Applying the Safe-And-Complete Framework to Practical Genome Assembly

Sebastian Schmidt
Department of Computer Science, University of Helsinki, Finland

Santeri Toivonen
Department of Computer Science, University of Helsinki, Finland

Paul Medvedev[1]
Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA, USA
Huck Institutes of the Life Sciences, The Pennsylvania State University, University Park, PA, USA
Department of Biochemistry and Molecular Biology, The Pennsylvania State University, University Park, PA, USA

Alexandru I. Tomescu[1]
Department of Computer Science, University of Helsinki, Finland

Abstract

Despite the long history of genome assembly research, there remains a large gap between the theoretical and practical work. There is practical software with little theoretical underpinning of accuracy on one hand and theoretical algorithms which have not been adopted in practice on the other. In this paper we attempt to bridge the gap between theory and practice by showing how the theoretical safe-and-complete framework can be integrated into existing assemblers in order to improve contiguity. The optimal algorithm in this framework, called the *omnitig algorithm*, has not been used in practice due to its complexity and its lack of robustness to real data. Instead, we pursue a simplified notion of omnitigs (*simple omnitigs*), giving an efficient algorithm to compute them and demonstrating their safety under certain conditions. We modify two assemblers (wtdbg2 and Flye) by replacing their unitig algorithm with the simple omnitig algorithm. We test our modifications using real HiFi data from the *D. melanogaster* and the *C. elegans* genomes. Our modified algorithms lead to a substantial improvement in alignment-based contiguity, with negligible additional computational costs and either no or a small increase in the number of misassemblies.

2012 ACM Subject Classification Applied computing → Computational biology; Mathematics of computing → Paths and connectivity problems; Theory of computation → Graph algorithms analysis

Keywords and phrases Genome assembly, Omnitigs, Safe-and-complete framework, graph algorithm, HiFi sequencing data, Assembly evaluation

Digital Object Identifier 10.4230/LIPIcs.WABI.2024.8

Supplementary Material *Software (Source Code)*: `https://github.com/algbio/practical-omnitigs` [33], archived at `swh:1:rev:bb1de69873c6b48f183e51bca2f48d2a057b8b64`

Funding This work was partially funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851093, SAFEBIO), and partially by the Academy of Finland (grants No. 322595, 328877). This material is based upon work supported by the National Science Foundation under Grants No. DBI-2138585, IIS 1453527 and OAC-1931531. Research reported in this publication was supported by the National Institute Of General Medical Sciences of the National Institutes of Health under Award Number R01GM146462. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Institutes of Health.

[1] Shared last-author contribution

Acknowledgements PM would like to thank John Hutton for early attempts to extend omnitigs to work in practice [12]. The authors wish to thank the Finnish Computing Competence Infrastructure (FCCI) for supporting this project with computational and data storage resources.

1 Introduction

Background. Genome assembly is a classical problem in bioinformatics that has received a lot of both theoretical and practical attention. On the practical side, many successful assemblers have been developed and have led to numerous biological discoveries (e.g. [23, 28]). On the theoretical side, there have been many attempts at modelling the problem and coming up with algorithms whose contiguity (and accuracy) is optimal with respect to these models (e.g. [37, 25, 5, 6, 3, 4]). Unfortunately, there remains a large gap between the theoretical and practical work, resulting in practical software with little theoretical underpinning of accuracy on the one hand and theoretical algorithms which have not been adopted in practice on the other [18].

The first group of theoretical approaches formulated assembly as the problem of finding a complete reconstruction of the genome (i.e. one string per genome). Initially, the proposed algorithm was to find an Eulerian cycle in a genome graph [26]. Later work described more sophisticated algorithms that maximised the probability of successfully finding the complete reconstruction, if one exists [2]. However, these formulations did not capture the fact that the conditions under which a complete reconstruction is feasible are extremely rare [2, 20]. Therefore, the second group of theoretical approaches formulated assembly as the problem of finding a set of sequences (called *omnitigs*) that were as long as possible and were guaranteed to be substrings of the genome (i.e. *safe*) [37]. It was possible to formally characterise how omnitigs looked like in the genome graph and how to efficiently find all possible omnitigs (i.e. a *complete* algorithm) in an idealised setting [5, 6, 3]. However, the omnitig algorithm requires complex data structures [6, 11] and omnitigs themselves are not safe in the presence of sequencing errors, missing coverage, or linear chromosomes [37]; as a result, omnitigs have not been applied in practice. Instead, most assembly software use the much simpler and more accurate notion of *unitigs*, i.e., paths whose inner nodes have in- and out- degree one.

Motivation. In this paper we attempt to bridge the gap between theory and practice by showing how the safe-and-complete framework can be integrated into existing assemblers in order to improve contiguity. To do so, we focus on a subclass of omnitigs, called *simple omnitigs*. Simple omnitigs are walks having a non-branching *core*, such that all nodes to the right of the core, except the last one, have out-degree one (i.e., a unique right extension), and all nodes to the left of the core, except the first one, have in-degree one (i.e., a unique left extension). See Figure 1 for an illustration, and Definition 1 for a formal definition.

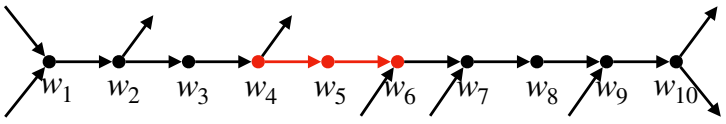
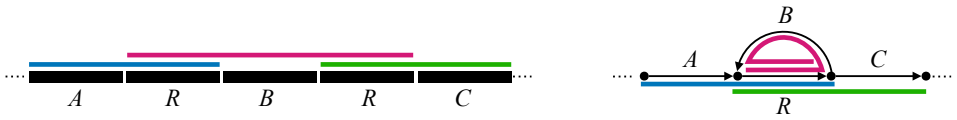
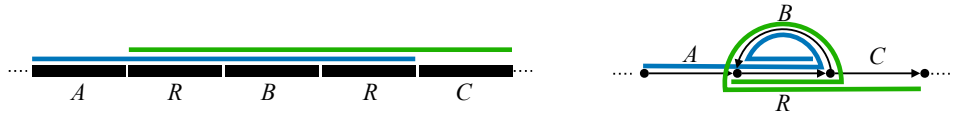


Figure 1 Example of a simple omnitig $(w_1, \ldots, w_{10})$. It has the unitig (w_4, w_5, w_6) as *core* (in red), all nodes to the right of the core, except the last one, (i.e. w_6, w_7, w_8, w_9) have exactly one out-going arc, and all nodes to the left of the core, except the first one, (i.e. w_2, w_3, w_4) have exactly one in-coming arc.



