Identifying Taxonomic Units in Metagenomic DNA Streams on Mobile Devices

Vicky Zheng, Ahmet Erdem Sariyuce, and Jaroslaw Zola

Abstract—With the emergence of portable DNA sequencers, such as Oxford Nanopore Technology MinION, metagenomic DNA sequencing can be performed in real-time and directly in the field. However, because metagenomic DNA analysis tasks, e.g., classification, taxonomic units assignment, etc., are compute and memory intensive, and the available methods are designed for batch processing, the current metagenomic tools are not well suited for mobile devices. In this work, we propose a new memory-efficient approach to identify Operational Taxonomic Units (OTUs) in metagenomic DNA streams on mobile devices. Our method is based on finding connected components in overlap graphs constructed over a real-time stream of long DNA reads as produced by the MinION platform. We propose an efficient algorithm to maintain connected components when an overlap graph is streamed and show how redundant information can be removed from the stream by transitive closures. We also propose how our algorithms can be integrated into a larger DNA analysis pipeline tailored for mobile computing. Through experiments on simulated and real-world metagenomic data, executed on the actual mobile device, we demonstrate that our resulting solution is able to recover OTUs with high precision. Our experiments also demonstrate the compounding benefits of introducing feedback loops in the DNA analysis pipeline.

Index Terms—Connected components, metagenomics, nanopore sequencing, streaming algorithms

1 Introduction

Recently introduced nanopore-based DNA sequencers, specifically Oxford Nanopore Technology (ONT) MinION [4], are revolutionizing how DNA-based studies are performed. Their key advantages are a small form factor and low energy consumption that make them fully portable and allow for easy deployment in the field, outside of a typical laboratory [22], [33]. Moreover, these devices can sequence DNA molecules directly (i.e., without extra steps like DNA amplification) and can stream the resulting DNA reads in near real-time. This makes them extremely attractive for metagenomic studies that involve processing DNA recovered directly from environmental samples. In recent years, MinION sequencers have been increasingly used for *in situ* studies, including, for example, tracking of COVID19, Ebola and Zika outbreaks [1], [12], [31], deployments in the Arctic and Antarctic [11], and even on the International Space Station [9] (we invite the reader to [18] for a broader discussion on mobile DNA sequencing).

One of the most common tasks in metagenomics is identification of Operational Taxonomic Units (OTUs) represented by clusters of highly similar DNA reads. OTUs often serve as a proxy representing microbial composition of the sequenced sample in cases where reads classification (e.g., by searching a DNA database of known organisms) is difficult or impossible. However, in the current mobile DNA sequencing workflows, identification of OTUs, along with any other DNA analytics, remains challenging. In Fig. 1, we outline the usual mobile DNA sequencing workflow using MinION. The device streams, in real-time, electric signals characterizing detected DNA fragments. These signals are basecalled to yield DNA reads, which are next processed using full-fledged bioinformatics tools. In a mobile setup, the sequencer is typically coupled with a portable host device with limited compute power, memory, and energy supply (e.g., tablet or a dedicated system-on-a-chip like MinIT [2]). Since the basecalling process is already compute and memory intensive, the bioinformatics analysis step has to be either offloaded to a cloud service (which is not always possible or desired) or postponed until sufficient compute resources become available. In both cases, the resulting delay between DNA read acquisition and the analytics is highly undesirable from the end-user's perspective, as it decreases the overall responsiveness.

In this work, we focus on the problem of identifying OTUs in mobile DNA read streams generated by MinION portable sequencers. Our goal is to provide a memory and compute efficient solution that could be deployed as a co-processing routine in portable DNA sequencing workflows operating on light-weight computational devices. Our approach is based on finding connected components in the similarity (or overlap) graphs constructed and streamed directly over the DNA read streams. Connected components have been demonstrated before as a robust representation of OTUs [13], [15], [29]. They are an attractive abstraction, as they are the starting point to multiple other tasks, including DNA assembly [30], reference-free taxonomic classification (or clustering) [17], [36], or species abundance estimation [10]. Intuitively, connected components build from the observation that metagenomic samples contain many organisms whose DNA reads

• *The authors are with the Department of Computer Science, University at Buffalo, Buffalo, NY 14260-1660 USA. E-mail: {vickyzhe, erdem, jzola}@buffalo.edu.*

Manuscript received 26 Feb. 2021; revised 14 Jan. 2022; accepted 22 Apr. 2022. Date of publication 5 May 2022; date of current version 3 Apr. 2023.
This work was supported by National Science Foundation under the under Grant CNS-1910193.
(Corresponding author: Vicky Zheng.)
Digital Object Identifier no. 10.1109/TCBB.2022.3172661

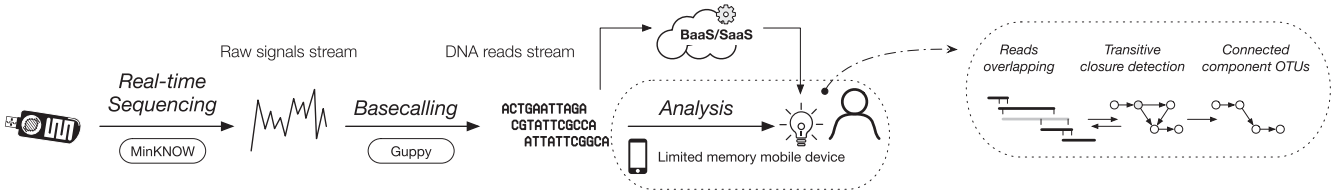

Fig. 1. Schematic representation of mobile DNA sequencing pipeline with MinION. Sequenced reads can be analyzed locally, or can be offloaded to a Back-end as a Service (or Software as a Service) for processing in a cloud. In this work, we propose an OTU identification method suitable to run directly on mobile devices.

should not assemble together in a similarity graph [29]. In this work, we make the following specific contributions:

- We propose efficient algorithms to identify transitive closures and maintain connected components in streamed DNA overlap graphs. These algorithms are able to identify transitive closures in expected constant time and allow us to maintain a minimal memory footprint while collecting connected component statistics to identify OTUs.
- We show how our algorithms can form a feedback loop with a DNA overlaps detection routine to further reduce memory footprint of the end-to-end processing workflow.
- Through experimental results on the actual mobile device, we demonstrate that the proposed methods are both memory and computationally efficient, and can identify OTUs in ONT MinION sequencing data.

2 Problem Formulation

We consider a mobile DNA sequencing pipeline as presented in Fig. 1. A portable DNA sequencer (specifically ONT MinION) is attached and controlled by a battery powered mobile host (e.g., laptop or system-on-a-chip) that is also responsible for DNA data processing. The sequencer delivers, in real-time, raw signals representing detected DNA molecules. Raw signals are comprised of the measurement of the ionic displacement of DNA molecules as they pass through a nanopore. These raw signals are immediately basecalled on the host machine yielding the actual DNA reads. To illustrate the rate at which the process happens: in our experiments, we usually observe that the sequencer delivers around 130 raw signals per second. The basecalling rate varies depending on the host machine capabilities. For example, using NVIDIA Nano Supercomputer-on-a-Chip with 128 GPGPU cores and 4 GB of main memory, the basecalling can be sustained at the rate of approximately 31 reads per second. Reads generated from nanopore-based sequencers are commonly tens of thousands of bases in length with the longest reported read being 2.3Mbp [21]. The resulting DNA reads stream is passed for the downstream analysis.

In this work, we are specifically interested in performing OTU identification directly on the host that receives streamed DNA reads. To this end, we use connected components as proxies for OTUs. As mentioned earlier, connected component have been demonstrated as a robust OTU representation in the past [15], [29]. To formalize the resulting problem, let $R = [r_1, \ldots, r_n]$ represent an input stream of DNA reads generated in real-time by a DNA sequencer. We have that for each $i < j$ read r_i precedes read r_j in the stream, which we will denote by $r_i \prec r_j$. The size of the stream, n, is not known *a priori*. For example, a user may decide to terminate a sequencing experiment at any point of time (e.g., after *sufficient* data has been collected), or may run an experiment for a specified time interval (e.g., two hours, which could be a small metagenomic experiment).

