Scalable sketching and indexing algorithms for large biological datasets

by

Barış C. Ekim

S.B., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author . . . .
Department of Electrical Engineering and Computer Science
August 26, 2022

Certified by . . . .
Bonnie A. Berger
Simons Professor of Mathematics
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Scalable sketching and indexing algorithms for large biological datasets

by

Barış C. Ekim

Submitted to the Department of Electrical Engineering and Computer Science
on August 26, 2022, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

DNA sequencing data continues to progress towards longer sequencing reads with increasingly lower error rates. In order to efficiently process the ever-growing collections of sequencing data, there is a crucial need for more time- and memory-efficient algorithms and data structures. In this thesis, we propose several ways to represent DNA sequences in order to mitigate some of these challenges in practical biological tasks. Firstly, we expand upon an existing k-mer (a substring of length k) -based approach, a *universal hitting set (UHS)*, to sample a subset of locations on a DNA sequence. We show that UHSs can be efficiently constructed using a randomized parallel algorithm, and propose ways in which UHSs can be used in sketching and indexing sequences for downstream analysis. Secondly, we introduce the concept of minimizer-space sequencing data analysis, where a set of *minimizers*, rather than DNA nucleotides, are the atomic tokens of the alphabet. We propose that minimizer-space representations can be seamlessly applied to the problem of *genome assembly*, the task of reconstructing a genome from a collection of DNA sequences. By projecting sequences into ordered lists of minimizers, we claim that we can achieve orders-of-magnitude improvement in runtime and memory usage over existing methods without much loss of accuracy. We expect these approaches to be essential for downstream bioinformatics applications, such as read mapping, metagenomics, and pangenomics, as well as to provide ways to better store, search, and compress large collections of sequencing data.

Thesis Supervisor: Bonnie A. Berger
Title: Simons Professor of Mathematics

Acknowledgments

I would like to thank the following people, without whom I would not have been able to complete this thesis.

I am deeply thankful to my advisor and mentor Prof. Bonnie Berger, whose support, feedback, contribution, and nurturing steered me through almost a decade of research. She is the true definition of a leader and the ultimate role model. I could not have imagined having a better advisor and mentor for my career.

Thanks to Dr. Yaron Orenstein, for providing the insight and knowledge I needed throughout part of this research. His unassuming approach to research and science is a source of inspiration.

Thanks to Prof. Rayan Chikhi, for taking me on as a mentee and continuing to have faith in me over the years. His continued guidance, insight, and an endless supply of fascinating ideas have made this an inspiring experience.

Thanks to Brian Hie, Rohit Singh, Samuel Sledzieski, and all other members of the Berger Lab for their helpful suggestions and feedback.

I am grateful for my parents and brother, whose unconditional love and support kept me confident and grounded, and still remind me of what is important in life.

Thanks to my cat, Çorba, whose constant demands for food helped me stay awake through many nights of writing.

Finally, I owe my deepest gratitude to my partner, Amanda. I simply couldn't do it without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Efficient algorithms for sequence analysis have played a central role in the era of high-throughput DNA sequencing. Many analyses such as read mapping (e.g. [66, 62]), genome assembly (e.g. [56]), and taxonomic profiling (e.g., [43, 49]) have benefited from milestone advances that effectively compress, or sketch, the data [40]; e.g. fast full-text search with the Burrows-Wheeler transform (BWT) [8], space-efficient graph representations with succinct de Bruijn graphs [11], and lightweight databases with MinHash sketches [51]. Large-scale data re-analysis initiatives [20, 32] further incentivize the development of efficient algorithms, as they aim to re-analyze petabases of existing public data.

However, there has traditionally been a tradeoff between algorithmic efficiency and loss of information, at least during the initial sequence processing steps. Consider short-read genome assembly: The non-trivial insight of chopping up reads into k-mers, thereby bypassing the ordering of k-mers within each read, has unlocked fast and memory-efficient approaches using de Bruijn graphs; yet the short k-mers—chosen for efficiency—lead to fragmented assemblies [2]. In modern sequence similarity estimation and read mapping approaches [66] information loss is even more drastic as large genomic windows are sketched down to comparatively tiny sets of *minimizers*—which index a sequence (window) by its lexicographically smallest k-mer [51], and enable efficient but sometimes inaccurate comparisons between gigabase-scale sets of sequences [26].

In this thesis, we present two minimizer-based approaches through which we are able to represent biological data using only a tunable fraction of the input sequences. In Chapter 2[*], we describe `PASHA` (Parallel Algorithm for Small Hitting Set Approximations), the first randomized parallel algorithm to efficiently generate *universal hitting sets*, which are sets of k-length substrings that are guaranteed to share at least one k-length substring with every possible sequence of length L. In Chapter 3[†], we introduce *minimizer-space de Bruijn graphs*, mdBG, which performs genome assembly in *minimizer-space*, using minimizers as building blocks of the assembly graph. We show that using only a fraction of nucleotides in the input sequences through minimizers allows us to lower running time and decrease memory usage by 1 to 2 orders of magnitude compared to current assemblers. We show that we can also tackle sequencing errors in minimizer space, and construct pangenome graphs in minimizer space accurately and efficiently.

We hope that both of these approaches will mitigate challenges in storing, compressing, and processing the ever-growing collections of sequencing data, and enable efficient and accurate downstream analysis to answer biologically significant questions.

[*]Chapter 2 of this thesis was previously published in [23].
[†]Chapter 3 of this thesis was previously published in [22].

Chapter 2

Efficiently computing near-optimal universal hitting sets

2.1 Background and preliminaries

Minimizer techniques were introduced to select k-mers from a sequence to allow efficient binning of sequences such that some information about the sequence's identity is preserved [59]. Formally, given a sequence of length L and an integer k, its *minimizer* is the lexicographically smallest k-mer in it. The method has two key advantages: selected k-mers are close; and similar k-mers are selected from similar sequences. Minimizers were adopted for biological sequence analysis to design more efficient algorithms, both in terms of memory usage and runtime, by reducing the amount of information processed, while not losing much or any information [47]. The minimizer method has been applied in a large number of settings [18, 65, 28].

Orenstein and Pellow *et al.* [52, 53] generalized and improved upon the minimizer idea by introducing the notion of a *universal hitting set* (UHS). For integers k and L, set $U_{k,L}$ is called a universal hitting set of k-mers if every possible sequence of length L contains at least one k-mer from $U_{k,L}$. Note that a UHS for any given k and L only needs to be computed once. Their heuristic `DOCKS` finds a small UHS in two steps: (i) remove a minimum-size set of vertices from a complete de Bruijn graph of order k to make it acyclic; and (ii) remove additional vertices to eliminate all $(L-k)$-long paths. The removed vertices comprise the UHS. The first step was solved optimally, while the second required a heuristic. The method is limited by runtime to $k \leq 13$, and thus applicable to only a small subset of minimizer scenarios. Recently, Marçais *et al.* [44] showed that there exists an algorithm to compute a set of k-mers that covers every path of length L in a de Bruijn graph of order k. This algorithm gives an asymptotically optimal solution for a value of k approaching L. Yet this condition is rarely the case for real applications where $10 \leq k \leq 30$ and $100 \leq L \leq 300$. The results of Marçais *et al.* show that for $k \leq 30$, the results are far from optimal for fixed L. A more recent method by DeBlasio *et al.* [17] can handle larger values of k, but with $L \leq 21$, which is impractical for real applications. Thus, it is still desirable to devise faster algorithms to generate small UHSs.

In this chapter, we present `PASHA` (Parallel Algorithm for Small Hitting Set Approximations), the first randomized parallel algorithm to efficiently generate near-optimal UHSs. Our novel algorithmic contributions are twofold. First, we improve upon the process of calculating vertex hitting numbers, i.e. the number of $(L-k)$-long paths they go through. Second, we build upon a randomized parallel algorithm for Set Cover to substantially speedup removal of k-mers for the UHS—the major time-limiting step—with a guaranteed approximation ratio on the k-mer set size. `PASHA` performs substantially better than current algorithms at finding a UHS in terms of runtime, with only a small increase in set size; it is consequently applicable to much larger values of k.

2.1.1 Preliminary definitions

For $k \geq 1$ and finite alphabet Σ, directed graph $B_k = (V, E)$ is a **de Bruijn graph** of order k if V and E represent k- and $(k+1)$-long strings over Σ, respectively. An edge may exist from vertex u to vertex v if the $(k-1)$-suffix of u is the $(k-1)$-prefix of v. For any edge $(u, v) \in E$ with label $\mathcal{L}$, labels of vertices u and v are the prefix and suffix of length k of $\mathcal{L}$, respectively. If a de Bruijn graph contains all possible edges, it is *complete*, and the set of edges represents all possible $(k+1)$-mers. An $\ell = (L-k)$-long path in the graph, i.e. a path of ℓ edges, represents an L-long sequence over Σ (for further details, see [2]).

For any L-long string s over Σ, k-mers set M **hits** s if there exists a k-mer in M that is a contiguous substring in s. Consequently, **universal hitting set** (UHS) $U_{k,L}$ is a set of k-mers that hits any L-long string over Σ. A trivial UHS is the set of all k-mers, but due to its size ($|\Sigma|^k$), it does not reduce the computational expense for practical use. Note that a UHS for any given k and L does not depend on a dataset, but rather needs to be computed only once.

Although the problem of computing a universal hitting set has no known hardness results, there are several NP-hard problems related to it. In particular, the problem of computing a universal hitting set is highly similar, although not identical, to the (k, L)-*hitting set* problem, which is the problem of finding a minimum-size k-mer set that hits an input set of L-long sequences. Orenstein and Pellow *et al.* [52, 53] proved that the (k, L)-*hitting set* problem is NP-hard, and consequently developed the near-optimal `DOCKS` heuristic. `DOCKS` relies on the Set Cover problem, which is the problem of finding a minimum-size collection of subsets $S_1, ..., S_k$ of finite set U whose union is U.

2.1.2 The `DOCKS` heuristic

`DOCKS` first removes from a complete de Bruijn graph of order k a *decycling set*, turning the graph into a directed acyclic graph (DAG). This set of vertices represent a set of k-mers that hits all sequences of infinite length. A minimum-size decycling set can be found by Mykkeltveit's algorithm [48] in $O(|\Sigma|^k)$ time. Even after all cycles, which represent sequences of infinite length, are removed from the graph, there may still be paths representing sequences of length L, which also need to be hit by the UHS. `DOCKS` removes an additional set of k-mers that hits all remaining sequences of length L, so that no path representing an L-long sequence, i.e. a path of length $\ell = L - k$, remains in the graph.

However, finding a minimum-size set of vertices to cover all paths of length ℓ in a directed acyclic graph (DAG) is NP-hard [54]. In order to find a small, but not necessarily minimum-size, set of vertices to cover all ℓ-long paths, Orenstein and Pellow *et al.* [52, 53] introduced the notion of a *hitting number*, the number of ℓ-long paths containing vertex v, denoted by $T(v, \ell)$. `DOCKS` uses the hitting number to prioritize removal of vertices that are likely to cover a large number of paths in the graph. This, in fact, is an application of the greedy method for the Set Cover problem, thus guaranteeing an approximation ratio of $O(1+\log(\max_v T(v, \ell)))$ on the removal of additional k-mers.

The hitting numbers for all vertices can be computed efficiently by dynamic programming: For any vertex v and $0 \leq i \leq \ell$, `DOCKS` calculates the number of i-long paths starting at v, $D(v, i)$, and the number of i-long paths ending at v, $F(v, i)$. Then, the hitting number is directly computable by

$$T(v, \ell) = \sum_{i=0}^{\ell} F(v, i) \cdot D(v, \ell - i)$$

and the dynamic programming calculation in graph $G = (V', E')$ is given by

$$\begin{aligned} \forall v \in V',\ D(v, 0) &= F(v, 0) = 1 \\ D(v, i) &= \sum_{(v,u) \in E'} D(u, i-1) \\ F(v, i) &= \sum_{(u,v) \in E'} F(u, i-1) \end{aligned} \tag{2.1}$$

Overall, `DOCKS` performs two main steps: First, it finds and removes a minimum-size decycling set, turning the graph into a DAG. Then, it iteratively removes vertex v with the largest hitting number $T(v, \ell)$ until there are no ℓ-long paths in the graph. `DOCKS` is sequential: In each iteration, one vertex with the largest hitting number is removed and added to the UHS output, and the hitting numbers are recalculated. Since the first phase of `DOCKS` is solved optimally in polynomial time, the bottleneck of the heuristic lies in the removal of the remaining set of k-mers to cover all paths of length $\ell = L - k$ in the graph, which represent all remaining sequences of length L.

As an additional heuristic, Orenstein and Pellow *et al.* [52, 53] developed `DOCKSany` with a similar structure as `DOCKS`, but instead of removing the vertex that hits the most $(L-k)$-

long paths, it removes a vertex that hits the most paths in each iteration. This reduces the runtime by a factor of L, as calculating the hitting number $T(v)$ for each vertex can be done in linear time with respect to the size of the graph. `DOCKSanyX` extends `DOCKSany` by removing X vertices with the largest hitting numbers in each iteration. `DOCKSany` and `DOCKSanyX` run faster compared to `DOCKS`, but the resulting hitting sets are larger.

2.2 Methods

2.2.1 Algorithm overview

Similar to `DOCKS`, `PASHA` is run in two phases: First, a minimum-size decycling set is found and removed; then, an additional set of k-mers that hits remaining L-long sequences is removed. The removal of the decycling set is identical to that of `DOCKS`; however, in `PASHA` we introduce randomization and parallelization to efficiently remove the additional set of k-mers. We present two novel contributions to efficiently parallelize and randomize the second phase of `DOCKS`. The first contribution leads to a faster calculation of hitting numbers, thus reducing the runtime of each iteration. The second contribution leads to selecting multiple vertices for removal at each iteration, thus reducing the number of iterations to obtain a graph with no $(L-k)$-long paths. Together, the two contributions provide orthogonal improvements in runtime.

2.2.2 Improved hitting number calculation

We reduce memory usage through algorithmic and technical advances. Instead of storing the number of i-long paths for $0 \leq i \leq \ell$ in both F and D, we apply the following approach (Algorithm 1): We compute D for all $v \in V$ and $0 \leq i \leq \ell$. Then, while computing the hitting number, we calculate F for iteration i. For this aim, we define two arrays: F_{curr} and F_{prev}, to store only two instances of i-long path counts for each vertex: The current and previous iterations. Then, for some j, we compute F_{curr} based on F_{prev}, set $F_{prev} = F_{curr}$, and add $F_{curr}(v) \cdot D(v, \ell - j)$ to the hitting number sum. Lastly, we increase j, and repeat the procedure, adding the computed hitting numbers iteratively. This approach allows the reduction of matrix F, since in each iteration we are storing only two arrays, F_{curr} and F_{prev}, instead of the original F matrix consisting of $\ell + 1$ arrays. Therefore, we are able to reduce memory usage by close to half, with no change in runtime.

