

Real-time resolution of short-read assembly graph using ONT long reads

Son Hoang Nguyen^{1*}, Minh Duc Cao¹, Lachlan Coin^{1,2,3,4*}

1 Institute for Molecular Bioscience, the University of Queensland, St Lucia, Brisbane, QLD 4072 Australia

2 Department of Microbiology and Immunology, The University of Melbourne, Parkville, 3010, Australia

3 Department of Clinical Pathology, The University of Melbourne, Parkville, 3010, Australia

4 Department of Infectious Disease, Imperial College London, London, W2 1NY, UK

*s.hoangnguyen@imb.uq.edu.au, l.coin@imb.uq.edu.au

Abstract

A streaming assembly pipeline utilising real-time Oxford Nanopore Technology (ONT) sequencing data is important for saving sequencing resources and reducing time-to-result. A previous approach implemented in **npScarf** provided an efficient streaming algorithm for hybrid assembly but was relatively prone to mis-assemblies compared to other graph-based methods. Here we present **npGraph**, a streaming hybrid assembly tool using the assembly graph instead of the separated pre-assembly contigs. It is able to produce more complete genome assembly by resolving the path finding problem on the assembly graph using long reads as the traversing guide. Application to synthetic and real data from bacterial isolate genomes show improved accuracy while still maintaining a low computational cost. **npGraph** also provides a graphical user interface (GUI) which provides a real-time visualisation of the progress of assembly. The tool and source code is available at <https://github.com/hsnguyen/assembly>.

Author summary

Introduction

Sequencing technology has reached a level of maturity which allows the decoding of virtually any piece of genetic material which can be obtained. However, the time from sample to result remains a barrier to adoption of sequencing technology into time critical applications such as infectious disease diagnostics or clinical decision making. While there exists real-time sequencing technology such as Oxford Nanopore Technologies (ONT), algorithms for streaming analyses of such real-time data are still in their infancy. Effective streaming methodology will help bridge the gap between potential and practical use. In addition, the release of Read Until API enables direct control to the actual molecular reading process which has paved the way for customized targeted sequencing on ONT platforms. To function properly, such methods must implement a real-time ultrafast operation on the streaming data, *e.g.* in ReadFish [1]. Up until now, there is a shortage of similar tools that can utilize this distinctive feature.

One particular strength of ONT technology is the production of ultra long reads. This is complementary to the dominant short read sequencing technology Illumina which is cheaper and has higher per base read quality but is unable to resolve the complex regions of the genome due to its read length limitation. A natural combination of the two technologies is in hybrid assembly of genomes. Previously, we developed **npScarf** [2] an algorithm to scaffold a draft assembly from Illumina sequencing simultaneously with ONT sequencing. However, **npScarf** ignores the rich connectivity information in the short read assembly graph, and as a result is relatively prone to mis-assembly compared to alternative methods.

Here, we present **npGraph**, a novel algorithm to resolve the assembly graph in real-time using long read sequencing. **npGraph** uses the stream of long reads to untangle knots in the assembly graph, which is maintained in memory. Because of this, **npGraph** has better estimation of multiplicity of repeat contigs, resulting in fewer misassemblies. In addition, we develop a visualisation tool for practitioners to monitor the progress of the assembly process.

Materials and methods

Application overview

npGraph makes use of an assembly graph generated from assembling short reads using a de Bruijn graph method such as **SPAdes** [3], **Velvet** [4] and **AbySS** [5]. The assembly graph consists of a list of contigs, and possible connections among these contigs. In building the assembly graph, the de Bruijn graph assembler attempts to extend each contig as far as possible, until there is more than one possible way of extending due to the repetitive sequences beyond the information contained in short reads. Hence each contig has multiple possible connections with others, creating knots in the assembly graph. **npGraph** uses the connectivity information from long reads to untangle these knots in real-time. With sufficient data, when all the knots are removed, the assembly graph is simplified to a path which represents the complete assembly.

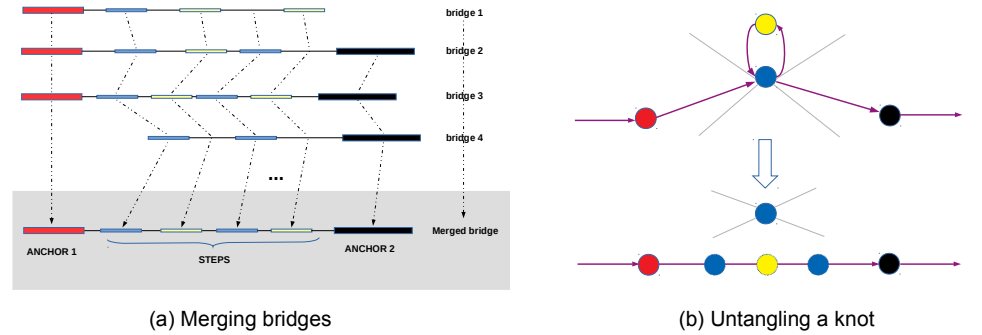


Fig 1. Graph resolving algorithm. (a) the bridges suggested by long reads are merged progressively with dynamic programming to find the best path connecting 2 anchors. (b) A knot (repetitive contig) is unwound following the best path (highlighted in purple) leading to the graph simplification.

npGraph aligns long reads to the contigs in the assembly graph. When a long read is aligned to multiple contigs, **npGraph** constructs candidate paths that are supported by the read. This strategy allows **npGraph** to progressively update the likelihood of the paths going through a knot. When sufficient data are obtained, the best path is confidently identified and hence the knot is untangled. In general, the bridging

algorithm used to determine the best path is a combination of progressive merging, accumulated scoring and decision making modules. It operates on each pair of unique contigs, or *anchors*, by using a *bridge* data structure maintaining 2 anchors and a list of *steps* in-between as shown in Figure 1(a). A set of candidate paths are listed and the best one can be selected amongst them given enough evidence. The de Bruijn graph is subsequently simplified when the bridge is replaced by a single edge representing the best path (1(b)).

We also provide a Graphical User Interface (GUI) for **npGraph**. The GUI includes the dashboard for controlling the settings of the program and a window for visualization of the assembly graph in real-time (Figure 2). In this interface, the assembly graph loading stage is separated from the actual assembly process so that users can check for the graph quality first before carry out any further tasks. A proper combination of command line and GUI can provide an useful streaming pipeline that copes well with MinION output data. The practice is to support the real-time monitoring of the results from real-time sequencing [2, 6, 7] that allow the analysis to take place abreast to a nanopore sequencing run.

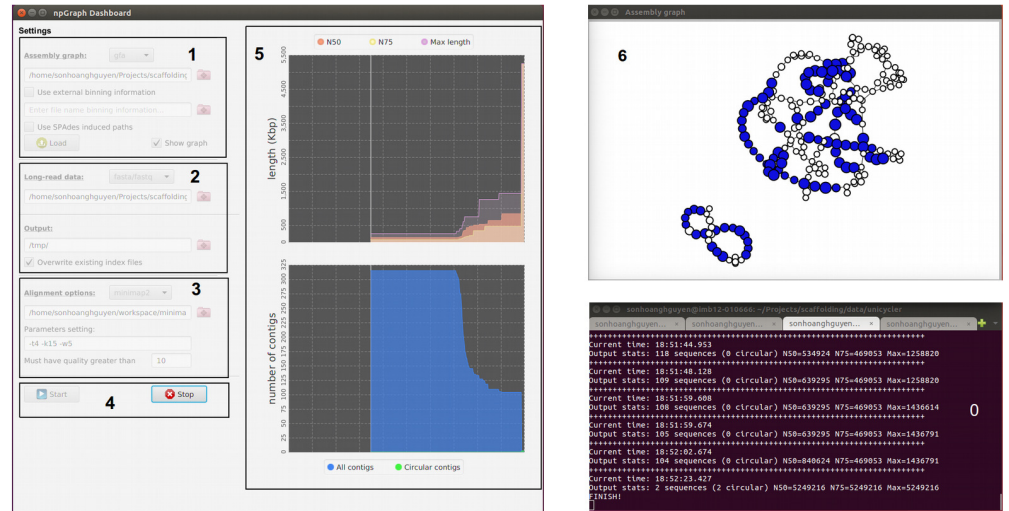


Fig 2. npGraph user interface including Console (0) and GUI components (1-6). The GUI consists of the Dashboard (1-5) and the Graph View (6). From the Dashboard there are 5 components as follow: 1 the assembly graph input field; 2 the long reads input field; 3 the aligner settings field; 4 control buttons (start/stop) to monitor the real-time scaffolding process; 5 the statistics plots for the assembly result.

