] and this is the first implementation of such approach to our best of knowledge. We evaluate vMUS-dBG on a sampled real data, demonstrating the assembly reconstruction from MUSs. We also evaluate the graph construction and traversal processes, resource usage, and highlighting its scalability.

2 Conceptual and Theoretical Framework

2.1 Preliminaries

Since assembly tools employ varying terminology, we provide definitions tailored to the graph structures used in this work. All graphs considered are directed. Let $G = (U, V)$ denote a graph with node set U and edge set V. A node u *precedes* (resp. *follows*) a node v if there is an edge $u \to v$ (resp. $v \to u$) in G. The degree of u, denoted $\deg(u)$, is the number of edges incident to u. The in-degree and out-degree of u are $\deg^-(u)$ and $\deg^+(u)$, which represent the number of nodes preceding and following u, respectively. A node with $\deg^-(u) = 0$ is a *source*, whiles a node with $\deg^+(u) = 0$ is a *sink*. A node u is *1-in-1-out* if $\deg^-(u) = \deg^+(u) = 1$.

For graphs $G_1 = (U_1, V_1)$ and $G_2 = (U_2, V_2)$, their union is $G_1 \cup G_2 = (U_1 \cup U_2,\ V_1 \cup V_2)$. A *walk* is a sequence of alternating nodes and edges in which consecutive elements are incident. A *path* is a walk that does not revisit nodes, beginning and ending at nodes of degree one, with all internal nodes of degree two. A *cycle* is a closed walk in which every node has degree two.

A node u is a *bifurcation* if it has more than one incoming edge or more than one outgoing edge. A node is a *sentinel* if it corresponds to the first or last k-mer of any input read. A *junction* is any node that is a bifurcation, a sentinel, or both. A directed path in G is a *maximal non-branching path* (MNBP) if its endpoints are junctions and all internal nodes are 1-in-1-out.

2.2 The De Bruijn Graphs

Let R be a DNA read over the alphabet $\Sigma = \{A, C, G, T\}$ of length N, and let $\mathcal{R} = \{R_1, \ldots, R_m\}$ be a set of reads. For indices $i \leq j$, denote the substring $R[i : j]$, and call any length-p substring of a read a p-mer. A p-mer is a *prefix* of R if it equals $R[1 : p]$, and a *suffix* if it equals $R[N - p + 1 : N]$.

2.2.1 De Bruijn Graph for a String

For a fixed p, the order-p de Bruijn graph $G_p(\mathcal{R}) = (U, V)$ is constructed as follows. Let F be the multiset of all p-mers extracted from the reads.

- **Nodes** U: Each node is a $(p-1)$-mer that appears as the prefix or suffix of some p-mer in F: $U = \{\, d \in \Sigma^{p-1} \mid \exists f \in F :\ d = \text{prefix}(f, p-1) \ \vee\ d = \text{suffix}(f, p-1) \,\}$.
- **Edges** V: Each directed edge corresponds to a p-mer and links the overlapping $(p-1)$-mer prefix and suffix: $V = \{\, (\text{prefix}(f, p-1),\ \text{suffix}(f, p-1)) \mid f \in F \,\}$. Thus, for $v_1 = a_1 \ldots a_{p-1}$ and $v_2 = a_2 \ldots a_p$, an edge $v_1 \to v_2$ exists whenever their $(p-2)$-overlap matches.

2.2.2 De Bruijn Graph for a Collection of Strings

For a set of reads $\mathcal{R}$ and an integer p, the order-p de Bruijn graph is obtained by taking the union of the graphs constructed from each read: $G(\mathcal{R}, p) = G(r_1, p) \cup G(r_2, p) \cup \cdots \cup G(r_m, p)$, where nodes with identical $(p-1)$-mer labels are merged. A walk (or path) t in $G(\mathcal{R}, p)$ is said to *spell* a read r if t is exactly the path corresponding to $G(r, p)$.

3 Methods

3.1 From *k*-mers to Minimum Uunique Substrings (MUS)

Figure 1 illustrates the full vMUS-dBG construction pipeline. The process begins by extracting all minimum unique substrings (MUSs) from the read set. Let $\mathcal{R}$ be the set of reads, and let $R_k \in \mathcal{R}$ be a read over the alphabet Σ. We denote by $|R_k|$ the length of R_k and by $R_k[i : j]$ the substring spanning positions i to j. A substring w of R_k is a MUS if it appears exactly once in the entire read set $\mathcal{R}$ (considering both forward and reverse complements) whiles its proper prefixes or suffixes are repeats. A maximal repeat (MR), on the other hand, is a substring with an occurrence greater than 2 and cannot be extended on either side while still being a repeat [14]. The complete set of MUSs extracted from R_k is denoted by $\mathcal{M}$.

To illustrate these definitions, consider the example read $R[1 : 10] =$ ATGCTAGCAC. A MUS is a substring that occurs exactly once in R and cannot be shortened at either end without losing its unique occurrence. In contrast, a repeat (maximal repeat) is a substring that occurs at least twice and cannot be extended to the left or right while preserving its repetitive nature. For this read, the MUSs are: $\{\text{AT}, \text{TG}, \text{CT}, \text{TA}, \text{AG}, \text{GCA}, \text{AC}\}$.

These MUSs coincide with the points where maximal repeats terminate, effectively acting as anchors at the boundaries of repeated regions. For example, the substring GC

appears twice in R, at positions $R[3:4]$ and $R[7:8]$. Each occurrence is immediately followed by a MUS: $R[4:5] = \text{CT}$ and $R[8:9] = \text{CA}$. This illustrates the structural relationship between MUSs and MRs. Thus, MUSs mark the endpoints of maximal repeats and thus partition the read into uniquely anchored segments. For notational simplicity, we refer to all MUSs as *words*, denoted $w(i,j) \in \mathcal{M}$.

