Real-time resolving assembly graph by ONT long reads data

Son Hoang Nguyen[1,*], **Minh Duc Cao**[2], **and Lachlan Coin**[1,*]

[1]Institute for Molecular Bioscience, the University of Queensland, St Lucia, Brisbane, QLD 4072 Australia; [2]4catalyzer

[*] To whom correspondence should be addressed. E-mails: l.coin@imb.uq.edu.au,s.hoangnguyen@imb.uq.edu.au

This manuscript was compiled on **November 29, 2019 20:56**

Abstract: A real-time assembly pipeline for Oxford Nanopore Technology (ONT) data is important for either saving sequencing resources or reducing turnaround time for data analyses. The previous approach from `npScarf` provided a fast-response streaming algorithm for such task but was relatively prone to mis-assemblies compared to other graph-based methods. Here we present `npGraph`, a real-time hybrid assembly software 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. The application on synthetic and real data set of isolate genomes show improved accuracy while still maintaining the required computational cost extremely low. With `npScarf`, the built-in graphical user interface (GUI) can provide users a comprehensive look-and-feel of the whole assembly process. The tool and source code is available at https://github.com/hsnguyen/assembly.

Keywords: hybrid assembly, assembly graph, real-time analysis, nanopore sequencing

Introduction

There is rich connectivity information in the assembly graph resulted from short read assembly. However, existing tools do not make use of this. Here, we present npGraph, a novel algorithm to resolve the assembly graph in real-time long read sequyencing. npGraph maintain the assembly grahp and as new long reads coming in, it untangle the knots thereby simplify the assembly graph. Because of this, npGraph has better estimation of multiplcity of repeat contigs and hence resulting in fewer misassembly (Not to mention in real-time). In addition, we develop a visualisation tool for practitioners to monitor the progress of assembling process.

.

Results

Resolving assembly graph in real-time

`npGraph` makes use of a assembly graph resulted from assembling short reads using a de Bruijin graph method such as `SPAdes` [1], `Velvet` [2] and *mathttAbySS* [3]. The assembly graph consists of a list of contigs, and possible connections among these contigs. In building the assembly graph, the de Bruijin graph assembler attemtps to extend each contig as long as possible, until there are more than one possible ways of extending due to the repeatitive sequences beyond the informaion contained in short reads. Hence cach contig has multiple possible connections with others, creating knots in the assembly graph. `npGraph` then uses the connectivity information from long reads to untangle these knots in real-time. With sufficient data, when all the knots are removed, the assemply graph is simplified to a path which represents the complete assembly.

`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 stratery 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.

We also develop a Graphical User Interface (GUI) for npgraph. The GUI includes the dashboard for controling the settings of the program and a window for visualization of the assembly graph in real-time (Figure 1). 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 [4–6] that allow the analysis to take place abreast to a nanopore sequencing run.

Evaluation using synthetic data

To evaluate the performance of the method, `npGraph` was tested along with `SPAdes`, its hybrid assembly module [7], `npScarf`, and Unicycler version 0.4.6 using Unicycler's synthetic data set [8] . The data set is a simulation of Illumina and MinION raw data, generated *in silico* based on random and available microbial references. We ran all hybrid assembly methods in batch-mode and the reciprocal results were examined by QUAST 5.0.2 [9].