Given a set of DNA reads, we can construct an overlap graph $G = (V, E)$ in which vertices V represent the reads, and two vertices, u and v, are connected by the directed edge, $u \rightarrow v$, denoted by $e = (u, v)$, if there is a *significant* overlap between a suffix of the read represented by u and a prefix of the read represented by v. Here significant overlap means that the length of the suffix-prefix match is beyond some predefined threshold and indicates that the two reads corresponding to u and v have been derived from neighboring portions of the unknown underlying genome. For now, we assume that an overlap detection tool (ODT) is available and capable of constructing an overlap graph over the stream R (see below).

Let R^i be the set of the first i reads from the stream R, and let $G^i = (V^i, E^i)$ be the overlap graph constructed over R^i. Moreover, let $\overline{G^i}$ denote an undirected graph constructed by treating all edges in E^i as undirected. Our goal is to dynamically identify and maintain, in a computationally and memory efficient way, a set $C^i = \{c^i_1, c^i_2, \ldots, c^i_{|C^i|}\}$ of all connected components found in graph $\overline{G^i}$, where c^i_j is the set of vertices in component j of graph $\overline{G^i}$. The final set of connected components, C^n, will represent the Operational Taxonomic Units over the set of DNA reads in R. In other words, we will expect that all reads within a given connected component will be coming from the same taxonomic unit (e.g., an organism).

We note that, when processing stream R, we are concerned only with the graph G^i and the set C^i (since these structures carry our information of interest), and do not have to explicitly store or maintain graph $\overline{G^i}$. Moreover, currently we are not concerned with the details of how the overlap graph is computed (e.g., what is the similarity threshold for suffix-prefix comparison, or how DNA reverse-complements are handled; we discuss this issue in Section 3.4). In other words, we are assuming that in our mobile DNA sequencing pipeline, there is an ODT that operates under the hardware constraints mentioned earlier. The ODT handles an incoming stream of reads generated by a sequencer and creates, with some precision and sensitivity, a new stream of edges induced by the incoming read. Specifically, given incoming read r_i, let v_i be its corresponding vertex in graph G^i. The ODT provides sets $N^+(v_i) \subseteq V^{i-1}$ and $N^-(v_i) \subseteq V^{i-1}$, such that for each $u \in N^+(v_i)$ there is an edge $(v_i, u) \in E^i$ and for each $u \in N^-(v_i)$ there exists

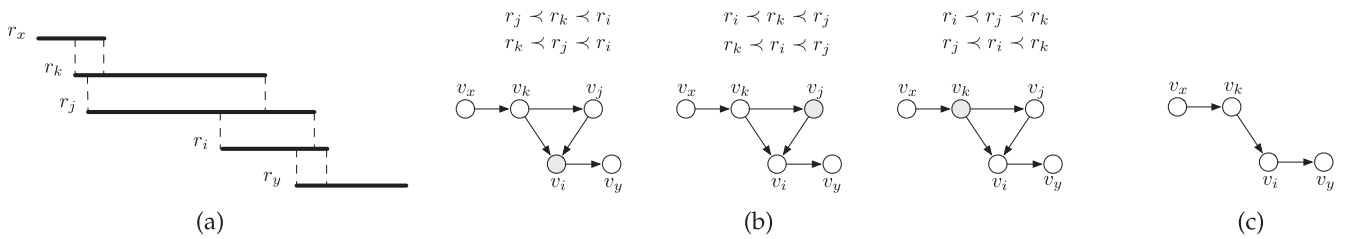

Fig. 2. (a) Example DNA reads with their overlaps marked. (b) Three different cases that may occur in the overlap graph constructed over reads in (a) if $r_x \prec r_y \prec r_i, r_j, r_k$, and ODT does not report overlap between $r_x \rightarrow r_j$ and $r_j \rightarrow r_y$. (c) Irreducible graph created by removing r_j as a transitive read. Connected components in the resulting graph do not change.

an edge $(u, v_i) \in E^i$ (note that in both cases u corresponds to a read that precedes r_i in R, i.e., $u \prec v_i$).

3 Proposed Approach

Given the above problem formulation, to create and maintain our desired set C^i, we could leverage one of the several existing algorithms for finding connected components in graph streams (we review them in Section 5). However, this approach has drawbacks because the current algorithms are tailored for general graph streams and are oblivious to the domain specific properties of the data. Therefore we take a slightly different route and exploit the (well known) fact that, in a typical DNA sequencing experiment, many DNA reads are sharing an overlap, and thus provide redundant information.

To better illustrate this property, consider a set of reads in Fig. 2a that come from the same region of a genome. Because read r_j overlaps with both r_i and r_k, and at the same time r_i and r_k are overlapping as well, we can eliminate read r_j without disconnecting the underlying overlap graph and hence without losing any critical information. This simplifies the graph on which we have to perform connected components identification, and reduces the number of reads (and hence graph nodes) we have to maintain in the memory. The redundant reads may be marked as such, and can be offloaded to a persistent storage and processed later, e.g., when more computational resources are available.

The second observation is that, in our problem, we never remove arbitrary nodes from the overlap graph. This simplifies data management as we can adopt tested data structures such as disjoint-set to identify connected components [19].

In our approach, outlined in the right-most panel of Fig. 1, we exploit both properties at the same time: we first provide an efficient strategy to identify redundant reads in a stream and then use a variant of union-find to track connected components. In the process, we provide feedback information to the ODT to further improve end-to-end performance on the entire workflow. Here we note that the feedback step is entirely optional, and it does not affect the performance of our method.

3.1 Finding Transitive Nodes

The first step in our approach is to decide whether an incoming read is redundant and hence can be removed from further processing. Let us consider again example reads in Fig. 2a and, for the purpose of presentation, let us assume that $r_x \prec r_y \prec r_i, r_j, r_k$. Depending on which of the reads r_i, r_j, r_k arrives last, we have one of three possible overlap graphs as shown in Fig. 2b. Here, read r_i is represented by vertex v_i, and shaded nodes correspond to the last arriving read. Any redundant read, in our case read r_j, shares suffix-prefix overlap with at least two other reads, in our case r_i and r_k. Now let us consider an induced subgraph over the redundant vertex (i.e., vertex corresponding to the redundant read, in our case v_j) and any two of its adjacent vertices that are also adjacent to each other. In this subgraph, one of the nodes must have two outgoing edges, one must have two incoming edges, and one must have one incoming and one outgoing edge (this node must correspond to a redundant read). Back to Fig. 2b, we can see that node v_j is redundant irrespective of the order in which reads are processed due to the transitive edge (v_k, v_i). We will call nodes that introduce transitive edges between two other nodes *transitive*, and since they correspond to redundant reads, we will eliminate them from processing.

For each incoming read, we now want to identify if it introduces any transitive nodes. This can be done by considering all possible triples it forms with its adjacent nodes. For instance, to find the transitive nodes introduced by the arriving node v_i (i.e., read r_i) in the first example in Fig. 2b, we need to check if it has a pair of incoming neighbors $u, w \in N^-(v_i)$ that share an edge (u, w) or (w, u). In the second example, to find the transitive nodes introduced by the arriving node, we check if it has an incoming neighbor $u \in N^-(v_j)$ and outgoing neighbor $w \in N^+(v_j)$ that share an edge (u, w). Finally, to find the transitive nodes introduced by the arriving node v_k, we need to check if it has a pair of outgoing neighbors $u, w \in N^-(v_k)$ that share an edge (u, w) or (w, u).

In general, we notice that there are three ways an incoming read can introduce transitive nodes. However, because we do not know which case we are dealing with, we need to consider all three. This intuition is captured in Algorithm 1.

For each incoming read v, the input to the algorithm are all overlaps with the previously processed nodes (represented by sets $N^+(v)$ and $N^-(v)$), and the current irreducible overlap graph G (we explain irreducibility below). In lines 3-9, we check if v has a pair of outgoing neighbors $u, w \in N^+(v)$ that share an edge $(u, w) \in E$ or $(w, u) \in E$. This scenario corresponds to the last case in Fig. 2b. In lines 10-16, we check if v has a pair of incoming neighbors $u, w \in N^-(v)$ that share an edge $(u, w) \in E$ or $(w, u) \in E$. This scenario corresponds to the first case in the Fig. 2b. Finally, in lines 17-21, we check if v has a pair of neighbors u, w where $u \in N^+(v)$ and $w \in N^-(v)$ share an edge $(w, u) \in E$. This scenario corresponds to the middle case in the Fig. 2b.