To further reduce memory usage, we use `float` variable type (of size 4 bytes) instead of `double` variable type (of size 8 bytes). The number of paths kept in F and D increase exponentially with i, the length of the paths. To be able to use the 8 bit exponent field, we initialize F and D to `float` minimum positive value. This does not disturb algorithm correctness, as path counting is only scaled to some arbitrary unit value, which may be 2^{-149}, the smallest positive value that can be represented by `float`. This is done in order to account for the high numbers that path counts can reach. The remaining main memory bottleneck is matrix D, whose size is $4 \cdot 4^k \cdot (\ell + 1)$ bytes.

Lastly, we utilized the property of a complete de Bruijn graph of order k being the line graph of a de Bruijn graph of order $k - 1$. While all k-mers are represented as the set of vertices in the graph of order k, they are represented as edges in the graph of order $k - 1$. If we remove edges of a de Bruijn graph of order $k - 1$, instead of vertices in a graph of order k, we can reduce memory usage by another factor of $|\Sigma|$. In our implementation we compute D and F for all vertices of a graph of order $k - 1$, and calculate hitting numbers for edges. Thus, the bottleneck of the memory usage is reduced to $4 \cdot 4^{k-1} \cdot (\ell + 1)$ bytes.

We parallelize the calculation of the hitting numbers to achieve a constant factor reduction in runtime. The calculation of i-long paths through vertex $v$only depends on the previously calculated matrices for the $(i-1)$-long paths through all vertices adjacent to v (Equation 2.1). Therefore, for some i, we can compute $D(v, i)$ and $F(v, i)$ for all vertices in V' in parallel, where V' is the set of vertices left after the removal of the decycling set. In addition, we can calculate the hitting number $T(v, \ell)$ for all vertices V' in parallel (similar to computing D and F), since the calculation does not depend on the hitting number of any other vertex (we call this parallel variant `PDOCKS` for the purpose of comparison with `PASHA`). We note that for `DOCKSany` and `DOCKSanyX`, the calculations of hitting numbers for each vertex cannot be

Algorithm 1 Improved hitting numbers calculation. *Input:* $G = (V, E)$

```
1: $D \leftarrow [|V|][\ell + 1]$, with $[|V|][0]$ initialized to $\mathbf{1}$
2: $F_{curr} \leftarrow [|V|]$
3: $F_{prev} \leftarrow [|V|]$ initialized to $\mathbf{1}$
4: $T \leftarrow [|V|]$ initialized to $\mathbf{0}$
5: for $1 \leq i \leq \ell$ do:
6:     for $v \in V$ do:
7:         for $(v, u) \in E$ do:
8:             $D[v][i] \mathrel{+}= D[u][i-1]$
9: for $1 \leq i \leq \ell + 1$ do:
10:    for $v \in V$ do:
11:        $F_{curr}[v] = 0$
12:        for $(u, v) \in E$ do:
13:            $F_{curr}[v] \mathrel{+}= F_{prev}[u]$
14:        $T[v] \mathrel{+}= F_{prev}[v] \cdot D[v][\ell - i + 1]$
15:    $F_{prev} = F_{curr}$
16: return $T$
```

computed in parallel, since the number of paths starting and ending at each vertex both depend on those of the previous vertex in topological order.

2.2.3 Parallel randomized k-mer selection

We now describe a randomized parallel k-mer selection procedure. Our goal is to find a minimum-size set of vertices that covers all ℓ-long paths. We can represent the remaining graph as an instance of the Set Cover problem. While the greedy algorithm for the second phase of `DOCKS` is serial, we will show that we can devise a parallel algorithm, which is close to the greedy algorithm in terms of performance guarantees, by picking a large set of vertices that cover nearly as many paths as the vertices that the greedy algorithm picks one by one.

In `PASHA`, instead of removing the vertex with the maximum hitting number in each iteration, we consider a set of vertices for removal with hitting numbers within an interval, and pick vertices in this set independently with constant probability. Considering vertices within an interval allows us to efficiently introduce randomization while still emulating the deterministic algorithm. Picking vertices independently in each iteration enables parallelization of the procedure. Our randomized parallel algorithm for the second phase of the UHS problem adapts that of Berger *et al.* [3] for the original Set Cover problem.

The UHS selection procedure is as follows: The input includes graph $G = (V, E)$ and randomization variables $0 < \varepsilon \leq \frac{1}{4}$, $0 < \delta \leq \frac{1}{\ell}$ (Algorithm 2). Let function `calcHit()` calculate the hitting numbers for all vertices, and return the maximum hitting number (line 2). We set $t = \lceil \log_{1+\varepsilon} T_{max} \rceil$ (line 3), and run a series of steps from t, iteratively decreasing t by 1. In step t, we first calculate the hitting numbers of all vertices (line 5); then, we define vertex set S to contain vertices with a hitting number between $(1+\varepsilon)^{t-1}$ and $(1+\varepsilon)^t$ for potential removal (lines 8-9).

Let P_S be the sum of all hitting numbers of the vertices in S, i.e. $P_S = \sum_{v \in S} T(v, \ell)$ (line 10). In each step, if the hitting number for vertex v is more than a δ^3 fraction of P_S, i.e. $T(v, \ell) \geq \delta^3 P_S$, we add v to the picked vertex set V_t (lines 11-13). For vertices with a hitting number smaller than $\delta^3 P_S$, we pairwise independently pick them with probability $\frac{\delta}{\ell}$. We test the vertices in pairs to impose pairwise independence: If an unpicked vertex u satisfies the probability $\frac{\delta}{\ell}$, we choose another unpicked vertex v and test the same probability $\frac{\delta}{\ell}$. If both are satisfied, we add both vertices to the picked vertex set V_t; if not, neither of them are added to the set (lines 14-16). This serves as a bound on the probability of picking a vertex. If the sum of hitting numbers of the vertices in set V_t is at least $|V_t|(1+\varepsilon)^t(1 - 4\delta - 2\varepsilon)$, we add the vertices to the output set, remove them from the graph, and decrease t by 1 (lines 17-20). The next iteration runs with decreased t. Otherwise, we rerun the selection procedure without decreasing t.

2.2.4 Performance guarantees

At step t, we add the selected vertex set V_t to the output set if $\sum_{v \in V_t} T(v, \ell) \geq |V_t|(1 + \varepsilon)^t(1 - 4\delta - 2\varepsilon)$. Otherwise, we rerun the selection procedure with the same value of t. We show in Appendix A.1 that with high probability, $\sum_{v \in V_t} T(v, \ell) \geq |V_t|(1+\varepsilon)^t(1 - 4\delta - 2\varepsilon)$. We also show that `PASHA` produces a cover $\alpha(1 + \log T_{max})$ times the optimal size, where $\alpha = 1/(1 - 4\delta - 2\varepsilon)$. In Appendix A.2, we give the asymptotic number of the selection

Algorithm 2 The selection procedure. *Input:* $G = (V, E), 0 < \varepsilon \leq \frac{1}{4}, 0 < \delta \leq \frac{1}{\ell}$

```
1: $R \leftarrow \{\}$
2: $T_{max} \leftarrow$ calcHit()
3: $t \leftarrow \lceil \log_{1+\varepsilon} T_{max} \rceil$
4: while $t > 0$ do
5:     if calcHit() $== 0$ then break
6:     $S \leftarrow \{\}$
7:     $V_t \leftarrow \{\}$
8:     for $v \in V$ do:
9:         if $(1+\varepsilon)^{t-1} \leq T(v, \ell) \leq (1+\varepsilon)^t$ then $S \leftarrow S \cup \{v\}$
10:    $P_S \leftarrow \sum_{v \in S} T(v, \ell)$
11:    for $v \in S$ do:
12:        if $T(v, \ell) \geq \delta^3 P_S$ then
13:            $V_t \leftarrow V_t \cup \{v\}$
14:    for $u, v \in S$ do:
15:        if $u \notin V_t$ and unirand(0,1) $\leq \frac{\delta}{\ell}$ and $v \notin V_t$ and unirand(0,1) $\leq \frac{\delta}{\ell}$ then
16:            $V_t \leftarrow V_t \cup \{u, v\}$
17:    if $\sum_{v \in V_t} T(v, \ell) \geq |V_t| \cdot (1+\varepsilon)^t (1 - 4\delta - 2\varepsilon)$ then
18:        $R \leftarrow R \cup V_t$
19:        $G = G(V \setminus V_t, E)$
20:        $t \leftarrow t - 1$
21: return $R$
```

steps and prove the average runtime complexity of the algorithm. Performance summaries in terms of theoretical runtime and approximation ratio are in Table 2.1.

2.3 Results

We compared PASHA and PDOCKS to extant methods on several combinations of k and L. We ran DOCKS, DOCKSany, PDOCKS, and PASHA over $5 \leq k \leq 10$, DOCKSanyX, PDOCKS, and PASHA for $k = 11$ and $X = 10$, and PASHA and DOCKSanyX for $X = 100, 1000$ for $k = 12, 13$ respectively, for $20 \leq L \leq 200$. We say that an algorithm is *limited by runtime* if for some value of $k \leq 13$ and for $L = 100$, its runtime exceeds 1 day (86400 seconds), in which case we stopped the operation and excluded the method from the results for the corresponding value of k. While running PASHA, we set $\delta = 1/\ell$, and $1 - 4\delta - 2\varepsilon = 1/2$ to set an emulation ratio

Table 2.1: **Summary of theoretical results for the second phase of different algorithms for generating a set of k-mers hitting all L-long sequences.** `PDOCKS` is `DOCKS` with the improved hitting number calculation, i.e. greedy removal of one vertex at each iteration. p_D, p_{DA} denote the total number of picked vertices for `DOCKS`/`PDOCKS` and `DOCKSany`, respectively. m denotes the number of parallel threads used, T_{max} the maximum vertex hitting number, and ε and δ `PASHA`'s randomization parameters.

Algorithm	`DOCKS`	`PDOCKS`	`DOCKSany`	`PASHA`
Theoretical runtime	$O((1+p_D)\|\Sigma\|^{k+1} \cdot L)$	$O((1+p_D)\|\Sigma\|^{k+1} \cdot L/m)$	$O((1+p_{DA})\|\Sigma\|^{k+1})$	$O((L^2 \cdot \|\Sigma\|^{k+1} \cdot \log^2(\|\Sigma\|^k))/(\varepsilon\delta^3 m))$
Approximation ratio	$1+\log T_{max}$	$1+\log T_{max}$	N/A	$(1+\log T_{max})/(1-4\delta-2\varepsilon)$

$\alpha = 2$ (see Section 2.2.4 and Appendix A.1). The methods were benchmarked on a 24-CPU Intel Xeon Gold (2.10GHz) with 754GB of RAM. We ran all tests using all available cores ($m = 24$ in Table 2.1).

2.3.1 Comparing runtimes and UHS sizes

We ran `DOCKS`, `PDOCKS`, `DOCKSany`, and `PASHA` for $k = 10$ and $20 \leq L \leq 200$. As seen in Figure 2-1A, `DOCKS` has a significantly higher runtime than the parallel variant `PDOCKS`, while producing identical sets (Figure 2-1B). For small values of L, `DOCKSany` produces the largest UHSs compared to other methods, and as L increases, the differences in both runtime and UHS size for all methods decrease, since there are fewer k-mers to add to the removed decycling set to produce a UHS.

We ran `PDOCKS`, `DOCKSanyX` (with $X = 10$), and `PASHA` for $k = 11$ and $20 \leq L \leq 200$. As seen in Figure 2-1C, for small values of L, both `PDOCKS` and `DOCKSanyX` have significantly higher runtimes than `PASHA`; while for larger L, `DOCKSanyX` and `PASHA` are comparable in their runtimes (with `PASHA` being negligibly slower). In Figure 2-1D, we observe that `PDOCKS` computes the smallest sets for all values of L. Indeed, its guaranteed approximation ratio is the smallest among all three benchmarked methods. While the set sizes for all methods converge to the same value for larger L, `DOCKSanyX` produces the largest UHSs for small values of L, in which case `PASHA` and `PDOCKS` are preferable.

`PASHA`'s runtime behaves differently than that of other methods. For all methods but `PASHA`, runtime decreases as L increases. Instead of gradually decreasing with L, `PASHA`'s runtime gradually decreases up to $L = 70$, at which it starts to increase at a much slower rate. This is explained by the asymptotic complexity of `PASHA` (Table 2.1). Since computing a UHS for small L requires a larger number of vertices to be removed, the decrease in runtime with increasing L up to $L = 70$ is significant; however, due to `PASHA`'s asymptotic complexity being quadratic with respect to L, we see a small increase from $L = 70$ to $L = 200$. All other methods depend linearly on the number of removed vertices, which decreases as L increases.

Despite the significant decrease in runtime in `PDOCKS` compared to `DOCKS`, pdocks was still limited by runtime to $k \leq 12$. Therefore, we ran `DOCKSanyX` with $X = 100$ and `PASHA` for $k = 12$ and $20 \leq L \leq 200$. As seen in Figures 2-1E and 2-1F, both methods follow a similar trend as in $k = 11$, with `DOCKSanyX` being significantly slower and generating significantly larger UHSs for small values of L. For larger values of L, `DOCKSanyX` is slightly faster, while `PASHA` produces sets that are slightly smaller.

At $k = 13$ we observed the superior performance of `PASHA` over `DOCKSanyX` with $X = 1000$ in both runtime and set size for all values of L. We ran `DOCKSanyX` and `PASHA` for $k = 13$ and $20 \leq L \leq 200$. As seen in Figures 2-1G and 2-1H, `DOCKSanyX` produces larger sets and is significantly slower compared to `PASHA` for all values of L. This result demonstrates that the slow increase in runtime for `PASHA` compared to other algorithms for $k < 13$ does not have a significant effect on runtime for larger values of k.

2.3.2 `PASHA` enables UHS for $k = 14, 15, 16$

Since all existing algorithms and `PDOCKS` are limited by runtime to $k \leq 13$, we report the first UHSs for $14 \leq k \leq 16$ and $L = 100$ computed using `PASHA`, run on a 24-CPU Intel Xeon Gold (2.10GHz) with 754GB of RAM using all 24 cores. Figure 2-2 shows runtimes and sizes of the sets computed by `PASHA`.

