HINGE: Long-Read Assembly Achieves Optimal Repeat Resolution

Govinda M. Kamath[1]*, Ilan Shomorony[2]*, Fei Xia[3]*, Thomas A. Courtade[2†], and David N. Tse[1,2†]

Long-read sequencing technologies have potential to produce gold-standard *de novo* genome assemblies, but fully exploiting error-prone reads to resolve repeats remains a challenge[1]. Aggressive approaches to repeat resolution often produce mis-assemblies, and conservative approaches lead to unnecessary fragmentation. We present HINGE, an assembler that achieves optimal repeat resolution by distinguishing repeats that can be resolved given the data from those that cannot. This is accomplished by adding "hinges" to reads for constructing an overlap graph where only unresolvable repeats are merged. As a result, HINGE combines the error resilience of overlap-based assemblers with repeat-resolution capabilities of de Bruijn graph assemblers. HINGE was evaluated on the long-read datasets from the NCTC project[2]. Besides producing more finished assemblies than the manual pipeline of NCTC based on the HGAP assembler[3] and Circlator[4], HINGE allows us to identify 40 datasets where unresolvable repeats prevent the reliable construction of a unique finished assembly. In these cases, HINGE outputs a visually interpretable assembly graph that encodes all possible finished assemblies consistent with the reads, while other approaches either fragment the assembly or resolve the ambiguity arbitrarily.

A central feature which distinguishes HINGE from existing long-read assemblers is its ability to generate an assembly graph with the maximum level of repeat resolution that is possible given the data. If a finished assembly of the genome is possible, such a graph would consist of a single cycle (in the case of a single circular chromosome). Otherwise, the next-best objective would be the construction of a *repeat graph*[5,6] where long repeats are collapsed into a single path. Such paths capture inherent ambiguities about the target genome that cannot be resolved given the data. Thus, constructing the *maximally resolved assembly* graph corresponds to minimizing the number of repeat-induced collapsed segments.

As a prerequisite to this task, one must first understand which repeat patterns can be reliably resolved given the set of reads. Early studies of this fundamental problem appeared in the context of sequencing by hybridization[7,8], and were later extended to shotgun sequencing through the notion of bridging[9]. A repeat is said to be *bridged* if at least one read completely contains one of its copies. As illustrated in Figure 1(a)-(e), for a given genome and set of reads, this allows us to define the maximally resolved assembly graph as the graph where only segments corresponding to *unbridged* repeats are collapsed. The *de novo* construction of such a graph yields the longest contigs that can be reliably constructed, and also describes the plausible arrangements of these contigs in the target genome.

The construction of a concise assembly graph has long been a problem of interest, having been a key component in assembly pipelines since the early days of sequencing projects[10]. With the introduction of de Bruijn graph-based approaches[5,6], the concept of a repeat graph was solidified as an intuitive way of representing long, potentially unresolvable, repeats. In the de Bruijn framework, the set of all k-mers is extracted from the reads, and used to build a graph where two k-mers that appear consecutively in a read are connected by an edge. This construction has the desirable property that the resulting graph is essentially Eulerian, and repeats longer than k base pairs are naturally collapsed into a single path. Furthermore, the graph construction is typically followed by repeat resolution steps, as illustrated in Figure 1(f). This allows several de Bruijn graph-based assemblers to produce a maximally resolved assembly graph where only unbridged repeats remain collapsed[5,6,11,12].

[1] Stanford University
[2] University of California, Berkeley
[3] Tsinghua University
* Contributed equally and listed in alphabetical order
† Corresponding authors: courtade@eecs.berkeley.edu, dntse@stanford.edu

In the context of third-generation long-read sequencing, however, standard de Bruijn graph approaches have not been as successful as they were in the case of short-read sequencing. Due to the high error rates associated with third-generation platforms, a large number of spurious k-mers is created, disrupting the structure of the de Bruijn graph. Recently, the concept of *solid* k-mers was proposed as a way to construct an "approximate" de Bruijn graph on a restricted set of reliable k-mers[13]. However, since overlapping reads only share a handful of solid k-mers, the resulting graph lacks the attractive features of de Bruijn graphs. In particular, the Eulerian structure is compromised and repeats are no longer properly collapsed into single paths. Overlap-based approaches, on the other hand, are more robust to read errors since they directly connect reads based on overlaps instead of first breaking them into *k*-mers. In fact, most available long-read assemblers[3,14–16] are based on the so-called overlap-layout-consensus (OLC) pipeline.

Unlike de Bruijn graphs, which are Eulerian in nature, read-overlap graphs are Hamiltonian. This means that the underlying genome sequence corresponds to a cycle that traverses every node (read) in the graph, which imposes well-known computational challenges[17]. Furthermore, the Hamiltonian paradigm does not yield a natural representation of repeat patterns, and the graph is typically riddled with unnecessary edges. In order to combat these issues, the string graph approach[18,19] was proposed, originally for the Celera assembler[1,10], and later adopted by several assembly pipelines[3,14–16]. Built via a *transitive reduction* procedure, the string graph is an overlap graph where the unique, non-repetitive parts of the genome correspond to simple, unbranched, paths. However, as illustrated in Figure 1(f), the string graph construction does not achieve optimal repeat resolution. In fact, long repeats -- both bridged and unbridged -- may result in undesirable graph motifs. In practice, only heuristics are used to combat these motifs, and building a maximally resolved overlap graph is quite difficult.