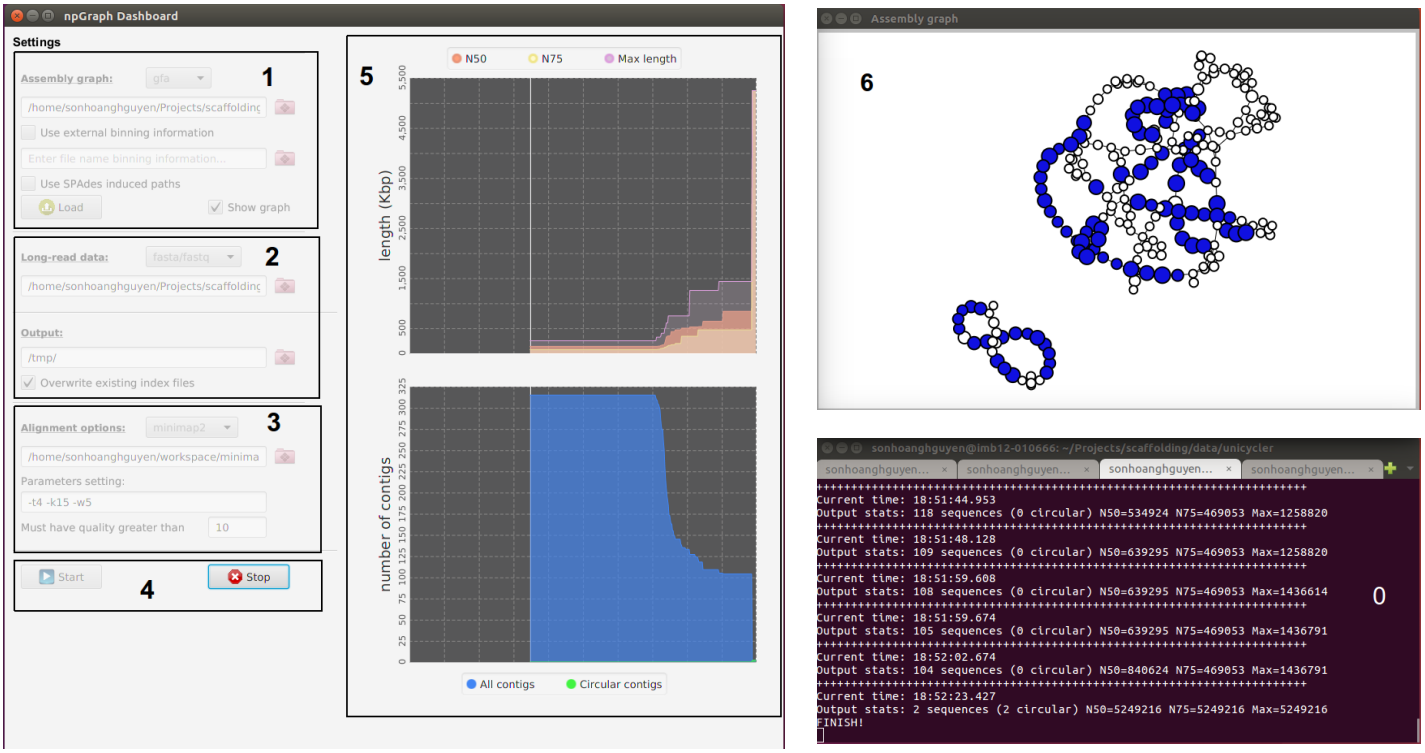


Figure 1: `npGraph` user interface including Console (**0**) and GUI components (**1**-**6**). The GUI consists of the Dashboard (**1**-**5**) and the Graph View (**6**). From the Dashboard there are 5 components as follow: **1** the assembly graph input field; **2** the long reads input field; **3**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

Table 1: Comparison of assemblies produced in batch-mode using `npGraph` and other hybrid assembly methods on representative `Unicycler`'s synthetic data downloaded from https://cloudstor.aarnet.edu.au/plus/index.php/s/dzRCaxLjpGpfKYW

Method	Assembly size (Mbp)	#Contigs	N50 (Kbp)	Mis-assemblies	Error (per 100 Kbp)	Run times (CPU hrs)
M. tuberculosis 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
K. pneumoniae 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
S. cerevisiae 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
S. sonnei 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
S. dysenteriae Sd197	4,560,911 bp					

the fastest hybrid assembler available. This feature is certainly more favoured for a real-time assembly pipeline that normally requires quick responses in very short intervals. Because `minimap2` is supposed to replace `BWA-MEM` for long-read sequencing data alignment, it likewise being chosen as the default aligner in the `npGraph` pipeline.