2.3.3 Density comparisons for the different methods

In addition to runtimes and UHS sizes, we report values of another measure of UHS performance known as *density*. The *density* of the minimizers scheme $d(M, S, k)$ is the fraction of

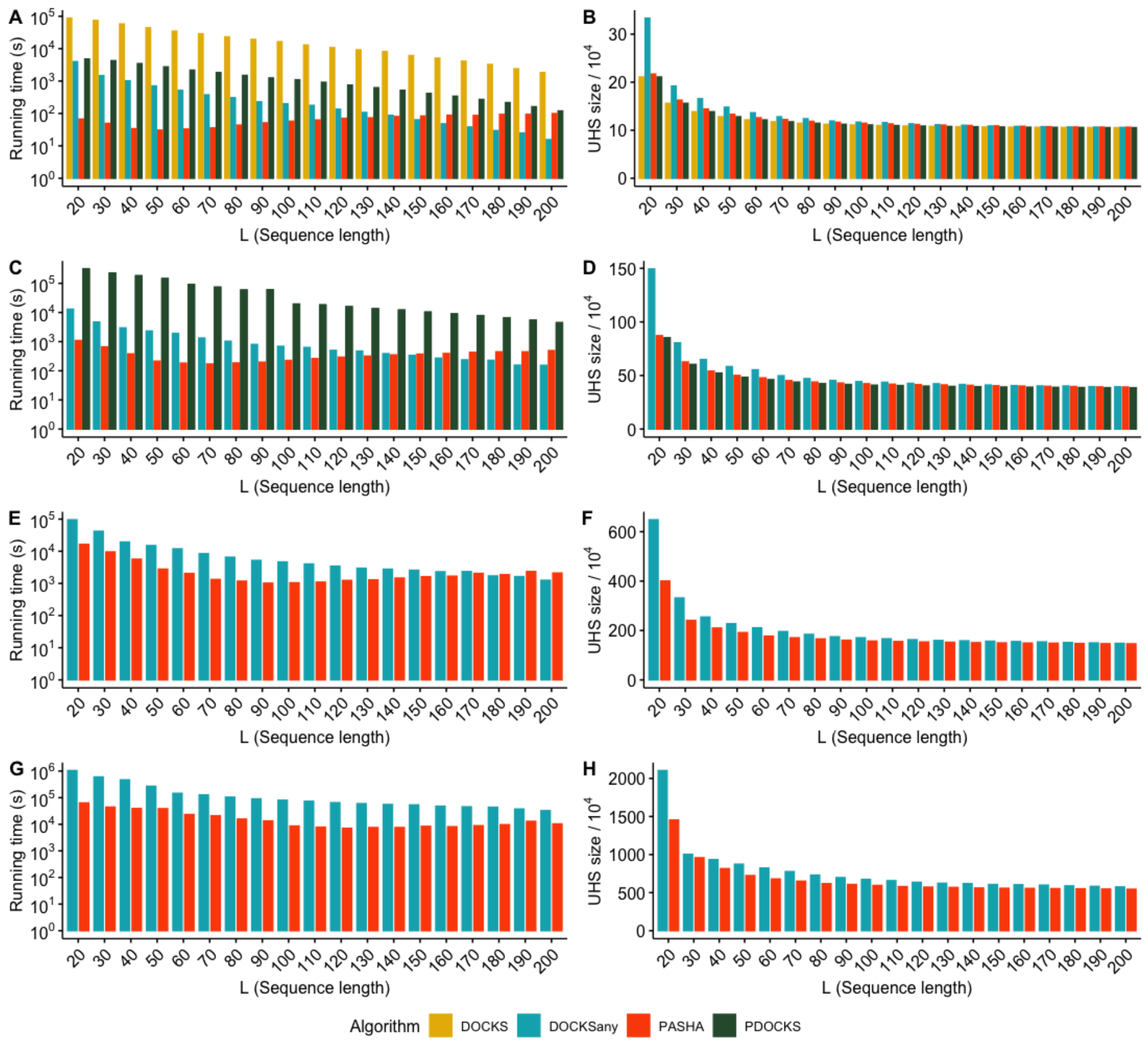


Figure 2-1: **Comparison of runtimes and set sizes of different UHS generation methods for $10 \leq k \leq 13$.** Runtimes (left) and UHS sizes (divided by 10^4, right) are given for values of $k = 10$ (A, B), 11 (C, D), 12 (E, F), and 13 (G, H) and $20 \leq L \leq 200$ for the different methods. Note that the y-axes for runtimes are in logarithmic scale.

selected k-mers' positions over the number of k-mers in the sequence. Formally, the density of scheme M over sequence S is defined as

$$d(M, S, k) = \frac{|M(S, k)|}{|S| - k + 1} \tag{2.2}$$

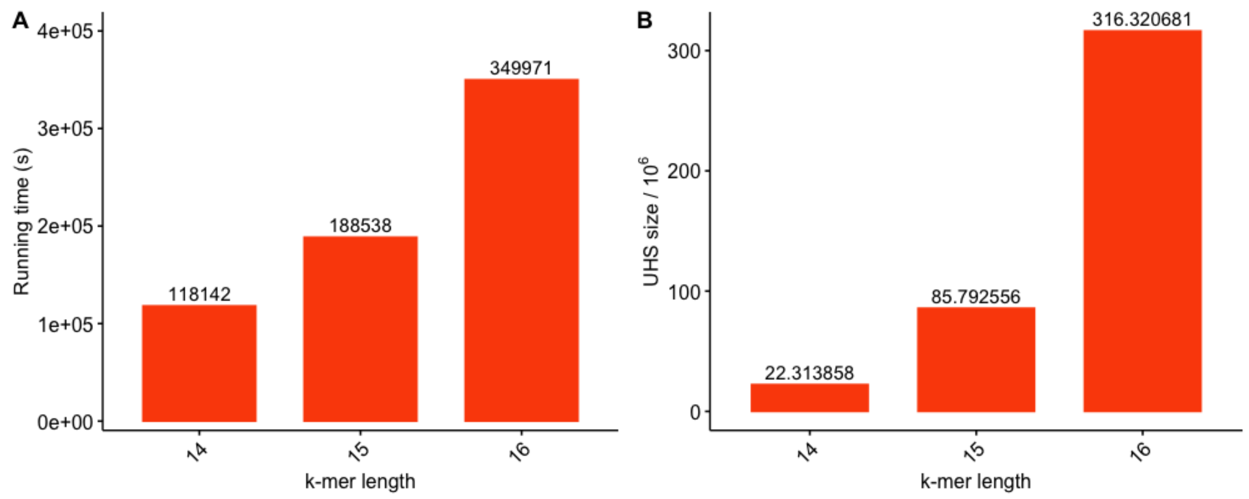


Figure 2-2: **Runtimes and set sizes of PASHA for $14 \leq k \leq 16$.** Runtimes (A) are in logarithmic scale, and UHS sizes (B) are divided by 10^6.

where $M(S, k)$ is the set of positions of the k-mers selected over sequence S.

We calculate densities for a UHS by selecting the lexicographically smallest k-mer that is in the UHS within each window of $L - k + 1$ consecutive k-mers, since at least one k-mer is guaranteed to be in each such window. Marçais *et al.* [46] showed that using UHSs for k-mer selection in this manner yields smaller densities than lexicographic or random minimizer selection schemes. Therefore, we do not report comparisons between UHSs and minimizer schemes, but rather comparisons among UHSs constructed by different methods.

Marçais *et al.* [46] also showed that the expected density of a minimizers scheme for any k and window size $L - k + 1$ is equal to the density of the minimizers scheme on a de Bruijn sequence of order L. This allows for exact calculation of expected density for any k-mer selection procedure. However, for $14 \leq k \leq 16$ we calculated UHSs only for $L = 100$, and iterating over a de Bruijn sequence of order 100 is infeasible. Therefore, we computed the approximate expected density on long random sequences, since the computed expected density on these sequences converges to the expected density [46]. In addition, we computed the density of different methods on the entire human reference genome (GRCh38).

We computed the density values of UHSs generated by `PDOCKS`, `DOCKSany`, and `PASHA` over 10 random sequences of length 10^6, and the entire human reference genome (GRCh38), for $5 \leq k \leq 16$ and $L = 100$, when a UHS was available for such (k, L) combination.

As seen in Figure 2-3, the differences in both approximate expected density and density computed on the human reference genome are negligible when comparing UHSs generated by the different methods. For most values of k, `DOCKS` yields the smallest approximate expected density and human genome density values, while `DOCKSany` generally yields lower human genome density values, but higher expected density values than `PASHA`. For $k \leq 6$, the UHS is only the decycling set; therefore, density values for these values of k are identical for the different methods.

Since there is no significant difference in the density of the UHSs generated by the different methods, other criteria, such as runtime and set size, are relevant when evaluating the performance of the methods: As k increases, `PASHA` produces sets that are only slightly smaller or larger in density, but significantly smaller in size and significantly faster than extant methods.

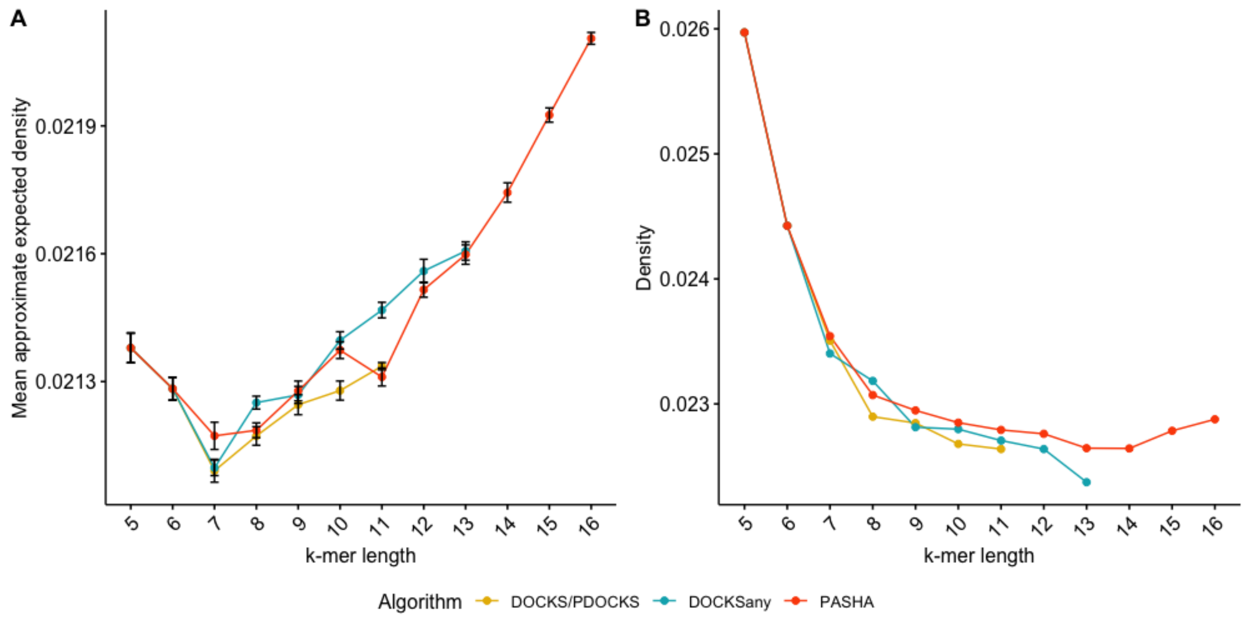


Figure 2-3: **Comparison of densities for different UHS generation methods.** Mean approximate expected density (A), and density on the human reference genome (B) for different methods, for $5 \leq k \leq 16$ and $L = 100$ are provided. Error bars represent one standard deviation from the mean across 10 random sequences of length 10^6. Density is defined by the fraction of selected k-mer positions over the number of k-mers in the sequence.

2.4 Discussion and future work

We presented an efficient randomized parallel algorithm for generating a small set of k-mers that hits every possible sequence of length L and produces a set that is a small guaranteed factor away from the optimal set size. Since the runtime of `DOCKS` variants and `PASHA` depend exponentially on k, these greedy heuristics are eventually limited by runtime. However, using these heuristics in conjunction with parallelization, we are newly able to compute UHSs for values of k and L large enough for most biological applications.

The improvements in runtime for the hitting number calculation are due to parallelization of the dynamic programming phase, which is the bottleneck in sequential `DOCKS` variants. A minimum-size set that hits all infinite-length sequences is optimally and rapidly removed; however, the remaining sequences of length L are calculated and removed in time polynomial in the output size. We show that a constant factor reduction is beneficial in mitigating this bottleneck for practical use. In addition, we reduce the memory usage of this phase by theoretical and technical advancements. Last, we build on a randomized parallel algorithm for Set Cover to significantly speed up vertex selection. The randomized algorithm can be derandomized, while preserving the same approximation ratio, since it requires only pairwise independence of the random variables [3].

One main open problem still remains from this work. Although the randomized approximation algorithm enables us to generate a UHS more efficiently, the hitting numbers still need to be calculated at each iteration. The task of computing hitting numbers remains as the bottleneck in computing a UHS. Is there a more efficient way of calculating hitting numbers than the dynamic programming calculation done in `DOCKS` and `PASHA`?

As for long reads, which are becoming more popular for genome assembly tasks, a k-mer set that hits all infinite long sequences, as computed optimally by Mykkeltveit's algorithm [48], is enough due to the length of these long read sequences. Still, due to the inaccuracies and high cost of long read sequencing compared to short read sequencing, the latter is still the prevailing method to produce sequencing data, and is expected to remain so for the near future.

We expect the efficient calculation of UHSs to lead to improvements in sequence analysis and construction of space-efficient data structures. Unfortunately, previous methods were limited to small values of k, thus allowing application to only a small subset of sequence analysis tasks. As there is an inherent exponential dependency on k in terms of both runtime and memory, efficiency in calculating these sets is crucial. We expect that the UHSs newly-enabled by `PASHA` for $k > 13$ will be useful in improving various applications in genomics.

Chapter 3

Minimizer-space de Bruijn graphs

3.1 Background and preliminaries

DNA sequencing data continues to improve from long reads of poor quality [1], used to assemble the first human genomes and Illumina short reads with low error rates ($\leq 1\%$) to longer reads with low error rates. For instance, recent Pacific Biosciences (PacBio) instruments can sequence 10 to 25 Kbp-long (HiFi) reads at $\leq 1\%$ error rate [64]. The R10.3 pore of Oxford Nanopore produces reads of hundreds of Kbps in length at a $\sim 5\%$ error rate. A tantalizing possibility is that DNA sequencing will eventually converge to long, nearly-perfect reads. These new technologies require algorithms that are both efficient and accurate for important sequence analysis tasks such as genome assembly [39].

In this chapter, we provide a highly-efficient genome assembly tool for state-of-the-art and low-error long-read data. We introduce minimizer-space de Bruijn graphs, mdBG, which instead of building an assembly over sequence bases — the standard approach that for clarity we refer to as *base-space* — newly performs assembly in *minimizer-space* (Figure 3-1), and later converts it back to base-space assemblies. Specifically, each read is initially converted to an ordered sequence of its minimizers [59, 37].

The order of the minimizers is important, as our aim is to reconstruct the entire genome as an ordered list. Our method differs from the with the classical MinHash technique, which converts sequences into unordered sets of minimizers to detect pairwise similarities between them [7]. To aid in assembly of higher-error rate data, we also introduce a variant of the partial order alignment (POA) algorithm, that operates in minimizer-space instead of base-space, and effectively corrects only the bases corresponding to minimizers in the reads. Sequencing errors that occur outside minimizers do not affect our representation. Those within minimizers cause substitutions or indels in minimizer-space (Figure 3-2), which can be identified and subsequently corrected in minimizer-space using POA (Figure 3-3).

Our key conceptual advance is that minimizers can themselves make up atomic tokens of an extended alphabet, which enables efficient long-read assembly that, along with error correction, leads to preserved accuracy. By performing assembly using a minimizer-space de Bruijn graph, we drastically reduce the amount of data input to the assembler, preserving accuracy, lowering running time, and decreasing memory usage by 1 to 2 orders of magnitude compared to current assemblers. Setting adequate parameters for the order of the de Bruijn graph and the density of our minimizer scheme allows us to overcome stochastic variations in sequencing depth and read length, in a similar fashion to traditional base-space assembly. To handle higher sequencing error rates, we correct for base errors by introducing the concept of minimizer-space partial order alignment (POA).