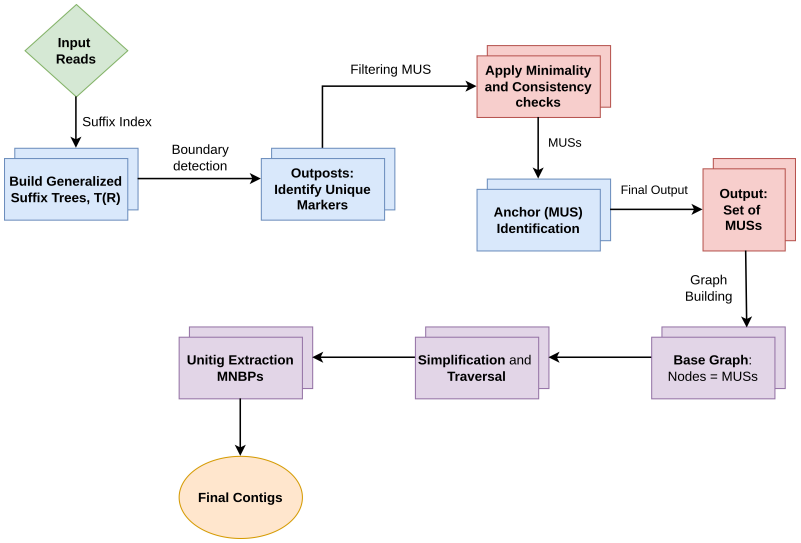


Fig. 1 Variable-length MUS-dBG (vMUS-dBG) pipeline. Stage 1 (MUS Extraction): 1) The algorithm builds a generalized suffix tree $\mathcal{T}(\mathcal{R})$ from the input read set $\mathcal{R}$. 2) It traverses this tree to identify *outposts*, which are unique prefixes of suffixes, and records their boundaries in the arrays $\texttt{Right}_k$ and $\texttt{Left}_k$. 3) It then determines the *anchor set* $A_{\mathcal{R}}(w)$, which serves as the operational definition of a minimal unique substring (MUS) for the reads. This procedure enforces three necessary properties–*consistency*, *left-minimality*, and *right-minimality*–to infer uniqueness in the set of reads. Stage 2 (Graph Construction): A MUS-based graph is then built from this set of MUSs, with nodes representing the MUSs and edges indicating the overlaps between consecutive MUSs. By traversing the MUS-based graph, unitigs are identified, leading to the construction and extraction of the final set of contigs.

3.2 vMUS-dBG Representation

For the set of reads $\mathcal{R} = \{\text{ATGCTA, GCTAGC, TAGCAC, GCACAT, ACATGC}\}$ from the circular genome R, the extracted MUSs are $A_{\mathcal{R}}(w) = \{\text{AT, TG, CT, TA, AG, GCA, AC}\}$, as shown in Table 1.

We denote an `occurrence` of a MUS $w \in \mathcal{M}$ in a read R_k by $(w, k, [s, e])$, where k indicates which read, s is the start index and e is the end index of w in R_k. The assembly graph is a directed multigraph $G_w(\mathcal{R}) = (U, V)$ where:

- The set of nodes U is composed of the collection of all MUSs ($\mathcal{A}$) and two special terminal nodes (for when the MUS do not start or end the read): a source node, U_{start}, and a sink node, U_{end} (Table 1). Formally, $U = A_{\mathcal{R}}(w) \cup \{U_{\text{start}}, U_{\text{end}}\}$. Thus, $U = \{AT, TG, CT, TA, AG, GCA, AC, U_{\text{start}}, U_{\text{end}}\}$.

Table 1 The set of MUSs extracted using Figure 1 for the set of reads $\mathcal{R}$ from the genome R.

Read, R_k	**MUS per read**
ATGCTA	AT, TG, CT, TA
GCTAGC	CT, TA, AG
TAGCAC	TA, AG, GCA, AC
GCACAT	GCA, AC, AT
ACATGC	AC, AT, TG

- The edge set V represent the structural transitions and genomic intervals between them.

For each directed edge $(u, v) \in V$, we assign a multiplicity $f(u, v)$ to represent the number of times the adjacency of that edge is observed across all reads. Multiple reads can induce the same directed edge pair $(u \to v)$. We formally define a flow function $f : U \times U \to \mathbb{N}$ as follows:

$$f(u, v) = \text{the number of reads that produce the edge } (u \to v).$$

As a result, the edge set forms a multiset where each edge transition $(u \to v)$ has a multiplicity of $f(u, v) \geq 1$. Additionally, for every edge instance (u, v), we maintain a list of attributes. For edges that correspond to two consecutive MUSs within a read, these attributes are:

- **Weight (δ):** Each occurrence of $(u \to v)$ in a read is annotated with a weight value describing the positional offset between the MUSs. If the MUS at node u occurs at positions $[s_u, e_u]$ and the MUS at node v at $[s_v, e_v]$ in R_k, then the weight is

$$\delta(u, v) = s_v - s_u. \tag{1}$$

- **Weight Tag (τ):** The explicit sequence representing the repeat unit between two MUSs:

$$\tau = \begin{cases} R_k[e_u + 1 : s_v - 1], & \delta > |u|, \\ \varepsilon, & \text{(overlap case/adjacent)}, \end{cases} \tag{2}$$

  where ε is the empty string.

3.2.1 Edge Construction Rules

For each edge occurrence contributed by a read R_k, we record the paired label (δ, τ). Thus, every directed edge $u \to v$ accumulates a list of such labels, denoted $\{(\delta, \tau)\}$, aggregated over all reads in which the edge occurs. These labels encode both the relative offset between consecutive MUSs. We define two types of edge annotations: (1) unique edges, where the consecutive MUSs are adjacent (overlap) (2) repeat edges,

where there is a gap and a MR covers the intervals. We denote all such collection of labels for an edge type (u, v) by $\mathcal{I}(u, v)$. To construct these edge labels, we distinguish three cases:

- **Prefix Edges (Initial Substring).** If the first MUS $(w_1, [s_1, e_1])$ occurrence begins at position $s_1 > 1$, then it does not start at the beginning of R_k. In this case, we set $\delta = s_1 - 1$ and define the tag as the prefix

$$\tau = R_k[1 : \delta],$$

  corresponding to the initial segment of the read preceding the first MUS.
- **Internal Edges (Consecutive MUSs).** For each consecutive pair of MUSs $(w_i, [s_i, e_i])$ and $(w_{i+1}, [s_{i+1}, e_{i+1}])$ in R_k (for $1 \leq i \leq p - 1$), we set

$$\delta = s_{i+1} - s_i.$$

  If $\delta > |u|$, then v begins after u ends, indicating a gap (as illustrated in Equation 2). We therefore assign

$$\tau = R_k[(e_i + 1) : (s_{i+1} - 1)],$$

  the intervening substring between the two MUSs. Otherwise, when $\delta \leq |u|$, the MUSs overlap, and we set $\tau = \varepsilon$ (the empty string).
- **Suffix Edges (Final Substring).** If the last MUS ends at position $e_p < |R_k|$, then the read contains a trailing segment not covered by any MUS. We record

$$\delta = |R_k| - e_p, \qquad \tau = R_k[(e_p + 1) : |R_k|],$$

  the suffix of the read following the final MUS.

