Assembling Long Accurate Reads Using de Bruijn Graphs

Anton Bankevich[1,3], Andrey Bzikadze[2], Mikhail Kolmogorov[1], Pavel A. Pevzner[1]
[1]Department of Computer Science and Engineering
[2]Graduate Program in Bioinformatics and Systems Biology
University of California at San Diego
[3]abankevich@eng.ucsd.edu

Abstract

Although the de Bruijn graphs represent the basis of many genome assemblers, it remains unclear how to construct these graphs for large genomes and large *k*-mer sizes. This algorithmic challenge has become particularly important with the emergence of long and accurate high-fidelity (HiFi) reads that were recently utilized to generate a semi-manual telomere-to-telomere assembly of the human genome using the alternative string graph assembly approach. To enable fully automated high-quality HiFi assemblies of various genomes, we developed an efficient jumboDB algorithm for constructing the de Bruijn graph for large genomes and large *k*-mer sizes and the LJA genome assembler that error-corrects HiFi reads and uses jumboDB to construct the de Bruijn graph on the error-corrected reads. Since the de Bruijn graph constructed for a fixed *k*-mer size is typically either too tangled or too fragmented, LJA uses a new concept of a multiplex de Bruijn graph with varying *k*-mer sizes. We demonstrate that LJA produces contiguous assemblies of complex repetitive regions in genomes including automated assemblies of various highly-repetitive human centromeres.

Introduction

The emergence of long and accurate HiFi reads, generated using the *consensus circular sequencing* technology (Wenger et al., 2019), opened a possibility to generate accurate and contiguous assemblies of large genomes (Nurk et al., 2020, Cheng et al., 2020). At the same time, it raised the challenge of constructing the *de Bruijn graphs* (Compeau et al., 2011) for large genomes and large *k*-mer sizes (e.g., comparable with the length of HiFi reads). Indeed, similarly to assembling short and accurate reads, the de Bruijn graph-based approaches have the potential to improve assemblies of *any* accurate reads. However, although the de Bruijn graphs represent the algorithmic engine of the most popular genome assemblers for short and accurate reads (Zerbino and Birney, 2008, Simpson et al., 2009, Peng et al., 2010, Bankevich et al., 2012), the existing HiFi assemblers HiCanu (Nurk et al., 2020) and Hifiasm (Cheng et al., 2020) are based on the alternative *string graph* approach (Myers, 2005).

Since HiFi reads are not unlike accurate Illumina reads with respect to repeat resolution (albeit at a different scale of repeat lengths), the de Bruijn graph approach is likely to work well for their assembly (see Lin et al. 2014 for a comparison of the de Bruijn graphs and the string graphs). Application of this approach to long HiFi reads requires either constructing the de Bruijn graph with large *k*-mer size or alternatively, using the de Bruijn graph with small *k*-mer-size for follow-up repeat resolution by threading long reads through this graph (Kolmogorov et al., 2019, 2020). However, it remains unclear how to (i) construct the de Bruijn graphs for large genomes and large *k*-mer sizes, (ii) error-correct HiFi reads using these graphs so that they become nearly error-free, and (iii) utilize the entire length of HiFi reads for resolving repeats that are longer than the *k*-mer size selected for constructing the de Bruijn graph. We introduce the de Bruijn graph-based *La Jolla Assembler* (*LJA*) that addresses these algorithmic challenges..

The existing genome assemblers are not designed for constructing the de Bruijn graphs with large *k*-mer sizes since their memory/time requirements become prohibitive when the *k*-mer size becomes large, e.g., simply storing all 5000-mers of the human genome requires ≅4 Tb of memory. For example, the SPAdes assembler (Bankevich et al., 2012), designed for assembling Illumina reads with the read-length below 300 bp, is typically used with the *k*-mer size below 100. Applying SPAdes to longer reads and significantly increasing the *k*-mer size beyond *k*=500 leads to time/memory bottlenecks. To reduce the memory footprint, some assembly algorithms avoid explicitly storing all *k*-mers: e.g., SPAdes (Bankevich et al., 2012) constructs a *perfect hash map* (Fredman et al., 1984) of all *k*-mers in reads, while MegaHit (Li et al., 2015) constructs the *Burrows-Wheeler Transform* (Burrows and Wheeler, 1984) of all reads. However, even with these improvements, the memory footprint remains large, not to mention that the running time remains proportional to the *k*-mer size.

Two approaches construct coarse versions of the de Bruijn graph with smaller time/memory requirements: the *repeat graph* approach (Pevzner et al., 2004) and the *sparse de Bruijn graph* approach (Ye et al., 2012). Recently, Kolmogorov et al., 2020 modified the Flye assembler for constructing the repeat graph of HiFi reads (in a metagenomic context), while Rautiainen and

Marschall, 2020 developed the MBG tool for assembling HiFi reads into a sparse de Bruijn graph. However, these graphs are less accurate than the de Bruijn graph with respect to representing the repeat structures, thus limiting their capabilities in assembling the most repetitive regions such as centromeres. Our jumboDB algorithm for constructing the de Bruijn graph for large *k*-mer sizes combines four algorithmic ideas: the *Bloom filter* (Bloom, 1970), the *rolling hash* (Karp and Rabin, 1987), the sparse de Bruijn graph (Ye et al., 2012), and the *disjointig* generation (Kolmogorov et al., 2019). Although each of these ideas was used in previous genome assembly studies (e.g., MBG uses the rolling hash to construct the sparse de Bruijn graph), jumboDB is the first approach where they are all combined for constructing the de Bruijn graphs. LJA launches jumboDB to construct the de Bruijn graph of HiFi reads, uses this graph to correct errors in HiFi reads, generates a much simpler graph of the error-corrected HiFi reads, and transform it into the *multiplex de Bruijn graphs* with varying *k*-mer sizes to take advantage of the full length of HiFi reads.

The paper is organized as follows. Section 1 introduces the concepts of the *compressed* de *Bruijn graph* (where each non-branching path is compressed into a single edge), sparse de Bruijn graph, and the Bloom filter. The traditional assembly approach constructs the de Bruijn graph *DB*(*Reads,k*) first and transforms it into the compressed de Bruijn graph *CDB*(*Reads,k*) afterward. Since this approach is impractical for large genomes and large *k*-mer sizes, jumboDB constructs *CDB*(*Reads,k*) without constructing *DB*(*Reads,k*). We first address a simpler problem of constructing the de Bruijn graph of a genome *CDB*(*Genome,k*) rather than the de Bruijn graph of reads *CDB*(*Reads,k*) - section 2 describes a modified version of the Minkin-Pham-Medvedev algorithm (Minkin et al., 2017) for constructing *CDB*(*Genome,k*) that serves as a stepping stone for constructing *CDB*(*Reads,k*). Since this algorithm becomes prohibitively slow in the case of large *k*-mer sizes, jumboDB utilizes the rolling hash (Karp and Rabin, 1987) to adapt it for large *k*-mer sizes. Since constructing *CDB*(*Reads,k*) faces time/memory bottlenecks, jumboDB first assembles reads into disjointigs that represent random walks through the (unknown) de Bruijn graph of reads and further generates the compressed de Bruijn graph of disjointigs. Although switching from reads to error-prone disjointigs may appear reckless, it is an important step for addressing the time/memory bottleneck since a disjointig-set *Disjointigs* result in a much lower coverage of a genome than the read-set *Reads* while resulting in the identical compressed de Bruijn graph, i.e., *CDB*(*Disjointigs,k*)=*CDB*(*Reads,k*). Section 3 explains how to achieve this goal and how to construct disjointigs using the sparse de Bruijn graph of reads. Section 4 summarizes various steps of jumboDB. Section 5 provides information about its running time and memory footprint of jumboDB. Section 6 describes a transDB algorithm for transforming *CDB*(*Reads,k*) into *CDB*(*Reads,k*+1) that serves as a stepping stone toward constructing the multiplex de Bruijn graph. Finally, section 7 describes an algorithm for constructing the multiplex de Bruijn graph and illustrates its applications to assembling human centromeres. The code of jumboDB is available at https://github.com/AntonBankevich/jumboDB.

Section 1: Compressed de Bruijn graph, sparse de Bruijn graph, and the Bloom filter

De Bruijn graphs. We define the *k-prefix* (*k-suffix*) of a string as the first (last) *k*-mer in this string. Given a string-set *Genome* (each string in this set is referred to as a chromosome) and an integer *k*, the de Bruijn multigraph *multiDB*(*Genome,k*) is defined as follows. Each *k*-mer

occurring in *Genome* corresponds to a vertex in the de Bruijn graph (identical *k*-mers correspond to the same vertex) and each (*k*+1)-mer corresponds to an edge connecting its *k*-prefix with its *k*-suffix (identical (*k*+1)-mers form parallel edges in the graph). The de Bruijn graph *DB*(*Genome,k*) is obtained from the de Bruijn multigraph *multiDB*(*Genome,k*) by substituting each set of parallel edges connecting two vertices with a single edge (the *multiplicity* of this edge is defined as the number of such parallel edges). Given a path *P* formed by edges $e_1,e_2,\ldots,e_n$ in *DB*(*Genome,k*), its *path-label* *label*(*P*) is defined as *label*(e_1)**lastSymbol*(e_2)*…**lastSymbol*(e_n), where *lastSymbol*(*e*) stands for the last symbol of *label*(*e*) and *x*y* stands for the concatenate of strings *x* and *y*. We say that a path *P spells label(P)*. A string-set *Genome* corresponds to a path-set that traverses each edge in *multiDB*(*Genome,k*) exactly once (and each edge in *DB*(*Genome,k*) at least once) and spells *Genome*. We refer to this path-set in *DB*(*Genome,k*) as the *genome traversal*.

The goal of genome assembly is to reconstruct the genome from its error-prone substrings referred to as *reads*. Given a read-set *Reads*, we construct the de Bruijn graph *DB*(*Reads,k*) (by assuming that each read is a mini-chromosome) and compute the *coverage* of an edge *e* in this graph as the number of times *label*(*e*) occurs in *Reads*. We say that a read-set *Reads* is *full* with respect to *Genome* if each (*k*+1)-mer from *Genome* appears in a read from *Reads*. For a full set of error-free reads, the graph *DB*(*Reads,k*) coincides with *DB*(*Genome,k*). In the case of error-prone reads, the traditional approach to genome assembly is to construct the graph *DB*(*Reads,k*) and further modify it with the goal of approximating the graph *DB*(*Genome,k*).

Compressed de Bruijn graphs. A vertex with the indegree *N* and the outdegree *M* is referred to as *N-in-M-out vertex*. A vertex is classified as *complex* if both its indegree and outdegree exceed 1, and *simple*, otherwise. A vertex is classified as *non-branching* if it is a 1-in-1-out vertex, and as a *junction*, otherwise. A junction is classified as a *dead-end* if it has no incoming or no outgoing edges, and a *cross-road,* otherwise. We refer to the set of all junctions in the graph *DB*(*Genome,k*) as *Junctions*(*Genome,k*). A path between junctions is classified as *non-branching* if all its intermediate vertices are non-branching.

Given a non-branching path *P* between junctions *v* and *w* in a graph, its *compression* results in substituting this path by a single edge (*v,w*) labeled by *label*(*P*). Compressing all non-branching paths in *DB*(*Genome*,*k*) results in the compressed de Bruijn graph *CDB*(*Genome*,*k*) (the graph *CDB*(*Reads*,*k*) is defined similarly). The compressed de Bruijn graph is more compact than the de Bruijn graph since it does not require storing all *k*-mers - the total length of all labels in *CDB*(*Genome*,*k*) is typically *k* times smaller than the total length of all labels in *DB*(*Genome*,*k*). The *coverage* of an edge in *CDB*(*Reads*, *k*) is defined as the average coverage of all edges in the non-branching path that was compressed into this edge.

A *subpartition* of an edge (*v,w*) of a graph substitutes it by two edges by “adding” a vertex *u* on this edge, i.e., deleting the edge (*v,w*) and adding a new vertex *u* to the graph along with edges (*v,u*) and (*u,w*). A *subpartition* of a graph *G*(*V,E*) is defined as a result of an arbitrary series of subpartitions. The *simple subpartition* of a graph *G*(*V,E*) is a graph with |*V*|+|*E*| vertices, resulting from a subpartition of each edge in *G.*

Sparse de Bruijn graphs. Given a set of *k*-mers *Anchors* (referred to as *anchors*) from a string-set *Genome,* we consider every two consecutive occurrences *a* and *a*' of these *k*-mers in *Genome* and generate a substring of *Genome* that starts at the first nucleotide of *a* and ends at the last nucleotide of *a'*. The resulting set of substrings is denoted *Split*(*Genome,Anchors*).

