nature biotechnology
ARTICLES
https://doi.org/10.1038/s41587-022-01220-6

Multiplex de Bruijn graphs enable genome assembly from long, high-fidelity reads

Anton Bankevich[1]✉, Andrey V. Bzikadze[2], Mikhail Kolmogorov[3], Dmitry Antipov[4] and Pavel A. Pevzner[1]✉

Although most existing genome assemblers are based on de Bruijn graphs, the construction of these graphs for large genomes and large *k*-mer sizes has remained elusive. This algorithmic challenge has become particularly pressing with the emergence of long, high-fidelity (HiFi) reads that have been recently used to generate a semi-manual telomere-to-telomere assembly of the human genome. To enable automated assemblies of long, HiFi reads, we present the La Jolla Assembler (LJA), a fast algorithm using the Bloom filter, sparse de Bruijn graphs and disjointig generation. LJA reduces the error rate in HiFi reads by three orders of magnitude, constructs the de Bruijn graph for large genomes and large *k*-mer sizes and transforms it into a multiplex de Bruijn graph with varying *k*-mer sizes. Compared to state-of-the-art assemblers, our algorithm not only achieves five-fold fewer misassemblies but also generates more contiguous assemblies. We demonstrate the utility of LJA via the automated assembly of a human genome that completely assembled six chromosomes.

The emergence of long and accurate reads opened a possibility to generate the first complete (telomere-to-telomere) assembly of a human genome and to get a glimpse into biomedically important genomic regions that evaded all previous attempts to sequence them[1]. The Telomere-to-Telomere (T2T) and Human Pangenome Reference projects are now using long and accurate reads for population-scale assembly of multiple human genomes and for diagnosing rare diseases that remained below the radar of short-read technologies[2].

These breakthroughs in genome sequencing were mainly achieved using HiFi reads[3]. However, assembly of HiFi reads is far from being straightforward: the complete assembly of a human genome was generated using a semi-manual effort of a large consortium rather than by an automated approach (Nurk et al., 2021). Because such time-consuming efforts are neither sustainable nor scalable in the era of population-scale sequencing, there is a need for an accurate (nearly error-free) tool for complete genome assembly.

We argue that this challenge requires an algorithm for constructing large de Bruijn graphs[4]—that is, de Bruijn graphs for large genomes and large *k*-mer sizes exceeding 1,000 nucleotides. Indeed, similarly to assembling short and accurate reads, the de Bruijn graph approach has the potential to improve assemblies of any type of accurate reads. However, although it represents the algorithmic engine of nearly all short-read assemblers[5,6], the problem of constructing large de Bruijn graphs remains open, and the existing HiFi assemblers HiCanu[7] and hifiasm[8] are based on the alternative *string graph* approach[9].

Because HiFi reads are even more accurate than Illumina reads, the de Bruijn graph approach is expected to work well for their assembly. Application of this approach to long HiFi reads requires either constructing the de Bruijn graph with a large *k*-mer size or, alternatively, using the de Bruijn graph with a small *k*-mer-size for follow-up repeat resolution by threading long reads through this graph. However, it remains unclear how to address three open algorithmic problems in assembling HiFi reads: (1) constructing large de Bruijn graphs, (2) error-correcting HiFi reads so that they become nearly error-free and thus amenable to applying the de Bruijn graph approach and (3) using the entire read-length for resolving repeats that are longer than the *k*-mer size.

The existing genome assemblers are not designed for constructing large de Bruijn graphs because their time/memory requirements become prohibitive when the *k*-mer size becomes large—for example, simply storing all 5,001-mers of the human genome requires ≅4 TB. For example, the SPAdes assembler[5] faces time/memory bottlenecks assembling mammalian genomes with the *k*-mer size exceeding 500. To reduce the memory, some assembly algorithms avoid explicitly storing all k-mers by constructing a *perfect hash map*[5] or the *Burrows–Wheeler transform* of all reads[10]. However, even with these improvements, the runtime (proportional to the *k*-mer size) remains large (Supplementary Note 1).

The *repeat graph* approach[11] and the *sparse de Bruijn graph* approach[12] construct coarse versions of the de Bruijn graph with smaller time/memory requirements. Recently, ref. [13] modified the Flye assembler for constructing the repeat graph of HiFi reads, and ref. [14] showed how to assemble HiFi reads into a sparse de Bruijn graph. However, these graphs represent coarse versions of the de Bruijn graph, thus limiting their capabilities in assembling the highly repetitive regions such as centromeres (Supplementary Note 2).

Here we introduce LJA, which includes three modules addressing all three challenges in assembling HiFi reads: jumboDBG (constructing large de Bruijn graphs), mowerDBG (error-correcting reads) and multiplexDBG (using the entire read-length for resolving repeats). jumboDBG combines four algorithmic ideas: the *Bloom filter*[15], the *rolling hash*[16], the sparse de Bruijn graph[12] and the *disjointig* generation[17]. Although each of these ideas was used in previous bioinformatics studies, jumboDBG is the first approach that combines them. LJA launches jumboDBG to construct the de

[1]Department of Computer Science and Engineering, University of California, San Diego, San Diego CA, USA. [2]Program in Bioinformatics and Systems Biology, University of California, San Diego, San Diego CA, USA. [3]Department of Biomolecular Engineering, University of California, Santa Cruz, Santa Cruz CA, USA. [4]Center for Algorithmic Biotechnology, Institute for Translational Biomedicine, Saint Petersburg State University, Saint Petersburg, Russia. ✉e-mail: abankevich@eng.ucsd.edu; ppevzner@ucsd.edu

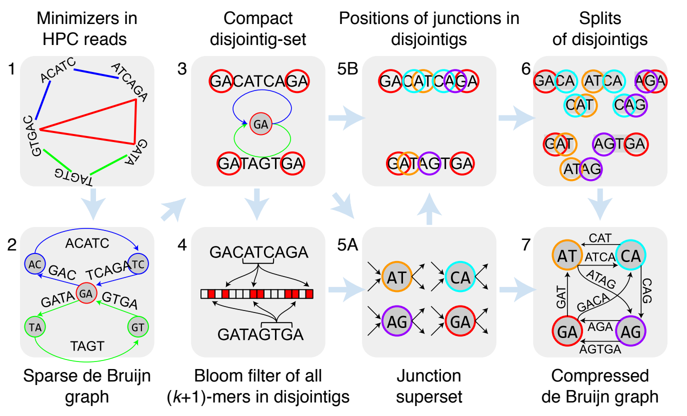

Fig. 1 | JumboDBG pipeline. (1) Generating the anchor-set *Anchors* = {GA,AC,TC,TA,GT} by finding all minimizers in *Reads*. For simplicity, the figure does not reflect that jumboDBG classifies all k-prefixes and k-suffixes of reads as minimizers. (2) Constructing a compact sparse de Bruijn graph *SDB*(*Reads*,*Anchors*). (3) Constructing a compact disjointig-set *Disjointigs* as edge-labels in *SDB*(*Reads*,*Anchors*). (4) Generating the Bloom filter for all $(k+1)$-mers in disjointigs. Each arrow directed from a $(k+1)$-mer to the Bloom filter illustrates its evaluation by one of the hash functions. (5) Using the Bloom filter to construct the junction-superset *Junctions*$^+$ and (5A) find positions of k-mers from *Junctions*$^+$ in disjointigs (5B). For simplicity, although the Bloom filter might generate some false-positive junctions (for example, TC), we only show the correct junctions in 5A. (6) Breaking disjointigs into splits. (7) Constructing the compressed de Bruijn graph *DB*(*Disjointigs*,k)=*DB*(*Reads*,k).

Bruijn graph, launches mowerDBG that uses this graph to correct nearly all errors in reads, launches jumboDBG again to generate a much simpler graph of the error-corrected reads and launches multiplexDBG to transform it into the *multiplex de Bruijn graph* with varying k-mer sizes to take advantage of the entire read-lengths. LJA also includes the LJApolish module that expands the collapsed homopolymer runs in the resulting assembly.

Although we benchmarked LJA, hifiasm and HiCanu on various genomes, evaluating the quality of the resulting assemblies is challenging because neither the complete reference for these genomes nor an automated pipeline for a reference-grade assembly validation are available yet[18]. We thus focused on benchmarking these assemblers using the HiFi read-set (referred to as the T2T dataset) from a haploid human CHM13 cell line assembled by the T2T consortium (Nurk et al., 2021). This painstakingly validated assembly represents the only accurate telomere-to-telomere sequence of a large genome available today. LJA generated the most contiguous assembly of this dataset (including complete assemblies of six human chromosomes without any misassemblies and only ten misassemblies for the entire human genome), reducing the number of assembly errors five-fold as compared to hifiasm and HiCanu. The accuracy of genome assemblers becomes critical in the era of population-wide complete genome sequencing because semi-manual validation of complete genome assemblies[18] is prohibitively time-consuming.

Results

Key algorithmic concepts used in the LJA pipeline. The goal of genome assembly is to reconstruct a genome from its error-prone fragments (*reads*). Given a string-set *Reads* and an integer k, the (uncompressed) de Bruijn graph *UDB*(*Reads*,k) is a directed graph where each vertex is a k-mer from *Reads*, and each $(k+1)$-mer $a_1a_2\ldots a_k\ a_{k+1}$ in reads corresponds to an edge connecting vertices $a_1a_2\ldots a_k$ and $a_2\ldots a_k\ a_{k+1}$. The uncompressed de Bruijn graph of a genome *UDB*(*Genome*,k) is defined by considering each chromosome in *Genome* as a single 'read'. We refer to an error-free read-set *Reads* that contains all $(k+1)$-mers from a string-set *Genome* as a *k-complete* read-set and note that *UDB*(*Genome*,k)=*UDB*(*Reads*,k) for a k-complete read-set.