Algorithm 1. FINDTRANSITIVE($G, v, N^+(v), N^-(v)$)

```
1: G = (V, E)   G is irreducible
2: L ← ∅   set of transitive nodes
3: for u ∈ N+(v) do
4:   for w ∈ N+(v), u ≠ w do
5:     if u ∉ L ∧ w ∉ L then
6:       if (u, w) ∈ E then
7:         L ← L ∪ {u}
8:       else if (w, u) ∈ E then
9:         L ← L ∪ {w}
10: for u ∈ N−(v) do
11:   for w ∈ N−(v), u ≠ w do
12:     if u ∉ L ∧ w ∉ L then
13:       if (u, w) ∈ E then
14:         L ← L ∪ {w}
15:       else if (w, u) ∈ E then
16:         L ← L ∪ {u}
17: for u ∈ N+(v) do
18:   for w ∈ N−(v) do
19:     if u ∉ L ∧ w ∉ L then
20:       if (w, u) ∈ E then
21:         L ← L ∪ {v}
22: return L
```

Although the processing steps in Algorithm 1 are simple, the entire procedure can be computationally expensive for a mobile system and streaming regime (keeping in mind that it is executed for each incoming read). Because we are searching for edges between incoming neighbors, edges between outgoing neighbors, and edges between incoming and outgoing neighbors, the process requires $\Theta((|N^-(v_i)| + |N^+(v_i)|)^2)$ edge queries. This is problematic, as v_i may have a high degree, and each edge query can be expensive depending on how G is stored in memory. However, we can make certain guarantees about our incoming node degree as long as G^{i-1} is *irreducible*. Here, we define an irreducible graph to be an overlap graph that does not contain transitive nodes.

Cost of Handling Transitive Nodes

In a streaming regime, we can maintain graph irreducibility by eliminating transitive nodes the moment they are introduced by an incoming node. We will call removing transitive nodes *transitive closures*. Maintaining irreducibility through transitive closures will not require any additional computational effort. However, it will necessitate a dedicated approach to maintain a coherent list of connected components.

Recall that by definition, an ODT detects an edge (u, v) if there is a significant overlap between the reads corresponding to u and v. Here, a significant overlap indicates the reads corresponding to u and v have been derived from neighboring portions of the underlying genome. If we have an irreducible overlap graph, we are guaranteed that for an incoming read v_i, $|N^-(v_i)| + |N^+(v_i)| \leq 4$ with $|N^-(v_i)| \leq 2$ and $|N^+(v_i)| \leq 2$. This can be shown through contradiction: Suppose we have an incoming read r_i and an irreducible graph G^{i-1}. Now, suppose $|N^+(v_i)| = 3$ (or $|N^-(v_i)| = 3$, both work). By definition, the three reads in $N^+(v_i)$ that overlap with r_i belong to the same portion of the genome as r_i. This is because their prefixes overlap with r_i. Since all three of the reads belong to the same portion of the genome, then they must also overlap with one another. Since they all overlap with one another, at least one of the reads is either contained within another read or contained within the overlap of the other two reads. This means that at least one of the reads is transitive which contradicts that G^{i-1} is irreducible. This contradiction shows that both $|N^+(v_i)|$ and $|N^-(v_i)| \leq 2$ and therefore $|N^-(v_i)| + |N^+(v_i)| \leq 4$ so long as G^{i-1} is irreducible. Notice that this is demonstrated in Fig. 2b.

We recognize that the above reasoning holds for the case where an overlap (a suffix-prefix match above some predefined threshold) between reads indicates that they are derived from the neighboring portion of a genome. This assumption is not always true in the real world (e.g., due to repeats in a genome and errors in sequencing). However, as we show later in Section 3.4, our method still performs acceptably well even when the assumption of graph irreducibility is violated.

3.2 Maintaining Graph and Components

Given an efficient routine to identify transitive nodes in the stream, we need a way to maintain the both graph structure and the corresponding connected components. Our solution must also be able to handle deletions of transitive nodes.

To maintain graph G^i, we use a simple adjacency list built on top of a hash table. For each vertex we maintain a list of its incoming and outgoing neighbors, and the neighbor lists are kept in a hash table with key derived from vertex identifier. In this way, we compensate for the fact that the size of the input stream is not known in advance. This solution is practical, efficient and takes into account small expected size of the neighbor lists.

To store and track connected components, we use a variant of the union-find data structure (UF) [19]. Specifically, we represent UF as a set of key-value pairs $\langle v_r, v_c \rangle$, where v_r is some node mapped to a component represented by v_c (i.e., v_c is the root of the component). Here, a set is again implemented over a hash table, to account for the fact that the size of UF will be changing dynamically.

Algorithm 2. STREAMCC(G^{i-1}, UF, $v_i, N^+(v_i), N^-(v_i)$)

```
1: G^{i-1} = (V^{i-1}, E^{i-1})
2: L ← FINDTRANSITIVE(G^{i-1}, v_i, N+(v_i), N−(v_i))
3: E− ← ∅
4: for u ∈ L do
5:   UF[u] ← nil
6:   E− ← E− ∪ {e | e = (u, w) ∧ e ∈ E^{i-1}}
7:   E− ← E− ∪ {e | e = (w, u) ∧ e ∈ E^{i-1}}
8: if v_i ∉ L then
9:   UF[v_i] ← v_i
10:  for u ∈ (N+(v_i) ∪ N−(v_i)) do
11:    if UF[u] < UF[v_i] then
12:      UF[v_i] ← UF[u]
13:  for u ∈ (N+(v_i) ∪ N−(v_i)) do
14:    if UF[u] ≠ UF[v_i] then
15:      P ← {r | UF[r] = u}
16:      for w ∈ P do
17:        UF[w] ← UF[v_i]
18: C^i ← UF   connected components represented viaUF
19: V^i ← (V^{i-1} ∪ {v_i}) \ L
20: E^i ← E^{i-1} \ E−
21: ODT(drop L)   optional feedback to ODT
```

3.3 Maintaining Components Over a Stream

Having all ingredients in place, we summarize our method for maintaining connected components in Algorithm 2. In the first step, we identify all transitive nodes that are safe to remove using our FindTransitive routine (line 2). For each transitive node, we first remove its corresponding entry from the UF structure (line 5). We then identify all associated edges to remove from G (lines 6 and 7). Because we are using hash tables for both UF and E, both of these operations can be done in amortized $O(|V|)$. Since G is reduced by transitive closures, we expect $|V|$ to be a small fraction of all processed nodes (which we confirmed via experimental results). If the incoming node v_i is not in the list of nodes to remove, we proceed with insertion in lines 8-17 (explained below). Finally, in lines 18-21, we remove all edges associated with transitive nodes along with the transitive nodes and reads themselves.

One critical advantage of our approach is that it can provide feedback to the ODT (line 21) instructing it which reads are potentially redundant. Since transitive nodes correspond to redundant reads, they may be safely offloaded (or dropped) to a persistent storage, and processed later as needed, thus saving memory. Moreover, by maintaining only non-redundant reads, an ODT may improve its performance as well. These properties turn out to be particularly crucial when executing on low-memory devices, since offloading reads from the main memory allows us to save gigabytes and hence operate on much larger data set then what would be normally possible (as we demonstrate in the experimental results).

Inserting Nodes

To insert a node v_i (lines 8-17), we process how v_i affects UF (lines 9-17). Node v_i can form its own component (if it has no neighbors), join a component (if its neighbors all belong to the same component), or merge components (if its neighbors belong to different components). First, we assume that v_i is forming its own component in line 11. Then, we handle the situation where it joins a component or merges components. If v_i has neighbors, then we identify the root of the component to which v_i belongs to (lines 10-12). We note that because we are incrementally maintaining UF with each incoming read, the resulting structure always has a depth of one (all nodes are connected to the root). Therefore, the cost of this step is $O(|N^-(v_i) \cup N^+(v_i)|)$, which is constant when G is irreducible.