Overall, as expected from hybrid methods, the error rate of the draft assembly hardly exceeded 100 bp per 100 Kbp (equivalent to 0.1%) in almost every cases. The only exception is `npScarf` as it would sacrify the accuracy for the continuity by using the nanopore bases to fill in the gap between 2 unlinked unique contigs. This happened for 2 *Shigella* simulated genomes containing unusually long stretches of repetitive elements ($>$ 10Kbp) for bacteria strains. Especially for the last case for *Shigella dysenteriae*, `npScarf` failed to finish its exhaustive path finding within reasonable time frame thus being excluded from the report. In fact, the indel errors (typical in nanopore sequencing) were found relatively low in the final contigs (Table 1). The majority of the differences accounted for the mismatched nucleotides caused by the alternative paths connecting the unique anchors from the backbone of the assembly. This phenomenon may root from homologous repeats or sequencing errors of the genome.

Amongst all assemblers, `Unicycler` applies an algorithm based on semi-global (or glocal) alignments [12] with the consensus long reads using `SeqAn` library. When testing with these data sets, it was on top in terms of computational cost but returned the most reliable and complete assembly. `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*.

Regarding streaming algorithms, `npScarf`, as aforementioned, used the most greedy scaffolding approach that might introduce extra mis-assemblies and errors. They might be negligible for the majority of common bacterial genomes with modest repeat profiles, but in extreme cases ,*e.g. Shigella* data sets, the obtained results would be more error-proned necessitating extra post-processing, polishing steps afterward. On the other hand, `npGraph` significantly reduced the errors compared to `npScarf`, sometimes even be the best option *e.g.* for *M. tuberculosis* and *K. pneumoniae*. For the yeast *S. cerevisiae* data set, its assembly best covered the reference genome but the number of mis-assemblies was up to 5. The unfavourable figures, namely mis-assemblies and error, were still high in case of *S. dysenteriae*. The reason was due to the complicated and extremely fragmented graph components containing a large number of small-scaled contigs that were difficult to be mapped with nanopore data. The progressive path finding module tried to induce the most likely solution from a stream of coarse-grained alignments and blundered in this case.

Hybrid assembly for real data sets

The complicated graph topology from *S. dysenteriae* simulation is not common in real-life application. Also, considering the overfitting of `Unicycler` with its own test sets, we collected real-life sample for further comparison. A number of sequencing data sets from *in vitro* bacterial samples [13] were used in this scenario. 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 being reported by QUAST for the comparison of the results. Instead, we investigated the number of circular sequences and `PlasmidFinder` 1.3 [14] mappings to obtain an evaluation on the accuracy and completeness of the assemblies. Table 2 shows the benchmark results of `npGraph` (using `minimap2`) against `Unicycler` on three data sets of bacterial species *Citrobacter freundii*, *Enterobacter cloacae* and *Klebsiella oxytoca*.

From the first data set, there was high similarity between final contigs generated by two assemblers. They shared the same number of circular ultimate sequences, including the chromosomal and other six replicons contigs. The only divergence lied on the biggest sequence ($\simeq$ 5.029Mbp) when the `Unicycler`'s chromosome was 48 nucleotides longer than that of `npGraph`. Five out of six identical replicons were confirmed as plasmids based on the occurences of appropriate Origin of replication sequences (PlasmidFinder database). In detail, two megaplasmids (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 newly replicon's DNA. Lastly, there were still 14.5Kbp unfinished sequences resulted in 3 linear contigs from `Unicycler` and 2 for `npGraph` respectively.

The assembly task for *Enterobacter cloacae* was observed more challenging as the chromosomal DNA sequence not been fully resolved using either method. The chromosome size was estimated to be approximately 4.8Mbp but had been broken into two smaller pieces. `npGraph` returned longer stretches of length 3.324Mbp and 1.534Mbp while the figures were 2.829Mbp and 1.978Mbp from `Unicycler`'s output. However, the number of circular sequences detected by `Unicycler` was one more than the other (2 versus 1). They were corresponding to 2 plasmids, namely IncR and repA. While the latter were recognized by both methods, the longer plasmid sequence was fragmented running `npGraph`. Similar to the previous data set, there were around 14Kbp of data were unable to be finished by the assemblers.

