Optimal Sequence Assembly via Sparse Read-Overlap Graphs

Ilan Shomorony[1], Samuel Kim[1], Thomas Courtade[1], and David Tse[2]
[1] UC Berkeley, [2] Stanford University

Abstract

Emerging long-read sequencing technologies have the potential to generate less fragmented *de novo* genome assemblies by reducing the ambiguity resulting from repeats in the sequence. In this context, assembly algorithms based on the de Bruijn graph framework seem unsuited to fully exploit the power of long reads. Instead, read-overlap based approaches – and in particular string graphs – are expected to play a central role in the next generation of assemblers, allowing near perfect assembly of whole genomes.

A key challenge in performing assembly using string graphs is that the true sequence corresponds to a (generalized) Hamiltonian path on the graph, and under most formulations the assembly problem becomes NP-hard. However, it is unclear a priori whether the instances where the problem is computationally difficult are indeed practically relevant. A more basic question can be asked from an information-theoretic point of view: when does the set of reads contain enough information to allow correct and unambiguous reconstruction of the true sequence?

In this work, utilizing insights from these feasibility considerations, we devise an algorithm to construct a sparse read-overlap graph that contains all information required for assembly. We describe sufficient conditions under which the assembly problem becomes an Eulerian path problem on the sparse read-overlap graph, and can thus be solved in linear time. By considering a probabilistic model for the sampling of the reads, these conditions can be translated into read length and coverage depth requirements, which are shown to nearly match information-theoretic lower bounds. We conclude that most instances of the assembly problem that are informationally feasible are also efficiently solvable.

1 Introduction

Modern DNA sequencing technologies are based on a two-step process. First, tens or hundreds of millions of fragments from random and unknown locations of the DNA sequence are read via *shotgun sequencing*. These fragments, called reads, are then merged to each other with the goal of recovering the true underlying genome. The task of reconstructing the original sequence from a large number of short reads is known as the *Assembly Problem* and is one of the fundamental algorithmic problems in bioinformatics.

While the assembly problem (AP) does not have a unique "canonical" formulation agreed upon by all bioinformaticians, the problem is widely regarded as computationally hard [1, 2]. A classical formalization, based on the "Occam's razor" (or parsimony) principle, models the AP as the *Shortest Common Superstring* problem (SCSP). Since SCSP is known to be NP-hard, early studies on the problem focused on the development of approximation algorithms [3–5]. Recent works on assembly algorithms, on the other hand, tend to model the AP as the problem of finding an appropriate path on a graph constructed from the reads. Graph-based approaches to AP are usually classified into two categories: approaches based on *read-overlap graphs*, and approaches based on *de Bruijn graphs*.

In a read-overlap graph, each vertex corresponds to a read and edges are used to represent the overlaps between reads. In this setting, the objective is to find a path that visits every vertex at least once, or a *generalized Hamiltonian path*[1]. Notice that in the framework of generalized Hamiltonian paths, long repeats can be naturally represented as nodes (or paths) that are visited multiple times by the sequence path, as illustrated in Fig. 1. As shown in [2], if we model AP as the problem of finding the shortest generalized Hamiltonian path or a generalized Hamiltonian path of a desired length, we once again have an NP-hard formulation.

Part of the excitement surrounding the introduction of the de Bruijn graph-based formulation [6] is that the sequence is no longer a generalized Hamiltonian path, but rather a path that traverses every *edge* on the graph. In a de Bruijn graph [7], vertices correspond to length-K substrings – or K-mers – extracted from the reads, and two K-mers are connected by an edge if they appear consecutively in the same read. Thus all edges should be included in the sequence path. We are now looking for an *Eulerian* path (if we require each edge to be traversed exactly once) or a *Chinese* path [1] (if we require each edge to be traversed at least once), both of which can be found in polynomial time. Nevertheless, in order to make sure that the resulting sequence is consistent with all the reads, one must require the Eulerian (or Chinese) path

[1]We use the term Generalized Hamiltonian path (or cycle) following the terminology in [2]. A path (or cycle) that visits every node at least once is also refered to in the literature as a *spanning* path (or cycle).

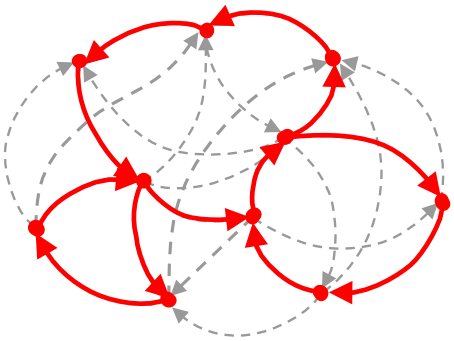


Figure 1: Illustration of a generalized Hamiltonian path. In a read-overlap graph, nodes that are visited multiple times by the path correspond to repeats in the sequence.

to contain the set of short paths corresponding to each read. This additional constraint results in a problem known as the Eulerian Superpath Problem [6], which is also known to be NP-hard [2].

In light of all these computational hardness results, it is natural to presume that the AP is fundamentally difficult. However, if we leave aside the question of the computational tractability of these formulations, is it clear that they lead to the reconstruction of the *true* sequence? As pointed out in [8], parsimony-based formulations often handle long repeats in an incorrect way. Real genomes often contain repeats that are longer than the sequencing reads, and these repeats tend to be under-represented in the shortest shortest sequence that is consistent with the data. To circumvent this issue, [8] proposed a maximum likelihood (ML) formulation for assembly. While such a formulation prevents the over-collapsing of repeats, devising algorithms to find the ML sequence given the read data is a daunting task, and existing approaches rely on the assumption of high coverage [8].

But are such high coverages fundamentally required for a complete assembly? A basic question can be formulated from an information-theoretic point of view: at what coverage do the reads contain enough *information* for the AP to be feasible? One can easily devise low-coverage instances of the problem where the absence of reads from some part of the sequence deem the task of reconstructing the true sequence *impossible*. In such cases, the computational hardness of the AP is irrelevant as there is not enough information for the sequence to be recovered, and the optimal solution under any formulation will in general be incorrect. Therefore, a more careful treatment of the assembly task should be concerned with two questions:

- When is there enough information in the set of reads to render the AP feasible?
- Can one devise efficient algorithms which recover the true sequence for the feasible instances of the AP?

The present paper should be understood as following the line of work of [1, 2, 8] in the study of the basic formulation of the AP and the computational challenges associated with it, but having the aforementioned questions as starting point.

This information-theoretic approach to the AP was first considered by Bresler et al in [9]. The authors characterized classes of instances of the AP where the set of reads do not contain enough information for the true sequence to be unambiguously reconstructed. More precisely, two types of repeat patterns were identified as the main informational bottlenecks: interleaved repeats and triple repeats. As illustrated in Fig. 2, if the true sequence contains a pair of interleaved repeats or a triple repeat, and the reads are not long enough to *bridge* these repeats, the read data is inherently ambiguous, and the assembly problem is essentially infeasible. Based on these observations, the authors of [9] derived necessary *bridging* conditions that the set of reads must satisfy to prevent the two ambiguous scenarios in Fig. 2(a) and (b) from occurring. As illustrated in Fig. 2(c), if a read bridges one of the copies of repeat C (i.e., starts in segment B1 and ends in segment D1), the ambiguity is resolved.

These bridging conditions naturally determine a minimum read length requirement ℓ_{crit} that the reads must meet in order for the AP to have a unique solution. Moreover, by considering a probabilistic framework where N reads of length L are sampled independently and uniformly at random from the sequence, it is possible to characterize the pairs (N, L) that guarantee that these bridging conditions are met with a target probability $1 - \epsilon$. This yields an information-theoretic lower bound on the (N, L) pairs for which the assembly of a specific genome is feasible, as illustrated by the black curves in Fig. 3. For any (N, L) point below this curve, no assembly algorithm can reconstruct the true sequence within

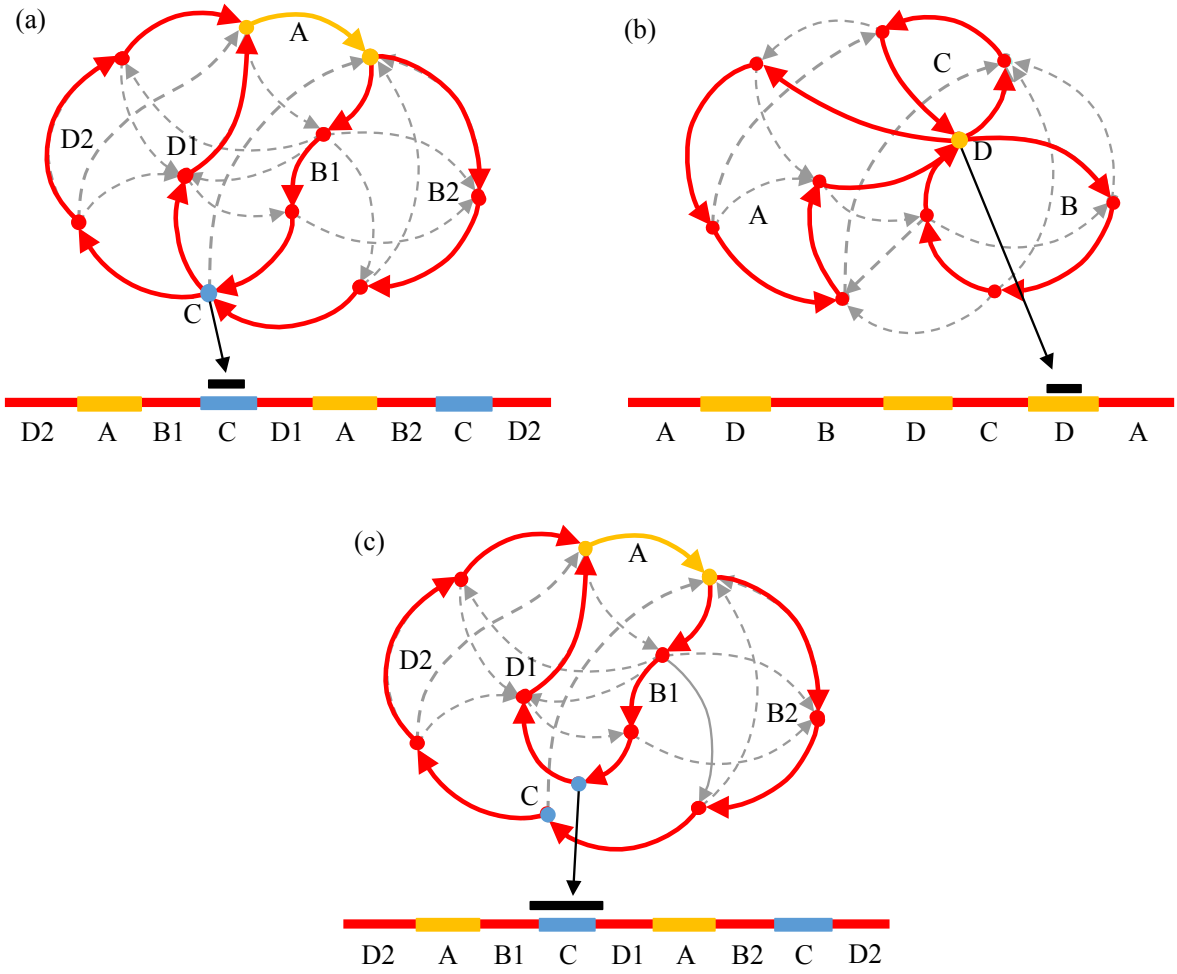


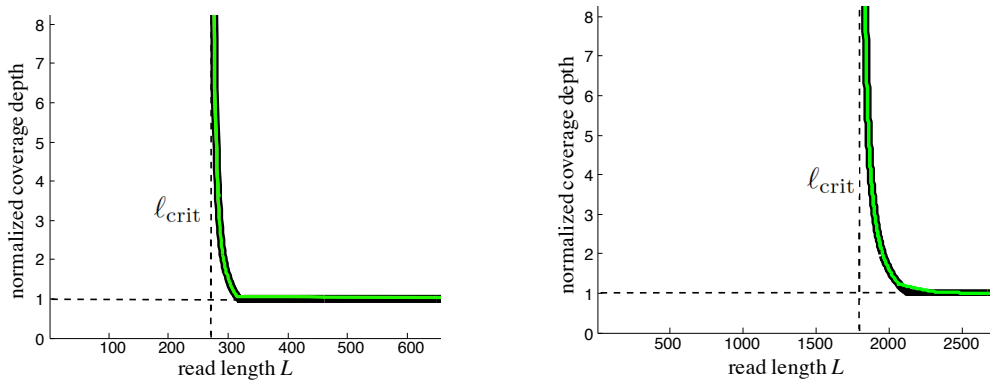
Figure 2: The two basic scenarios where the set of reads does not contain enough information to allow perfect assembly. (a) The existence of a pair of interleaved repeats (A and C) that are both longer than the read length L makes it impossible for any algorithm to distinguish between the cycles A-B1-C-D1-A-B2-C-D2 and A-B1-C-D2-A-B2-C-D1. (b) The existence of a triple repeat that is longer than L makes it impossible for any algorithm to distinguish between the cycles A-B-C and A-C-B. (c) If a read bridges one of the copies in the pair of interleaved repeats, the ambiguity is resolved as A-B1-C-D2-A-B2-C-D1 no longer corresponds to a cycle on the read-overlap graph.

the target success probability of $1-\epsilon$. In addition, [9] describes a de Bruijn graph based algorithm – Multibridging – whose performance, evaluated in terms of the (N, L) pairs for which the algorithm is successful with probability $1-\epsilon$, nearly matches this lower bound. Hence, the combination of the algorithm-independent lower bound (thick black curve in Fig. 3) and the upper bound provided by Multibridging (green curve in Fig. 3) establishes the fundamental feasibility region in terms of N and L for the assembly problem.