With error-prone data, we study two regimes: real PacBio HiFi read data ($< 1\%$ error rate) for *D. melanogaster* and Human, which turn out to require little adjustment for errors due to the very low rate, and synthetic 1 to 10% error rate data, which correspond to the range of error rates of Oxford Nanopore's recent technology. We also demonstrate that despite data reduction, running our `rust-mdbg` software on synthetic error-free and 4% error-rate data results in near-perfect reconstruction of a genome, the latter entirely due to our application of POA in minimizer-space.

To further demonstrate `rust-mdbg`'s capabilities, we use it to assemble two PacBio HiFi metagenomes, achieving runtimes of minutes as opposed to days, and memory usage two orders of magnitude lower than the current state-of-the-art `hifiasm-meta`, with comparable assembly completeness yet lower contiguity. As a versatile use case of minimizer-space analysis, we construct, to the best of our knowledge, the largest pangenome graph to date of 661K bacterial genomes, and perform minimizer-space queries of anti-microbial resistance (AMR) genes within this graph, identifying nearly all those with high sequence similarity to original bacterial genomes. Rapidly detecting AMR genes in a large collection of samples would facilitate real-time AMR surveillance [24], and mdBG provides a space-efficient alternative to indexed k-mer search.

Remarkably, our approach is equivalent to examining a tunable fraction (e.g. only 1%) of the input bases in the data, and should generalize to emerging sequencing technologies.

3.1.1 Comparison with related work

This work is at the confluence of three core ideas that were just recently proposed in three different genome assemblers: `Shasta` [61], `wtdbg2` [60] and `Peregrine` [15]. 1) `Shasta` transforms ordered lists of reads into minimizers (`Shasta` used the term *markers*) to produce an efficiently reduced representation of sequences that facilitates quick detection of overlaps between reads. A similar idea was previously used for read mapping and assembly in `minimap`/`miniasm` [35, 36] and edit distance calculation with Order Min Hash (OMH) [45]. 2) The `wtdbg2` idea extends the usual $\Sigma = \{\texttt{A,C,T,G}\}$ alphabet, which forms the basis of traditional genome de Bruijn graphs, to 256 bp windows: A 'fuzzy' de Bruijn graph is constructed by 'zooming out' of read sequences, and considering batches of 256 bps at a time. 3) The `Peregrine` idea can be broken down into two parts: i) pairs of consecutive minimizers can be indexed - and they are naturally less often repeated across a genome than isolated minimizers, and ii) a hierarchy of minimizers can be constructed so that fewer minimizers are selected than in classical methods, thus increasing the distance between minimizers.

In distantly-related independent work, a very recent pre-print [58] (MBG) demonstrates a similar idea as `Peregrine`, performing assembly by finding pairs of consecutive minimizers on reads. Although MBG does combine the concepts of minimizers and de Bruijn graphs, it is fundamentally different from the work presented here. Nodes in the MBG are classical k-mers over the DNA alphabet, whereas nodes in our representation are k-mers over an alphabet of minimizers. Two other related concepts to MBG are sparse de Bruijn graphs [65] and A-Bruijn graphs [31, 38], in which the nodes are a subset of the original de Bruijn graph nodes and the edge condition is relaxed so that overlaps may be shorter than $(k-1)$ when pairs of nodes are seen consecutively in a read.

Conceptually, our advance is in tightly combining both de Bruijn graphs and minimizers, introducing a non-trivial mix of previously-known ingredients. The concept of a de Bruijn graph was not considered in either the `Shasta` or the `Peregrine` assemblers; whereas in the `wtdbg2` assembler, de Bruijn graphs were considered, but not minimizers. Moreover, reducing the three aforementioned genome assemblers into a single idea for each of them, in terms of how they achieve algorithmic efficiency, is a contribution in itself and simplifies our presentation greatly. What we offer is essentially an ultra-fast variation of de Bruijn graphs, for long reads.

3.1.2 Preliminary definitions

The variable σ is used as a placeholder for an unspecified *alphabet* (a non-empty set of *characters*). We define $\Sigma_{\text{DNA}} = \{\texttt{A, C, T, G}\}$ as the alphabet containing the four DNA bases. Given an integer $\ell > 0$, Σ^ℓ is the alphabet consisting of all possible strings on Σ_{DNA} of length ℓ. To avoid confusion, we stress that Σ^ℓ is an unusual alphabet: Any 'character' of Σ^ℓ is itself a string of length ℓ over the DNA alphabet.

Given an alphabet σ, a *string* is a finite ordered list of characters from σ. Note that our strings will sometimes be on alphabets where each character cannot be represented by a single alphanumeric symbol. Given a string x over some alphabet σ and some integer $n > 0$, the *prefix* (respectively the *suffix*) of x of length n is the string formed by the first (respectively the last) n characters of x.

We now introduce the concept of a *minimizer*. In this paragraph, we consider strings over the alphabet Σ_{DNA}. We consider two types of minimizers: *universe* and *window*. Consider a function f that takes as input a string of length ℓ and outputs a numeric value within range $[0, H]$, where $H > 0$. Usually, f is a 4-bit encoding of DNA, or a random hash function (it does not matter whether the values of f are integers or whether H is an integer). Given an integer $\ell > 1$ and a coefficient $0 < \delta < 1$, a *universe* (ℓ, δ)*-minimizer* is any string m of length ℓ such that $f(m) < \delta \cdot H$. We define $M_{\ell,\delta}$ to be the set of all universe (ℓ, δ)-minimizers, and we refer to δ as the *density* of $M_{\ell,\delta}$.

This definition of a minimizer is in contrast with the classical one [59] which we recall here, although we will not use it. Consider a string x of any length, and a substring (*window*) y of length w of x. A *window* ℓ*-minimizer* of x given window y is a substring m of length ℓ of y that has the smallest value $f(m)$ among all other such substrings in y. Observe that universe minimizers are defined independently of a reference string, unlike window minimizers. They have been recently independently termed mincode syncmers [19]. We also performed experiments with an alternative concept to minimizers, Locally Consistent Parsing (LCP) [16], which replaces universal minimizers with *core substrings*: substrings that can be pre-computed for any given alphabet such that any sequence of length n includes $\sim n/\ell$ substrings of length ℓ on average.

We recall the definition of de Bruijn graphs. Given an alphabet σ and an integer $k \geq 2$, a de Bruijn graph of order k is a directed graph where nodes are strings of length k over σ (k-mers), and two nodes x, y are linked by an edge if the suffix of x of length $k-1$ is equal to the prefix of y of length $k-1$. This definition corresponds to the node-centric de Bruijn graph [12] generalized to any alphabet.

3.2 Methods

3.2.1 Classical and minimizer-space de Bruijn graphs

We say that an algorithm or a data structure operates in *minimizer space* when its operations are done on strings over the Σ^ℓ alphabet, with characters from $M_{\ell,\delta}$. Conversely, it operates in *base-space* when the strings are over the usual DNA alphabet Σ_{DNA}.

We introduce the concept of (k, ℓ, δ)*-min-mer*, or just k*-min-mer* when clear from the context, defined as an ordered list of k minimizers from $M_{\ell,\delta}$. We use this term to avoid confusion with k-mers over the DNA alphabet. Indeed, a k-min-mer can be seen as a k-mer over the alphabet Σ^ℓ, i.e. a k-mer in minimizer-space. For an integer $k > 2$ and an integer $\ell > 1$, we define a *minimizer-space de Bruijn graph* (mdBG) of order k as de Bruijn graph of order k over the Σ^ℓ alphabet. As per the definition in the previous section, nodes are k-min-mers, and edges correspond of identical suffix-prefix overlaps of length $k - 1$ between k-min-mers. Figure 3-1 shows an example.

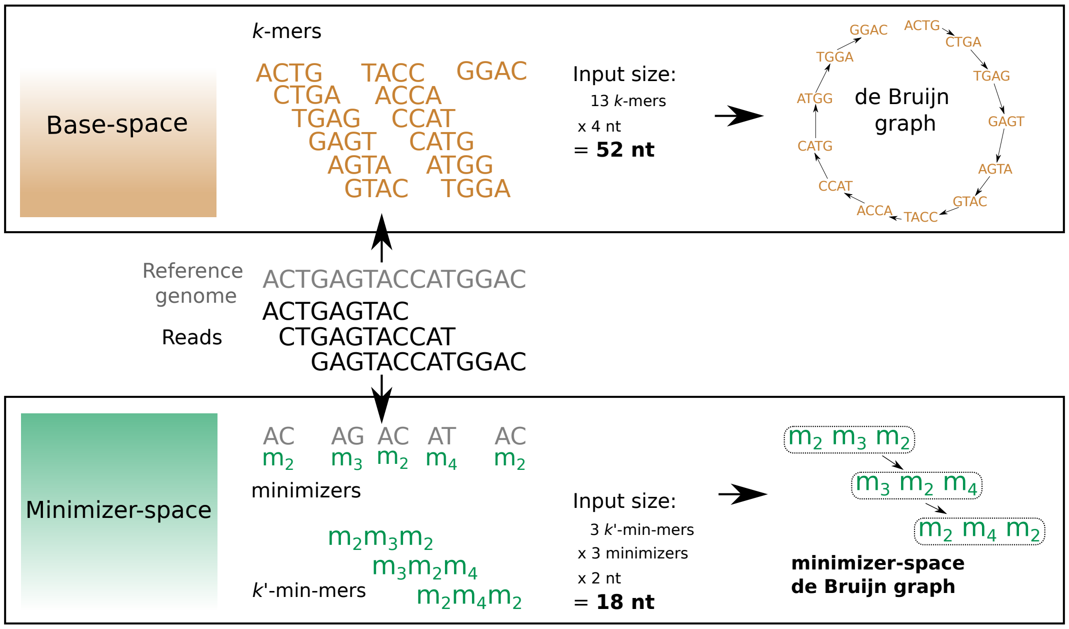


Figure 3-1: **An efficient assembly method for state-of-the-art genome sequencing (e.g. PacBio HiFi data).** Illustration of our minimizer-space de Bruijn graph (mdBG, bottom) compared to the original de Bruijn graph (top). Center horizontal section shows a toy reference genome, along with a collection of sequencing reads. Top box shows k-mers ($k = 4$) collected from the reads, which are the nodes of the classical de Bruijn graph. The input size of 52 nucleotides (nt) is depicted in boldface. Bottom box shows the position of minimizers in the reads for $\ell = 2$, and any ℓ-mer starting with nucleotide 'A' is chosen as a minimizer. k'-min-mers ($k' = 3$) are tuples of k' minimizers as ordered in reads, which constitute the nodes of the minimizer-space de Bruijn graph. The reduction in input size to 18 nucleotides (nt) is depicted in boldface.

We present our procedure for constructing mdBGs as follows. First, a set M of minimizers are pre-selected using the universe minimizer scheme from the previous section. Then, reads are scanned sequentially, and positions of elements in M are identified. A multiset V of k-min-mers is created by inserting all tuples of k successive elements in $M_{\ell,\delta}$ found in the reads into a hash table. Each of those tuples is a k-min-mer, i.e., a node of the mdBG. Edges of the mdBG are discovered through an index of all $(k-1)$-min-mers present in the k-min-mers.

mdBGs can be simplified and compacted similarly to base-space de Bruijn graphs, using similar rules for removing likely artefactual nodes (tips and bubbles), and performing path compaction. They are also bidirected, though we present them as directed here for simplicity.

By itself the mdBG is insufficient to fully reconstruct a genome in base-space, as in the best case it can only provide a sketch consisting of the ordered list of minimizers present in each chromosome. To reconstruct a genome in base-space, we associate to each k-min-mer the substring of a read corresponding to that k-min-mer. The substring likely contains base-space sequencing errors, which we address at the end of this paragraph. To deal with overlaps, we also keep track of the positions of the second and second-to-last minimizers in each k-min-mer. After performing compaction, the base sequence of a compacted mdBG can be reconstructed by concatenating the sequences associated to k-min-mers, making sure to discard overlaps. Note that in the presence of sequencing errors, or when the same k-min-mer corresponds to several locations in the genome, the resulting assembled sequence will be imperfect (similar to the output of `miniasm` [35]) which can be fixed by additional base-level polishing (not performed here).

3.2.2 How sequencing errors propagate to minimizer-space

In order to clarify the difference between base-space and minimizer-space in the presence of sequencing errors, we newly derive an expression of the expected error rate in minimizer-space (parameterized by k, ℓ, and δ), using a Poisson process model of random site mutations that was invoked by Mash [51]. Given the probability d of a single base substitution, the probability that no mutation will occur in a given ℓ-mer is $e^{-\ell d}$ under a Poisson model.

To estimate the number of erroneous k-min-mers in a read, we define for a given read R, the expected number n_R of universe (ℓ, δ)-minimizers in the read as $n_R = (|R| - \ell + 1) \cdot \delta$. Since a k-min-mer is erroneous whenever at least one of k universe (ℓ, δ)-minimizers within the k-min-mer is erroneous, the probability that a given k-min-mer is erroneous is then $1 - e^{-\ell dk}$. The number of k-min-mers obtained from the read is $n_R - k + 1$. Thus, the expected number of erroneous k-min-mers in a read is

$$(n_R - k + 1) \cdot (1 - e^{-\ell dk})$$

For instance, for a base-space mutation rate of $d = 0.01$, minimizer-space parameters $\ell = 12$, $k = 10$, and $\delta = 0.01$, and a read length of $|R| = 20000$, 70% of the k-min-mers in the read are erroneous. However, lowering the base-space mutation rate to $d = 0.001$ and keeping other values of k and ℓ identical renders only 10% of the k-min-mers erroneous within a read.

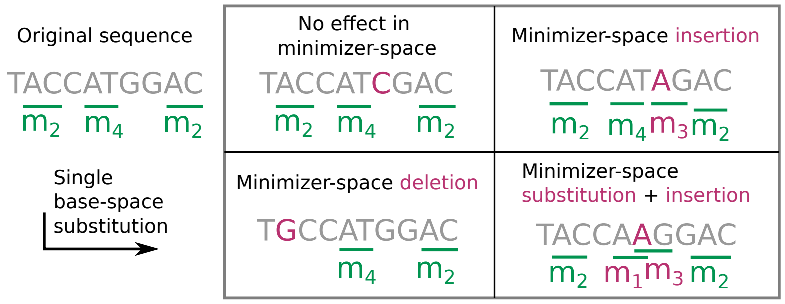


Figure 3-2: **Propagation of sequencing errors in base-space to minimizer-space**. Continuing the example in Figure 3-1, we consider a sequence along with its minimizers (left of the box). Each panel inside the box depicts the effect of a different mutation on the sequence. Top left panel: G→C (in purple) leads to no change in the minimizer-space representation as the mutation did not change or create any minimizer. Bottom left: A→G led to the disappearance of m_2. Top right: C→A made the m_3 minimizer appear. Bottom right: T→A affected two minimizers: m_4 was substituted for m_1, and m_3 was inserted.