Finally, assembly for *Klebsiella oxytoca* saw fragmented chromosome using `Unicycler` but it was a fully complete contig for `npGraph` with 6.156Mbp of size. The two assemblers shared 3 common circular sequences where two of them were confirmed plasmids. The first identical sequences represented a megaplasmid ($\simeq$ 113Kbp) with two variations of IncFII's origin of replication DNA being identified. The other agreed plasmid were IncL/M with 76Kbp of length. Particularly, there was one circular contig with length greater than 100Kbp but returned no hits to the plasmid database, suggesting the importance of *de novo* replicon assembly in combination with further interrogations. `Unicycler` detected another megaplasmid of size 108.4Kbp which was fractured by `npGraph`. The dissolution was also observed in `npGraph`

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)
Citrobacter freundii CAV1374	**5,029,534**	**5,029,486**	Chromosome
	109688	**109688**	IncFIB(pHCM2)_1_pHCM2_AL513384
	100,873	**100,873**	IncFIB(pB171)_1_pB171_AB024946
	85,575	**85,575**	IncL/M(pMU407)_1_pMU407_U27345
	43,621	**43,621**	repA_1_pKPC-2_CP013325
	3,223	**3,223**	-
	1,916	**1,916**	ColRNAI_1__DQ298019
	14,464|3	14,456|2	-
Enterobacter cloacae CAV1411	4,806,666|2	4,858,438|2	Chromosome
	90,451	90,693|2	IncR_1__DQ449578
	33,610	**33,610**	repA_1_pKPC-2_CP013325
	13,129|2	14,542|4	-
Klebsiella oxytoca CAV1015	6,153,947|5	**6,155,762**	Chromosome
	113,105	**113,105**	IncFII(SARC14)_1_SARC14_JQ418540; IncFII(S)_1__CP000858
	111,395	**111,395**	-
	108,418	109,209|13	IncFIB(K)_1_Kpn3_JN233704
	76,183	**76,186**	IncL/M(pMU407)_1_pMU407_U27345
	11,638	11,892|2	-

for the final contig of length 11.6Kbp where it failed to combine two smaller sequences into one.

In addition to what presented in Table 2, dot plots for the pair-wise alignments between the assembly contigs were generated and can be found in Appendix Figure 1. Interestingly, beside all other agreements, there was a structural difference using two methods for *E. cloacae* CAV1411 genome assembly. This was caused by the inconsistency of a fragment's direction on the final output contigs. However, when compare to a reference genome of the same bacteria strain (GenBank ID: CP011581.1 [15]), contigs generated by `npGraph` demonstrated a consistent alignment which was not the case for `Unicycler` results (Appendix Figure 2). Even though this might reflect a novel variation between bacterial samples of the same strain, it was more likely a mis-assembly by using `Unicycler`.

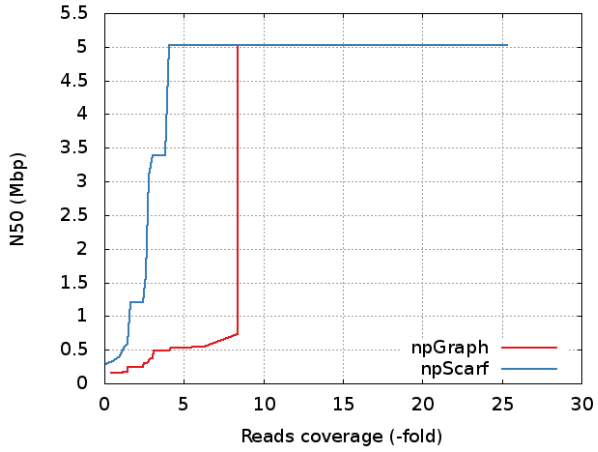
Overall, by testing with synthetic and real data, `npGraph` proved to be able to generate assemblies of comparative quality compared to other powerful batch-mode hybrid assemblers, such as `hybridSPAdes` or `Unicycler`. Furthermore, similar to `npScarf`, it has the advantage in term of supporting real-time assembly. The next section will address this utility and the interactive GUI bundled in `npGraph`.