1.1 Third-generation sequencing: Back to Read-Overlap Graphs?

One of the main features of the feasibility curves introduced in [9] is a critical read length ℓ_{crit}, below which assembly is infeasible. This critical read length is a function of the repeat statistics of a given genome and, as shown in [9], even for small bacterial genomes, ℓ_{crit} can be in the thousands. This renders the task of perfect assembly from short-read sequencing technologies impossible, and underscores the importance of the emerging long-read sequencing technologies.

In the context of this "third generation" of sequencing technologies, de Bruijn graph based assemblers, which were rather successful in the assembly of short-read sequencing data [6, 11–13], seem unsuited to fully exploit the power of long reads. The first reason is that by shredding the reads into K-mers, it is more difficult to take advantage of the



(a) *S. aureus* (b) *R. sphaeroides*

Figure 3: The thick black curve is a feasibility lower bound for any algorithm, and the green line represents the performance of the Multibridging algorithm [9]. In the vertical axis, the normalized coverage depth corresponds to N/N_{LW}, where $N_{LW} = \frac{G}{L}\log\left(\frac{G}{\epsilon}\right)$ is the Lander-Waterman coverage depth [10].

full length of the reads in order to resolve repeats and obtain long contigs. The second one is that the de Bruijn graph construction is very sensitive to read errors, as many false K-mers are created due to errors in the reads, and the high error rates associated with long-read sequencing technologies make this approach impractical.

Read-overlap approaches, on the other hand, by not breaking the reads into small K-mers, and by connecting reads thats contain long *approximate* overlaps, have the potential to generate much longer contigs, and in an noise-robust way. In fact, most available long-read assemblers [14–16] make use of the read-overlap approach.

However, the current theoretical understanding of the read-overlap framework is limited. The existence of repeats in the sequenced genome creates many *spurious* edges on the read-overlap graph. This makes the problem of identifying the correct sequence on the graph very challenging, and naturally cast as the (generalized) Hamiltonian path problem. While the work done in [9] shows that the informational limits of the assembly problem can be achieved (in an error-free scenario) by a de Bruijn graph based algorithm, it is unclear whether read-overlap approaches can achieve the same performance. So can we indeed achieve *de novo* whole-genome assembly of long-read sequencing data using this approach? Two natural questions arise:

- Can read-overlap based approaches achieve the informational limits characterized in [9]?
- For the informationally feasible instances of the assembly problem, can one find the true path on the read-overlap graph efficiently?

In this work, we answer both of these questions in the affirmative. Inspired by insights from the information-theoretic necessary conditions introduced in [9], we describe a new algorithm for the construction of a *sparse* read-overlap graph. Although sparse, we show that this read-overlap graph is "sufficient" for assembly, in the sense that with high probability the true sequence corresponds to a path on the graph as long as we are in the information-theoretic feasibility region (see Fig. 3). Furthermore, we prove that the true sequence corresponds to a path that traverses every edge of the sparse read-overlap graph. This reduces the assembly problem to the problem of identifying a Chinese Postman path, which can in general be solved efficiently. In fact, the instances of the Chinese Postman Problem we obtain can be further reduced to an instance of the Eulerian path problem, which can be solved in linear time. Therefore, just like the de Bruijn graph was used as a way to cast the AP as an Eulerian path problem, the techniques introduced in this paper can be seen as an alternative path towards an Eulerian path problem, but via sparse read-overlap graphs, as illustrated in Fig. 4. We conclude that, while the AP when stated in its full generality may be computationally intractable, the informationally feasible instances can be efficiently solved.

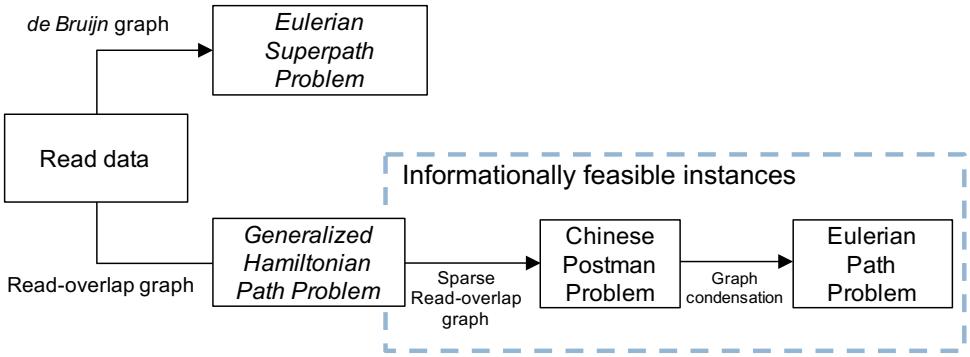


Figure 4: Our algorithm can be seen as reducing the assembly problem to an Eulerian path problem for the informationally feasible instances of the assembly problem.

2 Preliminaries

Let x be a string of ℓ symbols from the alphabet $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$. We let $|x| = \ell$ be the length of the string, and $x[i]$ be its ith symbol. A substring of x is a contiguous interval of the symbols in x, and is denoted as $x[i : j] \triangleq (x[i], x[i+1], ..., x[j])$. A substring of the form $x[1 : \ell]$ is called a prefix (or an ℓ-prefix) of x and will be denoted by x_ℓ. Similarly, a substring of the form $x[|x| - \ell + 1 : |x|]$ is a suffix (or an ℓ-suffix) of x, and will be denoted by x^ℓ. We say that strings x and y have an *overlap* of length ℓ if the ℓ-suffix of x and the ℓ-prefix of Y are the same sequence; i.e., $x^\ell = y_\ell$. We let $\bar{x}$ denote the *reverse completement* of x.

We assume that there exists an unknown DNA sequence s of length $|s| = G$ which we wish to assemble from a set of N reads $\mathcal{R}$. Throughout the paper, we will make two simplifying assumptions about the set of reads:

(a) All reads in $\mathcal{R}$ have length L.

(b) The reads in $\mathcal{R}$ are error-free.

The first assumption is made to simplify the exposition and all results in this paper can be modified in a straightforward manner to handle the case of reads of variable length. The second assumption is motivated by the existence of overlapping tools (such as DAligner [17] and Minimap [16]), which can efficiently identify sufficiently long overlaps between reads, and can be used as a first step to the algorithm herein described. By focusing on error-free reads we can emphasize our contribution in the context of the computational challenges associated with whole genome assembly.

For ease of exposition, we will assume that s is a circular sequence of length G; i.e., $s[t + G] = s[t]$ for any t. This way we will avoid edge effects and a read $x \in \mathcal{R}$ can correspond to any substring $s[t : t + L - 1]$, for $t = 1, ..., G$. We will use the standard probabilistic model for shotgun sequencing (based on the Lander-Waterman sampling model [10]). This means that each of the N reads is drawn independently and uniformly at random from the set of length-L substrings of s, $\{s[t : t + L - 1] : t = 1, ..., G\}$.

2.1 Read-Overlap graphs

A Read-overlap Graph $\mathcal{G} = (V, E, w, \mathrm{st})$ is a directed weighted graph (V, E), with edge weights $w : E \to \{1, 2, ...\}$ (the overlaps), where each vertex $v \in V$ has an associated string $\mathrm{st}(v)$ from the set of reads; i.e., $\mathcal{R} = \{\mathrm{st}(v) : v \in V\}$. If $(u, v) \in E$, then u and v must overlap by $w(u, v)$ symbols; i.e.,

$$\mathrm{st}(u)^{w(u,v)} = \mathrm{st}(v)_{w(u,v)}.$$

An example of a read-overlap graph is shown in Fig. 5(a). If $(u, v) \in E$, we will use $\mathrm{st}(u) \oplus \mathrm{st}(v)$ to denote the merging of $\mathrm{st}(u)$ and $\mathrm{st}(v)$ with an overlap of length $w(u, v)$; i.e., the concatenation of $\mathrm{st}(u)$ and $\mathrm{st}(v)^{L-w(u,v)}$. Moreover, if we have a path $p = (v_1, ..., v_k)$ in a read-overlap graph, we let

$$\mathrm{st}(p) = \mathrm{st}(v_1) \oplus \mathrm{st}(v_2) \oplus \ldots \oplus \mathrm{st}(v_k)$$

be the string corresponding to path p. The assembly problem corresponds to the problem of finding a path $p = (v_1, v_2, ..., v_N)$ on $\mathcal{G}$ such that $\mathrm{st}(p) = s$. Due to our assumption of a circular genome, we will be instead interested in finding a cycle $c = (v_1, v_2, ..., v_N, v_1)$ on $\mathcal{G}$ such that $\mathrm{st}(c)$ (or a cyclic shift of it) equals s.

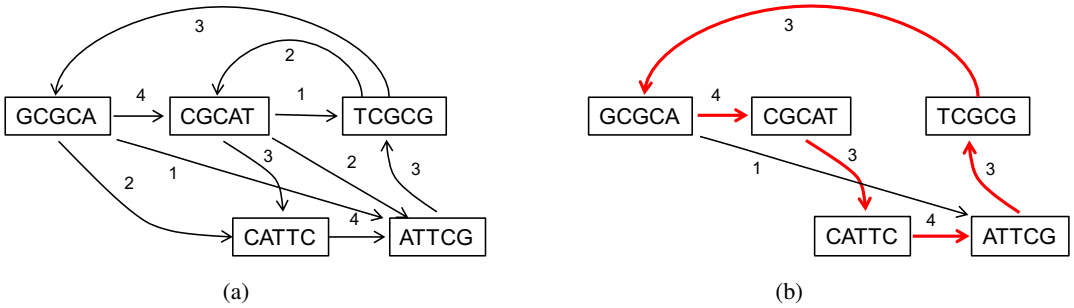


Figure 5: (a) Read-overlap graph from five reads of length 5. The weight of each directed edge (u, v) indicates the size of the matching suffix of u and prefix of v. (b) In the string graph paradigm, transitive edges are removed from the graph, making it easier to identify a Hamiltonian cycle (shown in red). Notice that spurious edges (not part of the Hamiltonian cycle) remain on the graph after the transitive reduction.

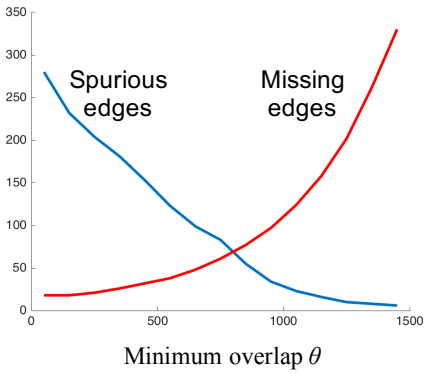
One of the challenges of trying to identify the cycle c corresponding to s on a read-overlap graph is the fact that the graph may contain many spurious edges that are not part of c. In fact, if all overlaps of size at least 1 are included in $\mathcal{G}$, it is easy to see that we have $|E| = O(N^2)$, making even the construction of such a graph impractical, as N can be of the order of 10^5-10^8. However, the definition of $\mathcal{G}$ above does not specify *which* overlaps should in fact be included as edges. Hence, one may choose to construct a sparse version of the graph by not including edges that seem unlikely to correspond to adjacent vertices in the cycle c, such as edges corresponding to "small" overlaps.