A vertex with the indegree N and the outdegree M is referred to as an *N-in-M-out vertex*. A vertex is *non-branching* if it is a one-in-one-out vertex and a *junction* otherwise. We refer to the set of all junctions (k-mers) in the graph *UDB*(*Reads*,k) as *Junctions*(*Reads*,k). A path between junctions is *non-branching* if all its intermediate vertices are non-branching. A set of k-mers from *Reads* forms a *junction-superset* if it contains all junctions in *UDB*(*Reads*,k).

The *compressed de Bruijn graph DB*(*Reads*,k) is a memory-efficient version of the uncompressed graph *UDB*(*Reads*,k) where each non-branching path is compressed into an appropriately labeled single edge (see Supplementary Notes 2 and 3 for the precise definition and the summary of terms used in this paper). Because LJA uses compressed de Bruijn graphs, we refer to them simply as the 'de Bruijn graphs' or *DB-graphs*. The *coverage* of an edge in *UDB*(*Reads*,k) is number of times the label of this edge occurs in *Reads*. The *coverage* of an edge in *DB*(*Reads*,k) is the average coverage of all edges in the non-branching path that was compressed into this edge.

The challenge of constructing a large de Bruijn graph. Because the compressed DB-graph *DB*(*Genome*,k) does not require storing all k-mers, the total length of all its edge-labels is up to k times smaller than the total length of all edge-labels in *UDB*(*Genome*,k). The traditional assembly approach constructs *UDB*(*Reads*,k) first and transforms it into *DB*(*Reads*,k). Because this approach is impractical for large genomes and large k-mer sizes, jumboDBG constructs *DB*(*Reads*,k) without constructing *UDB*(*Reads*,k).

Even though *DB*(*Reads*,k) is more memory-efficient than *UDB*(*Reads*,k), its direct construction also requires prohibitively large time/memory. jumboDBG thus assembles reads into disjointigs, sequences that are spelled by arbitrary walks through the (unknown) graph *DB*(*Reads*,k). Even in the case of error-free reads, a disjointig might represent a misassembled concatenate of segments from various regions of the genome rather than its contiguous substring[17]. Although switching from reads to misassembled disjointigs might appear reckless, it is an important step because a carefully chosen disjointig-set *Disjointigs* has a much smaller total disjointig-length than the total read-length while resulting in the same DB-graph *DB*(*Disjointigs*,k) as *DB*(*Reads*,k).

Even though constructing *DB*(*Disjointigs*,k) is an easier task than constructing *DB*(*Reads*,k), it still faces the time/memory bottleneck. jumboDBG addresses it by using the Bloom filter, a compact data structure for storing sets. It stores all $(k+1)$-mers from disjointigs in a Bloom filter formed by multiple independent *hash functions*, each mapping a $(k+1)$-mer into a bit array. The Bloom filter reports a *true positive* for all $(k+1)$-mers occurring in disjointigs but might also report a *false positive* for some $(k+1)$-mers that do not occur in disjointigs (with a small controlled probability). However, it never 'forgets' any inserted $(k+1)$-mer and thus never reports a *false negative*.

Outline of the LJA pipeline. Below we outline all steps of the LJA pipeline using the *T2T* dataset of HiFi reads that was semi-manually assembled by the T2T consortium into a sequence *T2TGenome* by integrating information generated by multiple sequencing technologies (CHM13 reference genome version 1.1). All datasets analyzed in this paper are described in Supplementary Note 4.

Figure 1 illustrates the work of the jumboDBG module (steps 1–7 of the LJA pipeline). Figure 2 illustrates the entire LJA pipeline.

Step 0: *Transforming all reads into homopolymer-collapsed reads.* Because errors in the length of *homopolymer runs* represent the

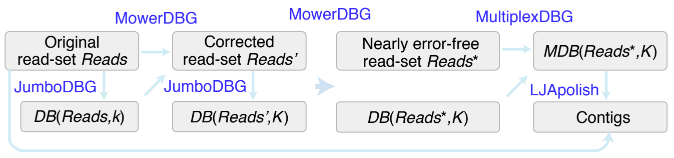

Fig. 2 | LJA pipeline. jumboDBG first constructs the de Bruijn graph *DB*(*Reads*,*k*) with a small *k*-mer size. mowerDBG uses this graph to correct errors in reads, resulting in an error-corrected read-set *Reads*′. Afterwards, jumboDBG constructs the de Bruijn graph *DB*(*Reads*′,*K*) on the error-corrected read-set with a large *K*-mer size. mowerDBG uses this graph to correct even more errors in reads, resulting in an error-corrected read-set *Reads**. Because error correction in mowerDBG simultaneously modifies the graph *DB*(*Reads*′,*K*) into the graph *DB*(*Reads**,*K*), there is no need to launch jumboDBG again for constructing *DB*(*Reads**,*K*). multiplexDBG complements the error-corrected read-set *Reads** by virtual reads and transforms *DB*(*Reads**,*K*) into the multiplex de Bruijn graph *MDB*(*Reads**,*K*). LJApolish uses the set of original reads *Reads* to expand HPC contigs formed by non-branching paths in this graph.

dominant source of errors in HiFi reads, LJA collapses each homopolymer run X…X in each read into a single nucleotide X, resulting in a *homopolymer-collapsed (HPC)* read. The entire LJA pipeline works with HPC reads, except for the last LJApolish module that expands each collapsed nucleotide X in the HPC assembly into a run X…X in such a way that its run-length coincides with the correct run-length in nearly all cases (the error rate in the run-lengths is below 0.00001). On average, uncollapsed reads in the T2T dataset have ≅2,000 errors per megabase of the total read-length. Transforming them into HPC reads reduces the error rate to ≅620 errors per megabase in the HPC genome (when comparing HPC reads to the HPC reference genome) and makes 38% of all HPC reads error-free.

Step 1: *Generating the anchor-set by finding all minimizers in the HPC read-set Reads*. Given a hash function defined on *k*-mers, a *minimizer* of a word is defined as a *k*-mer with a minimal hash in this word. A minimizer-set of a string is defined as the set of all minimizers over all its substrings of length *width*[19]. We modify the original concept of a minimizer of a linear string by adding its prefix and suffix *k*-mers to the set of its minimizers. A sensible choice of the parameter *width* (and a hash function) ensures that each read is densely covered by minimizers and that overlapping reads share minimizers, facilitating the assembly. jumboDBG generates the set of all minimizers in reads that we refer to as the *anchor-set Anchors*.

Step 2: *Constructing a compact sparse de Bruijn graph*. Because the direct construction of *DB*(*Reads*,*k*) faces the time/memory bottleneck, jumboDBG first assembles reads into disjointigs. Although the Flye assembler[17] constructs disjointigs by searching for overlapping reads, it is unclear how to extend this construction to highly repetitive regions (for example, centromeres) that Flye does not adequately reconstruct. Instead, jumboDBG constructs a sparse de Bruijn graph and generates disjointigs in this graph that are also disjointigs in the much larger (but unknown) de Bruijn graph.

Given a set of *k*-mers *Anchors* from a string-set *Reads*, we consider each pair *a* and *a*' of consecutive anchors in each read and generate a substring of the read (called a *split*) that starts at the first nucleotide of *a* and ends at the last nucleotide of *a*'. The resulting set of splits (after collapsing identical splits into a single one) is denoted *Splits*(*Reads*,*Anchors*). The sparse de Bruijn graph *SDB*(*Reads*,*Anchors*) is defined as a graph with the vertex-set *Anchors* and the edge-set *Splits*(*Genome*, *Anchors*). Each string in *Splits*(*Genome*,*Anchors*) represents the label of an edge connecting its *k*-prefix and *k*-suffix in *SDB*(*Reads*,*Anchors*).

A sparse de Bruijn graph *SDB*(*Reads*,*Anchors*) is *compact* if all its vertices (anchors) represent junctions in the graph *DB*(*Reads*,*k*). jumboDBG transforms the initially constructed graph *SDB*(*Reads*,*Anchors*) into a compact sparse de Bruijn graph *SDB*(*Reads*,*Anchors**) to facilitate the construction of a compact disjointig-set (see below).

Step 3: *Constructing a compact disjointig-set*. A disjointig-set is *complete* if its disjointigs contains all $(k+1)$-mers from *Reads*. A disjointig in the sparse de Bruijn graph *SDB*(*Reads*,*Anchors*) is *compact* if its *k*-suffix and *k*-prefix are both junctions in *DB*(*Reads*,*k*). A complete disjointig-set is *compact* if each disjointig in this set is compact. Because the set of all $(k+1)$-mers in a complete disjointig-set coincides with the set of all $(k+1)$-mers in reads, *DB*(*Reads*,*k*)=*DB*(*Disjointigs*,*k*) for a complete disjointig-set. Thus, the problem of constructing the compressed de Bruijn graph from reads is reduced to constructing the compressed de Bruijn graph of a complete disjointig-set. However, not every complete disjointig-set enables efficient construction of this graph. Below we show that a compact disjointig-set enables efficient construction of *DB*(*Disjointigs*,*k*). jumboDBG constructs a compact disjointig-set as the set of edge-labels in the compact sparse de Bruijn graph *SDB*(*Reads*,*Anchors**).