(a) Example of two occurrences of a repeat R, with simple omnitigs providing more context to R, namely AR and RC, and even capturing the full substring from the first to the second occurrence of R, namely RBR.

(b) Example where a substring ABC also appears as the variant ADC (with B replaced by D), which at the graph level induces a bubble. Simple omnitigs provide context around the variants B and D, in this case by recovering the full strings ABC and ADC.



(c) Example with the same genome as in (a), but the colored lines now indicate omnitigs, rather than simple omnitigs. Omnitigs allow to safely extend the repeat R both forwards and backwards within a single walk ($ARBR$ and $RBRC$).

Figure 2 Examples of simple omnitigs providing more context to repeats and bubbles. On the left we show the repeat structure at genome level, where we assume each labelled substring corresponds to a unitig; on the right, we show the repeat structure at assembly graph level, where each unitig is an arc. (Simple) omnitigs are shown as coloured lines.

The idea behind simple omnitigs was in fact known in the literature also before the safety framework (e.g. [19, 13, 14]), and also called Y-to-V transformation [37], though there are minor differences in edge-cases [5]. Though simple omnitigs are known to be not complete, they are nevertheless safe for a single circular chromosome [37]. On perfect data, they were shown to significantly improve length and contiguity over unitigs, while almost reaching that of omnitigs [37]. We therefore consider simple omnitigs as the "compromise candidate" for adopting the safe-and-complete framework to be used in practice.

Simple omnitigs are longer than unitigs and thus provide more context in the final assembly. They are effective for example in repeats or bubbles, as the contigs do not need to stop at the start or end of the structure, but can continue through the flanks as well. See Figures 2a and 2b for an example, and also Figure 2c for a comparison to omnitigs. Besides these examples, simple omnitigs also increase contig length in tangled regions of the graph that contain more complex structures than bubbles and repeats: whenever there is a node with indegree 1 but larger outdegree (or vice-versa), simple omnitigs can connect the unitigs starting or ending in such a node to produce longer contigs.

Contributions. In this paper, give the following contributions:

1. We prove that simple omnitigs remain safe even when there are multiple linear chromosomes, as long as no chromosome starts or ends inside them, in Section 2.2. This correctness guarantee closely mimics the recent correctness guarantee of classical unitigs [27], and shows that, in idealised conditions, simple omnitigs do not lose correctness compared to unitigs.

2. We give an algorithm for finding all maximal simple omnitigs (a *maximal* simple omnitig is one that is not a subwalk of another simple omnitig), that runs in time linear in the length of its output, namely in the total length of all maximal simple omnitigs (*linear output-sensitive time*), in Section 2.3. This matches the complexity of computing unitigs, but since unitigs do not overlap, their total length is linear also in the size of the input graph.
3. While it was already known that *all* maximal omnitigs can be computed in linear output-sensitive time [6], such an algorithm required complex data structures that are hard to implement. Our algorithm finding all maximal simple omnitigs is based on the notion of *core* of a walk (see Section 2.1), which leads to a simple and efficient procedure that can be easily applied to any graph-based assembler. To illustrate this, we integrate it into two widely-known graph-based genome assemblers, Flye [15] and wtdbg2 [29], which we discuss in Section 2.4 and in Appendix A.
4. We experimentally show on two real HiFi datasets that simple omnitigs substantially improve the assembly contiguity of our two modified assemblers, with negligible additional computational cost, in Section 3. For example, on *D. melanogaster* data, such assemblies have the same level of correctness, while having substantially higher alignment-based contiguity metrics than the original assemblers. In particular, our extension of wtdbg2 improved contiguity to be higher than that of the previously best assembler, hifiasm [7], while being more than 50 times faster than it. On *C. elegans* data, we improve the alignment-based contiguity of the best performing assembler Flye, albeit with a small increase in misassembly errors. We also improve the contiguity of wtdbg2, this time without any additional errors.

2 Methods

2.1 Definitions

A *graph* $G = (V, E)$ is defined to be directed with n *nodes* in V and m *arcs* in E. The *tail* of an arc $e = (u, v)$ is $\text{TAIL}(e) = u$ and its *head* is $\text{HEAD}(e) = v$. A w_1-w_ℓ *walk* $W = (w_1, \ldots, w_\ell)$ ($\ell \geq 1$) is a sequence of adjacent nodes. Its *tail* is $\text{TAIL}(W) = w_1$ and its *head* is $\text{HEAD}(W) = w_\ell$. A graph is *strongly connected* if each pair of nodes is connected by a walk. The concatenation of two walks $W = (w_1, \ldots, w_\ell)$ and $X = (x_1, \ldots, x_\ell)$ where $w_\ell = x_1$ is a walk $WX = (w_1, \ldots, w_\ell, x_2, \ldots, x_\ell)$. A *split* is a node with at least two outgoing arcs and a *join* is a node with at least two incoming arcs. Let $W = (w_1, \ldots, w_\ell)$ be a walk such that $\ell \geq 2$ holds. The *inner* nodes of W are the set of all nodes of W between w_1 and w_ℓ. The walk W is a *unitig* if its inner nodes have in- and out-degree one.[2] Let w_i be the first inner join of W, or w_ℓ if W has no inner join. Let w_j be the last inner split of W, or w_1 if W has no inner split. The *core* of W is its subwalk from w_j to w_i if $j < i$; otherwise, i.e. if $j \geq i$, we say that W *has no core*.

Note that cores of walks are unitigs. Indeed, if W is a walk having as core some w_j-w_i subwalk W', then W' is a unitig: since w_j is the *last* inner split of W, then all inner nodes of the w_j-w_i walk W' are not splits; likewise, if w_i is the *first* inner join of W, then all inner nodes of the w_j-w_i walk W' are not joins. Thus, all inner nodes of W' are neither joins, nor splits, and thus W' is a unitig.

[2] This is the definition that is consistently used throughout the safe-and-complete literature (e.g. [3]), but we note that it is slightly different from one used in graph compaction literature (e.g. [8, 9]).

We now have all the notions needed to define simple omnitigs:

▶ **Definition 1** (Simple omnitig). *We say that a walk $W = (w_1, \ldots, w_\ell)$ is a* simple omnitig *if it has a core. Equivalently, let w_i be the first inner join of W, or w_ℓ if W has no inner join. Let w_j be the last inner split of W, or w_1 if W has no inner split. Then we say that W is a simple omnitig if $j < i$.*

The *univocal extension* of a walk W is the maximal walk constructed by iteratively adding the unique out-neighbour of $\textsc{tail}(W)$ to the end of W and iteratively adding the unique in-neighbour of $\textsc{head}(W)$ to the beginning of W.