2.2 String graphs

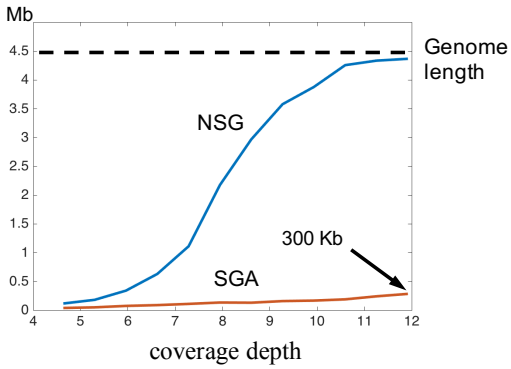
A general technique to sparsify read-overlap graphs was introduced in [18], through the paradigm of the string graph. The main idea of a string graph is to prune the read-overlap graph by removing all *transitive* edges. An edge (u, v) whose corresponding string can be equivalently obtained via the two-hop path (u, z, v) (i.e., $\mathrm{st}(u, v) = \mathrm{st}(u, z, v)$) is called a transitive edge, and can be removed from the graph without affecting the existence of a cycle c corresponding to the true sequence s.

In addition to this transitive reduction (which can be performed in linear time [18]), the original string graph paradigm also utilizes a minimum overlap parameter θ, and edges corresponding to overlaps smaller than θ are not included. As it turns out, θ controls an important tradeoff: a large value of θ gives rise to a sparser string graph, possibly missing many "true" edges, while a small value of θ leads to a denser graph, possibly containing many spurious edges and consequently yielding shorter contigs. This point is illustrated in Figure 6. Any repeat that is longer than θ can create a false positive edge and hence, choosing θ to be small makes the resulting assembly more fragmented (i.e., with a higher number of shorter and perhaps missassembled contigs). However, if we choose θ too large, we go into a coverage-limited regime, and many true edges are missing in the string graph. Moreover, we see from Figure 6(b) that, even if we are provided with the optimal value of θ, we may still be left with a string graph that yields a fairly fragmented assembly.

Such considerations naturally raise some questions. Is the global minimum overlap θ the right approach to control the sparsity of the string graph? Is the spurious/missing edges tradeoff fundamental or is it possible to simultaneously minimize the number of spurious edges and missing edges? As it turns out, the algorithm we will describe in Section 3 – the NOT-SO-GREEDY algorithm – constructs a sparse read-overlap graph which, under some conditions, simultaneously achieves the goals of no spurious edges and no missing edges. When this occurs, the assembly problem becomes the problem of finding a Chinese Postman cycle on the graph, which can be done efficiently, and yields a complete assembly of the genome. Furthermore, even when the conditions are not met and the graph contains spurious and/or missing edges, the resulting contigs are longer and fewer than what is obtained via a traditional string graph approach, as illustrated in Figure 6(b).



(a) Incorrect edges on string graph



(b) N50 value for *E. coli* K12

Figure 6: (a) Number of spurious and missing edges as a function of θ, for 10,000 (error-free) length-3000 reads from *E. coli* K12. (b) N50 as a function of the number of length-3000 reads obtained from string graph (using SGA [19]) with optimal choice of θ, and from NSG algorithm (Section 3). The N50 values were averaged over twenty realizations for each value of the number of reads.

2.3 The GREEDY algorithm

As the name suggests, the NOT-SO-GREEDY algorithm takes inspiration on a simple greedy approach, but tries to be more flexible in its choice of edges to include in the graph. The GREEDY algorithm [3] for assembly can be equivalently thought of as a rather extreme approach to constructing a sparse read-overlap graph. As described in Algorithm 1, in this approach, for each vertex $u \in V$, one only includes the edge (u, v) with the largest overlap $w(u, v)$. If we let $\mathcal{I}(v) = \{u \in V : (u, v) \in E\}$, $\mathcal{O}(v) = \{u \in V : (v, u) \in E\}$, we can define $\Delta_{\text{in}}(v) = |\mathcal{I}(v)|$ and $\Delta_{\text{out}}(v) = |\mathcal{O}(v)|$ as the indegree and outdegree of a vertex v in $\mathcal{G}$, and $\Delta(\mathcal{G}) = \max_{v \in V} \max[\Delta_{\text{in}}(v), \Delta_{\text{out}}(v)]$. The GREEDY algorithm always constructs a read-overlap graph where $\Delta(\mathcal{G}) = 1$, as illustrated in Fig. 7(a). This means that GREEDY can only hope to reconstruct cycles in the read-overlap graph that visit every vertex exactly once.

Algorithm 1 GREEDY sparsification

Input: $\mathcal{G} = (V, E, \text{st}, w)$
$\tilde{E} \leftarrow \emptyset$
$E_{\text{available}} \leftarrow E$
while $E_{\text{available}} \neq \emptyset$ **do**
  % Include available edge (u, v) of maximum weight
  $(u, v) \leftarrow \arg\max_{(u,v) \in E_{\text{available}}} w(u, v)$
  $\tilde{E} \leftarrow \tilde{E} \cup \{(u, v)\}$
  % Other outgoing edges of u and incoming
  % edges of v are no longer available
  $E_{\text{available}} \leftarrow E_{\text{available}} - (\mathcal{O}(u) \cup \mathcal{I}(v))$
Output: $\tilde{\mathcal{G}} = (V, \tilde{E}, \text{st}, w)$

However, as we observe in Fig. 7(b), if a sequence contains a long repeat and a read falls inside this repeat, we may need to allow the cycle to visit such a read twice in order to obtain a complete assembly. In other words, searching beyond (strict) Hamiltonian cycles and allowing generalized Hamiltonian cycles may give us more power in terms of which sequences we can fully assemble, and the simple GREEDY algorithm cannot achieve this.

This drawback of GREEDY can also be understood in terms of the informational feasibility region from [9]. One can verify that GREEDY makes an error with probability at least $1/2$ if $L \leq \ell_{\text{repeat}}$, where ℓ_{repeat} is the length of the longest repeat in s [9]. Because of this, the performance of GREEDY in the (N, L) feasibility region has a vertical asymptote at ℓ_{repeat}, which is in general greater than the true feasibility asymptote ℓ_{crit}, as illustrated in Fig. 8 for the Human

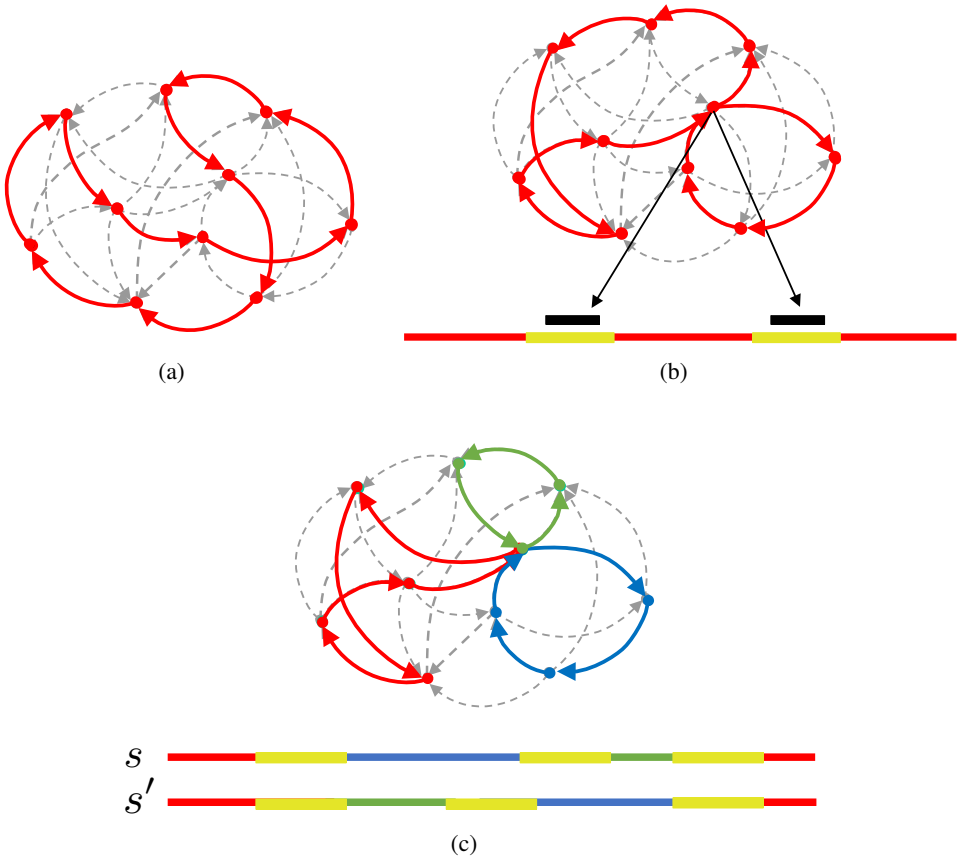


Figure 7: (a) Example of a cycle that visits every node exactly once (i.e., a Hamiltonian cycle). In this case, it is possible that the GREEDY algorithm will correctly select the edges corresponding to the true sequence. (b) Example of a cycle corresponding to a sequence containing a repeat that is longer than L, when a read falls inside this repeat. In this case, GREEDY may fail to recover the correct sequence, as it cannot generate a cycle that visits a node more than once. (c) If the sequence contains a triple repeat longer than L, there is ambiguity in the order in which the segments in between the repeat copies should be traversed. For this example, the same set of edges in the read-overlap graph can give raise to the cycle red-blue-green and the cycle red-green-blue. Hence, no algorithm can distinguish between sequences s and s'.

chromosome 19. We refer to Section 6.4 for more details on the critical read length ℓ_{crit}.

We conclude that in the process of sparsifying the read-overlap graph, if we want to achieve the information lower bound, we cannot be too greedy, and we must allow some nodes to have in and out-degree greater than 1. However, allowing arbitrarily large degrees goes against our goal of sparsifying the graph. Hence, an important question is how many times we must allow the cycle to visit the same vertex in order to achieve the informationally feasible region (blue region in Fig 8). Suppose the generalized Hamiltonian cycle corresponding to the true sequence s in the read-overlap graph is as illustrated in Fig. 7(c) and visits the same node three times, due to the existence of a triple repeat in the sequence. We notice that by reversing the order in which the generalized Hamiltonian cycle traverses the blue and green cycles, we obtain another sequence s' of the same length, which utilizes the exact same set of edges of the read-overlap graph. Therefore, it is impossible to distinguish between these two candidate sequences, and no algorithm can guarantee the correct assembly of the entire sequence with a probability greater than $1/2$. This means that any instance of the assembly problem in which the cycle corresponding to the true sequence s visits a given node three times or more cannot be solved unambiguously, and must be outside of the feasibility region of Fig. 8.

We conclude that for any feasible instance of the assembly problem the generalized Hamiltonian cycle in the read-overlap graph visits a node at most twice. The NOT-SO-GREEDY algorithm, which we describe in Section 3, exploits this fact by building a sparse read-overlap graph where the in and out-degree of every node is at most 2.

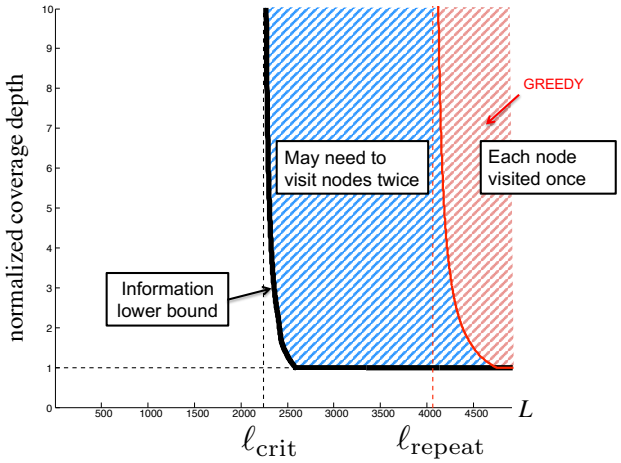


Figure 8: For the Human chromosome 19, GREEDY can only work if $L > \ell_{\text{repeat}} = 4100$, while the information limit only requires $L > \ell_{\text{crit}} = 2200$ (check numbers). In order to achieve the information lower bound, we need an algorithm that can find a generalized Hamiltonian path on the read-overlap graph where some vertices may need to be visited more than once.

3 Methods

The NOT-SO-GREEDY algorithm comprises two stages. In the first one, we construct a read-overlap graph $\mathcal{G}$ where all vertices have in and out-degree at most two. While such a graph is already sparse in the sense that $|E| \leq 2|V|$, it will still contain many spurious edges. Therefore, in order to further reduce it to a graph with no spurious edges, a pruning stage will follow with the goal of eliminating them. The resulting read-overlap graph, $\widetilde{\mathcal{G}}$, will have the property that the true sequence corresponds to a cycle that traverses every edge at least once; i.e., a Chinese Postman cycle. Furthermore, the additional property that the cycle visits every node at most twice allows us to further reduce the problem to an instance of the Eulerian cycle problem.