Algorithm in details

The work flow of **npGraph** mainly consists of 3 stages: (1) assembly graph pre-processing; (2) graph resolving and simplifying; (3) post-processing and reporting results. The first step is to load the assembly graph of Illumina contigs and analyze its components' property, including binning and multiplicity estimation. The second step works on the processed graph and the long read data that can be provided in real-time by ONT sequencer. Based on the paths induced from long reads, the assembly graph will be resolved on-the-fly. Finally, the graph is subjected to the last attempt of resolving and cleaning, as well as output the final results. The whole process can be managed by using either command-line interface or GUI. Among three phases, only the

first one must be performed prior to the MinION sequencing process in a streaming setup. The algorithm works on the assembly graph of Illumina contigs, so the terms *contigs* and *nodes* if not mentioned specifically, would be used interchangeable throughout this context.

Contigs binning

Contigs should belong to single or multiple groups, or *bins*, that would represent different assembly units, *e.g.* chromosome, plasmids, of different species if applying to a metagenomics dataset. A binning step is needed to assign membership of each contig to its corresponding group.

The first step is to cluster the big anchors (longer than 10Kbp and in/out degree less than 2) based on their *kmer* coverage. To achieve this, we applied DBSCAN clustering algorithm [8], which utilises a customised metric function to map contigs into a one-dimensional space. In order to define a customised metric which is **sample simple** and fast to calculate, we assumed that a single long contig itself consists of a Poisson distribution of *k-mers* count with the mean approximated by the contig's coverage. The metric is then determined by a distance function of two Poisson distributions based on Kullback-Leibler divergence (or relative entropy) between the Poisson distribution representing each contig [9].

Formally, assuming there are 2 Poisson distributions P_1 and P_2 with probability mass functions (PMF)

$$P_1(X = x, \lambda_1) = \frac{e^{-\lambda_1} \lambda_1^x}{x!}$$

and

$$P_2(X = x, \lambda_2) = \frac{e^{-\lambda_2} \lambda_2^x}{x!}$$

The Kullback-Leibler divergence from P_2 to P_1 is defined as:

$$D_{KL}(P_1||P_2) = \sum_x P_1(x) \log \frac{P_1(x)}{P_2(x)} = \mathbb{E}_{P_1} [\log \frac{P_1(x)}{P_2(x)}]$$

or in other words, it is the expectation of the logarithmic difference between the probabilities P_1 and P_2 , where the expectation is taken using P_1 . The log ratio of the PMFs is:

$$\log \frac{P_1(x)}{P_2(x)} = \log (e^{\lambda_2 - \lambda_1} (\frac{\lambda_1}{\lambda_2})^x) = x \log \frac{\lambda_1}{\lambda_2} + \lambda_2 - \lambda_1$$

Thus the divergence between P_1 and P_2 is:

$$D_{KL}(P_1||P_2) = \mathbb{E}_{P_1} [\log \frac{P_1(x)}{P_2(x)}] = \lambda_1 \log \frac{\lambda_1}{\lambda_2} + \lambda_2 - \lambda_1$$

Thus, the metric we used is a distance function defined as:

$$D(P_1, P_2) = \frac{D_{KL}(P_1||P_2) + D_{KL}(P_2||P_1)}{2} = \frac{1}{2}(\lambda_1 - \lambda_2)(\log \lambda_1 - \log \lambda_2)$$

Independent from the contigs clustering in the pre-processing step, additional evidence of nodes' uniqueness can be acquired using the long reads during the assembly process. Given enough data, the multiplicity of an ambiguous node can be determined based on the set of all bridges rooted from itself. On the other hand, external binning tools such as MetaBAT [10], maxbin [11] can be employed in **npGraph** as well.

Multiplicity estimation

Now bins of the main unique contigs had been identified, however, they only make up a certain proportion of the contigs set. From here, we need to assign bin membership and multiplicity for all other nodes of the graph, especially the repetitive ones. To do so, we relied on the graph's topology and the estimated read coverage of initial contigs from SPAdes. Given all contigs' coverage values as nodes' weight, we need to estimate those of edges and in return, using them to re-estimate the coverage for repetitive nodes if necessary. After this process, we will have a graph with optimized weighted components that would suggest their multiplicities more exactly. Basically the computation is described as in following steps:

0. Initialize every node weight as its corresponding contig coverage, all edges' weight as zeros.
1. Calculate distributed weights for edges by quadratic unconstrained optimization of the least-square function:

$$\frac{1}{2} \sum_i l_i ((\sum e_i^+ - c_i)^2 + (\sum e_i^- - c_i)^2)$$

where l_i and c_i is the length and weight of a node i in the graph;

$\sum e_i^+$ and $\sum e_i^-$ indicates sum of weights for incoming and outgoing edges from node i respectively. They are expected to be as close to c_i as possible thus the length-weighted least-square should be minimized.

The above function can be rewritten as:

$$f(x) = \frac{1}{2} x^T Q x + b^T x + r$$

and then being minimized by using gradient method.

2. Re-estimate weights of repetitive nodes based on their neighboring edges' measures and repeat previous optimization step. The weights are calculated iteratively until no further significant updates are made or a threshold of loop count is reached.

At this point, we can induce the copy numbers of nodes in the final assembly. For each node, this could be done by investigating its adjacent edges' multiplicity to estimate how many times it should be visited and from which bin(s). Multiplicities of insignificant nodes (of sequences with length less than 1,000 bp) are less confident due to greater randomness in sequencing coverage. For that reason, in **npGraph**, we did not rely on them for graph transformation but as supporting information for path finding.

Building bridges in real-time

Bridge is the data structure designed for tracking the possible connections between two anchored nodes (of unique contigs) in the assembly graph. A bridge must start from a unique contig, or *anchor* node, and end at another when completed. Located in-between are nodes known as *steps* and distances between them are called *spans* of the bridge. Stepping nodes are normally repetitive contigs and indicative for a path finding operation later on. In a complicated assembly graph, the more details the bridge, *a.k.a.* more steps in-between, the faster and more accurate the linking path it would resolve. A bridge's function is complete when it successfully return the ultimate linking path between 2 anchors.

The real-time bridging method considers the dynamic aspect of multiplicity measures for each node, meaning that a n -times repetitive node might become a unique

node at certain time point when its $(n - 1)$ occurrences have been already identified in other distinct unique paths. Furthermore, the streaming fashion of this method allows the bridge constructions (updating steps and spans) to be carried out progressively so that assembly decisions can be made immediately after having sufficient supporting data. A bridge in **npGraph** has several completion levels. When created, it must be rooted from an *anchor node* which represents a unique contig (level 1). A bridge is known as fully complete (level 4) if and only if there is a unique path connecting its two anchor nodes from two ends.

At early stages (level 1 or 2), a bridge is constructed progressively by alignments from long reads that spanning its corresponding anchor(s). In an example from Figure 1(a), bridges from a certain anchor (highlighted in red) are created by extracting appropriate alignments from incoming long reads to the contigs. Each of the steps therefore is assigned a weighing score based on its alignment quality. Due to the error rate of long reads, there should be deviations in terms of steps found and spans measured between these bridges, even though they represent the same connection. A continuous merging phase, as shown in the figure, takes advantage of a pairwise Needleman-Wunsch dynamic programming to generate a consensus list based on weight and position of each of every stepping nodes. The spans are calibrated accordingly by averaging out the distances. On the other hand, the score of the merged steps are accumulated over time as well. Whenever a consensus bridge is anchored by 2 unique contigs at both ends and hosting a list of steps with sufficient coverage, it is ready for a path finding in the next step.