We propose HINGE as a way to simultaneously attain the error resilience of overlap graph-based approaches and the appealing graph structure and repeat-resolution capability of de Bruijn graphs. At a high level, HINGE's approach to assembly can be thought of as a "repeat-aware" variation on the greedy algorithm[20]. While the classical greedy algorithm forces each read to have a single predecessor and a single successor, HINGE permits a small number of reads to violate this rule by adding *hinges* to them. As illustrated in Figure 1(f), hinges grant reads an additional flexibility that forces bifurcations on the graph at the ends of an unbridged repeat. A comparison with the traditional greedy approach is provided in Supplementary Figure 1. In Supplementary Figure 2, we validate this approach through a controlled experiment where we assemble a simulated genome with HINGE for different sets of reads and bridging scenarios.

We evaluated HINGE on the 997 bacterial genomes of the NCTC 3000 database that were available at the time of writing this manuscript[2]. Each of these datasets consists of PacBio SMRT long reads with coverage depths mainly in the range 30x to 80x. The current NCTC manual assembly pipeline uses the HGAP assembler[2,3] to produce a list of contigs, and Circlator[4] to circularize contigs. The assembly graphs produced by HINGE with no parameter tuning for each of these datasets are available online[21], along with the contig statistics of the NCTC pipeline results, and the assembly graph produced by Miniasm[14].

For 822 of the 997 available datasets, HINGE produced a finished non-fragmented assembly graph, with additional isolated small plasmids in many cases. In 40 of these datasets, HINGE identifies unresolvable repeats, and the final graph admits distinct traversals (See Supplementary Table 1). In order to compare our results with those obtained by the NCTC manual pipeline, we restricted our attention to those datasets for which NCTC reports the results of their assembly. As shown in Supplementary Table 2, even without a circularization tool, HINGE obtains significantly more finished assemblies than the NCTC pipeline. Among the cases where HINGE produces an assembly graph with multiple traversals, we find many examples where the intuitive layout of the graph produced by HINGE resembles the idealized cases in Figure 1(a), and allows one to visually assess the unresolvable repeat pattern in the genome. In Figure 2, we analyze three such cases, and compare the graph produced by HINGE with the contigs produced by the NCTC pipeline. We see that by focusing on obtaining a maximally resolved assembly graph rather than large contig N50 values, HINGE prevents several mis-assemblies the NCTC pipeline incurred. In Supplementary Figure 3, we present nine additional such cases. In Supplementary Figure 4,

we present several cases where HINGE resolves all repeats, producing a finished circular assembly, while the NCTC pipeline instead fragments the assembly.

In Figure 2(a), we examine NCTC11022. In this example, the incorrect resolution of a repeat by the NCTC pipeline causes the circular chromosomal contig to lose a 780 kb segment, returned as a separate contig. HINGE, on the other hand, first collapses the unbridged repeat (see Supplementary Figure 6), and then resolves it as the resulting graph only allows one full traversal. As shown in Figure 2(b), the NCTC pipeline returned a single circularized contig for NCTC9964, an apparent finished assembly. However, HINGE collapses an 18 kbp inverted repeat. As we show in Supplementary Figure 7, this repeat is unbridged, implying that there are two possible traversals of the graph, and thus two possible finished assemblies, both of which are captured by HINGE's graphical output. As we point out in Supplementary Figure 5, incorrect resolution of an inverted repeat can produce a false inversion of nearly half the genome length. In this sense, HINGE's careful treatment of repeats can prevent the generation of reference genomes containing significant mis-assemblies.

In Figure 2(c), we consider NCTC9657. In this example, the NCTC pipeline returned seven unidentified contigs, but HINGE returns a single large chromosomal connected component, and three small plasmids. In this case, HINGE produces a graph motif characteristic of an unbridged triple repeat, similar to Figure 1(d). As shown by a coverage analysis in Supplementary Figure 8(a), this is indeed a triple repeat and contig 1 of the NCTC pipeline incorrectly resolves it. In addition, we examine the plasmids produced by the NCTC pipeline in Figure 2(d), and note that two of them share an unbridged repeat (see also Supplementary Figure 8(b)). Therefore, it is possible that these two contigs came from a single plasmid, and HINGE keeps them merged on the graph to retain this unresolvable ambiguity.

As illustrated by these examples, HINGE seeks to construct a user-friendly, informative, overlap graph as its main output, as opposed to most OLC assemblers, which employ assembly graphs in their inner workings[3,15,16] but output a list of long contigs. To the best of our knowledge, Miniasm[14] is the only other assembler to produce a graph as the main assembly output. However, Miniasm is based on the string graph paradigm, which does not achieve the graph layout HINGE strives for as we empirically observe[21].