3.2.2 Node Properties (Prefix/Suffix Maximization)

For each node $u \in U$, we maintain auxiliary metadata that summarizes the appearance of u across the read set. Let $P(u)$ represent the property set associated with u, which includes two key descriptors:

- **Maximal Prefix**: The longest sequence of characters that precedes any occurrence of u in the reads, together with its length.
- **Maximal Suffix**: The longest sequence of characters that follows any occurrence of u in the reads, together with its length.

In addition, we record analogous global descriptors: the maximal prefix over the entire read set at the special start node U_{start}, and the maximal suffix over the entire read set at the terminal node U_{end}. Collectively, these metadata and edge construction rules define the structure illustrated in Figure 2.

3.2.3 The Assembly Graph

Definition 1 (Assembly Graph) The **assembly graph** is therefore given as a directed multigraph

$$G_{\mathcal{M}} = (U_{\mathcal{M}}, V_{\mathcal{M}}, f_{\mathcal{M}}, \mathcal{I}_{\mathcal{M}}, P(u)_{\mathcal{M}}),$$

$$
\begin{aligned}
\text{AT} &\rightarrow \{\text{TG} : 2\},\\
\text{TG} &\rightarrow \{\text{CT} : 1,\ \text{LAST_NODE} : 1\},\\
\text{CT} &\rightarrow \{\text{TA} : 2\},\\
\text{FIRST_NODE} &\rightarrow \{\text{CT} : 1\},\\
\text{TA} &\rightarrow \{\text{AG} : 2\},\\
\text{AG} &\rightarrow \{\text{LAST_NODE} : 1,\ \text{GCA} : 1\},\\
\text{GCA} &\rightarrow \{\text{AC} : 2\},\\
\text{AC} &\rightarrow \{\text{AT} : 2\}.
\end{aligned}
$$

Fig. 3 This is the assembly graph represented by using an adjacency list, where $U_{\text{start}} = FIRST_NODE$ and $U_{\text{end}} = LAST_NODE$. Each node corresponds to a MUS, in addition to $FIRST_NODE$ and $LAST_NODE$. For Example, Node AT has outgoing neighbors TG, with a flow of 2. Node TG connects to outgoing neighbors CT and $LAST_NODE$. Node CT has outgoing neighbors leading to TA, and so on.

where:

- $U_{\mathcal{M}} = \bigcup_i \mathcal{M}_i \cup \{U_{\text{start}}, U_{\text{end}}\}$,
- $V_{\mathcal{M}} \subseteq U \times U$ is the multiset of directed edges,
- $f_{\mathcal{M}} : V \rightarrow \mathbb{N}$ flow function or multiplicity, counting the number of times the adjacency (u, v) is observed,
- $\mathcal{I}_{\mathcal{M}} : V \rightarrow \mathbb{N} \times \Sigma^*$ annotation function, and $P(u)$ assigns to each vertex its prefix and suffix metadata. This is depicted in Figures Figure 2, 3, and 4.

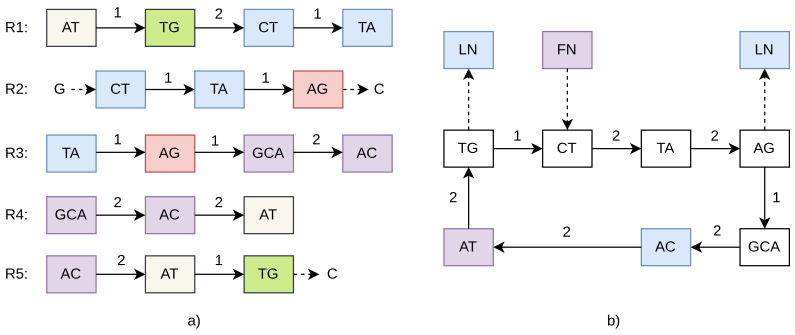


Fig. 2 vMUS-dBG representation for a set of reads $\mathcal{R}$={ATGCTA, GCTAGC, TAGCAC, GCACAT, ACATGC} from the circular genome R = ATGCTAGCAC. (a) All MUSs $A_{\mathcal{R}}(w)$={AT, TG, CT, TA, AG, GCA, AC} and the corresponding paths in $\mathcal{R}$ annotated with shifts, U_{start}, and U_{end}. (b) The resulting vMUS-dBG($\mathcal{R}, A_{\mathcal{R}}(w)$).

The detail structure of the graph construction is described in the next section.

$$
\begin{aligned}
(\mathrm{AT}, \mathrm{TG}) &\mapsto \{(1, \varepsilon), (1, \varepsilon)\}, \\
(\mathrm{TG}, \mathrm{CT}) &\mapsto \{(2, \varepsilon)\}, \\
(\mathrm{CT}, \mathrm{TA}) &\mapsto \{(1, \varepsilon), (1, \varepsilon)\}, \\
(\mathrm{FIRST_NODE}, \mathrm{CT}) &\mapsto \{(1, G)\}, \\
(\mathrm{TA}, \mathrm{AG}) &\mapsto \{(1, \varepsilon), (1, \varepsilon)\}, \\
(\mathrm{AG}, \mathrm{LAST_NODE}) &\mapsto \{(1, C)\}, \\
(\mathrm{AG}, \mathrm{GCA}) &\mapsto \{(1, \varepsilon)\}, \\
(\mathrm{GCA}, \mathrm{AC}) &\mapsto \{(2, \varepsilon), (2, \varepsilon)\}, \\
(\mathrm{AC}, \mathrm{AT}) &\mapsto \{(2, \varepsilon), (2, \varepsilon)\}, \\
(\mathrm{TG}, \mathrm{LAST_NODE}) &\mapsto \{(1, C)\}.
\end{aligned}
$$

Fig. 4 This figure depicts the edge data representation of the assembly graph. This includes tuples of the form (δ, τ), where each tuple encodes the weight length and the tag substring. The ε indicate an empty string indicating an overlap between consecutive MUSs. The flow function is defined as follows: $C(AT, TG) = 2$, $C(CT, TA) = 2$, $C(AC, AT) = 2$, and so on. The annotation function is represented as: $\mathcal{I}(AT, TG) = \{(1, \varepsilon), (1, \varepsilon)\}$ and $\mathcal{I}(TG, CT) = \{(2, \varepsilon)\}$, among other possible pairs.