Path finding algorithm

Given a bridge with 2 anchors, a path finding algorithm is invoked to find all candidate paths between them. Each of these paths is given a score of alignment-based likelihood which are updated immediately as long as there is an appropriate long read being generated by the sequencer. As more nanopore data arrives, the divergence between candidates' score becomes greater and only the top-scored ones are kept for the next round. We implement a modified stack-based version utilizing Dijkstra's shortest path finding algorithm [12] to reduce the search space when using Depth-First Search.

Due to false **positive** alignments from shorter contigs to the long reads, not all of the reported step nodes are necessary to be appeared **would appear** in the ultimate path resolved by the bridge. In most cases, the accumulated score of each step indicates its likelihood to be the true component of the final solution. For that reason, a strategy similar to binary searching is employed to find a path across 2 anchors of a bridge as shown in Algorithm 2. **Briefly, it recursively looks for sub-paths connecting pairs of step nodes in-between 2 anchors and return a final continuous path with the highest score.** Before that, We define Algorithm 1 as the *base* sub-routine to demonstrates the path finding algorithm for find paths between a *connectable* pair of two **step** nodes given their estimated distance. In which, function

$$\text{shortestTree}(\overrightarrow{\text{vertex}}, \text{distance}) : (V, Z) \rightarrow V^n$$

from line 3 of the algorithm's pseudo-code builds a shortest tree rooted from \overrightarrow{v} , following its direction until a distance of approximately d (with a tolerance regarding nanopore read for the nanopore error rate) is reached. This task is implemented based on Dijkstra algorithm. This tree is used on line 4 and in function *includedIn()* on line 19 to filter out any node or edge with ending nodes that do not belong to the tree.

Basically, the algorithm keeps track of a stack that contains sets of candidate edges to discover. During the traversal, a variable d is updated as an estimation for the distance to the target. A hit is reported if the target node is reached with a reasonable

Algorithm 1: Pseudo-code for finding paths connecting 2 nodes given their estimated distance.

Data: Assembly graph $G\{V, E\}$

Input: Pair of bidirected nodes \vec{v}_1, \vec{v}_2 and estimated distance d between them

Output: Set of candidate paths connecting \vec{v}_1 to \vec{v}_2 with reasonable distances compared to d

```

1 Function DFS( $\vec{v}_1, \vec{v}_2, d$ ):
2    $P := \text{new List}()$ 
3    $M := \text{shortestTree}(\vec{v}_2, d)$            // build shortest tree from  $\vec{v}_2$  with range  $d$ 
4   if  $M.\text{contain}(\vec{v}_1)$  then
5      $S := \text{new Stack}()$                  // stack of sets of edges to traverse
6      $\text{edgesSet} := \text{getEdges}(\vec{v}_1)$        // get all bidirected edges going from  $\vec{v}_1$ 
7      $S.\text{push}(\text{edgesSet})$ 
8      $p := \text{new Path}(\vec{v}_1)$              // init a path that has  $\vec{v}_1$  as root
9     while true do
10       $\text{edgesSet} := S.\text{peek}()$ 
11      if  $\text{edgesSet}.\text{isEmpty}()$  then
12        if  $p.\text{size}() \leq 1$  then
13          break // stop the loop when there is no more edge to discover
14         $S.\text{pop}()$ 
15         $d += p.\text{peekNode}.\text{length}() + p.\text{popEdge}().\text{length}()$ 
16      else
17         $\text{curEdge} := \text{edgesSet}.\text{remove}()$ 
18         $\vec{v} := \text{curEdge}.\text{getOpposite}(p.\text{peekNode}())$ 
19         $S.\text{push}(\text{getEdges}(\vec{v}).\text{includedIn}(M))$ 
20         $p.\text{add}(\text{curEdge})$ 
21        if  $\text{reach } \vec{v}_2 \text{ with reasonable } d$  then
22           $P.\text{add}(p)$ 
23         $d = \vec{v}.\text{length}() + \text{curEdge}.\text{length}()$ 
24   return  $P$ 

```

Algorithm 2: Recursive binary bridging to connect 2 anchor nodes.

Data: Assembly graph $G\{V, E\}$
Input: Bridge $B : \{\vec{v}_0, \dots, \vec{v}_k, \dots, \vec{v}_n\}$ with \vec{v}_0 and \vec{v}_n are two anchors, $\{\vec{v}_k\}, k = 1 \dots (n - 1)$ are steps in-between
Output: Set of candidate paths connecting \vec{v}_0 to \vec{v}_n that maximize the likelihood of the step list.

```
1 Function BinaryBridging( $B$ ):  
    /* search for the contig with maximum score from the step list (two ends  
       excluded) */  
2    $m := \text{argmax}_k(\vec{v}_k.\text{score}())$   
    /* if there is no step in-between, run path finding algorithm above directly  
       and return the result */  
3   if  $M.\text{size}() \equiv 2$  then  
4     return DFS( $B.\text{start}(), B.\text{end}().B.\text{distance}()$ )  
    /* divide the original bridge  $B$  into 2 bridges by  $v_m$ :  $BL$  and  $BR$  */  
5    $BL := \{B.\text{start}(), \dots, \vec{v}_m\}$   
6    $BR := \{\vec{v}_m, \dots, B.\text{end}()\}$   
    /* Return the join of running recursive function on two sub-bridges */  
7   return BinaryBridging( $BL$ )  $\bowtie$  BinaryBridging( $BR$ )
```

distance *i.e.* close to zero, within a given tolerance (line 21). A threshold for the traversing depth is set (150) to ignore too complicated and time-consuming path searching.

Note that the *length()* functions for node and edge are totally different. While the former returns the length of the sequence represented by the node, *i.e.* contig from short-read assembly, the latter is usually negative because an edge models a link between two nodes, which is normally an overlap (except for composite edges). For example, in a *k-mers* SPAdes assembly graph, the value of an edge is $-k + 1$.

In many cases, due to dead-ends, there not always exist a path in the assembly graph connecting two anchors as suggested by the alignments. In this case, if enough long reads coverage (20X) are met, a consensus module is invoked and the resulting sequence is contained in a *pseudo* edge.

Graph simplification in real-time

npGraph resolves the graph by reducing its complexity perpetually using the long reads that can be streamed in real-time. Whenever a bridge is finished (with a unique linking path), the assembly graph is *transformed* or *reduced* by replacing its unique path with a composite edge and removing any unique edges (edges coming from unique nodes) along the path. The assembly graph would have at least one edge less than the original after the reduction. The nodes located on the reduced path, other than 2 ends, also have their multiplicities subtracted by one and the bridge is marked as finally resolved without any further modifications.

Figure 3 presents an example of the results before and after graph resolving process in the GUI. The result graph, after cleaning, would only report the significant connected components that represents the final contigs. Smaller fragments, even unfinished but with high remaining coverage, are also presented as potential candidates for further downstream analysis. Further annotation utility can be implemented in the future better monitoring the features of interests as in **npScarf**.

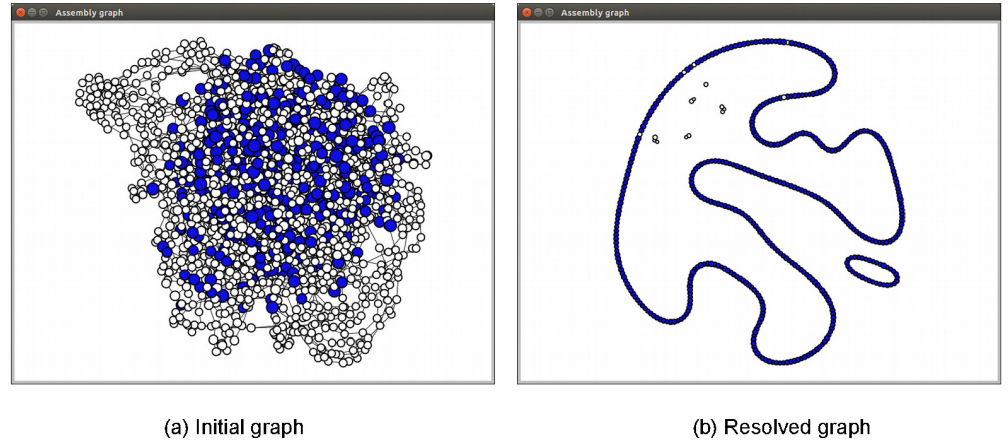


Fig 3. Assembly graph of *Shigella dysenteriae* Sd197 synthetic data being resolved by npGraph and displayed on the GUI Graph View. The SPAdes assembly graph contains 2186 nodes and 3061 edges, after the assembly shows 2 circular paths representing the chromosome and one plasmid.