To estimate the average ℓ-mer identity of a read, we provide an approximation of the minimizer-space error rate given the base-space error rate. As seen above, an ℓ-mer that was selected as a universe minimizer has probability $e^{-\ell d}$ to be mutated. Mutations that occur outside of universe minimizers may now still affect the minimizer-space representation by turning a non-minimizer ℓ-mer into a universe minimizer (see Figure 3-2). Under the simplifying assumption that this effect occurs independently at each position in a read, the probability that an ℓ-mer turns into a universe minimizer is the probability of a mutation within that ℓ-mer times the probability δ that a random ℓ-mer is a universe minimizer, i.e., $(1 - e^{-\ell d})\delta$. For a universe minimizer m, there are approximately $1/\delta$ neighboring ℓ-mers that are candidates for turning into universe minimizers themselves due to a base error. We will conceptually attach those ℓ-mers to m, and consider that an error in any of those ℓ-mers leads to an insertion error next to m.

Combining the above terms leads to the following minimizer-space error rate approximation:

$$1 - e^{-\ell d}(1 - (1 - e^{-\ell d})\delta)^{1/\delta} \tag{3.1}$$

For an error rate of $d = 5\%$, i.e. close to that of the Oxford Nanopore R10.3 chemistry, $\ell = 12$, and $\delta = 0.01$, the minimizer-space error rate is 65.1%, dropping to 2.3% when $d = 0.1\%$. This analysis indicates that parameters ℓ, k, δ and the base error rate d together play an essential role in the performance of a mdBG-guided assembly.

3.2.3 Correcting sequencing errors in minimizer-space

Long-read sequencing technologies from Pacific Biosciences (PacBio) and Oxford Nanopore (ONT) recently enabled the production of genome assemblies with high contiguity, albeit with a relatively high error rate ($\geq$ %5) in the reads, requiring either read error correction and/or assembly polishing, which are both resource-intensive steps [14, 41]. We will demonstrate that our minimizer-space representation is applicable to error-free sequencing reads and PacBio HiFi reads, which boast error rates lower than %1; however, in order to work with long reads with a higher error rate such as PacBio CLR and ONT, we present a resource-frugal error correction step that uses partial order alignment (POA) [33], a graph representation of a multiple sequence alignment (MSA), in order to rapidly correct sequencing errors that occur in the minimizer-space representation of reads. Stand-alone error correction modules such as `racon` [63] and `Nanopolish` [41] have also relied on POA for error correction of long reads; however, these methods work in base-space, and as such, are still resource-intensive. We present an error correction module that uses POA in minimizer-space that can correct errors in minimizer-space, requiring only the minimizer-space representation of reads as input.

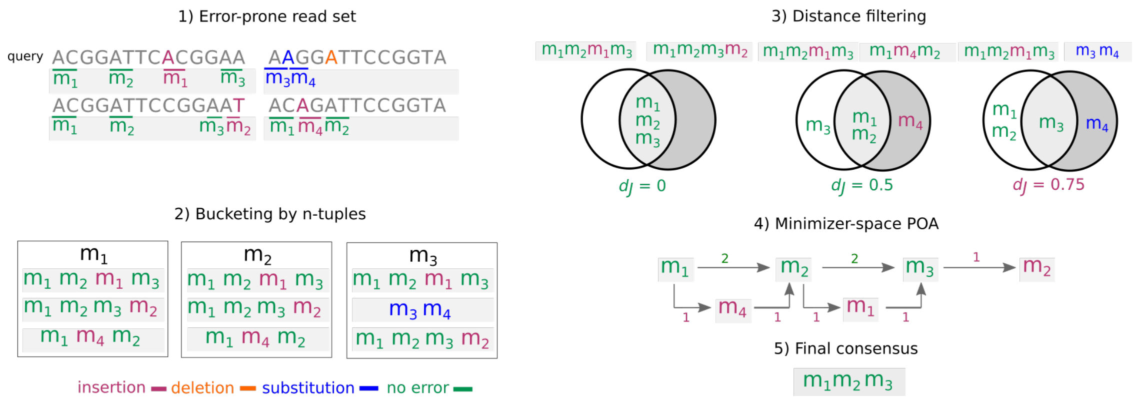


Figure 3-3: **Overview of the minimizer-space partial order alignment (POA) procedure** with a toy dataset of 4 reads. **1)** Error-prone reads and their ordered lists of minimizers ($\ell = 2$) are shown, with sequencing errors and the minimizers that are created as a result of errors denoted in colors (insertion as red, deletion as orange, substitution in blue, no errors in green). **2)** Before minimizer-space error-correction, the ordered lists of minimizers are bucketed using their n-tuples ($n = 1$). **3)** For a query ordered list (the first read in the read set in the figure), all ordered lists that share an n-tuple with the query are obtained, and the final list of query neighbors are obtained by applying a heuristically determined distance filter d_j (Jaccard distance threshold of $\varphi = 0.5$). **4)** A POA graph in minimizer-space is constructed by initializing the graph with the query, and aligning each ordered list that passed the filter to the graph iteratively (weights of poorly-supported edges are shown in red). **5)** By taking a consensus path of the graph, the error in the query is corrected.

An overview of the minimizer-space POA procedure is shown in Figure 3-3. The input for the procedure is the collection of ordered lists of minimizers obtained from all reads in the dataset (one ordered list per read). As seen earlier, the ordered list of minimizers obtained from a read containing sequencing errors will likely differ from that of an error-free read. However, provided the dataset has enough coverage, the content of other ordered lists of minimizers in the same genomic region can be used to correct errors in the query read in minimizer-space. To this end, we first perform a bucketing procedure for all ordered lists of minimizers using each of their n-tuples, where n is a user-specified parameter.

After bucketing, in order to initiate the error-correction of a query we collect its *neighbors*: other ordered lists likely corresponding to the same genomic region. We use a distance metric

(Jaccard or Mash [51] distance) to pick sufficiently similar neighbors. Once we obtain the final set of neighbors that will be used to error-correct the query, we run the partial order alignment (POA) procedure as described in [33], with the modification that a node in the POA graph is now a minimizer instead of an individual base, directed edges now represent whether two minimizers are adjacent in any of the neighbors, and edge weights represent the multiplicity of the edge in all of the neighbor ordered lists. After constructing the minimizer-space POA by aligning all neighbors to the graph, we generate a consensus (the best-supported traversal through the graph). Once the consensus is obtained in minimizer-space, we replace the query ordered list of minimizers with the consensus, and repeat until all reads are error-corrected. In order to recover the base-space sequence of the obtained consensus after POA, we store the sequence spanned by each pair of nodes in the edges, and generate the base-space consensus by concatenating the sequences stored in the edges of the consensus.

POA bucketing and preprocessing

In Algorithm 3, all tuples of length n of an ordered list of minimizers are computed using a sliding window (lines 4-6), and the ordered list of minimizers itself is stored in the buckets labeled by each n-tuple (line 7). We use bucketing as a proxy for set similarity, since each pair of reads in the same bucket will have an n-tuple (the label of the bucket), and will be more likely to come from the same genomic region.

Algorithm 3 Bucketing procedure for all ordered lists of minimizers

```
Input: Set of ordered list of minimizers $S$, bucket index length $n$
1: procedure BUCKET($S, n$)
2:     $B \leftarrow \{\}$                              ▷ Empty hash table of buckets
3:     for $s \in S$ do
4:         for $i = 0$ to $i = |s| - n + 1$ do
5:             $t \leftarrow s[i : i + n]$              ▷ $n$-tuple of $s$ starting at position $i$
6:             $B[t] \leftarrow B[t] \cup s$
7:     return $B$
```

The overview of the collection of neighbors for error-correcting a query ordered list of minimizers is shown in Algorithm 4. We obtain all n-tuples of a query ordered list, and collect the ordered lists in the previously populated buckets indexed by its n-tuples (lines 10-15). These ordered lists are viable candidates for neighbors, since they share a tuple of length at least n with the query ordered list; however, since a query n-tuple may not uniquely identify a genomic region, we apply a similarity filter to further eliminate candidates unrelated to the query. Using either Jaccard or Mash distance [51] as a similarity metric, for a user-specified threshold φ, we filter out all candidates that have distance $\geq \varphi$ to the query ordered list to obtain the final set of neighbors that will be used for error-correcting the query (lines 1-9).

Algorithm 4 Collection of neighbors for a given query ordered list

```
Input: A query ordered list of minimizers $q$ to be error-corrected, collection of buckets $B$,
    bucket index length $n$, distance function $d$, distance threshold $\varphi$
 1: function FILTER($q, C, d, \varphi$)
 2:     $F \leftarrow \{\}$                              ▷ Empty set of candidates that pass the filter
 3:     for $c \in C$ do
 4:         if $d(q, c) < \varphi$ then                 ▷ Apply distance threshold of $\varphi$ to a candidate
 5:             $F \leftarrow F \cup c$
 6:     return $F$
 7: procedure COLLECT($q, B, n, d, \varphi$)
 8:     $C \leftarrow \{\}$                              ▷ Empty set of candidate neighbors
 9:     for $i = 0$ to $i = |q| - n + 1$ do
10:         $t \leftarrow q[i : i + n]$                  ▷ $n$-tuple of $q$ starting at position $i$
11:         $C \leftarrow C \cup B[t]$
12:     $F \leftarrow$ FILTER($q, C, d, \varphi$)
13:     return $F$
```

POA graph construction and consensus generation

Algorithm 6 describes a canonical POA consensus generation procedure, similar to `racon` [63], except that here consensus is performed in minimizer-space.

The minimizer-space POA error-correction procedure is shown in Algorithm 5. For each neighbor of the query, we perform semi-global alignment between a neighbor ordered list and the graph, where for two minimizers m_i and m_j, a match is defined as $m_i = m_j$, and a mismatch is defined as $m_i \neq m_j$ (lines 17-19). After building the POA graph $G = (V, E)$

Algorithm 5 Minimizer-space POA graph construction and consensus generation

Input: A query ordered list of minimizers q to be error-corrected, collection of query neighbors N

```
 1: procedure POA($q, N$)
 2:     $G = (V, E) \leftarrow$ initializeGraph($q$)                ▷ As described in [33]
 3:     for $n \in N$ do
 4:         $G \leftarrow$ semiGlobalAlign($G, n$)                  ▷ As described in [33]
 5:     $\lambda \leftarrow \{\}$                                    ▷ Scoring table for nodes
 6:     $P \leftarrow \{\}$                                          ▷ Predecessor table for nodes
 7:     topologicalSort($G$)                                         ▷ Topological sorting of nodes
 8:     for $v \in V$ do
 9:         $e = (u, v) \leftarrow \max($inEdges($v$)$)$ ▷ Find the maximum-weighted incoming edge to $v$
10:         $\lambda[v] \leftarrow w_e + \lambda[u]$
11:         $P[v] \leftarrow u$
12:     $C \leftarrow$ CONSENSUS($V, \lambda, P$)   ▷ Described in the "Minimizer-space POA" Section
13:     return $C$
```

by aligning all neighbors in minimizer space, we generate a consensus to obtain the best-supported traversal through the graph. We first initialize a scoring λ, and set $\lambda[v] = 0$ for all $v \in V$. Then, we perform a topological sort of the nodes in the graph, and iterate through the sorted nodes. For each node v, we select the highest-weighted incoming edge $e = (u, v)$ with weight w_e, and set $\lambda[v] = w_e + \lambda(u)$. The node u is then marked as a predecessor of v (lines 21-28).

Minimizer-space POA evaluation set-up

We extracted chromosome 4 ($\sim$ 1.2 Mbp) of the *D. melanogaster* reference genome, and simulated reads using the command `randomreads.sh pacbio=t` of `BBMap` [9]. We generated one dataset per error rate value from 0% to 10%, keeping other parameters identical (24 Kbp mean read length and 70X coverage). Reads were then assembled using our implementation with and without POA, using parameters $\ell = 10$, $k = 7$, and $\delta = 0.0008$ experimentally determined to yield a perfect assembly with error-free reads. We evaluated the average read identity in minimizer-space using semi-global Smith-Waterman alignment between the sequence of minimizers of a read and the sequence of minimizers of the reference, taking

Algorithm 6 Consensus generation on POA graph

```
Input: The node set $V$ of the POA graph, scoring array $\lambda$, predecessor array $P$
1: function CONSENSUS($V, \lambda, P$)
2:     $C \leftarrow []$                                   ▷ Consensus path to be obtained
3:     $v_{max} \leftarrow \emptyset$                      ▷ Initialize the highest-scoring node
4:     for $v \in V$ do
5:         if $\lambda[v] > \lambda[v_{max}]$ then
6:             $v_{max} \leftarrow v$
7:     $v_{curr} \leftarrow v_{max}$                       ▷ Start traceback from highest-scoring node
8:     while $v_{curr} \neq \emptyset$ do
9:         $C \leftarrow C + [v_{curr}]$
10:        $v_{curr} \leftarrow P[v_{curr}]$               ▷ Move to predecessor of current node
11:    return $C$
```

BLAST-like identity (number of minimizer matches divided by the number of alignment columns). We also evaluated the length of the longest reconstructed contig in base-space as a proxy for assembly quality.

3.3 Results

An overview of our pipeline, implemented in Rust (`rust-mdbg`), is shown in Figure 3-4. We compared `rust-mdbg` to three recent assemblers optimized for low-error rate long reads: `Peregrine`, `HiCanu` [50] and `hifiasm` [10] (see Appendix A.4 for versions and parameters).

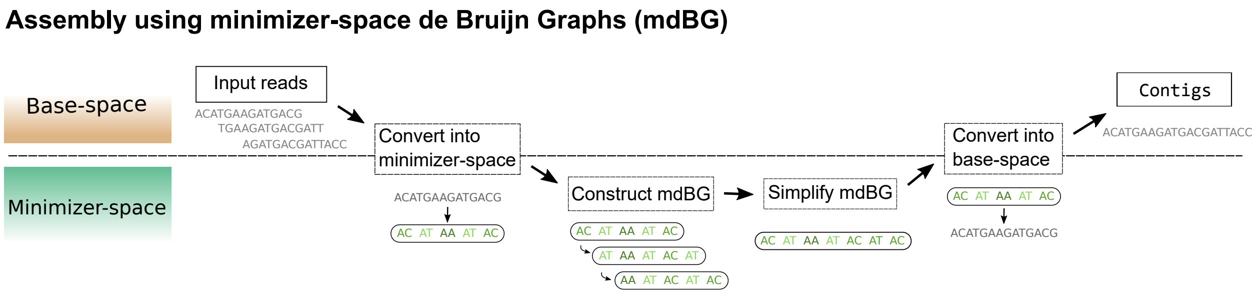


Figure 3-4: **Overview of the assembly pipeline using mdBG**. The region of the figure above (respectively below) the dotted line corresponds to analyses taking place in base-space (respectively minimizer-space). The input reads are converted into minimizer-space to construct a mdBG, which is then simplified and converted back into base-space to output contigs.

