Streaming Methods for Assembly Graph Analysis

By

CAMILLE SCOTT

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

(C. Titus Brown), Chair

(Rob Patro)

(Fereydoun Hormozdiari)

Committee in Charge

2022

Streaming Methods for Assembly Graph Analysis

Copyright © 2022

by

Camille Scott

"*Someone once told me that time was a predator that stalked us all our lives. But I rather believe that time is a companion who goes with us on the journey and reminds us to cherish every moment, because they'll never come again. What we leave behind is not as important as how we lived… after all, Number One, we're only mortal.*" (Captain Jean-Luc Picard, USS Enterprise (NCC-1701-D), United Federation of Planets)"

Acknowledgements

The process of my PhD and the production of this dissertation was a long and difficult journey. During it all, I transitioned genders, moved across the country and changed institutions from Michigan State University to the University of California Davis, met my partner and eventual wife, struggled through mental health problems, adopted a family of pets and found a network of close friends, and generally, built a life. A PhD is necessarily an effort supported by a whole community of people, and I have been privileged to find a community that is nothing short of phenomenal.

My wife Clarissa, who has cherished, supported, and pushed me through thick and thin since we met here in Davis.

My advisor Titus, who has guided and sometimes dragged me through my degree. If he has a defining attribute, it might be his extraordinarily deep well of patience and understanding; if not that, it is his firm dedication and commitment to supporting and encouraging his mentees.

My mom, dad, and sister in Michigan, who have shaped, encouraged, and inspired me in all my endeavours.

Russell, who has served as a mentor, inspiration, sometimes therapist, and most importantly, a great friend. Without him, I would never have finished this program.

Paula, Megan, and Joe, all of whom provided encouragement when I was low.

The members of the DIB Lab: Tessa, who has contibuted tons to `dammit`, and Luiz, who helped build out and explore `draff`; and of course, the many other lab members who have provided thoughtful feedback and discussion over the years.

And Star Trek, for instilling in me, from a young age, the idea that mashing together science and computers can cause all manner of fantastical phenomena. Make it so!

Camille Scott
March 2022
COMPUTER SCIENCE

Abstract

The advent of high-throughput sequencing has radically altered the theory and practice of biology. Massive volumes of sequence data have necessitated commensurate advances in computational approaches to biological analysis; methods for the assembly of shotgun sequencing data into complete genomes, transcriptomes, and metagenomes have been particularly foundational in enabling downstream study. More recently, sketching methods have allowed classification and comparison to scale to hundreds of thousands of samples.

This dissertation extends that work by exploring streaming implementations of several core approaches. First, we introduce a method for single-pass construction of the compact de Bruijn graph, with support for novel methods of dynamic assembly graph analysis. Next, we explore the saturation behavior of streaming sequence sketches, and introduce a novel sketch we call `draff` which uses Universal k-mer Hitting Sets to represent the shape of assembly graphs. Finally, we introduce several pieces of research software for sequence comparison, transcript annotation, and phylogenetic tree building, which showcase accessible, open access design philosophies. We also introduce the `goetia` library and toolset for efficient de Bruijn graph algorithms in C++ and Python.

Contents

List of Figures

Introduction

Motivation

Large-scale genome sequencing is a paradigm-shifting technology for biology of the 21st century. Since the sequencing of the human genome was declared complete in April of 2003, genomics has been a core part of the public conscience and rapidly grown as a tool for biological research. What started as a small-scale molecular biology method for understanding individual genes grew into projects mapping out the genomes of most major model organisms in great functional detail, sequencing entire populations of microbes from the environment, mapping hundreds of thousands of individual human genomes, and producing draft sequences for hundreds of thousands of species across the entire tree of life; as of 2021, the National Center for Biotechnology Information (NCBI) stored over 36 petabytes of raw sequencing information in its Sequence Read Archive (SRA), with many times that volume not submitted.

Background

Streaming Algorithms

Streaming, or online, algorithms are those that process data in the *stream model*: a stream of data is fed to the algorithm, one observation at a time in the order of arrival, and each item is processed at most once. Alternatively, we can say that a streaming algorithm makes at most one pass over the input data. The stream may either be finite or infinite; when we conceptualize by number of passes, we generally are considering finite streams. A closely related group is the semi-streaming algorithm, which are those that make *few* passes over the data, typically two or less. These algorithms are of course concerned with finite streams of data.

Streaming approaches find the most use when data is too large to fit in main memory; for example, when working with massive network traffic data one may only be concerned with some properties of the underlying graph which itself is much too large for main memory, necessitating streaming approaches that drill down specifically to those properties. In a genomics context, we might be interested in analyzing a set of sequencing reads without generating intermediate files or making multiple passes over large text files of sequences.

The data stream model was initially formalized by researchers at Digital Systems Research Center (1). They define a *data stream* as "a sequence of data items $x_1 \dots x_i \dots x_n$ such that the items are read once in increasing order of the indices i"; this definition has persisted to today. Their work further parameterized a computational model by the number of passes P, with work built on this definition sometimes referring to algorithms where $P > 1$ categorized as semi-streaming. While there is a paucity of surveys considering streaming algorithms in *general*, several focus on more specific applications while

giving some generalized overview of streaming models. In (2), streaming algorithms for massive graphs are explored. (3) explores the streaming model in contrast to traditional database management approaches, and further subdivides streaming methods: approximate query processing, sliding window approaches which consider only recent elements of a stream, sampling and "synopsis" approaches for when the rate of a stream is too high, and the tradeoffs between batching, sampling, and sketching. *Data Streams: Algorithms and Applications* (4) is a textbook survey of broader data stream problems, of which streaming algorithms are a subset.

Streaming in Omics

With omics data generation continuing to accelerate, interest in streaming approaches has grown. There are a number of advantages to a transition to streaming approaches in omics:

The "Data Deluge" Even with per-base sequencing costs stabilizing, the number of sequences generated by various institutions worldwide keeps growing at a geoemetric rate. However, the actual cost of the sequencing itself is dwarfed by the cost of the computing and downstream analysis of the data (5). Furthermore, the adoption of cloud-computing has shifted the computing costs for many institutions from a fixed to a variable-cost model; this model incentivizes minimizing intermediate file storage and efficient algorithms that are more easily distributed in a principled, independent way. Streaming approaches can reduce I/O and bandwidth mid-stream while transferring data to cloud services; they can help remove intermediate files that increase variable costs; and they can help decompose problems for distribution across disparate cloud systems. Streaming thus helps address both data volume and shifting funding models.

Real-time Sequencing Strategies While Next Generation Sequencing (NGS) technology operates in a massively parallel fashion, real-time single-molecule sequencing technologies like nanopore sequencing are slowly gaining primacy. In this new paradigm, polynucleotide molecules are moved through a pore and observed one base at a time, meaning that the process of sequencing itself is made streaming. These technologies allow much longer sequence lengths, and as their error rates decrease, it is expected they will supplant NGS technologies. As real-time sequencing becomes the norm, so will streaming algorithms that can analyze sequence as soon as they are available from the machine. Indeed, these sorts of approaches are already being developed, with algorithms for pathogen detection (6), barcode demultiplexing (7), and basecalling (8) already published.

Data Accessibility Streaming can act as a data accessibility multiplier. Institutions with limited resources benefit greatly from reduced storage requirements, as storage can represent a significant up-front cost. Reduced data requirements may also translate to reduced network requirements, which helps with accessiblity in locations where network access is at a premium. Finally, streaming approaches often go hand-in-hand with sketching approaches, further reducing computational requirements and helping to bring omics analysis to individual users.

Sequence Assembly: A Fundamental Problem in Genomics

Assembly refers to the reconstruction of a set of hidden sequences G_h from some set of fragments R of length L randomly sampled from G_h, with G_h being the underlying genome or transcriptome of an organism (or organisms) being sequenced. The introduction of shotgun sequencing in 1979 (9, 10) created a concrete need for algorithmic

solutions to the assembly problem, the first complexity analysis of a simplified sequence assembly problem having already been carried out in 1978 (11). The first simple genome assembler came later on in 1988, using a greedy overlap-layout-consensus (OLC) approach (12). The simplest approach, which (11) described, is casting it as the NP-Complete shortest common supersequence (SCS) problem. This approach is inadequate for actual samples, as genomes are plagued by repetitive sequence: tandem repeats (the same substring repeated back-to-back many times), interspersed repeats (from structures like transposable elements), and paralogs (genes which are repeated within a genome). Overlap detection is further complicated by error intrinsic to the sequencing process. Regardless of origin, these repeat classes all introduce a major challenge to assembly, and as pointed out in (9), break the SCS assumption of parsimony and drive the majority of work in the field.

de Bruijn Graphs, Assembly Graphs, and their Uses

Although other approaches exist (12, 13), almost all modern assemblers rely on the assembly graph paradigm. Broadly, a graph is built with sequence fragments as the nodes and overlaps between them as the edges, and simplified to remove transitive redundancy. Appropriate traversals on this graph yield sequences from G_h (14). The method of construction and structure further leads to two formulations of the assembly graph: the string graph (15) and de Bruijn graph. While the string graph method is commonly used (16–18) and important, in this work I focus on the de Bruijn graph method; both methods are bounded by the same complexity constraints (19), though each has its own practical and theoretical advantages.

A de Bruijn graph (dBG) $G(k, \Sigma)$ is a directed graph defined by its nodes, which are a set of fixed-length sequences of length k over an alphabet Σ, and its edges, which exist

when the sequences of a pair of nodes share a length $k-1$ suffix and prefix. These length-k sequences are referred to as k-mers, and the terms nodes and k-mers may be used interchangeably. This means that $G(k, \Sigma)$ has a maximum number of nodes $\|\Sigma\|^k$ and each node has a maximum in and out degree of $\|\Sigma\|$. If we define $pre(n_i)$ to be the length $k-1$ prefix of some node $n_i \in G$ and $suf(n_i)$ to be the length $k-1$ suffix, the out-neighbors of n_i can be discovered by querying $\{suf(n_i) + s, \forall s \in \Sigma\}$ and the in-neighbors accordingly: that is to say, in this construction, known as the *node-centric* de Bruijn graph, the edges are implied by the set of nodes, and it is only necessary to use an efficient set data structure to implement a basic dBG.

Let $\delta_{in}(v)$ be the in-degree of a node v, and $\delta_{out}(v)$ be the out-degree of a node v. A node v where $\delta_{in}(v) > 1$ or $\delta_{out}(v) > 1$ is a *decision node*, and its associated k-mer a *decision k-mer*.

de Bruijn graphs have been used to great effect in genomics applications since they were first applied to them (14). The principle advantage of the dBG is its simplification of the process of overlap detection between sequences: k-mers can stored as hashed integers for faster querying, and length $k-1$ overlaps can be discovered by using the same method as edge detection. Hence, when a collection of sequences sampled from an underlying larger set of sequences is broken down into its constituent k-mers as a dBG, all k-mer overlaps are known implicitly. This property, and the methods extending it, can be used for genome assembly, sequence classification and comparison, genomic variant discovery, and full graph-based alignment.

Compact de Bruijn Graphs

While the standard de Bruijn graph is a useful abstraction, in practice, most implementations only use it as an intermediate representation. The reasons are twofold: firstly,

because genomic dBGs tend to consist of many linear paths of k-mers with in- and out-degree of 1, such graphs contain large quantities of redundant k-mers; secondly, because k-mers are exact substrings over a sliding window on the input sequences, each error or point mutation in an input sequence can introduce up to k new nodes in the graph. As a result, even low error and polymorphism rates in moderate sized data produce graphs with hundreds of millions or billions of nodes.

These problems are addressed by the compact(ed) de Bruijn Graph (cDBG). In the cDBG, paths of the form $v_0 \dots v_i \dots v_n$, where $\delta_{in}(v_{1..n-1})$ and $\delta_{out}(v_{1...n-1})$ are exactly 1, are merged into single nodes; that is, linear paths are reduced to single nodes defined by their sequence, and the ends of these sequences may be either tips (that is, they have degree zero) or decision k-mers. This representation preserves the implicit edge definition, while avoiding the high overhead of storing redundant k-mers.

The compacted nodes, and their associated sequences, are referred to as *unitigs*. A unitig that can not be extended in either direction is a *maximal unitig*: this is, $\delta_{in}(v_0) = 0$ or v_0 is a decision node, and $\delta_{out}(v_n) = 0$ or v_n is a decision node. The term *unitig* was first introduced in (20), where the process of forming unitigs was referred to as "condensation." *Compaction* and *compact* or *compacted* de Bruin graph have since become the accepted terms. The most basic process of compaction was earlier described and implemented for the Velvet assembler (21), the first mainstream de Bruijn graph assembler; here it was introduced as a lossless graph simplification method.

There are a variety of data models for representing the cDBG. The simplest is a flat collection of unitigs; edges can be inferred by storing only the unitig ends, which correspond to the decision k-mers and tips, and querying these ends for neighbors. How this subset of k-mers is stored can vary by implementation, with simpler models using a hash table directly (22, 23), and later work transitioning to succinct data structures like the

FM-Index (24). Regardless, a key observation is that the compact de Bruijn graph is defined by its set of decision k-mers, which, along with the formalization of the node-centric de Bruijn graph (25) (where edges are stored implicitly, as defined previously), brings about the concept of Navigational Data Structures for assembly (26). (26) provides a good background survey of the methods used for navigational data structures up until 2015; since then, considerable work has gone into augmenting the graph with additional information, such as the colored compact de Bruijn graph, which is beyond the scope of this work.

Notably, the succinct data structures, while offering query and storage efficiency, are not straightforward to use in a dynamic fashion. Simpler membership structures, like the hash tables used in (21, 23) or Bloom filters (27), allow dynamic insertion of new k-mers, and hence edges, using the node-centric model. Dynamic data structures that allow efficient insertion and query are key to enabling streaming approaches.

Conclusion

The rapid and growing generation of sequence data, the advent of real-time sequencing technologies, and the need for accessible genomics analysis for a wide range of consumers can be partially addressed by the adoption of streaming approaches. While some methods have already been adapted for streaming, de Bruijn graph compaction, a key component of assembly pipelines, has not; further, sublinear methods remain uncommon. Here, we will first focus on adapting compaction for streaming, and show how this new paradigm also enables novel analysis methods for assembly graph structure. Then, building on that work, we will explore preliminary sub-linear approaches for filtering sequences prior to compaction and further downstream analysis. Our goal is to provide a framework for more streaming methods to build on in the future: we hope that, as technology

and algorithms improve, we will eventually see the evolution of end-to-end streaming pipelines, from sequencing instrument to biological inquiry.

Chapter 1

Streaming Construction of the Compact de Bruijn Graph

Chapter Authors Camille Scott and C. Titus Brown

Abstract

Traditionally, approaches for sequence fragment analysis such as assembly have been off-line; assembly, in particular, has been a process in which all data is first loaded into memory (generally, a graph abstraction) before further processing to extract contigs. As the volume of available data increases, and constantly updating data streams become ubiquitous, it is advantageous to consider on-line, or streaming, approaches which view the data as a stream of observations which are used to update an analysis or knowledge representation as they are integrated. Here, we present a streaming approach to construction of the compact de Bruijn Graph, a common assembly graph abstraction.

BACKGROUND

The procedure is described theoretically, and a reference implementation is explored on several example data sets.

Background

Context

Developments in high-throughput sequencing technology have made genomics, transcriptomics, metagenomics, and their variants core methods for biological inquiry. These sequencing technologies work by randomly sampling short fragments, or reads, from the underlying pool of DNA (or RNA) at a high redundancy, after which computational methods are used to assemble an approximation of the original sequence. While early methods directly computed overlaps between shallowly-sampled fragments (28), deep sequencing has necessitated the development of more clever data structures and algorithms for assembly. Assembly graphs are one such abstraction (9, 14, 25), and are used as an intermediate representation in most extant assemblers; the de Bruijn graph (15, 18, 29, 30) is a core method for constructing assembly graphs.

In the de Bruijn graph methods, reads are broken down into substrings of length k, or k-mers, from which the unique representatives become the nodes in the graph. Edges are defined when nodes have k-mers with exact overlaps of length $k-1$; this implicit definition of edges obviates the need for explicit overlap computation, making de Bruijn graphs well-suited to deep sequence experiments with short reads (see 1). While this property greatly improves the time scaling of the assembly problem, the de Bruijn graph suffers from extremely high storage requirements due to the explicit enumeration of k-mers. To address this, it is often compacted by the contraction of linear paths of non-branching

k-mers, forming the compact de Bruin graph (20).

A number of approaches for generating the compact de Bruijn Graph are already established. BCALM2 generates a compact de Bruijn Graph from short reads in parallel using ranked minimizers (25), building on their previous work on de Bruijn graph representation and compaction (31). TwoPaCo generates the cDBG from a collection of previously assembled genomes using a two-pass Bloom-filter approach to iteratively eliminate positions in the input genomes as potential decision k-mers (32). Bifrost efficiently constructs the colored cDBG using blocked Bloom filters and and minimizer indexing (33), while Cuttlefish uses finite-state automata to efficiently model the colored cDBG (34). Regardless of the method for generating the graph, extant approaches share a paradigm: they are off-line algorithms, algorithms which require access to the entire sample before proceeding with compaction, and make multiple passes over the data. In contrast, streaming (or online) algorithms view their data as a sequence of observations, and make one or fewer passes over that stream (35), while semi-streaming algorithms aim for a small, constant number of passes, generally two or fewer (36). Streaming approaches have several advantages: they minimize hard disk access, which aids in efficiency; they are often able to produce results opportunistically, as soon as they are available; they offer the potential for feedback between the data stream and the processing; and often, they are suited for handling streams of infinite length.

Here, we explore an implementation of a streaming de Bruijn graph compactor. Our implementation builds the compact de Bruijn graph opportunistically from a stream of sequence fragments, in one pass over the data. While semi-streaming de Bruijn graph compactors have been previously published (37, 38), our implementation produces a compact de Bruijn graph in one pass.

Definitions

For some string s, let $s[i]$ be the symbol at position i, 0-indexed, $s[i:j]$ be the substring of s starting at position i and not including position j, and $len(s)$ be the length of s. If s has length L, the k-mers of s are given by $kmers(s) = \{s[i:i+k] \mid 0 \leq i < L-k+1\}$; that is, all length k substrings of s. Let $pre(s) = s[0:k-1]$ and $suf(s) = s[L-k+1:L]$, that is, they yield the suffix and prefix of s.

We then define a de Bruijn graph as $G_{k,\Sigma} = (N, E)$, where N is a set of k-mers over alphabet Σ and E is the set of length k-1 exact overlaps between k-mers in N; that is, $E = \{e_{u \to v} \mid suf(u) = pre(v) \forall u, v \in N\}$. For convenience, assume $\Sigma = \{A, C, G, T\}$).

Given a set of sequences S, the de Bruijn graph $G_k(S)$ has $N = \bigcup_{s \in S} kmers(s)$. Note that our edge definition admits edges even when the corresponding $(k+1)$-mers containing the two overlapping k-mers are not present in any sequence in S; this is the node-centric de Bruijn graph, where edges are defined implicitly by the set of known nodes. The in-neighbors and out-neighbors of a node u can then be discovered by querying $\{suf(u) + \sigma \forall \sigma \in \Sigma\}$ and $\{\sigma + pre(u) \forall \sigma \in \Sigma\}$ in G_k, respectively. Let $\delta^-(u)$ and $\delta^+(u)$ yield these in-neighbors and out-neighbors.

Now let S be an ordered stream of sequences rather than a set. $S[t]$ is then the tth sequence in the stream, and $G_k(S)[t]$ is the de Bruijn graph with $N = \bigcup_{s \in S[0:t+1]} kmers(s)$. The total number of sequences in S may or may not be known. We refer to the position t in the stream as the *time*. *time* may be analogous to actual observation time or simply position within a file, depending on the source of the stream.

The compact de Bruijn graph (cDBG) is built from the de Bruijn graph by converting non-branching paths of k-mers into single nodes. Define $C(G_k) = (N_c, E_c)$ as the compact de Bruijn graph built from G_k; for brevity, we call it C_k. A node (k-mer) $u \in G_k$ is a

decision node (k-mer) if $|\delta_{in}(u)| > 1$ or $|\delta_{out}(u)| > 1$; that is, if its in-degree or out-degree is greater than one. N_d is the complete set of decision nodes in G_k. Consider a connected path $p = \{u_0, u_1, ..., u_L\} \in G_k$. p is a unitig if none of its nodes are decision nodes, and p cannot be further extended in either direction; a unitig in C_k is a unitig node. The complete set of unitig nodes will be called U, and we define $N_c = N_d \cup U$. The edges E_c of the cDBG are then $\{e_{u \to v} \mid suf(u) = pre(v) \forall u, v \in N_c\}$. Note that this implies that, while decision nodes may be neighbors, unitig nodes may not.

A unitig node with no neighboring decision nodes is an *island*; a unitig node with decision nodes on either side is *full*; and a unitig node with a decision node on only one side is a *tip*. The edge-case where $pre(v) = suf(v)$ for a unitig v is called a *circular* unitig. As with the de Bruijn graph, we refer to the cDBG at time t as $C_k[t]$.

Methods

Broadly, the underlying de Bruijn graph is implemented as a simple set. Sequences are broken down into constituent k-mers and hashed, and if a sequence contains any k-mers not previously observed, the cDBG is updated from the sequence. The minimum set of traversals is made from the new k-mers to find newly introduced decision k-mers which split existing unitigs and extensions of existing unitigs, and new unitigs are added when necessary.

The next sections describe the compaction process and implementation in detail.

Construction of the de Bruijn Graph

The de Bruijn graph is implemented as a simple set of k-mers. As such, it can be updated in a streaming fashion with no additional methods: a new sequence s_t from SS is broken down into its constituent k-mers and hashed using a rolling hash scheme (39), with new k-mers being added to the set and existing k-mers being incremented if a counting data structure is in use. The node-centric model saves memory by eliding edge storage at the cost of requiring a seed sequence to begin traversal.