Step 4: *Generating the Bloom filter of all* $(k+1)$*-mers in disjointigs*. Even though we have reduced constructing the DB-graph *DB*(*Reads*,*k*) to constructing *DB*(*Disjointigs*,*k*), even this simpler problem faces the time/memory bottleneck. To address it, jumboDBG constructs the Bloom filter[20,21] for storing all $(k+1)$-mers in disjointigs and uses the rolling hash to query them in O(1) instead of O(*k*) time.

Step 5: *Using the Bloom filter to construct the junction-superset*. The Bloom filter enables rapid construction of small junction-superset *Junctions*$^+$ even though the DB-graph of disjointigs has not been constructed yet. To achieve this goal, jumboDBG uses the Bloom filter to compute the upper bound on the indegree and outdegree of each vertex (*k*-mer) in the unknown DB-graph of disjointigs by checking which of its $4+4=8$ forward and backward extensions by a single nucleotide represent $(k+1)$-mers present in the Bloom filter. A *k*-mer is called a *joint* if the upper bounds on either its indegree or outdegree exceed 1. Because each junction is a joint, the set of all joints forms a junction-superset.

Step 6: *Using the junction-superset to break disjointigs into splits*. jumboDBG also uses the Bloom filter to rapidly identify the positions of all *k*-mers from *Junctions*$^+$ in disjointigs and to break disjointigs into splits afterward.

Step 7: *Using splits to construct DB*(*Disjointigs*,*k*)=*DB*(*Reads*,*k*). An *edge-subpartition* of an edge (*v*,*w*), in a graph 'substitutes' it with two edges by 'adding' a vertex *u* in the 'middle' of this edge. A *subpartition* of a graph is defined as a result of a series of edge-subpartitions. As described in the Methods, the string-set *Splits*(*Disjointigs*,*Junctions*$^+$) represents edge-labels of a subpartition of the graph *DB*(*Disjointigs*,*k*). jumboDBG compresses all one-in-one-out vertices in this graph to generate *DB*(*Disjointigs*,*k*)= *DB*(*Reads*,*k*).

Step 8: *Correcting errors in reads with mowerDBG*. Supplementary Note 5 illustrates that jumboDBG generates highly contiguous assemblies of a *k*-complete read-set sampled from *T2TGenome* for large values of *k*—for example, *k*=5,001. However, assembling real reads is challenging because the DB-graph *DB*(*T2T*,*k*) of the T2T read-set is much more complex than the DB-graph *DB*(*T2TGenome*,*k*). mowerDBG uses the DB-graph *DB*(*Reads*,*k*) to correct errors in the read-set *Reads*. LJA performs two rounds of error correction and launches mowerDBG twice, with a small *k*-mer size in the first round, resulting in an error-corrected read-set *Reads*', and a large *K*-mer size in the second round, resulting in a nearly error-free read-set *Reads** (default values *k*=501 and *K*=5,001).

jumboDBG constructs the graph *DB*(*T2T*,*k*) with 33,230,906 edges in only 2.7 h using 54 Gb of memory. However, over 99% of edges in this graph are triggered by errors in reads: if reads in

Table 1 | Benchmarking LJA, hifiasm and HiCanu on the T2T dataset

	IdealAssembler(20,000)	LJA	hifiasm	hiCanu
total length (Mbp)	3,055	**3,050**	3,043	3,331
#contigs / #contigs longer than 50 kb	401/401	665/**194**	**448**/250	1,447/427
#misassemblies*	0	**10**	58	47
#local misassemblies*	0	**46**	54	120
#100%-assembled / #95%-assembled chromosomes	5/5	**6/9**	1/1	1/2
#100%-assembled / #95%-assembled centromeres	5/5	**9/9**	3/3	2/2
genome fraction (%)	100	**99.74**	99.28	99.60
duplication ratio	1	**1.001**	1.003	1.094
# mismatches / #indels per 1 Mbp	0/0	**6.9**/7.8	7.7/**5.7**	23.3/94.9
longest alignment (Mbp)	160.6	**201.1**	181.6	142.3
total aligned length (Mbp)	3055	**3047**	3038	3327
NG50 (Mbp)	93.6	**96.7**	90.2	69.7
NGA50 (Mbp)	93.6	**96.7**	75.1	69.7
NGA75 (Mbp)	59.2	**44.0**	36.4	30.5

The assemblies were generated with hifiasm version 0.15.5-r352 and HiCanu version version 2.3 and benchmarked using QUAST-LG version 5.0.2 with *T2TGenome* as the reference. Because authors of the QUAST-LG[24] recommend using HPC contigs for identifying misassemblies (Supplementary Note 5), misassemblies were identified by a separate run of QUAST-LG with HPC contigs against the HPC reference. Analysis of misassemblies reported by QUAST-LG in the 30 longest contigs (for each assembly tool) using the Icarus tool[33] confirmed that they all represent structural errors (a large insertion, deletion or relocation) rather than alignment artifacts. LJA 100%-assembled chromosomes 3, 5, 7, 10, 12 and 20; HiCanu 100%-assembled chromosome 20; and hifiasm 100%-assembled chromosome 5. Because all assemblies might have small variations at the chromosome ends, 99.9%-assembled chromosomes are counted as 100%-assembled chromosomes. 'Ideal' refers to the theoretically optimal assembly of a *k*-complete read-set obtained by generating contigs as edge-labels of the graph DB(*T2TGenome*,20,000). NGA50 and NGA75 metrics are reported based on contigs broken at positions defined by misassemblies in HPC contigs. Note that, even though very few HPC reads in the T2T dataset are longer than 20,000 bp, the LJA assembly of this read-set might improve on the ideal assembly DB(*T2TGenome*,20,000) because it uses information about coverage for loop resolution (Supplementary Note 8). The reference length is 3,054,832,041. The number of LJA contigs (665) is smaller than the number of edges in the constructed multiplex de Bruijn graph (1,432) because removing overlaps between contigs in the final LJA output results in many short contigs (LJA removes contigs that become shorter than 5 kb).

the T2T dataset were error-free, jumboDBG would construct the DB-graph of the error-free read-set *T2TErrorFree* with only 214,517 edges in 0.6 h using 33 Gb of memory. mowerDBG corrects most errors in reads, resulting in a much smaller DB-graph *DB*(*T2T'*,*k*) with 297,176 edges on the error-corrected read-set *T2T'*. jumboDBG further constructs the DB-graph *DB*(*T2T'*,*K*) using a larger *K*-mer size with 79,908 edges. Afterwards, mowerDBG performs the second round of error-correction in this graph, resulting in a nearly error-free read-set *Reads** and a DB-graph *DB*(*T2T**,*K*) with only 6,516 edges that approximates the graph *DB*(*T2TErrorFree*,*K*) with 4,956 edges. We note that, because the read-set *T2TErrorFree* was constructed based on the reference *T2TGenome*, which excluded the heterozygous regions present in the CHM13 cell line ([22], Nurk et al., 2021), *DB*(*T2T**,*K*) might have some heterozygous edges missing in *DB*(*T2TErrorFree*,*K*).

Step 9: *Transforming the DB-graph DB*(*Reads**,*K*) *into the multiplex de Bruijn graph* (Fig. 4). The choice of the *k*-mer size greatly affects the complexity of the DB-graph: gradually increasing *k* leads to a less tangled but more fragmented DB-graph. This tradeoff affects the contiguity of assembly, particularly in the case when the *k*-mers coverage by reads is non-uniform, let alone when the read-set misses some genomic *k*-mers. Ideally, we would like to vary the *k*-mer size, reducing it in low-coverage regions (to avoid fragmentation) and increasing it in high-coverage regions (to improve repeat resolution). The *iterative de Bruijn graph* approach[5,23] is a step toward addressing this goal by incorporating information about the de Bruijn graphs for a range of *k*-mer sizes $k_1 < k_2 < \ldots < k_t$ into the de Bruijn graph. However, this approach still constructs a graph with a fixed k_t-mer size.

multiplexDBG transforms the DB-graph *DB*(*Reads**,*K*) into the multiplex de Bruijn graph *MDB*(*Reads**,*K*) with vertices labeled by strings of length varying from *K* to K^+, where K^+ is larger than *K* (default value $K^+ = 40{,}001$). It transforms *DB*(*T2T**,5,001) with 6,516 edges into *MDB*(*T2T**,5,001) with only 1,432 edges and generates HPC contigs. Note that labels of some vertices of this graph are longer than all reads in the T2T dataset because multiplexDBG adds *virtual reads* to the read-set (Supplementary Note 6).

Step 10: *Expanding HPC contigs.* LJApolish expands HPC contigs (edge-labels in the multiplex de Bruijn graph) and results in an accurate final assembly with only ≅15 single-base errors per million nucleotides.

Evaluating genome assemblies. Given a string *s* in a string-set *S*, we define $L^+(s)$ as the total length of all strings in *S* with length at least $|s|$. The *N50 metric* for a contig-set *S* is defined as the length of the longest contig *s* in *S* with $L^+(s) \leq 0.5{*}|S|$. See ref. [24] on similar NG50, NGA50, NGA75 or LGA95 metrics.