The sparse de Bruijn graph *SDB(Genome,Anchors*) is defined as a graph with the vertex-set *Anchors* and the edge-set *Split*(*Genome, Anchors*) (each string in *Split*(*Genome, Anchors)* represents a label of an edge connecting its *k*-prefix with its *k*-suffix). Two vertices in this graph may be connected by multiple edges in the case several substrings of *Genome* with different labels connect the same anchors. In contrast to the de Bruijn graph, with degrees bounded by the alphabet size, vertices of the sparse de Bruijn graph may have arbitrarily large degrees. A straightforward algorithm for constructing *SDB(Genome,Anchors)* takes O(|*Genome*| · |*Anchors*| · *k*) time.

When the anchor-set coincides with the junction-set *Junctions*=*Junctions*(*Genome,k*), each vertex of *CDB*(*Genome,k*) is an anchor and each edge corresponds to two consecutive anchors in the genome traversal. Therefore, the sparse de Bruijn graph *SDB*(*Genome,Junctions)* coincides with *CDB*(*Genome,k*). Moreover, if *Junctions*$^+$ is a *superset* of all junctions that contains all junctions as well as some *false junctions* (i.e., non-branching *k*-mers from *Genome*), *SDB*(*Genome,Junctions*$^+$) is a subpartition of *CDB*(*Genome,k*).

Bloom filter. jumboDB stores all (*k*+1)-mers from *Genome* in the Bloom filter *Bloom*(*Genome,k,BloomNumber,BloomSize*) formed by *BloomNumber* different independent *hash functions*, each mapping (*k*+1)-mers into a bit array of size *BloomSize*. The Bloom filter is a compact probabilistic data structure for storing sets that may report false positives but never false negatives (see Pell et al., 2012 and Chikhi and Rizk, 2013 for applications of the Bloom filter to de Bruijn graph construction). Storing all (*k*+1)-mers from *Genome* in a Bloom filter allows one to quickly query whether an arbitrary (*k*+1)-mer occurs in *Genome*. The Bloom filter reports "yes" for all (*k*+1)-mers occurring in *Genome* but may also report "yes" for some (*k*+1)-mers that do not occur in *Genome* (with a small probability).

Section 2: Constructing the compressed de Bruijn graph of a genome

Using the junction-superset to construct the compressed de Bruijn graph of a genome. We start by describing the algorithm for constructing *CDB*(*Genome, k*) of a circular genome *Genome* (Minkin et al., 2017) before addressing a more difficult problem of constructing *CDB*(*Reads,k*). The key observation is that if the junction-set *Junctions*=*Junctions*(*Genome,k*) was known, construction of *CDB*(*Genome,k*) would be a simple task because it coincides with the sparse de Bruijn graph *SDB*(*Genome,Junctions*). Moreover, even if the junction-set is unknown but a junction-superset *Junctions*$^+$ (with a small number of false junctions) is known, one can construct *CDB*(*Genome,k*) by first constructing *SDB*(*Genome,Junctions*$^+$) and compress all non-branching paths in the resulting graph. Below we explain how to generate a junction-superset with a small number of false junctions.

Generating a junction-superset of a genome. To generate a junction-superset *Junctions*$^+$, jumboDB uses the Bloom filter to compute the upper bound on the indegree and outdegree of

each *k*-mer from *Genome* (i.e., a vertex in *DB(Genome,k)*) by checking which of its 4+4=8 extensions by a single nucleotide on the right or on the left represent (*k*+1)-mers present in the Bloom filter. A *k*-mer is called a *non-joint* if the upper bounds on its indegree and outdegree are both equal to 1, and a *joint* otherwise. JumboDB forms the junction-superset from all joints. Since the Bloom filter can report false positives, this procedure may overestimate the indegree and/or outdegree of a *k*-mer, resulting in some false junctions formed by simple vertices. In the case of linear chromosomes, it may also overestimate the indegree and/or outdegree of some dead-end junctions, e.g., to misclassify a 0-in-1-out junction as a simple vertex. However, all cross-road junctions will be found, thus solving the problem of generating a junction-superset (at least in the case of circular genomes that do have dead-end junctions), constructing a subpartition of *CDB*(*Genome,k*), and further transforming it into *CDB*(*Genome,k*). jumboDB sets the parameter *BloomSize* to be proportional to the total genome length (in the number of (*k*+1)-mers) in such a way that the false positive rate of the Bloom filter does not exceed a threshold.

Since jumboDB stores the *k*-mer hashes in the Bloom filter (instead of *k*-mers themselves), it fills the Bloom filter by calculating hashes of hashes. Since many hashing approaches take O(*k*) time to compute a hash of a *k*-mer, the running time of the algorithm scales as a factor of *k* and becomes prohibitively large when *k* is large. Below we describe how jumboDB uses the rolling hash to reduce the amortized running time for computing the hash function to O(1).

Using the rolling hash to rapidly generate the junction-superset and construct the compressed de Bruijn graph. In addition to speeding up the algorithm for generating a junction-superset, we also need to speed-up the straightforward algorithm for constructing the graph *SDB*(*Genome,Junctions*$^+$). One can speed-up building *SDB*(*Genome,Junctions*$^+$) by constructing a hashmap of *Junctions*$^+$ to support a fast check whether a *k*-mer from *Genome* coincides with a *k*-mer from *Junctions*$^+$. However, if the hash function is calculated in O(*k*) time, this approach results in O(|*Genome*| ⋅ *k*) running time, which is still prohibitive for large *k*-mer sizes. jumboDB uses a 128-bit polynomial rolling hash (Karp and Rabin, 1987) of the *k*-mers from the genome to rapidly check whether two *k*-mers (one from *Genome* and one from *Junctions*$^+$) are equal and to reduce the running time to O(|*Genome*|). Since hashes of all *k*-mers from *Junctions*$^+$ are known, this approach computes hashes of all *k*-mers in *Genome* and reveals all junctions in O(|*Genome*|) time as well as reduces the running time for constructing *CDB*(*Genome, k*) to O(|*Genome*|).

The challenge of constructing the compressed de Bruijn graph from reads. We say that a genome *Genome bridges* an edge of the compressed de Bruijn graph *CDB*(*Genome,k*) if the label of this edge represents a substring of *Genome*. A genome is classified as *bridging* if it bridges all edges of *CDB*(*Genome,k*), and *non-bridging* otherwise.

jumboDB constructs a graph *jumboDB*(*Genome,k*) that coincides with the compressed de Bruijn graph *CDB*(*Genome, k*) in the case of a genome formed by circular chromosomes or any bridging genome. However, *jumboDB(Genome,k*) differs from *CDB*(*Genome, k*) in the case of a non-bridging genome, e.g., a genome formed by linear chromosomes that do not bridge all edges of *CDB*(*Genome, k*). However, after extending the junction-superset by all *k*-prefixes and

k-suffixes of all linear chromosomes, the same algorithm will construct the graph that represents a subpartition of *CDB*(*Genome, k*). Although this subpartition can be further transformed into *CDB*(*Genome, k*) by compressing all non-branching paths, the resulting algorithm may become slow if the number of linear chromosomes is large, resulting in a large increase in the size of the junction-superset. This increase becomes particularly problematic when we construct the compressed de Bruijn graph *CDB*(*Reads, k*) from error-prone reads (each such read represents a separate linear mini-chromosome). Even more problematic is the accompanying increase in the number of calls to the hash functions that scales proportionally to the coverage of the genome by reads. An additional difficulty is that, in the absence of the genome, it is unclear how to select the appropriate size of the Bloom filter that keeps the false-positive rate below a threshold — selecting *BloomSize* to be proportional to the total read-length leads to a high false-positive rate.

Below we describe how to address these problems by assembling reads into disjointigs that form a bridging genome for the graph *CDB*(*Reads,k*) and constructing the compressed de Bruijn graph from a bridging disjointig-set *Disjointigs* instead of the read-set. Even though each disjointig may represent an error-prone assembly of reads, we show that *CDB*(*Disjointigs,k*)=*CDB*(*Reads,k*). Using disjointigs allows us to set the parameter *BloomSize* to be proportional to the total disjointig-length instead of the total read-length, thus greatly reducing the memory footprint. jumboDB sets *BloomSize*=32 · *length*(*Disjointigs*), where *length*(*Disjointigs*) is the total length of disjointigs measured in the number of (*k*+1)-mers.

Section 3: Constructing the compressed de Bruijn graph from disjointigs

Disjointigs. We define a disjointig of a genome *Genome* as a string spelled by an arbitrary path in the de Bruijn graph *DB*(*Genome,k*). A disjointig is *compact* if it starts and ends in a junction. Note that although a disjointig does not necessarily represent a substring of the genome, all (*k*+1)-mers from a disjointig occur in the genome.

A set of disjointigs of *Genome* is *complete* if each (*k*+1)-mer from *Genome* is present in a disjointig from this set. A complete disjointig-set is *compact* if each disjointig in this set is compact. If a disjointig-set *Disjointigs* is complete then *CDB*(*Genome,k*) coincides with *CDB*(*Disjointigs,k*). However, *jumboDB*(*Disjontigs,k*) may differ from *CDB*(*Genome,k*) since jumboDB does not reconstruct edges of *CDB*(*Genome,k*) that are not bridged by *Disjointigs*. However, if a disjointig-set *Disjontigs* is compact, it forms a bridging genome, implying that *jumboDB*(*Disjontigs,k*) coincides with *CDB*(*Genome,k*). Thus, our goal is to generate an initial complete disjointig-set and to transform it into a compact disjointig-set. Similarly to a disjointig of a genome, we define a disjointig of a read-set as a string spelled by an arbitrary path in the graph *DB*(*Reads,k*). Below we show how to constructs a disjointig-set from a read-set.

Generating a complete disjointig-set by constructing the sparse de Bruijn graph of reads. Given a hash function on *k*-mers and an integer *width*, a *minimizer* of a string (Roberts et al., 2004) is defined as a *k*-mer with a minimal hash in its substring formed by *width* consecutive *k*-mers. A sensible choice of the parameter *width* ensures that each read is densely covered by minimizers and that overlapping reads share many minimizers, facilitating the assembly.

The sparse de Bruijn graphs, initially introduced for reducing memory of short-read assemblers (Ye et al., 2012), can also be applied for assembling long and accurate reads. Traditionally the anchor-set *Anchors* for constructing *SDB*(*Reads,Anchors*) is constructed as a set of all minimizers across all reads. However, if the *k*-prefix and/or *k*-suffix of a read are not anchors, then a prefix and/or a suffix of this read may be missing in *SDB*(*Reads,Anchors*) since only segments between anchors are added to this graph. We thus add all *k*-prefixes and *k*-suffixes of all reads to the anchor-set formed by all minimizers. In this case, the set of (*k*+1)-mers occurring on edges of *SDB*(*Reads,Anchors*) coincides with the set of all (*k*+1)-mers from reads. We refer to the resulting set of anchors as *Anchors=Anchors*(*Reads,width,k*). jumboDB constructs the graph *SDB*(*Reads,Anchors*) and generates a complete disjointig-set as the collection of all non-branching paths in this graph. It further transforms this disjointig-set into a compact disjointig-set by extending or shortening each disjointig as described below.

Generating a compact disjointig-set. We explain how to extend/shorten a disjointig using an example of a vertex *w* in the graph *SDB*(*Reads,Anchors*) that has one incoming edge *in* and two outgoing edges out_1 and out_2 (a similar approach is applicable to any vertex). The previously constructed disjointed-set includes the disjointig *in* (that we will extend) and disjointigs out_1 and out_2 (that we will shorten).

Edges out_1 and out_2 share their first *k*-mer (that labels vertex *w*) and possibly their second, third, etc. *k*-mers. Let *prefix*(out_1,out_2) be the longest common prefix of these edges and *last*(out_1,out_2) be the last *k*-mer of this prefix. While the vertex *w* is not necessarily a junction, the edges out_1 and out_2 must share a junction, specifically the junction *last*(out_1,out_2). We thus extend the disjointig *in* ending in *w* by concatenating it with the suffix of *prefix*(out_1,out_2) starting at position *k*, and shorten the disjointigs starting in *w* by removing their prefixes of length |*prefix*(out_1,out_2)|-*k*. The resulting disjointig-set contains the same collection of (*k*+1)-mers as the initial disjointig-set but the disjointigs that previously started/ended at an anchor *w*, now start/end at a junction *last*(out_1,out_2) of the graph *CDB(Reads,k).* Applying the described procedure to all vertices of the graph *SDB*(*Reads, Anchors*) transforms the initial disjointig-set into a compact disjointig-set.

Section 4: Outline of the jumboDB algorithm

Outline of the jumboDB algorithm. Below we summarize all steps of jumboDB. Appendix 1 describes its parameters, while Appendix 2 describes how jumboDB maps reads to this graph.