The *k-spectrum* S_k of a set of strings S is its set of substrings of length k. The (arc-centric) *de Bruijn graph* of a k-spectrum S_k is defined by vertex set S_{k-1} and for each $x \in S_k$, an arc from the $k-1$ prefix of x to the $k-1$ suffix of x. In a de Bruijn graph, each walk *spells* a string by spelling out its first node, and then appending the last character of each subsequent node in order.

2.2 Safety of simple omnitigs

Informally, a walk in a genome graph is *safe* if it is guaranteed to be in any genome that could have generated the genome graph. Here we will focus on the arc-centric de Bruijn graph of the reads as the genome graph. We cannot hope to generate only safe contigs if sequencing errors remain in the graph. However, given low sequencing error rates and high quality error correction algorithms, we assume an error-free setting (as done also in previous work on the safety framework see e.g. [37]). A recent work also showed that if there are gaps in coverage (i.e., genome areas from which no reads have been sequenced), then even the unitig algorithm is not safe [27]. Therefore, we assume in our theoretical model that we have error-free reads with perfect coverage, and so our genome graph is the de Bruijn graph built on the k-spectrum of the genome.

In this setting, it is already known that simple omnitigs are guaranteed to be substrings of a single-chromosome circular genome [3]. However, linear multi-chromosome genomes pose a challenge; for example, no safe and complete algorithm is known in this setting. However, as the following theorem shows, the conditions for a simple omnitig to not be substring in this setting are very narrow.

▶ **Theorem 2.** *Let $k \in \mathbb{N}$ and let S_k be the error-free k-spectrum of a linear genome with multiple chromosomes. Let $G = (V, E)$ be the arc-centric de Bruijn graph of S_k. Let L be the set of $k-1$-mers that are the first or last $k-1$-mer of some chromosome. If none of the inner nodes of a simple omnitig are in L, then its spelled string is a substring of some chromosome of the genome.*

Proof. Let $W' = (w_1, \ldots, w_j, \ldots, w_i, \ldots w_\ell)$ be a simple omnitig, where $W = (w_j, \ldots, w_i)$ is its core with $j < i$ by definition. By definition, all nodes $w_2, \ldots, w_{i-1}$ have a single incoming arc, and all nodes $w_{j+1}, \ldots, w_{\ell-1}$ have a single outgoing arc. Hence, for a walk that does not start or end with any inner node of W' to contain the arc (w_j, w_{j+1}), it needs to contain W' as subwalk.

Since each arc in E represents a k-mer of the genome, (w_j, w_{j+1}) is part of the genome, so it is contained in some chromosome. Each chromosome is an s-t walk C in G where $s, t \in L$, so there is some C that contains (w_j, w_{j+1}). By the argument above, this means that it contains W' as subwalk, so since G is error-free, W' is substring of the original genome. ◀

Thus, simple omnitigs are safe in the case of multiple linear chromosomes, as long as they do not contain the start or end k-mer of a chromosome. Note that these conditions are not complete, since it is known [3] that there are also simple omnitigs containing ends of chromosomes that are safe, based on more complex conditions about the topology of the graph.

In practice, missing coverage and errors in the reads may still cause simple omnitigs to contain more misassemblies than unitigs, even though both are safe in theory. Missing coverage or errors may cause branching nodes to miss some branches, allowing simple omnitigs to extend over a branching node where a unitig would stop. For example, if a node has two incoming arcs and one outgoing arc, then a unitig would stop there, while a simple omnitig may extend over the node. However, if the node is actually missing a second outgoing arc due to missing coverage or errors, then the simple omnitig would not be safe in the error-free graph that includes the missing branch. Hence it possibly has a misassembly at the branching node.

2.3 Computing simple omnitigs

In this section we give an algorithm to compute maximal simple omnitigs in any graph in linear output-sensitive time. Note that we cannot simply output all univocal extensions of maximal unitigs, because that would generate simple omnitigs that are non-maximal (i.e. a simple omnitig may have more than one unitig as a subwalk). Instead, our algorithm iterates over all maximal unitigs and checks that 1) if the first node has exactly one outgoing arc then it has no incoming arcs, and that 2) if the last node has exactly one incoming arc, then it has no outgoing arc. As we prove below, these two conditions hold if and only if the unitig is a core of some maximal simple omnitig. For those unitigs where these conditions hold, the algorithm outputs their univocal extension as a simple omnitig. The correctness of the algorithm follows from the following theorem:

▶ **Theorem 3.** *The* core of a maximal simple omnitig *is a walk* $W = (w_1, \ldots, w_\ell)$ *(*$\ell \geq 1$*) such that*

(a) *the core of* W *is* W*, and*

(b) *if* w_1 *has exactly one outgoing arc, then it has no incoming arcs, and*

(c) *if* w_ℓ *has exactly one incoming arc, then it has no outgoing arcs.*

Proof. Note that the cores of maximal simple omnitigs are unitigs by definition.

($\Rightarrow$) Let W' be the univocal extension of W, and a maximal simple omnitig. Then by definition, W is its core. Further, by definition, the core of a core W is W itself, so (a) holds. Next, if w_1 has exactly one outgoing arc, then, since W' is maximal, w_1 cannot have exactly one incoming arc. If it had more than one incoming arc, then W' would start at w_1 and the univocal extension of any incoming arc would contain W, and hence the whole W'. Since it is a univocal extension, it is a simple omnitig, so W' would not be maximal. Hence, if w_1 has exactly one outgoing arc, then it has no incoming arcs, so (b) holds. Symmetrically, if w_ℓ has exactly one incoming arc, then it has no outgoing arcs, so (c) holds.

($\Leftarrow$) Assume that W is not the core of a maximal simple omnitig. Then either (a) does not hold, or W is the core of a non-maximal simple omnitig, in which case its univocal extension W' is the subwalk of a maximal simple omnitig X' with a core $X \neq W$. Also, W is inside the univocal extension of X, and they share at most one node, which is the first of one and the last of the other. If W is right of X in X', then w_1 has exactly one outgoing arc, but at least one incoming arc, so (b) does not hold. If W is left of X in X', then w_ℓ has exactly one incoming arc, but at least one outgoing arc, so (c) does not hold. ◀

Since all cores are unitigs (as discussed above), to identify all maximal simple omnitigs, we first compute all maximal unitigs, then check which are also cores of maximal simple omnitigs, using Theorem 3, and then perform the maximal univocal extensions of these cores. Hence, we obtain the following theorem.