Result extraction and output

npGraph reports assembly result in real-time by decomposing the assembly graph into a set of longest straight paths (LSP), each of the LSP will spell a contig in the assembly report. The final assembly output contains files in both FASTA and GFAv1 format (<https://github.com/GFA-spec/GFA-spec>). While the former only retains the actual genome sequences from the final decomposed graph, the latter output file can store almost every properties of the ultimate graph such as nodes, links and potential paths between them.

A path $p = \{v_0, e_1, v_1, \dots, v_{k-1}, e_k, v_k\}$ of size k is considered as straight if and only if each of every edges along the path $e_i, \forall i = 1, \dots, k$ is the only option to traverse from either v_{i-1} or v_i , giving the transition rule. To decompose the graph, the tool simply mask out all incoming/outgoing edges rooted from any node with in/out degree greater than 1 as demonstrated in Figure 4. These edges are defined as branching edges which stop straight paths from further extending.

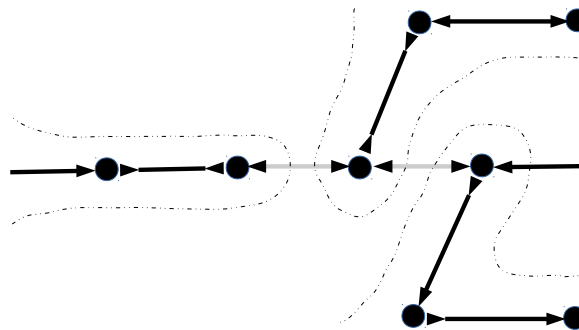


Fig 4. Example of graph decomposition into longest straight paths. Branching edges are masked out (faded) leaving only straight paths (bold) to report. There would be 3 contigs extracted by traversing along the straight paths here.

The decomposed graph is only used to report the contigs that can be extracted from

an assembly graph at certain time point. For that reason, the branching edges are only masked but not removed from the original graph as they would be used for further bridging.

Other than that, if GUI mode is enabled, basic assembly statistics such as N50, N75, maximal contigs length, number of contigs can be visually reported to the users in real-time beside the Dashboard. The progressive simplification of the assembly graph can also be observed at the same time in the Graph view.

Results

Evaluation using synthetic data

To evaluate the performance of the method, **npGraph** 1.1 was tested along with **SPAdes**, **SPAdes** hybrid from version 3.13.1, [13], **npScarf** (japsa 1.7-02), and **Unicycler** version 0.4.6 using **Unicycler**'s synthetic data set [14]. The data set is a simulation of Illumina and MinION raw data, generated *in silico* based on available microbial references. We ran hybrid assembly methods using the entire nanopore data and the reciprocal results were evaluated by QAST 5.0.2 [15].

Table 1 shows comparative results running different methods on 5 synthetic data sets, simulated from complete genomes of *Mycobacterium tuberculosis* H37Rv, *Klebsiella pneumoniae* 30660/NJST258_1, *Saccharomyces cerevisiae* S288c, *Shigella sonnei* 53G and *Shigella dysenteriae* Sd197.

To align the long reads to the assembly graph components, both **BWA-MEM** [16] or **minimap2** [17] were used in conjunction with **npGraph**. These two methods were chosen due to their proven efficiency and compatibility with streaming data. While **BWA-MEM** is a well-known classic aligner that can be adapted to work with third generation sequencing data, **minimap2** has been specially designed for this data type. We observed a slightly higher error rate (comprising the sum of mismatches and indels per 100kb) using **BWA-MEM** in comparison to **minimap2** for all simulations in Table 1. This is due to the fact that bridging paths induced using **BWA-MEM** were slightly less accurate due to more noise from the smaller *steps* in-between (Figure 1(a)). However, under almost circumstances, using either aligner resulted in final assemblies with comparable qualities. In terms of running time and resources required, **minimap2** always proved to be a better option, requiring markedly less CPU time than **BWA-MEM**. Utilising **minimap2**, **npGraph** is now the fastest hybrid assembler available.

Amongst all assemblers, **Unicycler** applies an algorithm based on semi-global (or glocal) alignments [18] with the consensus long reads generated with the **SeqAn** library. With all of the data sets tested, **Unicycler** required the most computational resources, but it also returned fewer mis-assemblies than the other approaches with a comparable rate of error (indels and mismatches) to **npGraph**. **hybridSPAdes** reported decent results with high fidelity at base level. As the trade-off, there were fewer connections satisfying its quality threshold, resulting in the fragmented assemblies with lower N50 compared to the other hybrid assemblers. This behaviour was clearly reflected in the last, also the most challenging task of assembly *S. dysenteriae*.

Of the two streaming algorithms, **npScarf** utilizes a fast but greedy scaffolding approach that can lead to mis-assemblies and errors. For bacterial genomes with modest complexity these are minimal (e.g. *K. pneumoniae*), but for those with severe repetitive elements, extra calibrations are needed to prevent the mis-assembly due to ambiguous alignments. On the other hand, **npGraph** significantly reduced the errors compared to **npScarf**, sometimes even proved to be the best option e.g. for *M. tuberculosis* and *K. pneumoniae*. For the yeast *S. cerevisiae* data set, the **npGraph** assembly best covered the reference genome but the number of mis-assemblies was up to 5. The unfavourable

Table 1. Comparison of assemblies produced in batch-mode using **npGraph** and other hybrid assembly methods on synthetic data.

Method	Assembly size (Mbp)	#Contigs	N50 (Kbp)	Mis-assemblies	Error (per 100 Kbp)	Run times (CPU hrs)
<i>M. tuberculosis</i> H37Rv 4,411,532 bp						
SPAdes	4.376	66	150.7	0	0.23	1.42
SPAdes hybrid	4.411	1	4410.5	0	0.86	1.61
Unicycler	4.412	1	4411.5	0	2.56	5.52
npScarf	4.432	4	4402.2	7	6.61	1.42 + 0.7
npGraph (bwa)	4.411	1	4411.4	0	2.63	1.42 + 0.64
npGraph (minimap2)	4.412	1	4411.5	0	0.68	1.42 + 0.02
<i>K. pneumoniae</i> 30660 5,540,936 bp						
SPAdes	5.469	64	270.2	0	0.07	1.36
SPAdes hybrid	5.543	8	4229.1	2	5.04	1.63
Unicycler	5.538	9	5263.2	0	1.85	4.34
npScarf	5.566	7	5259.1	4	35.6	1.36 + 0.95
npGraph (bwa)	5.535	5	5263.2	1	4.16	1.36 + 0.92
npGraph (minimap2)	5.541	6	5263.2	0	0.85	1.36 + 0.04
<i>S. cerevisiae</i> S288c 12,157,105 bp						
SPAdes	11.675	194	260.5	0	1.57	3.61
SPAdes hybrid	11.910	45	770.5	5	34.52	4.15
Unicycler	11.837	29	909.1	0	22.83	16.34
npScarf	11.990	22	796.8	53	85.5	3.61 + 4.35
npGraph (bwa)	12.000	151	913.1	3	38.68	3.61 + 4.12
npGraph (minimap2)	12.008	148	913.1	5	25.32	3.61 + 0.13
<i>S. sonnei</i> 53G 5,220,473 bp						
SPAdes	4.796	392	27.7	0	0.44	1.1
SPAdes hybrid	5.218	8	2195.5	2	41.98	1.36
Unicycler	5.221	5	4988.5	0	7.91	9.64
npScarf	6.426	20	1293.8	84	366.04	1.1 + 0.52
npGraph (bwa)	5.293	97	4988.5	3	14.87	1.1 + 0.57
npGraph (minimap2)	5.293	97	4988.5	3	8.31	1.1 + 0.08
<i>S. dysenteriae</i> Sd197 4,560,911 bp						
SPAdes	4.096	534	14.4	1	0.68	1.19
SPAdes hybrid	4.486	23	821.2	96	10.99	1.89
Unicycler	4.561	3	4369.2	0	12.96	8.46
npScarf	-	-	-	-	-	1.19 + -
npGraph (bwa)	4.553	3	4369.1	6	91.03	1.19 + 0.76
npGraph (minimap2)	4.548	3	4364.1	8	83.68	1.19 + 0.14

figures, namely mis-assemblies and error, were still high in case of *S. dysenteriae*, due to the complicated and extremely fragmented graph components containing a large number of small-scaled contigs that were difficult to map with nanopore data. The progressive path finding module tried to induce the most likely solution from a stream of coarse-grained alignments, without fully succeeding.