1. Generate the anchor-set *Anchors=Anchors*(*Reads,width,k*) by constructing the set of all minimizers in *Read*s and extending this set by all *k*-prefixes and *k*-suffixes of all reads.
2. Construct the sparse de Bruijn graph *SDB*(*Reads,Anchors*).
3. Construct a complete disjointig-set formed by all non-branching paths in *SDB*(*Reads,Anchors*).
4. Transform it into a compact disjointig-set *Disjointigs* by extending/shortening disjointigs.
5. Generate the Bloom filter *Bloom*(*Disjointigs,k,BloomNumber,BloomSize*) with *BloomSize*=32 · *length*(*Disjointigs*).

6. Compute the upper bounds on the indegree and outdegree of each *k*-mer from *Disjointigs* using the Bloom filter and the rolling hash.
7. Construct the junction-superset *Junctions*$^+$ as the set of all joints in *Disjointigs*.
8. Construct the string-set *Split(Disjointigs,Junctions*$^+$*)* to generate edges of a subpartition of the compressed de Bruijn graph *CDB*(*Disjointigs,k*).
9. Compress all 1-in-1-out vertices in this graph to generate *CDB(Disjointigs,k*) that coincides with *CDB*(*Reads,k*).

Section 5: Benchmarking jumboDB and error-correcting HiFi reads

Benchmarking jumboDB. We do not benchmark jumboDB against other de Bruijn graph construction tools since there are no tools for constructing the compressed de Bruijn graphs for large *k*-mers yet (the MBG tool (Rautiainen and Marschall, 2020) constructs the sparse de Bruijn graph, a coarse approximation of the compressed de Bruijn graph).

Table Time/Memory provides information about the running time and memory footprint of jumboDB. Since homopolymer runs represent the dominant source of errors in HiFi reads, we collapse each homopolymer run X...X in HiFi reads (and in the assembled genome) into a single nucleotide X and benchmark jumboDB using the datasets of homopolymer-collapsed (HPC) reads Below we list the benchmarking datasets that are described in details in Appendix 3:

- The **ECOLI** dataset contains HiFi reads from the *E. coli* genome.
- The **T2T** dataset contains HiFi reads from a human cell line generated by the Telomere-To-Telomere (T2T) consortium (Nurk et al., 2020). The T2T dataset was semi-manually assembled into a sequence *HumanGenome* by integrating information generated by multiple sequencing technologies.
- The **T2TErrorFree** dataset is derived by mapping reads from the T2T dataset to *HumanGenome* and substituting each mapped read by the genomic segment it spans.
- The **chrX** dataset, that we use for benchmarking our algorithm for error-correcting HiFi reads, is a subset of the T2T dataset that contains all reads originating from the chromosome X (referred to as chrX) The **cenX (cen6)** dataset is a subset of the chrX dataset that contains all reads originating from the centromere of chromosome X (chromosome 6) referred to as cenX (cen6). The **cenXErrorFree (cen6ErrorFree)** datasets contain error-free reads from the T2TcenX (T2Tcen6) datasets.

T2T								
	time (h)	memory (Gb)	#vertices	#edges	total edge length (Gbp)	#paths	median path length	#complex vertices
251	2.4	52	29806686	43750620	10	5566455	159	208737
511	2.7	54	23171326	33230906	15	5566321	111	95702

1001	3.2	65	17032673	23357700	18	5559705	69	36041
2001	3.6	81	12025225	14947471	23	5422657	35	11493
3001	4.2	90	9927841	11160720	25	5157197	22	5308
5001	3.9	93	8152006	7560390	27	4564678	10	1581
T2TErrorFree								
	time (h)	memory (Gb)	#vertices	#edges	total edge length (Gbp)	#paths	median path length	#complex vertices
251	0.6	32	311042	472380	2.1	874719	55	7432
511	0.6	33	143011	214517	2.1	590730	30	1230
1001	0.7	33	64371	95716	2.1	456338	13	220
2001	0.7	34	21724	31862	2.0	307824	5	16
3001	0.6	35	10035	14723	2.0	205879	4	8
5001	0.6	36	3530	4956	2.0	88460	3	2
ECOLI								
	time(min)	memory (Gb)	#vertices	#edges	total edge length (Gb)	#paths	median path length	#complex vertices
251	3.2	2.2	378261	561569	0.5	190456	1057	10187
511	2.5	2.2	316317	457755	0.7	190386	800	6546
1001	2.2	2.1	246417	338726	1.0	190134	518	3389
2001	2.5	2.1	180581	221930	1.3	186832	267	1240
3001	2.5	2.2	150975	164848	1.5	176924	156	574
5001	1.5	2.1	101569	88062	1.3	116270	52	18588

Table Time/Memory. The running time and memory footprint of jumboDB as well as the number of vertices and edges in the constructed compressed de Bruijn graphs for the T2T (top), T2TErrorFree (middle), and ECOLI (bottom) datasets. The table also provides information about the

number of paths (excluding single-edge paths), their median length (in the number of edges they traverse), and the number of complex vertices. The running time/memory footprint for the T2TErrorFree read-set hardly changes with an increase in the *k*-mer size, suggesting that the running time/memory footprint of jumboDB mainly depends on the size of the compressed de Bruijn rather than the *k*-mer size. All tools were benchmarked on a computational node with two Intel Xeon 8164 CPUs, with 26 cores each and 1.5 TB of RAM. All runs were done in 32 threads.

Error-correction of HiFi reads. Error-correction was first introduced in the context of Sanger reads (Pevzner et al., 2001) and became ubiquitous in both short-read and long-read assemblers (Chaisson et al., 2008, Kelly et al., 2010, Medvedev et al., 2011, Bankevich et al., 2012, Nikolenko et al., 2013, Lima et al., 2020). However, error-correction of long and accurate reads remains a poorly explored topic — the Hifiasm assembler (Cheng et al., 2020) is currently the only error-correcting tool for HiFi reads.

HiFi reads in the T2T dataset have a mean error rate of 0.2% per nucleotide. Collapsing all homopolymer runs reduces the error rate to 0.062%, with 38% of all reads in the T2T dataset being error-free. However, the remaining errors have to be corrected to ensure that the graph *CDB*(*Reads,k*) of the homopolymer-collapsed read-set represents a good approximation of the graph *CDB*(*Genome,k*) of the homopolymer-collapsed genome. For example, while the graph *CDB*(T2T,511) has ≅33 million edges, the graph *CDB*(T2TErrorFree,511) has only ≅214 thousand edges, illustrating that error correction is needed even after collapsing all homopolymer runs.

The HiFiasm error correction (Cheng et al., 2020) reduced the errors in reads to 210 errors per megabase and increased the percentage of error-free reads to ≈92%. Since the de Bruijn graphs have been successfully used for error-correcting short and accurate reads, we implemented a simple *path-rerouting* and *bulge-collapsing* approach to error-correct HiFi reads that is inspired by a more involved *graph simplification* procedure in the SPAdes assembler (Bankevich et al., 2012). Appendix 4 benchmarks this error correction approach on the chrX dataset and illustrates that it reduces the errors in reads to only 3.6 errors per megabase and increases the percentage of error-free reads to ≈96% (after "breaking" a small number of reads as described in Appendix 4). Even though it represents a significant reduction in the number of errors as compared to the Hifiasm error-correction procedure, we are now working on further optimization of the LJA error-correction procedure since the remaining errors may fragment the compressed de Bruijn graph and propagate during the multiplex graph construction step.

Section 6: Graph Transformation Algorithm: from *CDB*(*Reads,k*) to *CDB*(*Reads,k*+1)

Iterative construction of the compressed de Bruijn graph. Below we describe our transDB algorithm for transforming the graph *CDB*(*Reads,k*) into *CDB*(*Reads,K*) for *K*>*k* by iteratively increasing the *k*-mer size by 1 at each iteration. Although launching jumboDB to construct *CDB*(*Reads,k*) followed up by transDB transformations takes more time than simply launching jumboDB to construct *CDB*(*Reads,K*), we use it as a stepping stone toward the multiplex de Bruijn graph construction.

Below we consider graphs, where each edge is labeled by a string, and each vertex *w* is assigned an integer v*ertexSize*(*w).* We limit attention to graphs where suffixes of length *vertexSize*(*w*) for all incoming edges into *w* coincide with prefixes of length *vertexSize(w)* for all outgoing edges from *w*. We refer to the string of length *vertexSize*(*w*) that represents these

prefixes/suffixes as the label of the vertex *w*. Below we consider graphs with specified edge-labels and assume that vertex-labels can be inferred from these edge-labels and that different vertices have different vertex-labels. Although in this section, *vertexSize*(*w*) is the same for all vertices, it will vary among vertices in the multiplex de Bruijn graph.

Transition graph. Let *Transitions* be *an* arbitrary set of pairs of consecutive edges (*v,w*) and (*w,u*) in an edge-labeled graph *G*. We define the *transition graph G*(*Transitions*) as follows. Every edge *e* in *G* corresponds to two vertices e_{start} and e_{end} in *G*(*Transitions*) that are connected by a blue edge. This blue edge in *G*(*Transitions*) inherits the label of the edge *e* in *G* and we set $vertexSize(e_{start})=vertexSize(e_{end})=k+1$ (vertex-labels are uniquely defined by the (*k*+1)-suffixes/prefixes of the incoming/outgoing edges in each vertex). If an edge *e* in *G* is labeled by a (*k*+1)-mer, the corresponding blue edge in *G*(*Transitions*) is collapsed into a single vertex $e_{start}=e_{end}$ in *G*(*Transitions*). In addition to blue edges, each pair of edges *in*=(*v,w*) and *out*=(*w,u*) in *Transitions* adds a red edge between in_{end} and out_{start} to the graph *G*(*Transitions*). The label of this red edge is defined as a (*k*+2)-mer $symbol_{-(k+1)}(in)*label(w)*symbol_{(k+1)}(out)$, where $symbol_i(e)$ stands for the *i*-th symbol of *label*(*e*), and $symbol_{-i}(e)$ stands for the *i*-th symbol from the end of *label*(*e*). When the set *Transitions* includes all pairs of consecutive edges in graph *G,* the transition graph is the standard *line graph* (Wilson, 2015) of the simple subpartition of *G*.

Path graph. We say that a path *traverses* a vertex *w* in a graph if it both enters and exits this vertex. Given a path-set *Paths* in a graph, we denote the set of all paths containing an edge (*v,w*) as *Paths*(*v,w*), the set of all paths traversing a vertex *w* as *Paths*(*w*), and the set of all paths visiting incoming edges into vertex *v* as $Paths^+(v)$ (each path in $Paths^+(v)$ either traverses *v* or stops at *v*). Given a path-set *Paths* in a graph *G*, we define the set *Transitions*(*Paths*) as the set of all pairs of consecutive edges in all paths from *Paths*. A *path-graph* of a path-set *Paths* in a graph *G* is defined as the transition graph *G*(*Transitions(Paths*)).

Let *Paths* be the set of all read-paths in the compressed de Bruijn graph *G=CDB*(*Reads*,*k*). The graph *G(Transitions(Paths*)) is a subpartition of the graph *CDB*(*Reads*,*k*+1) (after properly defining edge-labels and ignoring colors of edges). For each edge *e* in the graph *CDB*(*Reads,k*), we maintain the set of paths *Paths*(*e*) containing this edge. A path $e_1, e_2, e_3,\ldots$ in *CDB*(*Reads*, *k*) corresponds to a blue-red path e_1, transition edge between e_1 and e_2, e_2, transition edge between e_2 and e_3, $e_3,\ldots$ in *G*(*Transitions*(*Paths*)), where labels of blue edges have lengths at least (*k*+1) and labels of red edges represent (*k*+2)-mers. Therefore, a straightforward approach to constructing the graph *G*(*Transitions*(*Paths*) (that recomputes labels from scratch at each iteration) faces the time/memory bottleneck since it nearly doubles the path lengths at each iteration. However, the compressed de Bruijn graph is getting less tangled with an increase in the *k*-mer size, implying that the vast majority of the newly introduced red edges are merely subpartitions of longer non-branching paths. Below we describe how transDB avoids the time/memory bottleneck by modifying rather than recomputing the edge labels from scratch.