In lines 13-17, we handle the situation where components are merged. We do this by checking if any of v_i's neighboring nodes have a different root (line 14). When this is the case, we reassign all of the nodes $w \in c^{i-1}_{\mathrm{UF}[w]}$, to $\mathrm{UF}[w] \leftarrow \mathrm{UF}[v_i]$. As a result, neighbors of v_i end up having the same component mapping as v_i. Note that gathering component members in line 15 requires searching through UF, and hence takes $O(|V|)$ time. However, this cost gets amortized over the course of execution as shown in experimental results.

3.4 Tuning to Real World Data

So far we have been working under the assumption that an ODT correctly identifies suffix-prefix overlaps beyond some predefined threshold between reads coming from the neighboring parts of a genome. Although this assumption is helpful in our understanding of transitive nodes, it is not entirely realistic. This is because DNA reads, especially in MinION sequencing, are error-prone and typical genomes are repetitive. Consequently, an ODT may either miss overlap between reads (e.g., due to sequencing error) or may detect spurious overlaps (e.g., due to repetition where similar reads come from different regions of a genome).

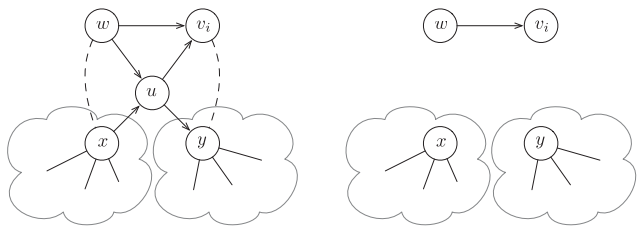

Fig. 3. Impact of false positive and false negative edges on connected components discovery when using transitive closure. Left: vertex v_i is added to the graph with single component consisting of vertices $\{w, u, x, y\}$. We consider two cases where edges (w, x) and (v_i, y) are incorrectly missing, or edges (x, u) and (u, y) are incorrectly introduced. Right: After performing transitive closure on u, in both cases the graph will consist of three connected components instead of one component.

If an ODT detects an overlap between reads that do not belong to neighboring portions of the genome, then this is a false positive. The connected components of the overlap graph that contains many false positives may not be a robust representations of OTUs; this is especially apparent in the presence of high sequencing error. However, they may still be informative, for example, OTUs may be describing higher taxonomic units (say genus or family) instead of representing individual species. In [29], Pell *et al*. found that if the false positive rate is above a so called percolation threshold (e.g., 18% in one of their tests on de Bruijns graphs), erroneous connections between the "true" components emerge and the resulting components do not represent species any more.

If an ODT misses an overlap between two reads that belong to neighboring portions of a genome, then this is a false negative. This can cause two components that are supposed to be connected to be disjoint. Disconnected components may also misrepresent the true OTUs, leading to incorrect assessment of the number of OTUs. False negatives are extremely common and can be dealt with by increasing the depth of coverage to get a single, complete component for each OTU. Increasing the depth of coverage means that more reads are sequenced from the sample. The intuition behind this is that the more reads that are sequenced, the more opportunities we give to the ODT to identify an overlap in that region and ideally form a single component. The increased depth of coverage causes overlap graphs to grow large as it introduces more nodes. However, this is exactly the problem that our transitive closures approach addresses.

In Fig. 3, we show a hypothetical scenario that demonstrates the impact of false positive/negative edges. In this scenario, node u is transitive, and therefore it will be removed by transitive closure. However, in the presence of false positive/negative edges, removing a transitive node may disconnect our graph. To see why, observe that nodes w and v_i are connected to u, and x and y are connected to u, and yet these two pairs are not connected to each other.

This could have happened because ODT missed edges (u, x) and (v_i, y) or the ODT incorrectly identified edges (x, u) and (u, y). In either case, removing u will disconnect the graph. We use the term *articulation points* to identify nodes that disconnect components upon their removal [14].

Protecting Articulation Points

When transitive nodes are articulation points, we know this is due to an ODT error. However, we cannot conclude whether this is due to false positive or false negative edges. Due to this uncertainty, we should not remove articulation because we cannot assume that articulation points correctly provide redundant information. Moreover, disconnecting components would require recomputing all components from scratch, thus degrading computational performance. Furthermore, to even determine whether a component has become disconnected is a challenging problem on its own [23]. Hence, in our approach we decided to protect articulation points from being removed.

Let nodes corresponding to the reads that contain the redundant read be *anchor nodes*. For example, in Fig. 3, u is transitive while v_i and w are anchor nodes. Instead of checking whether a node is an articulation point, we will impose a more strict, and inexpensive to compute, criteria: A transitive node must have *anchored* neighbors for it to be removed. A node is anchored if it shares an edge with an anchor node. If all of a transitive node's neighbors are anchored, then we can safely remove the transitive node. This is because a transitive node cannot be an articulation point if its neighbors are anchored. Anchor nodes must belong to the same component as the transitive node. If all of the transitive node's neighbors are anchored, then they all have at least two points of connection to the rest of the component: one point is with the transitive node and another point is with at least one of the anchored nodes. This allows us to safely remove the transitive node because the neighbors still maintain at least one point of connection to the rest of the component.

We can modify Algorithm 2 to enforce that we do not remove transitive nodes that may be articulation points. To do this, it is sufficient that we remove from set L any node whose neighbors are not all anchored. For each node $u \in L$, this necessary condition can be checked in $O(|N^-(u)| + |N^+(u)|)$ time because each neighbor of u requires a constant number of edge queries to see if it is anchored. Hence, this modification imposes only slight overhead compared to the original algorithm.

Effects of Preserving Transitive Nodes

We will refer to an overlap graph that has transitive nodes that are not articulation points removed as a *reduced* graph. In a reduced graph, we are guaranteed to not disconnect an existing component. However, in the following example, we illustrate how removing transitive nodes may still result in a fragmented component in the future. Suppose that nodes u, v_i and w precede nodes x and y in Fig. 3 ($u, v_i, w \prec x, y$). Since x and y have not been added to the graph yet, node u is not an articulation point and hence is removed by transitive closure. When x and y are added to the graph, they must end up in different components from w and v_i since u is missing. Although this may affect the component count, in the experimental results, we show that connected components are still robust representations of OTUs.

Another observation is that because we cannot remove all transitive nodes, we will not have the same degree guarantees as in an irreducible overlap graph. For example, in Fig. 3, we cannot remove u due to it being an articulation point. Later in the process, we may receive a read r_j, that has specifically suffix-prefix overlaps with the reads corresponding to w and x but also u. In such case, the out degree of v_j will be three, which violates our previously established degree constraints in irreducible graphs.

Fortunately, in practice, we observe that incoming nodes in the stream follow an exponential degree distribution, which is a side effect of transitive closures (see experimental results). If we assume an exponential distribution, then we can still guarantee that the expected degree of a node will be constant. Suppose an incoming node v_i has degree of k with probability p_k, where $p_k = (1 - e^{-1/\kappa})e^{-k/\kappa}$ and κ is some constant [27]. Then the expected average degree of an infinite stream is: $E[|N^+(v_i)|+|N^-(v_i)|] = \sum_{k=1}^{\infty} k \cdot p_k = \sum_{k=1}^{\infty} k(1 - e^{-1/\kappa})e^{-k/\kappa} = \frac{e^{-1/\kappa}}{1-e^{-1/\kappa}} = O(\kappa)$.

Since the average degree is bounded by κ, as long as κ is a small constant, maintaining transitive closures on a reduced graph will still perform similarly to an irreducible overlap graph. In our experimental results, we found that κ never exceeds three.

4 Experimental Results

To validate our proposed approach, we implemented Algorithms 1 and 2, including the extensions from Section 3.4, in a standalone C++ application (the code is open source and available from [35]). We performed a set of experiments using synthetic as well as actual, publicly available, GridION data [3]. In our experiments, we focused on performance (e.g., run time and memory use) as well as correctness characteristics (e.g., connected components and OTUs recovery and convergence) taking into account properties of the ODT. We also note that all experiments that involved random sampling or shuffling of the data were repeated multiple times, and we did not observe significant difference from the results reported below.