▶ **Theorem 4.** *Let $G = (V, E)$ be an arc-centric de Bruijn graph, with $|E| = m$. We can compute all maximal simple omnitigs of G in time $O(m + out)$, where out is the total length of the maximal simple omnitigs in G.*

Proof. All maximal unitigs can be identified in time $O(m)$ by a simple graph traversal, see e.g. [17, 37] among many. The idea of such an algorithm is to start at every node v that can be start of a maximal unitig, namely v has in-degree different from 1, or v has out-degree different from 1, and construct a maximal unitig by extending each out-going arc as long as the nodes traversed have in-degree and out-degree equal to 1.

Next, we apply Theorem 3 in order to check which of the maximal unitigs of G (which are at most $O(m)$ many) are also cores of maximal simple omnitigs, by checking conditions (a)–(c). Condition (a) trivially holds, because the core of a unitig is the unitig itself. The checks (b) and (c) require only checking the in- and out-degrees of the end-points of the maximal unitig, which take constant time.

Finally, computing and outputting the maximal univocal extension of all such cores satisfying Theorem 3 takes time linear in the length of the univocal extensions, namely the total length of all maximal simple omnitigs.

Thus, detecting all cores of maximal simple omnitigs takes time $O(m)$, and extending all such cores into maximal simple omnitigs takes time $O(out)$. Thus, the bound in the theorem statement holds. ◀

2.4 Injecting simple omnitigs into existing assemblers

In order for an assembler to be modifiable to output simple omnitigs instead of unitigs, it needs to work by building some kind of assembly graph and outputting unitigs from it. Furthermore, if the assembler does additional processing of the unitigs, then this processing needs to be either disabled or modified so that it becomes compatible with simple omnitigs. We identified two assemblers that lent themselves to the integration of simple omnitigs. We describe here the high-level changes we made to integrate simple omnitigs into these assemblers, with full details provided in Appendices A.1 and A.2.

The first assembler is wtdbg2 [29], which builds a "fuzzy de Bruijn graph" and then uses unitigs from this graph to build a "fragment graph." It then performs further error corrections on the fragment graph before finally outputting the consensus sequences of its unitigs as contigs. We made two modifications, as follows. First, we changed the fuzzy de Bruin graph construction so that it takes advantage of homopolymer compressed space (i.e. replacing all consecutive occurrences of a base with only one occurrence of such base). This was needed to improve the underlying quality of the graph, which is especially important for simple omnitigs. Second, we changed the final output to be the consensus of simple omnitigs (rather than of unitigs) on the error-corrected fragment graph.

The second assembler is Flye [15], which constructs a "repeat graph" followed by a repeat resolving and polishing step prior to outputting unitigs. We modified Flye by 1) disabling the post construction step of repeat resolving and polishing, and 2) outputting simple omnitigs instead of unitigs. We disabled the resolving and polishing steps because they were incompatible with injecting simple omnitigs. To make sure that this did not hamper Flye, we also ran Flye with unitigs and without the resolving and polishing step (i.e. with modification (1) but not (2)) and confirmed that the contiguity of the assembly did not substantially change.

Table 1 Properties of the reference genomes and the HiFi reads we use.

	Reference					Reads			
	N. chr	Raw len[1]	Length[1]	N50[1]	N75[1]	Num	N50[1]	Cov.[2]	SRA
D. melanogaster	7	138 mil	97 mil	18 mil	17 mil	1.1 mil	18k	94x	SRR10238607
C. elegans	6	100 mil	67 mil	12 mil	10 mil	130k	10k	19x	SRR22137522

[1]Shown in homopolymer compressed space

[2]With respect to the diploid genome length

3 Results

3.1 Experimental setup

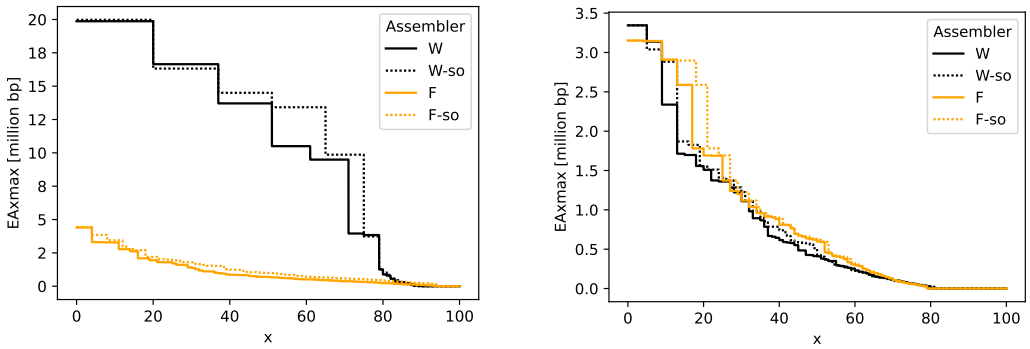
To evaluate the performance of our modified assemblers, we use two real datasets of Pacbio HiFi reads, one from *D. melanogaster* and the other from *C. elegans*. We use all chromosomes from each dataset for evaluation. We list the properties of these reference genomes and the corresponding reads in Table 1 and give full details in Appendix B. We measure accuracy and contiguity using a modified QUAST-LG [21] tool and the reference genome of the same *D. melanogaster* individual (`GCF_000001215.4`) and a different *C. elegans* individual (`GCA_000002985.3`). Following the observations of [1], we modified QUAST-LG to work in homopolymer compressed space. Otherwise, as [1] observed, QUAST-LG falsely reports misassemblies on genomic regions with long homopolymer runs.

Unlike unitigs, simple omnitigs can overlap, resulting in the same reference sequence being potentially present in more than one simple omnitig. This makes some QUAST-LG metrics misleading or inappropriate. In particular, we did not use the NGA50/NGA75 contiguity metrics that QUAST-LG reports by default. Instead, we implemented and used the EA50max and EA75max metrics, which are similar but robust against overlapping contigs. These work by aligning the contigs against the reference, identifying for each reference position the longest alignment (i.e. to a contig or part of a contig, if the contig is misassembled), and then reporting the 50th and 75th percentiles of the distribution of these lengths. For example, an EA50max value of ℓ means that 50% of the genomic positions are covered by a contig (or part of a contig, if misassembled) of length at least ℓ. The choice for the longest alignment was made because otherwise assemblers would be penalised for overlaps between longer and shorter contigs, even though the existence of the shorter contig does not reduce the quality of the longer contig. Furthermore, we modified QUAST-LG to only report at most one misassembly per reference position. Otherwise, a single misassembly in a unitig will count multiple times if there are numerous simple omnitigs containing that unitig. Appendix A.3 describes these modifications, as well as their justifications, in more detail.