Transforming simple vertices. A path $(v_1,\ldots v_i,\ldots,v_n)$ in a graph is called *out-unambiguous* (*in-unambiguous*) if the outdegrees (indegrees) of all vertices in this path except the first and the last one are equal to 1. A path $(v_1,\ldots v_i,\ldots,v_n)$ is called *unambiguous* if there is an edge (v_i,v_{i+1}) in

this path such that $(v_i,...,v_n)$ is an *out-unambiguous* path and $(v_1,...v_{i+1})$ is an *in-unambiguous* path. We refer to unambiguous paths in the compressed de Bruijn graph as *virtual reads*. Note that in the case when *Genome* is formed by circular chromosomes, all virtual reads in the compressed de Bruijn graph *CDB*(*Genome,k*) represent substrings of *Genome* and thus can be safely added to any read-set. In the case of linear chromosomes, we assume that the *k*-prefix and *k*-suffix of each chromosome correspond to dead-ends in the graph *CDB(Reads, k).*

Given a path-set *Reads* in a graph *G*, we call edges (*v,w*) and (*w,u*) in *G paired* if *Transitions(Reads)* contains this pair of edges. A vertex *w* in *G* is classified as *paired* if each edge incident to *w* is paired with at least one other edge incident to *w*, and *unpaired*, otherwise.

For a simple paired vertex, the local topology of the graph "around" this vertex remains the same after the graph transformation. In the framework of the Iterative de Bruijn graph (when the set of reads is complemented by virtual reads), the local topology of both paired and unpaired simple vertices (with the exception of the dead-end simple vertices) remains the same after this transformation. Below we describe how transDB speeds-up transformations of simple vertices.

For each parameter *k*, the vast majority of vertices in the compressed de Bruijn graph of reads are 2-in-1-out and 1-in-2-out vertices (Appendix 4). Below we describe the graph transformation of 2-in-1-out vertex *w* with incoming edges in_1 and in_2 and the outgoing edge *out* (transformations of *N*-in-1-out and 1-in-*N*-out vertices are performed similarly). This transformation merely substitutes the *k*-mer label of this vertex by the (*k*+1)-mer $label(w)*symbol_{k+1}(out)$. It preserves the label of the edge *out* and adds a single symbol $symbol_{k+1}(out)$ after the end of labels of edges in_1 and in_2.

Transforming complex vertices. The transformation of a complex *N*-in-*M*-out vertex *w* results in substituting this vertex by *N*+*M* vertices and adding up to $N \cdot M$ red edges (that connect *N* added vertices with *M* added vertices) to the graph. Each path from *Paths*(*w*) traversing *t* complex vertices will be transformed into a path with *t* red edges and the non-branching paths that may result from this transformation have to be compressed into single edges.

To efficiently implement this transformation, transDB generates the list of paths traversing each new red edge (for up to new *N***M* red edges for each complex *N*-in-*M*-out vertex) using the list of paths *Paths*(*v,w*) traversing each edge (*v,w*) in the graph. The transformation of a complex vertex *w* takes $|Paths^+(w)|$ time. The number of operations to transform all complex vertices is bounded by the sum of $|Paths^+(v)|$ over all complex vertices in the graph. Since the number of complex vertices in a compressed de Bruijn graph is small (see Appendix 4) and since processing a simple vertex takes constant time, the transformation of *CDB*(*Reads*, *k*) into *CDB*(*Reads*, *k*+1) is fast for large *k* (e.g., for *k*=5001 because the graph *CDB*(T2T,5001) is small) even though it can be rather slow for small *k* (e.g., for *k*=511 because the graph *CDB*(T2T,511) is large). For example, constructing the graph *CDB*(chrX,511) using jumboDB and further transforming it into the graph *CDB*(chrX,5001) using transDB takes 5+393 minutes, while the direct construction of *CDB*(chrX,5001) using jumboDB takes just 3 minutes. However, iterative increasing of the *k*-mer size during the construction of the multiplex de Bruijn graph is

only crucial for large k (e.g., greater than 5001) that, as we demonstrate below, can be done rapidly.

Section 7: Multiplex de Bruijn graph

Limitation of the de Bruijn graph approach to genome assembly. The choice of the k-mer size greatly affects the complexity of the graph $DB(Reads,k)$. There is no perfect choice since gradually increasing k leads to a less tangled but more fragmented de Bruijn graph. This trade-off affects the contiguity of assembly, particularly in the case when the k-mers coverage by reads is non-uniform, let alone when some genomic k-mers are missing in the read-set.

The coverage of an edge (a (k+1)-mer) in $DB(Reads,k)$ is defined as the total number of traversals of this edge by all read-paths. The (k+1)-mers from the genome that are missing in a read-set (coverage zero) are missing in $DB(Reads,k)$, reducing the contiguity of assembly. The (k+1)-mers with small coverage may also result in fragmented assemblies since our error-correction procedure may break regions with a small coverage. Ideally, we would like to use larger k-mer sizes in the high-coverage regions and smaller k-mer sizes in the low-coverage regions. The *iterative de Bruijn graph* approach (Peng et al., 2010) addresses this challenge by incorporating information about the de Bruijn graphs for a range of parameters $k_1 < k_2 < \ldots < k_t$ into the de Bruijn graph for a larger value $K > k_t$. Although this approach was implemented in many short-read assemblers (Peng et al., 2012, Bankevich et al., 2012, Peng et al., 2012), it still constructs a graph with a fixed k-mer size equal to K. Boucher et al., 2015 described the *variable-order de Bruijn graph* that compactly represents information about the de Bruijn graph of a read-set across multiple k-mer sizes. Lin and Pevzner, 2014 described a theoretical approach for constructing the de Bruijn graphs with vertices labeled by k-mers of varying sizes that however was not designed for practical genome assembly challenges.

Below we introduce the concept of the multiplex de Bruijn graph (with vertices labeled by k-mers of varying sizes) and describe its applications to genome assembly. We note that we add all virtual reads to the read-set *Reads* during the construction of the multiplex de Bruijn graph.

Multiplex graph transformation. The important property of $CDB(Genome,k)$ is that there exists a genome traversal of this graph. If there exists a genome traversal of the graph $CDB(Reads,k)$, we want to preserve it in $CDB(Reads,k+1)$ after the graph transformation. However, it is not necessarily the case since the transformation of $CDB(Reads,k)$ into $CDB(Reads,k+1)$ may create dead-ends (each unpaired vertex in $CDB(Reads,k)$ results in a dead-end in $CDB(Reads,k+1)$), thus "losing" the genome traversal that existed in $CDB(Reads,k)$. Below we describe a multiDB algorithm for transforming $CDB(Reads,k)$ into the multiplex de Bruijn graph $MDB(Reads,k+1)$ that avoids creating dead-ends whenever possible by introducing vertices of varying sizes in this graph.

multiDB transforms each paired vertex of $CDB(Reads,k)$ using the transDB algorithm and "freezes" each unpaired vertex by preserving its k-mer label and the local topology. It also freezes some vertices adjacent to the already frozen vertices even if these vertices are simple. Specifically, if a frozen vertex u is connected with a non-frozen vertex v by an edge of length

VertexSize(v) + 1, we freeze *v*. The motivation for freezing *v* is that, if we do not freeze it, we would need to remove the edge connecting *u* and *v* in *MDB*(*Reads, k*+1), disrupting the topology of the graph. multiDB continues the multiplex graph transformations for all paired vertices (while freezing unpaired vertices) with gradually increasing *k*-mer sizes from *k* to *K*, resulting in the multiplex de Bruijn graph *MDB*(*Reads,K*) with varying *k*-mer sizes.

We classify a read-set as *incomplete* if it does not contain reads supporting some genomic transitions through a vertex in the de Bruijn graph. Similarly to the iterative de Bruijn graph approach (Peng et al., 2010), although the graph *MDB*(*Reads,K*) results in a more contiguous assembly than *CDB*(*Reads,K*), there is a risk that some multiplex graph transformation may "destroy" the genome traversal and even lead to assembly errors in the case of an incomplete read-set. Appendix 6 describes these risks and illustrates that a multiplex graph transformation may be overly-optimistic (by transforming vertices that should have been frozen) and overly-pessimistic (by freezing vertices that should have been transformed). Appendices 7 and 8 describe how to minimize the risk of overly-optimistic graph transformations.

Assembling cenX. We first illustrate the construction of the multiplex de Bruijn graph using the error-free read-set cenXErrorFree. Afterward, we show that the multiplex de Bruijn graph approach results in a complete cenX assembly from the real cenX read-set. The graph *CDB*(cenXErrorFree, 5001) contains 34 vertices and 49 edges. We transformed this graph into the multiplex de Bruijn graph *MDB*(cenXErrorFree,40001) by increasing the *k*-mer size and reducing it to a single edge at the *K*-mer size equal to 37771, illustrating that error-free HiFi reads enable cenX assembly in a single contig. In contrast, the compressed de Bruijn graph *CDB*(cenXErrorFree,37771) cannot be constructed, because all reads in the set cenXErrorFree are shorter than 37771bp. We emphasize that the multiplex de Bruijn graph utilizes virtual reads that are often longer than real HiFi reads, explaining why it was important to increase the *K*-mer size beyond the length of all reads in the cenXErrorFree read-set.

To assemble the real cenX read-set, we implemented the traditional error correction based on the removal of tips and bulges from the de Bruijn graph (similar to the SPAdes graph simplification procedure (Bankevich et al., 2012)) coupled with a path rerouting procedure that keeps track of read paths in the graph (Appendix 4). This error correction resulted in significant improvement in the error rate (0.01%) and the percentage of error-free reads (≈94%). We constructed the graph *CDB(*cenXErrorCorrected,5001) with 34 vertices and 49 edges using jumboDB and transformed it into the multiplex de Bruijn graph *MDB*(cenXErrorCorrected, 40001) using multiDB. This multiplex de Bruijn graph has reduced to a single edge at the *K*-mer size equal to 37771, illustrating that HiFi reads enable cenX assembly into a single contig. Transformation of *CDB(*cenXErrorCorrected,5001) into *MDB*(cenXErrorCorrected,40001) takes less than a minute.

Assembling cen6. Extremely repetitive cen6 is one of the most difficult-to-assemble regions of the human genome that was recently assembled using ultralong Oxford Nanopore (ONT) reads (Bzikadze and Pevzner, 2020). Assembling cen6 using shorter HiFi reads is challenging since it contains long nearly identical repeats. Below we show that even if HiFi reads in the T2T

dataset were error-free, it still would not be possible to completely assemble cen6, illustrating that ultralong ONT reads are needed to generate telomere-to-telomere assemblies.

The graph *CDB*(cen6ErrorFree, 5001) contains 152 vertices and 226 edges (the read-set cen6ErrorFree contains no missing 5001-mers). multiDB transformed this graph into a small multiplex de Bruijn graph *MDB*(cen6ErrorFree, 40001) with only 10 vertices and 15 edges (Figure 1). Transformation of *CDB*(cen6ErrorFree, 5001) into *MDB*(cen6ErrorFree, 40001) takes less than two minutes. The large reduction in complexity of *MDB*(cen6ErrorFree, 40001) as compared to *CDB*(cen6ErrorFree, 5001) illustrates the value of multiplex de Bruijn graphs for follow up repeat resolution using ultralong ONT reads. Although our error-correction of the cen6 read-set made nearly all reads error-free, the remaining error-prone reads generate some bulges, making it difficult to construct the multiplex de Bruijn graph. Our next goal is to further optimize error correction to enable the construction of the multiple de Bruijn graph of the most repetitive genomic regions.

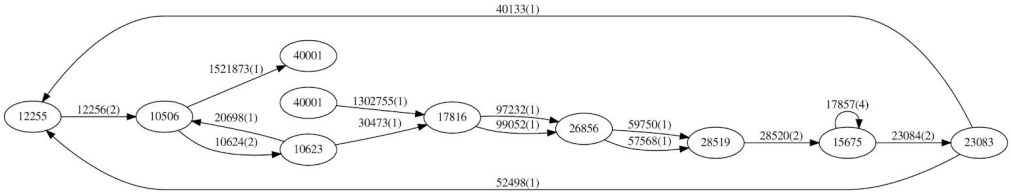

Figure 1. The multiplex de Bruijn graph *MDB*(cen6ErrorFree,40001) obtained from the compressed de Bruijn graph *CDB*(cen6ErrorFree, 5001). The lengths of frozen vertices are smaller than 40001. The length of an edge and its multiplicity are shown next to each edge.

Conclusions

The development of assembly algorithms for both short and accurate reads (e.g., reads generated by Sanger and Illumina technologies) and long and error-prone reads (e.g., reads generated by Pacific Biosciences and Oxford Nanopore technologies) started from applications of the overlap/string graph approach. Even though this approach has an inherent theoretical limitation (representing reads that are substrings of other reads results in fragmented assemblies) and becomes slow and error-prone with respect to detecting overlaps detection in the most repetitive regions, the alternative de Bruijn graph approach (Idury and Waterman, 1995, Pevzner et al., 2001) was often viewed as a theoretical concept rather than a practical genome assembly method.