While the implementation supports several probabilistic data structures taken from the `khmer` (40, 41) package, the reference implementation uses an exact hash set provided by the `parallel-hashmap` library (42). This limits the practical application to moderately-sized assembly graphs due to memory overhead.

cDBG Data Structure

The cDBG data structure consists of two maps: first, a basic hash table of known decision nodes N_d, and a map from k-mer hashes of the ends of unitig paths to the unitig nodes U. These two structures are sufficient to define the entire cDBG: neighbors of decision nodes are necessarily tips in U, and island unitigs can be discovered from their tips. Note that unlike in the standard unitig model, where decision k-mers are the ends of unitigs, we store decision k-mers as their own nodes; this is an implementation detail, and standard unitigs can be trivially deduced by joining our unitig nodes with their neighboring decision nodes. Unitig nodes store the hashes of their ends for reverse indexing, a 64-bit unsigned node ID, a 64-but unsigned component ID, the full sequence of their path in the dBG, and a meta attribute corresponding to their type: tip, full, circular, or decision. The component ID is not strictly necessary for compaction functionality and is currently only

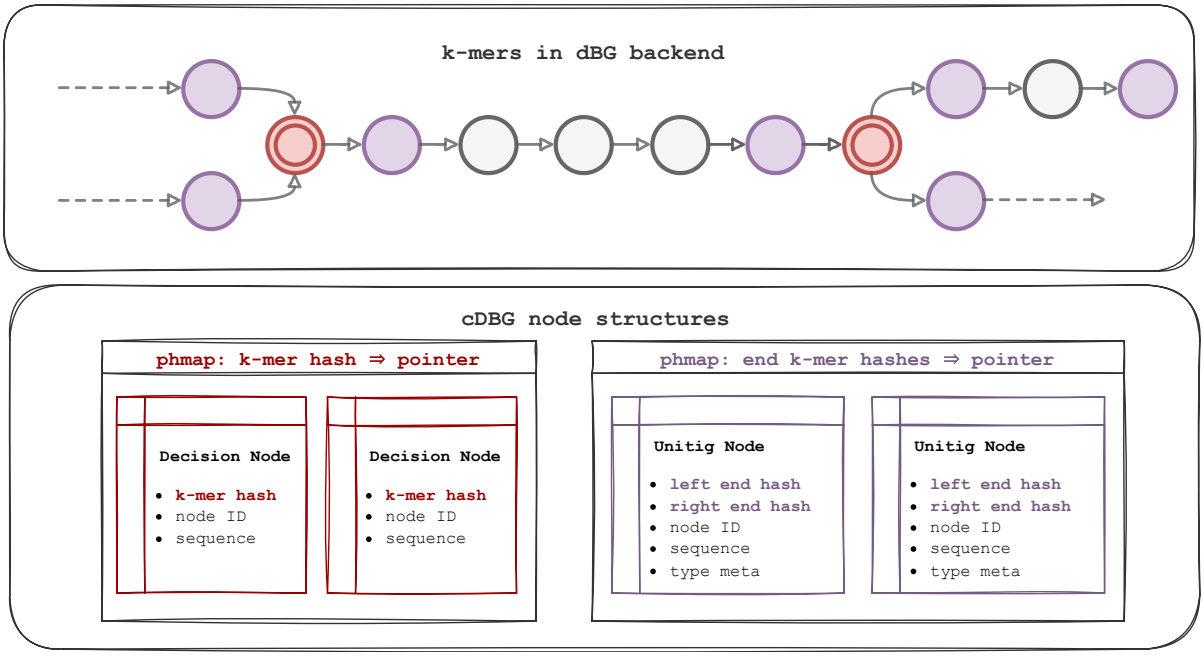

Figure 1.1: **goetia cDBG Storage Model** Unitigs are indexed by the hash value of their end k-mers; these are stored in a `parallel-hashmap` linking the hash values to the unitig pointers. Decision k-mers are index in their own hashmap by their k-mer hashes. Both store their sequences; in practice, for memory efficiency, we could store only the end k-mers to seed traversal, but we choose to store the entire unitig to be able to quickly write the cDBG to disk.

used for the analysis function that finds connected components.

Mutations to the cDBG

The streaming compact de Bruijn graph can be defined by its decision k-mers and the k-mers terminating its current unitigs. While unitigs are subject to mutation throughout streaming construction (splitting and merging), decision nodes within the cDBG are never deleted once created, a property which can be exploited during graph construction.

In the following sections, a *new* k-mer in G_t is a k-mer not present in G_{t-1}; that is, a k-mer first introduced in fragment t. A *new decision k-mer* is a new k-mer which is also a decision k-mer. An *induced decision k-mer* is a k-mer which is not newly introduced,

but became a decision k-mer at time t. A *new unitig node* is a unitig comprising only new k-mers. An existing unitig node is *split* when one its k-mers becomes an induced decision k-mer, and two unitig nodes are merged when a path of new k-mers exists between two of their tip k-mers. Given a sequence $s_t \in SS$, the *segments* of s_t are the paths of new k-mers in s_t, terminated at existing k-mers and new decision k-mers. Now, we will define the ways in which a segment can mutate the cDBG.

Inserting New Sequences and Finding New Segments

The first step to updating the cDBG with a new sequence is to insert its k-mers into the dBG and divide its new k-mers into their constituent segments. Once we have the segments, we use them to update the state of the cDBG. Alg. 1 describes this procedure: segment paths are split on existing k-mers and new decision k-mers. Only the first and last k-mers in a segment, and new decision k-mers, are able to induce an existing k-mer to become a decision k-mer, and thereby potentially split an existing unitig. Alg. 2 lays out the procedure for handling these k-mersr; see also lemma 1.0.1. With this procedure defined, we are able to update the cDBG from the new segment. We attempt to induce decision k-mers from the first and last k-mers in the segment and split unitigs if necessary; if the first or last k-mer is not a new decision k-mer and does not match up to an existing unitig tip (corr. 1.0.2.1), we search for induced decision k-mers only in the direction away from the segment. If neither end of the segment matches to an existing unitig, we produce a new island unitig with just the segment. The full procedure is detailed in 3.

Algorithm 1: FindSegments

Data: dBG $G_{k,t-1}$, sequence stream SS
Input: sequence $s_t \in SS$
Result: A list ℓ of new segments from s_t, set of new decision k-mers δ

```
U ⟵ kmers(s_t)
P ⟵ ∅
for u ∈ U do
    P.append(G.query(u))
    G.insert(u)
p ⟵ False
v ⟵ U[0]
segment ⟵ ∅
for v ∈ U, q ∈ P do
    if q = True then
        if p ≠ False then
            segment ⟵ {u}
        if G.lneighbors(v) > 1 || G.rneighbors(v) > 1 then
            if v ≠ U[0] then
                ℓ.append(segment)
            segment ⟵ {v}
            δ.insert(v)
    else if p = True then
        ℓ.append(segment)
        segment ⟵ ∅
    u ⟵ v
    p ⟵ q
if q = True then
    ℓ.append(segment)
return ℓ, δ
```

Algorithm 2: InduceDecisionKmers

Data: dBG $G_{k,t-1}$, cDBG $C(G_{k,t-1})$
Input: k-mer u which can induce decision k-mers, set Δ of new k-mers to ignore

```
for v ∈ G.neighbors(u) do
    if v ∉ Δ and IsDecision(v) then
        C.NewDecisionNode(v)
        C.SplitUnitig(v)
```

Algorithm 3: InsertSegment

Data: dBG $G_{k,t-1}$, cDBG $C(G_{k,t-1})$
Input: list of k-mers comprising *segment* from $s_t \in SS$, set of new decision k-mers δ, set of all new k-mers Δ

$l, r \longleftarrow \boldsymbol{True}$
if $segment.first \in \delta$ **then**
    $C.NewDecisionNode(segment.first)$
    $InduceDecisionKmers(segment.first, \Delta)$
else if $G.lneighbors(segment.first) \notin C.U$ **then**
    $InduceLeftDecisionKmers(segment.first, \Delta)$
else
    $l \longleftarrow \boldsymbol{False}$
    $C.ExtendUnitig(G.lneighbors(segment.first), segment)$
if $segment.last \in \delta$ **then**
    $C.NewDecisionNode(segment.last)$
    $InduceDecisionKmers(segment.last, \Delta)$
else if $.rneighbors(segmen.last) \notin C.U$ **then**
    $InduceRightDecisionKmers(segment.last, \Delta)$
else
    $r \longleftarrow \boldsymbol{False}$
    $C.ExtendUnitig(G.rneighbors(segment.last), segment)$
if $l = r = \boldsymbol{True}$ **then**
    $C.NewUnitig(segment)$

Splitting Existing Unitigs from Induced Decision k-mers

Splitting an existing unitig is the most costly and common operation. When an existing k-mer μ is induced as a decision k-mer, we must traverse into the dBG until finding a hash which links to the unitig in the cDBG. If the induced decision k-mer is a tip, we will discover the unitig immediately: this is an edge case we refer to as clipping, in which the unitig is shortened and its tip reindexed. Otherwise, μ has exactly two neighbors which are not new from which we can traverse into the existing unitig. We traverse through unitig k-mers until stopping at either a tip or an element of T, checking the index as we go. The discovered unitig is then split according the to the induced decision k-mer.

Correctness

Lemma 1.0.1 (Inducers' Lemma)**.** *A decision k-mer can only be induced by a new decision k-mer in s_t or the first or last k-mers in a new segment of s_t.*

Proof. Consider an existing non-decision k-mer $\mu \notin s_t$. An existing k-mer in s_t cannot induce μ to be a decision k-mer: if so, μ would already be a decision k-mer. Now consider a new k-mer ν in the fragment which is not the first or last k-mer of a segment. In order for ν to induce μ to be a decision k-mer, μ must be a neighbor of ν; because ν is not an end k-mer, it already has in-degree and out-degree at least one, from the new k-mers on either side of it. Now consider an end k-mer of a new segment: it has at least one new k-mer as its neighbor, and possibly an existing k-mer μ as its other neighbor. If μ is an in-neighbor of ν, and already has out-degree of one, then ν can induce μ while being a non-decision k-mer; this exists for the case where μ is an out-neighbor as well. □

Lemma 1.0.2 (Splitting Lemma)**.** *An existing unitig node can only be split by an induced decision k-mer.*

Proof. An existing unitig is, by definition, composed of existing non-decision k-mers. A new k-mer cannot split an existing unitig, also by definition: if the k-mer were part of the unitig, it would not be new. Thus, in order to split a unitig, we must convert an existing k-mer into a decision k-mer. □

Corollary 1.0.2.1 (Tip Corollary)**.** *If a k-mer μ which is the first or last k-mer of a new segment has a neighbor ν which is not a unitig tip, then ν must be an induced decision k-mer.*

Graph Metrics

In order to analyze the compact de Bruijn graph during construction, we optionally gather a number of metrics during the stream. The counts of the unitig node types described in Definitions (see 1) are taken whenever a CDBG mutation is performed via appropriate incrementation and decrementation atomic counters; we also track the types of mutations performed, including unitig splits, merges, extensions, clips, and deletions. Counts of unique k-mers are tracked when new k-mers are inserted into the dBG storage backends. These metrics are essentially "free" to compute, and so can be tracked with each sequence insertion.

Other, more advanced metrics are more expensive, and must be taken at larger intervals for performance. Timing for advanced metrics is determined by a tunable k-mer interval, which is described in more detail in Chapter 2. Briefly, the fundamental unit of time is the k-mer, instead of the inserted sequence, to correct for varying sequence lengths, with the interval size I generally set at some value much larger than the sequence length; for

most results shown here, the interval is $1,000,000$ k-mers. The actual number of k-mers between measurements is actually $I + \varepsilon$, where $\varepsilon \leq L$, so as to insert complete sequences. We save the above count metrics to disk (or output stream) at every interval, as the required disk access is trivial to write a single row of JSON formatted integers. The two less granular metrics are the histogram of unitig lengths and the number and sizes of connected components. The unitig length histogram is calculated by enumerating all unitigs and incrementing the appropriate histogram counter. As this does not require traversal, and in the majority of cases the number of unitigs is far less than the number of unique k-mers, this can be done efficiently, and so is performed at a modulus of the interval count (every 5 intervals for the results shown here).

Connectivity is much more resource intensive. First, we perform a breadth-first traversal of the complete cDBG, starting from the map of unitig ends and removing all ends in a connected component from the traversal queue until all unitigs are exhausted. The cDBG is, like the underlying dBG, node-centric, so finding unitig neighbors requires hashing the extensions of the k-mers at each unitig end. We then sample the sizes of all discovered components using reservoir sampling (43), by default taking a sample of $10,000$ components. This operation grows in complexity with the number of nodes and edges in the cDBG, and hence with the number of unique k-mers in the underlying dBG, though it is at least bounded by the maximum node degree $\|\Sigma\|$. To account for this growth, we back off the number of intervals between traversal using the exponential growth function with $r = 0.8$; that is, when the number of intervals between ticks during the stream is $\{1.08^m \| m \in 0 \dots N\}$, where N is the number of measurement occurrences. Because the graphs tend to converge on stable topologies later in the stream, we find that this balances measuring the interesting changes in the cDBG while being frugal with computation time.

Analysis Parameters

Results for streaming compaction were generated with `goetia` at commit a874e71d65 using the command `goetia cdbg build -K 31 -S PHMapStorage -H FwdLemireShifter --interval 5000000 --track-cdbg-metrics --pairing-mode split`. Streaming solid filtration was performed with `goetia solid filter -K 31 -S ByteStorage -H CanLemireShifter -x 2e9 -N 4 -C 3 -P 0.8 --pairing-mode split`; that is, reads are admitted if 80 of their k-mers have an abundance greater than or equal to 3 at the time of observation.

Benchmarking was performed on a system with an AMD Ryzen 9 5900X CPU, 64GB of RAM, and a SATA SSD. The download connection speed as measured with the `speedtest-cli` package (44) is approximately 300Mbit/s (~37MiB/s), which, in our tests, is faster than the FTP download speed for FASTA files from the ENA. The download stream tests were performed in serial to avoid interference between test runs.

Samples for analysis were selected from the European Nucleotide Archive (ENA). Accessions with `library_source` of `TRANSCRIPTOMIC` and `GENOMIC` were first subselected from the complete list of sample accessions; then, these were further reduced to those from the species *Caenorhabditis elegans*, *Mus musculus*, *Drosophila melanogaster*, *Saccharomyces cerevisiae*, and *Home sapiens*. Three accessions were randomly selected for each species and each library source.

Results

Implementation: the `goetia` library and tools

The streaming compactor is implemented in the `goetia` (45) package, which is hosted on GitHub and available under the MIT license. It includes implementations of several exact and probabilistic k-mer sets and counters, pluggable hashing methods for those counters, de Bruijn graph implementations built on top of those counters, streaming file processing and parsing utilities, and the core streaming compaction implementation. This functionality is implemented as a C++ library, which can be installed as a shared library via `bioconda` (46). High-performance Python bindings are provided using `cppyy` (47), which are used in a corresponding Python library to provide a command line interface (CLI), analysis tools, and unit testing. Unit tests are performed using a library of pytest fixtures called the `debruijnal-enhance-o-tron` (48). These fixtures randomly generate common assembly graph features parameterized by K, sequence lengths, and other constraints, allowing us to unit test compaction correctness in a manner similar to fuzzing. The dBG generators in the debruijnal-enhance-o-tron are implemented as an interface over a user-supplied dBG implementation, which means they bootstrap correctness-checking of the basic dBG and storage implementations as well.

The cDBG can be constructed as sequences are downloaded in real time

A key advantage of streaming methods is the ability to bypass writing the input data to disk. While streaming approaches are often more computationally complex than offline approaches, particularly when exact solutions are required, they can be valuable tools

for pipeline construction when scaling is less of a concern, for example with smaller data or when running on serverless architectures. Here, we show that our streaming compaction prototype functions at speeds practical for use as a prefilter for sequences directly downloaded from a remote source; that is, we can compact sequences at a rate commensurate with the speed of many sequence archives, with disk access only occurring when writing assembly graph metrics and saving the final resulting compact dBG. This prototype implementation struggles with memory scaling for large genomes, and as such, we have limited the input read data sets to samples with less than 100 million fragments.

When reading sequences from the ENA using `curl`, `goetia cdbg` compacts at rates ranging from 2 to 5 MiB/s. These are approximately the download rates from the ENA on the test connection. Compaction rates are largely dictated by the chosen de Bruijn graph parameter K, the error rate of the sequencing experiment, and the polymorphism rate of the sequence being sampled. The latter parameters drive the number of the decision nodes in the assembly graph: the more decision nodes in the assembly graph, the more fragmented, and the more graph operations required per insert.

While these rates are sufficient for compacting directly downloaded sequence, the compaction rate is not the only important factor. In particular, memory requirements, which are also driven by the number of decision nodes (as well as the number of unique k-mers), are often the limiting factor. With exact counting backends, memory use grows as additional k-mers are added to the de Bruijn graph; non-exact, fixed-memory backends can help alleviate this issue, at the cost of producing a compacted dBG not necessarily exactly equivalent to the actual cDBG. For exact backends, we have the option to terminate the compaction process when memory limits are increased. Assuming randomized ordering of the stream, the resulting cDBG will be equivalent to one built by downsampling the

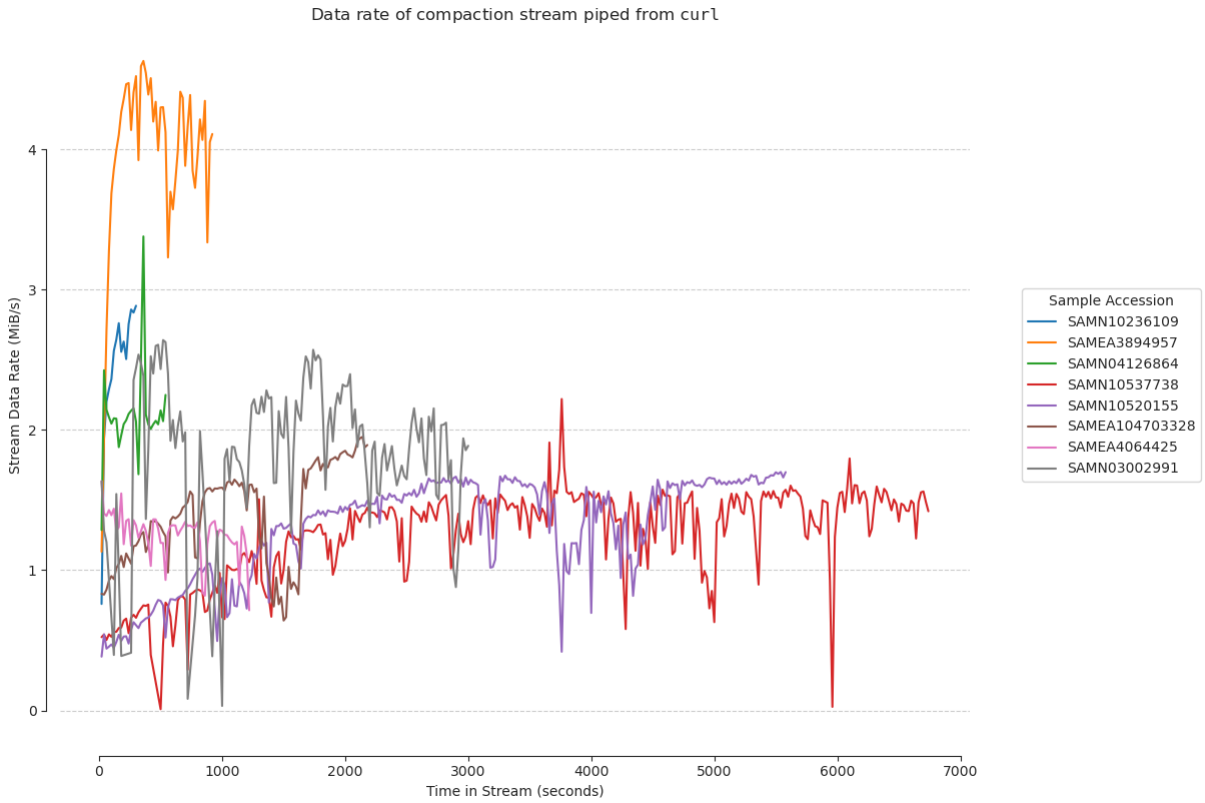

Figure 1.2: **Streamed compaction data rates of various samples.** Paired-end samples were streamed directly from the European Nucleotide Archive (ENA) using the `curl` command. The data rates were measured using the `pv` command as follows: `curl -sL [URL] | pv -tbn 2>> [RESULTS] 1>> [OUTPUT]`. `[OUTPUT]` is a unix pipe read by `goetia cdbg build`; data rates were aggregated at 5 second intervals between both the left and right hand sample results. Higher data rates correspond to higher compaction rates and better performance. Performance here is bottlenecked by either the download bandwidth or the compaction rate, whichever is slower. `SAMEA3894957` is an outlier here, with a higher download rate at the time of testing, and being a relatively small *S. cerevisiae* genome.

input sequences.

Streaming Compaction Performance is Dominated by dBG Construction

To assess the performance of streaming compaction in bioinformatics pipelines, we compare it to two other streaming k-mer methods: only hashing the k-mers in the input sequences with the same underlying hash function, and building the de Bruijn graph with the same storage backend. k-mer hashing is performed in dozens of existing programs, and here, regularly exceeds 40 million k-mers per second, or 600 thousand sequences per second. de Bruijn graph construction is mostly determined by the speed of the `insert` operation on the underlying storage backend. With the exact backend using `parallel-hashmap`, we approach 10 million k-mers per second or 100 thousand sequences per second; note that the hashing task is a subset of the dBG construction task. Compaction hovers around 3 million k-mers per second, or 50 thousand sequences per second. Fig. 1.3 shows the full results of this test. We show results in both k-mers per second and sequences per second, but the underlying timing in our implementation is based on k-mers, as sequence length varies between samples.

Compaction tends to be more than an order of magnitude slower than only hashing, and about three times slower than dBG construction. As dBG insertion is a subset of the compaction task, insertion speed is a top candidate for optimization. `goetia` also does little in the way of optimizing string operations, such as bit packing or efficiently allocating memory with freelists, suggesting considerable room for improvement over the existing prototype. Finally, this implementation is single-threaded. A storage backend using minimizer partitioning is included in the `goetia` library, which has the potential to enable multithreading in future versions, but we consider this currently outside the

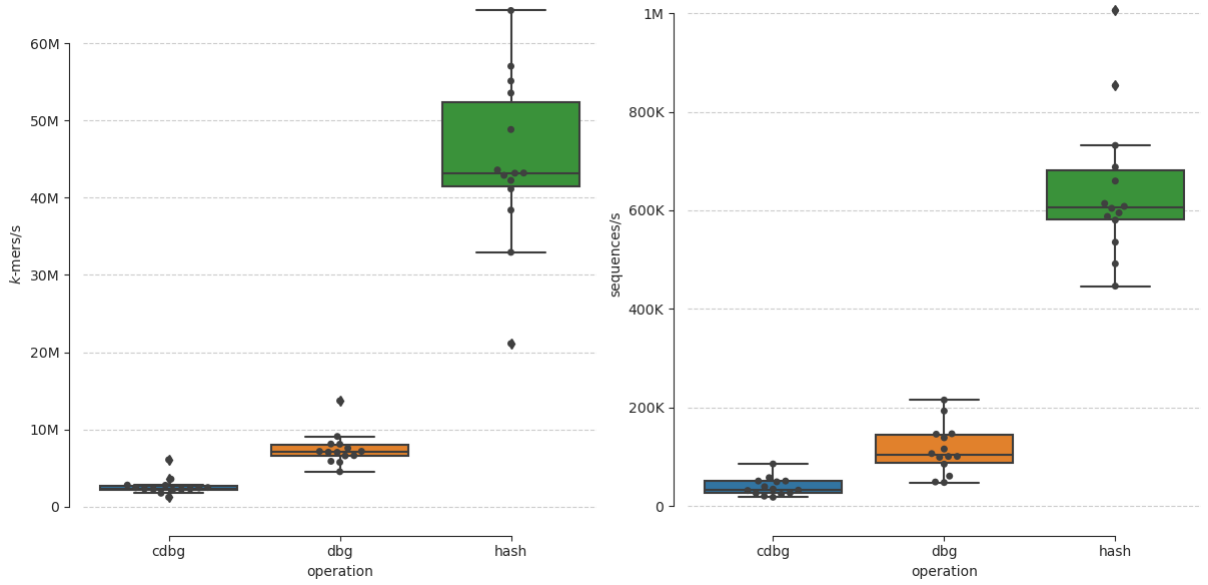

Figure 1.3: **Streaming processing rates of k-mer hashing. dBG construction, and compaction.** Processing rates in k-mers per second and sequences per second of increasingly complex dBG tasks. Each point is the rate result from a distinct transcriptomic sample. Higher values correspond to higher data rates and better performance.

scope of this work; some details on this backend can be found in Appendix A.