3.3.1 Ultra-fast, memory-efficient and highly-contiguous assembly of real HiFi reads using `rust-mdbg`

	D. mel 100x real HiFi reads				*D. mel* 50x simulated perfect reads				Human real HiFi reads		
Tool	`Peregr.`	`HiCanu`	`hifiasm`	`rust-mdbg`	`Peregr.`	`HiCanu`	`hifiasm`	`rust-mdbg`	`Peregr.`	`hifiasm`	`rust-mdbg`
Time	40m11s	7h43m	5h17m	**1m9s**	23m31s	8h12m	19h38m	**21s**	14h8m	58h41m	**10m23s**
Memory (GB)	12 GB	12 GB	21 GB	**1.5 GB**	16 GB	18 GB	51 GB	**<1 GB**	188 GB	195 GB	**10 GB**
# contigs	682	928	538	93	63	45	48	34	8109	431	805
NGA50 (M)	5.2	10.1	4.8	6.0	6.3	19.4	21.5	15.4	18.2	22.0	14.0
Complete (%)	93.9%	96.6%	96.6%	90.8%	98.2%	98.1%	98.2%	96.2%	97.0%	94.2%	95.5%
# misasm.	10	5	0	0	3	5	0	1	312	7942	1073

Table 3.1: **Assembly statistics of *D. melanogaster* real HiFi reads (left), simulated perfect reads (center), and Human real HiFi reads (right), all evaluated using the commonly-used `QUAST` program**. All assemblies were homopolymer-compressed. Wall-clock time is reported for 8 threads. NGA50 is a contiguity metric reported in Megabases (Mbp) by `QUAST` as the longest contig alignment to the reference genome so that shorter contig alignments collectively make up 50% of the genome length. The number of misassemblies is reported by `QUAST`. NGA50 and Genome fraction (Complete%) should be maximized, whereas all other metrics should be minimized. Only `Peregrine`, `hifiasm` and our method `rust-mdbg` were evaluated on Human assemblies, since `HiCanu` requires around an order of magnitude more running time.

We evaluated our software, `rust-mdbg`, on real PacBio HiFi reads from *D. melanogaster*, at 100X coverage, and HiFi reads for human (HG002) at $\sim$ 50X coverage, both taken from the `HiCanu` publication* [50].

Since our method does not resolve both haplotypes in diploid organisms, we compared against the primary contigs of `HiCanu` and `hifiasm`. In our tests with *D. melanogaster*, the reference genome consists of all nuclear chromosomes from the RefSeq accession GCA_000001215.4. Assembly evaluations were performed using `QUAST` [25] v5.0.2, and run with parameters recommended in `HiCanu`'s article [50]. `QUAST` aligns contigs to a reference genome, allowing to compute contiguity and completeness statistics that are corrected for misassemblies (NGA50 and Genome fraction metrics respectively in Table 3.1). Assemblies were all run using 8 threads on a Xeon 2.60 GHz CPU. For `rust-mdbg` assemblies, contigs shorter than 50 Kbp were filtered out similarly to [50]. We did not report the running time of the base-space conversion step and graph simplifications as they are under 15% of the running CPU time

*`https://obj.umiacs.umd.edu/marbl_publications/hicanu/index.html`

and run on a single thread, taking no more memory than the final assembly size, which is also less memory than the mdBG.

Table 3.1 (leftmost) shows assembly statistics for *D. melanogaster* HiFi reads. Our software `rust-mdbg` uses $\sim$ 33x less wall-clock time and 8x less RAM than all other assemblers. In terms of assembly quality, all tools yielded high-quality results. `HiCanu` had 66% higher NGA50 statistics than `rust-mdbg`, at the cost of making more misassemblies, 385x longer runtime and 8x higher memory usage. `rust-mdbg` reported the lowest Genome fraction statistics, likely due in part to an aggressive tip-clipping graph simplification strategy, also removing true genomic sequences.

Table 3.1 (rightmost) shows assembly statistics for Human HiFi (HG002) reads. `rust-mdbg` performed assembly 81x faster with 18x less memory usage than `Peregrine`, at the cost of a 22% lower contiguity and 1.5% lower completeness. Compared to `hifiasm`, `rust-mdbg` performed 338x faster with 19x lower memory, resulting in a less contiguous assembly (NG50 of 16.1 Mbp vs 88.0 Mbp for `hifiasm`) and 1.3% higher completeness.

Remarkably, the initial unsimplified mdBG for the Human assembly only has around 12 million k-min-mers (seen at least twice in the reads, out of 40 million seen in total) and 24 million edges, which should be compared to the 2.2 Gbp length of the (homopolymer-compressed) assembly and the 100 GB total length of input reads in uncompressed FASTA format. This highlights that the mdBG allows very efficient storage and simplification operations over the initial assembly graph in minimizer-space.

3.3.2 Minimizer-space POA enables correction of reads with higher sequencing error rates

We introduce minimizer-space partial order alignment (POA) to tackle sequencing errors. To determine the efficacy of minimizer-space POA and the limits of minimizer-space de Bruijn graph assembly with higher read error rates, we performed experiments on a smaller dataset. In a nutshell, we simulated reads for a single *Drosophila* chromosome at various error rates, and performed mdBG assembly with and without POA.

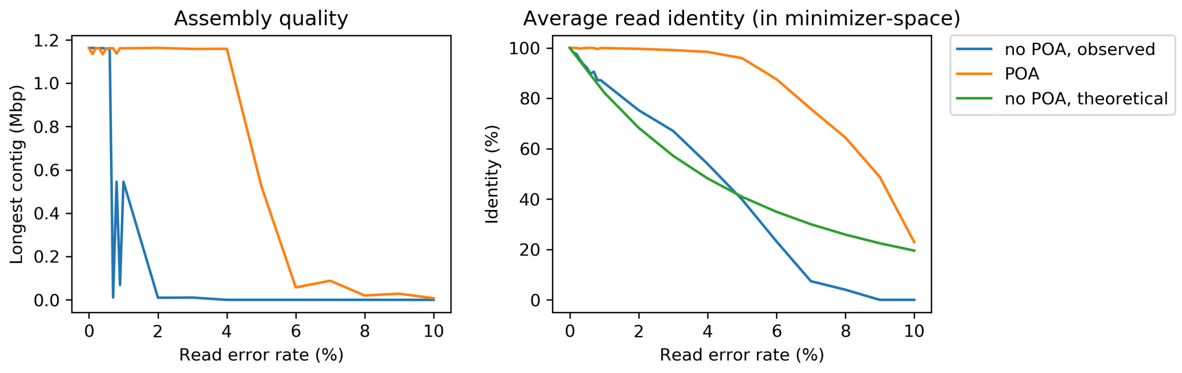


Figure 3-5: **Effect of our minimizer-space POA correction on mdBG assembly**. Reads from *D. melanogaster* chromosome 4 were simulated with base error rates ranging from 0%, 1%, . . ., up to 10%. Assemblies were run with and without minimizer-space POA correction. Left panel depicts the length of the longest contig for each assembly (uncorrected in blue, minimizer-space POA-corrected in orange). Right panel depicts the average read identity to the reference, computed in minimizer-space, for raw reads (observed in blue, and predicted by Equation 3.1 in green), and reads corrected by POA in minimizer-space (in orange).

Figure 3-5 (left) shows that the original implementation without POA is only able to reconstruct the complete chromosome into a single contig up to error rates of 1%, after which the chromosome is assembled into ≥ 2 contigs. With POA, an accurate reconstruction as a single contig is obtained with error rates up to 4%. We further verified that, up to 3% error rate, the reconstructed contig corresponds structurally exactly to the reference, apart from the base errors in the reads. At 4% error rate, a single uncorrected indel in minimizer-space introduces a $\sim$ 1 Kbp artificial insertion in the assembly.

Figure 3-5 (right) indicates that the minimizer-space identity of raw reads linearly decreases with increasing error rate. With POA, near-perfect correction can be achieved up to $\sim$ 4% error rate, with a sharp decrease at $>$ 5% error rates but still with an improvement in identity over uncorrected reads.

This highlights the importance of accurate POA correction: To put these results in perspective, mdBGs appear to be suitable to HiFi-grade data ($<$ 1% error rates) without POA and our POA implementation is almost, but not quite yet, able to cope with the error rate of ONT data (5%).

With POA, the runtime of our implementation was around 45 seconds and 0.4 GB of memory, compared to under 1 second and < 30 MB of memory without POA. Note that we did not use an optimized POA implementation; thus, we anticipate that further engineering efforts would significantly lower the runtime and possibly also improve the quality of correction.

3.3.3 Pangenome mdBG of a collection of 661K bacterial genomes allows efficient large-scale search of AMR genes

We applied mdBG to represent a recent collection of 661,405 assembled bacterial genomes [6]. To the best of our knowledge, this is the first de Bruijn graph construction of such a large collection of bacterial genomes. Previously only approximate sketches were created for this collection: a COBS index [5], allowing probabilistic membership queries of short k-mers ($k = 31$) [6], and sequence signatures (MinHash) using sourmash [57] and pp-sketch [34], none of which are graph representations.

The mdBG construction with parameters $k = 10$, $l = 12$, and $\delta = 0.001$ took 3h50m wall-clock running time using 8 threads, totaling 8 hours CPU time (largely IO-bound). The memory consumption was 58 GB and the total disk usage was under 150 GB. Increasing δ to 0.01 yields a finer-resolution mdBG but increases the wall-clock running time to 13h30m, the memory usage to 481 GB, and the disk usage to 200 GB.

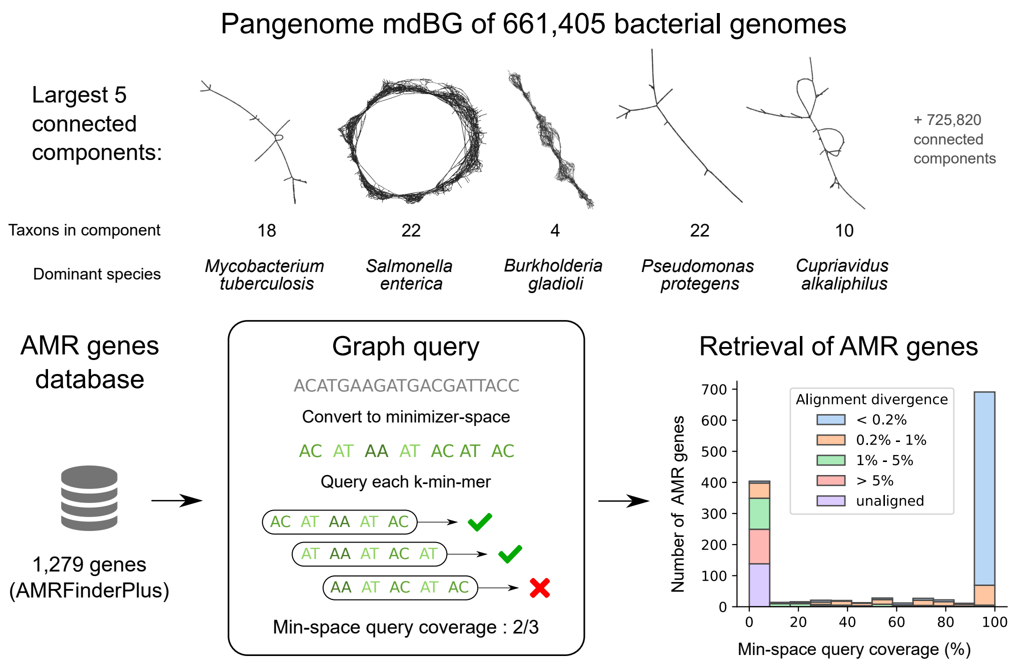
To compare the performance of mdBG with existing state-of-the-art tools for building de Bruijn graphs, we executed KMC3 [30] to count 63-mers and Cuttlefish [29] to construct a de Bruijn graph from the counted k-mers. KMC3 took 22 wall-clock hours and 191 GB memory using 8 threads, 2 TB of temporary disk usage, and 758 GB of output (56 billion distinct k-mers). Cuttlefish [29] did not terminate within three weeks of execution time. Hence, constructing the mdBG is at least two orders of magnitude more efficient in running time, and one order of magnitude in disk usage and memory usage.

Figure 3-6 shows the largest 5 connected components of the $\delta = 0.001$ bacterial pangenome mdBG. As expected, several similar species are represented within each connected component. The entire graph consists of 16 million nodes and 45 million edges (5.3 GB compressed GFA), i.e. too large to be rendered, yet much smaller than the original sequences (1.4 TB lz4-compressed).

To illustrate a possible application of this pangenome graph, we performed queries for the presence of AMR genes in the $\delta = 0.01$ mdBG. We retrieved 1,502 targets from the NCBI AMRFinderPlus 'core' database (the whole `amr_targets.fa` file as of May 2021) and converted each gene into minimizer-space, using parameters $k = 10$, $l = 12$, $\delta = 0.01$. Of these, 1,279 genes were long enough to have at least one k-min-mer (on average 10 k-min-mers per gene). Querying those k-min-mers on the mdBG, we successfully retrieved on average 61.2% of the k-min-mers per gene; however, the retrieval distribution is bimodal: 53% of the genes have $\geq 99\%$ k-min-mers found, and 31% of the genes have $\leq 10\%$ k-min-mers found.

Further investigation of the genes missing from the mdBG was done by aligning the 661k genomes collection to the genes (in base-space) using `minimap2` (7 hours running time over 8 cores). We found that a significant portion of genes (141, 11%) could not be aligned to the collection. Also, k-min-mers of genes with aligned sequence divergence of 1% or more (267, 20%) did not match k-min-mers from the collection, and therefore had zero minimizer-space query coverage. Finally, although we performed sequence queries on a text representation of the pangenome graph, in principle the graph could be indexed in memory to enable instantaneous queries at the expense of higher memory usage.

This experiment illustrates the ability of mdBG to construct pangenomes larger than supported by any other method, and those pangenomes record biologically useful information such as AMR genes. Long sequences such as genes (containing at least 1 k-min-mer) can be quickly searched using k-min-mers as a proxy. There is nevertheless a trade-off of minimizer-space analysis that is akin to classical k-mer analysis: Graph construction and queries are extremely efficient however they do not capture sequence similarity below a certain identity threshold (in this experiment, around 99%). Yet, the ability of the mdBG to quickly enumerate which bacterial genomes possess any AMR gene with high similarity could provide a significant boost to AMR studies.



	Graph construction (661k genomes, δ= 0.01)	Graph query (1,279 genes)
Performance	13h30 wall-clock hours, 481 GB memory	12 wall-clock minutes, <1 GB memory

Figure 3-6: **Pangenome mdBG of 661k bacterial genomes and retrieval of anti-microbial resistance genes**. Top panel: first five connected components of the pangenome graph are displayed (using Gephi software). Each node is a k-min-mer, and edges are exact overlaps of $k-1$ minimizers between k-min-mers. Middle panel: A collection of anti-microbial resistance gene targets was converted into minimizer-space, then each k-min-mer is queried in the 661k bacterial pangenome graph yielding a bimodal distribution of minimizer-space gene retrieval. The histogram is annotated by the minimal sequence diverge of each gene as aligned by `minimap2` to the pangenome over 90% of its length. Bottom panel: runtime and memory usage for each step. Note that the graph need only be constructed once in a preprocessing step.

3.3.4 Efficient assembly of real HiFi metagenomes using mdBG

We performed assembly of two real HiFi metagenome datasets (mock communities Zymo D6331 and ATCC MSA-1003, accessions SRX9569057 and SRX8173258). `Rust-mdbg` was run with the same parameters as in the human genome assembly for the ATCC dataset, and with slightly tuned parameters for the Zymo dataset (see Appendix A.4).