Even after it turned into the most popular method for assembling short and accurate reads, the development of algorithms for assembling long and error-prone reads again started from the overlap/string graph approach (Koren et al., 2012, Chin et al., 2013, 2016) since the de Bruijn graph approach was viewed as inapplicable to error-prone reads due to the “error myth” (Roberts et al., 2013) - since long *k*-mers from the genome typically do not even occur in error-prone reads, it seemed unlikely that the de Bruijn graph approach may assemble such reads. However, the development of the Flye (Kolmogorov et al., 2019) and wtdbg2 (Ruan and Li, 2020) assemblers demonstrated once again that, even for error-prone reads, the de Bruijn

graph-based assemblers result in accurate and order(s) of magnitude faster algorithms than the overlap/string graph approach.

Since the de Bruijn graph approach was initially designed for assembling accurate reads, it would seem natural to use it for assembling long and accurate reads. However, the history repeated itself and the first HiFi assemblers again relied on the overlap/string graph approach (Nurk et al., 2020, Cheng et al., 2020). We and Rautiainen and Marschall, 2000 described alternative de Bruijn graph approaches for assembling HiFi reads, illustrating that the "contest" between the de Bruijn graph approach and the overlap/string graph approach continues.

Bibliography

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., Pyshkin, A.V., Sirotkin, A.V., Vyahhi, .N, Tesler, G., Alekseyev, M.A., Pevzner, P.A. (2012) SPAdes: a new genome assembly algorithms and its applications to single cell sequencing. *J Comput Biol.,* 19:455-77

Bankevich, A., Pevzner, P. (2020) mosaicFlye: Resolving long mosaic repeats using long error-prone reads, *biorxiv,* doi: https://doi.org/10.1101/2020.01.15.908285

Bloom, B.H. (1970) Space/time tradeoffs in hash coding with allowable errors, *Communications of the ACM*, 13, 422–426,

Boucher, C., Bowe, A, Gagie, T., Puglisi, S.J., Sadakane, K. (2015) Variable-Order de Bruijn Graphs. *Data Compression Conference (DCC 2015)*, 383–392

Burrows, M., Wheeler, D. J. (1994), A block sorting lossless data compression algorithm, *Digital Equipment Corporation*, Technical Report 124.

Bzikadze, A. V. & Pevzner, P. A. (2020) Automated assembly of centromeres from ultra-long error-prone reads. *Nat. Biotechnol*ogy, 38, 1309-1316

Chaisson, M.J., Pevzner, P.A. (2008) Short read fragment assembly of bacterial genomes. *Genome Res.* 18, 324-30.

Cheng, H., Gregory T Concepcion, G.T., Feng, X., Zhang, H., Li, H. (2020) Haplotype-resolved de novo assembly with phased assembly graphs. *arXiv*:2008.01237

Chikhi, R., Rizk, G. (2013) Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms for Molecular Biology* 8, 22

Chin, C. S. *et al*. (2013) Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nat. Methods* 10, 563-569

Chin, C., Peluso, P., Sedlazeck, F. Nattestad, M., Concepcion, G.T., Clum, A., Dunn, C., O'Malley, R., Figueroa-Balderas, R., Morales-Cruz, A., Cramer, G.R. Delledonne, M., Luo, C., Ecker, J.R., Cantu, D., Rank, D.R., Schatz, M.C. (2016) Phased diploid genome assembly with single-molecule real-time sequencing. *Nat Methods* 13, 1050–1054

Compeau, P.E., Pevzner, P.A., Tesler, G. (2011) How to apply de Bruijn graphs to genome assembly *Nat Biotechnol.,* 29:987-91

Fredman, M.L., Komlos, J., Szemeredi,J (1984) Storing a sparse table with 0(1) worst case access time. *Journal of the Association for Computing Machinery*, 31:538–54

Idury, R.M. Waterman, M.S. (1995) A new algorithm for DNA sequence assembly. *J Comput Biol.*, 2:291-306.

Jain, C., Rhie, A., Zhang, H., Chu, C., Walenz, B.P., Koren, S., Phillippy, A.M. (2020) Weighted minimizer sampling improves long read mapping, *Bioinformatics*, 36, i111–i118

Karp, R.M., Rabin, M.O. (1987). Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*. 31: 249–260.

Kelley, D.R., Schatz, M.C. & Salzberg, S.L. (2010) Quake: quality-aware detection and correction of sequencing errors. *Genome Biology* 11, R116.

Kolmogorov, M., J. Yuan, Y. Lin, P.A. Pevzner. (2019) Assembly of long error-prone reads using repeat graphs. *Nature Biotechnology*, 37, 540

Kolmogorov, M., Bickhart, D.M., Behsaz, B., Gurevich, A., Rayko, M., Shin, S.B., Kuhn, K., Yuan, J., Polevikov, E., Smith T.P.L., Pevzner, P.A. (2020) metaFlye: scalable long-read metagenome assembly using repeat graphs. *Nature Methods,* 17, 1103-1110

Koren, S. *et al.* (2012) Hybrid error correction and de novo assembly of single-molecule sequencing reads. *Nat. Biotechnol.* 30, 693-700

Li, D., Liu, C.M, Luo, R., Sadakane, K., Lam, T.W. (2015) MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics* 31: 1674–1676.

Li, H., (2018). Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18), 3094-3100.

Lima, L., Marchet, C., Caboche, S., Da Silva, C., Istace, B., Aury, J.M., Touzet, H., Chikhi, R. (2020) Comparative assessment of long-read error correction software applied to Nanopore RNA-sequencing data, *Briefings in Bioinformatics*, 21, 1164–1181

Lin, Y., Pevzner, P.A. (2014) Manifold de Bruijn Graphs. *Lecture Notes in Bioinformatics*, 8701*:* 296-310

Lin, Y., Nurk, S., Pevzner P.A. (2014) What is the difference between the breakpoint graph and the de Bruijn graph? *BMC Genomics* 15 (Suppl 6) S6

Medvedev, P., Scott, E., Kakaradov, B., Pevzner, P. (2011)Error correction of high-throughput sequencing datasets with non-uniform coverage. *Bioinformatics*. 27: i137-41.

Minkin, I., Pham, S., Medvedev, P. (2017) TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics*, 33, 4024–4032

Myers, E. W. (2005) The fragment assembly string graph. *Bioinformatics* 21, ii79–ii85

Nikolenko, S.I., Korobeynikov, A.I. & Alekseyev, M.A. (2013) BayesHammer: Bayesian clustering for error correction in single-cell sequencing. *BMC Genomics* 14, S7

Nurk, S., Brian P. Walenz, B.P., Rhie, A., Vollger, M.R., Logsdon, G.A., Grothe, R., Karen H. Miga, K.H., Eichler, E.E., Phillippy, A.M., Koren, S. (2020) HiCanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads, *Genome Research* (in press)

Pell, J, Hintze., A, Canino-Koning, R., Howe, A., Tiedje, J.M., Brown, C.T. (2012) Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences of the United States of America*, 109:13272-13277

Peng, Y., Leung, H.C.M., Yiu, S.M., Chin, F.Y.L. (2010) IDBA–a practical iterative de Bruijn graph *de novo* assembler. *Lecture Notes in Computer Science*. 6044, 426–440.

Peng, Y., Leung, H.C.M., Yiu, S.M., Chin, F.Y.L. (2012) IDBA-UD: a *de novo* assembler for single-cell and metagenomic sequencing data with highly uneven depth. *Bioinformatics*, 28:1420–1428.

Pevzner, P.A., Tang, H., Waterman, M.S. (2001) An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences,* 98, 9748-9753

Pevzner, P.A., Tang, H. (2001) Fragment assembly with double-barreled data. *Bioinformatics,* Suppl 1:S225-33.

Pevzner P. Tang H. Tesler G. (2004) De novo repeat classification and fragment assembly. *Genome Res*. 14:1786–1796.

Rautiainen, M., Marschall, T. (2000) MBG: Minimizer-based Sparse de Bruijn Graph Construction. *Biorxiv*, doi: https://doi.org/10.1101/2020.09.18.303156

Roberts, M., Hayes, W., Hunt, B.R., Mount, S.M., Yorke, J.A. (2004) Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20, 3363–3369

Roberts, R.J., Carneiro, M.O. & Schatz, M.C. (2013) The advantages of SMRT sequencing. *Genome Biol* 14, 405 (2013).

Ruan, J., Li, H. (2020) Fast and accurate long-read assembly with wtdbg2. Nature Methods, 17, 155-158

Simpson J. Wong K. Jackman S., et al. (2009) ABySS: a parallel assembler for short read sequence data. *Genome Res*. 19:1117–1123.

Wenger, A.M., Peluso, P., Rowell, W.J., Chang, P.C., Hall, R.J., Concepcion, G.T., Ebler, J., Fungtammasan, A., Kolesnikov, A., Olson, N.D., Töpfer, A., Alonge, M., Mahmoud, M., Qian, Y., Chin, C.S., Phillippy, A.M., Schatz, M.C., Myers, G., DePristo, M.A., Ruan, J., Marschall, T., Sedlazeck, F.J., Zook, J.M., Li H., Koren, S., Carroll, A., Rank, D.R., Hunkapiller, M.W. (2019) Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nat Biotechnol.* 37:1155-1162.

Wilson, R.J., (2015) *Introduction to Graph Theory*. 5th Edition, Prentice Hall

Ye, C., Ma, Z.S., Cannon, C.H., Pop, M., Yu, D.W. (2012) Exploiting sparseness in de novo genome assembly. *BMC Bioinformatics*.;13 Suppl 6:S1.

Zerbino, D.R., Birney, E. (2008) Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res* **18**: 821–9.

Appendices

1. Parameters of jumboDB

Appendix 1: Parameters of jumboDB

Ideally, when one constructs the compressed de Bruijn graph of *Genome*, the parameter *BloomSize* should be selected to be proportional to the number of different (*k*+1)-mers in *Genome* in such a way that the false positive rate of the Bloom filter does not exceed a threshold (the default value is 10^{-4}). In the case of constructing the compressed de Bruijn graph of *Genome*, although the number of different (*k*+1)-mers in *Genome* is unknown, jumboDB uses |*Genome*| as a proxy for this number. As a tradeoff between the memory footprint and the false positive rate of the Bloom filter, we use 32 bits per (*k*+1)-mer resulting in approximately 10^{-4} false-positive rate. Increasing the number of hash functions decreases the false positive rate but, at the same time, increases the query time of the Bloom filter. To minimize the running time, jumboDB uses the default (small) value *BloomNumber*=5.

Although the total genome length is a good proxy for the number of different (*k*+1)-mers in *Genome,* the total read length greatly overestimates the number of different (*k*+1)-mers in *Reads,* thus making it unclear how to set the size of the Bloom filter for constructing *CDB*(*Reads,k*). Since jumboDB constructs *CDB*(*Disjointigs,k*) instead of *CDB*(*Reads,k*), it uses the total disjointing length instead of the total read length to set the parameter *BloomSize*, thus greatly reducing the memory footprint.

jumboDB uses a 128-bit polynomial rolling hash of the (*k*+1)-mers from reads. Although hashing may lead to *collisions* when different (*k*+1)-mers result in the same hash function, a 128-bit rolling hash has a very low probability of collisions. Indeed, if a hash is viewed as a pseudo-random function on a set of 128-bit integers, the probability of a collision during the construction of the compressed de Bruijn graph *CDB*(*Genome*,*k*) is extremely low even for large genomes. Moreover, the probability of a collision during the construction of the graph *CDB*(*Reads*,*k*) remains small for large read-sets - it is estimated as 10^{-17} in the case of the T2T read-set (described below) with approximately 10^{11} 511-mers. We performed multiple tests on *k*-mers from the T2T read-sets and detected no collisions.

jumboDB uses the default value *width*=*maxWidth*-*k*, where *maxWidth*=10000 for selecting minimizers.

Appendix 2: Mapping reads to the compressed de Bruijn graph

Each read traverses a *read-path* in the compressed de Bruijn graph *CDB*(*Reads,k*). A read-path is called a *single-edge* path if it traverses a single edge in *CDB*(*Reads,k*). After constructing *CDB*(*Reads,k*), jumboDB maps each read to this graph, generates its read-path, and identifies the starting (ending) position of this read within the starting (ending) edge of its read-path.

jumboDB discards all short reads (shorter than *width*+*k*-1) to ensure that each remaining read contains at least one minimizer. Given an edge in the graph *CDB*(*Reads,k*), we classify the *k*-mers starting at positions *width,* 2 · *width*, 3 · *width*, etc. in the label of this edge as the *padded* *k*-mers. jumboDB uses the padded *k*-mers to obtain information about the starting/ending positions of single-edge paths in the compressed de Bruijn graph (each such path traverses at least one padded *k*-mer).

jumboDB combines all padded *k*-mers for all edges and all junctions in *CDB*(*Reads,k*) into a single set and traverses all *k*-mers in each read in *Reads* to find out how this read traverses all vertices in this combined set. Using this information, it generates the read-path for each read and identifies the starting/ ending position of each read within the starting/ending edge of a read-path.