Assembly performance on streaming data

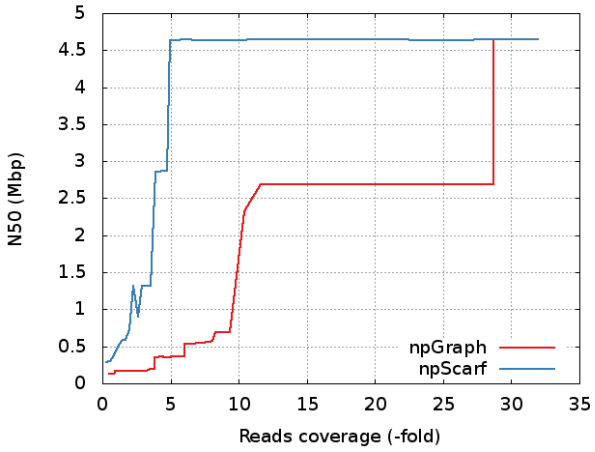
Figure 2 demonstrates the real-time mode performance of `npScarf` and `npGraph` via N50 statistics during the assembly of 4 example data sets. This experiment would discover the rate of completing genome assemblies of the new method, set aside the accuracy aspect which had already been discussed previously.

As can be observed from all the plots, `npGraph` and `npScarf` both converged to the same ultimate completeness but with different paces and patterns. Apparently it took more data for `npGraph` to finish the same genome than the other. The reason stems from the fact that the new algorithm used a more 'conservative' approach of bridge construction with at least 3 supporting long-reads for each to prevent any potential mis-assembly. Unlike `npScarf` when 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 but seeing no conflict until then.

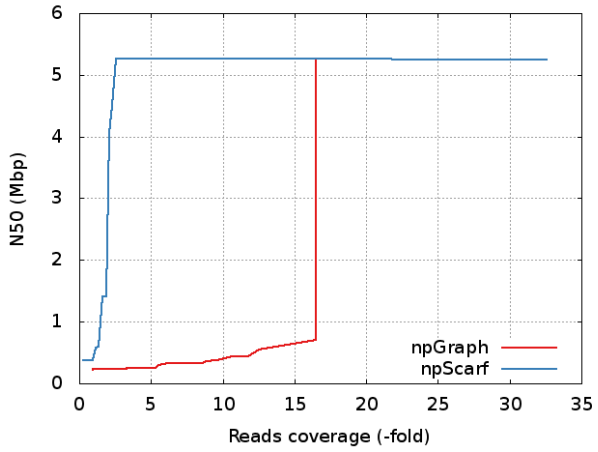
Once a unique path has been determined, the bridge can be formed to connect the fragments together into a longer sequence leading to the increase of N50. `npGraph` shows stable results so that it can directly report final assembly at any timepoint. As can be seen from Figure 2, the genomes were completed at slower speed at the beginning of the streaming process and faster toward the end due to the accumulation of bridging information. However, the plots here only depict the real-time assembly operation including one final post-processing step. If the process is stopped somewhere in the middle, the post-processing would be invoked to return more completed genome but with the risk of higher mis-assembly due to shortage of supporting data.



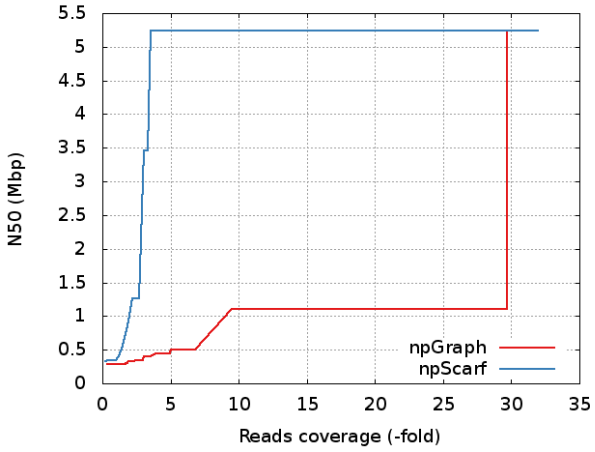
(a) *Citrobacter freundii* CAV1374



(b) *Escherichia coli* K12 MG1655



(c) *Klebsiella* 30660 NJST258



(d) *Klebsiella* NTUH K2044

Figure 2: N50 statistics of real-time assembly by `npScarf` and `npGraph`.