4.1 Test Data

To prepare benchmark data, we started from the ERR2906227 data set publicly available from the European Nucleotide Archive [3]. This data set has been generated using the ONT GridION sequencer, which is a scaled-up version of the portable MinION platform. We selected this particular data set since it represents metagenomic sequencing of a mock community with known microbial composition. Specifically, the data set is based on the Zymo Community Standards that comprises five Gram-positive bacteria, three Gram-negative bacteria, all eight organisms in the same abundance, and two types of yeast that make 4% of the community (more details regarding this particular data set and how sequencing had been performed can be found in [28]). We chose to use data sets with organisms in the equal abundance to better assess the effects of transitive closures on OTUs identification. In

Supplement 1, we provide additional experimental results for the case where abundance of species changes logarithmically.

To annotate the reads, i.e., assign them to one of the component organisms in the mock community, we performed read mapping using the `blast-2.10.1+` tool [7] and the reference genomes provided by Zymo [37] (maker of the mock community). For each mapped read, we selected its target OTU based on the best mapping score (i.e., to which reference genome it mapped the best). We disregarded reads with low mapping scores and reads that did not map uniquely. Furthermore, we eliminated very few reads mapping to the yeast genomes since they made less than 4% of all reads. The resulting data set consists of 2,969,089 classified (i.e., assigned to an OTU) reads. We will refer to the resulting data set as `ERR2906227`.

We used the annotated `ERR2906227` data set to simulate artificial but realistic reads using the `NanoSim` tool [34]. Starting from the actual GridION reads and the corresponding reference genomes, `NanoSim` first builds a statistical model of the GridION sequencer and then uses this model to derive new reads from the reference genomes. Since `NanoSim` reports from which position in the genome each simulated read has been derived, we can use this information to create a perfect ODT, that is, a tool in which no false positive or false negative edges are created. This enables us also to control for ODT performance (e.g., precision and sensitivity) when assessing performance of our algorithms. We refer to the resulting data set as `Sim`. The data set consists of 200,000 reads, where each of the eight reference genomes is represented by 25,000 reads.

For reproducibility purposes, we provide all details regarding data preparation in the accompanying web page [35].

4.2 Overlaps Detection

Detection of prefix-suffix overlaps in DNA reads is extensively studied topic, especially in the context of long reads such as those produced by MinION or PacBio platforms. However, although there are several ODTs available (e.g., `daligner` [26], `minimap2` [20], `MHAP` [8], `ELaSTIC` [36]), these tools do not provide any formal guarantees with respect to the quality and correctness of the discovered overlaps. Moreover, they currently are not meant to work in a streaming regime and are not designed to incorporate potential feedback provided by our algorithms (e.g., line 21 in Algorithm 1). Specifically, they do not expose any interface that would enable us to inform the ODT which reads are no longer required to perform OTU identification (such reads could be removed by the ODT from the main memory and from computations). Taking all of that into account, in order to test our solution while controlling for ODT quality and performance, we decided to simulate overlap graph streaming based on overlaps detected via batch processing.

To simulate the overlap process for the `ERR2906227` data set, we directly leveraged information provided by the `blast` tool when performing reads assignment to OTUs (as explained in the previous section). Specifically, we assumed that two reads overlap if they are mapped by `blast` to the same portions of the underlying reference genome such that they have a suffix-prefix overlap of at least size 1,000 nucleotides/characters (based on [25] we consider such overlap length significant). To simulate an ODT for the `Sim` data set, we used a similar approach. However, instead of using `blast`, we directly used the exact mapping information provided by `NanoSim` (as explained earlier). In both cases, we obtained a complete overlap graph *a priori*, including information about direction of the overlaps.

Since the overlap graphs are constructed by directly leveraging information provided by `blast` and `NanoSim`, the overlap graphs do not contain any false positive edges. This means that each component consists entirely of reads from a single input species and therefore are homogeneous. If we used an existing ODT, the constructed overlap graph may have false positive edges because it does not know the underlying true graph structure. Although we are not able to control for quality of correctness or give feedback to a real ODT, we provide additional results using `minimap2` in Supplement 1).

Given an overlap graph, we were able to directly emulate the corresponding stream of reads R and their neighborhoods N^+ and N^-. To achieve realistic behavior from the memory use perspective, our simulated ODT performed as follows. All reads from the stream R are stored in a FASTA file in the order in which they appeared in the stream (for each data set we generated several streams by randomly permuting the order of the reads and we observed no significant difference in performance of our algorithms for different streams). In the FASTA file, the name of each read encoded information about which other reads in the stream that read is overlapping with. When executing, our ODT iterated over the FASTA file and mimicked overlap detection, which involved parsing the read name to check which of the overlapping reads were already seen in the stream and are in the main memory. From that information, the ODT would generate sets N^+ and N^- and stream them to our OTU identification algorithm (using Linux/Unix pipes). Finally, the ODT was capable of receiving feedback from the OTU identification algorithm and used the feedback to reduce the number of reads maintained in the memory. Specifically, the ODT was maintaining reads as standard strings in a fast hash table, and would immediately erase any entries marked as redundant (recall line 21 in Algorithm 2).

4.3 Effectiveness of Transitive Closures

In the first experiment, we tested how maintaining transitive closures, hence eliminating redundant reads, affects our ability to recover connected components and their corresponding OTUs. To do this, we constructed a series of overlap graphs over the `Sim` data set, in each case randomly removing a fraction of edges to introduce false negatives. For each such created overlap graph, we simulated a streaming process and used our software to identify connected components from the stream. To obtain the actual true number of connected components in a given overlap graph, we used standard union-find algorithm on the graph without streaming. We note that in the ideal scenario, both methods should be identifying eight connected components, each corresponding to one OTU in the data set. The results of this experiment are summarized in Fig. 4. Here we use Actual to denote the number of connected components that are identifiable in the graph, and Transitive to denote the number of connected components recovered when the graph is processed using our method.

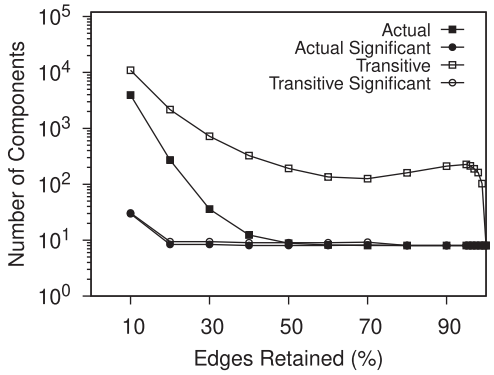

Fig. 4. Number of identified connected components depending on the percentage of edges retained in the Sim overlap graph that has eight components (y-axis is in log scale). Actual means components identified using union-find on the entire graph, Transitive means components identified using our method. Significant means components with more than three nodes.

From the figure we can make several observations. First, to correctly recover all eight OTUs, the overlap graph must contain at least 50% of edges. When edge retention is below 50%, the missing edges cause the graph to become highly disconnected and hence components no longer uniquely identify OTUs. Second, irrespective of the edge retention, transitive closures introduce exponentially more connected components than there are in the unprocessed graph. This is not entirely surprising considering our previous discussion on how missing edges can lead to disconnected nodes with transitive closures. As edge retention increases, so does expected node degree of the nodes in the underlying graph. This in turn introduces more redundancies therefore increasing the number of transitive closures performed. In our tests, the impact of transitive closures become less pronounced only after exceeding 95% edge retention resulting in sharp dip visible in Fig. 4.

While these results may suggest poor performance of our method, the picture drastically changes if we consider only connected components with more than one node. In Fig. 5, we inspect the component size distribution of components recovered in the streamed and actual graph. As edge retention increases, we observe that there are eight distinct components along with primarily singleton components. This clear gap in component sizes allows us to easily distinguish significant components that correspond to the target OTUs. If we assume that significant components are components of at least size three (denoted by Significant in Fig. 4), then from Fig. 4, we can see that we quickly converge to the desired number of components representing OTUs in both our streamed and actual graph. We note that we observe the same behavior in real world data set ERR2906227 as well as in data set with logarithmic abundance distribution (see Supplement 1). We also note that since the components of the underlying overlap graphs are homogeneous, and our algorithm does not introduce any false positive edges, all recovered components are also homogeneous. We provide more detailed numbers on effectiveness of transitive closures in Supplement 1.