3.3 *v*MUS-dBG Construction

The assembly graph $G_{\mathcal{M}} = (U_{\mathcal{M}}, V_{\mathcal{M}}, f_{\mathcal{M}}, \mathcal{I}_{\mathcal{M}}, P(u)_{\mathcal{M}})$ is now constructed by processing each read $read_k$ and its corresponding MUS occurrences. The construction of the graph is accomplished by iterating through each input read $R_k \in \mathcal{R}$. For every read, all occurrences of the MUS from the set $A_R(w)$ are identified. These occurrences are then sorted by their starting index to create an ordered sequence, S_k. The graph is generated by establishing directed edges between consecutive MUSs in this sequence, along with edges connecting the source node to the first MUS and from the last MUS to the sink node.

3.3.1 *v*MUS-dBG Construction Rules:

The count function for the edge multiplicities (flows), $f_{\mathcal{M}}$, and the collection of edge data lists, $\mathcal{I}_{\mathcal{M}}$, are initialized to zero and empty, respectively, for all possible node pairs (u, v). Similarly, the data structure $P(u)$ is initialized for all nodes u to track maximal prefix and suffix information.

- **Prefix Segment**: For a given read R_k, if its first identified MUS, w_1, starts at index $s_1 > 1$, a directed edge is established from the source node, U_{start}, to w_1. The flow of this edge, $f(U_{\text{start}}, w_1)$, is incremented. The edge data for this instance is defined by a weight $\delta = s_1 - 1$ and a tag $\tau = R_k[1 : s_1 - 1]$, and these are appended to $\mathcal{I}(U_{\text{start}}, w_1)$. Additionally, the maximal prefix information in $P(w_1)$ and $P(U_{\text{start}})$ is updated if the current prefix length is greater than the previously recorded length.
- **Internal Segments**: For each pair of consecutive MUS, $(w_i, [s_i, e_i])$ and $(w_{i+1}, [s_{i+1}, e_{i+1}])$, in the sorted sequence of MUS for a read, a directed edge is established from w_i to w_{i+1}. The flow $f(w_i, w_{i+1})$ is incremented. The edge data for

this instance is defined by a weight $\delta = s_{i+1} - s_i$ and a tag $\tau = R_k[e_i + 1 : s_{i+1} - 1]$, which are appended to $\mathcal{I}(w_i, w_{i+1})$.

- **Suffix Segment**: For a given read R_k, if its last identified MUS, w_p, ends at index $e_p < |R_k|$, a directed edge is established from w_p to the sink node, U_{end}. The flow $f(w_p, U_{\text{end}})$ is incremented. The edge data for this instance is defined by a weight $\delta = |R_k| - e_p$ and a tag $\tau = R_k$, which is appended to $\mathcal{I}(w_p, U_{\text{end}})$. The maximal suffix information in $P(w_p)$ and $P(U_{\text{end}})$ is updated if the current suffix length is greater than the previously recorded length.

The algorithm returns the populated graph represented by the tuple $(f, \mathcal{I}, P)$, where f is the count function for edge multiplicities, $\mathcal{I}$ is the collection of edge data, and P is the terminal data for each node, which fully describes the constructed assembly graph. This is depicted in Algorithm 1 and Figure 2 (a) and (b).

4 Graph Simplification and Traversal

4.0.1 Degree Calculation and Branching Logic

A key innovation of this algorithm is its redefinition of what constitutes a "branching/junction" point within a graph. Specifically, the definition of a branching node (junction) is determined by the count of distinct edges, considering both the total number of incoming and outgoing connections. In other words, if $\deg^+(u) > 1$, there are multiple ways to depart from the node u, even if all paths lead to the same node v. Therefore, a choice must be made regarding which specific edge instance to traverse, and this choice point is what constitutes a branch. Given the directed multigraph model $G_{\mathcal{M}} = (U, V, f, \mathcal{I}, P(u))$, the set of `edge instances` associated with an edge (u, v) is defined as:

$$\mathcal{I}(u, v) = \{(\delta_i, \tau_i)\}_{i=1}^{m},$$

where each pair $\delta_i \in \mathbb{Z}_{\geq 0}$ is the `weight` and $\tau_i \in \Sigma^*$ is the `tag` as defined in Equation 3.2. We therefore, distinguishes between two types of graph degrees:

- `Total Degrees (Flow)`: This represents the sum of all edge instances entering or leaving a node, including parallel edges. It serves as a measure of overall read support.
- `Unique Degrees`: This counts the number of `distinct nodes` connected to a given node, disregarding multiple parallel edges between the same two nodes and instead focusing on the unique paths available.

Formally, for any node $u \in U$, the total in-degree is given as $f^-(v) = \Sigma_{v \in U} f(u, v)$ and the total out-degree is $f^+(v) = \Sigma_{w \in U} f(v, w)$. Moreover, for any node $u \in U$, the unique in-degree is $\deg^-(v) = |\{v \in U | f(u, v) > 0\}|$ counting distinct predecessors, and the unique out-degree $\deg^+(v) = |\{w \in U | f(v, w) > 0\}|$ counting distinct successors. These considerations lead to the following definition:

Definition 2 (Junction Node) A node $u \in U$ is a `junction` if the following holds: $u \in \{U_{\text{start}}, U_{\text{end}}\}$, $\deg^-(u) \neq 1$, or $\deg^+(u) \neq 1$.