In order to establish a baseline of the state-of-the-art assembly performance on our datasets, we ran also hifiasm [7], LJA [1] and HiCanu [24]. All were run with default parameters (using the predefined mode for HiFi reads in HiCanu).

3.2 *D. melanogaster*

Figure 3a shows that simple omnitigs lead to a substantial improvement in assembly contiguity. For wtdbg2, this especially happens for the shorter contigs. For example, the EA75max is increased by 147%, to 9.9 mil, when incorporating simple omnitigs. This is consistent with previous observations on error-free data [37] and due to the fact that simple omnitigs



(a) *D. melanogaster.* **(b)** *C. elegans.*

Figure 3 Assembly contiguity of wtdbg2 (denoted by "W") and Flye (denoted by "F") and their simple omnitig versions (denoted with a "-so" suffix). The y-axis shows the EA$x$max metric, for $0 \leq x \leq 100$ on the x-axis (recall that EA$x$max is computed after breaking contigs at misassemblies). Differences between the curves on smaller x pertain changes in longer contigs, and for larger x they pertain changes in shorter contigs (same as with the well-known NGAx metric family). Results are in homopolymer compressed space.

typically extend contigs through parts of the graph that are more tangled and hence contain shorter unitigs. For Flye, EA$x$max is increased across the board, however the quality of the Flye assembly pipeline is low on this dataset, with or without modifications.

Table 2 Assembly accuracy and time and memory usage on *D. melanogaster*.

	wtdbg2			Flye			Others		
	W	W-int[1]	W-so	F	F-int[2]	F-so	hifiasm	LJA	HiCanu
EA50max ($\times 10^6$)	13.7	12.2	**14.5**	0.7[†]	0.7	1.0	14.1	6.0	13.4
EA75max ($\times 10^6$)	4.0	3.8	9.9	0.3	0.3	0.5	4.8	1.7	**13.0**
N. misassemblies	1	1	4	3	3	5	1	1	0
N. contigs	405	369	363	856	951	576	1,871	925	1,110
Genome fraction (%)	90.1	89.2	89.1	94.9	94.9	94.7	95.7	95.5	95.3
Duplication ratio	1.02	1.01	1.55	1.68	1.68	1.98	1.99	1.74	1.81
Wall-clock time (s)	942	-	1,521	24,465	-	20,261	75,936	63,307	39,118
Peak memory (GiB)	17	-	15	119	-	119	105	68	19

All statistics are shown in homopolymer compressed space. We highlight in bold the best values of the EA50max and EA75max metrics.

[1]Wtdbg2 with the homopolymer modification but without simple omnitigs.

[2]Flye with the post-construction steps disabled but without simple omnitigs. [†]Note that this number is much worse than the NGA50 reported for *D. melanogaster* on `https://github.com/fenderglass/Flye`. This is likely due using here the highly heterozygous cross of the A4 and ISO1 strains of *D. melanogaster*, while the variant assembled on the Flye website is plain ISO1.

Table 2 shows the full statistics for *D. melanogaster* assemblies, including for other assemblers. Overall, compared to the results of three other assemblers on this data, our modified wtdbg2 pipeline achieves the highest contiguity on longer contigs (i.e. an EA50max of 14.5 mil, improved by 0.8 mil), at the cost of three more misassemblies. We note that the contiguity metrics EA$x$max take the misassemblies into account, i.e. the length statistics are calculated after breaking contigs apart at the misassemblies. The choice of best assembler for

Table 3 Assembly accuracy and time and memory usage on *C. elegans*.

	wtdbg2			Flye			Others		
	W	W-int	W-so	F	F-int	F-so	hifiasm	LJA	HiCanu
EA50max ($\times 10^3$)	412	425	512	620	620	**628**	596	591	342
EA75max ($\times 10^3$)	**68**	66	67	64	63	63	43	-	-
N. contigs	389	394	392	280	276	274	346	239	475
N. misassemblies	4	2	4	2	2	5	6	1	0
Genome fraction (%)	80.8	80.8	81.2	79.8	79.8	79.8	77.4	72.4	74.8
Duplication ratio	1.00	1.00	1.02	1.00	1.00	1.05	1.01	1.01	1.01
Wall-clock time (s)	208	-	226	1,269	-	830	858	695	769
Peak memory (GiB)	4.5	-	5.7	17	-	17	18	6.6	5.2

The data format is the same as in Table 2. Some of the assemblers achieve less than 75% genome fraction, hence their EA75max is undefined.

this dataset represents a trade-off; however, we stress that our goal is to show that simple omnitigs improve each assembler with respect to its unitig version, rather than with respect to other assemblers.

We also confirmed that these improvements are due to simple omnitigs and not to the other modifications we made. Table 2 shows the EA50max and EA75max numbers for the intermediate versions of these assemblers which contain all the non-simple omnitig modifications. Their contiguity does not improve relative to the original assemblers.

Table 2 also shows other assembly statistics. Notably, there is a small increase in the number of misassemblies, after our modifications. We believe that this stems from errors of the sequences stored at the graph's branching nodes (i.e. with an in- or out- degree of more than one). In particular, such nodes always lie at the ends of unitigs (by definition of unitig). An artefact of QUAST-LG is that errors in the last ≈1kbp of a contig are not counted as a misassembly, hence errors in most branching nodes do not affect the number of misassembled unitigs. However, branching nodes are often absorbed into the middle of simple omnitigs, causing QUAST-LG to report a misassembly error. Our conjecture is also backed by the fact that wtdbg2 does not output any branching nodes, but trims them from the ends of the unitigs, without reporting in their publication why this is done.

Another effect of using simple omnitigs is that there is no longer a unique contig for each region of the genome. Table 2 shows the duplication ratio, which is higher for simple omnitigs than for unitigs. This is expected and in fact desired, since a region can flank and provide context for multiple bubbles at the same time. Note that most assemblers report a duplication ratio higher than one, since *D. melanogaster* is diploid, but the reference contains only one copy of each chromosome.

Finally, our modifications also have a slight effect on the genome fraction, but, since these are minor, we did not investigate these further. The major differences in genome fraction between the different baseline assemblers is due to inherent differences between those assemblers, rather than any effect of our modifications.

3.3 *C. elegans*

Figure 3b shows that simple omnitigs improve both wtdbg2 and Flye in contiguity for longer contigs (i.e. small values of x). In Table 3 we see that simple omnitigs increase the EA50max of wtdbg2 by 24% over default wtdbg2, with the same number of misassemblies as default wtdbg2. Flye achieves the best EA50max among the baseline assemblers, and the simple omnitig variant improves this even further, at the cost of three more misassemblies.