Formally, we will show that the NOT-SO-GREEDY correctly performs this reduction provided that the sequence is covered by the reads (i.e., each base is read by at least one read in $\mathcal{R}$; see also Section 6.4), and certain bridging conditions are satisfied. As illustrated in Fig. 9, a triple repeat A is said to be all-bridged if each of its copies is bridged by a read

Figure 9: A copy of a triple repeat A is bridged if there is a read that extends beyond the repeat at both ends. A triple repeat is said to be all-bridged if all of its copies is bridged.

(i.e., there is a read that extends at least one base before and one base after the repeat segment). The NOT-SO-GREEDY algorithm, which we describe in the remainder of this section, provides the following reduction.

Theorem 1. *If $\mathcal{R}$ covers s and all triple repeats in s are all-bridged,* NOT-SO-GREEDY *produces a read-overlap graph $\widetilde{\mathcal{G}}$ that contains a cycle c corresponding to s that traverses every edge.*

Theorem 1 implies that, under the assumption that triple repeats in s are all-bridged, AP can be reduced to an instance of the Chinese Postman Problem (CPP) on the read-overlap graph $\widetilde{\mathcal{G}}$. This corresponds to finding the smallest number of increments of edge multiplicities required to make the graph Eulerian.

As shown in [9], a necessary condition for AP to be feasible is that triple repeats are bridged (but not all-bridged). This means that the read length ℓ_{crit} required for AP to be feasible [9] is always sufficient for triple repeats to be all-bridged. Furthermore, as discussed in [9] and Section 6.4, under the standard model for shotgun sequencing, for most genomes considered the region where triple repeats are all-bridged is larger than the feasibility region. This means that for most feasible (N, L) pairs, with high probability NOT-SO-GREEDY reduces AP to CPP. In addition, in Section 6.3 we present an algorithm to further reduce the problem to an Eulerian Cycle Problem (ECP).

3.1 Building a Read-Overlap Graph with $\Delta(\mathcal{G}) = 2$

The main idea of the NOT-SO-GREEDY construction of the read-overlap graph with $\Delta(\mathcal{G}) = 2$ is to include, for a given node u, the edge (u, v) that would be included by the GREEDY algorithm and a second carefully chosen edge.

To describe this more precisely, consider the illustration in Fig. 10, where the nodes v with which u has an overlap are ordered in decreasing order of $w(u, v)$, breaking ties arbitrarily. If we assume that $w(u, v_1) \leq w(u, v_2) \leq ...$, then

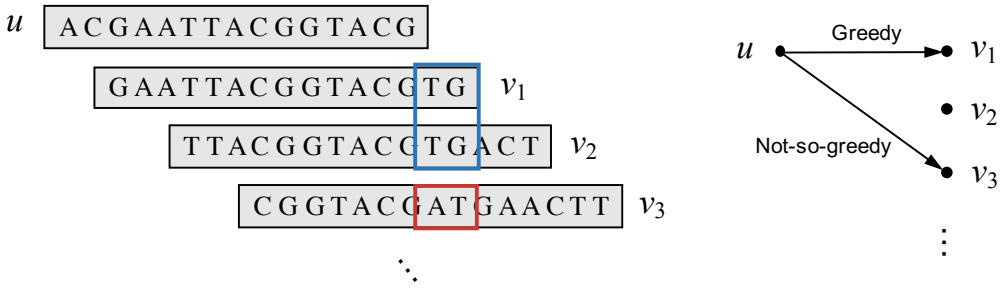


Figure 10: Choosing two out-edges of a node u.

the "greedy edge" (u, v_1) is the first to be included in the graph. To choose the second out-edge of u, we keep going down on this list until we find a node that disagrees with the extension of v_1 beyond u; i.e., we pick the first node v_i for which

$$\mathrm{st}(v_1)^{L-w(u,v_1)} \neq \mathrm{st}(v_i)[w(u, v_i) + 1 : L + w(u, v_i) - w(u, v_1)].$$

Note that this is equivalent to picking the first node v_i for which

$$w(v_1, v_i) < L - w(u, v_1) + w(u, v_i).$$

In the example in Fig. 10, $\mathrm{st}(v_3)$ disagrees with $\mathrm{st}(v_1)$ on its extension beyond u, so we choose (u, v_3) as the second edge (or the not-so-greedy edge). We point out that this choice of the second edge can be understood in the context of the transitive reduction used to construct string graphs (Section 2.2). It is easy to see that all edges that are skipped when we look for the not-so-greedy edge would be transitive edges, and would have been eliminated by the transitive reduction algorithm from [18]. However, by only looking for the greedy and not-so-greedy edges, we avoid the need to compute the entire string graph as prescribed by the algorithm in [18].

Algorithm 2 NOT-SO-GREEDY construction of $\mathcal{G}_{\mathrm{out}}$

Input: V, st, w
$E \leftarrow \emptyset$
for $u \in V$ **do**
  $V_{\mathrm{sorted}} \leftarrow \{v_1, ..., v_N\}$, where $w(u, v_1) \geq ... \geq w(u, v_N)$
  $E \leftarrow E \cup (u, v_1)$ % Choose (u, v_1) as greedy edge
  $i \leftarrow 2$
  % Go down V_{sorted} to find not-so-greedy edge
  while $w(v_1, v_i) = L - w(u, v_1) + w(u, v_i)$ **do**
    $i \leftarrow i + 1$
  $E \leftarrow E \cup (u, v_i)$
Output: $\mathcal{G}_{\mathrm{out}} = (V, E, \mathrm{st}, w)$

By repeating this procedure for every node, we construct a graph with the property that $\Delta_{\mathrm{out}}(u) \leq 2$ for every node u, which we refer to as $\mathcal{G}_{\mathrm{out}}$. This procedure is summarized in Algorithm 2. The notion of all-bridging allows us to obtain the following guarantee for the resulting graph $\mathcal{G}$:

Lemma 1. *If $\mathcal{R}$ covers s and all triple repeats in s are all-bridged, Algorithm 2 produces a read-overlap graph $\mathcal{G}_{\text{out}}$ containing a generalized Hamiltonian cycle c_s such that $\text{st}(c_s) = s$.*

Proof. Let $\mathcal{G}_{\text{out}} = (V, E, \text{st}, w)$ be the read-overlap graph produced by Algorithm 2. We start by considering the set of all starting positions t such that $s[t : t + L - 1] = \text{st}(u)$ for some read $\text{st}(u) \in \mathcal{R}$, and assume without loss of generality that $t_1 < t_2 < ... < t_M$. Notice that in general we may have $M > N$, as a read $\text{st}(u)$ could have come from multiple distinct positions in s, due to repeats. Let $u_i \in V$ be such that $\text{st}(u_i) = s[t_i : t_i + L - 1]$, for $1 \leq i \leq M$, and set $c_s = (u_1, u_2, ..., u_M)$. We will prove that, for an arbitrary $i \in \{1, ..., M\}$, $(u_i, u_{i+1}) \in E$ and $w(u_i, u_{i+1}) = L - (t_{i+1} - t_i)$. It will then follow that $\mathcal{G}_{\text{out}}$ contains c_s and $\text{st}(c_s) = s$.

Suppose by contradiction that (u_i, u_{i+1}) was not added to E by Algorithm 2. The assumption that $\mathcal{R}$ covers s guarantees that $t_{i+1} - t_i < L$. Therefore, if Algorithm 2 did not add (u_i, u_{i+1}) to E, there are two possibilities:

(i) Algorithm 2 added two outgoing edges (u_i, v_1) and (u_i, v_2) to E with $w(u_i, v_1) \geq w(u_i, v_2) \geq w(u_i, u_{i+1})$, and $v_1, v_2 \neq u_{i+1}$.

(ii) When (u_i, u_{i+1}) was considered by the algorithm, the condition in line 8 was satisfied; i.e., $w(v_1, u_{i+1}) = L - w(u_i, v_1) + w(u_i, u_{i+1})$.

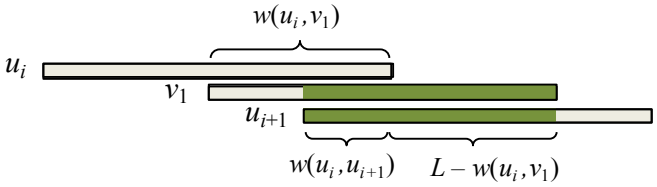


Figure 11: If we have $w(v_1, u_{i+1}) = L - w(u_i, v_1) + w(u_i, u_{i+1})$, the read $\text{st}(v_1)$ must be a substring of $\text{st}(u_i, u_{i+1})$.

Suppose (ii) was satisfied and we had some $v_1 \neq u_{i+1}$ satisfying $w(v_1, u_{i+1}) = L - w(u_i, v_1) + w(u_i, u_{i+1})$. By referring to Figure 11, we see that this implies that $\text{st}(v_1)$ must be a substring of $\text{st}(u_i, u_{i+1})$. But this means that for some $\tau \in \{t_i + 1, ..., t_{i+1}\}$, $\text{st}(v_1) = s[\tau : \tau + L - 1]$, contradicting the way in which we chose $t_1, ..., t_M$.

Suppose (i) was satisfied, and we have a picture similar to Fig. 12. Notice that the part of v_2 that extends beyond its

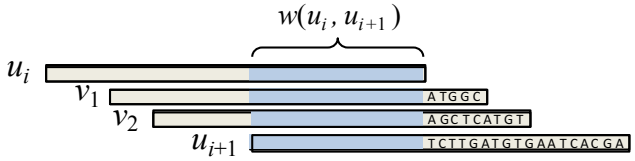


Figure 12: If Algorithm 2 adds two edges (u_i, v_1) and (u_i, v_2) to E before $(u_i, u_i + 1)$, we must have $w(u_i, v_1) \geq w(u_i, v_2) \geq w(u_i, u_{i+1})$. By construction, the extensions of v_1, v_2 and u_{i+1} beyond u_i must disagree with each other, implying that the blue segment is a triple repeat in s. From the way we chose u_i and u_{i+1} this triple repeat is not all-bridged.

overlap with u_i must not agree with the extension of v_1 beyond its overlap with u_i. Notice that the part of v that extends beyond its overlap with u also cannot agree completely with that of any v_1 or v_2, or else that would contradict our choice of t_i and t_{i+1}. Therefore, we conclude that the segment $Y = \text{st}(u_i)^{w(u_i, u_{i+1})}$ (shown in blue in Fig. 12) appears as a distinct segment three times in s; i.e., it is an triple repeat. Moreover, from the way we chose t_i and t_{i+1} there is no read bridging the copy of Y at $s[t_{i+1} : t_{i+1} + |Y| - 1]$. But this is a contradiction to the assumption that triple repeats are all-bridged. We conclude that (u_i, u_{i+1}) must have been added to E. □

Lemma 1 implies that $\mathcal{G}_{\text{out}}$ contains a cycle corresponding to the true sequence, and it is clear by construction that $\Delta_{\text{out}}(u) \leq 2$ for every node u. However, nodes in $\mathcal{G}_{\text{out}}$ may have in-degree larger than 2. In order to fix that, we let E_{out} be the set of edges produced by Algorithm 2, and we consider repeating Algorithm 2 modified in a straightforward way

to produce a second read-overlap graph $\mathcal{G}_{\text{in}} = (V, E_{\text{in}}, \text{st}, w)$ with $\Delta_{\text{in}}(u) \leq 2$ for every $u \in V$ and for which Lemma 1 also holds by symmetry. Finally, we construct the read-overlap graph $\mathcal{G} = (V, E, \text{st}, w)$, by setting $E = E_{\text{out}} \cap E_{\text{in}}$, and we obtain our desired property that $\Delta(\mathcal{G}) \leq 2$.

We point out that the construction procedure in Algorithm 2 is mainly for description purposes. In Section 6.1, we introduce an approach to find overlaps efficiently based on Rabin-Karp rolling hashes [20], which allows us to efficiently identify overlaps between reads in decreasing order of overlap size. Therefore, we do not need to perform a sorting of overlaps for every read as in line 4. This approach, which we refer to as decremental hashing (in analogy to the incremental hashing of [21]), yields a $O(NL)$ construction algorithm (for the case of error-free reads), summarized in Algorithm 4.

3.2 Pruning the Read-Overlap Graph

Next, we describe the second part of the NOT-SO-GREEDY algorithm, which attempts to prune $\mathcal{G}$ in order to reduce the number of spurious edges. The fact that this graph has $\Delta(\mathcal{G}) \leq 2$ will allow us to utilize the natural partition of the edge set into *zig-zag* components in order to identify edges that should be removed.