We used the standard benchmarking metrics[24] as well as the additional *completeness metric* aimed at high-quality assemblies. We denote the length of a string *s* as $|s|$ and the total length of all strings in a string-set *S* as $|S|$. A contig is *correctly assembled* if it has no misassemblies. *Completeness* of a chromosome assembly is defined as the length of the longest correctly assembled contig from this chromosome divided by the chromosome length (in percentages). A chromosome is *N%-assembled* if its completeness is at least *N*%.

Benchmarking LJA on the T2T read-set. We benchmarked assemblies of the T2T read-set dataset against *T2TGenome*—the only complete and carefully validated large genome reference available today[18]. Table 1 illustrates that LJA produced the most contiguous assembly (NG50 = 97 Mb) with six 100%-assembled and nine 95%-assembled chromosomes (compared to NG50 = 90 Mb and one 95%-assembled chromosome for hifiasm). We classify a

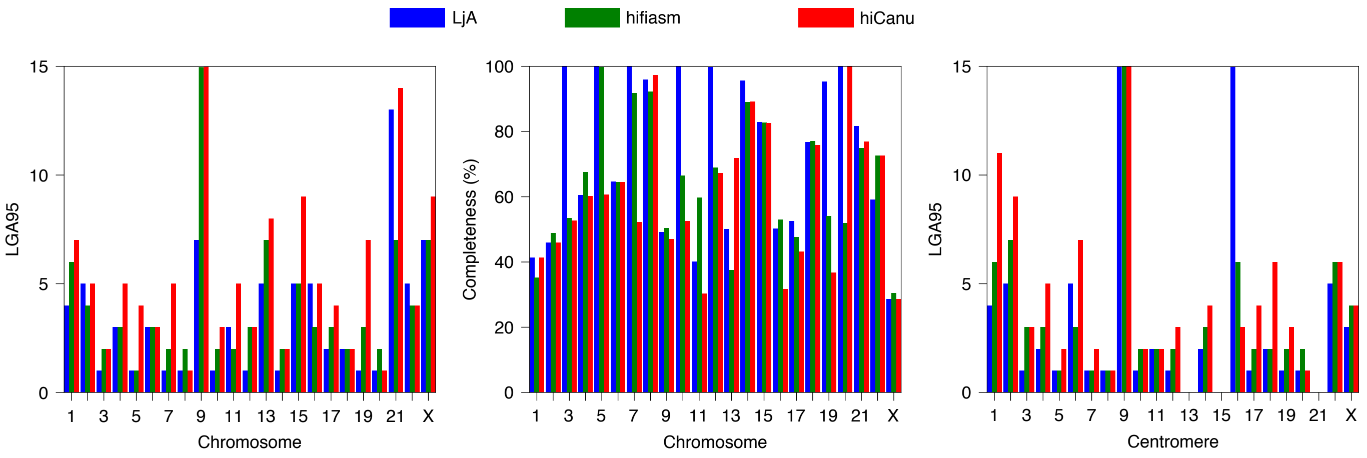

Fig. 3 | Chromosome-by chromosome LGA95 (left) and completeness (center) metrics as well as centromere-by-centromere LGA95 metric (right) for LJA (blue), hifiasm (green) and HiCanu (red) assemblies of the T2T read-set. LGA95 for each chromosome (centromere) is defined as the minimum number of *aligned blocks* needed to cover 95% of its length (aligned blocks are obtained by breaking contigs at misassembly breakpoints). LJA 100%-assembled six chromosomes and resulted in a better assembly (for example, assembly with lower LGA95) than hifiasm and HiCanu for most chromosomes. Although hifiasm resulted in lower LGA95 than LJA on chromosomes 2 (4 versus 5), 11 (2 versus 3), 16 (3 versus 5), 21 (7 versus 13) and 22 (4 versus 5), it made ten misassemblies in these five chromosomes compared to only one misassembly made by LJA. For example, for chromosome 21 with the biggest gap in the LGA95 values between hifiasm and LJA (7 versus 13), LJA resulted in a higher completeness (82%) than hifiasm (75%). As another example, LJA resulted in a much larger LGA95 = 22 on centromere 16 than hifiasm (6) and HiCanu (3) but made no errors (both hifiasm and HiCanu had three misassemblies). LGA95 values for centromeres 13, 15 and 21 (for all assemblers) are undefined because QUAST-LG reports multiple gaps in these centromeres. Long and highly repetitive rDDNA arrays are the only regions in the human genome that were not assembled by the T2T consortium. These regions are represented as simulated *rDDNA models* rather than correct rDDNA sequences in *T2TGenome*. Because centromeres 13, 15 and 21 include rDDNA models, QUAST-LG reports multiple gaps in their coverage by contigs that sum up to more than 5% of the lengths of these centromeres.

contig with length larger than or equal to NG50 (NG75) as *NG50-long* (*NG75-long*) contig. It turned out that 1 (4) out of all 12 NG50-long contigs assembled by LJA (hifiasm) are misassembled, resulting in a reduced NGA50 = 81 Mb for hifiasm. LJA (hifiasm) misassembled 1 out of its 24 (9 out of its 23) NG75-long contigs. Figure 3 demonstrates that LJA substantially improves on both hifiasm and HiCanu with respect to chromosome-by-chromosome and centromere-by-centromere assemblies.

LJA substantially reduced the number of misassemblies (10) as compared to hifiasm (58) and HiCanu (47). Although LJA produced more contigs than hifiasm (665 versus 448), this increased number does not indicate an inferior assembly but, rather, reflects the fact that LJA assembled an extra 13 Mb of the genome as compared to hifiasm (99.7% versus 99.3% genome fraction). In fact, hifiasm produced more contigs longer than 50 Kb than LJA (250 versus 194).

LJA and hifiasm made an order of magnitude fewer mismatches and indels than HiCanu. Analysis of these errors should take into account that an assembler covering a larger fraction of a genome might have additional errors in highly repetitive regions that are not covered by other assemblers. Although LJA made slightly more errors than hifiasm (34,293 versus 33,732), it made a smaller number of errors in regions assembled by both LJA and hifiasm (2,330 LJA errors were made in highly repetitive regions that were not assembled by hifiasm).

Table 1 also benchmarks the 'ideal' assembler on a *k*-complete read-set that outputs contigs as the edge-labels of DB(*T2TGenome*,20,000). It turned out that LJA assembly is similar in quality to the theoretically optimal assembly of error-free reads of length 20,000.

Supplementary Note 5 provides information about benchmarking individual LJA modules and illustrates that the runtime/memory of jumboDBG is largely defined by the size of the constructed DB-graph rather than the *k*-mer size.

Assembling a diploid human genome. Genome assemblers often collapse two heterozygous alleles (typically represented as bulges in the DB-graph) into a single copy to increase the contiguity of the *consensus assembly*, a mosaic of segments from maternal and paternal chromosomes. Diploid assemblers try to prevent such collapsing by (1) constructing a *phased assembly graph* that accurately represents the heterozygous alleles and (2) using the entire read-lengths and complementary technologies to increase the contiguity of paternal/maternal contigs in the phased assembly graph and generate a *haplotype-resolved assembly*[8,25]. Constructing an accurate and contiguous phased assembly graph is a critical step in both consensus and diploid assembly and the focus of the current efforts of the T2T project as it scales up from a single haploid to multiple diploid genomes. Indeed, a fragmented phased graph makes it difficult to generate consensus and haplotype-resolved assemblies, not to mention that errors in the graph likely trigger errors in these assemblies.

LJA generates a phased DB-graph of a diploid genome that represents an excellent starting point for generating these assemblies. We analyzed the phased LJA and hifiasm assemblies using the HG002 read-set from a diploid human genome. LJA generated an assembly with N50 = 383 kb and total length 5.8 Gb, whereas hifiasm generated an assembly with N50 = 310 kb and total length 6.7 Gb (before heterozygosity collapsing) that is ≅2.2 times larger than the human genome length. Although LJA generated a more contiguous phased assembly than hifiasm, it is unclear how to evaluate the accuracy of these assemblies in the absence of a validated complete reference genome for each haplome.

Although the LJA graph of the diploid read-set is much larger than the graph of the haploid T2T read-set (42,298 versus 1,440 edges), it has a rather simple 'bulged' structure, making it well-suited for the follow-up consensus and haplotype-resolution steps. Each bulge represents differing segments of paternal/maternal alleles (average length ≅150 kb) alternating with identical segments of paternal/maternal alleles (average length ≅30 kb).

Table 2 | Benchmarking LJA, hifiasm and HiCanu on MOUSE, MAIZE and FLY read-sets

Assembler	MOUSE			MAIZE			FLY		
	#contigs	total length (Gb)	NG50 (Mb)	#contigs	total length (Gb)	NG50 (Mb)	#contigs	total length (Gb)	NG50 (Mb)
LJA	1,282	2.71	24	1,310	2.15	26	313	0.22	9.5
hifiasm	658	2.61	21	1,136	2.18	35	933	0.24	10.0
HiCanu	3,334	2.67	15	1,992	2.17	23	6,439	0.32	9.2

Mouse, maize and fruit fly samples represent the C57BL/6J strain of *Mus musculus*[34], the B73 strain of *Zea mays*[34] and *Drosophila ananassae*[35], respectively. We used estimates of the lengths of the mouse, maize and fly genomes (2.7 Gb, 2.3 Gb and 0.22 Gb, respectively) for NG50 calculation. All assemblers were run with default parameters recommended for diploid genome assembly.