Table 3 also shows the genome fraction and duplication ratio metrics. As with *D. melanogaster*, our modifications have a negligible effect on the genome fraction, while increasing the duplication ratio. The increase is expected, since a single reference position can now be covered by more than one contig. We note that the genome fraction of all assemblers is only $\leq 81\%$ and the duplication ratio is only ~ 1, even though the genome is diploid. We suspect that these numbers may be due to the relatively low coverage of this dataset. In any case, the genome fraction and duplication ratios are a property of the dataset and the baseline assemblers, rather than our modifications; hence, we do not investigate them further.

3.4 Time and memory usage

Tables 2 and 3 also show the time and memory usage. Our modifications did not affect the memory usage for Flye and only slightly for wtdbg2. Adding simple omnitigs to Flye decreased its run-time, since we disabled a post processing step. For wtdbg2, the running time increased by 9-61%, though it remained very fast (e.g. is is $3-25$ times faster than the next fastest assembler). Since the focus of our study was contiguity and accuracy, we did not optimise our code for speed: it is likely that the time increase could be mostly avoided by removing superfluous disk I/O.

In comparison to other assemblers, we highlight that for *D. melanogaster*, our simple omnitigs modifications of wtdbg2 increases EA50max so that it surpasses the previously best assembler under this metric, namely hifiasm, while being almost 50 times faster than hifiasm. For *C. elegans*, our simple omnitigs modification of Flye further improves EA50max, while making it faster than next best performing under this metric, hifiasm.

4 Discussion and Conclusions

Despite much work on both the theoretical and practical aspects of genome assembly, it has remained challenging to apply novel theoretical ideas to the practical setting. Omnitigs are a powerful construct within the safe-and-complete theoretical framework, however, due to their complexity, they have not been applied in practice. Instead, we have taken a simpler construct, called simple omnitigs, and shown that it both has provable theoretical guarantees and is amenable to being plugged into existing assemblers. By combining co-located unitigs, simple omnitigs can provide correct flanking context around repeats and variations.

On the theoretical side, we have shown that given a multi-chromosomal linear genome, error-free reads, and perfect coverage, simple omnitigs are safe except for some corner cases; in other words, they are guaranteed to be substrings of the original genome. Note that the latter two requirements are necessary to prove correctness even in the simpler case of unitigs [27].

On the practical side, we have injected simple omnitigs into two popular assemblers (wtdbg2 and Flye) and tested them on two HiFi datasets. On *D. melanogaster*, this gave substantial improvements in alignment-based assembly contiguity. In particular, our modifications to wtdbg2 improved the EA50max metrics to the point where they were higher

than those of the previously best assembler, hifiasm, while being more than 50 times faster. On *C. elegans*, we saw similar contiguity improvements, with simple omnitigs improving the EA$x$max metrics of both wtdbg2 and Flye; compared with other tested assemblers, our modified Flye had the highest EA50max.

The above improvements come at the cost of a small increase in the number of misassemblies. To completely prevent simple omnitigs from introducing misassemblies, the assemblers seem to require additional modifications before integrating simple omnitigs. Assembler developers usually test the accuracy of the genome graph relying on unitigs and QUAST-LG metrics. However, by outputting simple omnitigs, we are uncovering other errors in the graph not captured by QUAST-LG metrics, including in the topology that connects between unitigs. Fixing those requires deeper understanding the internals of the assemblers and, perhaps, introducing simple omnitigs at an earlier stage of the development process.

Though more work is needed to incorporate simple omnitigs into "production-ready" assemblers, our work overcomes many of the barriers that have held back omnitigs from being used in practice. First, omnitigs require complicated data structures and algorithms, while our algorithm to compute simple omnitigs is simple to implement and understand. Second, the theory of omnitig safety does not extend to the linear multi-chromosome setting, while we showed that, except for corner cases, simple omnitigs remain safe. Third, the omnitig algorithm is too slow to complete in a multi-chromosome setting, while the simple omnitig algorithm is fast. In conclusion, we hope this work helps to bridge the gap between theory and practice of genome assembly by adapting a complicated theoretical construct (i.e. omnitigs) to work in a practical setting.

References

1 Anton Bankevich, Andrey V Bzikadze, Mikhail Kolmogorov, Dmitry Antipov, and Pavel A Pevzner. Multiplex de bruijn graphs enable genome assembly from long, high-fidelity reads. *Nature biotechnology*, pages 1–7, 2022.

2 Guy Bresler, Ma'ayan Bresler, and David Tse. Optimal assembly for high throughput shotgun sequencing. *BMC Bioinformatics*, 14(S5), April 2013. doi:10.1186/1471-2105-14-s5-s18.

3 Massimo Cairo, Shahbaz Khan, Romeo Rizzi, Sebastian Schmidt, Alexandru I Tomescu, and Elia C Zirondelli. The hydrostructure: a universal framework for safe and complete algorithms for genome assembly. *arXiv preprint arXiv:2011.12635*, 2020.

4 Massimo Cairo, Shahbaz Khan, Romeo Rizzi, Sebastian S. Schmidt, Alexandru I. Tomescu, and Elia C. Zirondelli. Cut paths and their remainder structure, with applications. In Petra Berenbrink et al., editors, *STACS 2023*, volume 254 of *LIPIcs*, pages 17:1–17:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.STACS.2023.17.

5 Massimo Cairo, Paul Medvedev, Nidia Obscura Acosta, Romeo Rizzi, and Alexandru I Tomescu. An optimal $O(nm)$ algorithm for enumerating all walks common to all closed edge-covering walks of a graph. *ACM Transactions on Algorithms (TALG)*, 15(4):1–17, 2019.

6 Massimo Cairo, Romeo Rizzi, Alexandru I. Tomescu, and Elia C. Zirondelli. Genome assembly, from practice to theory: Safe, complete and *Linear-Time*. *ACM Trans. Algorithms*, 20(1):4:1–4:26, 2024. doi:10.1145/3632176.

7 Haoyu Cheng, Gregory T Concepcion, Xiaowen Feng, Haowen Zhang, and Heng Li. Haplotype-resolved de novo assembly using phased assembly graphs with hifiasm. *Nature methods*, 18(2):170–175, 2021.

8 Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016.

9 Andrea Cracco and Alexandru I Tomescu. Extremely fast construction and querying of compacted and colored de bruijn graphs with ggcat. *Genome Research*, 33(7):1198–1207, 2023.