Streaming Methods Yield New Approaches for Studying Sequence Data and Assembly Graphs

Streaming construction of the cDBG intuitively enables dynamic analysis of the graph. Fig. 1.4 demonstrates this capability on several *Homo sapiens* transcriptomes. Each of the three samples shows distinct growth metrics, with the two Poly-A selected samples, on the left and right, sharing the most similarity, and the middle, high-coverage rRNA-depleted sample standing out. All three samples, as shown in the top row, have the highest proportion of unitigs in the range of 50-100bp. They also all show the slopes of their length proportions flattening out as more sequence is added, which we interpret as approaching having observed most of the core sequence content of the underlying sample.

We expect to see this "saturation" behavior when most of the cDBG topology has been filled in: once the core structural k-mers have been observed, we mostly accumulate error (49). We can further explore these unitig distributions by comparing them to their associated graph topologies demonstrated in the bottom row. The middle, rRNA-depleted sample has almost no unitigs longer than 100bp. It also has a much higher proportion of island unitigs (those disconnected from any other unitig) compared to its Poly-A selected counterparts, and its number of island unitigs follows the distinctive saturation curve. These observations suggest that this sample is dominated by many low-abundance transcripts, and also that most of the underlying sequence that there is to observe has been accounted for; the former is what we might expect given rRNA-depletion's affinity for many more RNA species (50), and the latter is supported by its higher coverage in comparison to its sister samples. We can also assume that any assembly of this cDBG will necessarily be highly fragmented.

In contrast, the two Poly-A selected samples show very few islands, while also showing distinctive peaks followed by flattening in their proportions of longer unitigs. Additionally, their numbers of "full" and "tip" unitigs remain highly correlated: we could expect this behavior when a bubbles are consistently being introduced to the graph, which result from sequence error in the presence of higher regional coverage.

Fig. 1.5 demonstrates the accumulation of decision nodes as the dBG is compacted. Most samples demonstrate the saturation behavior previously discussed. As additional sequence is added to the graph, decision nodes make up a higher proportion of the cDBG. Eventually, decision node proportion stabilizes.

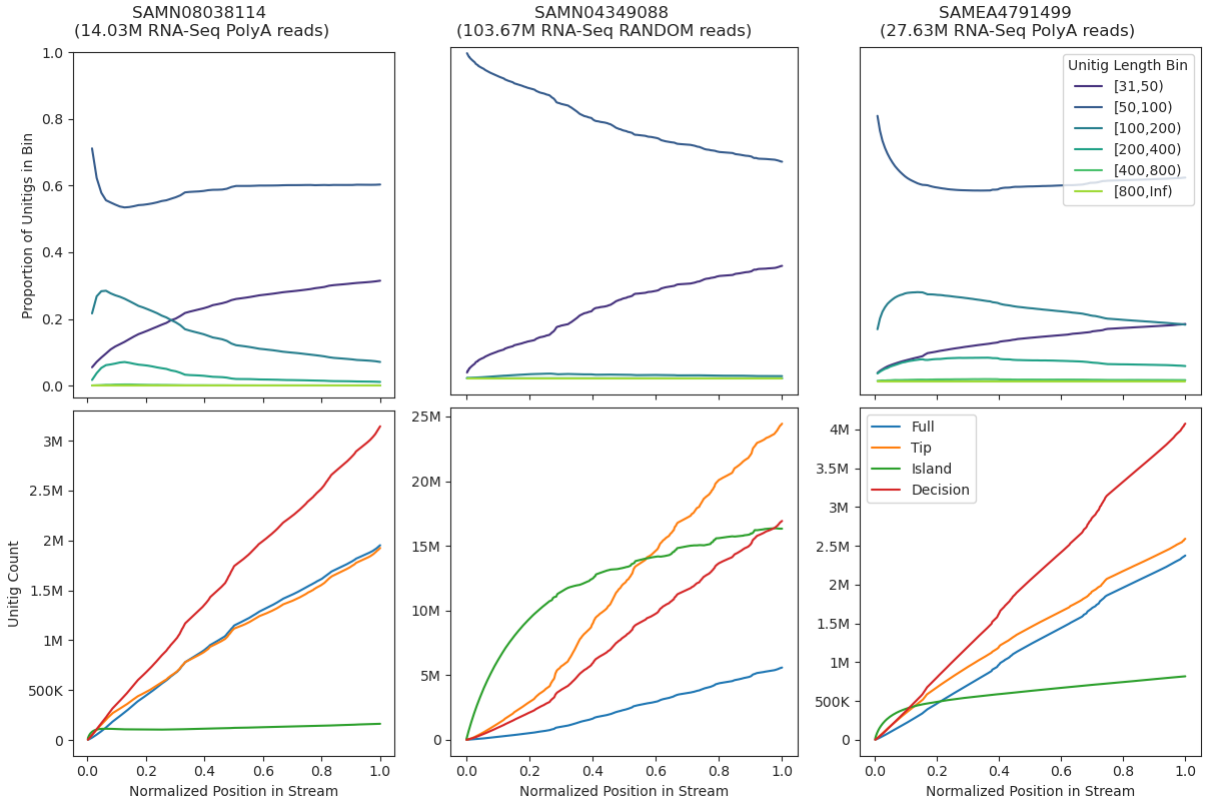

Figure 1.4: **Dynamic metrics of the compact de Bruijn graph during construction.** cDBG metrics for transcriptomes of three human samples. The left and right columns are Poly-A selected samples of 14-28M sequences; the middle column is a large rRNA-depleted sampled (see (51) for its details). The horizontal axis is the normalized position within the stream. In the top row, we show a compressed distribution of unitig lengths, binning unitigs into six categories, with the smallest possible unitig naturally being of size K. In the bottom row, we show the absolute counts of unitigs of four different types based on connectivity: "full", where both ends are decision k-mers; "tips", where one end is a decision k-mer; "islands," which are connected to nothing; and "decision," which correspond to decision k-mers. See 1 for definitions of all node types.

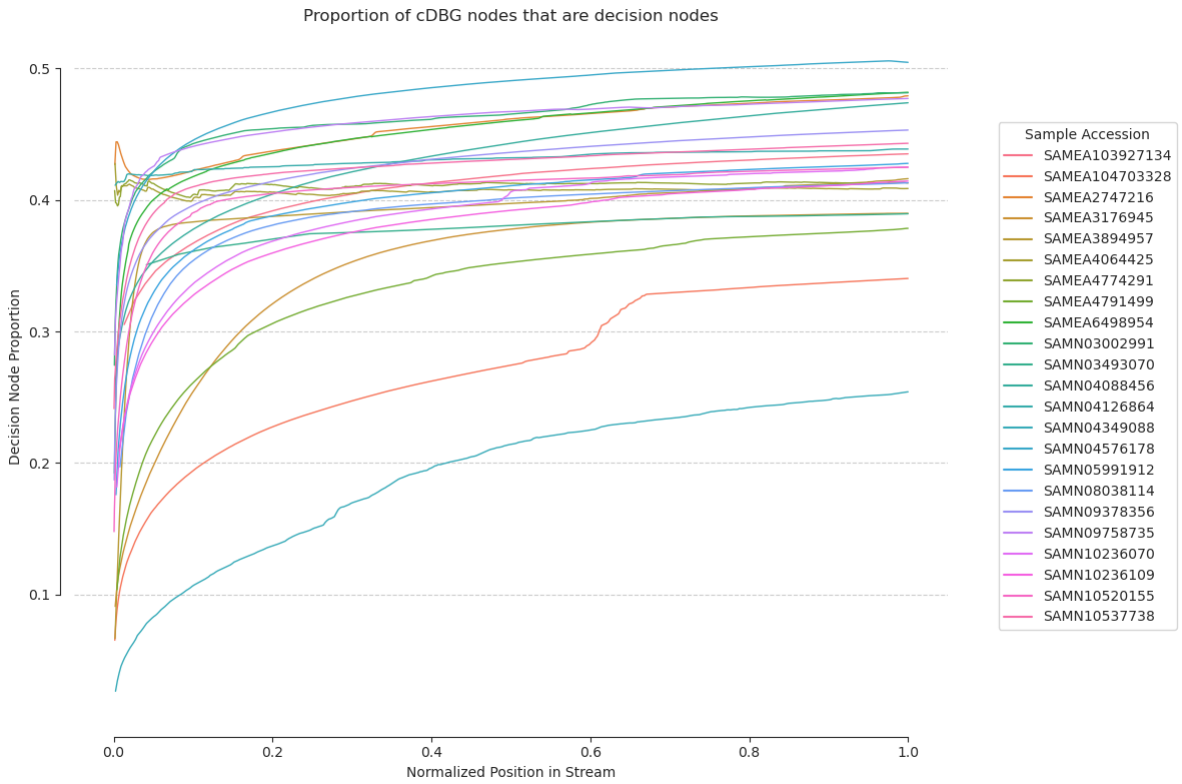

Figure 1.5: **Decision node saturation as a proportion of all nodes in the cDBG.** The proportion of decision nodes was calculated by dividing the number of decision nodes at the given time by the summed number of decision and unitigs nodes. Position in stream is the k-mer position and normalized by the end position of each sample.

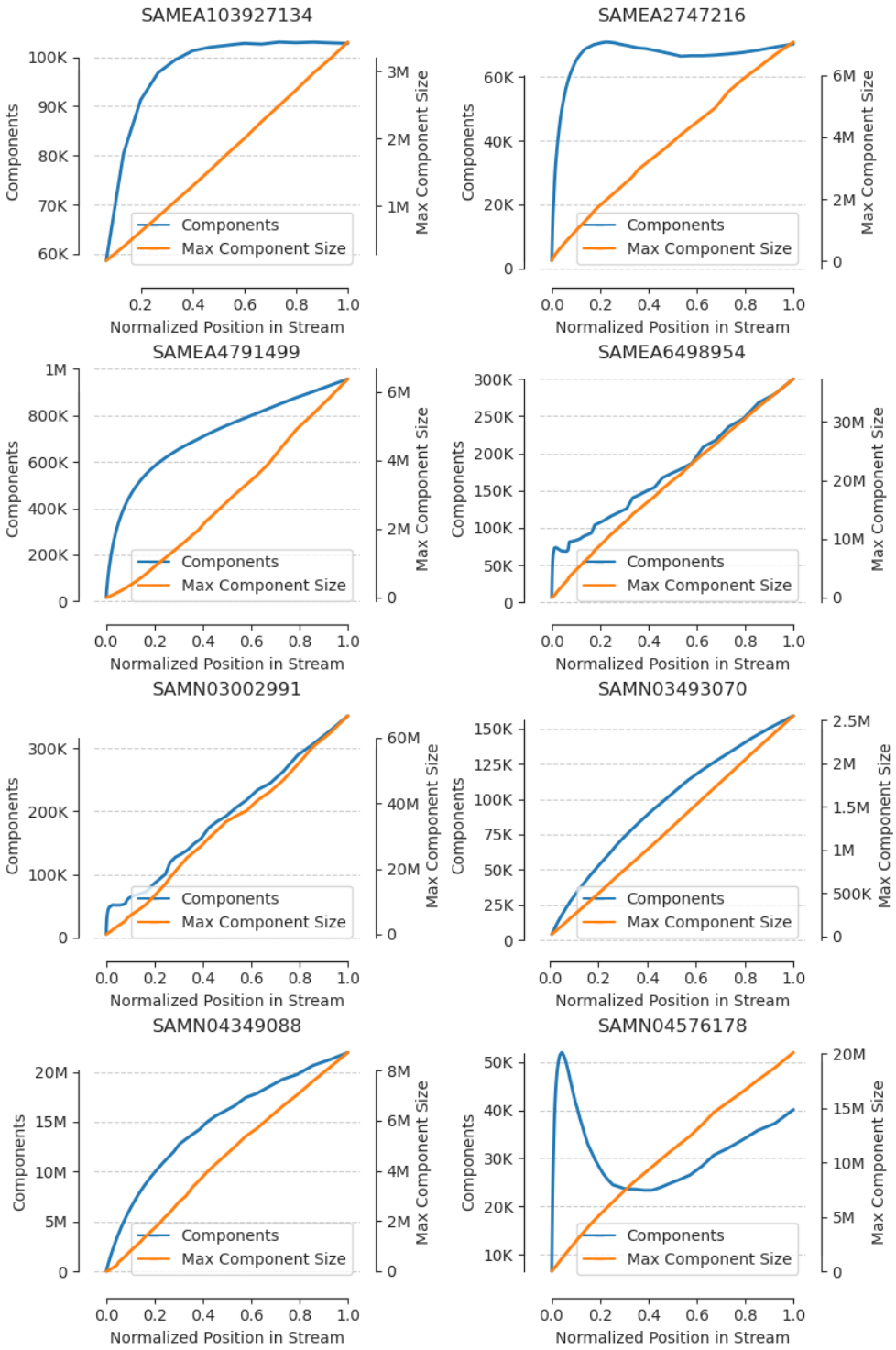

Figure 1.6: **Number of components and growth of largest component in transcriptomic samples.**

Streaming Compaction Can Participate in Pure-streaming Pipelines

Another key advantage of streaming approaches is the ability to build analysis pipelines with minimal intermediate data. Aside from the size of the raw reads themselves, which can be considerable, many programs output hundreds of gigabytes of intermediate files, reducing the accessibility of those methods where storage space is at a premium. For some analyses, streaming construction of the cDBG can avoid storing the input reads entirely: reads can be streamed directly from the SRA or ENA to streaming-enable quality control software such as Trimmomatic (52) or FASTX (53) for trimming and filtering, optionally through a pre-filter such as digital normalization (49, 54), and then compacted. To support these workflows, `goetia` implements a set of streaming filters: currently, a solidity filter which passes through reads with a set proportion of their k-mers with counts over a solid threshold, and a reimplementation of digital normalization using our library's efficient sequence parsing and storage backends.

Fig. 1.7 shows the results of a small pure-streaming compaction pipeline on the same three human RNA-seq samples as Fig. 1.4. These samples were run through the `goetia` streaming solidity filter with the resulting sequences piped directly into the streaming compactor. Briefly, the streaming solidity filter inserts the k-mers from input sequences into a de Bruijn graph with a Count-min Sketch backend, a fixed-memory approximate membership query (AMQ) datastructure with a known false-positive rate. Sequences pass the filter if a given proportion of their k-mers have a multiplicity greater than a given threshold. Here, the solidity threshold was set at 2 observations (`-C 2`) with a required proportion of 0.8 (`-P 0.8`). Filtering for solid sequences reduces the number of erroneous k-mers and is a common simplification tactic for compaction and assembly.

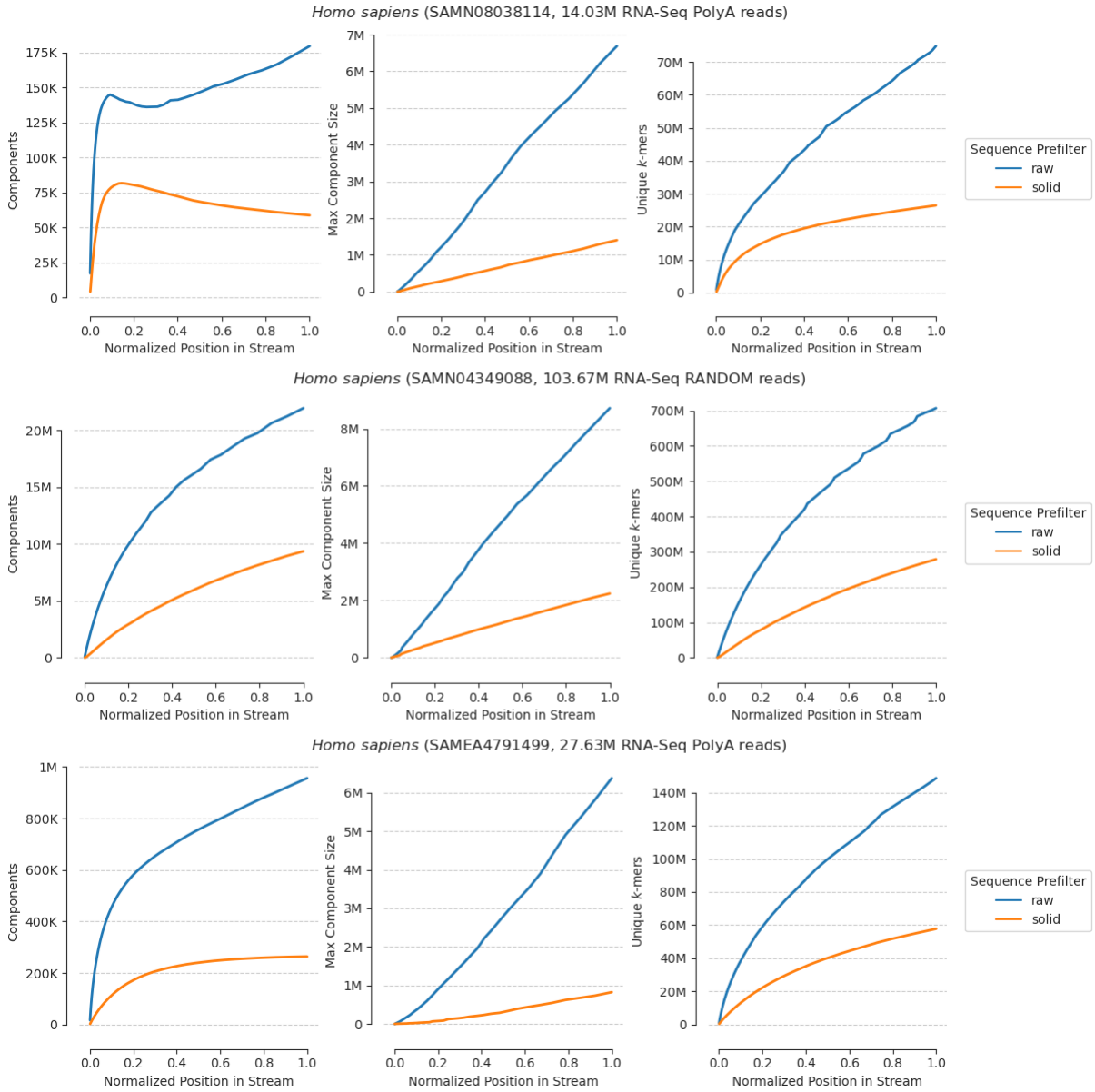

Figure 1.7: **Component metrics of three *Homo sapiens* RNA-seq samples in solidity pipeline.** RNA-seq data was compacted in a streaming manner, first with no pre-filter, and then with the `goetia filter solid` pre-filter. The sequences that passed the solidity filter were piped directly into the compaction program. These three samples represent a variety of coverage levels and two different selection methods.

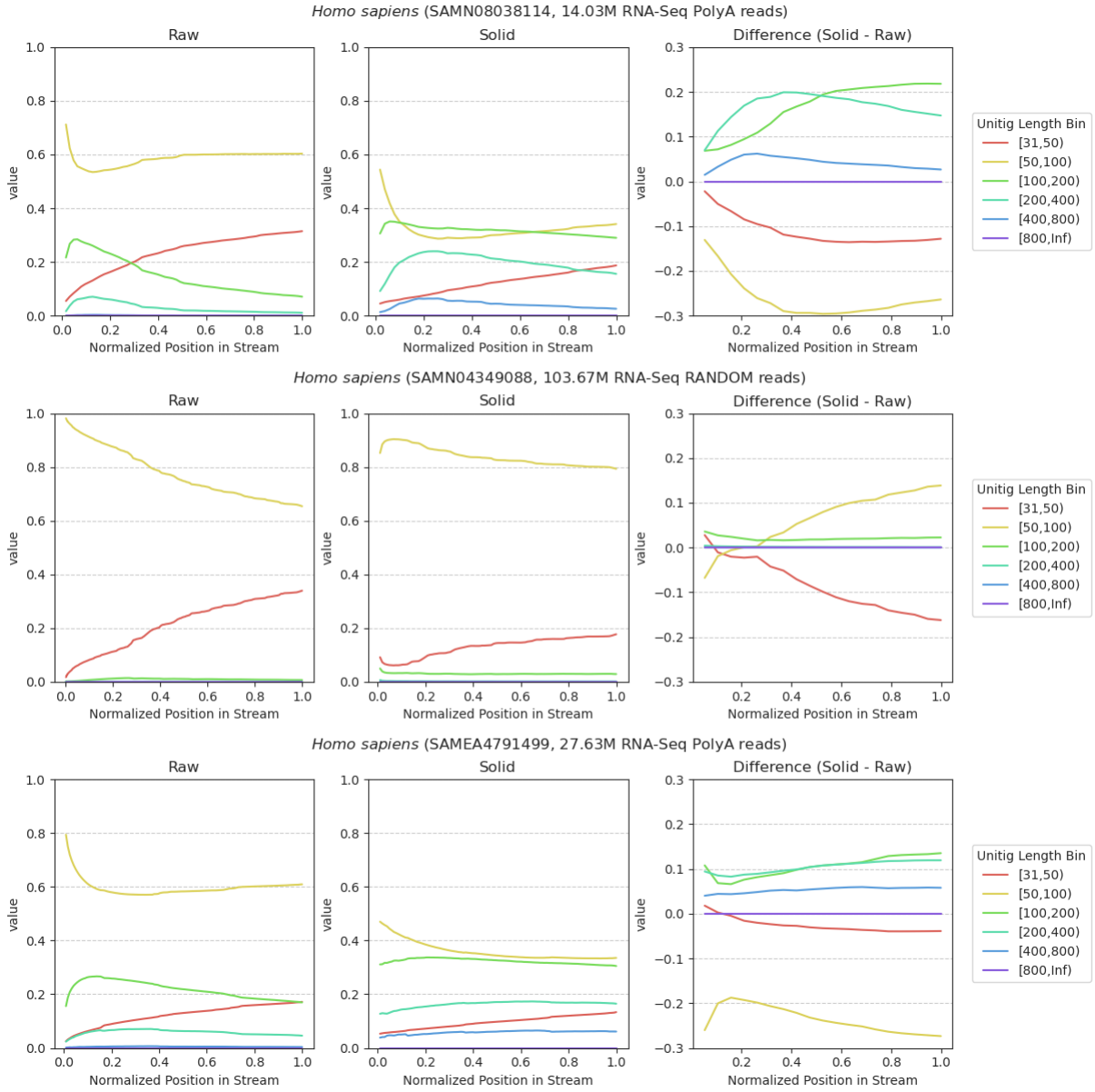

Figure 1.8: **Fragmentation metrics of *Homo sapiens* RNA-seq samples in solidity pipeline.** RNA-seq data was compacted in a streaming manner, first with no pre-filter, and then with the `goetia filter solid` pre-filter. Columns one and two show unitig length proportions of the raw and solid-filtered samples, respectively. The third column shows the difference between the solid and raw proportions at normalized time points: negative values here mean that solid filtering reduced the proportion at that point in the stream, and positive values mean it increased it.