The pruning algorithm starts by preprocessing the read-overlap graph $\mathcal{G}$ for the edge pruning operations by computing an *effective overlap* function $\tilde{w} : E \to \{1, 2, ...\}$. The idea is to check when the string $\text{st}(u)$ can be extended to the right or to the left in an unambiguous way. For example, suppose $(u, v_1) \in E$ and $(u, v_2) \in E$, and the corresponding reads are as shown in Figure 13. Then, read $\text{st}(u)$ must be followed by A, G, G in s, and we can imagine that $\text{st}(u)$ is virtually extended to the right by three positions. The effective overlaps can then be defined as $\tilde{w}(u, v_1) = w(u, v_1) + 3$ and $\tilde{w}(u, v_2) + w(u, v_2) + 3$. Reads can be virtually extended to the left in the same way. We will set $\tilde{w}(u, v) = \infty$

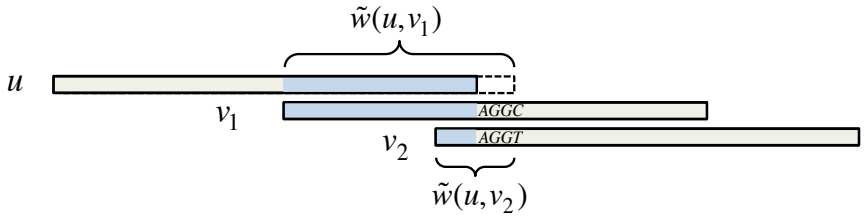


Figure 13: In this example $\text{st}(u)$ can be virtually extended to the right in an unambigous manner by three symbols and we have the effective overlap values $\tilde{w}(u, v_i) = w(u, v_i) + 3$ for $i = 1, 2$.

whenever $\Delta_{\text{out}}(u) = 1$ or $\Delta_{\text{in}}(v) = 1$. This will prevent the algorithm from ever deleting edge (u, v). A more detailed description of the computation of the effective overlaps is presented in Algorithm 5 in Section 6.2. Notice that we do not actually modify the function $\text{st}(\cdot)$, and the read extensions are only a way to visualize the computation of the effective overlaps $\tilde{w}(\cdot, \cdot)$.

Once all effective overlaps are computed, we move to the pruning phase, described in Algorithm 3. In this part, the actual string associated with each read is no longer needed, and the algorithm works solely on the graph (V, E) and the effective overlaps $\tilde{w}(\cdot, \cdot)$. The algorithm will keep at all times a graph (V, E) with the property that $1 \leq \Delta_{\text{in}}(v), \Delta_{\text{out}}(v) \leq 2$ for every vertex $v \in V$. At each iteration, our goal is to reduce the size of the subset of edges

$$U = \{(u, v) : \Delta_{\text{out}}(u) = \Delta_{\text{in}}(v) = 2\},$$

and the algorithm stops when $U = \emptyset$. The set U can be thought of as the set of edges in E whose presence in the cycle c_s corresponding to the true sequence s is unknown. More precisely, any edge $(u, v) \notin U$ must have $\Delta_{\text{out}}(u) = 1$ or $\Delta_{\text{in}}(v) = 1$ and if the read-overlap graph maintained by the algorithm contains the cycle c_s that visits every node, it must traverse (u, v). Notice also that any edge $(u, v) \notin U$ will have $\tilde{w}(u, v) = \infty$.

For an edge $(u_2, v_1) \in U$, we will let $\text{hd}(u_2, v_1)$ be the (unique) edge sharing the same head vertex as (u_2, v_1), and $\text{tl}(u_2, v_1)$ be the (unique) edge sharing the same tail vertex as (u_2, v_1). Algorithm 3 starts by identifying the edges in U (i.e., those with $\tilde{w}(u, v) \neq \infty$) and then sorting them in increasing order of effective overlap value $\tilde{w}(\cdot, \cdot)$. If during the sorting two edges $(u_1, v_1), (u_2, v_2) \in U$ are found to have $\tilde{w}(u_1, v_1) = \tilde{w}(u_2, v_2)$, and we have $\tilde{w}(u_1, v_1) = \tilde{w}(\text{hd}(u_1, v_1)) = \tilde{w}(\text{tl}(u_1, v_1))$, (u_2, v_2) is placed before (u_1, v_1) in the sorting. Ties are broken arbitrarily otherwise.

The algorithm operates by taking the first (u_2, v_1) from the sorted U and considering the "zig-zag" structure formed by edges $(u_1, v_1) = \mathrm{hd}(u_2, v_1)$, $(u_2, v_2) = \mathrm{tl}(u_2, v_1)$, which must exist by the definition of U and be unique by the bounded-degree property. At this point the algorithm considers two cases to decide which transformation to make to the graph. If $(u_1, v_2) \in E$, we have the "hourglass" structure. If in addition all four edges $(u_1, v_1), (u_1, v_2), (u_2, v_1), (u_2, v_2)$ have the same effective overlap value $\tilde{w}(\cdot, \cdot)$, the algorithm will add a virtual read that will create an X-node in the read-overlap graph, as illustrated in Figure 14. Otherwise, the algorithm will simply delete (u_2, v_1). The same pruning

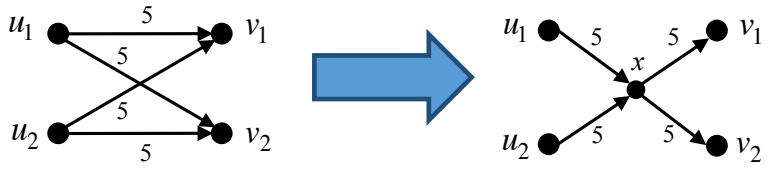


Figure 14: If an hourglass structure $(u_1, v_1), (u_1, v_2), (u_2, v_1), (u_2, v_2)$ with $\tilde{w}(u_1, v_1) = \tilde{w}(u_1, v_2) = \tilde{w}(u_2, v_1) = \tilde{w}(u_2, v_2)$ is found, we create a vertex x, whose string $\mathrm{st}(x)$ corresponds to the effective overlap between u_i and v_j (they are all the same), delete edges $(u_1, v_1), (u_1, v_2), (u_2, v_1), (u_2, v_2)$ and add edges $(u_1, x), (u_2, x), (x, v_1), (x, v_2)$ to E.

procedure is applied iteratively, until $U = \emptyset$.

Algorithm 3 NOT-SO-GREEDY pruning phase

1: Input: $\mathcal{G} = (V, E, \mathrm{st}, w)$ with $\Delta(\mathcal{G}) = 2$
2: Use Algorithm 5 to compute effective overlaps $\tilde{w}(\cdot, \cdot)$
3: $U \leftarrow \{(u, v) : \Delta_{\mathrm{out}}(u) = \Delta_{\mathrm{in}}(v) = 2\}$
4: Sort U according to $\tilde{w}(u, v)$
5: **while** $U \neq \emptyset$ **do**
6: $(u_2, v_1) \leftarrow$ first edge in U
7: $(u_1, v_1) \leftarrow \mathrm{hd}(u_2, v_1)$
8: $(u_2, v_2) \leftarrow \mathrm{tl}(u_2, v_1)$
9: **if** $(u_1, v_2) \in U$ and $\tilde{w}(u_1, v_1) = \tilde{w}(u_2, v_1) = \tilde{w}(v_1, v_2) = \tilde{w}(u_1, v_2)$ **then**
10: Create new node x
11: Perform transformation in Fig. 14
12: **else**
13: Delete edge (u_2, v_1)
14: Update set U and weights $\tilde{w}(\cdot, \cdot)$
15: Output: $\widetilde{\mathcal{G}} = (V, E, \mathrm{st}, w)$

This iterative pruning procedure is summarized in Algorithm 3. Notice that computing the effective overlaps (Algorithm 5) for each edge $(u, v) \in E$ has a running time of $O(|E|) = O(N)$ since $|E| \leq 2N$. Algorithm 3 then sorts the edges in U, which has a worst-case time complexity of $O(N \log N)$, and performs $O(|E|) = O(N)$ pruning iterations, each of which can be done in constant time. Finally, we notice that since the reads in $\mathcal{R}$ are assumed to be all distinct from each other (which can be achieved by a preprocessing step), we must have $L \geq \log_4 N$, and we conclude that the running time for the NOT-SO-GREEDY algorithm is $O(NL)$.

Furthermore, as stated in Theorem 1, provided that $\mathcal{R}$ covers the sequence s and triple repeats are all-bridged, the resulting graph $\widetilde{\mathcal{G}}$ contains a cycle c_s corresponding to the true sequence s that traverses every edge. We are now in a position to prove this result.

Proof of Theorem 1. From Lemma 1, we know that the graph $\mathcal{G} = (V, E, \mathrm{st}, w)$ obtained via the NOT-SO-GREEDY construction contains a generalized Hamiltonian cycle c_s corresponding to the true sequence s. Therefore, the computed effective overlaps $\tilde{w}(\cdot, \cdot)$ (Algorithm 5) are all consistent with s. Moreover, the hourglass to X transformation (Fig. 14) that is performed when the condition in line 9 is satisfied cannot affect the existence of the cycle c_s (although it technically modifies the cycle to traverse the newly created node). When line 9 is not satisfied, we instead delete the middle edge of a Z structure (Fig. 15). We will prove that this pruning operation cannot remove any edge from c_s. Then, since

any edge $(u, v) \in E - U$ must have either $\Delta_{\text{out}}(u) = 1$ or $\Delta_{\text{in}}(v) = 1$ and thus be part of c_s, we will conclude that, when $U = \emptyset$, all edges in E will belong to c_s.

Suppose edge (u_2, v_1) is the edge from U with the smallest overlap value, and let $(u_1, v_1) = \text{hd}(u_2, v_1)$ and $(u_2, v_2) = \text{tl}(u_2, v_1)$. We consider two cases. First, suppose $\max[\tilde{w}(u_1, v_1), \tilde{w}(u_2, v_2)] > w(u_2, v_1)$, as illustrated in Fig. 15. Suppose by contradiction that the string $\text{st}(u_2) \oplus \text{st}(v_1)$ is part of s (which must be the case if (u, v) is part

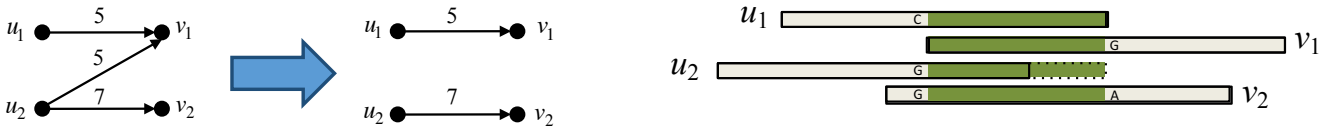


Figure 15: (a) If a Z structure $(u_1, v_1), (u_2, v_1), (u_2, v_2)$ with $\max[\tilde{w}(u_1, v_1), \tilde{w}(u_2, v_2)] > w(u_2, v_1)$ is found, edge (u_2, v_1) is deleted. (b) The virtual extensions used to compute the effective overlaps $\tilde{w}(\cdot, \cdot)$ guarantee that if a Z structure as in (a) is found, then u_1, u_2, v_1 and v_2 contain a common substring of length $\tilde{w}(u_2, v_1)$ preceded by two distinct bases in u_1 and u_2, and followed by two distinct bases in v_1 and v_2. Moreover, the fact that $\tilde{w}(u_2, v_2) > \tilde{w}(u_2, v_1)$ guarantees that if $\text{st}(u_2) \oplus \text{st}(v_1)$ is a substring of s, then the green segments in reads u_1 and v_2 cannot coincide with the green segment on $\text{st}(u_2) \oplus \text{st}(v_1)$, which implies the existence of an unbridged triple repeat.

of c_s). Let Y be the string corresponding to the effective overlap between u_2 and v_1 (green segment in Fig. 15(b)). Due to the definition of the effective overlaps, the symbol preceding Y in u_1 is different from the symbol preceding Y in u_2 and v_2. Similarly, the symbol following Y in v_1 and v_2 must be distinct. This guarantees that Y must appear at least three times in s: one in $\text{st}(u_1)$, one in $\text{st}(u_2) \oplus \text{st}(v_1)$ and one in $\text{st}(v_2)$, and no two of these can coincide. Therefore, Y is a triple repeat in s. Moreover, from the construction of c_s (see proof of Lemma 1), there cannot be any read between entirely contained in $\text{st}(u_2) \oplus \text{st}(v_1)$, implying that the copy of Y in the segment $\text{st}(u_2) \oplus \text{st}(v_1)$ is not bridged. But this contradicts the assumption that triple repeats are all-bridged.