4.4 Performance Characteristics

In the second set of experiments, we assessed performance characteristics of our algorithm in terms of runtime and memory use. As discussed earlier, the cost of performing transitive closures and incrementally maintaining connected components depends on the average degree of each incoming node, irrespective of the performance of the ODT. In Fig. 6, we plot the incoming node degree distribution against the fitted exponential distribution when streaming the Sim data set. From the figure, we can see that the exponential distribution indeed closely fits our data for various levels of edge retention. Moreover, κ, parameter of the underlying exponential distribution, consistently remains a small constant (we recorded $1.08 \leq \kappa \leq 2.4$). This confirms that transitive closures can be performed in amortized constant time. We note that we observe the same behavior in real world data set ERR2906227.

To assess the cost of incrementally maintaining connected components, as discussed in Section 3.3, we show the effect of incoming reads on component formation in Fig. 7. Initially, when nodes enter the graph, they form singletons. Then, the components grow larger and begin to merge, thus reducing the number of components. Finally, the number levels off and remains constant. This indicates that as components get larger, merging becomes less frequent, supporting our claim that the costly task of merging components gets amortized over the course of execution.

Memory Characteristics

In Fig. 8, we show how eliminating redundant reads through transitive closures affects storage rates throughout the streaming process. Here, the storage rate refers to the fraction of reads, as well as their corresponding graph nodes, that the ODT should maintain in the main memory (such that its functioning does not affect OTU identification). From this figure, we can see that as edge retention improves (i.e., we

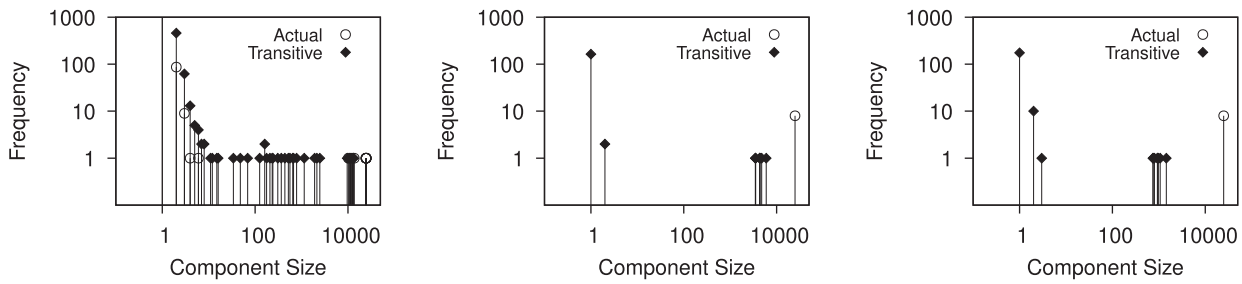

Fig. 5. Frequency of components of given size when 10% (left plot), 50% (middle plot) and 90% (right plot) of the edges in the Sim overlap graph are retained (both axes are in log scale). As edge retention increases, there are eight large components along with primarily singleton components.

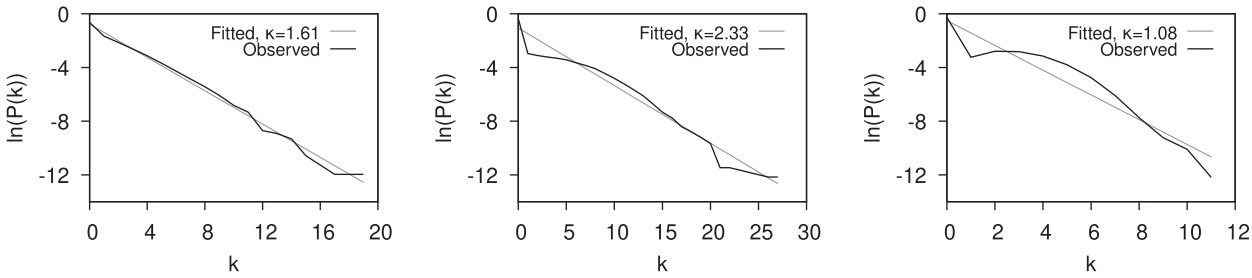

Fig. 6. Node degree distribution observed when streaming `Sim` data set and corresponding fitted exponential distribution, when 10% (left plot, $\kappa = 1.61$, $R^2 = 0.99$), 50% (middle plot, $\kappa = 2.33$, $R^2 = 0.96$) and 90% (right plot, $\kappa = 1.08$, $R^2 = 0.92$) of the edges are retained.

have fewer false negatives), we store a smaller percentage of nodes. This makes sense intuitively because increased edge retention introduces more redundancies, which lead to more transitive closures. The same reasoning applies to why storage use improves over time. Specifically, as more nodes are introduced, more overlaps are being detected and hence more redundancies are discovered and eliminated via transitive closures.

To further demonstrate the practical implications of our feedback mechanism, we combined the transitive closure algorithms with the simulated ODT and deployed them on an actual mobile device to demonstrate the potential to use our tool in *in situ* experiments. For that purpose, we used the NVIDIA Jetson Nano SCoC. The device comes with a quad-core ARM Cortex-A57 processor, NVIDIA Maxwell with 128 GPGPU cores, and 4 GB 64-bit LPDDR4 1600 MHz of main memory. Our software setup was based on `Linux Ubuntu 18.04.3` configured with 8 GB of swap memory (more details of our hardware/software configuration are available from the SMARTEn project [5]). It is worth noting that a very similar hardware configuration is offered by Oxford Nanopore in their portable MinIT platform [2]. Nevertheless, the tool is hardware agnostic and can be deployed on any device ranging from data center through desktops/laptops to mobile devices.

To measure the memory usage of the ODT combined with our algorithms, we executed the experiments (in isolation and without any additional non-OS processes running) and monitored the main memory and swap memory usage using the Linux `sysstat/sar` tool. Here, we considered both scenarios in which the ODT either acted on or ignored the feedback from the transitive closure algorithm. Fig. 9 shows the results for the `Sim` data set. Irrespective of the number of edges retained, incorporating the feedback has clear advantage in both reducing the memory footprint and execution time. This is satisfyingly expected and aligns with the numbers we reported for reads retention in Fig. 8. Considering that the average MinION read length in our experiments was around 4 Kbp (with many reads exceeding 41 Kbp), discarding even relatively small number of reads, as in the case of `Sim` data set, allows us to reduce memory use by around 1 GB. This constitutes 25% of the total main memory available in our NVIDIA Nano SCoC. That kind of savings become critical in the case of much larger real world data set, as we show in the following section.

4.5 Real World Data

In the final set of experiments, we assessed how effective our proposed solution is in recovering OTUs in real world data. We classified all reads in the `ERR2906227` data set as discussed earlier, and used the resulting classification to assign them to OTUs. To obtain the actual number of connected components in the `ERR2906227` overlap graph, we again used the standard union-find algorithm. The algorithm returned ten connected components and nine of them were significant instead of the expected eight. This discrepancy between the number of connected components and the number of OTUs is explained by the complexity of one of the reference genomes. Specifically, the genome of *Salmonella* is highly repetitive, which causes repetitive reads to cluster in one region of the genome thus not providing sufficient information to connect other regions. We note that this fragmentation is not a side effect of transitive closures. In fact, component fragmentation due to a repetitive genome is a common and unavoidable problem due to the inability of ODTs to distinguish between very long repetitive portions of the genome.

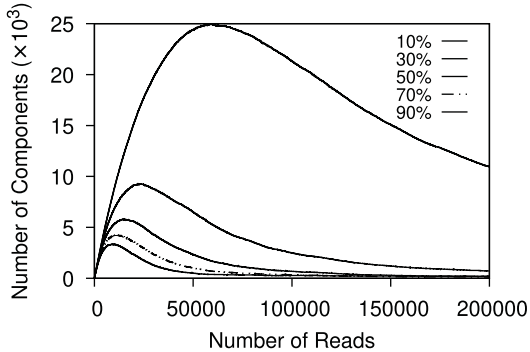

Fig. 7. Number of connected components identified as a function of the number of reads processed when different percentage of edges are retained in the `Sim` data set.