10 C. elegans Sequencing Consortium*. Genome sequence of the nematode c. elegans: a platform for investigating biology. *Science*, 282(5396):2012–2018, 1998.

11 Loukas Georgiadis, Giuseppe F Italiano, and Nikos Parotsidis. Strong connectivity in directed graphs under failures, with applications. *SIAM Journal on Computing*, 49(5):865–926, 2020.

12 John Hutton. Extended safe contigs in the face of incomplete coverage. Masters thesis, Pennsylvania State University, 2018.

13 Benjamin Grant Jackson. *Parallel methods for short read assembly*. Iowa State University, Ph.D. thesis, 2009.

14 Carl Kingsford, Michael C Schatz, and Mihai Pop. Assembly complexity of prokaryotic genomes using short reads. *BMC Bioinformatics*, 11(1), 2010. doi:10.1186/1471-2105-11-21.

15 Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin, and Pavel A Pevzner. Assembly of long, error-prone reads using repeat graphs. *Nature biotechnology*, 37(5):540–546, 2019.

16 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

17 Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. *Genome-scale Algorithm Design: Bioinformatics in the Era of High-throughput Sequencing*. Cambridge University Press, 2023.

18 Paul Medvedev. Theoretical analysis of sequencing bioinformatics algorithms and beyond. *Commun. ACM*, 66(7):118–125, June 2023. doi:10.1145/3571723.

19 Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. Computability of models for sequence assembly. In *WABI*, pages 289–301, 2007.

20 Paul Medvedev and Mihai Pop. What do eulerian and hamiltonian cycles have to do with genome assembly? *PLOS Computational Biology*, 17(5):e1008928, May 2021. doi:10.1371/journal.pcbi.1008928.

21 Alla Mikheenko, Andrey Prjibelski, Vladislav Saveliev, Dmitry Antipov, and Alexey Gurevich. Versatile genome assembly evaluation with quast-lg. *Bioinformatics*, 34(13):i142–i150, 2018.

22 Felix Mölder, Kim Philipp Jablonski, Brice Letcher, Michael B Hall, Christopher H Tomkins-Tinch, Vanessa Sochat, Jan Forster, Soohyun Lee, Sven O Twardziok, Alexander Kanitz, et al. Sustainable data analysis with snakemake. *F1000Research*, 10, 2021.

23 Sergey Nurk, Sergey Koren, Arang Rhie, Mikko Rautiainen, Andrey V Bzikadze, Alla Mikheenko, Mitchell R Vollger, Nicolas Altemose, Lev Uralsky, Ariel Gershman, et al. The complete sequence of a human genome. *Science*, 376(6588):44–53, 2022.

24 Sergey Nurk, Brian P Walenz, Arang Rhie, Mitchell R Vollger, Glennis A Logsdon, Robert Grothe, Karen H Miga, Evan E Eichler, Adam M Phillippy, and Sergey Koren. Hicanu: accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads. *Genome research*, 30(9):1291–1305, 2020.

25 Nidia Obscura Acosta, Veli Mäkinen, and Alexandru I Tomescu. A safe and complete algorithm for metagenomic assembly. *Algorithms for Molecular Biology*, 13(1):1–12, 2018.

26 Pavel A. Pevzner, Haixu Tang, and Michael S. Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2001.

27 Amatur Rahman and Paul Medvedev. Assembler artifacts include misassembly because of unsafe unitigs and underassembly because of bidirected graphs. *Genome Research*, 32(9):1746–1753, 2022.

28 Arang Rhie, Shane A McCarthy, Olivier Fedrigo, Joana Damas, Giulio Formenti, Sergey Koren, Marcela Uliano-Silva, William Chow, Arkarachai Fungtammasan, Juwan Kim, et al. Towards complete and error-free genome assemblies of all vertebrate species. *Nature*, 592(7856):737–746, 2021.

29 Jue Ruan and Heng Li. Fast and accurate long-read assembly with wtdbg2. *Nature methods*, 17(2):155–158, 2020.

30 Steven L Salzberg, Adam M Phillippy, Aleksey Zimin, Daniela Puiu, Tanja Magoc, Sergey Koren, Todd J Treangen, Michael C Schatz, Arthur L Delcher, Michael Roberts, et al. Gage: A critical evaluation of genome assemblies and assembly algorithms. *Genome research*, 22(3):557–567, 2012.

31 Sebastian Schmidt. Flye YV. https://github.com/sebschmi/Flye, 2024.

32 Sebastian Schmidt. homopolymer-compress-rs. `https://github.com/sebschmi/homopolymer-compress-rs`, 2024.

33 Sebastian Schmidt. practical-omnitigs, 2024. Software, swhId: `swh:1:rev:bb1de69873c6b48f183e51bca2f48d2a057b8b64` (visited on 2024-08-14). URL: `https://github.com/algbio/practical-omnitigs`.

34 Sebastian Schmidt. QUAST 5.0.2 modified to be robust against overlapping contigs. `https://github.com/sebschmi/quast`, 2024.

35 Sebastian Schmidt. wtdbg2-homopolymer-decompression. `https://github.com/sebschmi/wtdbg2-homopolymer-decompression`, 2024.

36 Sebastian Schmidt. wtdbg2 YV. `https://github.com/sebschmi/wtdbg2`, 2024.

37 Alexandru I Tomescu and Paul Medvedev. Safe and complete contig assembly through omnitigs. *Journal of Computational Biology*, 24(6):590–602, 2017.

38 Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.

A Omitted Implementation Details

Our algorithm reporting simple omnitigs is implemented in Rust in [33]. The code is in the subfolder `implementation` and can be run with `cargo run -release -compute-trivial-omnitigs -non-scc` (plus arguments specifying input and output files and formats). Note that since the assemblers use bidirected graphs, we first convert the bidirected graphs to doubled graphs as in [27].

A.1 Modifying wtdbg2

The assembler wtdbg2 [29] comes with two binaries, the first `wtdbg2` which computes contigs as an overlapping sequence of reads, and the second `wtpoa-cns` which computes a consensus sequence for each of these contigs. All our modifications happen in and around the first binary. The `wtdbg2` binary first builds a fuzzy de Bruijn graph and from which it computes unitigs after some error corrections, and then builds a fragment graph from these unitigs. In the fragment graph the assembler does another round of error corrections before it reports unitigs from this fragment graph and outputs their overlapping sequences of reads. We modified `wtdbg2` to output simple omnitigs instead of unitigs from the error corrected fragment graph. For this we run `wtdbg2` once and output the fragment graph using its builtin output functions. Then we compute simple omnitigs on this graph and run `wtdbg2` again, this time loading the simple omnitigs from disk and outputting their sequences of overlapping reads instead of the unitigs in the end. We can run `wtdbg2` two times this way and get consistent node/edge identifiers since these are deterministic, even when running `wtdbg2` in parallel.