Now suppose that instead we have $\tilde{w}(u_1, v_1) = \tilde{w}(u_2, v_2) = \tilde{w}(u_2, v_1)$. Based on our sorting rule for U, the only way (u_2, v_1) would be considered for deletion before (u_1, v_1) and (u_2, v_2) is if the graph $\mathcal{G}$ (the input to Algorithm 3) contained the zig-zag structure illustrated in Fig. 16 and all edges in it had the same effective overlap $\tilde{w}(u_2, v_1)$. Since

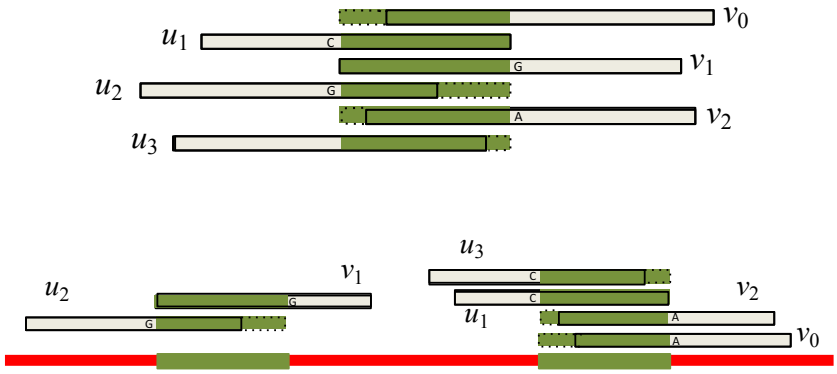


Figure 16: If a Z structure $(u_1, v_1), (u_2, v_1), (u_2, v_2)$ with $\tilde{w}(u_1, v_1) = \tilde{w}(u_2, v_2) = \tilde{w}(u_2, v_1)$ is found, based on our sorting rule for U, we must have nodes v_0 and u_3 with $\tilde{w}(u_1, v_0) = \tilde{w}(u_3, v_2) = \tilde{w}(u_2, v_1)$. The virtual extensions used to compute the effective overlaps $\tilde{w}(\cdot, \cdot)$ guarantee that if a Z structure as in (a) is found, then u_1, u_2, v_1 and v_2 contain a common substring of length $\tilde{w}(u_2, v_1)$ preceded by two distinct bases in u_1 and u_2, and followed by two distinct bases in v_1 and v_2. Moreover, the fact that $\tilde{w}(u_2, v_2) > \tilde{w}(u_2, v_1)$ guarantees that if $\text{st}(u_2) \oplus \text{st}(v_1)$ is a substring of s, then the green segments in reads u_1 and v_2 cannot coincide with the green segment on $\text{st}(u_2) \oplus \text{st}(v_1)$, which implies the existence of an unbridged triple repeat.

$\tilde{w}(u_3, v_2) = \tilde{w}(u_2, v_2)$, reads u_1, u_2 and u_3 (after the virtual extension) contain an identical substring Y of length $\tilde{w}(u_2, v_2)$ (shown in green in Fig. 16). Suppose by contradiction that $(u_2, v_1) \in c_s$. Since triple repeats are assumed to be all-bridged, Y cannot be a triple repeat, or it would be unbridged at (u_2, v_1). Hence, reads u_1, u_3, v_0 and v_2 must all be covering the same copy of Y, as illustrated in Fig. 16. Since no two reads are identical, either $\text{st}(u_1)$ or $\text{st}(u_3)$

starts before the other on s. If $\mathrm{st}(u_3)$ starts before (as illustrated in Fig. 16), $\mathrm{st}(u_1)$ is a substring of $\mathrm{st}(u_3) \oplus \mathrm{st}(v_2)$, and Algorithm 2 would not have added edge (u_3, v_2) to the graph. Otherwise, if $\mathrm{st}(u_1)$ starts before $\mathrm{st}(u_3)$, $\mathrm{st}(u_3)$ is a substring of $\mathrm{st}(u_1) \oplus \mathrm{st}(v_0)$, and Algorithm 2 would not have added edge (u_1, v_0) to the graph.

In both cases, we reach a contradiction, and we conclude that any edge (u, v) deleted by Algorithm 3 in line 13 is not part of c_s. Therefore, when $U = \emptyset$ and the algorithm stops, c_s must traverse all remaining edges in E. □

Theorem 1 implies that the in the graph $\widetilde{\mathcal{G}}$ produced by the NOT-SO-GREEDY algorithm the true sequence s corresponds to a cycle that traverses every edge. Hence, rather than a generalized Hamiltonian cycle, we are now looking for a Chinese Postman cycle. The problem of finding the shortest such cycle, the Chinese Postman Problem, can be solved in polynomial time, but with a running time of $O(|V|^2|E|) = O(N^3)$ [22].

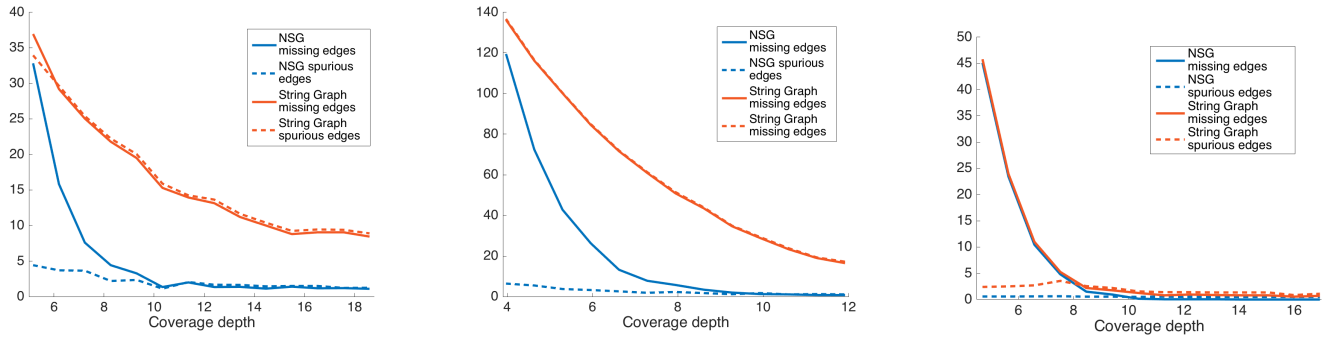
However, when triple repeats are all-bridged, the cycle c_s has the additional property that it cannot visit any node (and, consequently, any edge) more than twice. Otherwise, we would have a triple repeat of length at least L in s, which cannot be bridged by the reads. As it turns out, if we know that the cycle c_s visits every node in $\widetilde{\mathcal{G}}$ at most twice, one can find in linear time which edges should be traversed twice. This reduces the problem further to an instance of the Eulerian Cycle Problem, which can be solved in linear time. We defer the formal description of this reduction to Section 6.3.

4 Results

While the NOT-SO-GREEDY algorithm presented in Section 3 is motivated by the goal of perfect assembly, it can be seen as a general approach for the construction of a sparse read-overlap graph. Once the graph is constructed, one can use algorithms for condensing the nodes of the read-overlap graph (see [9] for instance), and extract contigs from the resulting graph.

Notice that the result in Theorem 1 states that if triple repeats are all-bridged, the resulting graph will contain a simple contig, but this assumption may not be satisfied in general. To evaluate NOT-SO-GREEDY in a more general setting, we implemented the algorithm (available at `github.com/samuelhkim/nsg`) and considered its performance on several different genomes when compared to a string-graph assembler (SGA [19] and the Incremental Hashing assembler [21]).

The results in this section were obtained by sampling error-free reads of a fixed length uniformly at random from real genomes (whose references were available from [23]), and then running different assembly algorithms on these simulated datasets. Focusing on simulated reads from real genomes allows us to study the performance of NOT-SO-GREEDY across many independent experiments and a wide variety of coverage depths, and emphasize the conceptual advantages of a sparse read-overlap graph. Combining the NOT-SO-GREEDY algorithm with an overlapper (such as DAligner [17] and Minimap [16]) so that it can be applied to long-read datasets is the goal of our current research.



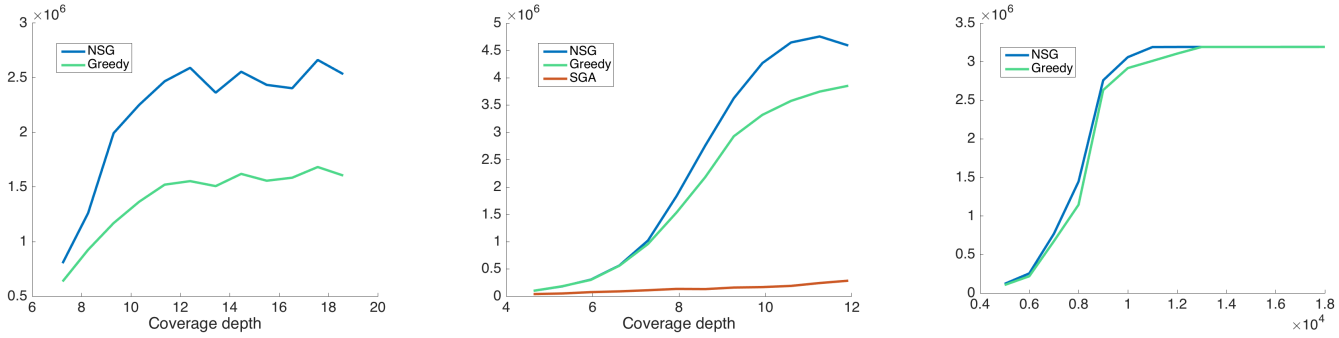
(a) *S. aureus* (b) *E. coli* K12 (c) *R. sphaeroides*

Figure 17: Spurious edges and missing edges in the read-overlap graph produced by NOT-SO-GREEDY and a string graph assembler [21] as a function of the coverage depth $c = NL/G$ for reads of length $L = 3000$. The number of spurious edges and missing edges were averaged over twenty realizations of the read set.

In Fig. 17, we analyze the read-overlap graph produced by the NOT-SO-GREEDY algorithm and by a string graph assembly algorithm [21]. The read-overlap graph is evaluated in terms of the number of missing edges and spurious

edges. In order to compute these values, we utilized the reference genome to produce the "perfect" read-overlap graph (where only reads that are adjacent in the genome have an edge between them) and compared it to the read-overlap produced by the different algorithms. As mentioned in Section 2.2, the minimum overlap θ controls a tradeoff between the number of missing and spurious edges in the string graph, so we chose θ for the string graph assembly algorithm so that the number of missing edges and spurious edges are the same. We notice that the graph produced by NOT-SO-GREEDY has fewer spurious edges and missing edges across all coverage depths and genomes shown. In fact, for the higher values of coverage depth, as predicted by Theorem 1, the graph produced by NOT-SO-GREEDY often has zero missing edges and spurious edges (notice that the values shown in Figure 17 are averaged over twenty realizations of the read sets). We also notice that for the *R. sphaeroides* genome, which is not very repetitive, the performance of both algorithms is very similar.

It is important to point out that the number of missing and spurious edges for both approaches are small in comparison to the total number of edges. For instance, at coverage depth $c = 10$ for *S. aureus*, the "perfect" read-overlap graph contains roughly $N = cG/L \approx 10,000$ reads. This means that the false discovery rate and misdetection rate for the string graph approach are 0.015%. However, even such a small number of incorrect edges on the graph can significantly reduce the quality of the assembly. This is demonstrated in Fig. 18, where we show the N50 values obtained from the read-overlap graphs produced by different algorithms.



(a) *S. aureus* (b) *E. coli* K12 (c) *R. sphaeroides*

Figure 18: N50 as a function of the coverage depth for different algorithms. The N50 values were averaged over twenty realizations for each value of the number of reads.

We see that the (average) N50 values achieved by NOT-SO-GREEDY converge to the genome length as the coverage depth increases much quicker than the N50 values achieved by a string graph approch. We point out that for the N50 experiments, we evaluated the performance of the string graph using SGA [19] as opposed to the assembler from [21] because computing N50 from the SGA output is easier. In addition, the minimum overlap threshold θ for SGA was chosen to maximize N50 (as opposed to equating the number of missing and spurious edges, as in the experiments illustrated in Figure 17).