Discussion

Streaming assembly methods had been proven to be useful in saving time and resources compared to the conventional batch algorithms with examples included *e.g.* `Faucet` [16] and `npScarf` [5]. 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 nanopore sequencing which is well-known by the real-time property. 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 [5].

Hybrd approaches are still common practice in genome assembly and data analyses when Illumina sequencing remains the most favoured option in terms of cost and accuracy to date. 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 the consequence, it is rational to combine two sources of data together in a hybrid method that can offer accurate and comptete 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 contigs-based streaming algorithm, `npScarf`'s results might suffer from mis-assemblies [8, 17]. A default setting were optimzed 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 low quality nanopore reads thus the accuracy of the final assembly is affected as well. To tackle the quality issue while maintaining its streaming feature, a bridging method by assembly graph traversing is proposed. In which, after the construction of a compact DBG assembly graph, the next step is to traverse the graph, resolve the repeats and identify 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` [7] or `Unicycler` [8]. In general, the available tools employ batch-mode algorithms on the whole long-read data set to generate the final

genome assembly. In which, the `SPAdes` hybrid assembly module, from its first step, exhaustively looks for the most likely paths (with mininum edit distance) on the graph for each of the long read given but only ones supported by at least two reads are attained. In the next step, these paths will be subjected to a decision-rule algorithm, namely `exSPAnder` [18], 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 the latter 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. For that reason, the first direction is more suitable for a real-time pipeline.

Nonetheless, the challenge to adapt this approach into a real-time mechanism is obvious, mainly from building a progressive implementation for path-finding and graph reducing module which are essential for a streaming graph assembler. To achieve that, we apply a modified DFS (depth-first search) mechanism and a dynamic voting algorithm into an on-the-fly graph resolver. The method is implemented in `npGraph`, a user-friendly tool with GUI that can traverse the assembly graph and bridge its components in real-time as long as the nanopore sequencing process is still running.

We use different hybrid data sets to study the `npGraph`'s performance and compare with other available tools with similar functionality, such as `npScarf`, `Unicycler` and `hybridSPAdes`. Even though all data were from microbial genomes, the complexity of the input graphs range from few hundred to ten thousands of components, synthesized *in silico* or sequenced from *in vitro* samples.

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, however, the sequencing cost per base is still high. 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 a less greedy approach based on graph traversal. This might reduce the bridging rate when the linking conditions become more strict, but the concurrent results reported 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.

As a hybrid assembler, 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-enought 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.

Methods

The work flow of `npGraph` mainly consists of 3 stages: (1) assembly graph preprocessing; (2) graph resolving and simplifying; (3) postprocessing and reporting results. The first step is to load the graph of Illumina contigs and retrieve their metadata which are helpful for the next steps, *e.g.*binning and multiplicity estimation. The second step works on the processed input and augmented information from the previous one. In combination with path inducing from long reads, the assembly graph is then traversed and resolved in real-time. 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 interchangable throughout this context.

Contigs binning

Contigs from each completed genome are expected to be assigned to an unique group, or *bin*, that would represent the final assembly unit, *e.g.* chromosome , plasmids, or even particular species genome in a metagenomic community. As a result, the binning phase would assist the multiplicity estimation submodule that can differentiate repetitive contigs from unique ones.

Each contig is embeded in a single node in the assembly graph and an edge between two nodes indicates their overlap (link) properties. This step is to cluster the biggest nodes (contigs longer than 10Kbp) into different sets, namely *core* groups, based on the their degree (number of connections) and coverage values. DBSCAN clustering algorithm [19] is applied for this task by default. The rationale is to approximate a coverage value of an extremely long contig (which

can be splitted into more than 10,000 *k-mers*) to be a sampled mean of a Poisson distribution (of *k-mers* count). The metric is a distance function based on Kullback-Leibner divergence [20], or relative entropy, of two Poisson distributions.