Algorithm 1 Assembly Graph Construction From MUS

Initialization:
Initialize flow matrix $f(u, v) \leftarrow 0$ for all $u, v \in U$.
Initialize edge data lists $\mathcal{I}(u, v) \leftarrow \emptyset$ for all $u, v \in U$.
Initialize node terminal data structure $P(u)$ with empty tags and zero lengths for all $u \in U$.

for each read $R_k \in \mathcal{R}$ **do**
  Let $\mathcal{O}_k$ be the multiset of all MUS occurrences in R_k, as $(w, [s, e])$.
  Sort $\mathcal{O}_k$ by s to obtain $\mathcal{S}_k = \big((w_1, [s_1, e_1]), \ldots, (w_p, [s_p, e_p])\big)$.
  if $\mathcal{S}_k = \emptyset$ **then**
    continue to next read.
  end if
  ▷ Prefix Segment
  if $s_1 > 1$ **then**
    $f(U_{\text{start}}, w_1) \leftarrow f(U_{\text{start}}, w_1) + 1$
    Append $(\delta = s_1 - 1, \tau = R_k[1 : s_1 - 1])$ to $\mathcal{I}(U_{\text{start}}, w_1)$
    Update prefix data for w_1 and U_{start} if needed
  end if
  ▷ Internal Segments
  for $i = 1$ to $p - 1$ **do**
    $f(w_i, w_{i+1}) \leftarrow f(w_i, w_{i+1}) + 1$
    Append $(\delta = s_{i+1} - s_i, \tau = R_k[e_i + 1 : s_{i+1} - 1])$ to $\mathcal{I}(w_i, w_{i+1})$
  end for
  ▷ Suffix Segment
  if $e_p < |R_k|$ **then**
    $f(w_p, U_{\text{end}}) \leftarrow f(w_p, U_{\text{end}}) + 1$
    Append $(\delta = |R_k| - e_p, \tau = R_k)$ to $D(w_p, U_{\text{end}})$
    Update suffix data for w_p and U_{end} if needed
  end if
end for

Output: Return $(f, \mathcal{I}, P)$

Definition 3 (Source/Sink:) A node $u \in U$ is source node if $\deg^-(u) = 0$ and $\deg^+(u) > 0$. A node $v \in \mathcal{M}$ is sink node if $\deg^+(v) = 0$ and $\deg^-(v) > 0$.

Proposition 1 (Best Edge Instance Selection:) *Given an edge (u, v) and a list of its associated data tuples $\mathcal{I}(u, v)$, the best instance (δ, τ) is selected based on the following rules:*

- *Terminal Edges: If $u = U_{start}$ or $v = U_{end}$, the best instance is the one with the maximum length of τ. In case of a tie, the instance with the smallest weight δ is chosen.*
- *Internal Edges: For all other edges, the best instance is the one with the smallest weight δ. In case of a tie, the instance with the maximum length of τ is chosen.*

This is given mathematically as:

$$(\delta^*, \tau^*) = \begin{cases} \arg\max\limits_{(\delta,\tau)\in\mathcal{T}(u,v)} (|\tau|, -\delta), & \text{if } u = U_{\text{start}} \text{ or } u = U_{\text{end}}, \\ \arg\min\limits_{(\delta,\tau)\in\mathcal{T}(u,v)} (\delta, -|\tau|), & \text{otherwise.} \end{cases}$$

4.1 Path Extension and Sequence Construction

4.1.1 Graph Simplification and Unitig Extraction

After the assembly graph construction, we simplify $G_{\mathcal{M}}$ by removing trivial linear paths. We call a node $u \in U \setminus \{U_{\text{start}}, U_{\text{end}}\}$ strictly linear if it has exactly one unique predecessor and successor, and the total flow of 1. That is, $\deg^-(v) = 1 \wedge \deg^+(v) = 1 \wedge f^-(v) = 1 \wedge f^+(v)$.

We now describe the path collapsing operation. Let $P = (u, v, w)$ be a path of nodes in the $G_{\mathcal{M}}$ where v is a strictly linear node. Let $e_1 = (u, v)$ be the unique incoming edge and $e_2 = (v, w)$ be the unique outgoing edge. Let $\mathcal{I}(e_1) = (\delta_1, \tau_1)$ and $\mathcal{I}(e_2) = (\delta_2, \tau_2)$, then the path P is collapsed by modifying $G_{\mathcal{M}}$. This is achieved by removing v, e_1, and e_2 from the path and adding a new edge $e_{12} = (u, w)$. The new edge now has $f(e_{12}) = 1$ and $\mathcal{I}(e_{12}) = (\delta_1 + \delta_2, \tau_1 \oplus v \oplus \tau_2)$, where $\oplus$ denotes string concatenation. From this, we now obtain a simplified graph given as $G_S = (U, V, f, \mathcal{I})$ after an iterative application of the graph collapsing operation until no strictly linear nodes remain.

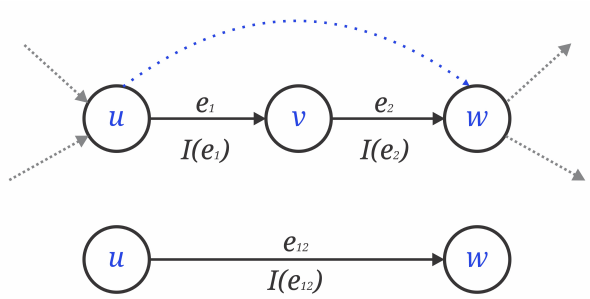


Fig. 5 Graph simplification where node v and edges e_1, e_2 removed and replaced with a new edge $e_{12} = (u, w)$.

4.1.2 Unitig Extraction

Unitigs are then extracted from G_S by decomposing it into a set of maximal non-branching paths. Thus, in G_S, a node $v \in U$ is a junction if $\deg^-(v) \neq 1$, or $\deg^+(v) \neq 1$.

Definition 4 (Maximal Non-Branching Paths, MNBP) A MNBP is a path $P = (v_0, v_1, \ldots, v_k)$ in $G_{\mathcal{S}}$ such that (1) v_0 is a source node, or junction, (2) v_k is a sink, or

junction, (3) All internal nodes $v_1, \ldots, v_{k-1}$ are non-branching, and (4) Each edge instance $(\delta, \tau) \in \mathcal{I}(u, v)$ is used at most once across all extracted paths. Unitigs correspond to MNBP in the assembly graph.

The sequence $S(P)$ is now constructed by concatenating the sequence of each node with the sequence tag from the preceding edge in G_S. The reconstruction is built as follows given the path $P = (v_0, v_1, \ldots, v_k)$ (Algorithm 2). Let $\text{seq}(v)$ denote the nucleotide sequence of node v, and let the edge $e_i = (v_i, v_{i+1})$ be associated with a distance δ_i and a weight tag τ_i. The reconstructed sequence $S(P)$ is defined as follows:

Initial Sequence (Base Case)

The starting sequence S_0 depends on the nature of the source node v_0:

$$S_0 = \begin{cases} \tau_0 \cdot \text{seq}(v_1) & \text{if } v_0 = U_{\text{start}} \\ \text{seq}(v_0) & \text{if } v_0 \neq U_{\text{start}} \end{cases} \tag{3}$$

where $\cdot$ denotes the string concatenation operator.