5 Discussion

In this work we introduced a new algorithm for the construction and subsequent pruning of read-overlap graphs. Unlike most other approaches, the NOT-SO-GREEDY algorithm takes its inspiration from information-theoretic considerations about the AP problem. More precisely, the algorithm is tailored to succeed in the instances of the AP that are feasible; i.e., in the instances where the set of reads $\mathcal{R}$ contains enough information to allow unambiguous reconstruction of the complete sequence. Furthermore, rather than focusing on devising an algorithm to solve a specific optimization-based formulation of the AP (such as the shortest GHC, or the CPP), we design an algorithm which provably reconstructs the *true* underlying sequence, as long as certain bridging conditions are satisfied.

In this context, a natural question is whether this approach is also solving some combinatorial optimization problem. However, as mentioned in [8], parsimony-based formulations tend to encourage an over-collapsing of the repeats, and the optimal solution can be different from the true underlying sequence. One way to prevent this undesirable property of the optimal solution, is to consider a genie-aided formulation where the target genome length G is given. While G is

not available in practice, this combinatorial problem excludes the possibility of shrinking the sequence by compressing repeats, and provides a better setting to analyze the computational complexity (or a lower bound to it) of reconstructing the true sequence.

It is not difficult to see that, like most formulations of the AP, the problem finding a GHC of a desired length G is in general NP-hard (as one could use it to solve the Hamiltonian cycle problem). In this context, the results of the present paper can be understood as characterizing a set of instances where this problem can be solved in linear time. As it turns out, this set of instances essentially corresponds to the instances of the assembly problem where the set of reads contain enough information to prevent the ambiguities described in Fig. 2.

In addition to the implications on our understanding of the computational hardness of the AP, the NOT-SO-GREEDY algorithm can be understood as a general technique to sparsify read-overlap graphs. As shown in Section 4, even when the bridging conditions assumed in Theorem 1 are not satisfied, most of the edges that are removed by the NOT-SO-GREEDY algorithm are indeed spurious edges. This leads to assemblies that are significantly more contiguous (and achieve higher N50 scores) than those obtained via a string graph approach with no additional pruning.

References

[1] Paul Medvedev, Konstantinos Georgiou, Gene Myers, and Michael Brudno. Computability of models for sequence assembly. In *Algorithms in Bioinformatics*, pages 289–301. Springer, 2007.

[2] Niranjan Nagarajan and Mihai Pop. Parametric complexity of sequence assembly: theory and applications to next generation sequencing. *Journal of computational biology*, 16(7):897–908, 2009.

[3] J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theoret. Comput. Science*, 57:131–145, 1988.

[4] Avrim Blum, Tao Jiang, Ming Li, John Tromp, and Mihalis Yannakakis. Linear approximation of shortest superstrings. *Journal of the ACM (JACM)*, 41(4):630–647, 1994.

[5] Ming Li. Towards a dna sequencing theory (learning a string). In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, pages 125–134. IEEE, 1990.

[6] 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.

[7] P. Pevzner. DNA physical mapping and alternating Eulerian cycles in colored graphs. *Algorithmica*, 13:77–105, 1995.

[8] Paul Medvedev and Michael Brudno. Maximum likelihood genome assembly. *Journal of computational Biology*, 16(8):1101–1116, 2009.

[9] G. Bresler, M. Bresler, and D. Tse. Optimal assembly for high throughput shotgun sequencing. *BMC Bioinformatics*, 2013.

[10] Eric S Lander and Michael S Waterman. Genomic mapping by fingerprinting random clones: a mathematical analysis. *Genomics*, 2(3):231–239, 1988.

[11] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. Idba–a practical iterative de bruijn graph de novo assembler. In *Research in Computational Molecular Biology*, pages 426–440. Springer, 2010.

[12] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

[13] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012.

[14] Chen-Shan Chin, David H Alexander, Patrick Marks, Aaron A Klammer, James Drake, Cheryl Heiner, Alicia Clum, Alex Copeland, John Huddleston, Evan E Eichler, et al. Nonhybrid, finished microbial genome assemblies from long-read smrt sequencing data. *Nature methods*, 10(6):563–569, 2013.

[15] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nature biotechnology*, 2015.

[16] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *arXiv preprint arXiv:1512.01801*, 2015.

[17] Gene Myers. Efficient local alignment discovery amongst noisy long reads. In *Algorithms in Bioinformatics*, pages 52–67. Springer, 2014.

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

[19] J. T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 26:367–373, 2010.

[20] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31:249–260, 1987.

[21] I. Ben-Bassat and B. Chor. String graph construction using incremental hashing. *Bioinformatics*, 30(24):3515–3523, 2010.

[22] Jack Edmonds and Ellis L Johnson. Matching, euler tours and the chinese postman. *Mathematical programming*, 5(1):88–124, 1973.

[23] National center for biotechnology information.

6 Supplementary Material

6.1 Decremental hashing of strings

In order to efficiently find overlaps between reads and obtain an $O(NL)$ algorithm, we will utilize a version of the Rabin-Karp rolling hash [20]. Given a length-ℓ string $x \in \Sigma^\ell$, we will define the hashing operation as

$$h(x) = \sum_{i=1}^{L} \phi(x[i])\alpha^{L-i} \pmod p, \tag{1}$$

where ϕ assigns a numeric value to symbols in Σ, α is a fixed integer, and p is a large prime. For the results presented in this paper, we use $\phi(a) = 1$, $\phi(g) = 2$, $\phi(c) = 3$, and $\phi(g) = 4$, $\alpha = 13$ and $p = 1000000007$. **verify these numbers**

The advantage of using the hash function in (1) is that, given the ℓ-prefix (or the ℓ-suffix) of a read r, the $(\ell-1)$-prefix (or the $(\ell-1)$-suffix) can be easily computed. More precisely, from (1) it follows that

$$h(r[1:\ell-1]) = \alpha^{-1}\left[h(r[1:\ell]) - \phi(r[\ell])\right],$$

where α^{-1} refers to the inverse of α in $\mathbb{F}_p$, and also that

$$h(r[L-\ell+1:L]) = \alpha\left[h(r[L-\ell:L]) - \phi(r[L-\ell])\alpha^{\ell}\right].$$

Since both of these operations can be computed in time $O(1)$, it is possible to hash all ℓ-prefixes and ℓ-suffixes in $\mathcal{R}$ for $\ell = L-1, L-2, ..., 1$ in time $O(NL)$.

Algorithm 4 NOT-SO-GREEDY construction of $\mathcal{G}_{\text{out}}$ using decremental hashing

Input: V, st
$E \leftarrow \emptyset\ \forall u \in V$
for $\ell = L-1, ..., 1$ **do**
  for $u = 1, ..., N$ **do**
    if $|\mathcal{O}(u)| < 2$ **then**
      $B\left(h\left(\text{st}(u)^\ell\right)\right) \leftarrow B\left(h\left(\text{st}(u)^\ell\right)\right) \cup \{u\}$
  for $v = 1, ..., N$ **do**
    for $k = 1, 2$ **do**
      $u \leftarrow k$th node in $B\left(h\left(\text{st}(v)_\ell\right)\right)$
      if $|\mathcal{O}(u)| = 1$ **then**
        $z \leftarrow$ node in $\mathcal{O}(u)$
        if $\text{st}(z)^{L+\ell-w(u,z)} \neq \text{st}(v)_{L+\ell-w(u,z)}$ **then**
          $E \leftarrow E \cup \{(u, v)\}$
      else
        $E \leftarrow E \cup \{(u, v)\}$
Output: $\mathcal{G}_{\text{out}} = (V, E, \text{st}, w)$

In Algorithm 4 we present an alternative way to construct the same graph $\mathcal{G}_{\text{out}}$ produced by Algorithm 2. This implementation uses the decremental hashing technique described above in order to efficiently look for overlaps of length ℓ, for $\ell = L, L-1, ..., 1$. At iteration ℓ, we first hash the ℓ-suffix of every read $\text{st}(u)$ that has $\Delta_{\text{out}}(u) < 2$, adding u to the bucket; i.e.,

$$B(h(\text{st}(u)^\ell)) \leftarrow B(h(\text{st}(u)^\ell)) \cup \{u\}, \tag{2}$$

where we use $B(h(x))$ to refer to the bucket of $h(x)$. We then go through all the reads once again, and for each read v, we hash its ℓ-prefix, and iterate through the reads previously placed in that bucket. Each read $\text{st}(u)$ in the bucket has an overlap of length ℓ with $\text{st}(v)$.

If $|\mathcal{O}(u)| = 0$, there is nothing to check, and we can add the edge (u, v) to the graph. If $\mathcal{O}(u) = \{z\}$, as illustrated in Fig. 11, we need to verify whether the green segments in z and v agree with each other. If they do, we discard edge (u, v). Otherwise, we add it to the graph. At the end of Algorithm 4, we obtain $\mathcal{G}_{\text{out}}$. An analogous version of Algorithm 2 can then be run to generate the $\mathcal{G}_{\text{in}}$, and the final read-overlap graph $\mathcal{G} = (V, E, \text{st}, w)$ is constructed by taking the intersection of the edge sets of $\mathcal{G}_{\text{in}}$ and $\mathcal{G}_{\text{out}}$.

6.1.1 Running time

For each value of $\ell \in \{L, L-1, ..., 1\}$, Algorithm 4 first hashes the ℓ-suffix of each read u. Since we use the Rabin-Karp-based decremental hash, given the $(\ell+1)$-suffix of u, we can compute $h\left(\text{st}(u)[L-\ell+1:L]\right)$ in time $O(1)$. Then, for each value of ℓ, we take each read v and hash its ℓ-prefix, again in $O(1)$ due to the decremental hashing. We then iterate over at most two nodes u in the bucket $B\left(h\left(\text{st}(v)_\ell\right)\right)$. For each such node u, we need to check whether v overlaps with $z \in \mathcal{O}(v)$. This corresponds to checking whether $\text{st}(z)^{L+\ell-w(u,z)} \neq \text{st}(v)_{L+\ell-w(u,z)}$ (line 12). Since $L - (w(u,z) - \ell) + 1 > \ell$, the hash for $\text{st}(z)^{L+\ell-w(u,z)}$ and $\text{st}(v)_{L+\ell-w(u,z)}$ have been previously computed, and this equality can be verified in time $O(1)$. We conclude that the worst-case running time for the algorithm is $O(NL)$.

6.2 Computing effective overlaps

Algorithm 3 starts with the virtual read extension step and the computation of the effective overlap values $\tilde{w}(u, v)$ for each $(u, v) \in E$, as described in Section 3.2. The pseudocode for this preprocessing step is in Algorithm 5. For each

Algorithm 5 Computing effective overlaps

```
1:  Input: G = (V, E, st, w)
2:  for (u, v) ∈ E do
3:      if Δ_out(u) = 1 or Δ_in(v) = 1 then
4:          w̃(u, v) ← ∞
5:      else
6:          z_1 ← I(v) \ {u}
7:          z_2 ← O(u) \ {v}
8:          a ← max{t : st(u)[t] ≠ st(z_1)[t + w(u, v) − w(z_1, v)]}
9:          b ← min{t : st(v)[t] ≠ st(z_2)[t − w(u, v) + w(u, z_2)]}
10:         w̃(u, v) ← (L − a) + (b − w(u, v) − 1)
```

$(u, v) \in E$, Algorithm 5 checks the extent of the agreement between u and z_1 to the left of the overlap between u and v and the extent of the agreement between v and z_2 to the right of the overlap between u and v. Fig. 19 illustrates how $\tilde{w}(u, v)$ is computed.

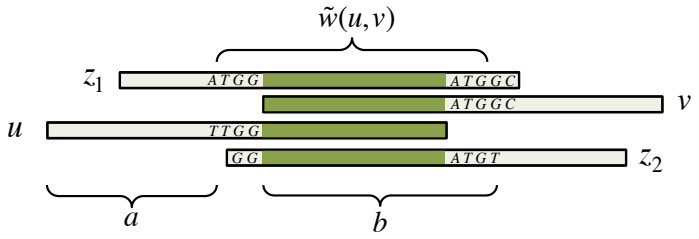


Figure 19: Algorithm 5 computes the effective overlap between u and v as $\tilde{w}(u, v) = (L - a) + (b - w(u, v) - 1)$.

6.3 Reduction to Eulerian Problem

Theorem 1 guarantees that the read-overlap graph $\widetilde{\mathcal{G}}$ produced by the NOT-SO-GREEDY algorithm contains a cycle c_s that traverses every edge of the graph, provided that the reads in $\mathcal{R}$ cover s and triple repeats are all-bridged. In addition, when these conditions are satisfied, the cycle c_s can visit any node at most twice. Otherwise, we must have a triple repeat

of length at least L, which cannot be bridged by reads of lenght L. As it turns out, this allows us to straightforwardly identify which edges in $\widetilde{\mathcal{G}}$ should be traversed twice by c_s. This can be seen as a way to compute the edge multiplicities that turn the simple graph $\widetilde{\mathcal{G}}$ into an Eulerian graph. Algorithm 6 provides an $O(|E|) = O(N)$ procedure to do that.