Here, we make several observations: first, that solidity filtering drastically reduces the number of components and unique k-mers in the assembly graph; second, that it helps alleviate the number of nodes connected to a single, large component often observed in RNA-seq samples, particularly in those with Poly-A selection; and third, that the slopes of the component unique k-mer accumulation are increasingly flattened and demonstrate more "saturation" behavior when compared to raw sequence. The latter observation can be explained by both the removal of single-occurrence k-mers introduced by sequencing error and the domination of RNA-seq samples by very low-abundance sequence. Note also how the maximum component size of the middle, rRNA-depleted sample remains on par with the two Poly-A selected samples, even while having orders of magnitude more individual components.

Fig. 1.8 revisits the unitig length histograms of the three human samples. In all cases, solidity filtering reduces the proportion of the shortest unitigs (those of length k to length 50) while increasing the proportion of of longer unitigs; in other words, cDBG fragmentation is reduced. In the rRNA-depleted sample in particular, the comparative proportion of the smallest unitigs, compared to the raw sample, reduces as the stream is consumed. In absolute terms (middle column), this proportion stabilizes compared to the raw sample (left column). We expect the proportion of length $50 - 100$ unitigs to drop and stabilize throughout a stream, as these are unitigs which will, when first observed, be "seeded" by individual reads – that is, the read length is less than 100, and further legitimate sequence within the same component should serve to either extend these unitigs linearly or split them with decision k-mers caused by splice variants. We also observe that all classes of unitigs demonstrate more stabilizing behavior, with relative proportions becoming more correlated over time, compared to their raw counterparts; the better, higher-coverage regions of the assembly graph are more dominant in solid-filtered

samples, with low-coverage and erroneous sequence mostly skipped.

Discussion

We introduce a novel streaming approach for the construction of the compact de Bruijn graph. We describe that methodology of a prototype implementation of this approach which attempts to minimize graph traversals during the mutation of existing unitigs, allowing for useable streaming compaction on samples of "reasonable" size. Our approach values avoiding disk access in particular, trading for higher memory usage; instead, we help alleviate memory constraints by enabling integration into streaming pre-filtering pipelines which help reduce the number of unique k-mers in a stream. We see two clear avenues for future work: first, on reducing memory requirements by fully embracing AMQ data structures as k-mer storage backends, as our reference implementation chose exact backends for simplicity; and second, enabling multithreading through the use of our partitioned storage backends, described in Appendix A. AMQ backends introduce a tradeoff between accepting erroneous unitigs as a result of traversals through paths created by false-positive k-mers and increasing the complexity of the implementation by tracking these false positives in a streaming manner. We believe that the latter solution is made tractable by an extra filter in the decision k-mer induction step, but have avoided the implementation in this work in favor of exploring dynamic assembly graph behavior.

We further show that streaming compaction allows a new suite of tools for analysing the effects of coverage and genomic complexity on assembly graph construction. Future work will explore sub-linear approaches to compaction making use of this saturation behavior. Chapter 2 dives into this property in context of streaming sketch algorithms, and provides a possible sub-linear pre-filter compaction that could be added to a pipeline including

error correction, solidity filtering, and in-silico normalization. Streaming approaches are inherently amenable to this sort of flexibility in data processing pipelines; avoiding intermediate state, in combination with low-memory approaches enabled by sketching and AMQ methods, makes parameter sweeps that add and remove pipeline stages more tractable for the average bioinformatician.

While we have focused on second-generation high-throughput sequence technologies, we believe streaming compaction could eventually find use when integrated with recent-generation single-molecule real-time sequence technology. In particular, we hope to someday explore sub-linear compaction approaches that communicate with SMRT instruments live. As SMRT sequencers gain market share and increased software tooling, research in streaming methods will necessarily need to follow in order to keep up with the data deluge and help alleviate the complexities of deploying sequencers both in the field and in clinical settings. We hope this work provides inspiration on what is possible with these technologies.

Chapter 2

Streaming Signature Saturation

Chapter Authors Camille Scott and Luiz Irber

Introduction

In the previous chapter, we showed that streaming de Bruijn graph compaction opens new avenues for exploring assembly graph architecture. In particular, we can directly explore architecture as a function of coverage by tracking changes to the graph over time. Many metrics tend toward stability as more sequence is consumed, suggesting an avenue for sub-linear approaches. Sub-linear algorithms are those which produce a result without examining an entire data stream. Such algorithms, in general, are concerned with *property testing*: producing a yes or no answer as to whether input data satisfies some constraint (55). For most such problems, an exact solution is clearly impossible in sub-linear time; however, with fields where data generation outpaces data analysis, quickly producing approximate solutions can be preferable to slowly producing exact solutions.

The "data deluge" of genomics has made the field a prime candidate for approximate and probabilistic approaches, with many such methods developed and improved upon in recent years (56, 57). Sketching approaches, especially those built on MinHash, have received considerable attention for the ability to efficiently classify and cluster many large samples in low memory.

In this work, we perform a preliminary exploration of sketching for intra-sample comparison. We focus on the stability of sketches during the data stream, where refer to high stability as saturation. The authors of Mash point out the potential of comparing a stream to a known database and terminating after meeting a classification threshold (58). Here, we explore the potential of comparing a series of sketches of a stream to themselves during construction. Our expectation is to observe saturation behavior in these streams, opening the door for a sub-linear method without the need for a reference database. We implement a set of tools for streaming sketch analysis in the `goetia` software (45) and demonstrate its use with MinHash and FracMinHash sketches, and with a novel type of sketch tailored to saturation detection we refer to as the Draff sketch.

Methods

The Draff Sketch

The `draff` sketch is, in brief, a fixed-length vector where each position corresponds to the number of unique W-mers from a sequence sample that map to a k-mer in a Universal k-mer hitting set $UHS(W, k)$. The universal k-mer hitting set (UHS) is a set of k-mers from an alphabet Σ where any possible sequence of length W over Σ must contain a k-mer from the set (59). Universal k-mer hitting sets are an improvement over classic

minimizer schemes, in that they are pre-computed and hence fixed-length any set of input sequences, and that they are smaller than minimizer schemes over the same W and k. The `draff` sketch takes advantage of this fixed-length property to produce compressed representations that are comparable between samples. Each k-mer in the set is assigned a position in the sketch vector and a storage backend; the backend can be any k-kmer data structure that can count unique members, with our reference implementation supporting exact sets, Bloom filters, and HyperLogLog (60) counters. For each sequence s_t in the sequence stream S, each W-mer from s_t is assigned to a k-mer from the hitting set, with the k-mer with the smallest hash value being chosen if there is mere than one. The W-mer is then added to the counting backend associated with the assigned k-mer. The sketch vector is computed by querying the number of unique W-mers associated with each k-mer in the UHS.

The size of the sketch depends on the parameters W and k. Sketches built from a larger underlying UHS (which will be those with larger values of k) will have a higher resolution; that is, were theoretical hitting sets be developed with values up to $W = k$, the resulting sketch would converge to a bit vector with length equal to $\|\Sigma\|^W$, or in other words, a position for every W-mer. Practical values of k for which we have precomputed hitting sets are $k = \{7, 8, 9, 10\}$ with W values up to 200. For example, the UHS with $W = 31$ and $k = 9$ has $41,346$ hitting k-mers and hence the draff sketch has this same length.

W-mers and k-mers are hashed simultaneously using a rolling hash scheme (39). To efficiently track the minimum k-mer for determining a hitting k-mer, we use an ascending minima scheme (61). When using full set backends such as the Bloom filter or exact sets, the data structure is inherently a partitioned de Bruijn Graph; this partitioning scheme also allows for straightforward multithreading support.

`sourmash` MinHash Sketches

For comparison, we make use of the MinHash sketch for genomics (58) as implemented in the `sourmash` package (62), as well as the FracMinHash sketch further built on in `sourmash` (63). MinHash sketching is related to shingling and minimizer schemes. For some sketch size N, length k, and random hash function f, the k-mers from sequences $s_t \in S$ of a sequence stream are hashed with f and the N hashes with lowest value kept; hence, these methods are sometimes referred to as bottom sketches. The resulting set of size N is the MinHash sketch, and sketches can be compared to each other with the Jaccard similarity methods. The closely related FracMinHash extends MinHash to non-fixed sizes. Instead of keeping the bottom N hashes, a cutoff hash value proportional to the size of the hash space is chosen, and all hash values less than this cutoff are kept. Thus, the FracMinHash grows in proportion to the input data, and represents a random sample of k-mers with probability of the cutoff value divided by the maximum hash value. While this means FracMinHashes have unbounded growth, it also allows implementation of additional operations such as containment queries.

Timing and Distance Metrics

As our methods rely on measuring changes in our underlying sketches throughout a stream, robustly determining when and how to make these measurements is fundamental. The fundamental unit of time in our framework is the observed k-mer: time t is incremented for each k-mer in the stream. As such, a single sequence fragment s covers an $\texttt{len}(s) - K + 1$-length time window in the stream. Using k-mers instead of sequences makes time deltas comparable between samples with different sequence lengths and input sequences over varying size, for example, those that have been subjected to quality or

abundance trimming. Timing of distance measurements and windowed operations is performed at a parameterized frequency interval I. The time t is compared to the previous recorded time t_p; when $t - t_p \geq I$, the queued measurement operations are performed and t_p is set to t. The implementation of this system is done at the parsing level. When a new sequence $s_i \in S$ is parsed, and its k-mers consumed by the underlying analyzer (sketch, dBG, filter, compactor, or otherwise), t is incremented by $\texttt{len}(s) - K + 1$ and the prior comparison made. If I is exceeded, parsing yields to metrics operations. In this system, the actual length of measurement deltas will differ from I by some positive value ϵ; however, $I \gg L$, generally exceeding $500,000$, so ε is trivial. Excepting a small ε allows sequence consumers to benefit from the entire connected path of k-mers from a fragment s when necessary.

Let the time points corresponding to these intervals be referred to as $i_0 \dots i_j \dots i_N$, where N is the k-mer time length of the stream, and the sketch associated with these time points as $A_0 \dots A_j \dots A_N$. During streaming sketching, the distance functions $\delta(A_j, A_{j+1})$ are called between sketches at successive intervals, and the result d_j added to the distance list $\Delta = \{d_0 \dots d_j \dots d_{N-1}\}$. When our sketch is a `sourmash` Min-hash sketch, δ is $1 - Jaccard$, where $Jaccard$ is the Jaccard similarity. For the `draff` sketch, δ can be any standard distance function: we commonly use the Euclidean, cosine, correlation, and Canberra (64) distances.

Results

Rolling Signature Distances Display Saturation Behavior

Fig. 2.1 demonstrates saturation behavior of FracMinHash sketches on a transcriptomic sample taken from the MMETSP project (65, 66). The y-axis shows the Jaccard distances d_j between the sketches at measurement intervals i_j and i_{j-1}. Each line represents a different measurement interval length I, that is, successively lower measurement frequencies. Clearly, successive comparisons require normalization by sampling interval; expected distance between successive sketches is necessarily conditioned on the size of the underlying set of sequences (transcripts or chromosomes) and the error rate of the sequence instrument. For all sampling intervals, the successive distances (similarities) converge toward 1: the further into the stream, the more new k-mers within a sampling window are dominated by sequencing error. The middle plot compares each the sketch at each time point i_j to the final sketch at point t_N. Here, we hope to see the distance curve climb steeply, then become more linear upon saturation; that is, its derivative should become constant. In the rightmost plot, to help remove the effects of erroneous k-mers, we instead compare to a sketch built from an assembled and processed transcriptome from the sample. Here, the curve much more dramatically flattens, but does not approach 1. Visually, we can see that the transition occurs at a similar point in the stream as in the middle plot, which lends support to our saturation hypothesis.

Fig. 2.1 showed FracMinHash sketches, which, while allowing greater sensitivity, have unbounded size. For streams of sequence fragments, as opposed to assembled genomes, this property becomes problematic, with sketches growing to unwieldy sizes due to low-coverage and erroneous k-mers. Figs. 2.2 and 2.3 show distance curves for regular

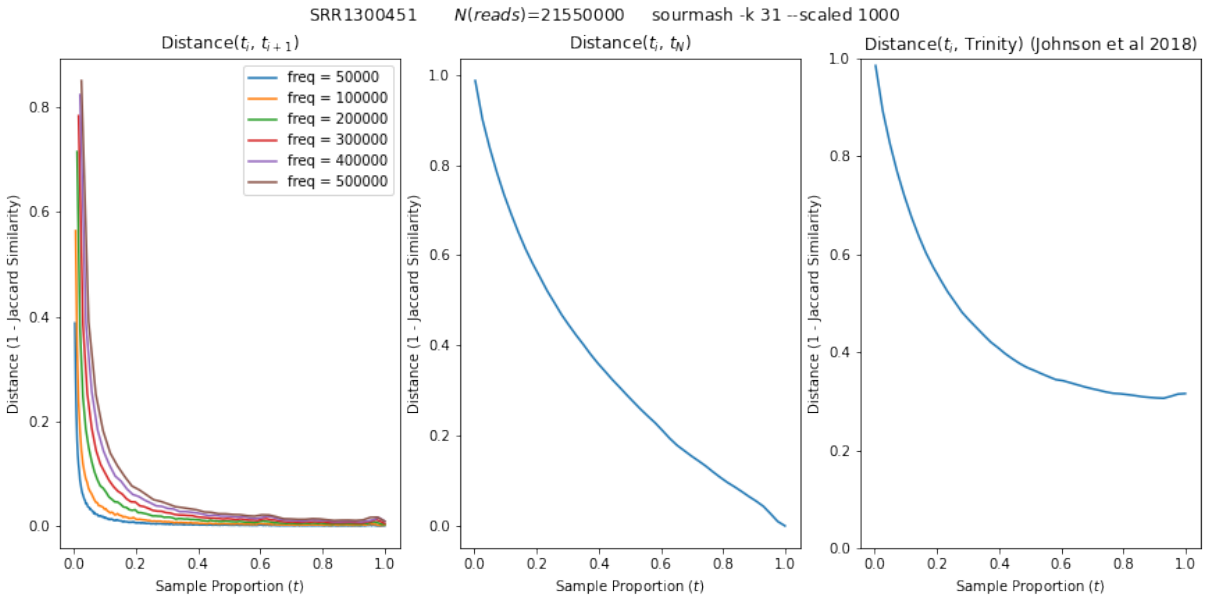

Figure 2.1: **Sketching distance curves showing saturation of a transcriptomic sample.** Sourmash FracMinHash sketch at $scaled = 1000$, $K = 31$, computed on a 21.5 million read transcriptomic sample. *Left*: Distance ($1.0 - Jaccard$) between successive sketches, computed at varying sampling intervals. *Middle*: Distances between sketches at given time intervals and the final sketch. *Right*: Distances between sketches at given time intervals and the downstream sequences as assembled with Trinity in (66).

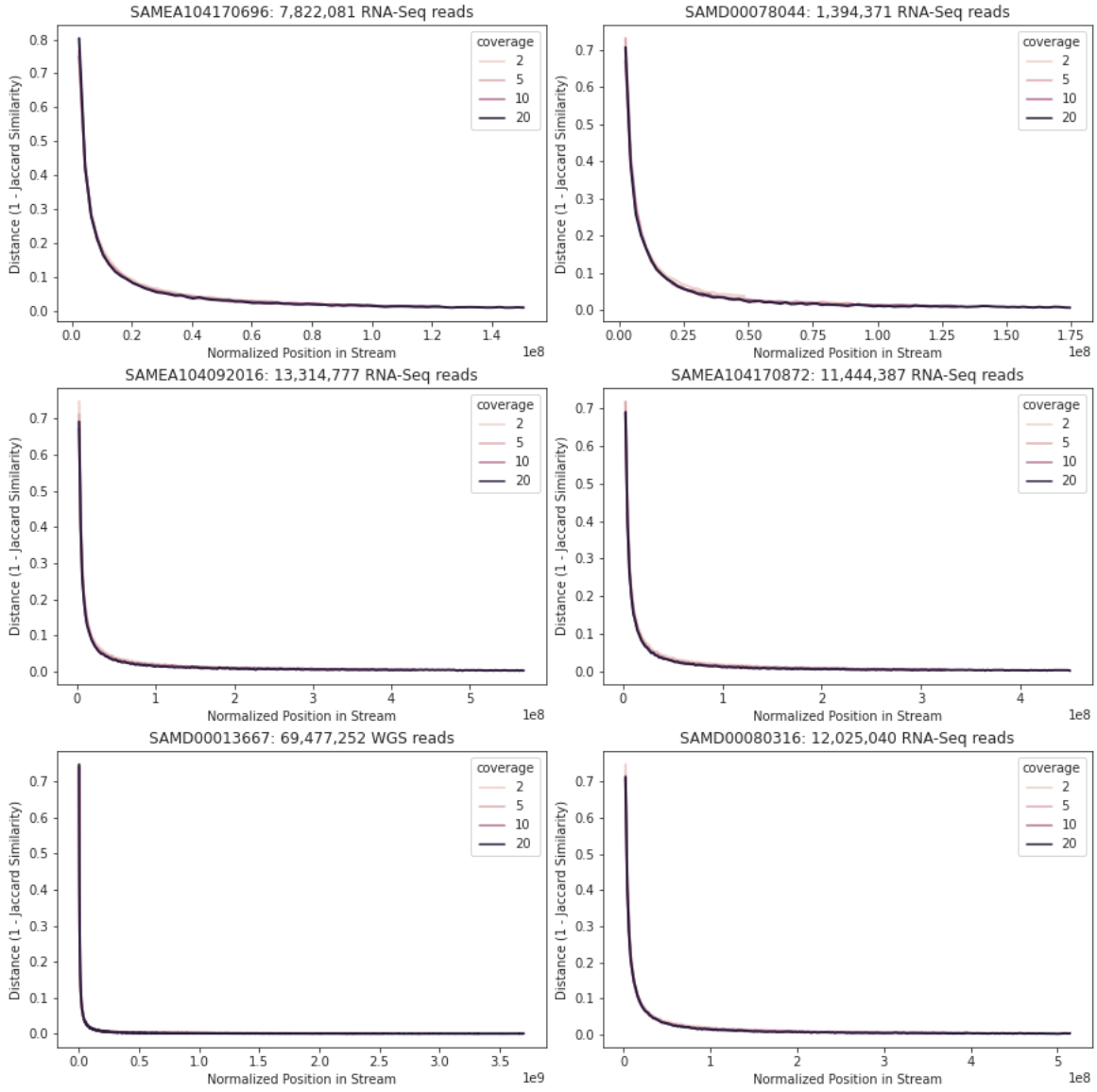

Figure 2.2: **Streaming Jaccard Similarity of digitally-normalized sourmash-MinHash signatures.** MinHash sketches computed with `sourmash` using $K = 31$ and $N = 25000$. Distances are 1.0-Jaccard similarity between successive sketches, computed at an interval of 2000000 k-mers. Digital normalization is used as a pre-filter to the sketch stream, with each curve representing a difference coverage cutoff C. Higher coverage thresholds admit more sequences, causing the lower-threshold streams to saturate more slowly at a given *relative* position in the stream.

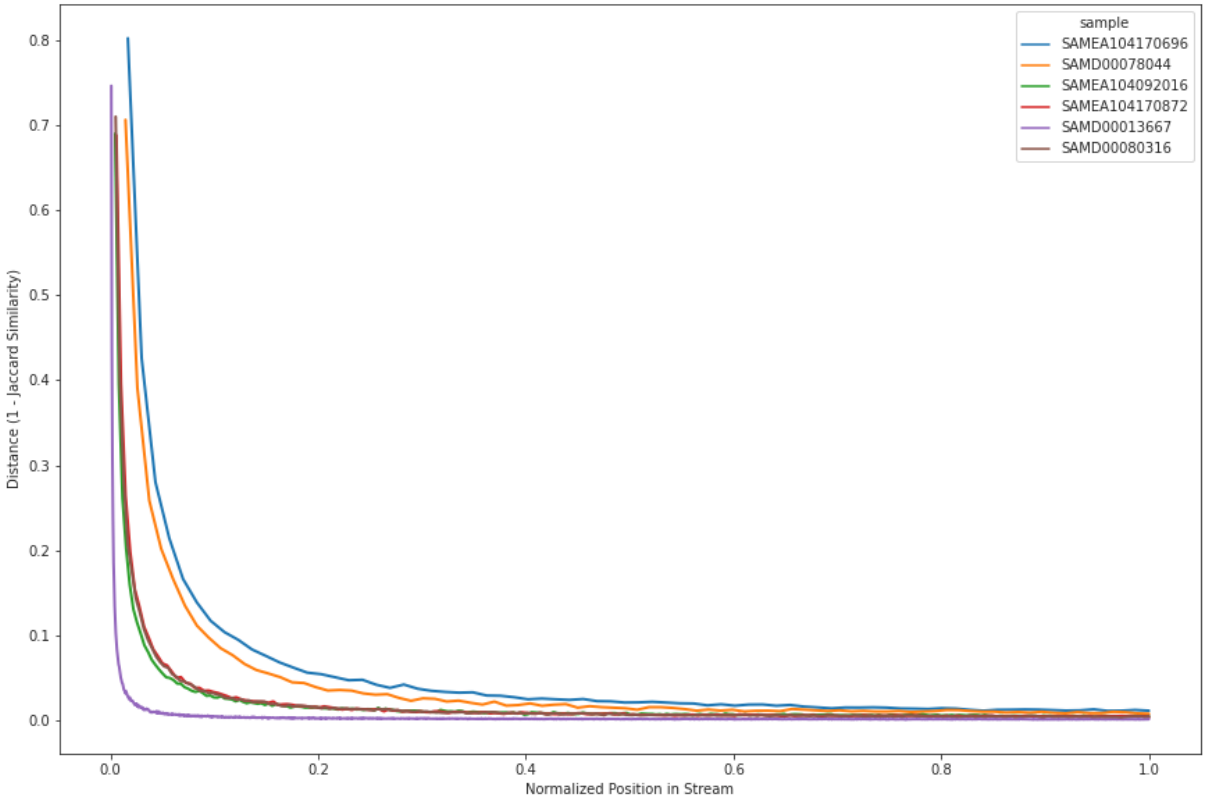

Figure 2.3: **Streaming Jaccard Similarity of sourmash-MinHash signatures.** Distance curves of the same samples as in Fig. 2.2 at digitally-normalized maximum coverage of 20.