Table 3.2 shows the results of `rust-mdbg` assemblies in comparison to `hifiasm-meta`, a metagenome-specific flavor of `hifiasm`. In a nutshell, `rust-mdbg` achieves roughly two orders of magnitude faster and more memory-efficient assemblies, while retaining similar completeness of the assembled genomes. Although `rust-mdbg` metagenome assemblies are consistently more fragmented than `hifiasm-meta` assemblies, the ability of `rust-mdbg` to very quickly assemble a metagenome enables instant quality control and preliminary exploration of gene content of microbiomes at a fraction of the computing costs of current tools.

Zymo D6331

Species	**Abundance**	`hifiasm`	`rust-mdbg`
A. muciniphila	1.36%	100.000%	100.000%
B. fragilis	13.13%	99.994%	99.997%
B. adolescentis	1.34%	100.000%	99.730%
C. albican	1.61%	67.832%	39.821%
C. difficile	1.83%	99.996%	99.978%
C. perfringens	0.00%	0.005%	0.005%
E. faecalis	0.00%	0.006%	0.006%
E. coli B1109	8.44%	100.000%	97.918%
E. coli b2207	8.32%	100.000%	98.663%
E. coli B3008	8.25%	100.000%	99.558%
E. coli B766	7.83%	96.913%	96.270%
E. coli JM109	8.37%	100.000%	97.852%
F. prausnitzii	14.39%	100.000%	100.000%
F. nucleatum	3.78%	100.000%	99.960%
L. fermentum	0.86%	100.000%	100.000%
M. smithii	0.04%	99.840%	87.175%
P. corporis	5.37%	99.561%	99.561%
R. hominis	3.88%	100.000%	100.000%
S. cerevisiae	0.18%	69.522%	39.556%
S. enterica	0.02%	6.232%	4.619%
V. rogosae	11.02%	100.00%	100.000%
Running time		34h29m	**55s**
Memory usage		83 GB	**0.9 GB**

ATCC MSA-1003

Species	**Abundance**	`hifiasm`	`rust-mdbg`
A. baumannii	0.18%	99.839%	99.955%
B. pacificus	1.80%	100.000%	99.998%
B. vulgatus	0.02%	81.846%	70.895%
B. adolescentis	0.02%	5.238%	0.644%
C. beijerinckii	1.80%	99.993%	99.993%
C. acnes	0.18%	100.000%	100.000%
D. radiodurans	0.02%	82.499%	53.659%
E. faecalis	0.02%	54.979%	21.048%
E. coli	18.0%	100.000%	100.000%
H. pylori	0.18%	100.000%	100.000%
L. gasseri	0.18%	97.779%	98.142%
N. meningitidis	0.18%	98.593%	99.030%
P. gingivalis	18.0%	91.740%	99.938%
P. aeruginosa	1.80%	99.706%	99.726%
R. sphaeroides	18.0%	99.748%	100.000%
S. odontolytica	0.02%	8.179%	1.046%
S. aureus	1.80%	100.000%	100.000%
S. epidermidis	18.0%	100.000%	100.000%
S. agalactiae	1.80%	99.496%	99.976%
S. mutans	18.0%	99.995%	100.000%
Running time		59h16m	**3m51s**
Memory usage		313 GB	**1.3 GB**

Table 3.2: **Metagenome assembly statistics of the Zymo D6331 dataset (left) and the ATCC MSA-1003 dataset (right) using `hifiasm-meta` and `rust-mdbg`.** The **Abundance** column shows the relative abundance of the species in the sample. The two rightmost columns show the species completeness of the assemblies as reported by `metaQUAST`. Table cells below 10% completeness are colored in red, below 98% in orange, and above in green.

3.4 Discussion and future work

Three areas we hope to tackle in our assembly implementation are: 1) its reliance on setting adequate assembly parameters, 2) lack of base-level polishing, and 3) haplotype separation. Regarding 1), we are experimenting with automatic selection of parameters ℓ, k and δ. A heuristic formula is presented along with its implementation and results in the GitHub repository of `rust-mdbg`; however, it leads to lower-quality results (e.g. 1 Mbp N50 for the HG002 assembly versus 14 Mbp in Table 3.1). We also provide a preliminary multi-k assembly script inspired by IDBA [55]. While automatically setting mdBG parameters is fundamentally a more complex task than just determining a single parameter (k) in classical de Bruijn graphs, we anticipate that similar techniques to `KmerGenie` [13] could be applicable, where optimal values of (ℓ, k, δ) would be found as a function of the k-min-mer abundance histogram.

Regarding directions 2) and 3), polishing could be performed as an additional step, by feeding the reads and the unpolished assembly to a base-space polishing tool such as `racon` [63]. Haplotype separation might prove more difficult to incorporate in mdBGs: Unlike HiFi assemblers which use overlap graphs with near-perfect overlaps, minimizer-space de Bruijn graphs cannot differentiate between exact and inexact overlaps in bases that are not captured by a minimizer. However, an immediate workaround is to perform haplotype phasing on resulting contigs, using tools such as `HapCut2` [21] or `HapTree-X` [4].

We anticipate that k-min-mers could become a drop-in replacement for ubiquitously-adopted k-mers for the comparison and indexing of long, highly similar sequences, e.g. in genome assembly, transcriptome assembly, and taxonomic profiling.

Chapter 4

Conclusion

In Chapter 2, we presented a novel randomized parallel algorithm, `PASHA`, to compute a small set of k-mers which together hit every sequence of length L. It is based on two algorithmic innovations: (i) improved calculation of hitting numbers through paralleization and memory reduction; and (ii) randomized parallel selection of additional k-mers to remove. We demonstrated the scalability of `PASHA` to larger values of k up to 16. Notably, the universal hitting sets need to be computed only once, and can then be used in many sequence analysis applications. We expect our algorithms to be an essential part of the sequence analysis toolkit.

In Chapter 3, we presented a data structure which we call a **minimizer-space de Bruijn graph (mdBG)**, where, instead of single nucleotides as tokens of the de Bruijn graph, we use short sequences of nucleotides known as minimizers, which allow for an even more compact representation of the genome in what we call *minimizer space*. Minimizer-space de Bruijn graphs store only a small fraction of the nucleotides from the input data while preserving the overall graph structure, enabling them to be orders of magnitude more efficient than classical de Bruijn graphs. By doing so, we can reconstruct whole genomes from accurate long-read data in minutes – about a hundred times faster than state-of-the-art approaches – on a personal computer, while using significantly less memory and achieving similar accuracy.

To enable assembly of reads with up to a 4% error rate (e.g. from emerging Oxford

Nanopore data, which offers high sequencing throughput, low cost and ultra-long read lengths), we newly correct for read errors by performing minimizer-space partial order alignment (POA), in which sequencing errors in a query read are corrected by aligning other reads from the same genomic region to the query in minimizer space.

We also showed that we can build very large minimizer-space de Bruijn graphs that can be queried for biologically useful questions by constructing a graphical pangenome of a large and diverse collection of 661,405 bacterial genomes. This collection of several terabytes has never before been represented as a pangenome graph (a graph that represents multiple genomes simultaneously). Such a task is computationally nearly impossible using state-of-the-art methods, which would take weeks and terabytes of RAM to complete. We showed that our method completes the construction in roughly 3 hours with low memory usage, and the connected components in the mdBG distinguish species, allowing us to quickly search for antimicrobial resistance genes inside the entire pangenome.

Given the rise of next-generation sequencing technologies and faster and less expensive genome assembly, we expect our advances to be essential to the convergence among next-generation sequencing (NGS), cloud computing, and precision and personalized medicine, and beneficial in creating the infrastructure necessary to formulate and test disease mechanisms and develop new treatments at scale.

Appendix A

Supplementary Information and Proofs

A.1 Emulating the greedy `DOCKS` algorithm

The greedy Set Cover algorithm was developed independently by Johnson and Lovász for unweighted vertices [27, 42]. Lovász [42] proved:

Theorem A.1.1. *The greedy algorithm for Set Cover outputs cover R with $|R| \leq (1 + \log T_{max})|OPT|$, where T_{max} is the maximum cardinality set.*

We adapt a definition for an algorithm emulating the greedy algorithm for the Set Cover problem to the second phase of `DOCKS` [3]. We say that an algorithm for the second phase of `DOCKS` **α-emulates** the greedy algorithm if it outputs a set of vertices serially, during which it selects vertex set A such that

$$\frac{|A|}{|P_A|} \leq \frac{\alpha}{T_{max}},$$

where P_A is the set of ℓ-long paths covered by A. Using this definition, we come up with a near-optimal approximation by the following theorem:

Theorem A.1.2. *An algorithm for the second phase of `DOCKS` that α-emulates the greedy algorithm produces cover $R \subseteq V$ with $|R| \leq \alpha(1+\log T_{max})|OPT|$, where OPT is the optimal cover.*

Proof. We define the *cost* of covering path p as $\mathcal{C}(p) = \frac{|S|}{|P_S|}$, where S is the set of vertices selected in the selection step in which p was covered, and P_S the set of ℓ-long paths covered by S. Then, $\sum_{p \in P_S} \mathcal{C}(p) = |S|$.

Let P_ℓ be set of all ℓ-long paths in G. A **fractional cover** of graph $G = (V, E)$ is function $\mathcal{F} : V \to \{0, 1\}$ s.t. for all $p \in P_\ell$, $\sum_{v \in p} \mathcal{F}(v) \geq 1$. The optimal cover $\mathcal{F}_{OPT}$ has minimum $\sum_{v \in V} \mathcal{F}_{OPT}(v)$.

Let $\mathcal{F}$ be such an optimal fractional cover. The size of the cover produced is

$$|R| = \sum_{p \in P_\ell} \mathcal{C}(p) \leq \sum_{v \in V} \Big(\mathcal{F}(v) \sum_{p \in P_v} \mathcal{C}(p) \Big)$$

where P_v is the set of all ℓ-long paths through vertex v.

Lemma A.1.3. *There are at most $\frac{\alpha}{k}$ paths $p \in P_v$ such that $\mathcal{C}(p) \geq k$ for any v, k.*

Proof. Assume the contrary: Before such path p is covered, $T(v, \ell) > \frac{\alpha}{k}$. Thus,

$$\frac{|S|}{|P_S|} \geq k > \alpha / T(v, \ell) \geq \alpha / T_{max},$$

contradicting the definition. □

Suppose we rank the $T(v, \ell)$ paths $p \in P_v$ by decreasing order of $\mathcal{C}(p)$. From the above remark, if the ith path has cost k, then $i \leq \alpha/k$. Then, we can write

$$\sum_{p \in P_v} \mathcal{C}(p) \leq \sum_{i=1}^{T(v,\ell)} \alpha/i \leq \alpha \sum_{i=1}^{T(v,\ell)} 1/i \leq \alpha(1 + \log T(v, \ell)) \leq \alpha(1 + \log T_{max})$$

Then,

$$\sum_{p \in P_\ell} \mathcal{C}(p) \leq \sum_{v \in V} \mathcal{F}(v) \alpha (1 + \log T_{max})$$

and finally

$$|R| \leq \alpha(1 + \log T_{max}) |OPT|.$$

□

In `PASHA`, we ensure that in step t, the sum of vertex hitting numbers of selected vertex set V_t is at least $|V_t|(1+\varepsilon)^t(1-4\delta-2\varepsilon)$. We now show that this is satisfied with high probability in each step.

Theorem A.1.4. *With probability at least 1/2, the sum of vertex hitting numbers of selected vertex set V_t at step t is at least $|V_t|(1+\varepsilon)^t(1-4\delta-2\varepsilon)$.*

Proof. For any vertex v in selected vertex set V_t at step t, let X_v be an indicator variable for the random event that vertex v is picked, and $f(X)=\sum_{v\in V_t} X_v$.

Note that $\mathrm{Var}[f(X)] \leq |V_t|\cdot\delta/\ell$, and $|V_t|\geq \ell/\delta^3$, since we are given that no vertex covers a δ^3 fraction of the ℓ-long paths covered by the vertices in V_t. By Chebyshev's inequality, for any $k\geq 0$,

$$\Pr[|f(X)-\mathrm{E}[f(X)]|\geq k(|V_t|\cdot\delta/\ell)]\leq\frac{1}{k^2}$$

and with probability 3/4,

$$(f(X)-\mathrm{E}[f(X)])^2\leq 4|V_t|^2\cdot\frac{\delta^4}{\ell^2}$$

and

$$|f(X)-\mathrm{E}[f(X)]|\leq 2|V_t|\cdot\frac{\delta^2}{\ell}.$$

Let P_{V_t} denote the set of ℓ-long paths covered by vertex set V_t. Then,

$$|P_{V_t}|\geq\sum_{u\in V_t}T(u,\ell)X_u-\sum_{p\in P_{V_t}}\sum_{u,v\in p}X_uX_v$$

We know that $\sum_{u\in V_t}T(u,\ell)X_u\geq|V_t|(1+\varepsilon)^{t-1}$, which is bounded below by $((\delta-2\delta^2)\cdot|V_t|(1+\varepsilon)^{t-1})/\ell$. Let $g(X)=\sum_{p\in P_{V_t}}\sum_{u,v\in p}X_uX_v$. Then,

$$\mathrm{E}[g(X)]=\sum_{p\in P_{V_t}}\mathrm{E}[\sum_{u,v\in p}X_uX_v]=\sum_{p\in P_{V_t}}\binom{l}{2}(\delta/\ell)^2=\sum_{p\in P_{V_t}}\frac{(\ell-1)\cdot\delta^2}{2\ell}\leq\sum_{p\in P_{V_t}}\frac{\delta^2}{2}.$$

Hence, with probability at least 3/4,

$$g(X) \leq 4\mathrm{E}[g(X)] \leq 2\delta^2 \cdot |V_t|(1+\varepsilon)^t$$

Both events hold with probability at least 1/2, and the sum of vertex hitting numbers is at least

$$\begin{aligned}((\delta - 2\delta^2) \cdot |V_t|(1+\varepsilon)^{t-1}) \cdot \ell - 2\delta^2 \cdot |V_t|(1+\varepsilon)^t &\geq |V_t|(1+\varepsilon)^{t-1}(\delta\ell - 2\delta^2\ell - 2\delta^2 - 2\delta^2\varepsilon) \\ &\geq |V_t|(1+\varepsilon)^t(\delta\ell - 2\delta^2\ell - 2\delta^2 - 2\delta^2\varepsilon)/(1+\varepsilon) \\ &\geq |V_t|(1+\varepsilon)^t(1 - 4\delta - 2\varepsilon).\end{aligned}$$

□

A.2 PASHA runtime analysis

Here, we show the number of the selection steps and the average-time asymptotic complexity of PASHA.

Lemma A.2.1. *The number of selection steps is* $O(\log|V|\log|P_\ell|/(\varepsilon\delta^3 m))$.