Appendix 3: Information about benchmarking datasets

dataset	#reads	median read length (kbp)	coverage	Genome length (Mb)
ECOLI	95,514	14.5	400	3.5
T2T	5,567,158	17.2	32	3100
T2TErrorFree	5,567,034	17.2	32	3100
chrX	272,732	17.2	32	154
cenX/cenXErrorFree	6,527	17.3	35	3.3
cen6/cen6ErrorFree	11,409	16,8	60	3.4

Supplementary Table Datasets. Information about read-sets used for benchmarking jumboDB. The ECOLI dataset is available from the SRA database (accession number SRR10971019). The T2T dataset is available at https://github.com/nanopore-wgs-consortium/chm13. The total length of *HumanGenome* after compressing all homopolymer runs is 2,133,004,165. The chrX dataset was generated by mapping the T2T dataset to *HumanGenome* using Winnowmap (Jain, 2020) and selecting reads that mapped to chrX. In rare cases when a read maps to multiple nearly identical instances of a repeat, Winnowmap

outputs both *primary* and *secondary* read alignments. Although using primary alignments works well for a vast majority of regions in the human genome, primary alignments incorrectly map some reads in the most repetitive regions such as cen6, result in low coverage of some repeat instances, and thus negatively affect the generation of datasets containing error-free reads. We thus used both primary and secondary alignments in such regions, e.g., for generating the Cen6ErrorFree dataset.

Appendix 4: Error correction of HiFi reads

Error-correcting reads by read-rerouting. Two paths in a graph are called *compatible* if they both start in the same vertex and both end in the same vertex. Given compatible paths *P* and Q** and a path *P* that contains *P** as a subpath, the (*P**,*Q**)-*rerouting* substitutes *P* by a new path where the subpath *P** of *P* is substituted by *Q*.*

At each step of its error-correction procedure, LJA selects a low-coverage subpath *P** of a read-path *P* in the graph *CDB*(*Reads,k*) and attempts to find a read-path *Q* that contains a higher-coverage *P**-compatible subpath *Q*.* Afterward, it corrects errors in *P* by performing the (*P**,*Q**)-rerouting of *P.* After this (*P**,Q*)-rerouting, it reduces (increases) the coverage of all edges in the subpath *P** (*Q**) by 1. Below we describe how LJA searches for candidates for read-rerouting.

Analyzing low-coverage paths in the compressed de Bruijn graph. We denote the edit distance between strings *v* and *w* as *distance*(*v,w*). We classify strings *v* and *w* as *similar* if the edit distance between them does not exceed *fraction**min{|*v*|,|*w*|}, where *fraction* is a parameter (the default value=0.01). We classify strings *v* and *w* as *possibly-similar* if the difference between their lengths does not exceed *fraction**min{|*v*|,|*w*|}. Two compatible subpaths in the compressed de Bruijn graph are *similar* (*possibly-similar*) if they spell similar (possibly-similar) strings.

We classify an edge in a compressed de Bruijn graph as a *low-coverage* edge if its coverage is below a threshold *minCoverage*, and a *high-coverage* edge, otherwise. Appendix 5 describes how LJA selects this threshold (*minCoverage*=2 for the T2T dataset). LJA partitions each read-path *P* into the high-coverage and low-coverage edges (resulting in its partitioning into alternating *low-coverage* and *high-coverage subpaths*) and attempts to error-correct low-coverage subpaths.

When a genome is known, we classify edges in *CDB*(*Reads*,*k*) that are visited by the genome traversal as the *correct* edges. Figure CoverageCorrectEdges illustrates that the vast majority of correct edges are high-coverage edges in the graph *CDB*(*chrX*,511). This graph has only 32 isolated edges that correspond to reads that do not share 511-mers with any other reads because they have a surprisingly high error-rate. In contrast, the graph *CDB*(*chrX*,5001), constructed for a larger *k*-mer size, has many more reads (1454) that correspond to isolated edges. However, after performing error correction and constructing the graph *CDB*(*chrX**,5001) on the error-corrected read-set *chrX**, only 398 reads form isolated edges in this graph and do not share 5001-mers with other reads.

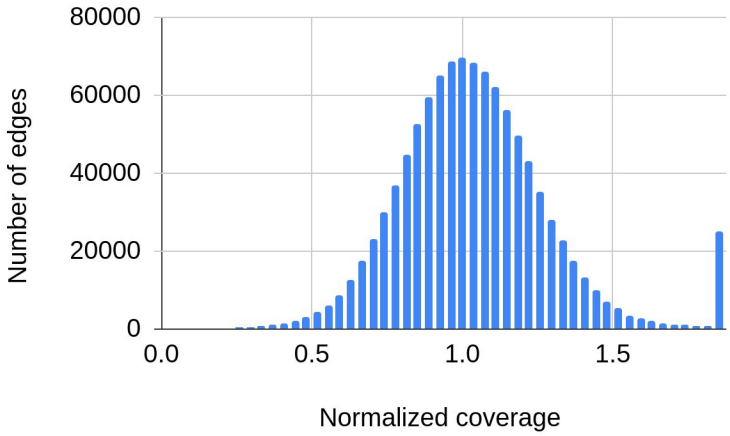

Figure CoverageCorrectEdges. Histogram of the normalized coverage of correct edges in the graph *CDB*(*chrX*,511). The normalized coverage of an edge is defined as the coverage of this edge divided by the median coverage of all edges in the graph. The genome traversal of chrX is formed by 1,180,374 edges in *CDB*(*chrX*,511) (only 56/71/112 of them have a small coverage 1/2/3). Since 6841 511-mers from the assembled chrX do not appear in reads, the genome traversal of chrX in *CDB*(*chrX*,511) consists of 20 paths rather than a single path in *CDB*(*chrX*,511).

When a genome is known, one can align each read (corresponding to a read-path P in *CDB*(*Reads*,*k*)) to the genome and identify the genomic segment spanned by this read. This genomic segment typically corresponds to a high-coverage path $\mathbb{P}$ in *CDB*(*Reads*,*k*) that however may differ from P. An edge in a read-path P is called *correct* if it is also an edge of $\mathbb{P}$ (and the corresponding edges are aligned against each other in the read-genome alignment), and *incorrect*, otherwise. The correct and incorrect edges partition the read-path P into the correct and incorrect subpaths. Given an incorrect subpath P^* of P, its *valid correction* is defined as substituting this subpath with a subpath of $\mathbb{P}$ that P^* is aligned to (all other corrections are classified as *invalid*) 98.2% of the incorrect subpaths in the chrX dataset are low-coverage subpaths and 99.95% of low-coverage subpaths in the chrX dataset are incorrect subpaths.

A subpath of a path P is called *external* if it contains the first or the last edge of P, and *internal*, otherwise. Figure LowCoverageSubpathLength illustrates that the vast majority of low-coverage internal subpaths in the graph *CDB*(*Reads,k*) spell strings of length close to k. Figure ErrorNumberPerRead provides information about the number of reads with the specified number of error-clusters in the chrX dataset (with reads subjected to the homopolymer collapsing (HPC) procedure) before and after the error correction step. 37% (96%) of the reads in the chrX dataset are error-free before (after) our error correction procedure.

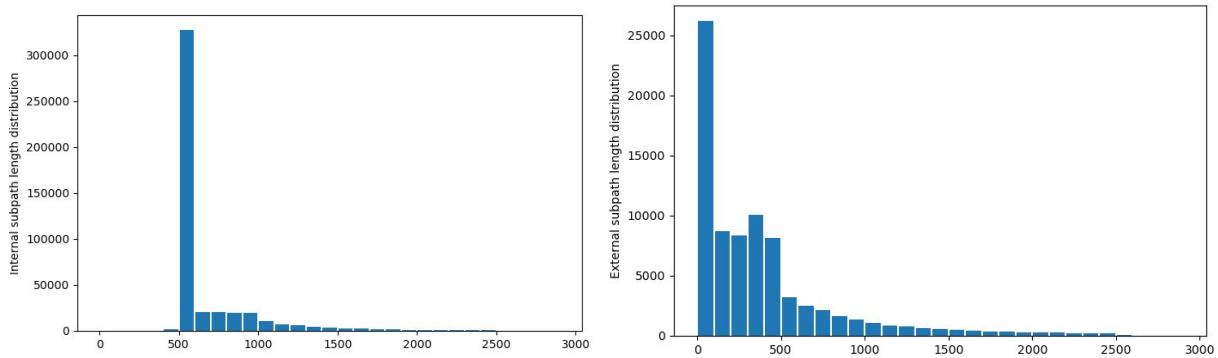

Figure LowCoverageSubpathLength. Distribution of lengths of strings spelled by all 455548 internal (left) and 79398 external (right) low-coverage subpaths of read-paths from the chrX datasets in the graph *CDB*(chrX,511).

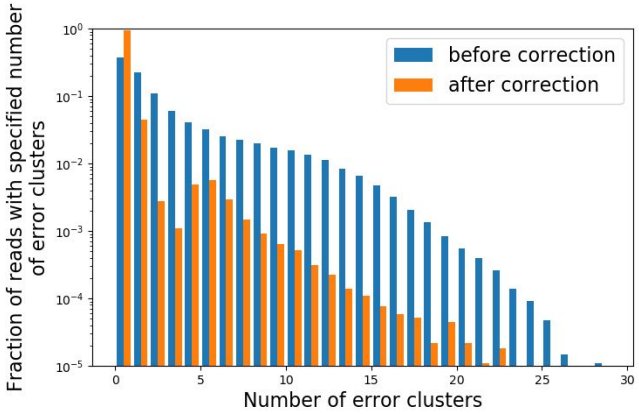

Figure ErrorNumberPerRead. The number of reads with the specified number of error-clusters in the chrX dataset before and after the error correction step. Two errors in a read belong to the same *error-cluster* if they are located close to each other in this read, i.e., within the distance *k* (the histogram is constructed for *k*=511). The x-axis specifies the number of error-clusters and the y-axis shows the fraction of reads with the specified number of error-clusters (in logarithmic scale). Number of error-clusters are given after the homopolymer collapsing (HPC) step. 37% of the reads in the chrX datasets are error-free and 22% have a single error-cluster. After error-correction, 96% of the HPC reads in the chrX dataset are error-free and 3.7% have a single error-cluster.

Below we describe how LJA corrects reads by performing the read-rerouting and bulge-collapsing operations.

Bypasses. We first describe how LJA error-corrects internal low-coverage subpaths by read-rerouting. Subsection “Error-correcting external low-coverage subpaths” describes how it corrects external subpaths.

Given compatible and similar subpaths *P** and *Q** of some read-paths in the compressed de Bruijn graph, *Q** is classified as a *bypass of P** if it is a high-coverage subpath. For each low-coverage internal subpath *P** of a read-path *P,* LJA searches for a read-path *Q* and its subpath *Q** that represents a bypass of *P**.

We classify a low-coverage internal subpath as a *no-bypass, uni-bypass,* or *multi-bypass* if it has no bypasses, a single bypass, and multiple bypasses, respectively. Since classifying bypasses into these three categories may be time-consuming (particularly, in the case of long low-coverage subpaths), LJA uses a slightly different but fast classification of bypasses. Specifically, for each low-coverage internal subpath *P** that starts at a vertex *source* and ends at a vertex *sink*, LJA considers all read-paths that traverse both *source* and *sink* and identifies all possibly-similar subpaths of these read-paths that are compatible with *P**. If there are no high-coverage subpaths in this set, *P** is classified as a no-bypass. If all high-coverage subpaths in this set are identical (resulting in a subpath *Q**), *P** is classified as a uni-bypass. Otherwise, it is classified as a multi-bypass.

444, 414,399, and 1,314 out of all 416157 low-coverage internal subpaths in the graph *CDB*(chrX,511) are no-bypasses, uni-bypasses, and multi-bypasses, respectively

Rerouting uni-bypasses. Since a uni-bypass *P** has a single bypass *Q**, it is a candidate for a (*P**,*Q**)-rerouting if *P** and *Q** are similar. However, jumboDB skips the time-consuming similarity check since possibly-similar subpaths turned out to be similar in the vast majority of cases. After rerouting all internal uni-bypasses, 76% of reads in the chrX dataset become error-free. Only 1522 out of 414,399 internal uni-bypasses (0.35%) resulted in invalid re-routings. We note that an invalid re-routing does not necessarily lead to an error in the final assembly since it can be corrected at the follow-up error-correction steps.