MinHash sketches with size $N = 25000$. As we can see, the standard MinHash sketch also shows saturation behavior. The lower-coverage RNA-Seq samples, for example the top-right sample with approximately 1.4m sequences, show saturation at relative positions much further into the stream, and also show a wider spread between the different coverage levels of digital normalization. For the $70m$ sequence whole-genome sample of *C. elegans*, in the bottom left, change in distance stabilizes much sooner. We posit two reasons for this: firstly, the (average) coverage of this sample is much higher than the relatively smaller RNA-seq sample; and secondly, the genomic sample has more unique k-mers than the RNA-seq sample, with k-mer cardinality many orders of magnitude higher than the size of the MinHash sketch. The latter point is of great importance, as clearly, very large MinHash sketches will be necessary to detect significant differences in the sketch throughout the stream.

The Draff Sketch Shows Improved Saturation Behavior over MinHash

draff sketches show similar saturation behavior when successive distance measurements are made. Fig. 2.4 shows this on a selection of WGS and RNA-seq samples. The left panel shows the cosine distance between sketches rapidly flattening early in the stream, as in the MinHash examples. However, distances with the draff sketches can be measured at many more significant digits than the MinHash sketches, providing more sensitivity; to help highlight behavior later in the stream, on the right, we show these distances on a *log* scale. We can see that distances continue to reduce at a small, but steady, rate, with some exceptions. Periodicity suggests some of the samples may have technical artifacts, with the *log* scale amplifying their effects.

Fig. 2.5 shows distances between the sketch at a given point and the final sketch of the

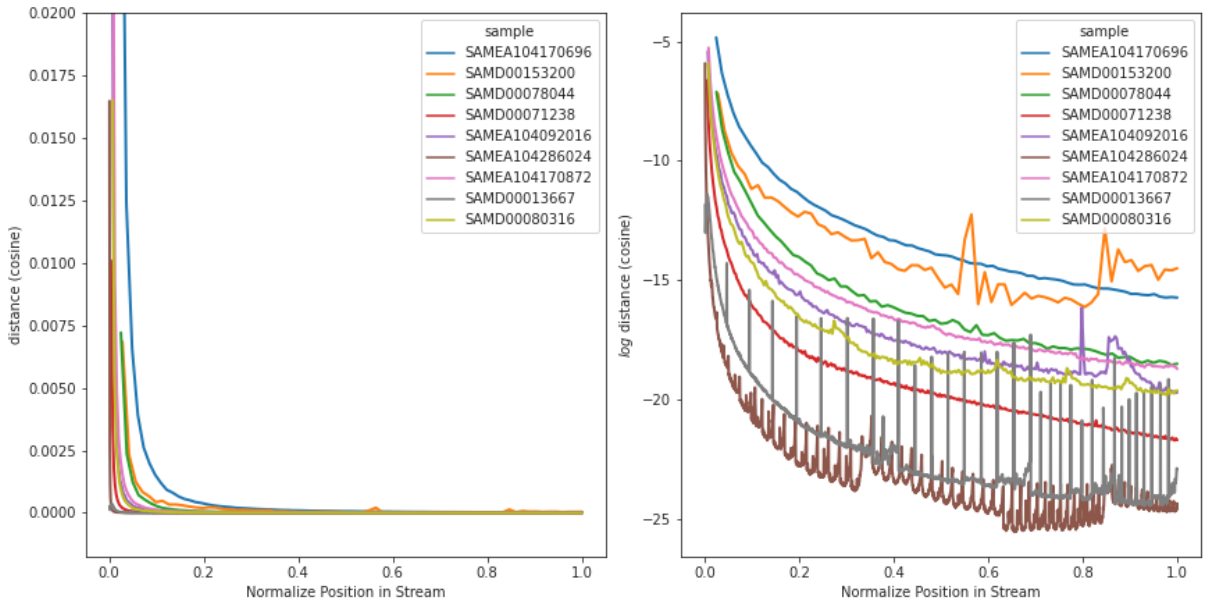

Figure 2.4: **Streaming cosine distance of `draff` sketches.** Draff sketches computed with $W = 31$, $K = 9$. On the y-axis, cosine distance calculated between successive sketches; on the x-axis, normalized time in number of k-mers. On the left are the exact cosine distances; on the right, we show them on a *log* scale. As in 2.3, these are digitally normalized to a maximum coverage of 20.

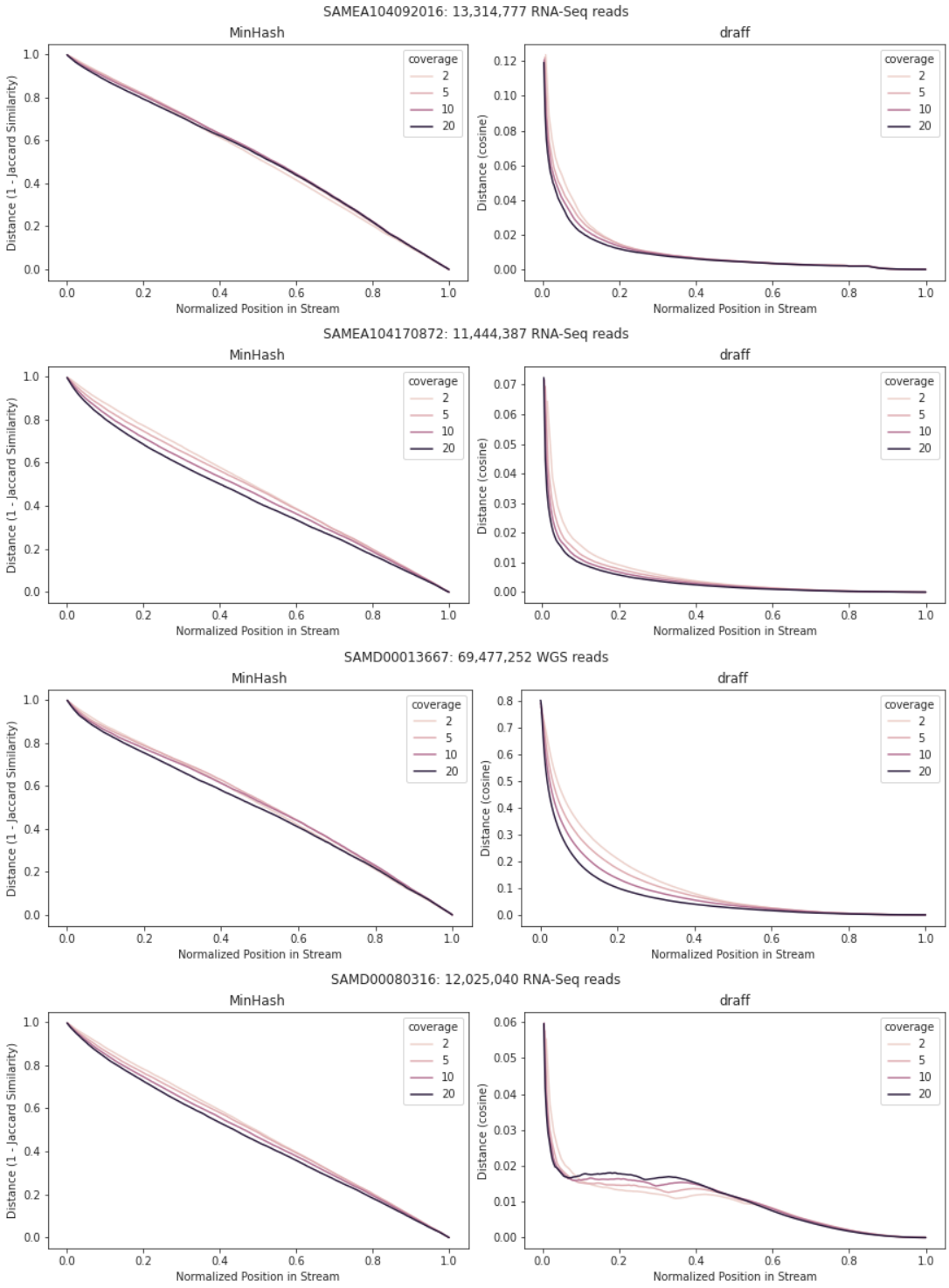

Figure 2.5: **MinHash and draff streaming distances as compared to representative sketches.** draff sketches with $W = 31$, $K = 9$ and MinHash sketches with $K = 31$, $N = 25000$ are compared. The distances are between the sketch at a given time point and the final sketch in the stream, as in the middle plot of Fig. 2.1.

stream for MinHash and draff. Here, the draff sketch shows its greater utility for the saturation problem. For a sketch to best capture stability *de novo*, we expect distance from the sketch of the final stream to have an inflection point that occurs when the underlying sampled sequence has been observed and new fragments transition to mostly introducing error. The signal for the MinHash sketches is dominated by error, and they fail to capture this effect. We posit that MinHash sketches of sufficient size should act similarly to the draff sketch; however, MinHash sketches are bound by worst-case $\mathcal{O}(log(n))$ insertion time, which imposes scaling challenges on MinHash, whereas insertion into the draff sketch is always $\mathcal{O}(1)$ (for the implemented exact and AMQ backends).

The Draff Sketch Enables Novel Dimensional Analyses of Assembly Graphs

Because the Draff sketch is, intuitively, a compressed "snapshot" of k-mer space, in the form of vector of feature counts as opposed to the set of specific k-mer hashes in MinHash sketches, we are able use some more traditional numeric approaches with them. In Fig. 2.6, we view the stream of sketches as a matrix with each row a draff sketch corresponding to measurement times $i_0 \dots i_N$. We then perform a Principle Component Analysis (PCA) along a length-10 sliding window of the rows, and extract the explained variance of the top two components for each window. In this way, we reduce the change in the compressed "k-mer space" over time to a 2-dimensional representation. This offers a different view on the saturation behavior. Rather than distances between successive sketches decreasing and then stabilizing, we instead see the variance of a slice of sketches decreasing; moreover, a reduction in explanatory power of the top 2 components can be interpreted as the locations of new unique k-mers becoming more diffuse as we progress through the stream. This provides further support for our saturation model where changes in the stream post-

saturation are dominated by erroneous k-mers.

Discussion

We introduce a new sketch tailored for streaming saturation detection, which we call draff, leveraging universal hitting sets, and explore sketch saturation on it and MinHash sketches. Both methods show stability of distances when successively compared in a stream, suggesting a method for calling saturation of a sample in a streaming context *de novo*. The draff sketch has especially desirable properties for the task: a high level of granularity in the face of large numbers of unique k-mers; constant-time insertion even for large sketches; and canonical stability in self-similarity.

There are many avenues to explore here in the future. There is the need to formally describe the relationship between sketch size, sequence error rate, and sample size in the context of self-similarity. There is necessarily a multiple-testing problem to correct for which has not been studied here, as well as more specific limitations on comparing highly-similar MinHash sketches: for MinHashes of size N, the shortest distance we can compute between them is $1 - (N - 1)/(N + 1)$, and whether this distance has any significance is dependent on the size of the underlying genome. draff sketches avoid this problem by accounting for all unique k-mers, but better determining these limitations for this usage is of interest. For draff, comparing the effectiveness of different distance metrics would be highly valuable, as here we only explore the cosine distance.

The previous issues directly relate to developing functions for calling saturation. Potential functions include: a specific distance cutoff, under which we immediately terminate; a distance cutoff where we terminate after the mean distance of a sliding window meets the threshold; measuring the variance of distances over a sliding window, wherein we try to

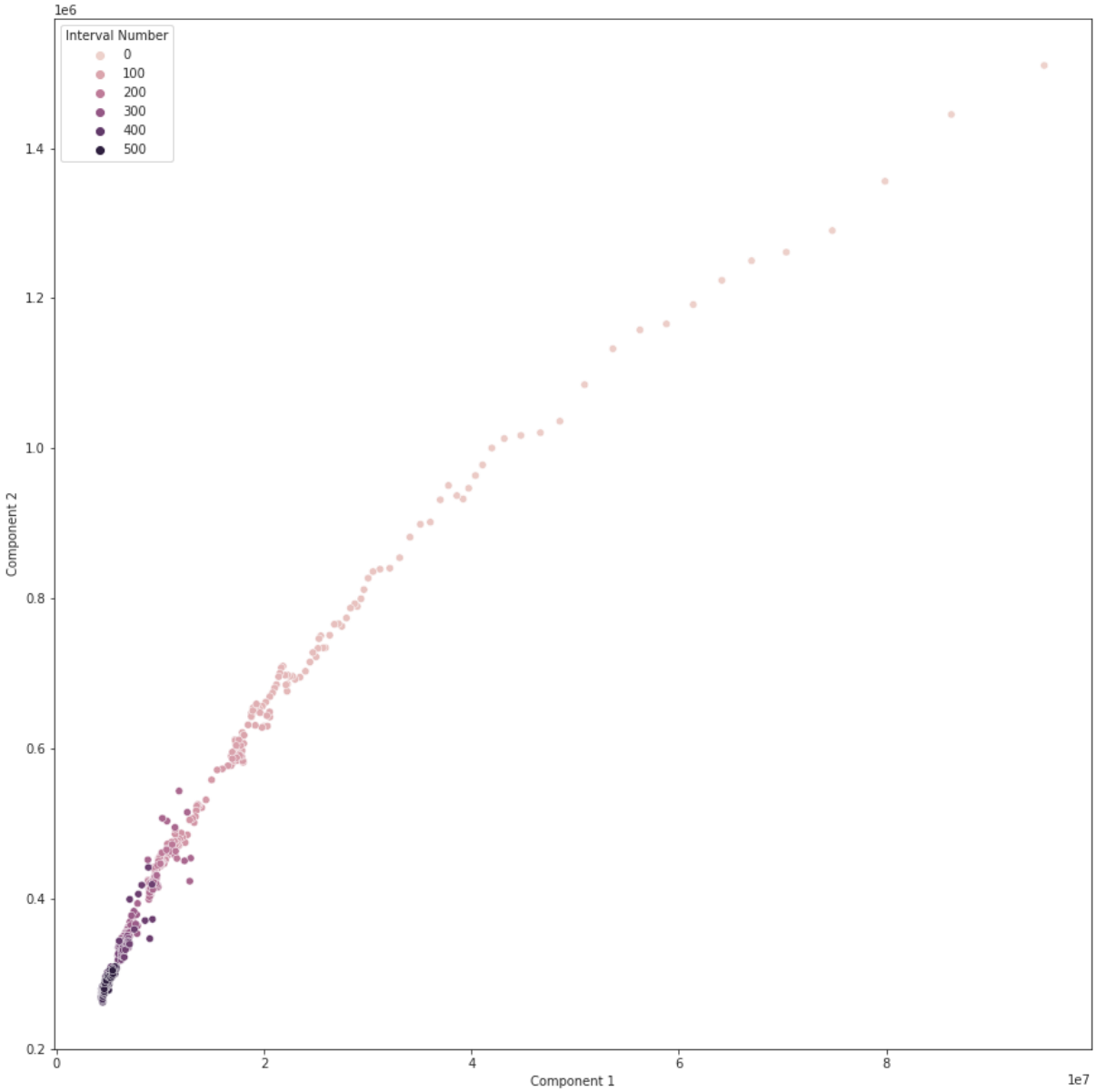

Figure 2.6: **Top-two principle components of sliding window over streaming draff sketches.** Rather than computing distance metrics between successive pairs of sketches, Principle Component Analysis was run over a sliding window of 10 sketches at a time and explained variance of the top two components extracted. Each point is a PCA window proceeding through the sample.

more directly measure the stability of the sketches; or other statistical moments on the stream of distances. We briefly explored the behavior of the draff sketch in W-mer space via Principal Component Analysis, which opens up another potential set of methods for calling saturation.

Finally, there is the question of integrating this streaming method with the streaming compaction described in Chapter 1. Streaming saturation detection is a natural pre-filter for streaming compaction, providing a potential sub-linear solution for construction of an inexact compact de Bruijn graph. The downstream effects on the resulting cDBG are of particular interest, as well as the results of including the method along with sequence error correction, solidity filtering, and digital normalization in pure-streaming pipelines. Future work would focus on these pipelines holistically.

Chapter 3

dammit 2.0: an open, accessible transcriptome annotator

Chapter Authors Camille Scott and N. Tessa Pierce

dammit was a project that grew out of work from early on in my PhD on de novo RNA-seq assembly of the sea lamprey, *Petromyzon marinus*. The project eventually lead to a publication making use of version of the resulting transcripts and annotations (67), though a publication concerning the re-assembly itself was abandoned. Being somewhat unsatisfied that the project did not produce a first-author publication, and having produced a significant amount of code for munging annotations, I decided to put it together into a piece of software, that I somewhat affectionately named dammit as a nod to the many inherent irritations involved in downstream sequence analysis. Some weeks into putting the software together, and close to a useable prototype, a Friend Of The Lab who will go unnamed tweeted the project's GitHub page and suggested people use it, resulting in a userbase and the necessity to maintain it. Eventually, it was used in a

number of projects, and another member of the lab, N. Tessa Pierce, took on some work in maintenance and improvement. With an established userbase and some difficulties with the underlying workflow system, we decided to rewrite a version 2.0 of the project using Snakemake. Considering the amount of work that went into writing and maintaining the software, and its apparent usefulness to some in the community, I decided to include it as part of this dissertation.

Abstract

dammit is a automated de novo transcriptome annotation pipeline designed with the express purpose of being easy to install and configure. While a number of annotation protocols and software pipelines already existed when dammit was first conceived, they tended to be unapproachable to beginners. dammit automates database download and preparation, as well as dependency installation; it need only be installed as a Python package via PyPI or conda, after Snakemake is used internally to manage environments for each annotation step. Annotation is achieved through homology searches against protein databases, hidden Markov model and covariance model profile searching for protein domains and non-coding RNAs, gene model prediction, completeness assessment, and orthology searches against user databases. dammit uses mostly easily available software such as hmmer and Infernal, occasionally making use of less well-known software in the pursuit of performance. The resulting annotations are collected and merged into a Generic Feature Format Version 3 (GFF3) file, with annotation summaries added to FASTA headers. dammit also exposes a suite of simple command line tools for file formation conversion, homology filtering, metrics, and FASTA munging.

Introduction

RNA-seq is now a foundational tool for genomics. Reference-based RNA-seq enables researchers to characterize gene expression levels under differing conditions, while de novo assemblies of RNA-seq data provide a useful resource for researchers attempting to characterize the gene content of an organism, especially in non-model systems (68). Importantly, due to its cost-effectiveness, transcriptome assembly is an accessible option for labs with limited resources; whereas a complete de novo genome assembly requires high coverage and often long reads to properly characterize genic content, a transcriptome assembly is often sufficient to discover expressed transcripts of interest. Annotation is generally the first step downstream of assembly, and annotators often make for complex and unwieldy software. Due to their tendency to rely on a variety of third-party software and databases, they can be difficult to install and configure, especially for those with limited bioinformatics expertise. To make matters worse, many annotators trade performance for familiarity in their supporting software; for example, while many aspiring bioinformaticians may be familiar with BLAST (69), its relatively lackluster performance when comparing many transcripts to a large reference database can be a burden to labs with limited computational resources.

dammit addresses these issues by fully embracing a philosophy of openness and accessibility. The authors contend that truly open scientific software must also maximize *accessibility* and *availability*: tools with cumbersome installation processes, cloud-based solutions that rely on stable centralized hosting, and tools that only run on a limited selection of operating systems and hardware are compromised in their support of open science. Hence, we rely on performant, well-tested open source third-party tools; implement our core software in a well supported language environment; and use the stable,

easily accessible `conda` ecosystem for distribution. dammit provides a solution for users wanting automated local installation, good performance, and ease-of-use.

Methods

Implementation

dammit is written in Python and uses tools within the ecosystem. The first version was written using `pydoit` (70), with each step in the database preparation and annotation process decomposed into a `pydoit` task, allowing for easy dependency tracking and resume capability. Version 2 is a complete rewrite using Snakemake (71); while the underlying pipeline remains mostly the same, the `pydoit` tasks have been converted into reusable Snakemake wrappers and the command line interface (CLI) wraps the Snakemake CLI. Snakemake is widely used within the field and has a rapidly growing community. Furthermore, it enables first-class support of High Performance Compute (HPC) system schedulers, a collection of reusable wrappers to make tasks portable, and automated per-task environment management.

The dammit CLI exposes a number of subcommands. The core `run` subcommand wraps Snakemake and executes the pipeline which is defined in the associated snakefiles. A `config` subcommand allows reporting and editing configuration options globally and per-project; all configuration is done through standard YAML files, with global configuration and temporary storage adhering to the XDG Base Directory Specification (72). The remaining subcommands implement various format parsing and transformation, filtering, and statistical functions, which are then leveraged in the Snakemake pipeline. For example, `dammit rename-fasta` parses an input FASTA file and renames the headers in

a standardized format, optionally with a user-supplied regular expression; this subcommand is also the first task run by the annotation pipeline. This allows users to make use of the internal `dammit` tooling independently of annotation.

File parsing is implemented in the companion library `ope` (73). This library provides parsing support for GFF3, HMMER and Infernal tabular output, MAF, and TransDecoder FASTA, returning the parsed data in `pandas` DataFrames (74, 75) for use within the Python data science ecosystem. `dammit` provides converters to output the corresponding DataFrames as GFF3.

dammit is primarily developed and tested on 64-bit Linux and distributed via conda. We currently require a minimum of Python 3.7.

All documentation is written in Markdown and managed with MkDocs. Documentation can be accessed at the dammit Github Pages site (76).

Operation

The core subcommands of the program are `dammit run annotate` and `dammit run databases`. These commands support a `--pipeline` argument which specifies which subset of the possible databases to use; the available options are "default," "quick," "full," and "nr." The "default" pipeline uses all the databases and programs described the following sections except for `uniref90` and `nr`. "full" adds `uniref90` to this pipeline, and `nr` uses the NCBI non-redundant protein database, which is particularly large. "quick" excludes HMMER, Infernal, and the `OrthoDB` conditional reciprocal best LAST tasks.

`dammit run database` will download and prepare the databases necessary for the given `--pipeline` argument. Preparation includes steps such as running `hmpress` on Pfam-A or `lastdb` on FASTA collections. `dammit run annotate` runs the selected pipeline on

the input transcriptome; this task will fail if the associated database preparation has not been performed.

Hardware requirements vary mostly with the databases used for annotation and their preparation. Database preparation for a default run can be completed in under 16 GB of RAM, with the most intensive step being LAST's `lastdb`; preparation including uniref90 or `nr` uses considerably more RAM. Advanced users can make detailed adjustments to LAST's parameters with the config file to reduce RAM and run-time.

Annotation is less RAM intensive than the database preparation step, instead requiring more CPU time. gnu-parallel (77) is used to help scale HMMER, Infernal, and LAST. A default run splits the input file across multiple cores on a single machine. Advanced users can pass the appropriate Snakemake profile arguments for cluster execution following the Snakemake documentation (78). A typical run can be completed in less than 12 hours on a machine with 8 cores; once again, adding `uniref90` or `nr` greatly increases the required time.