Recursive Path Extension

For each subsequent edge (v_i, v_{i+1}) where $v_{i+1} \neq U_{\text{end}}$, we define the overlap ϕ_i as:

$$\phi_i = |\text{seq}(v_i)| - \delta_i \tag{4}$$

The sequence is extended iteratively such that $S_{i+1} = S_i \cdot \Delta(v_i, v_{i+1})$, where the transition component $\Delta(v_i, v_{i+1})$ is determined by the spatial relationship between the MUS anchors. Let $L_i = |\text{seq}(v_{i+1})| - \phi_i$ be the length of the non-overlapping suffix. The transition component is defined as:

$$\Delta(v_i, v_{i+1}) = \begin{cases} \text{suffix}(\text{seq}(v_{i+1}), L_i) & \text{if } \phi_i > 0 \\ \text{seq}(v_{i+1}) & \text{if } \phi_i = 0 \\ \tau_i \cdot \text{seq}(v_{i+1}) & \text{if } \phi_i < 0 \end{cases} \tag{5}$$

General Form

For a path where $v_0 \neq U_{\text{start}}$ and $v_k \neq U_{\text{end}}$, the complete genomic sequence is represented by the concatenation of the initial node and all subsequent transition components:

$$S(P) = \text{seq}(v_0) \cdot \prod_{i=0}^{k-1} \Delta(v_i, v_{i+1}) \tag{6}$$

Definition 5 (Final Contig Assembly (Algorithm 2)) For each path $P_U = (U_1, \ldots, U_k) \in P_{\text{final}}$, reconstruct the base path P_B by concatenating internal paths of the original unitigs. Then, apply the sequence reconstruction function

$$S(P_B, \lambda_B)$$

Algorithm 2 Greedy Flow-Based Path Traversal

Require: Unitig graph $G_U = (V_U, E_U)$, residual edge flow function $f'_U : E_U \to \mathbb{Z}_{\geq 0}$, and starting nodes $V_{start} \subseteq V_U$
Ensure: Set of reconstructed contig paths $\mathcal{P}_{final}$

$\mathcal{P}_{final} \leftarrow \emptyset$
$V_{seeds} \leftarrow \{u \in V_U \mid u \text{ is a source or branching node}\}$
for all $u_{start} \in V_{seeds}$ **do**
  for all $u_{next} \in \{v \mid (u_{start}, v) \in E_U \text{ and } f'_U(u_{start}, v) > 0\}$ **do**
    Initialize path $P \leftarrow (u_{start}, u_{next})$
    $u_{curr} \leftarrow u_{next}$
    $active \leftarrow$ true
    while $active$ **do**
      $S \leftarrow \{v \in V_U \mid (u_{curr}, v) \in E_U \text{ and } f'_U(u_{curr}, v) > 0\}$
      if $|S| = 0$ **then** ▷ Reached a dead end/tip
        $active \leftarrow$ false
      else if $|S| = 1$ **then** ▷ Linear extension
        $u_{next} \leftarrow$ the unique element in S
        $P \leftarrow P \cdot (u_{next})$ ▷ Concatenate node to path
        $f'_U(u_{curr}, u_{next}) \leftarrow f'_U(u_{curr}, u_{next}) - 1$
        $u_{curr} \leftarrow u_{next}$
      else ▷ Branching node resolution
        $f_{max} \leftarrow \max_{v \in S} f'_U(u_{curr}, v)$
        $S_{best} \leftarrow \{v \in S \mid f'_U(u_{curr}, v) = f_{max}\}$
        if $|S_{best}| = 1$ **then** ▷ Clear majority flow exists
          $u_{best} \leftarrow$ the unique element in S_{best}
          $P \leftarrow P \cdot (u_{best})$
          $f'_U(u_{curr}, u_{best}) \leftarrow f'_U(u_{curr}, u_{best}) - 1$
          $u_{curr} \leftarrow u_{best}$
        else ▷ Ambiguous flow (tie); terminate path
          $active \leftarrow$ false
        end if
      end if
    end while
    $\mathcal{P}_{final} \leftarrow \mathcal{P}_{final} \cup \{P\}$
  end for
end for
return $\mathcal{P}_{final}$

using original edge labels λ_B to generate the final contiguous sequence.

5 Experiments and Results

We built a de Bruijn graph using variable-length Minimum Unique Substrings (vMUS-dBG). The analysis was performed on a PowerEdge M620 (VRTX) server equipped

with 640.0 GB of RAM and powered by two Intel(R) Xeon(R) CPU E5-2670 v2 processors running at 2.50 GHz, and running Ubuntu Server 22.04.5 LTS. The implementation is written in both C++ and Python, leveraging the advantages of each language. The source code for vMUS-dBG can be accessed at the provided link git@github.com:fandrew19/vMUS-dBG-Assembler.git.

We compared *v*MUS-dBG with BCalm2 [16] for graph construction with the default configuration DKSIZE_LIST="32 64 96 128 160 192 224 256 320". BCalm2 is a highly parallel implementation of an algorithm that first counts all the k-mers, then analyzes the overlaps between these k-mers to construct the Compressed De Bruijn Graph (CdBG). To ensure a fair comparison, the program is executed on a single core and with default parameters. We used HiFi data from *E. coli bacteria K12* with run identification SRR10971019. We randomly downsampled the reads to 24x coverage. We arbitrarily varied k-mer sizes (k values from 21 to 251) in BCalm2 to examine how the graph changes with varying k-mer sizes. We evaluated the assembly outputs from each run using QUAST [17]. Figures 6, 7, 8, and 9 depict the structural perturbations with respect to varying k, compared to our variable length construction.

Computationally, the fixed-k approach is more efficient than *v*MUS-dBG, using less memory and processing time across all k values. This efficiency stems from our custom implementation of the suffix tree which required significant amount of memory. Structurally, *v*MUS-dBG generates significantly fewer contigs (172), approximately 12 to 28 times fewer than any fixed-k assembly (2078-4868). However, it produces larger contigs (36,126 bp), which significantly outperform the 5,819 bp seen in the fixed-k-mer values as illustrated in Figure 6.