Rerouting multi-bypasses. Given a multi-bypass *P**, LJA computes its edit distance with each its bypass to identify the closest bypass *Q1**. Although in the vast majority of cases *Q1** is a valid correction of *P**, in rare cases it is not. To detect cases where *Q1** is not a valid correction of *P*, LJA* performs an additional *triangle* test inspired by a similar test in the mosaicFlye assembler (Bankevich and Pevzner, 2020). For each bypass *Q2** different from *Q1**, it tests if *distance*(*P*,Q2**)=*distance*(*P*,Q1**)+*distance*(*Q1*,Q2*).* If this triangle test holds, it performs the (*P**,*Q1**)-rerouting of the multi-bypass *P**. In most cases, the triangle test is equivalent to checking whether the edit operations to transform *P** into *Q1** represent a subset of edit operations to transform *P** into *Q2*,* a rather strong condition. The triangle condition leads to correcting 1790 out of 2035 multi-bypasses (1511 of them represent valid corrections).

Error-correcting external low-coverage subpaths. The simplest way to deal with external low-coverage subpaths is to simply shorten each read-path by deleting its low-coverage prefix and/or suffix. However, as Figure LowCoverageSubpathLength illustrates, this procedure significantly reduces the length of some read-paths and thus may negatively affect the ability of the multiplex de Bruijn graph to resolve some repeats.

LJA considers each low-coverage external subpath *P** and attempts to error-correct a read-path *P* (that contains this subpath) by substituting *P** with a high-coverage subpath in *CDB*(*Reads,k*). For simplicity, below we consider *ending* external subpaths that end in the last edge of a read (*starting* external subpaths that start in the first edge of a read are analyzed similarly).

Given an ending subpath *P**, LJA considers all reads passing through its first vertex and identifies their high-coverage subpaths that start at this vertex. For each such subpath, it identifies its prefix that has the lowest edit distance from *P** and analyzes the set of the identified similar prefixes. Similar to the classification of each internal low-coverage subpath into a no-bypass, uni-bypass, or multi-bypass, LJA classifies each ending low-coverage subpath into a *no-suffix*, *uni-suffix*, or *multi-suffix* and error-corrects it using the previously described algorithm for three types of bypasses of internal subpaths, but this time applied to three types of ending (external) subpaths.

This procedure results in the rerouting of 79150 out of 79398 external low-coverage subpaths. LJA removes the remaining 248 low-coverage external subpaths, thus shortening the corresponding reads.

Error-correction by bulge-collapsing. We refer to all parallel edges between vertices *v* and *w* in the graph *CDB*(*Reads,k*) as a *bulge* (denoted as *bulge*(*v,w*)). An edge in a bulge with the highest coverage is referred to as a *heavy edge* and all other edges are referred to as *light edges*. We classify a bulge as *collapsible* if the total coverage of its edges does not exceed *coverageAmplifier* · *medianCoverage*, where *coverageAmplifier* is a parameter (the default value 1.5) and *medianCoverage* stands for the median coverage of all edges in the graph. An edge in a bulge is *correct* if it represents a substring of the genome and erroneous, otherwise. A

collapsible bulge is *reducible* if it has a single correct edge and *foolproof* if this single edge is heavy.

Bulge collapsing refers to a procedure that removes all light edges in a bulge and reroutes all read-paths containing these edges through a heavy edge of this bulge. Although the graph *CDB*(*chrX,*511) has no bulges, the graph *CDB*(*chrX*,*511) on the error-corrected read-set *chrX** has 144 bulges and 140 of them are collapsible. Moreover, 128 (124) out of 140 collapsible bulges in *CDB*(*chrX*,*511) are reducible (foolproof). Since the vast majority of collapsible bulges *CDB*(*chrX*,*511) are foolproof, LJA iteratively collapses all collapsible bulges until no collapsible bulges are left.

An additional round of error-correction with larger *k*-mer sizes. After read-rerouting and bulge-collapsing in the graph *CDB*(*Reads,*511), 92.5% of reads in the error corrected read-set have perfect alignment to the homopolymer-compressed reference genome and the error rate in reads is reduced to 36 errors per megabase. The remaining errors are often supported by multiple reads and thus are difficult to distinguish from genomic variations. To further reduce the error rate, after correcting errors using the compressed de Bruijn graph for a relatively small *k*-mer size (e.g., *k*=511), jumboDB switches to the compressed de Bruijn graph for a larger *k*-mer size (e.g., *k*=5001) and performs one more round of error-correction. The rationale for this additional round of error-correction is that it becomes less likely for the same error to be supported by large *k*-mers than by smaller *k*-mers. Indeed, the graph *CDB*(*chrX*,*5001) has 750 bulges and 744 of them are collapsible. 738 (536) out of 744 collapsible bulges in *CDB*(*chrX*,*5001) are reducible (foolproof) bulges. After the rerouting procedure (*k*=5001 and *minCoverage*=2) corrects 7990 reads we are left with 484 no-bypasses and no multi-bypasses.

After two rounds of error-correction, the initial chrX read-set is transformed into the error-corrected read-set chrX*, where 96% of reads are error-free and the error rate in reads is reduced to only 3.6 errors per megabase. However, the de Bruijn graph constructed from error-corrected reads for *k*=5001 does not contain low-coverage edges, suggesting that the remaining 4% of reads are corrected consistently with each other even though they do not perfectly match the reference genome due to either heterozygous sites or *read corruption*. The read corruption refers to incorrect correction of reads spanning one repeat copy to match another slightly different repeat copy. We emphasize that read corruption does not necessarily lead to errors in the final assembly since all original reads are realigned against the assembly at the final consensus step that often fixes differences introduced by the corrupted reads.

The compressed de Bruijn graph on error-corrected reads. After read-routing and bulge-collapsing, we consider all 1395 *uncorrected reads* that still contain uncorrected bypasses and break each such read into shorter reads by removing each uncorrected bypass. Figure CoverageDrop analyzes the “survival” of genomic *k*-mers in the read-set and illustrates that breaking even a small number of reads may have a negative effect of breaking the compressed de Bruijn graph into multiple components. It also illustrates that our switch from a small *k*-mer size (511) to a much larger *k*-mer size (5001) may be somewhat too-aggressive since some genomic segments of length 5001 are not spanned by any reads in the chrX dataset (let alone, by reads after the read-breaking procedure). We are currently working on optimizing the parameters of error correction (e.g., a switch from *k*=511 to *k*=2501 instead of *k*=5001 appears to be a safer strategy) and minimizing the negative effects of the read-breakage procedure.

After all error-correction steps, the compressed de Bruijn graph on 5001-mers consists of 95 vertices and 120 edges and is broken into 19 connected components, indicating breaks in coverage.

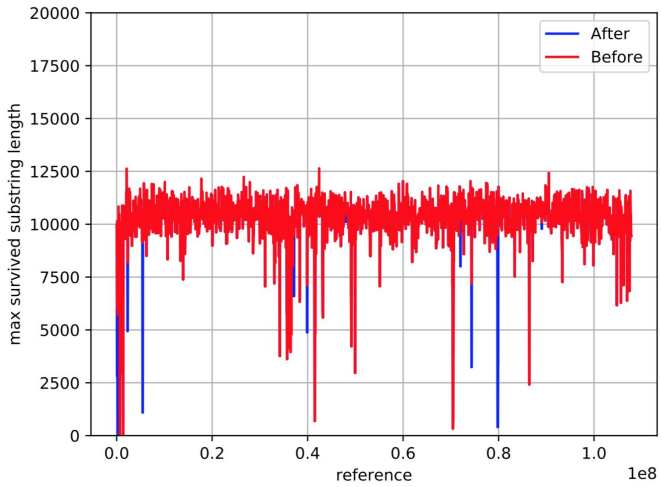

Figure CoverageDrop. Effect of error correction on the survival of genomic *k*-mers in the read-set. For each position in chrX, among all reads that cover this position, we select the one with the longest suffix that starts at this position and compute the length of this suffix. The red (blue) curve corresponds to this metric computed for the reads before (after) error correction. Since reads contain sequencing errors, this metric at each position of chrX is an upper bound on the length of the longest surviving *k*-mer in the read-set. Afterward, we compute the minimum of this function in each window of length 100 kb centered at each position.

Appendix 5: Selecting the minimum coverage threshold

To set the threshold *minCoverage*, we select a long contig (that we refer to as *Genome*) that is represented by a single copy in the genome. We denote the set of all *k*-mers in *Genome* (referred to as *genomic k*-mers) as *Kmers*(*Genome,k*). We identify all reads (referred to as *Reads*) that originated from the selected contig and compute the set of *high-coverage k*-mers $Kmers_{Coverage}(Reads,k)$ as the set of all *k*-mers in *Reads** with coverage exceeding a threshold *Coverage*.

In the case when the threshold *Coverage* perfectly separates correct and incorrect *k*-mers from *Genome,* the set of genomic *k*-mers coincides with the set of high-coverage *k*-mers in *Reads,* i.e., $Kmers_{Coverage}(Reads,k)=Kmers(Genome,k)$. We define false positives as high-coverage *k*-mers that do not appear in the genome:

$$FalsePositives_{Coverage}(Reads,Genome,k)=|Kmers_{Coverage}(Reads,k))\setminus Kmers(Genome,k)|$$

and false negatives as genomic *k*-mers that have low-coverage:

$$FalseNegatives_{Coverage}(Reads,Genome,k)=|Kmers(Genome,k)\setminus Kmers_{Coverage}(Reads,k))|$$

Figure FalsePositives/Negatives shows the distribution of $FalsePositives_{Coverage}(T2T,HumanGenome,k)$ and $FalseNegatives_{Coverage}(T2T,HumanGenome,k)$ for varying values of *Coverage* and for *k*=511. Since our error-correction procedure works well with respect to removing false positive edges and is less efficient with respect to restoring false negative edges, we select a rather low minimum coverage threshold to minimize the number of false-negative edges. E.g., when *minCoverage* equals 2, 3, and 4, the number of false

negatives (false positives) equals 0.6, 0.8, 1.0 (101, 44, 27) millions, respectively. jumboDB sets *minCoverage* as 0.1**medianCoverage*.

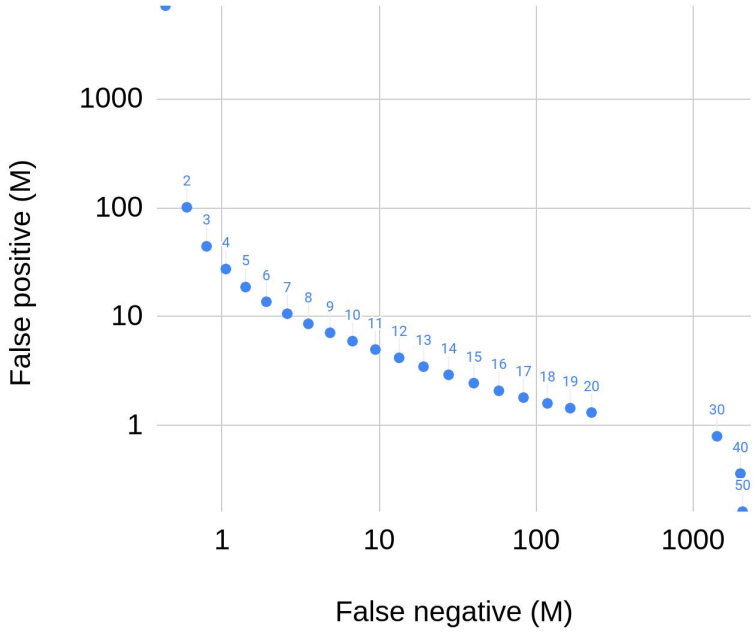

Figure FalsePositives/Negative. The number $FalsePositives_{Coverage}$(*T2T,HumanGenome,k*) (x-axis) and $FalseNegatives_{Coverage}$(*T2T,HumanGenome,k*) (y-axis) for varying values of *Coverage* and for *k*-mer size 511 (in millions). The numbers were computed for the entire *HumanGenome* and are given in the logarithmic scale.

Appendix 6: Limitations of the multiplex graph transformation procedure

Figure ThreeRepeats shows a circular genome ARDARCBRCE that traverses the repeat R three times via subpaths ARD, ARC, and BRC and an incomplete read-set that supports only two of these three subpaths (e.g., does not support ARC). This example illustrates the case when a graph transformation results in a fragmented assembly since the multiplex de Bruijn graph of reads "loses" the genome traversal. Indeed, after a series of transformations, when the *k*-mer size becomes equal to the length of the repeat R, this repeat will be transformed into a single "red" vertex that will be classified as paired because each incoming edge into this vertex is paired with an outgoing edge from this vertex (and vice versa).