Proof. The number of steps is $O(\log|V|/\varepsilon)$, and within each step, there are $O(\log|P_S|/(\delta^3 m))$ selection steps (where P_S is the sum of vertex hitting numbers of the vertex set S for that step and m the number of threads used), since we are guaranteed to remove a δ^3 fraction of the paths during that step. Overall, there are $O(\log|V|\log|P_\ell|/(\varepsilon\delta^3 m))$ selection steps. □

Theorem A.2.2. *For* $\varepsilon < 1$*, there is an approximation algorithm for the second phase of DOCKS that runs in* $O((L^2 \cdot |\Sigma|^{k+1} \cdot \log^2(|\Sigma|^k))/(\varepsilon\delta^3 m))$ *average time, where* m *is the number of threads used, and produces a cover of size at most* $(1+\varepsilon)(1+\log T_{max})$ *times the optimal size.*

Proof. Follows immediately from Theorem A.1.2 and Lemma A.2.1. □

A.3 `gfatools` command lines

The following (relatively aggressive) GFA assembly graph simplifications rounds were performed for all mdBG assemblies, using `https://github.com/lh3/gfatools/`. Rounds are of two types: `-t x,y` removes tips having at most x segments and of maximal length y bp, and `-b z` removes bubbles of maximal radius z bp. In addition, `gfa_break_loops.py` is a custom script (available in the `rust-mdbg` GitHub repository) that removes self-loops in the assembly graph, as well as an arbitrary edge in $x \leftrightarrow y$ cycles.

```
gfatools asm -t 10,50000 -t 10,50000 -b 100000 -b 100000 -t 10,50000 \
             -b 100000 -b 100000 -b 100000 -t 10,50000 -b 100000 \
             -t 10,50000 -b 1000000 -t 10,150000 -b 1000000 -u > $base.tmp1.gfa
gfa_break_loops.py $base.tmp1.gfa > $base.tmp2.gfa
gfatools asm $base.tmp1.gfa  -t 10,50000 -b 100000 -t 10,100000  \
             -b 1000000 -t 10,150000 -b 1000000 -u > $base.tmp3.gfa
gfa_break_loops.py $base.tmp3.gfa > $base.tmp4.gfa
gfatools asm $base.tmp4.gfa  -t 10,50000 -b 100000 -t 10,100000 \
             -b 1000000 -t 10,200000 -b 1000000 -u > $base.msimpl.gfa
```

A.4 Genome assembly tools, versions and parameters

`HiCanu` (v2.1) was run with default parameters, `hifiasm` (commit `8cb131d`) with parameters `-l0 -f0`, and `Peregrine` (commit `008082a`) with command line: `8 8 8 8 8 8 8 8 8 -with-consensus -shimmer-r 3 -best_n_ovlp 8`. `rust-mdbg` was run with parameters $k = 35$, $\ell = 12$, and $\delta = 0.002$ for *D. melanogaster*, and $k = 21$, $\ell = 14$, $\delta = 0.003$ for HG002.

For metagenomes, `rust-mdbg` was run with parameters $k = 21$, $\ell = 14$, $\delta = 0.003$ for the ATCC MSA-1003 dataset (same parameters as the human dataset), and $k = 40$, $\ell = 12$, $\delta = 0.004$ for the Zymo D6331 dataset. `Hifiasm-meta` (commit `cda13b8`) was run with parameters `-S -lowq-10 50` for ATCC MSA-1003 and default for Zymo.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Serafim Batzoglou, David B Jaffe, Ken Stanley, Jonathan Butler, Sante Gnerre, Evan Mauceli, Bonnie Berger, Jill P Mesirov, and Eric S Lander. ARACHNE: a whole-genome shotgun assembler. *Genome Research*, 12(1):177–189, 2002.

[2] Bonnie Berger, Jian Peng, and Mona Singh. Computational solutions for omics data. *Nature Reviews Genetics*, 14(5):333, 2013.

[3] Bonnie Berger, John Rompel, and Peter W Shor. Efficient NC Algorithms for Set Cover with Applications to Learning and Geometry. *Journal of Computer and System Sciences*, 49(3):454–477, 1994.

[4] Emily Berger, Deniz Yorukoglu, Lillian Zhang, Sarah K Nyquist, Alex K Shalek, Manolis Kellis, Ibrahim Numanagić, and Bonnie Berger. Improved haplotype inference by exploiting long-range linking and allelic imbalance in RNA-seq datasets. *Nature Communications*, 11(1):1–9, 2020.

[5] Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. COBS: A compact bit-sliced signature index. In *26th International Conference on String Processing and Information Retrieval (SPIRE)*, LNCS, pages 285–303. Springer, October 2019. preprint arXiv:1905.09624.

[6] Grace A Blackwell, Martin Hunt, Kerri M Malone, Leandro Lima, Gal Horesh, Blaise TF Alako, Nicholas R Thomson, and Zamin Iqbal. Exploring bacterial diversity via a curated and searchable snapshot of archived DNA sequences. *bioRxiv*, 2021.

[7] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997.

[8] Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer, 1994.

[9] Brian Bushnell. BBMap: A fast, accurate, splice-aware aligner. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2014.

[10] Haoyu Cheng, Gregory T Concepcion, Xiaowen Feng, Haowen Zhang, and Heng Li. Haplotype-resolved de novo assembly with phased assembly graphs. *arXiv preprint arXiv:2008.01237*, 2020.

[11] Rayan Chikhi, Jan Holub, and Paul Medvedev. Data structures to represent sets of k-long DNA sequences. *arXiv preprint arXiv:1903.12312*, 2019.

[12] Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the representation of de Bruijn graphs. In *International Conference on Research in Computational Molecular Biology*, pages 35–55. Springer, 2014.

[13] Rayan Chikhi and Paul Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31–37, 2014.

[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] Chen-Shan Chin and Asif Khalak. Human genome assembly in 100 minutes. *bioRxiv*, page 705616, 2019.

[16] Süleyman Cenk Şahinalp and Uzi Vishkin. Symmetry breaking for suffix tree construction. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*, STOC '94, page 300–309, New York, NY, USA, 1994. Association for Computing Machinery.

[17] Dan DeBlasio, Fiyinfoluwa Gbosibo, Carl Kingsford, and Guillaume Marçais. Practical universal k-mer sets for minimizer schemes. In *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, pages 167–176. ACM, 2019.

[18] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

[19] Robert Edgar. Syncmers are more sensitive than minimizers for selecting conserved k-mers in biological sequences. *PeerJ*, 9:e10805, 2021.

[20] Robert C Edgar, Jeff Taylor, Tomer Altman, Pierre Barbera, Dmitry Meleshko, Victor Lin, Dan Lohr, Gherman Novakovsky, Basem Al-Shayeb, Jillian F Banfield, et al. Petabase-scale sequence alignment catalyses viral discovery. *bioRxiv*, 2020.

[21] Peter Edge, Vineet Bafna, and Vikas Bansal. HapCUT2: Robust and accurate haplotype assembly for diverse sequencing technologies. *Genome Research*, 27(5):801–812, 2017.

[22] Barış Ekim, Bonnie Berger, and Rayan Chikhi. Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer. *Cell Systems*, 12(10):958–968, 2021.

[23] Barış Ekim, Bonnie Berger, and Yaron Orenstein. A randomized parallel algorithm for efficiently finding near-optimal universal hitting sets. In *International Conference on Research in Computational Molecular Biology*, pages 37–53. Springer, 2020.

[24] MJ Ellington, O Ekelund, Frank Møller Aarestrup, R Canton, M Doumith, Christian Giske, Hajo Grundman, Henrik Hasman, MTG Holden, Katie L Hopkins, et al. The role of whole genome sequencing in antimicrobial susceptibility testing of bacteria: report from the eucast subcommittee. *Clinical microbiology and infection*, 23(1):2–22, 2017.

[25] Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. QUAST: Quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 2013.

[26] Chirag Jain, Arang Rhie, Haowen Zhang, Claudia Chu, Sergey Koren, and Adam Phillippy. Weighted minimizer sampling improves long read mapping. *bioRxiv*, 2020.

[27] David S Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9(3):256–278, 1974.

[28] Jolanta Kawulok and Sebastian Deorowicz. CoMeta: Classification of metagenomes using k-mers. *PloS One*, 10(4):e0121453, 2015.

[29] Jamshed Khan and Rob Patro. Cuttlefish: Fast, parallel, and low-memory compaction of de Bruijn graphs from large-scale genome collections. *bioRxiv*, 2020.

[30] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. KMC 3: counting and manipulating k-mer statistics. *Bioinformatics*, 33(17):2759–2761, 2017.

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

[32] Alexander Lachmann, Denis Torre, Alexandra B Keenan, Kathleen M Jagodnik, Hoyjin J Lee, Lily Wang, Moshe C Silverstein, and Avi Ma'ayan. Massive mining of publicly available RNA-seq data from human and mouse. *Nature Communications*, 9(1):1–10, 2018.

[33] Christopher Lee, Catherine Grasso, and Mark F Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002.

[34] John A Lees, Simon R Harris, Gerry Tonkin-Hill, Rebecca A Gladstone, Stephanie W Lo, Jeffrey N Weiser, Jukka Corander, Stephen D Bentley, and Nicholas J Croucher. Fast and flexible bacterial genomic epidemiology with PopPUNK. *Genome Research*, 29(2):304–316, 2019.

[35] Heng Li. Minimap and miniasm: Fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 2016.

[36] Heng Li. Minimap2: Pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 2018.

[37] Yang Li et al. MSPKmerCounter: a fast and memory efficient approach for k-mer counting. *arXiv preprint arXiv:1505.06550*, 2015.

[38] Yu Lin, Jeffrey Yuan, Mikhail Kolmogorov, Max W Shen, Mark Chaisson, and Pavel A Pevzner. Assembly of long error-prone reads using de Bruijn graphs. *Proceedings of the National Academy of Sciences*, 113(52):E8396–E8405, 2016.

[39] Glennis A Logsdon, Mitchell R Vollger, and Evan E Eichler. Long-read human genome sequencing and its applications. *Nature Reviews Genetics*, pages 1–18, 2020.

[40] Po-Ru Loh, Michael Baym, and Bonnie Berger. Compressive genomics. *Nature Biotechnology*, 30(7):627–630, 2012.

[41] Nicholas J Loman, Joshua Quick, and Jared T Simpson. A complete bacterial genome assembled de novo using only nanopore sequencing data. *Nature Methods*, 12(8):733–735, 2015.

[42] László Lovász. On the ratio of optimal integral and fractional covers. *Discrete Mathematics*, 13(4):383–390, 1975.

[43] Jennifer Lu and Steven Salzberg. Ultrafast and accurate 16S microbial community analysis using Kraken 2. *bioRxiv*, 2020.

[44] Guillaume Marçais, Dan DeBlasio, and Carl Kingsford. Asymptotically optimal minimizers schemes. *Bioinformatics*, 34(13):i13–i22, 2018.

[45] Guillaume Marçais, Dan DeBlasio, Prashant Pandey, and Carl Kingsford. Locality-sensitive hashing for the edit distance. *Bioinformatics*, 35(14):i127–i135, 2019.

[46] Guillaume Marçais, David Pellow, Daniel Bork, Yaron Orenstein, Ron Shamir, and Carl Kingsford. Improving the performance of minimizers and winnowing schemes. *Bioinformatics*, 33(14):i110–i117, 2017.

[47] Guillaume Marçais, Brad Solomon, Rob Patro, and Carl Kingsford. Sketching and sublinear data structures in genomics. *Annual Review of Biomedical Data Science*, 2019.

[48] Johannes Mykkeltveit. A Proof of Golomb's Conjecture for the de Bruijn Graph. *Journal of Combinatorial Theory*, 13(1):40–45, 1972.

[49] Sumaiya Nazeen, Yun William Yu, and Bonnie Berger. Carnelian uncovers hidden functional patterns across diverse study populations from whole metagenome sequencing reads. *Genome Biology*, 21(1):1–18, 2020.

[50] Sergey Nurk, Brian P Walenz, Arang Rhie, Mitchell R Vollger, Glennis A Logsdon, Robert Grothe, Karen H Miga, Evan E Eichler, Adam M Phillippy, and Sergey Koren. HiCanu: Accurate assembly of segmental duplications, satellites, and allelic variants from high-fidelity long reads. *bioRxiv*, 2020.

[51] Brian D Ondov, Todd J Treangen, Páll Melsted, Adam B Mallonee, Nicholas H Bergman, Sergey Koren, and Adam M Phillippy. Mash: Fast genome and metagenome distance estimation using MinHash. *Genome Biology*, 17(1):132, 2016.

[52] Yaron Orenstein, David Pellow, Guillaume Marçais, Ron Shamir, and Carl Kingsford. Compact universal k-mer hitting sets. In *International Workshop on Algorithms in Bioinformatics*, pages 257–268. Springer, 2016.

[53] Yaron Orenstein, David Pellow, Guillaume Marçais, Ron Shamir, and Carl Kingsford. Designing small universal k-mer hitting sets for improved analysis of high-throughput sequencing. *PLoS Computational Biology*, 13(10):e1005777, 2017.

[54] Marie Paindavoine and Bastien Vialla. Minimizing the number of bootstrappings in fully homomorphic encryption. In *Selected Areas in Cryptography – SAC 2015*, pages 25–43. Springer International Publishing, 2016.

[55] Yu Peng, Henry CM Leung, Siu-Ming Yiu, and Francis YL Chin. IDBA: A practical iterative de Bruijn graph de novo assembler. In *Annual International Conference on Research in Computational Molecular Biology*, pages 426–440. Springer, 2010.

[56] Paul A Pevzner, Haixu Tang, and Glenn Tesler. De novo repeat classification and fragment assembly. *Genome Research*, 14(9):1786–1796, 2004.

[57] N Tessa Pierce, Luiz Irber, Taylor Reiter, Phillip Brooks, and C Titus Brown. Large-scale sequence comparisons with sourmash. *F1000Research*, 8, 2019.

[58] Mikko Rautiainen and Tobias Marschall. MBG: Minimizer-based sparse de Bruijn graph construction. *bioRxiv*, 2020.

[59] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

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

[61] Kishwar Shafin, Trevor Pesout, Ryan Lorig-Roach, Marina Haukness, Hugh E Olsen, Colleen Bosworth, Joel Armstrong, Kristof Tigyi, Nicholas Maurer, Sergey Koren, et al. Nanopore sequencing and the Shasta toolkit enable efficient de novo assembly of eleven human genomes. *Nature Biotechnology*, pages 1–10, 2020.

[62] Ariya Shajii, Ibrahim Numanagić, Alexander T Leighton, Haley Greenyer, Saman Amarasinghe, and Bonnie Berger. A python-based programming language for high-performance computational genomics. *Nature Biotechnology*, pages 1–2, 2021.

[63] Robert Vaser, Ivan Sović, Niranjan Nagarajan, and Mile Šikić. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome Research*, 27(5):737–746, 2017.

[64] Aaron M Wenger, Paul Peluso, William J Rowell, Pi-Chuan Chang, Richard J Hall, Gregory T Concepcion, Jana Ebler, Arkarachai Fungtammasan, Alexey Kolesnikov, Nathan D Olson, et al. Accurate circular consensus long-read sequencing improves variant detection and assembly of a human genome. *Nature Biotechnology*, 37(10):1155–1162, 2019.

[65] Chengxi Ye, Zhanshan Sam Ma, Charles H Cannon, Mihai Pop, and W Yu Douglas. Exploiting sparseness in de novo genome assembly. *BMC Bioinformatics*, 13(6):S1, 2012.

[66] Deniz Yorukoglu, Yun William Yu, Jian Peng, and Bonnie Berger. Compressive mapping for next-generation sequencing. *Nature Biotechnology*, 34(4):374–376, 2016.