Three new ingredients distinguish HINGE from other assembly pipelines. First, a pre-assembly phase is responsible for *hinging* the reads. Intuitively, hinges are placed at the beginning and end of unbridged repeats, giving reads the freedom to "bend" at those points. This allows other reads to match an *internal* segment of the hinged read, as illustrated in the third row of Figure 1(f). We point out that the beginning and end of repeats can be detected based on the alignments found by tools such as DAligner[22], which we run on the read dataset prior to the HINGE pipeline. As illustrated by *pile-o-grams* (see Supplementary Figure 10), the beginning and end of "piles" of alignments indicate the beginning and end of repeats. After annotating repeat boundaries, a careful cross-checking among annotated reads allows us to filter out annotations pertaining to bridged repeats. This is described in detail in Figure 3 and in the Online Methods.

The second algorithmic innovation in HINGE lies in the graph layout step. Most OLC assemblers adopt the string graph paradigm[18,19], which often produces assembly graphs that are unnecessarily dense. HINGE replaces the string graph algorithm with a variant of the greedy algorithm. This follows a recent line of theoretical work that found that variants of the greedy algorithm (such as the “not-so-greedy” algorithm[23] and the greedy merging algorithm[24]) can produce a sparse overlap graph without mis-assemblies. As described in Supplementary Figure 1, if hinges are correctly placed, a *hinge-aided* greedy algorithm can prevent the mis-assemblies produced by the classical greedy approach.

The third new ingredient introduced by HINGE is the use of global information to resolve repeats. As illustrated in Figure 2(a) and Supplementary Figure 5(a), simple unbridged repeats can be resolved. The sparsity and Eulerian-like nature of the graph produced by HINGE allows such repeats to be algorithmically identified straightforwardly. The graph can then be untangled, resolving the repeat.

Through this novel approach to repeat resolution and graph representation, HINGE brings a fresh perspective to the assembly problem. By focusing on the construction of a maximally resolved assembly

graph in a user-friendly fashion, HINGE is well aligned with the recent push for the standardization of graph references, as opposed to the traditional contig representation. The HINGE graph is a natural representation of a set of possible assemblies, and is amenable to further repeat resolution, which can be attempted using additional long-range information such as paired-end reads, Hi-C reads, or by leveraging biological insight. Finally, we point out that while the repeat complexity is relatively mild in the bacterial genomes we consider (as evidenced by the large number of finished assemblies), it is much more severe in higher organisms[25]. This highlights the importance of the careful treatment to repeats carried out by HINGE and the value of the proposed method to genome assembly.