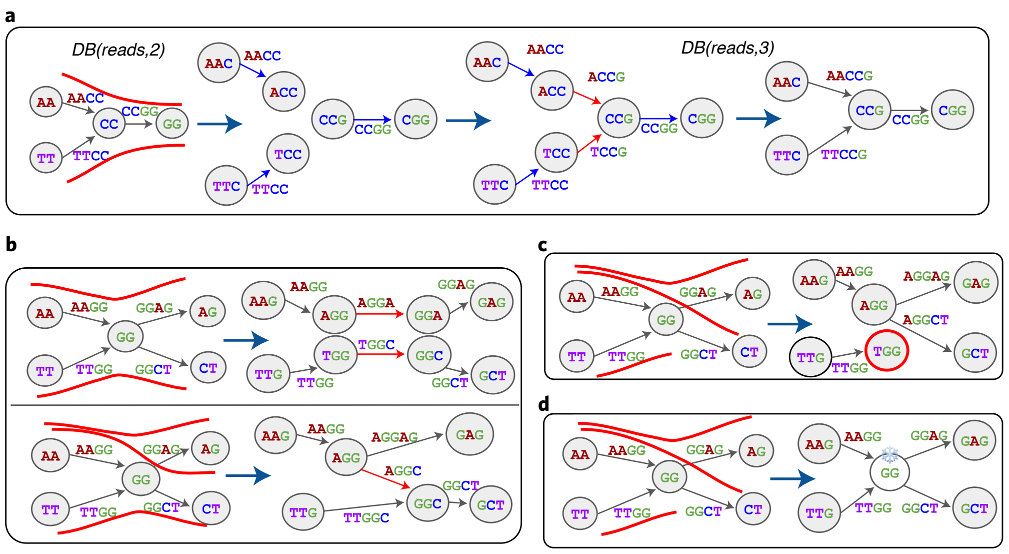

Fig. 4 | Constructing a multiplex de Bruijn graph. a, Iterative construction of the compressed de Bruijn graph.ransformation of *DB*(*Reads*,2) into *DB*(*Reads*,3) for the read-set *Reads* = {**AACCGG,TTCCGG**}. The first read defines a transition between edges **(AA,CC)** and **(CC,GG)**, whereas the second read defines a transition between edges **(TT,CC)** and **(CC,GG)**. The resulting transition-set defines a transition-graph that coincides with *DB*(*Reads*,3). The read-paths that define the transition-set are depicted as red curves traversing *DB*(*Reads*,2). multiplexDBG 'tears apart' edges of the *DB*(*Reads*,2) and increases the *k*-mer length within the vertices of the resulting isolated edges. Afterwards, it introduces new red edges that correspond to connections induced by transitions, resulting in a path-graph. Compressing non-branching paths in the path-graph (ignoring the edge colors) results in *DB*(*Reads*,3). Because vertex '**CC**' is simple, graphs *DB*(*Reads*,2) and *DB*(*Reads*,3) have the same topology. The shown 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 outgoing edge from this vertex and adds a single symbol to the labels of incoming edges into this vertex. **b**, Transforming a complex vertex. The read-sets *Reads*$_1$ = {**AAGGAG,TTGGCT**} (above) and *Reads*$_2$ = {**AAGGAG**, **TTGGCT**, **AAGGCT**} (below) result in the same graphs *DB*(*Reads*$_1$,2) and *DB*(*Reads*$_2$,2) but different graphs *DB*(*Reads*$_1$,3) and *DB*(*Reads*$_2$,3). Because vertex **GG** in the graph *DB*(*Reads*$_1$,2)=*DB*(*Reads*$_2$,2) is complex, the topology of the graphs *DB*(*Reads*$_1$,3) and *DB*(*Reads*$_2$,3) depends on the read-set. *DB*(*Reads*$_1$,3) consists of two connected components (top), whereas *DB*(*Reads*$_2$,3) is a single-component graph because it has an extra edge (labeled **AGGC**) introduced by the additional read **AAGGCT**. **c**, Limitation of the de Bruijn graph approach to genome assembly. Reckless resolution of unpaired complex vertices might disconnect the genome traversal. For a read-set *Reads* = {**AAGGAG**, **AAGGCT**, **TTGG**}, because the read-path of **TTGG** ends inside the complex vertex **GG**, the vertex **TGG** (with red outline) represents a dead-end. **d**, Multiplex de Bruijn graph transformation. The multiplex de Bruijn graph for the same read-set avoids reckless resolution of complex vertices by freezing unpaired complex vertices.

Assembling mouse, maize and fruit fly genomes. We benchmarked LJA, hifiasm and HiCanu on the MOUSE, MAIZE and FLY HiFi read-sets from the inbred mouse, maize and fly species. It is unclear how to compare the reference genomes with HiFi assemblies for these datasets because the quality of these assemblies might exceed the quality of the references. For example, although Table 2 illustrates that LJA and hifiasm improved on HiCanu with respect to NG50 metric, this metric does not account for errors in assemblies and references. For example, hifiasm assembled the MAIZE dataset with the highest NG50 but made more misassemblies (6,081 versus 5,594 for LJA), resulting in a rather low NGA50 (1.4 Mb for both hifiasm and LJA). The large number of misassemblies for all inbred datasets suggests that many of them might be triggered by errors in the reference genome or differences between maternal/paternal alleles (even after inbreeding). Supplementary Note 7 illustrates that existing HiFi assemblers generate rather different results, implying that some of them make many assembly errors. In the absence of an automated validation pipeline, it remains unclear how to detect these errors in the era of complete genome sequencing.

Discussion

The development of assembly algorithms started from applications of the overlap/string graph approach. Even though this approach becomes slow and error-prone with respect to detecting overlaps in the highly repetitive regions, the alternative DB-graph approach[26,27] was often viewed as a theoretical concept rather than a practical method.

Even after it turned into the most popular method for short-read assembly, the development of algorithms for assembling long error-prone reads again started from the overlap/string graph approach[28–30] because the DB-graph approach was viewed as inapplicable to error-prone reads[31]. Indeed, because long *k*-mers from the genome typically do not even occur in error-prone reads, it seemed unlikely that the DB-graph approach might assemble such reads. However, the development of Flye[17] and wtdbg2 (ref. [32]) demonstrated, once again, that the DB-graph-based long-read assemblers result in accurate and order(s) of magnitude faster algorithms than the overlap/string graph approach.

Because the DB-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[7,8]. We described an alternative approach for assembling HiFi reads, illustrating that the 'contest' between the overlap/string graph and the de Bruijn graph approach continues. Benchmarking on the T2T dataset demonstrated that LJA improves on the state-of-the-art HiFi assemblers with respect to both contiguity and accuracy. Although it is unclear how to conduct benchmarking without validated complete reference genomes, LJA results on the HG002 dataset illustrate that it generates highly contiguous phased assemblies. Although this paper focuses on phased assemblies, it has immediate implications for the downstream applications because phased assemblies represent a stepping stone for both consensus and haplotype-resolved assemblies. For example, bulge-collapsing and tip removal in the phased LJA assembly of the HG002 read-set results in a contiguous consensus assembly with N50 = 54 Mb. We are now developing the diploidLJA tool for haplotype-resolved assembly and the nanoLJA tool for combining HiFi and Oxford Nanopore reads to improve the contiguity of assemblies.

Online content

Any methods, additional references, Nature Research reporting summaries, source data, extended data, supplementary information, acknowledgements, peer review information; details of author contributions and competing interests; and statements of data and code availability are available at https://doi.org/10.1038/s41587-022-01220-6.

Received: 27 September 2021; Accepted: 11 January 2022;
Published online: 28 February 2022