Hybrid assembly for real data sets

A number of sequencing data sets from *in vitro* bacterial samples [19] were used to further explore differences in performance between **npGraph** and **Unicycler**. The data included both Illumina paired-end reads and MinION sequencing based-call data for

each sample. Due to the unavailability of reference genomes, there were fewer statistics reported by QUAST for the comparison of the results. Instead, we investigated the number of circular sequences and PlasmidFinder 1.3 [20] mappings to obtain an evaluation on the accuracy and completeness of the assemblies (Table 2) on three data sets of bacterial species *Citrobacter freundii*, *Enterobacter cloacae* and *Klebsiella oxytoca*.

Table 2. Assembly of real data sets using Unicycler and npGraph with the optimized SPAdes output. Circular contigs are highlighted in **bold**, fragmented assemblies are presented as X|Y where X is the total length and Y is the number of supposed contigs making up X.

	Unicycler	npGraph	Replicons (based on PlasmidFinder 1.3)
<i>Citrobacter freundii</i> CAV1374	5,029,534 109,688 100,873 85,575 43,621 3,223 1,916 14,464 3	5,029,486 109,688 100,873 85,575 43,621 3,223 1,916 14,456 2	Chromosome IncFIB(pHCM2)_1_pHCM2.AL513384 IncFIB(pB171)_1_pB171.AB024946 IncL/M(pMU407)_1_pMU407.U27345 repA_1.pKPC-2_CP013325 - ColRNAI.1_DQ298019 -
<i>Enterobacter cloacae</i> CAV1411	4,806,666 2 90,451 33,610 13,129 2	4,858,438 2 90,693 2 33,610 14,542 4	Chromosome IncR_1_DQ449578 repA_1.pKPC-2_CP013325 -
<i>Klebsiella oxytoca</i> CAV1015	6,153,947 5 113,105 111,395 108,418 76,183 11,638	6,155,762 113,105 111,395 109,209 13 76,186 11,892 2	Chromosome IncFII(SARC14)_1_SARC14.JQ418540; IncFII(S)_1_CP000858 - IncFIB(K)_1_Kpn3.JN233704 IncL/M(pMU407)_1_pMU407.U27345 -

There was high similarity between final contigs generated by two assemblers on all of these datasets. For the *Citrobacter freundii* dataset, they share the same number of circular sequences, including the chromosomal and other six replicons contigs in the , with only 48 nucleotides difference in the length of the main chromosome. Five out of six identical replicons could be confirmed as plasmids based on the occurrence of origin of replication sequences from the PlasmidFinder database. In detail, two megaplasids (longer than 100Kbp) were classified as IncFIB while the other two mid-size replicons, 85.6Kbp and 43.6Kbp, were incL and repA respectively, leaving the shortest one with 2Kbp of length as ColRNAI plasmid. The remaining circular sequence without any hits to the database was 3.2Kbp long suggesting that it could be phage or a cryptic plasmid. Both assemblers had 14.5Kbp of unfinished sequences split amongst 3 linear contigs from Unicycler and 2 for npGraph.

The assembly task for *Enterobacter cloacae* was more challenging and the chromosomal DNA remained fragmented in two contigs for both methods (of length 3.324Mbp and 1.534Mbp for npGraph compared to 2.829Mbp and 1.978Mbp for Unicycler). Both methods detected two plasmids (IncR and repA), and Unicycler returned complete circular sequences for both plasmids, while npGraph returned circular sequence for one plasmid, while the other was fragmented into two contigs. Similar to the assembly of *Citrobacter freundii* , there was around 14Kbp of data which was unable to be finished by the assemblers (split into 2 and 4 contigs for Unicycler and npGraph respectively).

Finally, the assembly for *Klebsiella oxytoca* saw fragmented chromosome using

Unicycler (with 5 contigs) which was a fully complete single contig for **npGraph** with 6.156Mbp of size. The two assemblers shared 3 common circular sequences of which two were confirmed plasmids. The first identical sequence represented a megaplasmid ($\simeq 113$ Kbp) with two copies of IncFII origin of replication DNA being identified. The other 76Kbp plasmid circularised by both was IncL/M with of length. The third circular contig of length 111Kbp returned no hits to the plasmid database, suggesting the importance of *de novo* replicon assembly in combination with further interrogation. **Unicycler** detected another megaplasmid of size 108.4Kbp which was fractured by **npGraph**. A fragmented contig was also observed in **npGraph** for the final contig of length 11.6Kbp where it failed to combine two smaller sequences into one.

In addition to what is presented in Table 2, dot plots for the pair-wise alignments between the assembly contigs were generated and can be found in S1 Fig. This identified a structural difference between **npGraph** and **Unicycler** assembly for the *E. cloacae* CAV1411 genome assembly. This was caused by the inconsistency of a fragment’s direction on the final output contigs. Comparison to a reference genome from the same bacteria strain (GenBank ID: CP011581.1 [21]), demonstrated that contigs generated by **npGraph** produced consistent alignment, but not those generated by **Unicycler** (S2 Fig.). However, we cannot at this stage rule out genuine structural variation between the two samples .

Assembly performance on streaming data

In order to investigate the rate at which the two streaming hybrid assembly algorithms completed bacterial assemblies, we plot the N50 as a function of long-read coverage on the 4 datasets described in the previous section (Figure 5). This revealed that **npGraph** and **npScarf** both converge to the same ultimate completeness but at different rates. **npScarf** converged more quickly than **npGraph**, due to the fact that it is able to build bridges with only 1 spanning long-read, whereas **npGraph** requires 3 reads. Unlike **npScarf** where the connections could be undone and rectified later if needed, a bridge in **npGraph** will remain unchanged once created. The plot for *E. coli* data clarifies this behaviour when a fluctuation can be observed in **npScarf** assembly at $\simeq 3$ -folds data coverage. On the other hand, the N50 length of **npGraph** is always a monotonic increasing function. The sharp *jumping* patterns suggested that the linking information from long-read data had been stored and exploited at certain time point decided by the algorithm. In addition, at the end of the streaming when the sequencing is finished, **npGraph** will try for the last time to connect bridges with less than 3 supporting reads which are otherwise not part of conflicting bridges.

Discussion

Streaming assembly methods have been proven to be useful in saving time and resources compared to conventional batch algorithms with examples including *e.g.* **Faucet** [22] and **npScarf** [2]. The first method allows the assembly graph to be constructed incrementally as long as reads are retrieved and processed. This practice is helpful dealing with huge short-read data set because it can significantly reduce the local storage for the reads, as well as save time for a De Bruijn graph (DBG) construction while waiting for the data being retrieved. **npScarf**, on the other hand, is a hybrid assembler working on a pre-assembly set of short-read assembly contigs. It functions by scaffolding the contigs using real-time nanopore sequencing. The completion of genome assembly in parallel with the sequencing run provides explicit benefits in term of resource control and turn-around time for analysis [2].