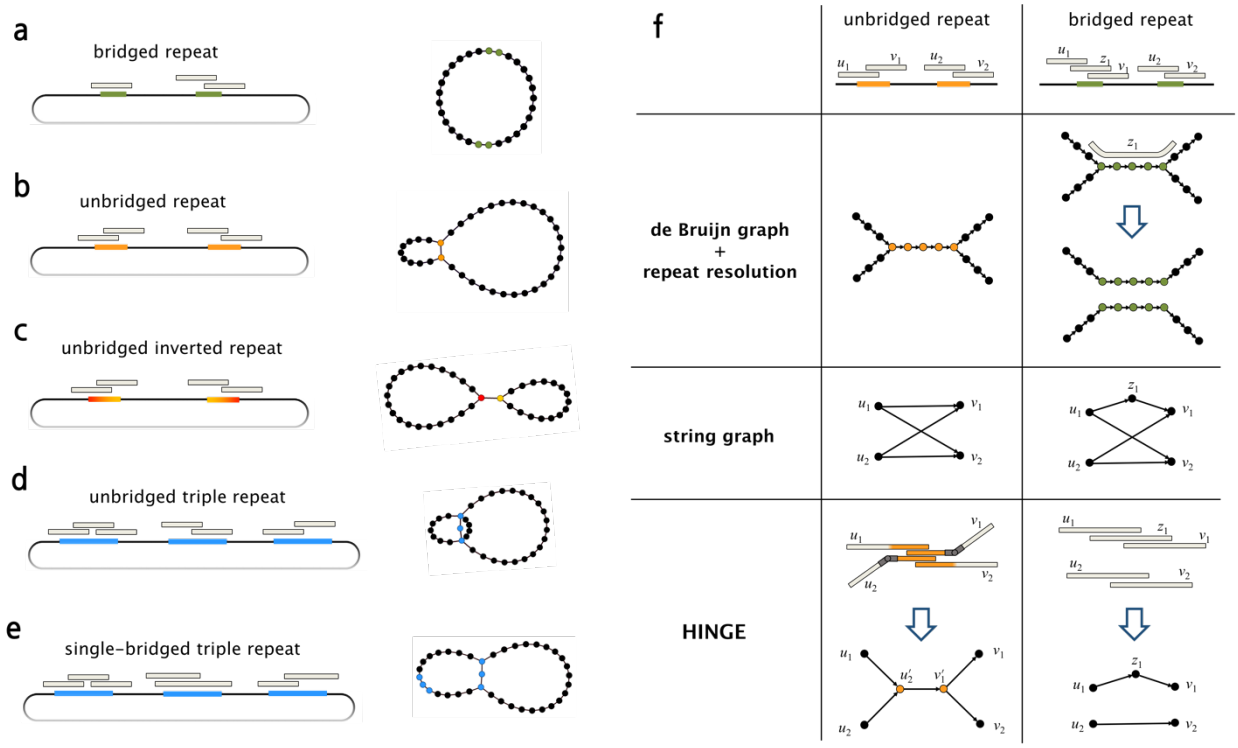

Figure 1: The goal of HINGE is to produce a maximally resolved assembly graph, where repeats that are bridged by the reads are not collapsed, and repeats that are unbridged are collapsed in a natural way, similar to what is achieved with de Bruijn graphs. **(a)** If at least one of the two copies of a repeat is bridged (green segments), the maximally resolved assembly graph should separate the two copies. **(b)** If a genome has an unbridged repeat (orange segments) then the two copies of the repeat should be collapsed into one in the assembly graph. Notice that in the absence of other *interleaved* repeats (see Supplementary Figure 3), this graph only allows one traversal, and the repeat can be resolved. **(c)** The maximally resolved representation of an unbridged inverted repeat (i.e., reverse-complemented copies) is a dumbbell-like graph. As shown in Supplementary Figure 3, an unbridged inverted repeat yields a graph with two possible traversals. **(d)** If the genome contains an unbridged triple repeat (shown in blue), then the assembly graph collapses all three copies. As in the case of the inverted repeat, such a graph allows two distinct traversals. **(e)** If one of the copies of the triple repeat is bridged, the two remaining copies behave as a regular unbridged repeat. **(f)** The representation of a bridged and an unbridged repeat in the de Bruijn graph approach, in the standard string graph approach, and according to HINGE. The de Bruijn graph approach collapses the repeated segment, which allows a natural repeat resolution step if a bridging read is found. The representation in the string graph (if there is no read entirely contained in the repeat) is an hourglass-like motif, which can be difficult to detect (though there are many heuristics to attempt to do this). HINGE identifies the start and end of an unbridged repeat and places hinges at the corresponding reads. When the repeat is not bridged, the hinges allow the correct segments corresponding to the repeat to collapse in the graph. When a repeat is bridged, however, HINGE does not place any hinges on the reads, and the repeat is resolved on the graph. This way, HINGE emulates the graph constructed via a de Bruijn graph approach, but under an overlap-graph framework.