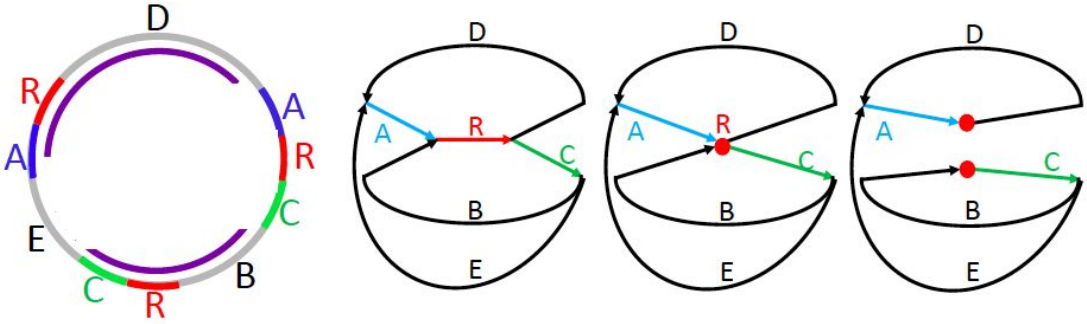

Figure ThreeRepeats. A circular genome ARDARCBRCE (first panel), its compressed de Bruijn graph (second panel), its multiplexed de Bruijn graph after the edge R is transformed into a paired vertex (third panel), and its multiplexed de Bruijn graph after transforming this paired vertex (fourth panel). The read-set includes each two-edge path corresponding to a pair of consecutive edges in the genome as well as the three-edge paths ARD and BRC shown in purple. However, it does not include the three-edge path ARC. After the repeat R is transformed into a single vertex, this vertex is classified as paired since each incoming edge is paired with an outgoing edge and vice versa (A is paired with D and B is paired with C). Transforming this vertex results in a multiplex de Bruijn graph that does not adequately represent the genome since it "loses" the genome traversal.

Another limitation of the multiplex graph transformation procedure is illustrated by an example of a linear genome ARBRC and an incomplete read-set that contains a read ARB spanning one instance of the repeat R but does not contain a read BRC spanning another instance of this repeat. In this case, after a series of transformations, when the *k*-mer size becomes equal to the length of the repeat R, this repeat will be transformed into an unpaired vertex that has to be frozen. However, an addition of a virtual read BRC (when it is relatively "safe") reclassifies this vertex as paired and enables a graph transformation at this vertex.

Appendices 7 and 8 describe how to modify the multiplex graph transformation algorithm so that it freezes some paired vertices (to address the complication shown in Figure ThreeRepeats) and transforms some unpaired vertices (to address the complication described in the above paragraph).

Appendix 7: Adding virtual reads to avoid overly-optimistic graph transformations

The *multiplicity* of an edge in the graph *CDB*(*Genome,k*) is defined as the number of times the genome traversal visits this edge. We assume that each edge in the constructed graph *jumboDB*(*Reads,k*) corresponds to an edge in an (unknown) graph *CDB*(*Genome,k*) and attempt to assign multiplicities to edges of *jumboDB*(*Reads,k*) that approximate multiplicities of the corresponding edges in *CDB*(*Genome,k*). Below we assume that the multiplicity of each edge in the graph *jumboDB*(*Reads,k*) is given (see subsection "Analyzing variations in the *k*-mer coverage") and that the total multiplicity of all incoming edges into each vertex (except for dead-ends) equals the total multiplicity of all outgoing edges from this vertex.

Given a path-set *Paths*, *the in-flow through an edge* (*v,w*) is defined as the number of pairs of edges (*v,w*) and (*w,u*) that form a transition in *Transitions*(*Paths*). Similarly, *the out-flow through an edge* (*w,u*) is defined as the number of pairs of edges (*v,w*) and (*w,u*) that form a transition in *Transitions*(*Paths*). An incoming edge into (outgoing edge from) a vertex *w* is called *balanced* if its multiplicity equals the in-flow (out-flow) through this edge, and *unbalanced*, otherwise. A vertex *w* is called *balanced* if all its incoming/outgoing edges are balanced, and *unbalanced* otherwise.

If the edge multiplicities were known for the example shown in Figure ThreeRepeats, an edge A entering the vertex R and the edge C leaving the vertex R would be classified as unbalanced. Therefore, the missing transition through the unbalanced vertex R can be restored by simply adding a virtual read AC to the read-set and making the vertex R balanced.

We thus find all unbalanced vertices and analyze all incoming/outgoing unbalanced edges for these vertices. If an unbalanced vertex has a single incoming unbalanced edge *e* (or a single outgoing unbalanced edge *e*), we turn it into a balanced vertex by adding virtual reads that connect the edge *e* with other unbalanced edges for this vertex. These virtual reads lead to an increased number of resolved vertices and thus reduce the number of frozen vertices in the multiplex de Bruijn graph.

The concept of a balanced vertex assumes that we can accurately compute the multiplicity of edges in the compressed de Bruijn graph, a difficult task. Previous studies addressed this problem by using both the graph topology and the coverage of edges by reads (Pevzner and Tang, 2001, Kolmogorov et al., 2019). Appendix 9 provides statistics on variations in the *k*-mer coverage by HiFi reads.

Appendix 8: Freezing vertices to avoid overly-optimistic graph transformations

To evaluate how often the missing transitions trigger overly-optimistic graph transformations (Figure ThreeRepeats), we collected the following statistics. Given a read-set generated from a known genome and a parameter *k*, we compute $coverage_k(i)$ – the number of mapped reads in the read-set that bridge a *k*-nucleotide long segment starting at position *i* in the genome. Positions with $coverage_k(i)=0$ model a situation when a *k*-mer in a genome is not spanned by any reads. We thus compute the *non-spanning probability p(k)* that a randomly selected *k*-mer in a genome has no spanning reads and assume that it provides a good approximation of the probability that an instance of a *k*-nucleotide long repeat is not bridged by any reads (Figure NonSpanningProbability). We use these probabilities to estimate the risk that increasing the *k*-mer-size during the multiplex graph transformations will lead to a situation illustrated in Figure ThreeRepeats.

Using the precomputed non-spanning probabilities for all values of *k*, we classify a vertex in the graph *CDB*(*Reads,k*) as *weakly resolved* if *p(k)* exceeds *non-spanningThreshold* (the default value 0.001), i.e., if the risk of missing transitions in this vertex is high. We modify the graph transformation algorithm to freeze (rather than resolve) all weakly resolved unbalanced vertices to avoid the risk of getting into a situation illustrated in Figure ThreeRepeats.

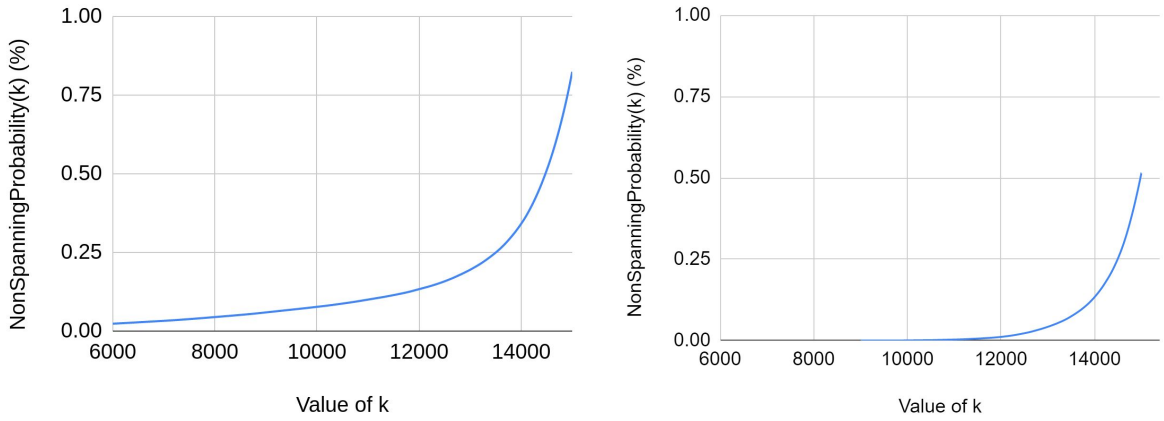

Figure NonSpanningProbability. The estimated non-spanning probability *p(k)* for the T2T read-set (for *k* from 9000 to 15000) based on a segment from chromosome 6 (left) and the entire chromosome X (right). (Left) The spanning probability is computed for a 55-Mb segment of chromosome 6 that does not contain the centromere (positions from 0 to 55 Mb). The spanning probabilities for *k* below 9000 are zeros (all *k*-mers are spanned) but grow fast with increasing the *k*-mer size (p(10000)=0.03%, p(11000)=0.2%, p(12000)=1.1%). (Right) The spanning probabilities computed for the entire chromosome X (that contains the centromere) are significantly higher, likely because some reads are misaligned in highly repetitive regions such as the centromere. The spanning probability grows fast with increasing the *k*-mer size (p(10000)=0.08%, p(11000)=0.10%, p(12000)=0.13%).

Appendix 9: Analyzing variations in the *k*-mer coverage by reads

To analyze variations in coverage, we define the *normalized coverage* of each *k*-mer in the graph *jumboDB*(*Reads,k*) as its coverage by reads divided over the average coverage across all *k*-mers in the genome. The function $cov_k(x)$ is defined as the fraction of *k*-mers with normalized coverage below *x*. We further define the *normalized coverage* of an edge in the graph *jumboDB*(*Reads,k*) as its normalized coverage by reads divided over the average normalized coverage across all *k*-mers in the genome. The function $edge\text{-}cov_k(x)$ is defined as the fraction of edges in *jumboDB*(*Reads,k*) with normalized coverage below *x*. For the T2T read-set, $cov_{500}(0.5)=0.006$, $cov_{500}(1.5)=0.98$, $cov(0.5)=0.01$, and $cov(1.5)=0.83$.

Figure CoverageFunctions illustrates that the coverage-based estimates of multiplicity become more reliable when they are computed for smaller values of *k*. We thus estimate the edge multiplicities for the initially constructed graph *jumboDB*(*Reads,k*) for small *k*-mer size and use them to propagate multiplicities of edges for larger values of *k* during graph transformations.

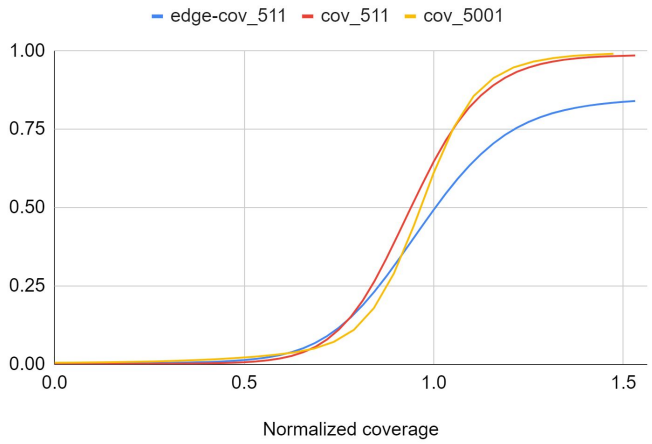

Figure CoverageFunctions. Functions $cov_{511}(x)$ and $cov_{5001}(x)$, as well as the function $edge\text{-}cov_{511}(x)$, for the T2T read-set. The average coverage across all 511-mers (5001-mers) in *HumanGenome* is equal to 32 (19).

We further mapped each edge *e* in the graph *jumboDB*(T2T,*k*) to a most similar edge *e'* in the graph *CDB*(*HumanGenome*,*k*) and computed the ratio of the normalized coverage of *e* and the multiplicity of *e'*. Figure Multiplicity shows the distribution of this value over all edges of the graph *jumboDB*(T2T,*k*).

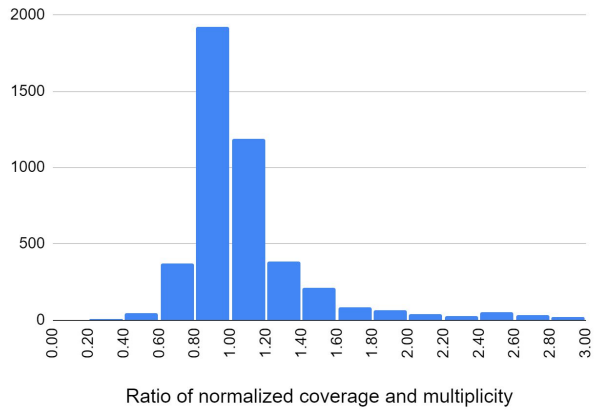

Figure Multiplicity. Comparing the normalized coverage of edges in *jumboDB*(*T2T,511*) with the multiplicities of edges in *CDB*(*HumanGenome,511*). The histogram is generated for edges of length at least 5000 bp as the coverage estimates for shorter edges are less reliable.