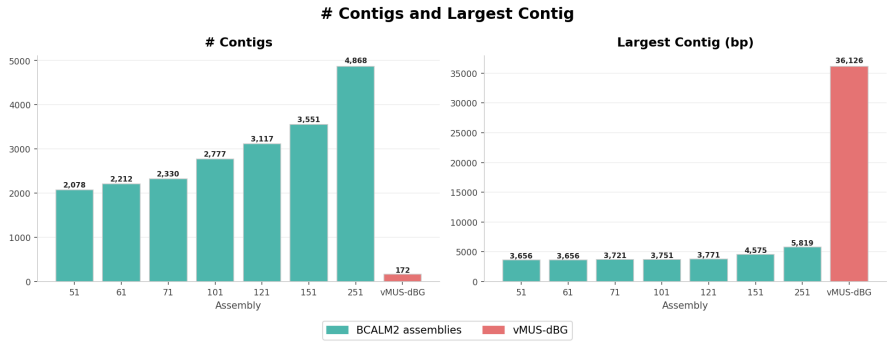


Fig. 6 Assembly contig counts and the largest contiguous sequence observed across the fixed-k (k=51-251 bp) approach and *v*MUS-dBG.

In terms of contiguity, we observe consistently short contigs across all k values, with N50 values ranging from 753 bp (k=51) to 972 bp (k=251). In contrast, the N50 value for *v*MUS-dBG is notably high at 15,420 bp, demonstrating adaptability approximately 16-20 times better than that of any fixed-k assembly. Furthermore, *v*MUS-dBG achieves a higher NA50 (13,317 bp) than the fixed k-mer approach across all tested k values (753 bp-972 bp). This is illustrated in Figure 7.

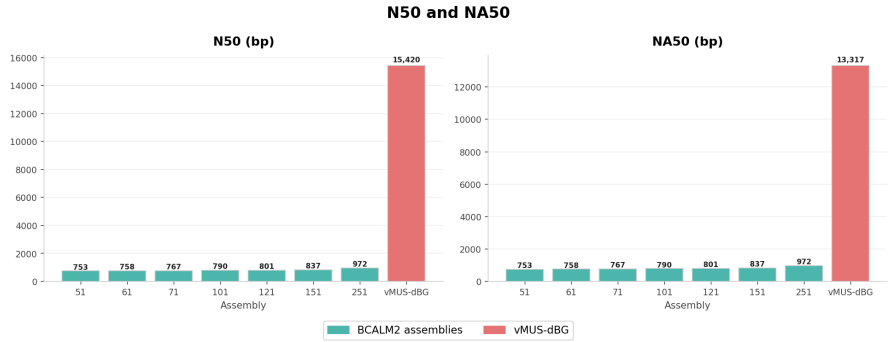


Fig. 7 Assembly contiguity regarding N50 (raw contig lengths) and NA50 (aligned blocks) was higher for the *v*MUS-dBG method compared to the fixed-k approach at all k-values.

The genome fraction, which measures the extent of coverage of the reference genome, ranges from 34% (k=51) to 85% (k=251), with coverage steadily increasing as k increases. Nevertheless, none of the fixed-k assemblies cover the entire genome; even at k=251, there remains 15% uncovered, primarily due to graph fragmentation at higher k values. On the other hand, *v*MUS-dBG provides only 27.6% coverage, which is lower than that of k=51.

Additionally, the duplication ratio presents a complementary perspective. All fixed-k assemblies maintain a duplication ratio close to 1.0 (ranging from 1.004 to 1.142), indicating virtually no redundancy. The slight increase at k=251 (1.142) suggests that the assembler begins to produce overlapping contigs as the graph structure becomes more complex at higher k values. In contrast, *v*MUS-dBG has a duplication ratio of 2.0, meaning that, on average, each assembled base is represented twice. This aligns with the approach's unique-repeat awareness (Figure 8).

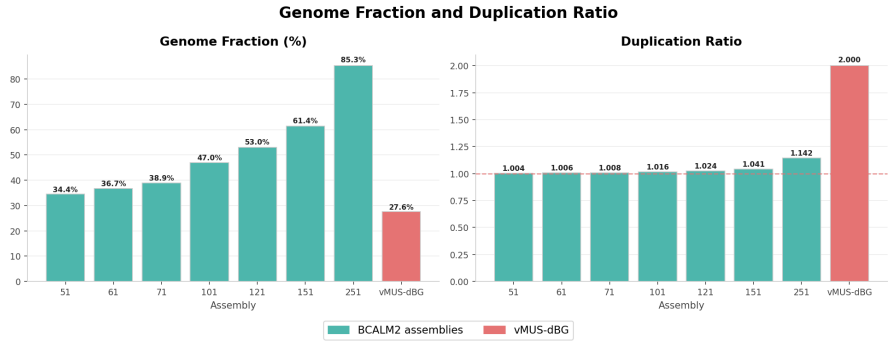


Fig. 8 In terms of assembly completeness, the fixed-k approach outperforms *v*MUS-dBG, while *v*MUS-dBG shows a higher duplication ratio of 2.0.

The total assembled length, which includes the reference length of 4,641,652 bp for context, reveals a key trade-off in assembly philosophy. The fixed-k assemblies range

from 1.6 Mbp (for k=51) to 4.5 Mbp (for k=251), with k=251 coming closest to the reference length. However, even at k=251, the assembled bases comprise numerous short, fragmented contigs with minimal errors. In contrast, vMUS-dBG assembles 2.57 Mbp, falling between k=101 and k=121 in raw length, but it delivers much better contiguity and contig quality (Figure 9).