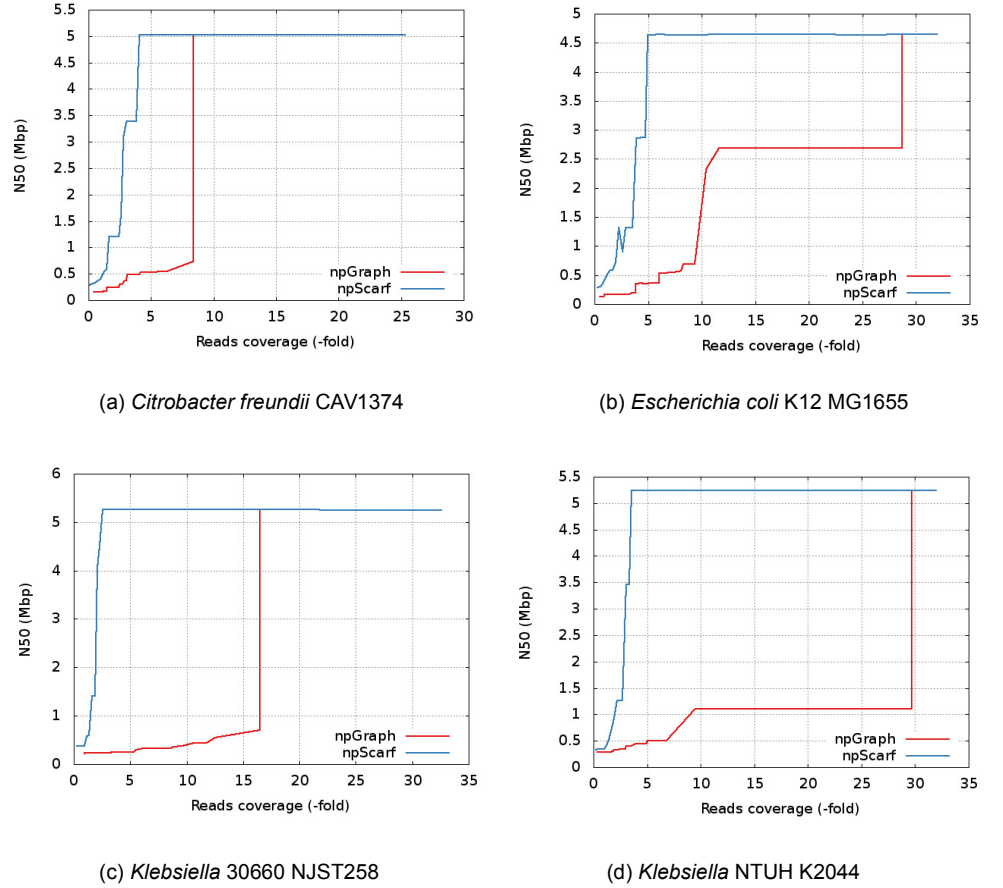


Fig 5. N50 statistics of real-time assembly by **npScarf** and **npGraph** for 4 bacterial samples.

Hybrid approaches are still common practice in genome assembly and data analyses while Illumina sequencing retains cost and accuracy benefits over long-read sequencing. On the other hand, the third-generation sequencing methods such as Pacbio or Oxford Nanopore Technology are well-known for the ability to produce much longer reads that can further complete the Illumina assembly. As a consequence, it is rational to combine two sources of data together in a hybrid method that can offer accurate and complete genomes at the same time. **npScarf**, following that philosophy, had been developed and deployed on real microbial genomes.

However, due to the greedy bridging approach of the contig-based streaming algorithm, **npScarf**'s results can suffer from mis-assemblies [14, 23]. A default setting was optimized for microbial genomes input but cannot fit for all data from various experiments in practice. Also, the gap filling step has to rely on the lower accuracy nanopore reads thus the accuracy of the final assembly is also affected. To tackle the quality issue while maintaining the streaming feature of the approach, a bridging method by assembly graph traversing has been proposed in this manuscript. Our approach uses as its starting point a compact DBG assembly graph, followed by graph-traversal, repeat resolution and identification of the longest possible un-branched paths that would represents contigs for the final assembly.

Hybrid assembler using nanopore data to resolve the graph has been implemented in

hybridSPAdes [13] as well as **Unicycler** [14]. The available tools employ batch-mode algorithms on the whole long-read data set to generate the final genome assembly. The **SPAdes** hybrid assembly module, from its first step, exhaustively looks for the most likely paths (with minimum edit distance) on the graph for each of the long read given but only ones supported by at least two reads are retained. In the next step, these paths will be subjected to a decision-rule algorithm, namely **exSPAndeR** [24], for repeat resolution by step-by-step expansion, before output the final assembly. On the other hand, **Unicycler**'s hybrid assembler will initially generate a consensus long read for each of the bridge from the batch data. The higher quality consensus reads are used to align with the assembly graph to find the best paths bridging pairs of anchored contigs. While this method employs the completeness of the data set from the very beginning for a consensus step, the former only iterates over the batch of possible paths and relies on a scoring system for the final decision of graph traversal. Hence, in theory it can be adapted to a real-time pipeline.

The challenge in adapting graph-based approaches into streaming algorithm comes mainly from building a progressive implementation for path-finding and graph reducing module. To achieve this, we apply a modified DFS (depth-first search) mechanism and a dynamic voting algorithm into an on-the-fly graph resolver.

By testing with synthetic and real data, we have shown that **npGraph** can generate assemblies of comparative quality compared to other powerful batch-mode hybrid assemblers, such as **hybridSPAdes** or **Unicycler**, while also providing the ability to build and visualise the assembly in real-time.

Conclusion

Due to the limits of current sequencing technology, application of hybrid methods should remain a common practice in whole genome assembly for the near future. On the other hand, the ONT platforms are evolving quickly with significant improvement in terms of data accuracy and yield and cost. Beside, the real-time property of this technology has not been sufficiently exploited to match its potential benefits. **npScarf** had been introduced initially to address these issues, however, the accuracy of the assembly output was affected by its greedy alignment-based scaffolding approach. Here we present **npGraph**, a streaming hybrid assembly method working on the DBG assembly graph that is able to finish short-read assembly in real-time while minimizing the errors and mis-assemblies drastically.

Compared to **npScarf**, **npGraph** algorithm employs more rigorous approach based on graph traversal. This might reduce the assembly errors because the bridging method is more accurate so that the reporting results are more reliable. The performance of **npGraph** is comparable to **Unicycler** while consuming much less computational resources so that it can work on streaming mode. Also, the integrated GUI allows users to visualize its animated output in a more efficient way.

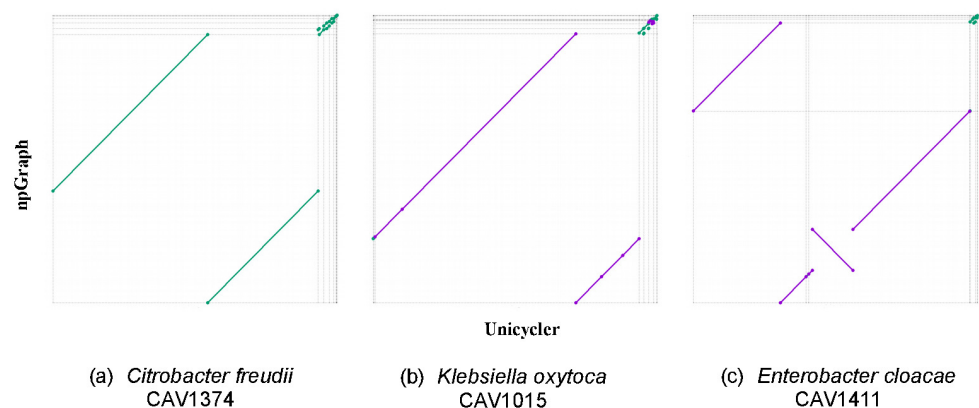
On the other hand, similar to **Unicycler**, **npGraph** relies on the initial assembly graph to generate the final assembly. The algorithm operates on the assumption of a high quality assembly from a well-supplied source of short-read data for a decent assembly graph to begin with. It then consumes a just-enough amount of data from a streaming input of nanopore reads to resolve the graph. Finally, extra pre-processing and comprehensive binning on the initial graph could further improve the performance of the streaming assembler.

Supporting information

S1 Fig. Dotplot generated by MUMmer for assembly results of Unicycler versus npGraph.