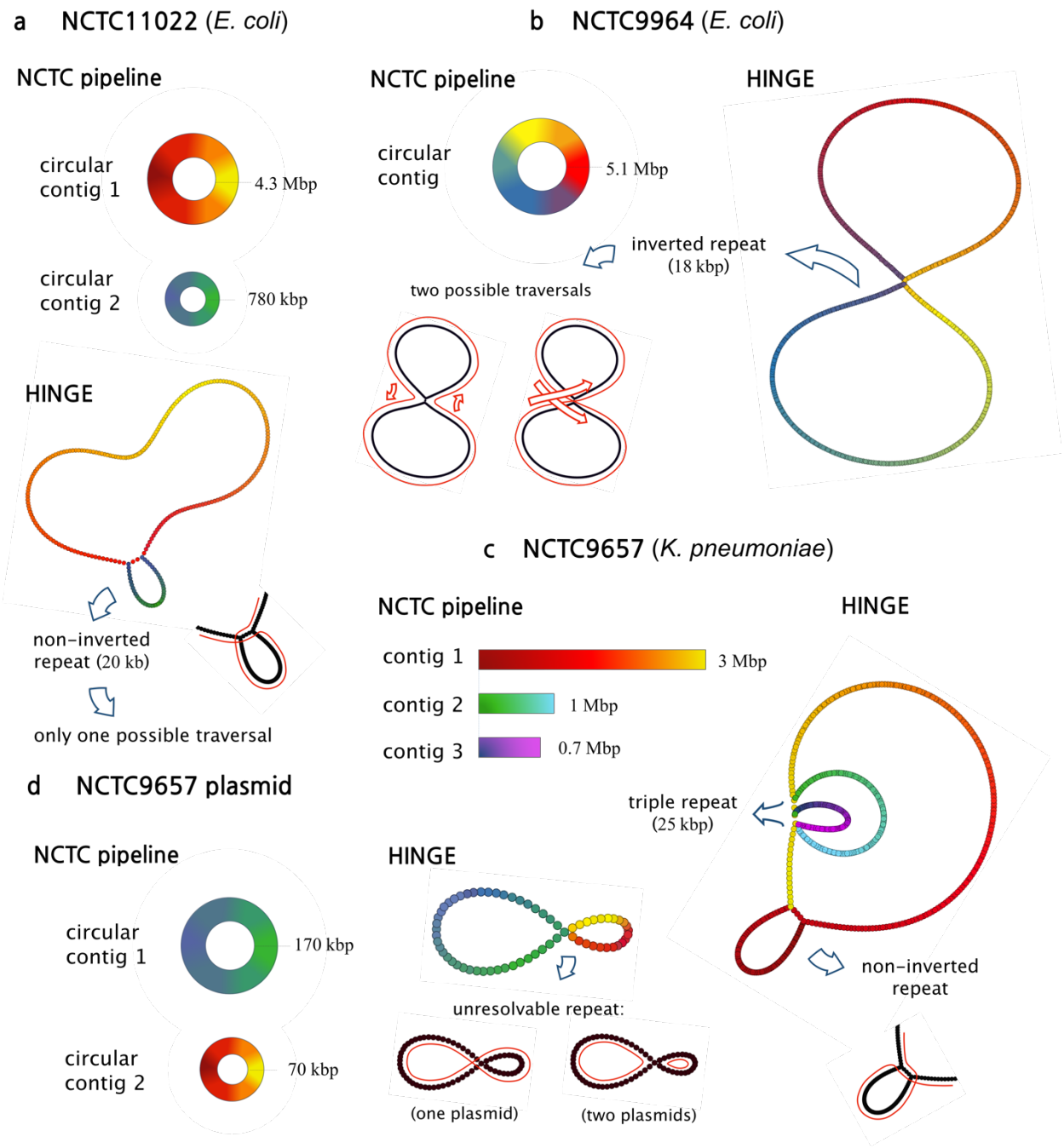

Figure 2: (a) On the NCTC11022 dataset (*Escherichia coli*), the NCTC pipeline returned two large (circular) contigs due to an incorrect resolution of a 20 kbp unbridged repeat (see Supplementary Figure 5). By first collapsing this repeat and then resolving it due to the existence of a unique traversal of the graph, HINGE produces a single large chromosomal contig of length 5.1 Mbp. The nodes in the HINGE graph are colored according to the position the corresponding reads align to in the NCTC pipeline contigs. **(b)** On the NCTC9964 dataset (*Escherichia coli*), the NCTC pipeline returned one circular chromosomal contig of length 5.1 Mbp. HINGE outputs a graph where an inverted repeat of length 18 kbp is collapsed. As shown in Supplementary Figure 6, this repeat is unbridged and hence the graph admits two distinct traversals. In order to prevent a mis-assembly, HINGE does not attempt to resolve the repeat. **(c)** The NCTC pipeline on the NCTC9657 dataset (*Klebsiella pneumoniae*) returned seven unidentified contigs

(three large ones). HINGE's output graph has one large connected component and two smaller plasmid connected components. The large component corresponds to a circular chromosome with an unbridged triple repeat and an unbridged repeat, and is a combination of the three large contigs of the NCTC pipeline. We note that contig 1 of the NCTC pipeline incorrectly resolves a triple repeat that is fundamentally unresolvable given the reads (see Supplementary Figure 7), hence creating a mis-assembly. **(d)** Here we show one of the smaller components HINGE outputs for NCTC9657. Due to an unbridged repeat, there are two possible resolutions (two plasmids or a single plasmid), and HINGE outputs the unresolved graph to prevent a mis-assembly.