In our experiments, we also realised that since HiFi reads have mostly base duplications and deletions as errors, we can homopolymer-compress the reads to get a more accurate assembly. It turned out that when reporting simple omnitigs, we get a lot more misassemblies, but when using homopolymer compression as well, the misassemblies are not a problem, at least for D. melanogaster. To still get an assembly in uncompressed space, we decompress the contigs before calling `wtpoa-cns`. This is simple since the overlapping sequences of reads that form the contigs of `wtdbg2` carry the ids of the original reads, so we can directly replace the subsequence of characters (adjusting the start and end index according to the compression). Note that, even though we evaluate in homopolymer compressed space in the end, this decompression is still important as it makes the consensus stage run in homopolymer decompressed space, thus producing an assembly in decompressed space. This allows for a

fairer comparison of the modified wtdbg2 to the other assemblers, which all output contigs in uncompressed space. We call the variant of wtdbg2 that just uses homopolymer compression "W-int" and the variant that uses both homopolymer compression and simple omnitigs "W-so". The modified wtdbg2 is available at [36]. Homopolymer compression is done with [32] and decompression is done with [35]. Hompolymer compression and decompression is implemented in Rust and run with `cargo run -release - -compute-threads 28` (plus arguments specifying input and output files and formats). The modified wtdbg2 is run with the parameters proposed for HiFi reads: `-x ccs -g <REFERENCE_LENGTH> -t 28` (plus arguments specifying input and output files and formats).

A.2 Modifying Flye

The assembler Flye [15] runs in multiple stages. We have disabled resolving repeats and polishing, which does not seem to make any difference in our experiments. Instead, we directly report the unitigs of the repeat graph as computed by Flye. This variant is called "F-int". When using simple omnitigs, we report simple omnitigs from the repeat graph instead of unitigs by calling our tool from within Flye (and we keep resolving repeats and polishing disabled). This variant is called "F-so". The modified version of Flye is available at [31]. We run it with parameters `-g <REFERENCE_LENGTH> -t 28 -pacbio-hifi` (plus arguments specifying input and output files and formats). To disable resolving of repeats and polishing, we add `-stop-after contigger`.

A.3 Modifying QUAST-LG

We evaluate the assembly using QUAST-LG 5.0.2 [21]. Since QUAST uses minimap2 [16] for alignment which was reported in [1] to work better on homopolymer-compressed data, we run QUAST in homopolymer-compressed space by passing it a homopolymer-compressed reference and homopolymer-compressed contigs. When using simple omnitigs, contigs may overlap even when reported from a perfectly correct genome graph. Hence, none of the metrics that QUAST uses by default is able to accurately capture the contiguity of the assembly. Even the most advanced metric built into QUAST, the NGAx group of metrics (e.g. NGA50 or NGA75), produce higher numbers than they should if an assembler e.g. outputs the same contigs twice or outputs overlapping contigs.

We instead implement the EA$x$max group of metrics within QUAST. It is inspired by the E-size [30] which gives the expected value of the average length of a contig aligned to a uniformly randomly chosen base in the reference. The E-size has some weaknesses though. First, it is not robust against misassemblies. This is easily fixed in the same way as it is done for NGAx, by aligning the contigs to the reference and using continuous (enough) alignments to compute the metric. Second, the E-size uses averaging for each base, resulting in unwanted effects such as a reduced E-size for assemblers that report many short contigs that potentially overlap with others. This might not actually have been intended by the designers of the E-size, which wanted it to answer the question: "How many genes will be completely contained within assembled contigs or scaffolds, rather than split into multiple pieces?" [30]. A gene that is completely contained inside a long contig is not affected by also being contained partially within a short contig. So we choose to compute the maximum alignment to each reference base, instead of the average. Lastly, most works in genome assembly [29, 15, 1, 7, 24] use percentile-based metrics like N50, NG50 or NGA50 and do not compute the expected value over a uniform distribution of the reference bases. To stick

with existing literature, we therefore compute the EA$x$max metrics percentile-based as well.

We define the EA$x$max metric on the alignments as follows:

a) For each reference base, store the length of the maximum alignment to the reference base.

b) Sort the reference bases by length of maximum alignment descending, and report the value at index $x/100 * G$, where G is the length of the reference.

We additionally modified QUAST to output misassemblies that occur at the same position on the reference just once. Otherwise, since a single unitig could be output multiple times as part of different simple omnitigs, we would count the same misassembly on the same unitig multiple times. But a misassembly within a unitig is not introduced by reporting simple omnitigs, but instead an error produced by the assembler itself, so we only want to count it once. To compute the unique misassemblies, we first sort all misassemblies by their position on the reference, and then merge those that are less than X bp apart, however keeping the maximum length of a unique misassembly below X (instead of transitively merging misassemblies into one big unique one much longer than X). X is given by the extensive misassembly threshold in QUAST, which is 3000 by default for large genomes.

The modified version of QUAST is available at [34]. We run it with the following parameters: `-large -t 28 -min-alignment 20000 -extensive-mis-size 500000 -min-identity 90` (plus arguments specifying input and output files and formats). Increasing the minimum size of extensive misassemblies to 500000 is inspired by [24], which employ the same parameters to evaluate HiCanu on the same D. melanogaster data to avoid falsely reported misassemblies due to heterozygosity. We use the same parameters for C. elegans, to avoid misassemblies for the same reason, and additionally because our C. elegans individuum does not exactly match the reference.

B Data Availability and Reproducibility

Our experiments are implemented with snakemake [22] and available here [33]. The repository contains a conda `environment.yml` file at the root, and in this conda environment the experiments can be reproduced by running `snakemake -cores 28 report_all`. We have run our experiments on a cluster running slurm [38].

For D. melanogaster, we use the same data as in [24], using the reference with the accession `GCF_000001215.4` and HiFi reads from the sequence read archive with the accession `SRR10238607` with a median and mean read length of 24.4kbp. We filtered the reference to include only assigned sequences (keep only Chr 2L, Chr 2R, Chr 3L, Chr 3R, Chr 4, Chr M, Chr X and Chr Y).

For C. elegans, we use the official reference of the C. elegans Sequencing Consortium [10], available with accession `GCA_000002985.3`. We use HiFi reads submitted by the Korea Research Institute of Bioscience and Biotechnology with accession `SRR22137522`. The reference includes all autosomes I to V and one sex chromosome X.