Assuming there are 2 Poisson distributions P_1 and P_2 with density functions

$$p_1(x, \lambda_1) = \frac{e^{-\lambda_1}\lambda_1^x}{\Gamma(x+1)}$$

and

$$p_2(x, \lambda_2) = \frac{e^{-\lambda_2}\lambda_2^x}{\Gamma(x+1)}$$

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

$$D_{KL}(P_1||P_2) = \int_{-\infty}^{\infty} p_1(x) \log \frac{p_1(x)}{p_2(x)} dx$$

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 with regard to P_1. The log ratio of the density functions is

$$\log \frac{p_1(x)}{p_2(x)} = x \log \frac{\lambda_1}{\lambda_2} + \lambda_2 - \lambda_1$$

take expectation of this expression with regard to P_1 with mean λ_1 we have

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

The metric we used is the distance defined as

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

We implemented a simple binning algorithm based on the clustered contigs, the graph topology as well as statistics calculated by length and coverage of each contigs. Additional binning step is taken place in real-time using the long reads as well. In this context, the repetition or uniquess of ambigous nodes are determined with sufficient supporting evidence from the long-spanning reads. Generally, the algorithm performs well for isolate or simple simulated metagenomics, but ineffective for complicated community. In that case, external binning tools can be employed for more comprehensive binning task and the results can be augmented to `npGraph` input. If external binning algorithm is employed, the resulting output must be converted into a text file that specifies the corresponding bin of every contigs, just like output from MetaBAT. By that, each line of the file would be: < `contig_ID` > < `bin_ID` > where `bin_ID` = 0 indicates unspecified binned contigs.

Multiplicity estimation

Due to the possible divergence of sequencing coverage relative to the real abundance of sequences, especially for the shorter contigs, an optimization step is carried out to alleviate this issue. The re-estimation is basically carried out by following two steps.

1. From nodes coverage, estimate edges' value 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 coverage of a node i in the graph;
   $\sum e_i^+$ and $\sum e_i^-$ indicates sum of the values of incoming and outgoing edges from i respectively. The above function and be rewritten as:

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

   and then being minimized by using gradient-based method.

2. Update nodes' coverage based on itself and its neighbor edges' measures.

The calibration is iterative until no further improvements are made or a threshold loop count is reached. The coverage measures of nodes (which represent contigs) are propated throughout the graph via connecting edges for calibrations prior to the multiplicity estimation.

Based on the coverage values of all the edges and the graph's topology, we induce the copy numbers of every significant nodes (long contigs) in the final paths. For each node, this could be done by investigating its adjacent edges and answering the questions of how many times it should be visited, from which abundance groups. Multiplicities of insignificant nodes (of sequences with length less than 1,000 bp) can be estimated in the same way but usually with less confident due to more complicated connections and greater variation of coverage values. For that reason, in `npGraph`'s algorithm, they are only used as augmented information to calculate likelihood scores of candidate paths containing corresponding nodes.

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 inbetween 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 inbetween, 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.