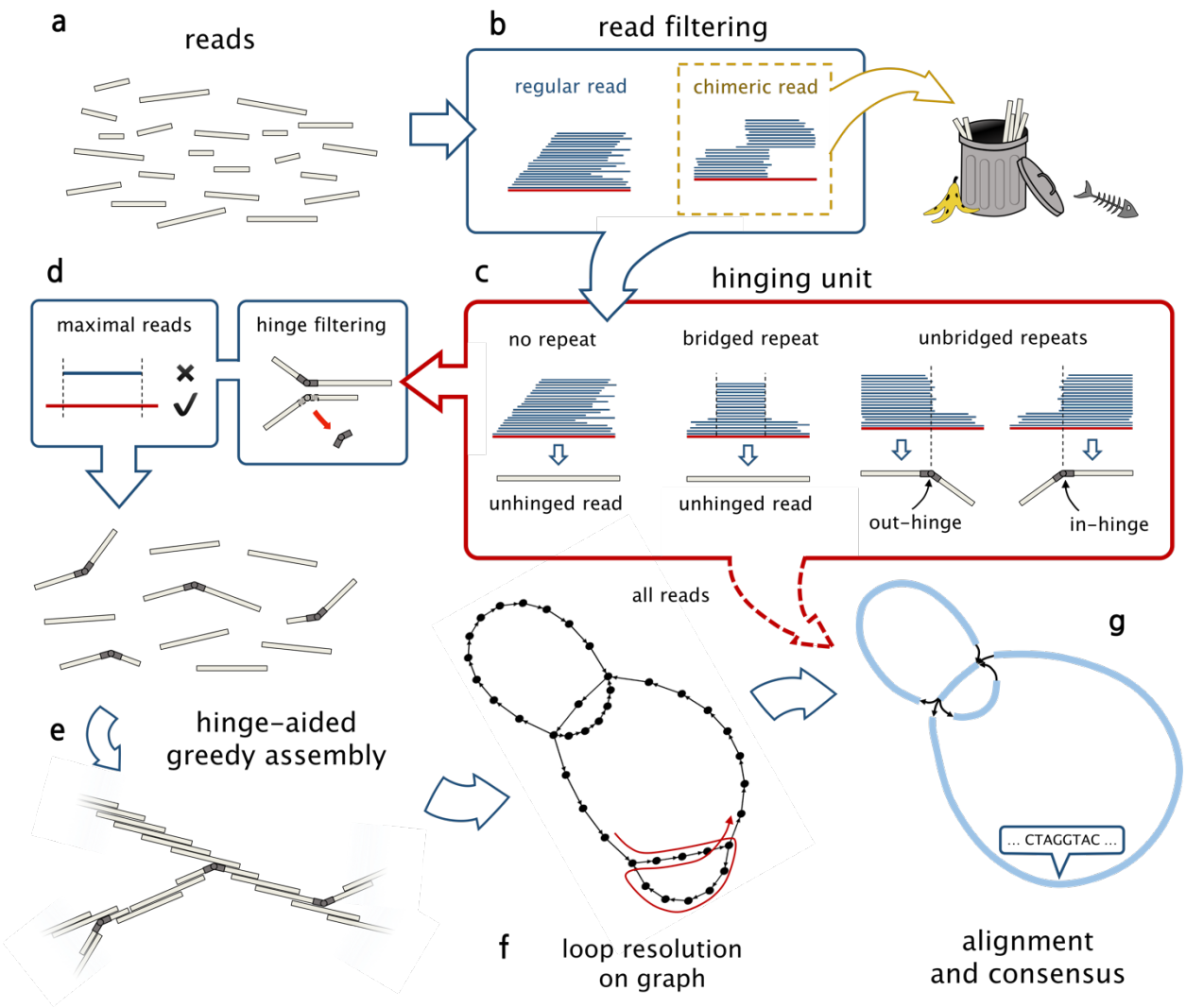

Figure 3: HINGE pipeline: **(a)** The input to the HINGE pipeline is a set of PacBio SMRT reads. We use DAligner[22] to obtain pairwise alignments between these reads, which allow a pile-o-gram to be constructed for each read. **(b)** We then detect reads with chimeric segments and discard them from the pipeline. Reads with chimeric segments have a sharp change in the set of reads shown on the pile-o-gram, or segments where no other read aligns. Reads with these patterns can be detected and discarded from the assembly pipeline. **(c)** The remaining reads are then passed on to the hinging unit. Here we identify reads that contain part of a repeat, but do not bridge it. A large increase in the number of matches starting (resp. ending) at the same point in the pile-o-gram indicates the start (resp. end) of a repeat. This yields repeat annotations on the reads. We then check if reads with repeat annotations bridge a repeat (as explained in Supplementary Figure 8). If not, we place a hinge on the read. **(d)** In order to make sure that we only have one hinge for the beginning (resp. end) of each repeat, we match hinges that correspond to the beginning of the same repeat, and only keep the hinge whose read extends the most into the repeat (See Supplementary Figure 9 for details). We also identify maximal reads, i.e., reads that are not contained in any other read, and pass them on to the layout unit. **(e)** In the layout unit we use a hinge-aided greedy algorithm to build the read overlap graph. The basic algorithm is the greedy algorithm,

but the matches considered while picking the greedy predecessor/successor of a read include internal matches starting/ending at hinges. (see Supplementary Figure 9 for details). **(f)** After obtaining the read-overlap graph, we resolve repeats that admit only one traversal. This is done when all traversals of the graph must resolve the repeat in the same way. **(g)** Finally, we extract contigs from the read-overlap graph and use standard consensus methods to return the graph corresponding to our assembly with the corresponding sequences.

Methods

The HINGE assembly pipeline, depicted in Figure 3, has been designed to assemble PacBio SMRT reads. Here we describe technical aspects of the workflow.

Read database and alignment

We use DAZZ_DB[26] to maintain a database of the PacBio reads. We use DAligner[22] to get pairwise alignments between all reads.

No initial error-correction step

Unlike most available long-read assembly pipelines, HINGE bypasses an initial error correction step. To the best of our knowledge, Miniasm[14] is the only other long-read assembler that dispenses with this step. The fact that standard overlappers like DAligner[22] can obtain pairwise alignments at error rates of around 15% allows us to use this approach. By avoiding the typical initial error-correction step, we avoid the risk of removing the heterogeneity that may differentiate certain approximate repeats, which can cause them to become indistinguishable given the error-corrected reads. The idea of bypassing the error correction step was proposed in talks by Gene Myers.