References

1. Nurk, S. et al. The complete sequence of a human genome. *bioRxiv* https://doi.org/10.1101/2021.05.26.445798 (2021).
2. Miller, D. E. et al. Targeted long-read sequencing identifies missing disease-causing variation. *Am. J. Hum. Genet.* **108**, 1436–1449 (2021).
3. Wenger, A. M. et al. Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nat. Biotechnol.* **37**, 1155–1162 (2019).
4. Compeau, P. E., Pevzner, P. A. & Tesler, G. How to apply de Bruijn graphs to genome assembly. *Nat. Biotechnol.* **29**, 987–991 (2011).
5. Bankevich, A. et al. SPAdes: a new genome assembly algorithm and its applications to single cell sequencing. *J. Comput. Biol.* **19**, 455–477 (2012).
6. Zerbino, D. R. & Birney, E. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.* **18**, 821–829 (2008).
7. Nurk, S. et al. HiCanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads. *Genome Res.* **30**, 1291–1305 (2020).
8. Cheng, H. et al. Haplotype-resolved de novo assembly with phased assembly graphs. *Nat. Methods* **18**, 170–175 (2021).
9. Myers, E. W. The fragment assembly string graph. *Bioinformatics* **21**, ii79–ii85 (2005).
10. Li, D., Liu, C. M., Luo, R., Sadakane, K. & Lam, T. W. MEGAHIT: an ultra-fast single-node solution for large and complex metagenomics assembly via succinct de Bruijn graph. *Bioinformatics* **31**, 1674–1676 (2015).
11. Pevzner, P., Tang, H. & Tesler, G. De novo repeat classification and fragment assembly. *Genome Res.* **14**, 1786–1796 (2004).
12. Ye, C., Ma, Z. S., Cannon, C. H., Pop, M. & Yu, D. W. Exploiting sparseness in de novo genome assembly. *BMC Bioinformatics* **13**, S1 (2012).
13. Kolmogorov, M. et al. metaFlye: scalable long-read metagenome assembly using repeat graphs. *Nat. Methods* **17**, 1103–1110 (2020).
14. Rautiainen, M. & Marschall, T. MBG: minimizer-based sparse de Bruijn graph construction. *Bioinformatics* **37**, 2476–2478 (2021).
15. Bloom, B. H. Space/time tradeoffs in hash coding with allowable errors. *Commun. ACM* **13**, 422–426 (1970).
16. Karp, R. M. & Rabin, M. O. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.* **31**, 249–260 (1987).
17. Kolmogorov, M., Yuan, J., Lin, Y. & Pevzner, P. A. Assembly of long error-prone reads using repeat graphs. *Nat. Biotechnol.* **37**, 540 (2019).
18. Mc Cartney, A. M. et al. Chasing perfection: validation and polishing strategies for telomere-to-telomere genome assemblies. Preprint at https://www.biorxiv.org/content/10.1101/2021.07.02.450803v1 (2021).
19. Roberts, M., Hayes, W., Hunt, B. R., Mount, S. M. & Yorke, J. A. Reducing storage requirements for biological sequence comparison. *Bioinformatics* **20**, 3363–3369 (2004).
20. Chikhi, R. & Rizk, G. Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms Mol. Biol.* **8**, 22 (2013).
21. Minkin, I., Pham, S. & Medvedev, P. TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics* **33**, 4024–4032 (2017).
22. Bzikadze, A. V. & Pevzner, P. A. Automated assembly of centromeres from ultra-long error-prone reads. *Nat. Biotechnol.* **38**, 1309–1316 (2020).
23. Peng, Y., Leung, H. C. M., Yiu, S. M. & Chin, F. Y. L. IDBA—a practical iterative de Bruijn graph de novo assembler. in *Research in Computational Molecular Biology. RECOMB 2010. Lecture Notes in Computer Science* Vol. 6044, 426–440 (2010).
24. Mikheenko, A., Prjibelski, A., Saveliev, V., Antipov, D. & Gurevich, A. Versatile genome assembly evaluation with QUAST-LG. *Bioinformatics* **34**, i142–i150 (2018).
25. Garg, S. et al. Chromosome-scale, haplotype-resolved assembly of human genomes. *Nat. Biotechnol.* **39**, 309–312 (2021).
26. Idury, R. M. & Waterman, M. S. A new algorithm for DNA sequence assembly. *J. Comput. Biol.* **2**, 291–306 (1995).
27. Pevzner, P. A., Tang, H. & Waterman, M. S. An Eulerian path approach to DNA fragment assembly. *Proc. Natl Acad. Sci. USA* **98**, 9748–9753 (2001).
28. Chin, C. S. et al. Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nat. Methods* **10**, 563–569 (2013).
29. Chin, C. et al. Phased diploid genome assembly with single-molecule real-time sequencing. *Nat. Methods* **13**, 1050–1054 (2016).
30. Koren, S. et al. Hybrid error correction and de novo assembly of single-molecule sequencing reads. *Nat. Biotechnol.* **30**, 693–700 (2012).
31. Roberts, R. J., Carneiro, M. O. & Schatz, M. C. The advantages of SMRT sequencing. *Genome Biol.* **14**, 405 (2013).
32. Ruan, J. & Li, H. Fast and accurate long-read assembly with wtdbg2. *Nat. Methods* **17**, 155–158 (2020).
33. Mikheenko, A., Valin, G., Prjibelski, A., Saveliev, V. & Gurevich, A. Icarus: visualizer for de novo assembly evaluation. *Bioinformatics* **32**, 3321–3323 (2016).
34. Hon, T. et al. Highly accurate long-read HiFi sequencing data for five complex genomes. *Sci. Data* **7**, 399 (2020).
35. Tvedte, E. S. et al. Comparison of long-read sequencing technologies in interrogating bacteria and fly genomes. *G3 (Bethesda)* **11**, jkab083 (2021).

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© The Author(s), under exclusive licence to Springer Nature America, Inc. 2022

Methods

This section is organized as follows. Before constructing the compressed de Bruijn graph *DB*(*Reads*,*k*), we describe a simpler yet still open problem of constructing the compressed de Bruijn graph *DB*(*Genome*,*k*) of a large *circular* string *Genome* for a large *k*-mer size. This problem[21] serves as a stepping stone for constructing a much larger graph *DB*(*Reads*,*k*). After describing the algorithm for solving this problem, we describe complications that arise in the case of constructing the compressed de Bruijn graph from a genome with linear chromosomes. Afterwards, we describe how to modify the algorithm for constructing *DB*(*Genome*,*k*) into the algorithm for constructing *DB*(*Reads*,*k*). Because this transformation faces the time/memory bottleneck, we describe how jumboDBG first assembles reads into disjointigs. Afterwards, we describe steps 8 (error-correcting reads), 9 (constructing multiplex de Bruijn graph) and 10 (expanding HPC assembly) of the LJA pipeline. Supplementary Note 9 describes LJA parameters.

Because the algorithm for constructing the de Bruijn graph of a circular genome does not require construction of disjointigs (step 3), we number its steps 4–7 as 4G–7G to be consistent with the previously described steps of jumboDB:

Step 4G: Generating the Bloom filter for all ($k+1$)-mers in *Genome*

Step 5G: Using the Bloom filter to construct a junction-superset *Junctions*$^+$ of *Genome*

Step 6G: Using the set *Junctions*$^+$ to break *Genome* into splits

Step 7G: Using splits to construct *DB(Genome,k)*

Sparse de Bruijn graphs. Given a set of *k*-mers *Anchors* from a string-set *Genome*, the sparse de Bruijn graph *SDB(Genome,Anchors)* is a graph with the vertex-set *Anchors* and the edge-set *Splits*(*Genome,Anchors*) (each split in *Splits*(*Genome,Anchors)* represents a label of an edge connecting its *k*-prefix with its *k*-suffix). Two vertices in this graph might be connected by multiple edges with different labels corresponding to different splits with the same *k*-prefixes and *k*-suffixes. A straightforward algorithm for constructing *SDB*(*Genome,Anchors*) takes O(|*Genome*|·|*Anchors*|·*k*) time.

A string-set *Genome* corresponds to a path-set that traverses each edge in the de Bruijn *DB*(*Genome*,*k*) at least once and spells *Genome*. We refer to this path-set as the *genome traversal*.

When the set of anchors is equal to the set of junctions *Junctions* = *Junctions*(*Genome*,*k*), each vertex of *DB*(*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 *DB*(*Genome*,*k*). Moreover, if *Junctions*$^+$ is a *superset* of all junctions, that contains all junctions as well as some *false junctions* (that is, non-branching *k*-mers from *Genome*), *SDB*(*Genome,Junctions*$^+$) is a subpartition of *DB*(*Genome*,*k*).

If the junction-set *Junctions* = *Junctions*(*Genome*,*k*) was known, construction of *DB*(*Genome*,*k*) would be a simple task because it coincides with *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 *DB*(*Genome*,*k*) by constructing *SDB*(*Genome,Junctions*$^+$) and compressing all its non-branching paths.

Step 4G: *Generating the Bloom filter for all* ($k+1$)*-mers in Genome*. In the case of assembling reads, jumboDBG stores all ($k+1$)-mers from disjointigs in a Bloom filter *Bloom*(*Disjointigs*,*k*,*BloomNumber*,*BloomSize*) formed by *BloomNumber* different independent *hash functions*, each mapping a ($k+1$)-mer into a bit array of size *BloomSize*. In the case of a circular genome, it constructs the Bloom filter in the same way by assuming that this genome forms a single disjointig. Storing all ($k+1$)-mers in a Bloom filter allows one to quickly query whether an arbitrary ($k+1$)-mer occurs in disjointigs and thus speed up the de Bruijn graph construction[20]. Given hash functions $h_1,h_2,\ldots,h_{BloomNumber}$ and a *k*-mer *a*, one can quickly check whether all bits $h_1(a),h_2(a),\ldots,h_{BloomNumber}$ of the Bloom filter are equal to 1, an indication that the *k*-mer *a* might have been stored in the Bloom filter. Supplementary Note 9 describes how jumboDBG sets the parameters of the Bloom filter.

Step 5G: *Using the Bloom filter to construct the junction-superset Junctions*$^+$ *of Genome*. To generate a junction-superset *Junctions*$^+$ with a small number of false junctions, jumboDBG uses the Bloom filter to compute the upper bound on the indegree and outdegree of each vertex in *UDB(Genome,k)* as described in the Results.

A vertex in a graph is *complex* if both its indegree and outdegree exceed 1 and *simple* otherwise. A junction is a *dead-end* if it has no incoming or no outgoing edges and a *crossroad* otherwise. In the case of a genome with linear chromosomes, the Bloom filter might overestimate the indegree and/or outdegree of some dead-end junctions—for example, to misclassify a zero-in-one-out junction as a simple vertex. However, all crossroad junctions will be correctly identified, thus generating a junction-superset (in the case of a circular genome that does not have dead-end junctions), constructing a subpartition of *DB*(*Genome*,*k*) and further transforming it into *DB*(*Genome*,*k*).