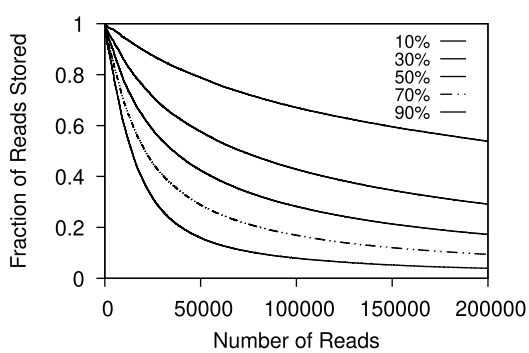

Fig. 8. Fraction of nodes maintained in the memory as a function of the number of reads processed when different percentage of edges are retained in the `Sim` data set.

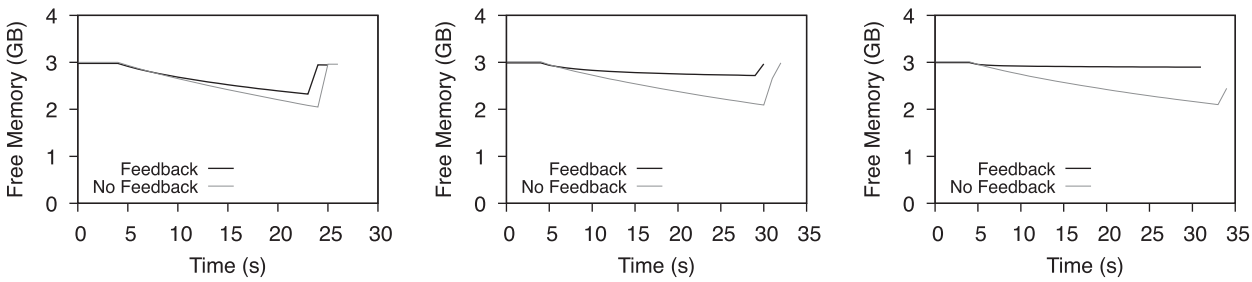

Fig. 9. Main memory available when executing the streaming algorithm together with the simulated ODT for the Sim data set, when 10% (left plot), 50% (middle plot) and 90% (right plot) of the edges are retained. Feedback refers to the case when ODT uses information from the streaming algorithm to drop unneeded reads from memory. No Feedback refers to the case when ODT does not leverage information provided by the streaming algorithm. We can see that No Feedback consistently uses more main memory than Feedback.

We simulated the streaming process of the ERR2906227 overlap graph, and recorded the same statistics as for the simulated data. Our method was able to recover all nine significant components (and thus their corresponding OTUs) after processing 2,004,350 reads (about two thirds of all reads). In our experiments, we found that the exponential degree distribution was a good fit with $R^2 = 0.76$ and $\kappa = 2.4$. This confirms that the assumption of amortized constant time for performing transitive closures holds for the real world data set as well.

Throughout processing, the maximum number of nodes stored was 30,799 (less than 2% of the total reads processed) clearly demonstrating effectiveness of maintaining transitive closures. This result is of particular importance from the mobile processing perspective. Fig. 10 again shows the actual memory and swap usage when executing on the NVIDIA Nano SCoC. When ODT utilizes the feedback, the memory usage remains relatively low (slightly over 1 GB), and no swap memory is required. All this despite the fact that all 2,969,089 reads in the ERR2906227 stream are taking over 12 GB. When the ODT was executed without the feedback mechanism, the main memory was saturated after the first 8 minutes of processing, and the ODT started using swap memory. However, after another 60 minutest of execution the swap also was saturated and the ODT process was terminated by the operating system (after processing slightly over 80% of the stream). In summary, the feedback mechanism allowed us to process relatively large real world data that would otherwise be impossible using our modest mobile resources.

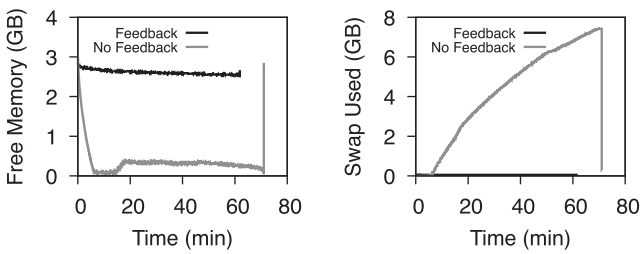

Fig. 10. Main memory available (left) and swap memory used (right) when executing the streaming algorithm together with the simulated ODT for the ERR2906227 data set. Feedback refers to the case when ODT uses information from the streaming algorithm to drop unneeded reads from memory. No Feedback refers to the case when ODT does not leverage information provided by the streaming algorithm. We can once again see that Feedback uses less main memory than No Feedback. We also see that No Feedback eventually exhausts main memory and swap and crashes.

5 Related Work

The idea of using transitive closure to reduce computational complexity has been used previously in the context of DNA assembly, where DNA reads are pieced together to reconstruct longer DNA strings [30]. For example, in [24], Myers discusses DNA string graphs in which vertices represent prefixes and suffixes of reads, and edges represent non-matching substrings between two overlapping reads. He then shows that transitive edges can be removed without affecting the ability to reconstruct the assembly. While in our approach we essentially exploit the same principle, i.e., redundancy of overlapping reads, our focus is on streaming and not memory prohibitive batch processing.

Disjoint-set forests, also known as union-find, is the most efficient data structure to find connected components in a graph. Because of its popularity and versatility, there are several adaptations of union-find for various computational setups. For example, Isenburg and Shewcuk [16] adapted the union-find algorithm for a streaming 3D grid network to use in image processing, Agarwal et al. considered I/O efficient solutions for terrain analysis [6], and Simsiri et al. studied work-efficient parallel adaptations of union-find for incremental graph connectivity [32]. Laura and Santaroni introduced the first semi-streaming algorithm that makes a few passes to find strongly connected components in a directed graph [19]. These methods, however, are primarily focused on general streams where graph nodes and edges can be inserted or removed at any point of time. Moreover, they assume that the executing environment has significant main memory available. In our case, the problem has a slightly different flavor. On the one hand, the stream is easier to handle because we consider only node insertions and specific node removals. Due to the nature of metagenomic reads, we can also expect bounded number of edges introduced with every node insertion. On the other hand, we have very limited access to the main memory and computational power (typically, available memory is around 4-8 GB) in any mobile setup. Consequently, our primary focus is on maintaining minimal memory footprint, while delivering the desired statistics.

6 Conclusion

The growing popularity and rapid adoption of portable DNA sequencing platforms necessitates the development of new computational strategies to enable *in situ* DNA analytics. In

this work, we introduce OTUs identification method based on connected components abstraction and operating on DNA streams. The method can be used to accelerate mobile execution of multiple types of DNA analysis, including metagenomic DNA assembly and classification.

The key element of our solution is memory efficient handling of connected components emerging in streams of DNA reads and their overlap graphs. Through formal and experimental analysis, we show that if the degree distribution of nodes in the streamed overlap graph follows an exponential distribution (which is the case in real-world data) our method has minimal computational cost.

The OTUs identification method we describe in this work builds directly from the earlier observations that connected components may approximate OTUs. While this approach is very computationally appealing, it is not bulletproof. As demonstrated by our experimental results, and observed by others [29], if the number of false positive edges in the overlap graph exceeds certain thresholds, the resulting connected components no longer represent OTUs at the species level. This limitation should be taken into account when selecting the ODT to include in the processing pipeline, and when deciding on the downstream processing tasks.

Our method is introduced in conjunction with the idea of DNA processing pipeline that incorporates feedback loops between different stages in the pipeline. Our experiments demonstrate the compounding benefits of having feedback loops. In our proposed pipeline, performing transitive closures reduces the workload of our ODT by reducing the number of reads the ODT has to manage; the ODT is then able to reduce the workload of components identification by reducing overall overlap graph size. The feedback loops make our algorithms suitable for mobile computing devices.