Chimeric read filter

Chimeric reads are the result of a sequencing error, and are usually made up of multiple segments that come from different parts of the genome. If not properly handled, these reads create mis-assemblies. HINGE's chimera filter unit is the first place in the pipeline where pile-o-grams (See Supplementary Figure 9(f)) are used. We mark a read segment as chimeric if the set of reads aligned to it undergoes an abrupt change. We also mark a read segment as chimeric if the number of matches goes below a fixed threshold. For each read, we keep the longest segment without any chimeric segments. If this segment is shorter than a threshold, we discard the read completely.

Hinge placement

The next step in the HINGE pipeline is the hinge placement. To do so, we first annotate repeats by detecting the start/end of large number of internal matches on reads. On the pile-o-gram, this usually corresponds to a large pile of matches starting/ending at the same point (See Supplementary Figures 9 and 10). We then verify whether the repeat annotation corresponds to a repeat that is bridged by that read. This is done by making sure that the internal matches starting (resp. ending) at the repeat annotation do not end in a pile (resp. do not start in a pile). If a repeat annotation is detected as bridged, no hinge is placed (although the annotation is maintained for downstream use). Otherwise an in-hinge or out-hinge is placed, depending on whether that is the beginning or the end of the repeat. This process is illustrated in Supplementary Figure 11. In this step, we also filter out the reads which are contained in other reads to retain only maximal reads.

Hinge processing

After placing the hinges, we attempt to remove many of them. The reason for that is two-fold. First, if a repeat is bridged, we do not want the repeat to collapse, so no hinges should be placed on the corresponding reads. Second, for each unbridged repeat, we only want one read to have an in-hinge and one read to have an out-hinge. As illustrated in Figure 3(d), we want to keep the in-hinge and out-hinge on the reads that extend the most into the repeat.

We achieve this via a two-step hinge filtering. In the first step, we take each read containing a hinge, and consider all of its overlaps. By an overlap, we mean a match between the end of a read and the beginning of another read. If the match instead occurs at the interior of at least one of the reads, we refer to it as an

internal match. We remove an in-hinge if the read has a (suffix) overlap that starts before the hinge. Similarly, we kill an out-hinge if the read has a (prefix) overlap that ends after the hinge.

In the second stage of hinge filtering, we kill an in-hinge if another read has an in-hinge corresponding to the start of the same repeat, but the read extends more into the repeat. Similarly, an out-hinge is killed if another out-hinge is found on a read that extends more into the repeat. These two filtering steps are illustrated in Supplementary Figure 11.

We point out that, in the first filtering stage, when a read has its hinge removed due to a bridged repeat, we mark this read as "poisoned". This annotation is used in the greedy layout algorithm to prevent a bifurcation from occurring at a bridged repeat, guaranteeing that the two copies remain separate. We describe this in detail in Supplementary Figure 12.

Hinge-aided greedy layout

Once the reads have been properly hinged, we move on to the graph layout step, where we employ a variation of the greedy algorithm. Each read picks its left extension to be its longest prefix match, and its right extension to be the longest suffix match. Unlike in the classical greedy algorithm, we do not restrict our search to overlaps. Instead, we consider (non-poisoned) overlaps and internal matches that start/end at a hinge.

Repeat resolution

Once constructed, the graph allows us to identify certain repeats that, although unbridged, can still be resolved based on the graph layout. As illustrated in Figure 2(a) and Supplementary Figure 5(a), when a repeat loop allows only one possible traversal, the loop can be untangled. We point out that the sparse and Eulerian-like nature of the graph produced by the hinge-aided greedy algorithm are important to allow this repeat resolution to be done in an automated fashion.

Handling Read Orientation and Double-Strandedness

Since the orientation of the reads is unknown, as it is typical in all assembly pipelines one must consider each read and its reverse complement. Hence for each read we in fact create two nodes in the graph, and the constructed graph is symmetric. At the end of the graph construction, for visualization purposes, we overlay each node and its reverse complement.

Consensus

In order to generate consensus sequences for the resulting graph contigs, we first create a draft assembly by simply concatenating sections of the error-prone reads corresponding to unbranched paths on the graph. We then consider the alignment of all the original reads onto these draft contigs, and utilize a majority-based consensus to clean up these draft sequences. We reuse some code from FALCON[15] to perform this task. The result is output as a GFA file. We point out that the final contig sequences can be optionally run through Quiver[3] to further polish the assembly.

Graph Visualization

All assembly graphs produced by HINGE were visualized using Gephi[27].

Software availability

The HINGE assembler is available online at: https://github.com/fxia22/HINGE.

Acknowledgements

The authors would like to thank Shoudan Liang and Jason Chin of Pacific Biosciences for useful discussions, and Lior Pachter of UC Berkeley for helpful comments and suggestions during the preparation of this manuscript. The authors are grateful to Nick Grayson and Julian Parkhill of The Wellcome Trust Sanger Institute for feedback and help with interpreting the results on the NCTC datasets. GMK would also like to thank John Lamping of Human Longevity Inc., chatting with whom drove him to take a data-driven approach to this project. Finally, GMK, IS, and FX would like to thank Gene Myers for presentations and work that were an inspiration to them, and for several tools that made this work possible.