Use Cases

Database Preparation

Annotation necessarily requires a number of databases. dammit automates the retrieval and preparation of its databases with the `dammit run databases` subcommand. Running this subcommand checks for their presence by default; with the `--install` flag, they will be downloaded and prepared. dammit uses the following databases:

Pfam-A: Profile hidden markov models for searching protein domains with `hmmer` (79).

USE CASES

Rfam: Covariance models for searching RNA secondary structure with Infernal (80).

OrthoDB: Hierarchical collection of protein ortholog groups (81).

BUSCO: Collections of core genes for major domains of life. dammit can use any of the available BUSCO databases; we recommend choosing one from a clade closely related to your target species (82).

uniref90: A comprehensive collection of most known proteins clustered at 90% sequence similarity (83).

nr: The entire non-redundant protein sequence database from the NCBI (84).

Databases are stored in the users home directory by default (in `~/.local/share/dammit`), and the location can be changed either with the `--database-dir` flag or by setting a `DAMMIT_DB_DIR` environment variable.

Annotation Pipeline

`dammit run annotate` takes a "standard" FASTA file as input. First, the FASTA headers are renamed and a mapping between the original and new names is output; this avoids the issues that some programs have with certain characters in the header lines. Then, a basic set of sequence statistics are gathered, such as the number of transcripts, the k-mer redundancy, and the GC percent. BUSCO (82) is run in transcriptome mode to assess completeness using whichever BUSCO database is specified by the user. `TransDecoder.LongOrfs` (85) then finds open reading frames and produces a set of basic protein translations; these translations are used to search against Pfam-A with hmmer

(86), which are then fed into `TransDecoder.Predict` to get a final set of gene model predictions. Infernal (87) is used to search Rfam for ncRNAs. OrthoDB and optionally uniref90 or nr are searched using LAST (88) If the user supplies any custom protein databases, they are searched for orthologs using Conditional Reciprocal Best Hits (90) as implemented in `shmlast` (91).

An example annotation run on the file `cdna.fa` with the user-supplied protein database `pep.fa` might look like so:

```
dammit run annotate cdna.fa --user-databases pep.fa --busco-group eukaryota
--n_threads 2
```

This will create a new folder called `cdna.fa.dammit`, which can be changed with the `--output` flag. This folder contains the files `cdna.fa.dammit.gff3` and `cdna.fa.dammit.fa` with the main results, along with many intermediate files. The GFF3 file contains the aggregated results, with coordinates in transcript space; this file can be furthered interrogated downstream with other analysis packages or viewed with an explorer such as IGV (92). The GFF3 uses terms compliant with the Sequence Ontology Project (93). dammit also outputs a new FASTA file with summary annotation information in each header line.

Chapter 4

shmlast: An improved implementation of Conditional Reciprocal Best Hits with LAST and Python

Chapter Authors Camille Scott

DOI https://doi.org/10.21105/joss.00142

shmlast originally grew out of the dammit annotation project. Primarily, it aims to reimplement Conditional Reciprocal Best Hits algorithm, using the python data science ecosystem and the efficient LAST aligner. It also includes the classic Reciprocal Best Hits algorithm, which seems often-implemented in hacky scripts that ride along with various annotation projects, but rarely implemented in standalone, maintained software. It was

originally a component of dammit, but the program has clear use as a standalone tool, and so was separated into its own project. The program was published in JOSS at the above referenced DOI.

Summary

Conditional Reciprocal Best Hits (CRBH) was originally described by Aubry et al.(90) and implemented in the crb-blast package. CRBH is a method for finding orthologs between two sets of sequences which builds on the traditional Reciprocal Best Hits (RBH) method; it improves RBH by finding an expect-value cutoff per alignment length, and then selecting non-reciprocal alignments which meet the minimum threshold.

Unfortunately, the original implementation uses the relatively slow NCBI BLAST+ (69), and is implemented in Ruby, which requires users to leave the Python-dominated bioinformatics ecosystem. shmlast makes CRBH available to users in Python, while also greatly improving performance by using the LAST aligner (89) for initial homology searches. Other improvements include outputting the list of cutoffs generated by its model along with a plot of the decision boundary to aid in quality control, as well as using gnu-parallel (77) to parallelize execution across multiple cores or nodes in a cluster environment.

Methods

RBH is a relatively old method for determining orthologs between two sequence databases. Orthology is distinguished from sequence similarity by descent; while two sequences with high similarity are likely to have structural of functional homology, they are orthologous if they share a common ancestor sequence. It is difficult to know

for sure whether two sequences are orthologs, but orthologous groups of sequences are critical for a variety of analyses surrounding structure, function, and evolution, and particularly, for annotation. As such, many methods have been developed for separating high-similarity alignments from their orthologous counterparts. RBH is the regal elder of these methods, and although it is simplistic compared to newer clustering and graph-based methods, it remains in wide use due to its low false-positive rate and ease of implementation. It is performed as follows: given two sets of sequences A and B, sequences $a_i \in A$ and $b_j \in B$ are Reciprocal Best Hits if b_j has the highest scoring sequence alignment in B for a_i and a_i has the highest scoring sequence alignment in A for b_j. a_i and b_j then have a high probability of being orthologs.

While this method works well for finding orthologs between two sets of proteins from different species, it is less effective for annotating newly assembled transcriptomes from existing protein databases. Transcriptomes are confounded by alternative splicing, causing several transcripts to share subsequences, which may prevent RBH detection between a translated transcript and its protein, even when an orthology relationship exists. Aubry et al., and this implementation, circumvent that problem by first using the reciprocals to establish a score cutoff for each alignment length, and then keeping *any* alignment which passes that cutoff. This prevents alignments with high-likelihood of being orthologs based on sequence identity from being discarded due to the high specificity of RBH.

Performance

shmlast benefits immensely from the use of LAST over BLAST. It scales well by using gnu-parallel, and can be distributed across clusters for particularly large runs.

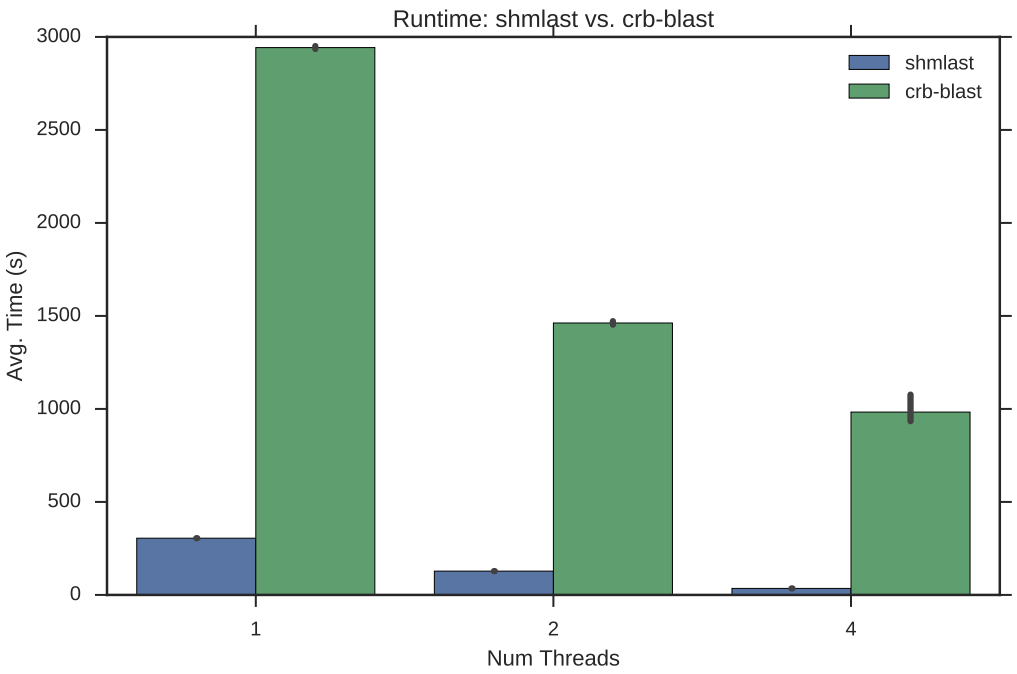

Figure 4.1: Performance comparison with *Schizosaccharomyces pombe* as the query transcriptome and *Nematostella vectensis* as the target proteome.

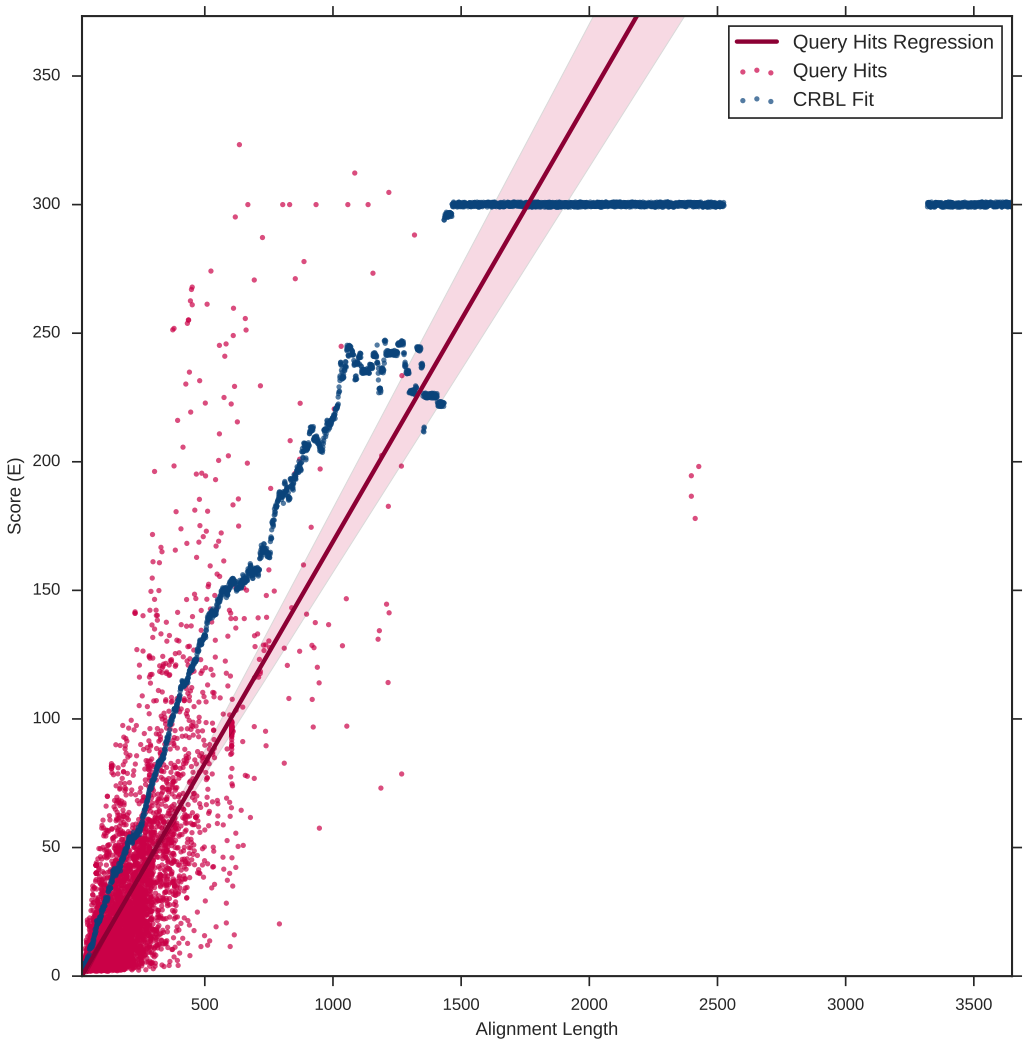

Figure 4.2: CRBH model generated from the performance comparison. Hits with scores above the blue dotted line will be kept.

Chapter 5

SuchTree: Fast, thread-safe computations with phylogenetic trees

Chapter Authors Russell Y. Neches and Camille Scott

DOI https://doi.org/10.21105/joss.00678

SuchTree was a project that grew out of Russell Neches's frustration with the scaleabiltiy of extant phylogenetic tree libraries. At the time, we were collaborating and coworking quite often, which naturally ended up with me helping writing the software. SuchTree is a prime example of the openness and accessibility goals of dammit, while also conforming to UNIX principles of software development: it does a handful of tasks, very efficiently, in a way that can be easily composed with other tools in its ecosystem. The library was published in JOSS at the above referenced DOI.

Summary

Python has several packages for working with phylogenetic trees, each focused on somewhat different aspects of the field. Some of the more active projects include :

- `DendroPy`, a multi-purpose package for reading, writing, manipulating, simulating and analyzing phylogenetic trees in Python (94)
- `ete3`, a package for analysis and visualization of phylogenetic trees with Python or command line tools (95)
- `Pylogeny`, an analytical tool for reshaping and scoring trees with GPU support via the `BEAGLE` library (96)
- The `Bio.Phylo` subpackage in biopython collects useful tools for working with common (and not so common) file formats in phylogenetics, along with utilities for analysis and visualization (97)
- The `skbio.tree` module in `scikit-bio` is a base class for phylogenetic trees providing analytical and file processing functions for working with phylogenetic trees (98)

Each of these packages allow trees to be manipulated, edited and reshaped. To make this possible, they must strike a balance between raw performance and flexibility, and most prioritize flexibility and a rich set of features. This is desireable for most use cases, but computational scaling challenges arise when using these packages to work with very large trees. Trees representing microbial communities may contain tens of thousands to tens of millions of taxa, depending on the community diversity and the survey methodology.

`SuchTree` is designed purely as a backend for analysis of large trees. Significant advantages in memory layout, parallelism and speed are achieved by sacrificing the ability to

manipulate, edit or reshape trees (these capabilities exist in other packages). It scales to millions of taxa, and the key algorithms and data structures permit concurrent threads without locks.

`SuchTree` supports co-phylogenies, with functions for efficiently extracting graphs and subgraphs for network analysis, and has native support for `igraph` and `networkx`.

In addition to the software itself, the repository includes a collection of 51 curated co-phylogenies gathered from the literature grouped into three categories by the type of ecology (frugivory, parasitism and pollination), and two collections of simulated co-phylogenies grouped by the type of simulation (independent evolution and perfect coevolution).

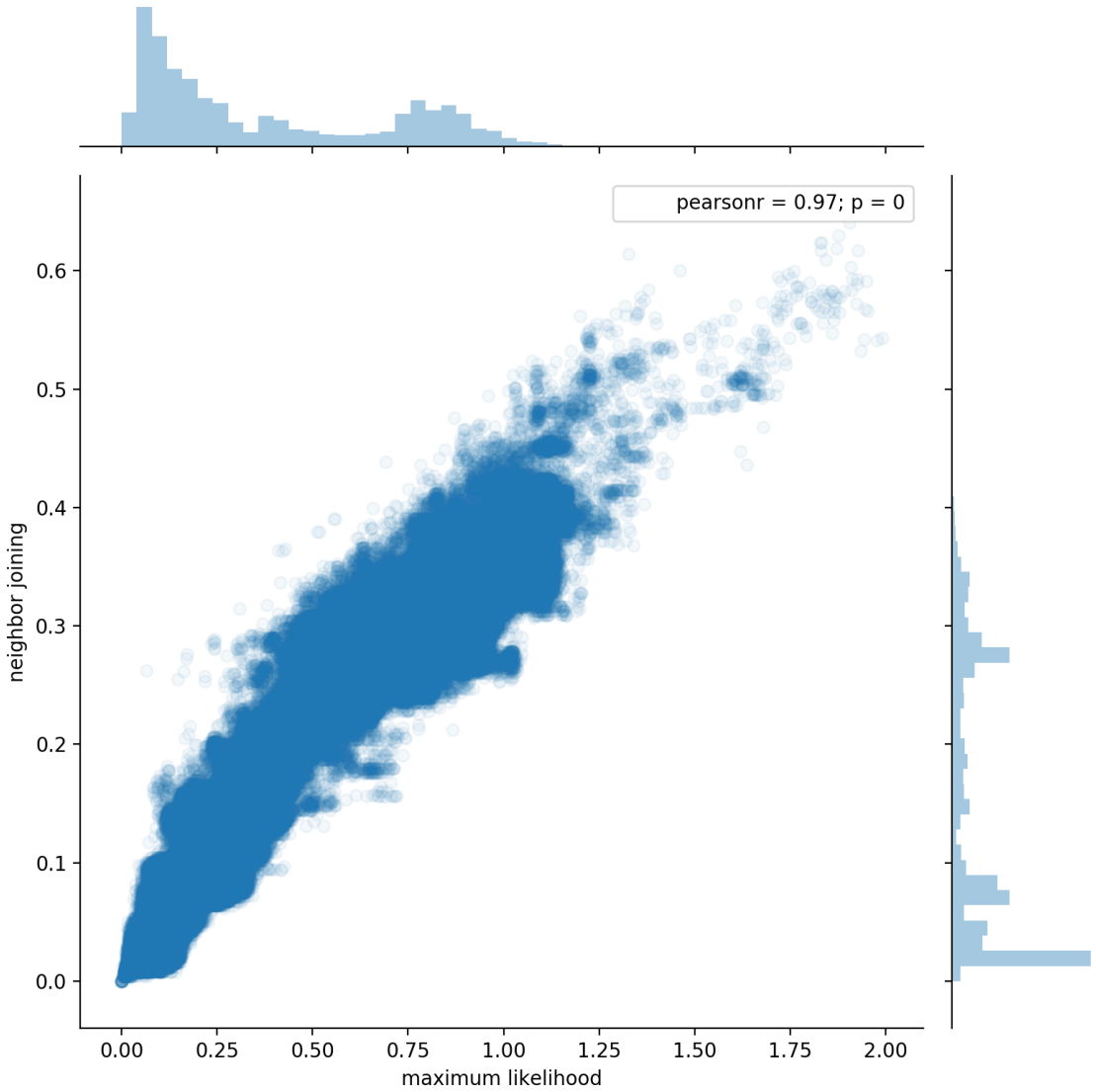

Figure 5.1: Two phylogenetic trees of 54,327 taxa were constructed using different methods (approximate maximum likelihood using `FastTree` (99, 100) and the `neighbor joining` agglomerative clustering method). To explore the different topologies of the trees, pairs of taxa were chosen at random and the patristic distance between each pair was computed through each of the two trees. This plot shows 1,000,000 random pairs sampled from 1,475,684,301 possible pairs (0.07%). The two million distances calculations required about 12.5 seconds using a single thread.

Conclusion

In this dissertation, we have demonstrated methods for steaming compaction of the de Bruijn graph and dynamic assembly graph analysis, and explored approaches for sub-linear sequence analysis with sketches. Streaming approaches offer a new perspective on *de novo* sequence analysis: firstly, by providing new ways of analyzing the structure of our sequencing data, and secondly, by broadening the ways we can build analysis pipelines as a whole. We further demonstrate several pieces of software that follow a design pattern favoring greater accessibility. `dammit` works to build a transcriptome annotation pipeline using state of the art workflow approaches, while providing commonly used utilities in the form of small command line tools. `shmlast` works to improve the accessibility of an existing tool by rewriting it to use the Python ecosystem and exposing its core functionality as a library. Both prioritize ease-of-use and teachability, and have been used in a workshop setting with novice bioinformaticians. We then demonstrate `SuchTree`, which provides an efficient phylogenetic tree library by adopting the UNIX philosophy of simple, limited-purpose tools. The core software product for this dissertation, `goetia`, provides a rich library in both the Python and C++ ecosystems, along with a set of command line tools, that is easily installable through the widely adopted `conda` package manager.

The common thread is prioritization of community: streaming methods that strive to

be workable when storage space is limited, composability through open source code and software distribution, and reproducibility through workflow systems. The document itself was written and compiled by building on the aggiedown (101) package and Manubot citation processor (102). By adopting these principles, we hope that this work can be reused, remixed, and extended in the years to come.

Appendix A

The `goetia` Library and Software

`goetia` is an open source software package implementing the methods described in this dissertation. It comprises three main components: 1. A C++ library, `libgoetia`, which implements k-mer storage and hashing, de Bruijn graph traversal, file and stream sequence parsing, sketch construction, and sequence filtering;2 2. A set of Python bindings for `libgoetia`, generated with `cppyy` (47), which expose the library in a user friendly form; 3. A Python library implementing asynchronous interaction with `libgoetia`'s various processors, the `goetia` command-line interface, and a suite of unit tests.

All code is released under the MIT license and hosted on GitHub (45). The package is distributed via the `conda` ecosystem on the `bioconda` channel (46). Due to the complexity of the build process introduced by `cppyy`, the package is not currently distributed on PyPI; distribution on PyPI may be explored in the future if the effort appears worthwhile.

`libgoetia`

The C++ library is responsible for the "heavy lifting" of the package. To enable composibility and efficiency, it makes much use of of templating and the curiously-recurring template pattern (CRTP) to avoid virtual function calls. The library is organized into namespaces by purpose.

Hashing

k-mer hashing is a core bottleneck in de Bruijn graph construction, as well as many sketching algorithms. Because of the centrality of hashing, many of `goetia`'s classes are parameterized by hashing method. Hashing capability is modularized; the basic interfaces simply accept k-mer sequences and produce integers, and additional, more advanced interfaces support efficient rolling hashing and neighbor-searching.

Hash types

The types of hash values are templated to allow composability and flexibility. The core type is `Hash`, which is parametrized with a `value_type` for the underlying integer to be used. Casting to `value_type` or calling `value()` on a `Hash` object will return the underlying integer. Generally, this is assumed to be `uint64_t`, which is the default, but larger (or smaller) integers can be substituted. Next is type `Canonical`, which like `Hash` takes a `value_type`, but actually stores two values: the forward and reverse-complement hash values. When a `Canonical` is cast to `value_type` or hash its `value()` method called, it returns to smaller of its forward and reverse-complment hash values.

Building on the two foundational hash types is `Kmer`. This is parameterized by a hash type corresponding to the interface of `Hash`, and stores a `std::string` corresponding to the k-mer associated with the hash value. Related is the `Shift`: this also takes a hash type template parameter, as well as a non-type parameter of type `bool` for its direction. A shift stores a hash value and a single `char` symbol representing the single base from the dBG alphabet Σ that was added to the $k-1$ suffix (or prefix) of the `Shift`'s predecessor.

Finally, the `Partitioned` and `Wmer` templates are used for minimizer schemes. A `Partitioned` models an underlying hash value with `hash_type` and an abritrary associated `uint64_t` partition ID, which may be calculated via minimizer, universal hitting set, or other methods. `Wmer` stores a `Hash` along with a `Partitioned` to model the hash value of a full length W-mer, the hash value of its associated minimizer k-mer, and the partitioned ID of that minimizer k-mer.

Hashers

W and k-mer hashing algorithms are built on a standard interface for roll hash functions useful for k-mer hashing. The foundation for this interface is the `HashShifter`, which takes as a template parameter a shift policy implementing the specific rolling hashing algorithm, and requires the following methods be defined on this policy:

- `get()`: Return the current hash value.
- `shift_right(char, char)`, `shift_left(char, char)`: Remove a character from the current state and hash in a new character.
- `hash_base(char *)`: Hash an entire k-mer (set the starting state before shifting).
- `static hash(char *)`: Statelessly hash an entire k-mer.