421

422

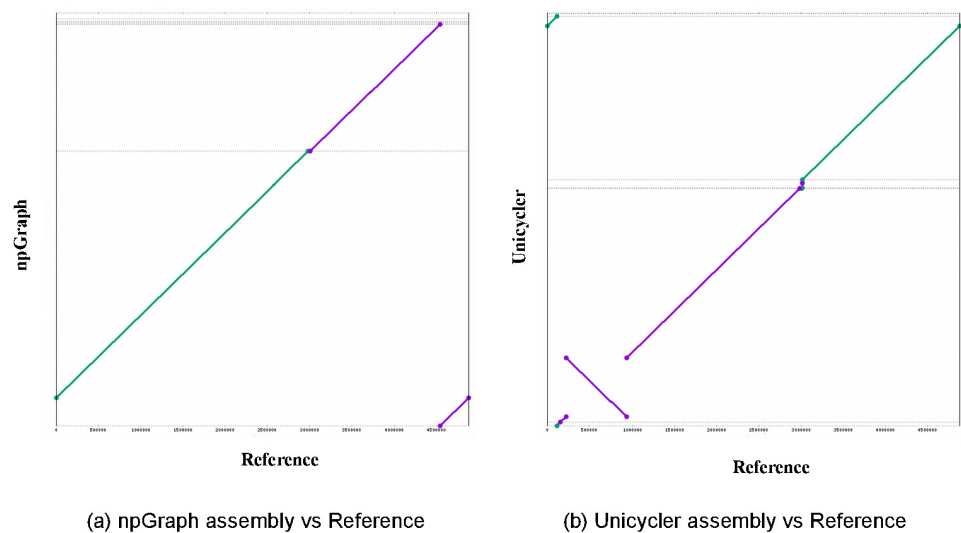


Supporting Figure S1. Dotplot generated by MUMmer for assembly results of Unicycler versus npGraph. Structural agreements between two methods were found in (a) *C.freundii* and (b) *K.oxytoca* assembly contigs. On the other hand, for (c) *E.cloacae* sample, there was a disagreement detected between 2 largest contigs given by two assembly algorithms.

S2 Fig. Alignments of an *Enterobacter cloacae* reference genome to assembly sequences generated by (a) Unicycler and (b) npGraph

423

424



Supporting Figure S2. Alignments of an *Enterobacter cloacae* reference genome to assembly sequences generated by (a) ~~Unicycler~~npGraph and (b) npGraph~~Unicycler~~. The former suggests a structural variant, the latter is virtually an 1-to-1 mapping.

S1 Table. Benchmarking different methods using **lrscaf**, npScarf, npGraph, hybridSPAdes and Unicycler hybrid assembler with the synthetic data set.

425
426

Method	Assembly size (bp)	#Contigs	N50 (bp)	Mis- assemblies	Mismatch (per 100Kbp)	Indel (per 100Kbp)
random sequences no repeats						
lrscaf	-	-	-	-	-	-
npScarf	4110000	3	4000000	0	0.00	0.00
npScarf_wag	4109516	3	4000000	0	0.00	0.00
npGraph-bwa	4110000	3	4000000	0	0.00	0.00
npGraph-mm2	4110000	3	4000000	0	0.00	0.00
hybridSPAdes	4110231	3	4000077	0	0.00	0.07
Unicycler	4110000	3	4000000	0	0.00	0.00
random sequences some repeats						
lrscaf	4113104	6	4002298	0	0.00	0.00
npScarf	4110940	3	4002715	0	0.00	0.66
npScarf_wag	4437094	7	2795129	0	0.00	1.07
npGraph-bwa	4110000	3	4000000	0	0.00	0.00
npGraph-mm2	4110000	3	4000000	0	0.00	0.00
hybridSPAdes	4107566	3	3999364	0	0.02	0.02
Unicycler	4110000	3	4000000	0	0.00	0.00
random sequences many repeats						
lrscaf	4119020	12	4002117	0	0.03	0.79
npScarf	4316934	8	3963485	24	11.55	34.54
npScarf_wag	5215965	16	1515563	37	0.32	7.22
npGraph-bwa	4110000	3	4000000	0	0.32	0.15
npGraph-mm2	4110000	3	4000000	0	0.32	0.15
hybridSPAdes	4108190	3	3999621	0	0.68	0.15
Unicycler	4110000	3	4000000	0	0.32	0.15
<i>Acinetobacter</i> A1						
lrscaf	3896799	4	3886381	0	0.47	0.36
npScarf	3912299	3	3870269	4	4.74	13.02
npScarf_wag	3945166	3	3906368	1	7.00	14.02
npGraph-bwa	3918374	2	3909643	0	16.34	0.61
npGraph-mm2	3885898	2	3877167	1	18.05	1.03
hybridSPAdes	3929948	53	3353679	0	35.48	3.22
Unicycler	3917745	2	3909014	0	2.50	0.13
<i>Acinetobacter</i> AB30						
lrscaf	4367657	27	1035967	13	30.11	1.42
npScarf	4512464	7	4304628	35	57.95	72.87
npScarf_wag	5315235	13	1267710	136	73.55	8.15
npGraph-bwa	4335342	2	4148952	1	16.93	1.45
npGraph-mm2	4335790	1	4335790	0	6.97	0.25
hybridSPAdes	4337369	3	2701005	0	12.80	1.39
Unicycler	4333041	1	4333041	1	6.42	0.53

Continued on next page

Supporting Table 1 – *Continued from previous page*

Method	Assembly size (bp)	#Contigs	N50 (bp)	Mis- assemblies	Mismatch (per 100Kbp)	Indel (per 100Kbp)
<i>E. coli</i> K12 MG1655						
lrscaf	4732001	16	692794	7	3.42	0.58
npScarf	4649902	2	4641702	4	14.94	34.35
npScarf_wag	4687952	3	4641732	0	6.55	1.96
npGraph-bwa	4641743	1	4641743	0	4.50	0.43
npGraph-mm2	4641820	1	4641820	0	3.88	0.26
hybridSPAdes	4644555	1	4641036	0	0.62	0.09
Unicycler	4641650	1	4641650	0	3.43	0.26
<i>E. coli</i> O25b H4-ST131						
lrscaf	5410854	21	910949	9	13.70	0.87
npScarf	5245913	3	5095571	7	7.05	18.81
npScarf_wag	5292700	3	3469617	9	9.03	1.55
npGraph-bwa	5237821	7	4049493	1	3.38	0.31
npGraph-mm2	5249799	3	5110117	0	2.40	0.15
hybridSPAdes	5252762	8	4258948	2	5.43	0.57
Unicycler	5249442	3	5109760	0	4.02	0.27
<i>Klebsiella</i> 30660 NJST258 1						
lrscaf	5512518	20	777919	26	3.52	0.38
npScarf	5559772	7	5259053	4	17.18	13.48
npScarf_wag	5613780	7	5268535	6	1.59	1.92
npGraph-bwa	5534843	8	5263229	0	3.15	0.76
npGraph-mm2	5534878	8	5263264	0	2.75	0.74
hybridSPAdes	5545668	8	5545668	2	4.95	0.09
Unicycler	5537860	9	5263196	0	1.34	0.51
<i>Klebsiella</i> MGH 78578						
lrscaf	5823340	38	559414	7	5.92	1.27
npScarf	5729304	5	5316429	12	14.90	20.27
npScarf_wag	5754443	5	3026286	16	8.17	2.56
npGraph-bwa	5695801	7	5311745	1	12.65	1.25
npGraph-mm2	5696302	6	5315267	0	6.06	0.44
hybridSPAdes	5706470	11	5315273	1	3.82	0.67
Unicycler	5694231	14	5315096	0	5.38	0.21
<i>Klebsiella</i> NTUH-K2044						
lrscaf	5470282	8	2951488	9	3.04	2.62
npScarf	5471696	2	5249198	6	4.82	8.55
npScarf_wag	5530559	3	5249369	2	2.25	1.35
npGraph-bwa	5472629	2	5248476	0	2.52	0.31
npGraph-mm2	5472655	2	5248503	0	1.21	0.26
hybridSPAdes	5473572	2	5248894	0	0.44	0.15
Unicycler	5472697	2	5248545	0	2.41	0.35
<i>Mycobacterium tuberculosis</i> H37Rv						
lrscaf	4260869	12	3215170	8	6.69	2.27
npScarf	4498245	4	4402238	8	5.15	2.68
npScarf_wag	4506056	4	4410942	3	6.81	2.59
npGraph-bwa	4411406	1	4411406	0	1.88	0.43
npGraph-mm2	4411532	1	4411532	0	0.68	0.00
hybridSPAdes	4413942	1	4410519	0	0.75	0.11
Unicycler	4411538	1	4411538	0	2.22	0.34