Steps 6G and 7G: *Using the set Junctions*$^+$ *to break Genome into splits and using splits to construct DB(Genome,k)*. To construct *SDB*(*Genome,Junctions*$^+$), one can generate a Bloom filter for computing a junction-superset *Junctions*$^+$ and check which *k*-mers from *Genome* coincides with a *k*-mer from *Junctions*$^+$. Both these tasks require computing hashes of each *k*-mer, a procedure that usually takes O(*k*) time and becomes slow when *k* is large. For example, constructing a hashmap of *Junctions*$^+$ results in a prohibitively slow algorithm for constructing *SDB*(*Genome,Junctions*$^+$) with O(|*Genome*|·*k*) runtime.

To compute the hash function in O(1) rather than O(*k*) time, jumboDBG uses a 128-bit polynomial rolling hash of *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 time to construct the compressed de Bruijn graph to O(|*Genome*|). Similarly, to speed up the construction of *Junctions*$^+$, instead of storing *k*-mers, jumboDBG stores their rolling hashes in the Bloom filter, thus reducing the runtime from O(|*Genome*|·*k*) to O(|*Genome*|). Therefore, it constructs the compressed de Bruijn graph of a circular genome in O(|*Genome*|) time that does not depend on the *k*-mer size.

Constructing the compressed de Bruijn graph from reads. The described algorithm for constructing the de Bruijn graph of a circular genome can be applied to any string-set *Genome*, resulting in a graph that we refer to as *DB**(*Genome*,*k*). However, although *DB**(*Genome*,*k*)=*DB*(*Genome*,*k*) for a genome formed by circular chromosomes, it is not the case for a genome with linear chromosomes (or a genome represented by a *k*-complete error-free read-set).

We say that a string-set *Genome bridges* an edge of the compressed de Bruijn graph *DB*(*Genome*,*k*) if the label of this edge represents a substring of *Genome*. A genome is called *bridging* (with respect to a given *k*-mer size) if it bridges all edges of *DB*(*Genome*,*k*) and *non-bridging* otherwise. For example, a genome formed by 'chromosomes' *Genome* = {ATGC,GCACC} is non-bridging because *DB*(*Genome*,2) consists of a single edge with label ATGCACC that does not represent a substring of *Genome*.

Although *DB**(*Genome*,*k*)=*DB*(*Genome*,*k*) in the case of a bridging genome, *DB*(Genome,k)* does not necessarily coincide with *DB*(*Genome*,*k*) for a non-bridging genome—for example, a genome with linear chromosomes that does not bridge all edges of *DB*(*Genome*,*k*). However, after extending the junction-superset by *k*-prefixes and *k*-suffixes of all linear chromosomes, the same algorithm will construct the graph that represents a subpartition of *DB*(*Genome*,*k*). Although this subpartition can be further transformed into *DB*(*Genome*,*k*) by compressing all non-branching paths, the resulting algorithm becomes slow when the number of linear chromosomes is large, resulting in a prohibitively large junction-superset. This increase becomes problematic when one constructs the compressed de Bruijn graph *DB*(*Reads*,*k*) because each read represents a 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 low: selecting it to be proportional to the total read-length (as described in Supplementary Note 9) results in the prohibitively large memory. Even if the genome size is known, it is unclear how to select *BloomSize* because the number of different *k*-mers in reads affects the false-positive rate.

jumboDBG addresses these problems by assembling reads into compact disjointigs that form a bridging genome for the graph *DB*(*Reads*,*k*) and constructing the compressed de Bruijn graph *DB*(*Disjointigs*,*k*)=*DB*(*Reads*,*k*) from the resulting disjointig-set *Disjointigs* instead of the read-set *Reads*. LJA sets the parameter *BloomSize* to be proportional to the total disjointig-length (that is typically an order of magnitude smaller than the total read-length), thus greatly reducing the memory footprint.

Step 3: *Constructing a compact disjointig-set from a read-set*. We defined the concepts of complete and compact disjointig-sets in the Results. Similarly to a disjointig of a read-set, a disjointig of a genome is defined as a string spelled by an arbitrary path in *DB*(*Genome*,*k*). If a disjointig-set *Disjointigs* is complete, then *DB*(*Genome*,*k*)=*DB*(*Disjointigs*,*k*). However, the graph *DB**(*Disjointigs*,*k*) constructed by jumboDBG might differ from *DB*(*Genome*,*k*) because it does not include edges of *DB*(*Genome*,*k*) that are not bridged by *Disjointigs*. However, if a disjointig-set *Disjointigs* is compact, it forms a bridging genome, implying that *DB** (*Disjointigs*,*k*)=*DB*(*Genome*,*k*).

jumboDBG constructs a compact disjointig-set as a set of edge-labels in the compact sparse de Bruijn graph *SDB*(*Reads*,*Anchors**). 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 the *k*-suffix of a read are not anchors, they might be missing in *SDB*(*Reads*,*Anchors*) because only segments between anchors are added to this graph. As described in the Results, jumboDBG modifies the concept of a minimizer of a linear string by adding its *k*-prefix and *k*-suffix to the set of its minimizers, resulting in the set *Anchors* = *Anchors*(*Reads*,*width*,*k*). jumboDBG constructs the sparse de Bruijn graph *SDB*(*Reads*,*Anchors*), transforms it into a compact sparse de Bruijn graph *SDB*(*Reads*,*Anchors**) as described in Supplementary Note 10 and generates a compact disjointig-set as labels of all non-branching paths in this graph.

Step 8. *Correcting errors in reads and constructing the graph DB*(*Reads**,*K*) *on the error-corrected read-set Reads** *using a larger K-mer size*. Because an error in a single position of a read triggers an error in each *k*-mer covering this position, and because a typical HiFi read has one error per 500 nucleotides on average, the fraction of correct *k*-mers (among all *k*-mers in reads) becomes rather low when *k* exceeds 500, resulting in a complex de Bruijn graph of reads that does not adequately represent the genome. Ideally, we would like to construct the de Bruijn graph using as large a k as possible—for example, $k=15{,}000$, slightly below the typical read-length in the T2T dataset. However, this approach results in a highly fragmented de Bruijn graph because reads in this dataset do not span a large fraction of genomic 15,000-mers. Although reducing k to, say, 5,000 addresses this complication (nearly all genomic 5,000-mers are spanned by reads), most 5,000-mers in reads are erroneous, preventing their assembly.

LJA attempts to minimize the effect of (1) errors in reads and (2) incomplete coverage of genomic k-mers by constructing and error-correcting the de Bruijn graphs with a small k-mer size and a large K-mer size with default values $k=501$ and $K=5{,}001$. As described in the Results, this two-round error-correction results in a nearly error-free read-set *Reads** (Fig. 2).

mowerDBG uses the de Bruijn graph of HPC reads for detecting errors in these reads. Because ≅69% of 501-mers in HPC reads are correct, the correct 501-mers have high coverage, in contrast to low-coverage erroneous 501-mers that form *bulges* and *tips* (with the exception of k-mers that contain some tandem dinucleotide repeats discussed in Supplementary Note 11). Afterwards, mowerDBG uses the *path-rerouting* and *bulge-collapsing* error-correction approaches to simultaneously correct reads and the graph.

Even though the previous paragraph might create an impression that errors in HiFi reads can be corrected by simply applying error-correcting approaches developed for short reads (for example, from SPAdes assembler), correction of HiFi reads faces unique challenges that we outline below. First, our goal is to correct reads (rather than to correct the de Bruijn graph as in SPAdes) because we need to rescue correct k-mers for the second round of error correction with a large K-mer size. Second, we need to perform nearly perfect error correction even in highly repetitive regions (for example, centromeres) that short-read assemblers do not even try to assemble. Third, in difference from short-read assembly, the target k-mer size (501) is a small fraction of a typical read-length (15,000). As a result, analyzing a bulge in a highly repetitive region requires not only analysis of the graph structure (like in SPAdes) but also analysis of all read-paths traversing this bulge.

To address these complications, mowerDBG complements path-rerouting and bulge-collapsing by additional steps referred to as correcting dimers and correcting pseudo-correct reads. Supplementary Note 11 describes the mowerDBG algorithm.

Step 9: *Transforming the de Bruijn graph into the multiplex de Bruijn graph.* Below we describe a graph transformation algorithm for transforming the graph $DB(Reads,k)$ into $DB(Reads,k^+)$ for $k^+>k$ by iteratively increasing the k-mer size by 1 at each iteration. Although launching jumboDBG to construct $DB(Reads,k)$, followed by these transformations, takes more time than simply launching jumboDBG to construct $DB(Reads,k^+)$, 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 $vertexSize(w) \geq k$. 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. We consider graphs with specified edge-labels (vertex-labels can be inferred from these edge-labels) and assume that different vertices have different vertex-labels. We will start by analyzing graphs with the same vertex size for all vertices and will later transition to the multiplex de Bruijn graphs that have vertices of varying sizes.