`HashShifter` is also templated with an `Alphabet`, which may be one of the plain

nucleotides $\{A, C, G, T\}$, called `DNA_SIMPLE`, the plain nucleotides plus $\{N\}$, called `DNAN_SIMPLE`, and the full IUPAC defined nucleotide alphabet `IUPAC_NUCL`. The alphabet classes define their own validation and complementation methods.

There are specialized policies for: the Lemire rolling hash algorithm, for both `Hash` and `Canonical` (ie, strand-aware and canonical k-mer versions), and the rolling UHS minimizer shifter, which yields `Wmer` objects with their associated hitting k-mers instead of plain `Hash` objects.

Finally, to help build toward de Bruijn graph traversal, there is the `HashExtender` template. The `HashExtender` has the hash state via its parent `HashShifter` while also adding a character ring buffer to track the current k-mer of its hash, which we call the cursor. It extends the `HashShifter::shift_left` and `HashShifter::shift_right` methods to update this buffer, and also implements the `left` and `right_extensions` interfaces for generating the $\|\Sigma\|$ possible prefix or suffix neighbors of a k-mer.

k-mer Storage

Storage classes implement efficient k-mer collections. All follow a base interface named `Storage`, which takes a template parameter `value_type` for the type of the hashed k-mers to be stored. The interface defines the following core methods:

- `insert(value_type)`: Insert an item into the collection, incrementing the associated counter if supported, and returns `true` if the item is newly observed.
- `insert_and_query(value_type)`: Perform an `insert` and return the count after insertion.
- `query(value_type)`: Retrieve the existence or count of the item.
- `n_unique_kmers()`: Report the number of unique elements in the collection.

- `reset()`: Remove all elements from the collection and return to default state.

Collections implementing the interface also define the following traits:

- `is_counting`: Whether the collection supports abundance counts or only behaves as a set.
- `is_probabilistic`: Whether the collection tracks items exactly or with error.
- `bits_per_slot`: Number of bits used per item, for bit-vector based data structures.
- `params_type`: Tuple for the parameter types needed to instantiate the class.
- `default_params`: Default parameters to use if not supplied.

`goetia` provides a number of data structures implementing this interface.

Basic Storage Classes

Bloom filter: `BitStorage` A Bloom filter is a probabilistic, sub-linear space data structure for set membership testing. It uses reduced space at the cost of a small, predictable false positive rate, with no false negatives. Memory usage is fixed; it is parameterized by size of its underlying bit vector and a number of hash functions, the two of which together decide the false positive rate (103). It is not possible the enumerate the members of a Bloom filter. The Bloom filter code is a modified version of that in the `khmer` (104) library.

Count-min sketch: `NibbleStorage` and `ByteStorage` A Count-Min sketch (103) is another probabilistic, sub-linear space data structure. It operates similarly to the Bloom filter, but stores an approximate count of its items instead of simple presence. `NibbleStorage` uses 4-bit counters, while `ByteStorage` uses 8-bit counters. These too are modified versions of the those found in `khmer`.

`Sparsepp` Hash Set: `SparseppSetStorage` An exact set representation using the `Sparsepp` hash table (105). `Sparsepp` is an iteration on Google's `sparsehash` (106), and provides a highly memory-efficient set with fast $O(1)$ insertions and queries.

`parallel-hashmap`: `PHMapStorage` Another exact set representation using the `parallel-hashmap` library (42). Written by the same authors as a successor `Sparsepp`, it is an essentially unilateral improvement. It uses SIMD instructions for internal probing and is optionally thread-safe. `PHMapStorage` is the recommended (and default) exact store.

Counting Quotient Filter: `QFStorage` A probabilistic sub-linear space data structure developed as a unilateral improvement over Count-min Sketches (107). It makes use of SIMD instructions to speed up rank-and-select operations and has good cache locality, the latter a significant weakness of Bloom filters and CM sketches. These advantages are achieved at a considerable increase in implementation complexity.

Partitioned Storage

The `PartitionedStorage` exposes the same interface as the basic `Storage`, but supports partitioning the underlying data into disjoint sets. The API is agnostic to the method of partitioning; all that is required is a number of partitions and an additional parameter specifying the partition in which to store an item while inserting and querying. The `PartitionedStorage` is template-parameterized with one of the formerly described storage classes, and a number of data structures equal to the number of partitions are constructed. Thus, care must be taken when using underlying storage classes with a fixed, pre-allocated amount of memory, so as to not over-allocate available system mem-

ory. The expected partitioning method for k-mers are minimizer methods, particularly the Universal k-mer Hitting Set (UHS), which is part of the hashing functionality.

Partitioned storage offers several advantages, at the cost of increased processing time spent on determining partitions. For storages with poor cache locality, an appropriately-determined partitioning methodology can reduce cache misses during query and graph traversal. Partitioning also allows fine-grained per-partition locking of the dBG, and thus efficient parallel algorithms. Both these advantages necessitate locality-sensitive partitioning methods such as minimizer and UHS schemes. UHS schemes used for partitioning are described in 2.

de Bruijn Graphs

The `goetia::dBG` and `goetia::PdBG` classes build on the `Storage`, `PartitionedStorage`, and hashing classes to implement functional de Bruijn graphs. The core `dBG` and `PdBG` interface defines:

- `insert(std::string)`: to hash and add an individual k-mer to the graph via the underlying `Storage` class.
- `insert_and_query(std::string)`: To add a k-mer and return its count post-insertion.
- `query(std::string)`: to retrieve the current presence or count of a k-mer.
- `insert_sequence(std::string`: To insert the k-mers of a sequence into the graph, along with a number of overloads for returning hashes, returning the set of new k-mers, return the vector of counts, and so on.

`dBG` and `PdBG` inherit the functionality of the `HashShifter`, `HashExtender`, and

`UnitigWalker` classes, the latter of which is described below; hence, `dBG` also is capable of traversing itself.

The `cDBG` class is templated on a `dBG` type, and hence a `HashShifter` type, and implements the storage and retrieval of unitigs in its associated de Bruijn graph. The `cDBG` class only manages the unitigs via their tips and the decision k-mers; the compacted graph is build using the `StreamingCompactor` class, which puts together all the prior functionality. Its primary public interface is simply the `insert_sequence` method, which adds a sequence to the underlying de Bruijn graph and appropriately updates the unitigs in the `cDBG` that the sequence intersects.

de Bruijn Graph Traversal

The apex of the `HashShifter` and `dBG` hierarchy is the `UnitigWalker`. It uses the functionality of the `HashExtender` to walk a `dBG` that is bound to it and extract unitigs. It reports detailed state information its primary interfaces in the form of the `Walk` class, which defines:

- `kmer_type start`: A `Kmer` representing the start point of the traversal.
- `vector<Shift> path`: A vector of `Shift` representing the path taken by the traversal.
- `TraversalState end_state`: An `enum` holding the reason a traversal was terminated.

From this information, a walk can be converted to a unitig via its `to_string` method. `TraversalState` can have the following values:

- `STOP_FWD`: There was no neighbor to traverse to (in or out-degree of zero).

- `DECISION_FWD`: Traversal stopped at a decision k-mer with in/out degree greater than one in the direction of traversal.
- `DECISION_BKW`: Traversal passed through a decision k-mer with degree greater than one in the opposite direction of traversal, and so backed off by one k-mer.
- `STOP_SEEN`: Traversal stopped because it hit a k-mer already seen in this traversal: ie, it formed a perfect loop.
- `STOP_MASKED`: Traversal stopped because it hit a k-mer in the explicit set of provided k-mers to stop at.
- `STOP_CALLBACK`: Traversal stopped because it was ordered to by a bound callback function.
- `BAD_SEED`: Traversal did not start in the first place because the start k-mer was not the underlying de Bruijn graph.
- `GRAPH_ERROR`: Some fatal error in the graph topology occurred.
- `STEP`: Traversal was able to successfully take a step (a single neighbor in the direction of traversal with no reverse decision node).

The `UnitigWalker` itself defines `filter_nodes`, which takes a list of extensions from `HashExtender` and removes those that are not in the associated graph and `in_neighbors` and `out_neighbors` to wrap `filter_nodes` with the current cursor of `HashExtender`. For movement around the graph, it defines:

- `step_left`, `step_right`: Attempt to take a single step in the given direction, and return a resulting `TraversalState` and list of neighbors.
- `walk_left`, `walk_right`: Repeatedly step in the given direction and continue until a stop state is reached, returning a `Walk`.
- `walk`: Walk bidirectionally and return both walks.

The `dBG` and `PdBG` classes inherit from this class as a mixin, and so all this functionality (and state) is also found on a given `dBG` object. `UnitigWalker` can be instantiated with an existing `dBG` or `PdBG` by passing a pointer to its constructor.

Sketches

Currently, two types of sequence sketches are supported. The first is a MinHash sketch as implemented in sourmash. `goetia::SourmashSketch` wraps a `MinHash *` exported by the sourmash `rust` core; this can be extracted from the Python MinHash object in sourmash via FFI and passed on to the `goetia` C++ wrapper. The second is `goetia::UnikmerSketch`, which is the `draff` implementation. This is a template parameterized with a `Storage` backend and a `Hash` type. Both these sketches implement `insert_sequence` as their primary public facing interface.

Sequence Parsing and Processors

`goetia` provides `FastxParser` which wraps the `kseq` FASTA/Q parser; its primary public API is the `next` method which returns a `goetia::Record` object with attributes for the sequence header, the sequence itself, and the quality score, if present. `goetia::SplitPairedReader` bundles two `FastxParser` objects and parses and validates paired samples.

`goetia::FileProcessor` is a CRTP base-class that wraps a sequence parser, iterates the steam, and handles the resulting sequences in a generic fashio. The constructor takes an integer interval for stream timing; the `advance` method iterates the parser and handles the emitted sequences until the interval is exceeded, returning the number of parsed

sequencees, elapsed time (number of k-mers), and a `bool` value for whether the parser has been exhausted.

Clients derive `FileProcessor` to implement sequence handling. `InserterProcessor` takes a class with an `insert_sequence` method as a template parameter, calling `insert_sequence` on every sequence in the stream. This is used to build and compact dBGs and update sketches. Similarly, `goetia::FilterProcessor` has a template argument that expects a class implementing a `filter_sequence` method, which should return a `bool` indicating whether a sequence passes some critera; this is used, for example, for the solid filter and digital normalization implementations.

Colophon

This document is set in EB Garamond, Source Code Pro and Lato. The body text is set at 11pt with *lmr*.

It was written in R Markdown and LaTeX, and rendered into PDF using aggiedown and bookdown.

This document was typeset using the XeTeX typesetting system, and the University of California Thesis class. Under the hood, the elements of the document formatting source code have been taken from the Latex, Knitr, and RMarkdown templates for UC Berkeley's graduate thesis, and Dissertate: a LaTeX dissertation template to support the production and typesetting of a PhD dissertation at Harvard, Princeton, and NYU

The source files for this thesis, along with all the data files, have been organised into a repository which is available at `https://github.com/camillescott/dissertation`. A hard copy of the thesis can be found in the University of California Davis library.

This version of the thesis was generated on 2022-04-05 01:40:30. The repository is currently at this commit:

The computational environment that was used to generate this version is as follows:

```
_libgcc_mutex             0.1                 conda_forge    conda-forge
_openmp_mutex             4.5                       1_gnu    conda-forge
```

_r-mutex	1.0.1	anacondar_1	conda-forge
appdirs	1.4.4	pypi_0	pypi
attrs	21.4.0	pypi_0	pypi
binutils_impl_linux-64	2.36.1	h193b22a_2	conda-forge
binutils_linux-64	2.36	hf3e587d_6	conda-forge
bwidget	1.9.14	ha770c72_1	conda-forge
bzip2	1.0.8	h7f98852_4	conda-forge
c-ares	1.18.1	h7f98852_0	conda-forge
ca-certificates	2022.2.1	h06a4308_0	
cairo	1.16.0	h6cf1ce9_1008	conda-forge
cattrs	1.10.0	pypi_0	pypi
certifi	2021.10.8	pypi_0	pypi
charset-normalizer	2.0.12	pypi_0	pypi
click	8.0.4	pypi_0	pypi
curl	7.81.0	h2574ce0_0	conda-forge
errorhandler	2.0.1	pypi_0	pypi
font-ttf-dejavu-sans-mono	2.37	hab24e00_0	conda-forge
font-ttf-inconsolata	3.000	h77eed37_0	conda-forge
font-ttf-source-code-pro	2.038	h77eed37_0	conda-forge
font-ttf-ubuntu	0.83	hab24e00_0	conda-forge
fontconfig	2.13.96	ha180cfb_0	conda-forge
fonts-conda-ecosystem	1	0	conda-forge
fonts-conda-forge	1	0	conda-forge
freetype	2.11.0	h70c0345_0	
fribidi	1.0.10	h516909a_0	conda-forge
gcc_impl_linux-64	9.4.0	h03d3576_12	conda-forge

gcc_linux-64	9.4.0	h391b98a_6	conda-forge
gettext	0.21.0	hf68c758_0	
gfortran_impl_linux-64	9.4.0	h0003116_12	conda-forge
gfortran_linux-64	9.4.0	hf0ab688_6	conda-forge
graphite2	1.3.14	h23475e2_0	
gsl	2.7	he838d99_0	conda-forge
gxx_impl_linux-64	9.4.0	h03d3576_12	conda-forge
gxx_linux-64	9.4.0	h0316aca_6	conda-forge
harfbuzz	3.1.1	h83ec7ef_0	conda-forge
icu	68.2	h9c3ff4c_0	conda-forge
idna	3.3	pypi_0	pypi
isbnlib	3.10.10	pypi_0	pypi
jbig	2.1	h7f98852_2003	conda-forge
jinja2	3.0.3	pypi_0	pypi
jpeg	9e	h7f98852_0	conda-forge
jsonschema	4.4.0	pypi_0	pypi
kernel-headers_linux-64	2.6.32	he073ed8_15	conda-forge
krb5	1.19.2	hcc1bbae_3	conda-forge
ld_impl_linux-64	2.36.1	hea4e1c9_2	conda-forge
lerc	3.0	h9c3ff4c_0	conda-forge
libblas	3.9.0	13_linux64_openblas	conda-forge
libcblas	3.9.0	13_linux64_openblas	conda-forge
libcurl	7.81.0	h2574ce0_0	conda-forge
libdeflate	1.10	h7f98852_0	conda-forge
libedit	3.1.20210910	h7f8727e_0	
libev	4.33	h516909a_1	conda-forge

libffi	3.4.2	h7f98852_5	conda-forge
libgcc-devel_linux-64	9.4.0	hd854feb_12	conda-forge
libgcc-ng	11.2.0	h1d223b6_12	conda-forge
libgfortran-ng	11.2.0	h69a702a_12	conda-forge
libgfortran5	11.2.0	h5c6108e_12	conda-forge
libgit2	1.3.0	hee63804_1	conda-forge
libglib	2.70.2	h174f98d_4	conda-forge
libgomp	11.2.0	h1d223b6_12	conda-forge
libiconv	1.16	h516909a_0	conda-forge
liblapack	3.9.0	13_linux64_openblas	conda-forge
libnghttp2	1.47.0	h727a467_0	conda-forge
libnsl	2.0.0	h7f98852_0	conda-forge
libopenblas	0.3.18	pthreads_h8fe5266_0	conda-forge
libpng	1.6.37	hed695b0_2	conda-forge
libsanitizer	9.4.0	h79bfe98_12	conda-forge
libssh2	1.10.0	ha56f1ee_2	conda-forge
libstdcxx-devel_linux-64	9.4.0	hd854feb_12	conda-forge
libstdcxx-ng	11.2.0	he4da1e4_12	conda-forge
libtiff	4.3.0	h542a066_3	conda-forge
libuuid	2.32.1	h14c3975_1000	conda-forge
libwebp-base	1.2.2	h7f98852_1	conda-forge
libxcb	1.14	h7b6447c_0	
libxml2	2.9.12	h72842e0_0	conda-forge
libzlib	1.2.11	h36c2ea0_1013	conda-forge
lz4-c	1.9.3	h9c3ff4c_1	conda-forge
make	4.3	hd18ef5c_1	conda-forge

manubot	0.5.2	pypi_0	pypi
markupsafe	2.1.0	pypi_0	pypi
ncurses	6.3	h9c3ff4c_0	conda-forge
openssl	1.1.1m	h7f8727e_0	
packaging	21.3	pypi_0	pypi
pandoc	2.9.2.1	0	conda-forge
panflute	1.12.5	pypi_0	pypi
pango	1.48.10	h54213e6_2	conda-forge
pcre	8.45	h9c3ff4c_0	conda-forge
pcre2	10.37	h032f7d1_0	conda-forge
pip	22.0.3	pyhd8ed1ab_0	conda-forge
pixman	0.40.0	h36c2ea0_0	conda-forge
pybase62	0.4.3	pypi_0	pypi
pyparsing	3.0.7	pypi_0	pypi
pyrsistent	0.18.1	pypi_0	pypi
python	3.10.2	h85951f9_3_cpython	conda-forge
python_abi	3.10	2_cp310	conda-forge
pyyaml	6.0	pypi_0	pypi
r-askpass	1.1	r41hcfec24a_2	conda-forge
r-assertthat	0.2.1	r41hc72bb7e_2	conda-forge
r-backports	1.2.1	r41hcfec24a_0	conda-forge
r-base	4.1.1	hb93adac_1	conda-forge
r-base64enc	0.1_3	r41hcfec24a_1004	conda-forge
r-bh	1.75.0_0	r41hc72bb7e_0	conda-forge
r-bitops	1.0_7	r41hcfec24a_0	conda-forge
r-bookdown	0.24	r41hc72bb7e_0	conda-forge

r-boot	1.3_28	r41hc72bb7e_0	conda-forge
r-brew	1.0_6	r41hc72bb7e_1003	conda-forge
r-brio	1.1.2	r41hcfec24a_0	conda-forge
r-bslib	0.3.1	r41hc72bb7e_0	conda-forge
r-cachem	1.0.6	r41hcfec24a_0	conda-forge
r-callr	3.7.0	r41hc72bb7e_0	conda-forge
r-catools	1.18.2	r41h03ef668_0	conda-forge
r-cli	3.0.1	r41h03ef668_1	conda-forge
r-clipr	0.7.1	r41hc72bb7e_0	conda-forge
r-clisymbols	1.2.0	r41hc72bb7e_1003	conda-forge
r-codetools	0.2_18	r41hc72bb7e_0	conda-forge
r-colorspace	2.0_2	r41hcfec24a_0	conda-forge
r-commonmark	1.7	r41hcfec24a_1002	conda-forge
r-covr	3.5.1	r41h03ef668_0	conda-forge
r-cowplot	1.1.1	r41hc72bb7e_0	conda-forge
r-cpp11	0.4.0	r41hc72bb7e_0	conda-forge
r-crayon	1.4.1	r41hc72bb7e_0	conda-forge
r-credentials	1.3.1	r41hc72bb7e_0	conda-forge
r-crosstalk	1.1.1	r41hc72bb7e_0	conda-forge
r-curl	4.3.2	r41hcfec24a_0	conda-forge
r-desc	1.4.0	r41hc72bb7e_0	conda-forge
r-devtools	2.4.2	r41hc72bb7e_0	conda-forge
r-diffobj	0.3.5	r41hcfec24a_0	conda-forge
r-digest	0.6.28	r41h03ef668_0	conda-forge
r-dplyr	1.0.7	r41h03ef668_0	conda-forge
r-dt	0.19	r41hc72bb7e_0	conda-forge

r-ellipsis	0.3.2	r41hcfec24a_0	conda-forge
r-evaluate	0.14	r41hc72bb7e_2	conda-forge
r-fansi	0.4.2	r41hcfec24a_0	conda-forge
r-farver	2.1.0	r41h03ef668_0	conda-forge
r-fastmap	1.1.0	r41h03ef668_0	conda-forge
r-fontawesome	0.2.2	r41hc72bb7e_0	conda-forge
r-fs	1.5.0	r41h03ef668_0	conda-forge
r-generics	0.1.0	r41hc72bb7e_0	conda-forge
r-gert	1.4.1	r41h29657ab_1	conda-forge
r-ggplot2	3.3.5	r41hc72bb7e_0	conda-forge
r-gh	1.3.0	r41hc72bb7e_0	conda-forge
r-git2r	0.28.0	r41hf628c3e_1	conda-forge
r-gitcreds	0.1.1	r41hc72bb7e_0	conda-forge
r-glue	1.4.2	r41hcfec24a_0	conda-forge
r-gtable	0.3.0	r41hc72bb7e_3	conda-forge
r-highr	0.9	r41hc72bb7e_0	conda-forge
r-htmltools	0.5.2	r41h03ef668_0	conda-forge
r-htmlwidgets	1.5.4	r41hc72bb7e_0	conda-forge
r-httpuv	1.6.3	r41h03ef668_0	conda-forge
r-httr	1.4.2	r41hc72bb7e_0	conda-forge
r-ini	0.3.1	r41hc72bb7e_1003	conda-forge
r-isoband	0.2.5	r41h03ef668_0	conda-forge
r-jquerylib	0.1.4	r41hc72bb7e_0	conda-forge
r-jsonlite	1.7.2	r41hcfec24a_0	conda-forge
r-knitr	1.35	r41hc72bb7e_0	conda-forge
r-labeling	0.4.2	r41hc72bb7e_1	conda-forge

r-later	1.2.0	r41h03ef668_0	conda-forge
r-lattice	0.20_45	r41hcfec24a_0	conda-forge
r-lazyeval	0.2.2	r41hcfec24a_2	conda-forge
r-lifecycle	1.0.1	r41hc72bb7e_0	conda-forge
r-magrittr	2.0.1	r41hcfec24a_1	conda-forge
r-markdown	1.1	r41hcfec24a_1	conda-forge
r-mass	7.3_54	r41hcfec24a_0	conda-forge
r-matrix	1.3_4	r41he454529_0	conda-forge
r-memoise	2.0.0	r41hc72bb7e_0	conda-forge
r-mgcv	1.8_38	r41he454529_0	conda-forge
r-mime	0.12	r41hcfec24a_0	conda-forge
r-munsell	0.5.0	r41hc72bb7e_1004	conda-forge
r-nlme	3.1_153	r41h859d828_0	conda-forge
r-openssl	1.4.5	r41he36bf35_1	conda-forge
r-pillar	1.6.4	r41hc72bb7e_0	conda-forge
r-pkgbuild	1.2.0	r41hc72bb7e_0	conda-forge
r-pkgconfig	2.0.3	r41hc72bb7e_1	conda-forge
r-pkgload	1.2.3	r41h03ef668_0	conda-forge
r-plogr	0.2.0	r41hc72bb7e_1003	conda-forge
r-plyr	1.8.6	r41h03ef668_1	conda-forge
r-praise	1.0.0	r41hc72bb7e_1005	conda-forge
r-prettyunits	1.1.1	r41hc72bb7e_1	conda-forge
r-processx	3.5.2	r41hcfec24a_0	conda-forge
r-promises	1.2.0.1	r41h03ef668_0	conda-forge
r-ps	1.6.0	r41hcfec24a_0	conda-forge
r-purrr	0.3.4	r41hcfec24a_1	conda-forge