Author Contributions

GMK and IS designed the algorithm. FX implemented a testbed to test and experiment with assembly algorithms. GMK, IS, and FX implemented the HINGE algorithm, ran it on the dataset, visualized and interpreted the results. TAC and DNT supervised the project. All authors wrote the paper.

Competing Financial Interests

The authors declare no competing financial interests.

Bibliography

1. Myers, E. W. A History of DNA Sequence Assembly. *Information Technology* **58,** 126–132 (2016).
2. Public Health England reference collections - Wellcome Trust Sanger Institute. Available at: http://www.sanger.ac.uk/resources/downloads/bacteria/nctc/. (Accessed: 25th June 2016)
3. Chin, C.-S. *et al.* Nonhybrid, finished microbial genome assemblies from long-read SMRT sequencing data. *Nat. Methods* **10,** 563–569 (2013).
4. Hunt, M. *et al.* Circlator: automated circularization of genome assemblies using long sequencing reads. *Genome Biol.* **16,** 294 (2015).
5. Pevzner, P. A. & Tang, H. Fragment assembly with double-barreled data. *Bioinformatics* **17 Suppl 1,** S225–33 (2001).
6. Mulyukov, Z. & Pevzner, P. A. EULER-PCR: finishing experiments for repeat resolution. *Pac. Symp. Biocomput.* 199–210 (2002).
7. Ukkonen, E. Approximate string-matching with q-grams and maximal matches. *Theor. Comput. Sci.* **92,** 191–211 (1992).
8. Pevzner, P. A. DNA physical mapping and alternating Eulerian cycles in colored graphs. *Algorithmica* **13,** 77–105 (1995).
9. Bresler, G., Bresler, M. & Tse, D. Optimal assembly for high throughput shotgun sequencing. *BMC Bioinformatics* **14 Suppl 5,** S18 (2013).
10. Myers, E. W. *et al.* A Whole-Genome Assembly of Drosophila. *Science* **287,** 2196–2204 (2000).
11. Butler, J. *et al.* ALLPATHS: de novo assembly of whole-genome shotgun microreads. *Genome Res.* **18,** 810–820 (2008).
12. Peng, Y., Yu, P., Leung, H. C. M., Yiu, S. M. & Chin, F. Y. L. IDBA – A Practical Iterative de Bruijn Graph De Novo Assembler. *Lecture Notes in Computer Science* 426–440 (2010).

13. Lin, Y. *et al.* Assembly of Long Error-Prone Reads Using de Bruijn Graphs. *Biorxiv* (2016). doi:10.1101/048413
14. Li, H. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics* (2016). doi:10.1093/bioinformatics/btw152
15. Chin, C.-S. *et al.* Phased Diploid Genome Assembly with Single Molecule Real-Time Sequencing. *Biorxiv* (2016). doi:10.1101/056887
16. Berlin, K. *et al.* Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nat. Biotechnol.* **33,** 623–630 (2015).
17. Nagarajan, N. & Pop, M. Parametric complexity of sequence assembly: theory and applications to next generation sequencing. *J. Comput. Biol.* **16,** 897–908 (2009).
18. Myers, E. W. Toward simplifying and accurately formulating fragment assembly. *J. Comput. Biol.* **2,** 275–290 (1995).
19. Myers, E. W. The fragment assembly string graph. *Bioinformatics* **21 Suppl 2,** ii79–85 (2005).
20. Tarhio, J. & Ukkonen, E. A greedy approximation algorithm for constructing shortest common superstrings. *Theor. Comput. Sci.* **57,** 131–145 (1988).
21. HINGE on NCTC 3000. Available at: https://web.stanford.edu/~gkamath/NCTC/report.html. (Accessed: 26th June 2016)
22. Myers, E. W. Efficient Local Alignment Discovery amongst Noisy Long Reads. *Lecture Notes in Computer Science* 52–67 (2014).
23. Shomorony, I., Kim, S., Courtade, T. & Tse, D. N. Information-Optimal Genome Assembly via Sparse Read-Overlap Graphs. *to appear in Proc. of ECCB 2016*
24. Shomorony, I., Kamath, G. M., Xia, F., Courtade, T. A. & Tse, D. N. C. Partial DNA Assembly: A Rate-Distortion Perspective. *arXiv* 2016 (2016).
25. Koren, S. *et al.* Reducing assembly complexity of microbial genomes with single-molecule sequencing. *Genome Biol.* **14,** R101 (2013).

26. Myers, E. W. thegenemyers/DAZZ_DB. *GitHub* Available at: https://github.com/thegenemyers/DAZZ_DB. (Accessed: 1st July 2016)
27. Bastian, M., Heymann, S. & Jacomy, M. Gephi: an open source software for exploring and manipulating networks. in *International AAAI Conference on Weblogs and Social Media* (2009).