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 that inherits the label of the edge e in G (Fig. 4a). 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 addition to blue edges, each pair of edges $in=(v,w)$ and $out=(w,u)$ in *Transitions* adds a red *transition edge* between in_{end} and out_{start} to the transition graph. The label of this edge is defined as a $(k+2)$-mer formed by the concatenate $symbol_{-(k+1)}(in)*label(w)*symbol_{(k+1)}(out)$, 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 $label(e)$.

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 $Paths(v,w)$ and the set of all paths traversing a vertex w as $Paths(w)$. We define the set $Transitions(Paths)$ as the set of all pairs of consecutive edges in all paths from *Paths*. A *path-graph* $G(Paths)$ of a path-set *Paths* is defined as the transition-graph $G(Transitions(Paths))$.

Let *Paths* be the set of all read-paths in the compressed de Bruijn graph $G=DB(Reads,k)$. A straightforward approach to constructing the graph $G(Paths)$, which recomputes labels from scratch at each iteration, nearly doubles the path lengths at each iteration and thus faces the time/memory bottleneck. multiplexDBG avoids this time/memory bottleneck by modifying rather than recomputing the edge labels from scratch, as described in Supplementary Note 13.

Multiplex de Bruijn graph transformation. Given a path-set *Paths* in a graph G, we call edges (v,w) and (w,u) in G *paired* if the transition-set $Transitions(Paths)$ contain this pair of edges. A vertex w in G is *paired* if each edge incident to w is paired with at least one other edge incident to w and *unpaired* otherwise.

The important property of $DB(Genome,k)$ is that there exists a genome traversal of this graph. Given a genome traversal of the graph $DB(Reads,k)$, we want to preserve it in $DB(Reads,k+1)$ after the graph transformation. However, it is not necessarily the case because the transformation of $DB(Reads,k)$ into $DB(Reads,k+1)$ might create dead-ends (each unpaired vertex in $DB(Reads,k)$ results in a dead-end in $DB(Reads,k+1)$), thus 'losing' the genome traversal that existed in $DB(Reads,k)$ (Fig. 4c). Below we describe a single iteration of the algorithm for transforming $DB(Reads,k)$ into the multiplex de Bruijn graph $MDB(Reads,k)$ that avoids creating dead-ends whenever possible by introducing vertices of sizes $(k+1)$ in this graph.

multiplexDBG transforms each paired vertex of $DB(Reads,k)$ using the graph transformation 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, multiplexDBG freezes v (Fig. 4d). The motivation for freezing v is that, if we did not freeze it, we would need to remove the edge connecting u and v in $MDB(Reads,k)$, disrupting the topology of the graph. multiplexDBG continues the 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 k-mer varying in sizes from k to K^+. Supplementary Note 14 illustrates that multiplex transformations might be overly optimistic (by transforming vertices that should have been frozen) and overly pessimistic (by freezing vertices that should have been transformed).

Step 10: *Expanding HPC contigs.* Although LJA enables an accurate LJA assembly of HPC reads into HPC contigs (edge-labels in the multiplex de Bruijn graph), these contigs have to be expanded (de-collapsed) using information about homopolymer runs in the original reads. Supplementary Note 15 describes how LJApolish expands HPC contigs.

Reporting Summary. Further information on research design is available in the Nature Research Reporting Summary linked to this article.

Data availability

All assemblies generated by LJA are available at https://zenodo.org/record/5552696#.YV3MkVNBxH4. All described datasets are publicly available through the corresponding repositories. All HiFi data were obtained from the National Center for Biotechnology Information Sequence Read Archive (SRA). The SRA access codes for all datasets are specified in Supplementary Note 2 'Information about datasets'. The CHM13 reference (version 0.9) generated by the T2T consortium (referred to as T2TGenome) can be found at https://s3.amazonaws.com/nanopore-human-wgs/chm13/assemblies/chm13.draft_v0.9.fasta.gz.

Code availability

The LJA code uses the open-source libraries spoa (version 4.0.5) and ksw2 (version 4e0a1cc) and is available at https://github.com/AntonBankevich/LJA. All software tools used in the analysis and their versions and parameters are specified in the text of the paper and in Supplementary Note 9 'LJA parameters'.

Acknowledgements

A. Bankevich, A. Bzikadze and P.A.P. were supported by National Science Foundation EAGER award 2032783. D.A. was supported by Saint Petersburg State University (grant ID PURE 73023672). We thank A. Korobeynikov and A. Mikheenko for useful suggestions on assembling HiFi reads using SPAdes and benchmarking.

Author contributions

All authors contributed to developing the LJA algorithms and writing the paper. A. Bankevich (jumboDBG and mowerDBG), A. Bzikadze (multiplexDBG) and D.A. (LJApolish) implemented the LJA algorithm. A. Bankevich benchmarked LJA and other assembly tools. A. Bankevich and P.A.P. directed the work.

Competing interests

The authors declare no competing interests.

Additional information

Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s41587-022-01220-6.

Correspondence and requests for materials should be addressed to Anton Bankevich or Pavel A. Pevzner.

Peer review information *Nature Biotechnology* thanks Ergude Bao and the other, anonymous, reviewer(s) for their contribution to the peer review of this work.

Reprints and permissions information is available at www.nature.com/reprints.

nature research

Corresponding author(s):	Pavel Pevzner
Last updated by author(s):	2021/9/29

Reporting Summary

Nature Research wishes to improve the reproducibility of the work that we publish. This form provides structure for consistency and transparency in reporting. For further information on Nature Research policies, see our Editorial Policies and the Editorial Policy Checklist.

Statistics

For all statistical analyses, confirm that the following items are present in the figure legend, table legend, main text, or Methods section.

n/a	Confirmed	
☐	☒	The exact sample size (*n*) for each experimental group/condition, given as a discrete number and unit of measurement
☒	☐	A statement on whether measurements were taken from distinct samples or whether the same sample was measured repeatedly
☒	☐	The statistical test(s) used AND whether they are one- or two-sided *Only common tests should be described solely by name; describe more complex techniques in the Methods section.*
☒	☐	A description of all covariates tested
☒	☐	A description of any assumptions or corrections, such as tests of normality and adjustment for multiple comparisons
☐	☒	A full description of the statistical parameters including central tendency (e.g. means) or other basic estimates (e.g. regression coefficient) AND variation (e.g. standard deviation) or associated estimates of uncertainty (e.g. confidence intervals)
☒	☐	For null hypothesis testing, the test statistic (e.g. *F*, *t*, *r*) with confidence intervals, effect sizes, degrees of freedom and *P* value noted *Give P values as exact values whenever suitable.*
☒	☐	For Bayesian analysis, information on the choice of priors and Markov chain Monte Carlo settings
☒	☐	For hierarchical and complex designs, identification of the appropriate level for tests and full reporting of outcomes
☒	☐	Estimates of effect sizes (e.g. Cohen's *d*, Pearson's *r*), indicating how they were calculated

Our web collection on statistics for biologists contains articles on many of the points above.

Software and code

Policy information about availability of computer code

Data collection	no software was used for data collection
Data analysis	All software tools used in the analysis and their versions/parameters are specified in the text of the paper and the Supplementary Note "LJA Parameters".The developed software is deposited to github as specified in the Code Availability section.spoa and ksw2 software libraries have been incorporated in the LJA codebase as described in the manuscript

For manuscripts utilizing custom algorithms or software that are central to the research but not yet described in published literature, software must be made available to editors and reviewers. We strongly encourage code deposition in a community repository (e.g. GitHub). See the Nature Research guidelines for submitting code & software for further information.

Data

Policy information about availability of data

All manuscripts must include a data availability statement. This statement should provide the following information, where applicable:

- Accession codes, unique identifiers, or web links for publicly available datasets
- A list of figures that have associated raw data
- A description of any restrictions on data availability

All information about data is provided in "Data Availability" statement

All HiFi data were obtained from the NCBI Sequence Read Archive. The SRA access codes for all datasets are specified in Supplementary Note 2 “Information about datasets.” The CHM13 reference (v0.9) generated by the T2T consortium (referred to as T2TGenome) can be found at https://s3.amazonaws.com/nanopore-human-wgs/chm13/assemblies/chm13.draft_v0.9.fasta.gz.

Field-specific reporting

Please select the one below that is the best fit for your research. If you are not sure, read the appropriate sections before making your selection.

☒ Life sciences ☐ Behavioural & social sciences ☐ Ecological, evolutionary & environmental sciences

For a reference copy of the document with all sections, see nature.com/documents/nr-reporting-summary-flat.pdf

Life sciences study design

All studies must disclose on these points even when the disclosure is negative.

Sample size	sample size is not applicable to this algorithmic development
Data exclusions	no data were excluded in this study
Replication	replication is not applicable to this algorithmic study
Randomization	no randomization was performed in this algorithmic study
Blinding	no blinding was performed in this algorithmic study

Reporting for specific materials, systems and methods

We require information from authors about some types of materials, experimental systems and methods used in many studies. Here, indicate whether each material, system or method listed is relevant to your study. If you are not sure if a list item applies to your research, read the appropriate section before selecting a response.

Materials & experimental systems

n/a	Involved in the study
☒	☐ Antibodies
☒	☐ Eukaryotic cell lines
☒	☐ Palaeontology and archaeology
☒	☐ Animals and other organisms
☒	☐ Human research participants
☒	☐ Clinical data
☒	☐ Dual use research of concern

Methods

n/a	Involved in the study
☒	☐ ChIP-seq
☒	☐ Flow cytometry
☒	☐ MRI-based neuroimaging