r-r6	2.5.1	r41hc72bb7e_0	conda-forge
r-rappdirs	0.3.3	r41hcfec24a_0	conda-forge
r-rcmdcheck	1.4.0	r41h785f33e_0	conda-forge
r-rcolorbrewer	1.1_2	r41h785f33e_1003	conda-forge
r-rcpp	1.0.7	r41h03ef668_0	conda-forge
r-rematch2	2.1.2	r41hc72bb7e_1	conda-forge
r-remotes	2.4.1	r41hc72bb7e_0	conda-forge
r-reshape2	1.4.4	r41h03ef668_1	conda-forge
r-rex	1.2.0	r41hc72bb7e_1	conda-forge
r-rlang	0.4.12	r41hcfec24a_0	conda-forge
r-rmarkdown	2.11	r41hc72bb7e_1	conda-forge
r-roxygen2	7.1.2	r41h03ef668_0	conda-forge
r-rprojroot	2.0.2	r41hc72bb7e_0	conda-forge
r-rstudioapi	0.13	r41hc72bb7e_0	conda-forge
r-rversions	2.1.1	r41hc72bb7e_0	conda-forge
r-sass	0.4.0	r41h03ef668_0	conda-forge
r-scales	1.1.1	r41hc72bb7e_0	conda-forge
r-sessioninfo	1.1.1	r41hc72bb7e_1002	conda-forge
r-shiny	1.7.1	r41h785f33e_0	conda-forge
r-sourcetools	0.1.7	r41h9c3ff4c_1002	conda-forge
r-stringi	1.7.5	r41hcabe038_0	conda-forge
r-stringr	1.4.0	r41hc72bb7e_2	conda-forge
r-sys	3.4	r41hcfec24a_0	conda-forge
r-testthat	3.1.0	r41h03ef668_0	conda-forge
r-tibble	3.1.5	r41hcfec24a_0	conda-forge
r-tidyselect	1.1.1	r41hc72bb7e_0	conda-forge

r-tinytex	0.34	r41hc72bb7e_0	conda-forge
r-usethis	2.1.0	r41hc72bb7e_0	conda-forge
r-utf8	1.2.2	r41hcfec24a_0	conda-forge
r-vctrs	0.3.8	r41hcfec24a_1	conda-forge
r-viridislite	0.4.0	r41hc72bb7e_0	conda-forge
r-waldo	0.3.1	r41hc72bb7e_0	conda-forge
r-whisker	0.4	r41hc72bb7e_1	conda-forge
r-withr	2.4.2	r41hc72bb7e_0	conda-forge
r-xfun	0.27	r41h03ef668_0	conda-forge
r-xml2	1.3.2	r41h03ef668_1	conda-forge
r-xopen	1.0.0	r41hc72bb7e_1003	conda-forge
r-xtable	1.8_4	r41hc72bb7e_3	conda-forge
r-yaml	2.2.1	r41hcfec24a_1	conda-forge
r-zeallot	0.1.0	r41hc72bb7e_1003	conda-forge
r-zip	2.2.0	r41hcfec24a_0	conda-forge
ratelimiter	1.2.0.post0	pypi_0	pypi
readline	8.1.2	h7f8727e_1	
requests	2.27.1	pypi_0	pypi
requests-cache	0.9.3	pypi_0	pypi
sed	4.8	he412f7d_0	conda-forge
setuptools	60.9.3	pypi_0	pypi
six	1.16.0	pypi_0	pypi
sqlite	3.37.2	hc218d9a_0	
sysroot_linux-64	2.12	he073ed8_15	conda-forge
tk	8.6.12	h27826a3_0	conda-forge
tktable	2.10	hb7b940f_3	conda-forge

toml	0.10.2	pypi_0	pypi
tzdata	2021e	he74cb21_0	conda-forge
url-normalize	1.4.3	pypi_0	pypi
urllib3	1.26.8	pypi_0	pypi
wheel	0.37.1	pyhd8ed1ab_0	conda-forge
xorg-kbproto	1.0.7	h14c3975_1002	conda-forge
xorg-libice	1.0.10	h516909a_0	conda-forge
xorg-libsm	1.2.3	hd9c2040_1000	conda-forge
xorg-libx11	1.7.2	h7f98852_0	conda-forge
xorg-libxext	1.3.4	h7f98852_1	conda-forge
xorg-libxrender	0.9.10	h7f98852_1003	conda-forge
xorg-libxt	1.2.1	h7f98852_2	conda-forge
xorg-renderproto	0.11.1	h14c3975_1002	conda-forge
xorg-xextproto	7.3.0	h14c3975_1002	conda-forge
xorg-xproto	7.0.31	h14c3975_1007	conda-forge
xz	5.2.5	h516909a_1	conda-forge
zlib	1.2.11	h36c2ea0_1013	conda-forge
zstd	1.5.2	ha95c52a_0	conda-forge

References

1. Henzinger M, Raghavan P, Rajagopalan S (1999) Computing on data streams. *External Memory Algorithms* (American Mathematical Society), pp 107–118.

2. Zhang J (2010) A Survey on Streaming Algorithms for Massive Graphs. *Managing and Mining Graph Data* (Springer US), pp 393–420.

3. Babcock B, Babu S, Datar M, Motwani R, Widom J (2002) Models and issues in data stream systems. *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems - PODS '02* (ACM Press). doi:10.1145/543613.543615.

4. Muthukrishnan S (2005) Data Streams: Algorithms and Applications. *FNT in Theoretical Computer Science* 1(2):117–236.

5. Muir P, et al. (2016) The real cost of sequencing: scaling computation to keep pace with data generation. *Genome Biology* 17(1):53.

6. Cao MD, et al. (2016) Streaming algorithms for identification of pathogens and antibiotic resistance potential from real-time MinIONTM sequencing. *GigaScience* 5(1):32.

7. Nguyen SH, Duarte TPS, Coin LJM, Cao MD (2017) Real-time demultiplexing

Nanopore barcoded sequencing data with npBarcode. *Bioinformatics* 33(24):3988–3990.

8. Teng H, et al. (2018) Chiron: translating nanopore raw signal directly into nucleotide sequence using deep learning. *GigaScience* 7(5):giy037.

9. Simpson JT, Pop M (2015) The Theory and Practice of Genome Sequence Assembly. *Annual Review of Genomics and Human Genetics* 16(1):153–172.

10. Staden R (1979) A strategy of DNA sequencing employing computer programs. *Nucleic acids research* 6(7):2601–2610.

11. Maier D (1978) The complexity of some problems on subsequences and supersequences. *Journal of the ACM (JACM)* 25(2):322–336.

12. Tarhio J, Ukkonen E (1988) A greedy approximation algorithm for constructing shortest common superstrings. *Theoretical computer science* 57(1):131–145.

13. Huang X, Madan A (1999) CAP3: A DNA sequence assembly program. *Genome research* 9(9):868–877.

14. Myers EW (1995) Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology* 2(2):275–290.

15. Myers EW (2005) The fragment assembly string graph. *Bioinformatics* 21(Suppl 2):ii79–ii85.

16. Gonnella G, Kurtz S (2012) Readjoiner: A fast and memory efficient string graph-based sequence assembler. *BMC Bioinformatics* 13(1):82.

17. Simpson JT, Durbin R (2012) Efficient de novo assembly of large genomes using compressed data structures. *Genome Research* 22(3):549–556.

18. Li H (2012) Exploring single-sample SNP and INDEL calling with whole-genome de novo assembly. *Bioinformatics* 28(14):1838–1844.

19. Medvedev P, Georgiou K, Myers G, Brudno M (2007) Computability of models for sequence assembly. *International Workshop on Algorithms in Bioinformatics* (Springer), pp 289–301.

20. Movahedi NS, Forouzmand E, Chitsaz H (2012) De novo co-assembly of bacterial genomes from multiple single cells. *Bioinformatics and Biomedicine (Bibm), 2012 Ieee International Conference on* (IEEE), pp 1–5.

21. Zerbino DR, Birney E (2008) Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res* 18(5):821–9.

22. Zerbino DR, Birney E (2008) Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res* 18(5):821–829.

23. Simpson JT, et al. (2009) ABySS: A parallel assembler for short read sequence data. *Genome Res* 19(6):1117–1123.

24. Ferragina P, Manzini G Opportunistic data structures with applications. *Proceedings 41st Annual Symposium on Foundations of Computer Science* (IEEE Comput. Soc). doi:10.1109/sfcs.2000.892127.

25. Chikhi R, Limasset A, Medvedev P (2016) Compacting de bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics* 32(12):i201–i208.

26. Chikhi R, Limasset A, Jackman S, Simpson JT, Medvedev P (2015) On the Representation of De Bruijn Graphs. *Journal of Computational Biology* 22(5):336–352.

27. Pell J, et al. (2012) Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proc Natl Acad Sci USA* 109(33):13272–13277.

28. Roach JC (1995) Random subcloning. *Genome Research* 5(5):464–473.

29. Zerbino DR, Birney E (2008) Velvet: Algorithms for de novo short read assembly using de Bruijn graphs. *Genome Research* 18(5):821–829.

30. Pevzner PA, Tang H, Waterman MS (2001) An eulerian path approach to dna fragment assembly. *Proceedings of the National Academy of Sciences* 98(17):9748–9753.

31. Chikhi R, Limasset A, Jackman S, Simpson JT, Medvedev P (2015) On the representation of de bruijn graphs. *Journal of Computational Biology* 22(5):336–352.

32. Minkin I, Pham S, Medvedev P (2016) TwoPaCo: an efficient algorithm to build the compacted de Bruijn graph from many complete genomes. *Bioinformatics*:btw609.

33. Holley G, Melsted P (2020) Bifrost: highly parallel construction and indexing of colored and compacted de Bruijn graphs. *Genome Biol* 21(1):249.

34. Khan J, Patro R (2021) Cuttlefish: fast, parallel and low-memory compaction of de Bruijn graphs from large-scale genome collections. *Bioinformatics* 37(Supplement_1):i177–i186.

35. McGregor A (2014) Graph stream algorithms: A survey. *ACM SIGMOD Record* 43(1):9–20.

36. Feigenbaum J, Kannan S, McGregor A, Suri S, Zhang J (2005) On graph problems in a semi-streaming model. *Theoretical Computer Science* 348(2-3):207–216.

37. Rozov R, Goldshlager G, Halperin E, Shamir R (2017) Faucet: Streaming de novo assembly graph construction. *Bioinformatics* 34(1):147–154.

38. El-Metwally S, Zakaria M, Hamza T (2016) LightAssembler: Fast and memory-

efficient assembly algorithm for high-throughput sequencing reads. *Bioinformatics* 32(21):3215–3223.

39. Lemire D, Kaser O (2010) Recursive n-gram hashing is pairwise independent, at best. *Computer Speech & Language* 24(4):698–710.

40. Crusoe MR, et al. (2015) The khmer software package: Enabling efficient nucleotide sequence analysis. *F1000Research* 4.

41. Pell J, et al. (2012) Scaling metagenome sequence assembly with probabilistic de Bruijn graphs. *Proceedings of the National Academy of Sciences* 109(33):13272–13277.

42. GitHub - greg7mdp/parallel-hashmap: A family of header-only, very fast and memory-friendly hashmap and btree containers. *GitHub*. Available at: https://github.com/greg7mdp/parallel-hashmap [Accessed November 19, 2021].

43. Li K-H (1994) Reservoir-sampling algorithms of time complexity O (n (1 + log(N / n))). *ACM Trans Math Softw* 20(4):481–493.

44. GitHub - sivel/speedtest-cli: Command line interface for testing internet bandwidth using speedtest.net *GitHub*. Available at: https://github.com/sivel/speedtest-cli [Accessed March 3, 2022].

45. GitHub - camillescott/goetia: Streaming de Bruijn and Compact de Bruijn Graph Algorithms *GitHub*. Available at: https://github.com/camillescott/goetia [Accessed October 20, 2021].

46. Grüning B, et al. (2018) Bioconda: sustainable and comprehensive software distribution for the life sciences. *Nature Methods* 15(7):475–476.

47. Lavrijsen WTLP, Dutta A (2016) High-Performance Python-C++ Bindings with PyPy and Cling. *Institute of Electrical and Electronics Engineers (IEEE)*.

doi:10.1109/pyhpc.2016.008.

48. GitHub - camillescott/debruijnal-enhance-o-tron: pytest fixtures for testing de Bruin Graphs *GitHub*. Available at: https://github.com/camillescott/debruijnal-enhance-o-tron [Accessed January 28, 2022].

49. Brown CT, Howe A, Zhang Q, Pyrkosz AB, Brom TH (2012) A reference-free algorithm for computational normalization of shotgun sequencing data. *arXiv preprint arXiv:12034802*. Available at: http://arxiv.org/abs/1203.4802 [Accessed March 27, 2014].

50. Chen L, et al. (2020) Paired rRNA-depleted and polyA-selected RNA sequencing data and supporting multi-omics data from human T cells. *Sci Data* 7(1). doi:10.1038/s41597-020-00719-4.

51. Bell D, Bell AH, Bondaruk J, Hanna EY, Weber RS (2016) In-depth characterization of the salivary adenoid cystic carcinoma transcriptome with emphasis on dominant cell type. *Cancer* 122(10):1513–1522.

52. Bolger AM, Lohse M, Usadel B (2014) Trimmomatic: a flexible trimmer for Illumina sequence data. *Bioinformatics* 30(15):2114–2120.

53. FASTX-Toolkit Available at: http://hannonlab.cshl.edu/fastx_toolkit/ [Accessed February 3, 2022].

54. Zhang Q, Pell J, Canino-Koning R, Howe AC, Brown CT (2014) These Are Not the K-mers You Are Looking For: Efficient Online K-mer Counting Using a Probabilistic Data Structure. *PLoS ONE* 9(7):e101271.

55. Rubinfeld R, Shapira A (2011) Sublinear Time Algorithms. *SIAM J Discrete Math* 25(4):1562–1588.

56. Marçais G, Solomon B, Patro R, Kingsford C (2019) Sketching and Sublinear Data Structures in Genomics. *Annu Rev Biomed Data Sci* 2(1):93–118.

57. Rowe WPM (2019) When the levee breaks: a practical guide to sketching algorithms for processing the flood of genomic data. *Genome Biol* 20(1). doi:10.1186/s13059-019-1809-x.

58. Ondov BD, et al. (2016) Mash: fast genome and metagenome distance estimation using MinHash. *Genome Biol* 17(1). doi:10.1186/s13059-016-0997-x.

59. Orenstein Y, Pellow D, Marçais G, Shamir R, Kingsford C (2017) Designing small universal k-mer hitting sets for improved analysis of high-throughput sequencing. *PLoS Comput Biol* 13(10):e1005777.

60. Flajolet P, Fusy É, Gandouet O, Meunier F (2007) HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. *Discrete Mathematics & Theoretical Computer Science* DMTCS Proceedings vol. AH,...(Proceedings). doi:10.46298/dmtcs.3545.

61. Available at: `https://people.cs.uct.ac.za`.

62. Titus Brown C, Irber L (2016) sourmash: a library for MinHash sketching of DNA. *JOSS* 1(5):27.

63. Irber L, et al. (2022) Lightweight compositional analysis of metagenomes with FracMinHash and minimum metagenome covers. doi:10.1101/2022.01.11.475838.

64. Lance GN, Williams WT (1966) Computer Programs for Hierarchical Polythetic Classification ("Similarity Analyses"). *The Computer Journal* 9(1):60–64.

65. Keeling PJ, et al. (2014) The Marine Microbial Eukaryote Transcriptome Sequencing Project (MMETSP): Illuminating the Functional Diversity of Eukaryotic Life in the

Oceans through Transcriptome Sequencing. *PLoS Biol* 12(6):e1001889.

66. Johnson LK, Alexander H, Brown CT (2019) Re-assembly, quality evaluation, and annotation of 678 microbial eukaryotic reference transcriptomes. *GigaScience* 8(4). doi:10.1093/gigascience/giy158.

67. Ren J, et al. (2015) Genome-wide analysis of the ATP-binding cassette (ABC) transporter gene family in sea lamprey and Japanese lamprey. *BMC Genomics* 16(1). doi:10.1186/s12864-015-1677-z.

68. Garber M, Grabherr MG, Guttman M, Trapnell C (2011) Computational methods for transcriptome annotation and quantification using RNA-seq. *Nat Methods* 8(6):469–477.

69. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990) Basic local alignment search tool. *Journal of Molecular Biology* 215(3):403–410.

70. doit - Automation tool Available at: https://pydoit.org/ [Accessed December 2, 2021].

71. Mölder F, et al. (2021) Sustainable data analysis with Snakemake. *F1000Res* 10:33.

72. XDG Base Directory Specification Available at: https://specifications.freedesktop.org/basedir-spec/basedir-spec-latest.html [Accessed January 6, 2022].

73. GitHub - camillescott/ope: Run tools on FASTA files with gnu parallel, and then parse the results. *GitHub*. Available at: https://github.com/camillescott/ope [Accessed January 6, 2022].

74. Reback J, et al. (2020) *pandas-dev/pandas: Pandas 1.0.3* (Zenodo) doi:10.5281/zenodo.3715232.

75. McKinney W (2010) Data Structures for Statistical Computing in Python (Austin, Texas), pp 56–61.

76. dammit! Available at: http://dib-lab.github.io/dammit/ [Accessed January 6, 2022].

77. Tange O (2011) GNU Parallel - The Command-Line Power Tool.*;login: The USENIX Magazine* 36(1):42–47.

78. Cluster Execution — Snakemake 7.0.1 documentation Available at: https://snakemake.readthedocs.io/en/stable/executing/cluster.html [Accessed March 1, 2022].

79. Finn RD, et al. (2016) The Pfam protein families database: Towards a more sustainable future. *Nucleic Acids Research* 44(D1):D279–D285.

80. Griffiths-Jones S, Bateman A, Marshall M, Khanna A, Eddy SR (2003) Rfam: An RNA family database. *Nucleic Acids Research* 31(1):439–441.

81. Kriventseva EV, et al. (2015) OrthoDB v8: Update of the hierarchical catalog of orthologs and the underlying free software. *Nucleic Acids Research* 43(Database issue):D250–256.

82. Simao FA, Waterhouse RM, Ioannidis P, Kriventseva EV, Zdobnov EM (2015) BUSCO: Assessing genome assembly and annotation completeness with single-copy orthologs. *Bioinformatics*. doi:10.1093/bioinformatics/btv351.

83. Suzek BE, Huang H, McGarvey P, Mazumder R, Wu CH (2007) UniRef: Comprehensive and non-redundant UniProt reference clusters. *Bioinformatics* 23(10):1282–1288.

84. RefSeq non-redundant proteins Available at: https://www.ncbi.nlm.nih.gov/refseq/about/nonredundantproteins/ [Accessed March 1, 2022].

85. Haas BJ, et al. (2013) De novo transcript sequence reconstruction from RNA-Seq: Reference generation and analysis with Trinity. *Nature protocols* 8(8). doi:10.1038/nprot.2013.084.

86. Durbin R, Eddy S, Krogh A, Mitchison G (1998) *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids* (Cambridge University Press) Available at: http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0521629713 [Accessed May 11, 2016].

87. Nawrocki EP, Eddy SR (2013) Infernal 1.1: 100-fold faster RNA homology searches. *Bioinformatics* 29(22):2933–2935.

88. Sheetlin SL, Park Y, Frith MC, Spouge JL (2014) Frameshift alignment: Statistics and post-genomic applications. *Bioinformatics* 30(24):3575–3582.

89. Kiełbasa SM, Wan R, Sato K, Horton P, Frith MC (2011) Adaptive seeds tame genomic sequence comparison. *Genome Research* 21(3):487–493.

90. Aubry S, Kelly S, Kümpers BMC, Smith-Unna RD, Hibberd JM (2014) Deep Evolutionary Comparison of Gene Expression Identifies Parallel Recruitment of Trans -Factors in Two Independent Origins of C 4 Photosynthesis. *PLOS Genet* 10(6):e1004365.

91. Scott C (2017) shmlast: An improved implementation of Conditional Reciprocal Best Hits with LAST and Python. *JOSS* 2(9):142.

92. Robinson JT, et al. (2011) Integrative Genomics Viewer. *Nature biotechnology* 29(1):24–26.

93. Eilbeck K, et al. (2005) The Sequence Ontology: A tool for the unification of genome annotations. *Genome Biology* 6:R44.

94. Sukumaran J, Holder MT (2010) DendroPy: A python library for phylogenetic computing. *Bioinformatics* 26(12):1569–1571.

95. Huerta-Cepas J, Serra F, Bork P (2016) ETE 3: Reconstruction, analysis, and visualization of phylogenomic data. *Molecular biology and evolution* 33(6):1635–1638.

96. Safatli A, Blouin C (2015) Pylogeny: An open-source python framework for phylogenetic tree reconstruction and search space heuristics. *PeerJ Computer Science* 1:e9.

97. Talevich E, Invergo BM, Cock PJ, Chapman BA (2012) Bio.Phylo: A unified toolkit for processing, analyzing and visualizing phylogenetic trees in biopython. *BMC bioinformatics* 13(1):209.

98. Biocore (2018) Scikit-bio. *GitHub repository*.

99. Price MN, Dehal PS, Arkin AP (2009) FastTree: Computing large minimum evolution trees with profiles instead of a distance matrix. *Molecular biology and evolution* 26(7):1641–1650.

100. Price MN, Dehal PS, Arkin AP (2010) FastTree 2–approximately maximum-likelihood trees for large alignments. *PLOS ONE* 5(3):e9490.

101. GitHub - ryanpeek/aggiedown: An R Markdown template using the bookdown package for preparing a PhD thesis at the University of California Davis *GitHub*. Available at: https://github.com/ryanpeek/aggiedown [Accessed March 16, 2022].

102. Himmelstein DS, et al. (2019) Open collaborative writing with Manubot. *PLoS Comput Biol* 15(6):e1007128.

103. Bloom BH (1970) Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13(7):422–426.

104. Crusoe MR, et al. (2015) The khmer software package: enabling efficient nucleotide sequence analysis. *F1000Res* 4:900.

105. GitHub - greg7mdp/sparsepp: A fast, memory efficient hash map for C++ *GitHub*. Available at: https://github.com/greg7mdp/sparsepp [Accessed November 19, 2021].

106. GitHub - sparsehash/sparsehash: C++ associative containers *GitHub*. Available at: https://github.com/sparsehash/sparsehash [Accessed November 19, 2021].

107. Pandey P, Bender MA, Johnson R, Patro R (2017) A General-Purpose Counting Filter. *Association for Computing Machinery (ACM)*. doi:10.1145/3035918.3035963.