Continued on next page

Supporting Table 1 – Continued from previous page

Method	Assembly size (bp)	#Contigs	N50 (bp)	Mis-assemblies	Mismatch (per 100Kbp)	Indel (per 100Kbp)
<i>Saccharomyces cerevisiae</i> S288c						
lrscf	12486884	214	254587	27	16.45	1.97
npScarf	11986800	24	796769	51	62.12	21.46
npScarf_wag	12003203	21	917017	21	69.14	5.47
npGraph-bwa	11921736	40	913090	3	38.04	1.94
npGraph-mm2	11920984	38	913198	2	20.66	0.95
hybridSPAdes	12027533	45	770543	5	32.58	1.94
Unicycler	11847655	72	909114	0	21.81	1.04
<i>Shigella dysenteriae</i> Sd197						
lrscf	4678609	175	376897	26	3.91	2.35
npScarf	4586075	173	36560	55	120.14	111.59
npScarf_wag	5462918	92	98791	105	147.48	79.28
npGraph-bwa	4564058	6	4369264	5	80.64	11.16
npGraph-mm2	4558920	6	4364264	7	75.51	10.98
hybridSPAdes	4519131	23	821249	96	9.57	1.42
Unicycler	4560901	3	4369231	0	11.88	1.05
<i>Shigella sonnei</i> 53G						
lrscf	5266645	128	429353	27	9.72	1.19
npScarf	6441461	20	1953896	82	164.02	219.52
npScarf_wag	-	-	-	-	-	-
npGraph-bwa	5211544	4	4988532	0	14.53	0.31
npGraph-mm2	5211527	4	4988519	0	8.56	0.17
hybridSPAdes	5223875	8	2195455	2	41.92	0.06
Unicycler	5220517	5	4988548	0	7.39	0.52
<i>Streptococcus suis</i> BM407						
lrscf	2231488	10	683404	6	11.96	1.37
npScarf	2183951	3	2146594	0	21.51	9.17
npScarf_wag	2289880	3	1493189	1	3.17	1.96
npGraph-bwa	2154623	6	2131479	1	5.25	0.28
npGraph-mm2	2149876	6	2146774	0	1.44	0.28
hybridSPAdes	2172703	2	2146237	0	6.82	0.09
Unicycler	2170829	2	2146250	0	2.67	0.32

Data availability

All data used for benchmarking were from public sources. The synthetic data was downloaded from <https://cloudstor.aarnet.edu.au/plus/index.php/s/dzRCaxLjpGpfKYW>. The real bacterial sequencing data was from NCBI under project accession number PRJNA353060.

Acknowledgments

References

1. Payne A, Holmes N, Clarke T, Munro R, Debebe B, Loose M. Nanopore adaptive sequencing for mixed samples, whole exome capture and targeted panels. bioRxiv

2. Cao MD, Nguyen SH, Ganesamoorthy D, Elliott AG, Cooper MA, Coin LJ. Scaffolding and completing genome assemblies in real-time with nanopore sequencing. *Nature Communications*. 2017;8:14515.
3. Bankevich A, Nurk S, Antipov D, Gurevich AA, Dvorkin M, Kulikov AS, et al. SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing. *Journal of Computational Biology*. 2012;19(5):455–477. doi:10.1089/cmb.2012.0021.
4. Zerbino DR, Birney E. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*. 2008;18(5):821–9.
5. Simpson JT, Wong K, Jackman SD, Schein JE, Jones SJM, Birol I. ABySS: A parallel assembler for short read sequence data. *Genome Research*. 2009;19(6):1117–1123.
6. Cao MD, Ganesamoorthy D, Cooper MA, Coin LJM. Realtime analysis and visualization of MinION sequencing data with npReader. *Bioinformatics*. 2016;32(5):764–766. doi:10.1093/bioinformatics/btv658.
7. Nguyen SH, Duarte TP, Coin LJ, Cao MD. Real-time demultiplexing Nanopore barcoded sequencing data with npBarcode. *Bioinformatics*. 2017;33(24):3988–3990.
8. Ester M, Kriegel HP, Sander J, Xu X. A density-based algorithm for discovering clusters in large spatial databases with noise. *AAAI Press*; 1996. p. 226–231.
9. Kullback S, Leibler RA. On information and sufficiency. *The annals of mathematical statistics*. 1951;22(1):79–86.
10. Kang DD, Froula J, Egan R, Wang Z. MetaBAT, an efficient tool for accurately reconstructing single genomes from complex microbial communities. *PeerJ*. 2015;3:e1165.
11. Wu YW, Tang YH, Tringe SG, Simmons BA, Singer SW. MaxBin: an automated binning method to recover individual genomes from metagenomes using an expectation-maximization algorithm. *Microbiome*. 2014;2(1):26.
12. Dijkstra EW. A note on two problems in connexion with graphs. *Numerische mathematik*. 1959;1(1):269–271.
13. Antipov D, Korobeynikov A, McLean JS, Pevzner PA. hybridSPAdes: an algorithm for hybrid assembly of short and long reads. *Bioinformatics*. 2016;32(7):1009–1015. doi:10.1093/bioinformatics/btv688.
14. Wick RR, Judd LM, Gorrie CL, Holt KE. Unicycler: resolving bacterial genome assemblies from short and long sequencing reads. *PLOS Computational Biology*. 2017;13(6):e1005595.
15. Mikheenko A, Prjibelski A, Saveliev V, Antipov D, Gurevich A. Versatile genome assembly evaluation with QUAST-LG. *Bioinformatics*. 2018;34(13):i142–i150.
16. Li H. Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. *arXiv*. 2013; p. 3.

17. Li H. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*. 2016;32(14):2103–2110. doi:10.1093/bioinformatics/btw152.
18. Brudno M, Malde S, Poliakov A, Do CB, Couronne O, Dubchak I, et al. Glocal alignment: finding rearrangements during alignment. *Bioinformatics*. 2003;19(suppl_1):i54–i62.
19. George S, Pankhurst L, Hubbard A, Votintseva A, Stoesser N, Sheppard AE, et al. Resolving plasmid structures in Enterobacteriaceae using the MinION nanopore sequencer: assessment of MinION and MinION/Illumina hybrid data assembly approaches. *Microbial genomics*. 2017;3(8).
20. Carattoli A, Zankari E, García-Fernandez A, Larsen MV, Lund O, Villa L, et al. PlasmidFinder and pMLST: in silico detection and typing of plasmids. *Antimicrobial agents and chemotherapy*. 2014; p. AAC-02412.
21. Potter RF, D’souza AW, Dantas G. The rapid spread of carbapenem-resistant *Enterobacteriaceae*. *Drug Resistance Updates*. 2016;29:30–46.
22. Rozov R, Goldshlager G, Halperin E, Shamir R. Faucet: streaming de novo assembly graph construction. *Bioinformatics*. 2017;34(1):147–154.
23. Giordano F, Aigrain L, Quail MA, Coupland P, Bonfield JK, Davies RM, et al. De novo yeast genome assemblies from MinION, PacBio and MiSeq platforms. *Scientific reports*. 2017;7(1):3935.
24. Prjibelski AD, Vasilinets I, Bankevich A, Gurevich A, Krivosheeva T, Nurk S, et al. ExSPANDer: a universal repeat resolver for DNA fragment assembly. *Bioinformatics*. 2014;30(12):i293–i301.
25. Qin M, Wu S, Li A, Zhao F, Feng H, Ding L, Ruan J. LRScaf: improving draft genomes using long noisy reads. *BMC Genomics*. 2019;20(1):955