Algorithm 6 Identifying edge multiplicities

Input: $\widetilde{\mathcal{G}} = (V, E)$
$\text{mult}(u, v) \leftarrow 1$ for $(u, v) \in E$
$\text{visited}(u) \leftarrow 0$ for $u \in V$
$A \leftarrow \emptyset$
for $(u, v) \in E$ **do**
  if $|\mathcal{I}(u)| = 2$ and $|\mathcal{O}(u)| = 1$ **then**
    $A \leftarrow A \cup \{(u, v)\}$
for $(u_0, v_0) \in A$ **do**
  $(u, v) \leftarrow (u_0, v_0)$
  $\text{mult}(u, v) \leftarrow 2$
  while $|\mathcal{O}(v)| = 1$ and $\text{visited}(v) = 0$ **do**
    $u \leftarrow v$
    $v \leftarrow$ node in $\mathcal{O}(v)$
    $\text{mult}(u, v) \leftarrow 2, \quad \text{visited}(u) \leftarrow 1$
Output: $\text{mult}(\cdot)$

It is clear that the loop over the set E to identify edges (u, v) with $|\mathcal{I}(u)| = 2$ and $|\mathcal{O}(v)| = 1$ (set A) runs in $O(|E|) = O(N)$ time, since for $\widetilde{\mathcal{G}}$, $|E| \leq 2N$. We then pick one edge at a time from A and use it as the starting point of a path. As we move along this path, as illustrated in Fig. 20, we change the multiplicity of the edges to two, and mark their starting node as visited. This guarantees that in our search we can traverse an edge (u, v) at most once, and the run

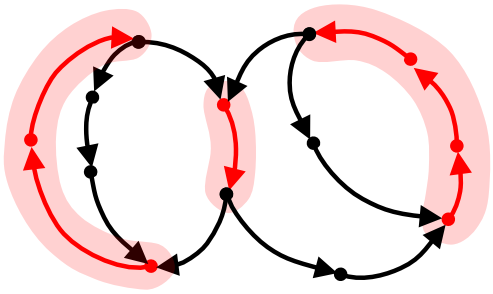


Figure 20: Algorithm 6 identifies paths starting on an edge (u, v) with $|\mathcal{I}(u)| = 2$ and ending on an edge (u, v) with $|\mathcal{O}(v)| = 2$, and sets the multiplicity of all edges on these paths to be two. In this example, the paths identified by the algorithm are highlighted in red.

time for the second for loop is also $O(|E|) = O(N)$. The correctness guarantee is stated in the following lemma.

Lemma 2. *Suppose $\widetilde{\mathcal{G}} = (V, E)$ contains a cycle that traverses every edge and visits every node at most twice. Suppose in addition that there is at least one edge that is traversed exactly once. Then the edge multiplicities computed by Algorithm 6 induce a multigraph where c_s is an Eulerian cycle.*

Proof. Let $c_s = (v_0, v_1, ..., v_M, v_0)$, where we may have v_i and v_j corresponding to the same node for some i and j. Let $t(v_i, v_{i+1}) \in \{1, 2\}$ be the number of times c_s traverses edge (v_i, v_{i+1}). Notice that, if $t(v_i, v_{i+1}) = 1$ and $t(v_{i+1}, v_{i+2}) = 2$, then we must have $|\mathcal{I}(v_{i+1})| = 2$, and $(v_{i+1}, v_{i+2}) \in A$, where $A = \{(u, v) : |\mathcal{I}(u)| = 2 \text{ and } |\mathcal{O}(u)| = 1\}$ is the set constructed in the beginning of Algorithm 6. Since c_s visits every edge in the graph, any edge (v_j, v_{j+1}) with $t(v_j, v_{j+1}) = 2$ can be reached via a path $(v_i, v_{i+1}, ..., v_j, v_{j+1})$ such that $|\mathcal{I}(v_i)| = 2$ and $t(v_k, v_{k+1}) = 2$ for $k = i, ..., j$. Moreover, it must be the case that $|\mathcal{O}(v_k)| = 1$ for $k = i, ..., j$, or else we would have a node v_k that is visited three times by c_s. We conclude that any edge (v_j, v_{j+1}) with $t(v_j, v_{j+1}) = 2$ can be reached by

a path $(v_i, v_{i+1}, ..., v_j, v_{j+1})$ that starts at an edge $(v_i, v_{i+1}) \in A$ and follows edges (v_k, v_{k+1}) in a unique manner. But these are precisely the paths that are found by Algorithm 6, and we conclude that Algorithm 6 sets $\text{mult}(u, v) = 2$ for every edge (u, v) with $t(u, v) = 2$. □

6.4 Bridging Conditions and Information Limits

In this section, we provide a brief description of the bridging conditions and the information limits for assembly feability derived in [9]. Given a set of reads $\mathcal{R}$ from a sequence s, an obvious necessary condition for perfect assembly is that the reads *cover* the string s; i.e., every base in the sequence should be part of at least one read. More precisely, we will require that every interval of length $L-1$ in s contains the starting point of at least one read. Notice that we consider intervals of lenght $L-1$ and not L so that consecutive reads overlap by at least one base. Since a read $r \in \mathcal{R}$ can technically correspond to multiple locations in s due to repeats, we formalize the notion of coverage as follows.

Definition 1. *A set of reads $\mathcal{R}$ covers a sequence s if for $1 \leq t \leq G$ there is a read $r \in \mathcal{R}$ such that $r = s[\tau : \tau + L - 1]$ for $\tau \in [t : t + L - 2]$.*

The Poisson approximation to the sampling process introduced in [10] allows one to approximate the number of reads required to guarantee that $\mathcal{R}$ covers a sequence of length G with a desired probability $1 - \epsilon$, as

$$N_{LW} = \frac{G}{L} \log \frac{G}{\epsilon}, \tag{3}$$

which we refer to as the Lander-Waterman coverage. The first necessary condition for perfect assembly to be possible (with probability $1 - \epsilon$) is that $N \geq N_{LW}$.

A *repeat* of length ℓ in s is a substring $x \in \Sigma^\ell$ appearing at distinct positions t_1 and t_2 in s; i.e., $s[t_1 : t_1 + \ell - 1] = s[t_2 : t_2 + \ell - 1] = x$, that is maximal; i.e., $s[t_1 - 1] \neq s[t_2 - 1]$ and $s[t_1 + \ell] \neq s[t_2 + \ell]$. A repeat is bridged if there is a read that extends beyond one copy of the repeat in both directions, or more formally:

Definition 2. *A repeat $s[t_1 : t_1 + \ell - 1] = s[t_2 : t_2 + \ell - 1] = x$ is bridged if there is a read $r = s[\tau : \tau + L - 1]$ for $\tau \in [t_1 + \ell + 1 - L : t_1 - 1]$ or $\tau \in [t_2 + \ell + 1 - L : t_2 - 1]$. Notice that this implicitely requires $L \geq \ell + 2$.*

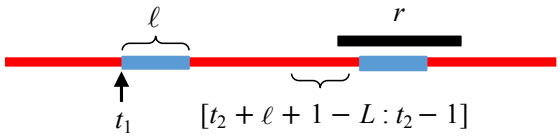


Figure 21: Illustration of a bridged repeat in s.

Two pairs of repeats $s[a_1 : a_1 + \ell - 1]$, $s[a_2 : a_2 + \ell - 1]$ and $s[b_1 : b_1 + k - 1]$, $s[b_2 : b_2 + k - 1]$ are *interleaved* if $a_1 < b_1 \leq a_2 < b_2$. This is illustrated in Fig. 22. Due to the circular DNA model described in Section 2, since a subsequence $s[t : t + \ell - 1]$ can also be written as $s[t + mG : t + mG + \ell - 1]$ for any integer m, we additionally require that $b_2 - a_1 < G$. The length of a pair of interleaved repeats $s[a_1 : a_1 + \ell - 1]$, $s[a_2 : a_2 + \ell - 1]$ and $s[b_1 : b_1 + k - 1]$, $s[b_2 : b_2 + k - 1]$ is defined as $\min(\ell, k)$. We say that an interleaved repeat is bridged if one of its four copies (two from each repeat) is bridged.

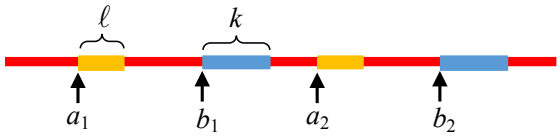


Figure 22: Illustration of interleaved repeats in s.

A triple repeat of length ℓ is a substring x that appears at three distinct locations in s (possibly overlapping); i.e., $s[t_1 : t_1+\ell-1] = s[t_2 : t_2+\ell-1] = s[t_3 : t_3+\ell-1] = x$ for distinct t_1, t_2 and t_3 (modulo G, given the circular DNA assumption). A triple repeat is said to be bridged if at least one of its copies is bridged, and it is said to be all-bridged if all of its copies are bridged, as illustrated in Fig. 23.

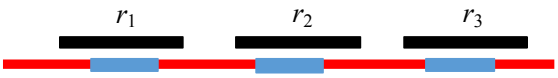


Figure 23: Illustration of a triple repeat in s that is all-bridged by reads r_1, r_2 and r_3.

The main result in terms of information requirements for perfect assembly derived in [9] is the following.

Theorem 2 ([9])**.** *If a triple repeat in s is not bridged or an interleaved repeat in s is not bridged, then there exists a sequence $s' \neq s$ with the same likelihood as s. In other words, unambiguous assembly is only possible if all interleaved repeats are bridged and triple repeats are bridged.*

Under the uniform sampling model described in Section 2, and given a known genome sequence s, one can compute the pairs (N, L) for which the probability that the necessary conditions described in Theorem 2 are satisfied. This is illustrated by the black curves in Figure 3, and by similar curves computed for several other genomes in [9]. In particular, we notice that if we let $\ell_{\text{inter}}(s)$ be the length of the longest pair of interleaved repeats in s and $\ell_{\text{triple}}(s)$ be the length of the longest triple repeat in s, Theorem 2 implies that

$$L \geq \ell_{\text{crit}}(s) \triangleq \max[\ell_{\text{inter}}(s), \ell_{\text{triple}}(s)] + 2$$

is a requirement for perfect assembly of s, and corresponds to the vertical asymptote of the curves in Fig. 3. Notice that the vertical axis corresponds to the normalized coverage depth, given by N/N_{LW}, where N_{LW} is given in (3). Since a necessary condition for assembly is that $\mathcal{R}$ covers the sequence s, after normalizing the vertical axis, the curves in Fig. 3 have a horizontal asymptote at 1.

In addition to the necessary conditions stated in Theorem 2, [9] also provided sufficient conditions for assembly. More precisely, [9] introduced an algorithm called MULTIBRIDGING with the following theoretical guarantee.

Theorem 3 ([9])**.** MULTIBRIDGING *correctly reconstructs the sequence s if*

(a) $\mathcal{R}$ covers s,

(b) Interleaved repeats in s are bridged,

(c) Triple repeats in s are all-bridged.

As in the case of the necessary conditions, one can compute pairs (N, L) for which the conditions in Theorem 3 are satisfied with a desired probability $1 - \epsilon$. Notice that the only difference between the sufficient conditions in Theorem 3 and the necessary conditions for assembly is that MULTIBRIDGING requires triple repeats to be all-bridged as opposed to being (single) bridged. This implies that the resulting sufficiency curve on the (N, L) plot (shown in green in Fig. 3) has the same vertical asymptote at $\ell_{\text{crit}}(s)$ and horizontal asymptote at 1 as the necessary curve.

Furthermore, the random nature of the reading process guarantees that in general the number of reads required for $\mathcal{R}$ to all-bridge triple repeats is not significantly larger than the number of reads required for triple repeats to be single-bridged. In fact, for most genomes considered in [9], an analysis of the repeat statistics of the sequence revealed that $\ell_{\text{inter}}(s) > \ell_{\text{triple}}(s)$. In such a regime, the requirement that $L \geq \ell_{\text{crit}}(s) = \ell_{\text{inter}}(s) + 2$ makes the all-bridging of triple repeats "easier" than the bridging of interleaved repeats. For this reason, the necessary and sufficient conditions from [9] nearly coincide for most genomes considered, fully establishing the information limits for assembly. In cases where $\ell_{\text{inter}}(s) < \ell_{\text{triple}}(s)$, the two curves may exhibit a small gap [9].