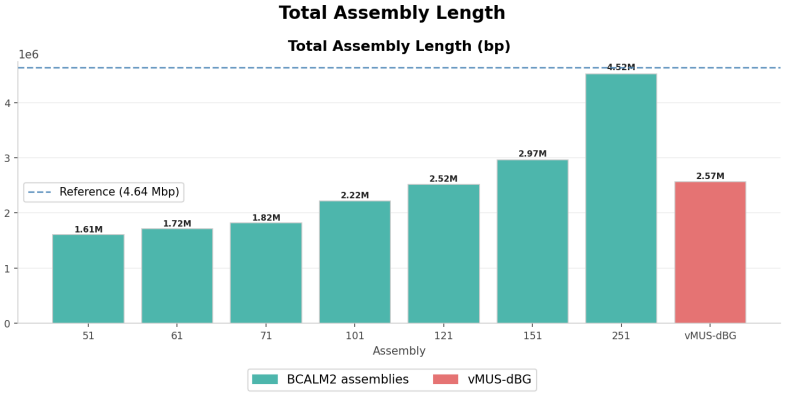


Fig. 9 Total assembled sequences. Total assembled sequences showed that each k had different assembled lengths, ranging from 1.6 Mb to 4.5 Mb, compared to the vMUS-dBG, which measured 2.5 Mb.

5.1 Discussion

This analysis highlights a fundamental trade-off between the breadth of coverage and the continuity of assembly. The fixed k-mer approach is effective at capturing a large portion of the genome but struggle with resolving repeats, leading to a fragmented assembly consisting of very short contigs. In contrast, the vMUS-dBG method effectively leverages variable-length information to overcome the ambiguities that fixed-k methods face. By employing minimum unique substrings instead of fixed k-mers, vMUS-dBG produces contigs that are significantly longer. However, the high duplication ratio (2.0) and discrepancy between N50 and NA50 indicate that vMUS-dBG preserves repeat ambiguity as alternative paths rather than collapsing them. This is by design; the graph encodes uncertainty rather than forcing incorrect resolution.

5.2 Conclusion

We introduced a new method, the variable-length Minimum Unique Substring-based de Bruijn graph (vMUS-dBG), that integrates sequence uniqueness and repeat boundaries directly into the graph's structure. By embedding these characteristics into the topology of the graph, the vMUS-dBG reconceptualizes repeat resolution as an inherent property of the graph, rather than a parameter-sensitive outcome that depends on a fixed k-mer choice.

Evaluation using HiFi sequencing data from *E. coli K-12* demonstrates that the vMUS-dBG effectively prevents catastrophic repeat collapse and excessive fragmentation often observed in fixed-k graphs. These findings establish Minimum Unique Substrings as both a biologically relevant and algorithmically feasible basis for repeat-aware genome assembly.

The method is expected to benefit significantly from higher sequencing depth, as increased coverage will produce more overlaps between MUS and enhance graph connectivity. Additionally, it will improve with rigorous error correction since single-base errors can disrupt uniqueness and lead to false MUS outcomes. In this study, our scalability was limited by the memory requirements of constructing generalized suffix trees and extracting MUS in our custom implementation, which restricted experimentation with higher-coverage datasets on standard hardware.

Switching from a suffix tree to a suffix-array-based framework substantially reduces memory overhead and enables practical deployment on typical computing systems, thereby enhancing scalability without compromising the method's theoretical foundation.

6 Conflicts of Interest

The authors declare no competing interests.

Funding

This work was supported by the U.S. National Institutes of Health (NIH) Common Fund under grant numbers U24HG006941 (to SPS) and 1U2RTW010679 (to AFA and SPS). The content of this publication solely reflects the views of the authors and does not necessarily represent the official views of the NIH. The funding body had no role in the study design, data collection and analysis, interpretation of results, or manuscript preparation.

References

[1] Myers, E.W.: The fragment assembly string graph. Bioinformatics **21**(suppl_2), 79–85 (2005)

[2] Idury, R.M., Waterman, M.S.: A new algorithm for dna sequence assembly. Journal of computational biology **2**(2), 291–306 (1995)

[3] Pevzner, P.A., Tang, H., Waterman, M.S.: An eulerian path approach to dna fragment assembly. Proceedings of the national academy of sciences **98**(17), 9748–9753 (2001)

[4] Compeau, P.E., Pevzner, P.A., Tesler, G.: How to apply de bruijn graphs to genome assembly. Nature biotechnology **29**(11), 987–991 (2011)

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

[6] Peng, Y., Leung, H.C., Yiu, S.-M., Chin, F.Y.: Idba–a practical iterative de bruijn graph de novo assembler. In: Annual International Conference on Research in Computational Molecular Biology, pp. 426–440 (2010). Springer

[7] Bankevich, A., Nurk, S., Antipov, D., Gurevich, A.A., Dvorkin, M., Kulikov, A.S., Lesin, V.M., Nikolenko, S.I., Pham, S., Prjibelski, A.D., *et al.*: Spades: a new genome assembly algorithm and its applications to single-cell sequencing. Journal of computational biology **19**(5), 455–477 (2012)

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

[9] Lin, Y., Pevzner, P.A.: Manifold de bruijn graphs. In: International Workshop on Algorithms in Bioinformatics, pp. 296–310 (2014). Springer

[10] Boucher, C., Bowe, A., Gagie, T., Puglisi, S.J., Sadakane, K.: Variable-order de bruijn graphs. In: 2015 Data Compression Conference, pp. 383–392 (2015). IEEE

[11] Belazzougui, D., Gagie, T., Mäkinen, V., Previtali, M., Puglisi, S.J.: Bidirectional variable-order de bruijn graphs. In: LATIN 2016: Theoretical Informatics, pp. 164–178. Springer, ??? (2016)

[12] Díaz-Domínguez, D., Onodera, T., Puglisi, S.J., Salmela, L.: Genome assembly with variable order de bruijn graphs. bioRxiv, 2022–09 (2022)

[13] Li, H., Durbin, R.: Genome assembly in the telomere-to-telomere era. Nature Reviews Genetics **25**(9), 658–670 (2024)

[14] Ilie, L., Smyth, W.F.: Minimum unique substrings and maximum repeats. Fundamenta Informaticae **110**(1-4), 183–195 (2011)

[15] Mieno, T., Kuhara, Y., Akagi, T., Fujishige, Y., Nakashima, Y., Inenaga, S., Bannai, H., Takeda, M.: Minimal unique substrings and minimal absent words in a sliding window. In: International Conference on Current Trends in Theory and Practice of Informatics, pp. 148–160 (2020). Springer

[16] Chikhi, R., Limasset, A., Medvedev, P.: Compacting de bruijn graphs from sequencing data quickly and in low memory. Bioinformatics **32**(12), 201–208 (2016)

[17] Gurevich, A., Saveliev, V., Vyahhi, N., Tesler, G.: Quast: quality assessment tool for genome assemblies. Bioinformatics **29**(8), 1072–1075 (2013)