While our proposed solution addresses the question of how to identify connected components in a stream, it is based on the assumption that the processing pipeline includes an ODT that is able to work efficiently over the streamed DNA reads and incorporate potential feedback. While such ODTs are not yet readily available, we hope that our results will convince other researchers to pursue this mechanism. Currently, we are investigating an adaptive ODT operating directly on the raw signals produced by MinION sequencer (i.e., bypassing basecalling stage). A new real-time DNA processing pipeline and raw signal ODT are both part of the SMARTEn [5] project, our broader effort in mobile DNA processing.

REFERENCES

[1] Nanopore sequencing of the SARS-CoV-2 virus. [Online]. Available: https://tinyurl.com/yduddxk9

[2] MinIT, 2018. [Online]. Available: https://nanoporetech.com/products/minit

[3] European nucleotide archive, 2019. [Online]. Available: https://www.ebi.ac.UK/ena/browser/view/ERR2906227

[4] Oxford Nanopore, 2020. [Online]. Available: https://nanoporetech.com

[5] SMARTEn, 2020. [Online]. Available: https://cse.buffalo.edu/jzola/smarten/

[6] P. Agarwal, L. Arge, and K. Yi, "I/O-efficient batched union-find and its applications to terrain analysis," *ACM Trans. Algorithms*, vol. 7, no. 1, pp. 1–21, 2010.

[7] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *J. Mol. Biol.*, vol. 215, no. 3, pp. 403–410, 1990.

[8] K. Berlin, S. Koren, C. Chin, J. Drake, J. Landolin, and A. Phillippy, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nat. Biotechnol.*, vol. 33, no. 6, pp. 623–630, 2015.

[9] S. Castro-Wallace *et al.*, "Nanopore DNA sequencing and genome assembly on the international space station," *Sci. Rep.*, vol. 7, no. 1, pp. 1–12, 2017.

[10] A. Dilthey, C. Jain, S. Koren, and A. Phillippy, "Strain-level metagenomic assignment and compositional estimation for long reads with MetaMaps," *Nat. Commun.*, vol. 10, no. 1, pp. 1–12, 2019.

[11] A. Edwards *et al.*, "In-field metagenome and 16S rRNA gene amplicon nanopore sequencing robustly characterize glacier microbiota," *bioRxiv*, 2019.

[12] N. Faria *et al.*, "Mobile real-time surveillance of Zika virus in Brazil," *Genome Med.*, vol. 8, no. 1, pp. 1–4, 2016.

[13] P. Flick, C. Jain, T. Pan, and S. Aluru, "A parallel connectivity algorithm for de Bruijn graphs in metagenomic applications," in *Proc. IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–11.

[14] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient algorithms for graph manipulation," *Commun. ACM*, vol. 16, no. 6, pp. 372–378, 1973.

[15] A. Howe, J. Jansson, S. Malfatti, S. Tringe, J. Tiedje, and C. Brown, "Tackling soil diversity with the assembly of large, complex metagenomes," *Proc. Nat. Acad. Sci.*, vol. 111, no. 13, pp. 4904–4909, 2014.

[16] M. Isenburg and J. Shewchuk, "Streaming connected component computation for trillion voxel images," in *MASSIVE*, 2009.

[17] S. Juul *et al.*, "What's in my pot? real-time species identification on the MinION," *bioRxiv*, 2015.

[18] S. Ko, L. Sassoubre, and J. Zola, "Applications and challenges of real-time mobile dna analysis," in *Proc. Int. Workshop Mobile Comput. Syst. Appl.*, 2018, pp. 1–6.

[19] L. Laura and F. Santaroni, "Computing strongly connected components in the streaming model," in *Proc. Int. Conf. Theory Pract. Algorithms Comput. Syst.*, 2011, pp. 193–205.

[20] H. Li, "Minimap2: Pairwise alignment for nucleotide sequences," *Bioinformatics*, vol. 1, 2018, Art. no. 7.

[21] M. Loose, V. Rakyan, N. Holmes, and A. Payne, "Whale watching with BulkVis: A. graphical viewer for Oxford nanopore bulk FAST5 files," *Bioinformatics*, vol. 35, no. 13, 2019.

[22] H. Lu, F. Giordano, and Z. Ning, "Oxford Nanopore MinION sequencing and genome assembly," *Genomic. Proteomic. Bioinf.*, vol. 14, no. 5, pp. 265–279, 2016.

[23] R. McColl, O. Green, and D. Bader, "A new parallel algorithm for connected components in dynamic graphs," in *Proc. Int. Conf. High Perform. Comput.*, 2013, pp. 246–255.

[24] E. Myers, "The fragment assembly string graph," *Bioinformatics*, vol. 21, no. Suppl 2, pp. ii79–ii85, 2005.

[25] E. Myers, "A history of DNA sequence assembly," *Inf. Technol.*, vol. 58, no. 3, pp. 126–132, 2016.

[26] E. Myers, "The dresden azzembler for long reads project (dazzler blog)," 2020. [Online]. Available: https://dazzlerblog.wordpress.com/

[27] M. Newman, S. Strogatz, and D. Watts, "Random graphs with arbitrary degree distributions and their applications," *Phys. Rev. E*, vol. 64, no. 2, 2001, Art. no. 026118.

[28] S. Nicholls, J. Quick, S. Tang, and N. Loman, "Ultra-deep, long-read nanopore sequencing of mock microbial community standards," *Gigascience*, vol. 8, no. 5, 2019, Art. no giz043.

[29] J. Pell, A. Hintze, R. Canino-Koning, A. Howe, J. Tiedje, and C. Brown, "Scaling metagenome sequence assembly with probabilistic de Bruijn graphs," *Proc. Nat. Acad. Sci.*, vol. 109, no. 33, pp. 13 272–13 277, 2012.

[30] M. Pop, "Genome assembly reborn: Recent computational challenges," *Brief. Bioinf.*, vol. 10, no. 4, pp. 354–366, 2009.

[31] J. Quick *et al.*, "Real-time, portable genome sequencing for Ebola surveillance," *Nature*, vol. 530, no. 7589, pp. 228–232, 2016.

[32] N. Simsiri, K. Tangwongsan, S. Tirthapura, and K. Wu, "Work-efficient parallel union-find," *Concurrency Comput. Pract. Experience*, vol. 30, no. 4, 2018, Art. no. e4333.

[33] M. Walter *et al.*, "MinION as part of a biomedical rapidly deployable laboratory," *J. Biotechnol.*, vol. 250, pp. 16–22, 2017.

[34] C. Yang, J. Chu, R. Warren, and I. Birol, "NanoSim: Nanopore sequence read simulator based on statistical characterization," *GigaScience*, vol. 6, no. 4, 2017, Art. no. gix010.

[35] V. Zheng, "Identifying taxonomic units in metagenomic DNA streams - source code," 2020. [Online]. Available: https://github.com/vickymzheng/transclosures
[36] J. Zola, "Constructing similarity graphs from large-scale biological sequence collections," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2014, pp. 500–507.
[37] ZymoBIOMICS, "ZymoBIOMICS mock community reference genome," 2020. [Online]. Available: https://s3.amazonaws.com/zymo-files/BioPool/ZymoBIOMICS.STD.refseq.v2.zip.

Vicky Zheng received the BS and PhD degrees in computer science from the University at Buffalo, in 2016 and 2021, respectively. She worked with the Scalable Computing Research Group where her research focused on real time analysis of DNA streams. She is now working as a research scientist in the industry.

Ahmet Erdem Sariyuce received the BS degree in computer engineering from Middle East Technical University, and the PhD degree in computer science and engineering from Ohio State University. He is currently an assistant professor with the Department of Computer Science and Engineering, University at Buffalo. His research interests include graph mining and management.

Jaroslaw Zola (Senior Member, IEEE) received the MSc degree in computer science from the Technical University of Czestochowa, Poland, in 2001, and the PhD degree in computer science from the Grenoble Institute of Technology, France, in 2005. He is currently an associate professor with the Department of Computer Science and Engineering, University at Buffalo. His research interests include applications of high performance and mobile computing techniques in biomedical applications. He serves as an associate editor of the *the IEEE Transactions on Parallel and Distributed Systems*.