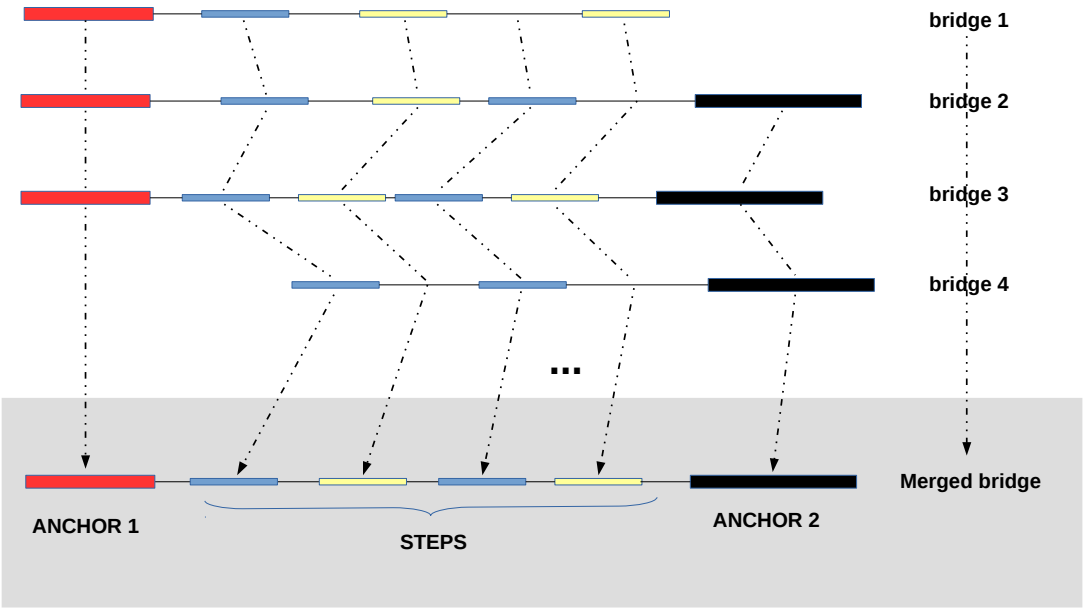


Figure 3: Bridge merging progressively in real-time. Sequencing long reads induce alignments to the contigs where new bridges are created respectively. Bridges sharing same anchors can be merged together to form the ultimate, more comprehensive bridge (with more step nodes and better approximation of spans between them).

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 3, 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, eventhough 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 [21] to reduce the search space when using Depth-First Search.

Due to false alignments from shorter contigs to the long reads, not all of the reported step nodes are neccessary to be appeared 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.

Before that, we define Algorithm 1 to demonstrates the path finding algorithm for two nodes given their estimated distance. In which, function `shortestTree`$(\overrightarrow{vertex}, distance) : (V, Z) \rightarrow V^n$ 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 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.

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 $\overrightarrow{v_1}, \overrightarrow{v_2}$ and estimated distance d between them
Output: Set of candidate paths connecting $\overrightarrow{v_1}$ to $\overrightarrow{v_2}$ with reasonable distances compared to d

```
1  Function DFS(v1→, v2→, d):
2      P:=new List()
3      M:=shortestTree(v2→, d)                          // build shortest tree from v2→ with range d
4      if M.contain(v1→) then
5          S:=new Stack()                                // stack of sets of edges to traverse
6          edgesSet:=getEdges(v1→)                       // get all bidirected edges going from v1→
7          S.push(edgesSet)
8          p:=new Path(v1→)                              // init a path that has v1→ as root
9          while true do
10             edgesSet:=S.peek()
11             if edgesSet.isEmpty() then
12                 if p.size() ≤ 1 then
13                     break                             // stop the loop when there is no more edge to discover
14                 S.pop()
15                 d+=p.peekNode.length() + p.popEdge().length()
16             else
17                 curEdge := edgesSet.remove()
18                 v→:=curEdge.getOpposite(p.peekNode())
19                 S.push(getEdges(v→).includedIn(M))
20                 p.add(curEdge)
21                 if reach v2→ with reasonable d then
22                     P.add(p)
23                 d-=v→.length() + curEdge.length()
24     return P
```

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

Data: Assembly graph $G\{V, E\}$
Input: Brigde $B : \{\overrightarrow{v_0}, ...\overrightarrow{v_k}, ..., \overrightarrow{v_n}\}$ with $\overrightarrow{v_0}$ and $\overrightarrow{v_n}$ are two anchors, $\{\overrightarrow{v_k}\}, k = 1 \ldots (n-1)$ are steps inbetween
Output: Set of candidate paths connecting $\overrightarrow{v_0}$ to $\overrightarrow{v_2}$ 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:=argmax_k(v_k→.score())
       /* if there is no step inbetween, run path finding algorithm above directly and return the result   */
3      if M.size() ≡ 2 then
4          return DFS(B.start(), B.end().B.distance())
       /* divide the original bridge B into 2 bridges by v_m:  BL and BR   */
5      BL:={B.start(), ..., v_m→}
6      BR:={v_m→, ..., B.end()}
       /* Return the join of running recursive function on two sub-bridges   */
7      return BinaryBridging(BL) ⋈ BinaryBridging(BR)
```

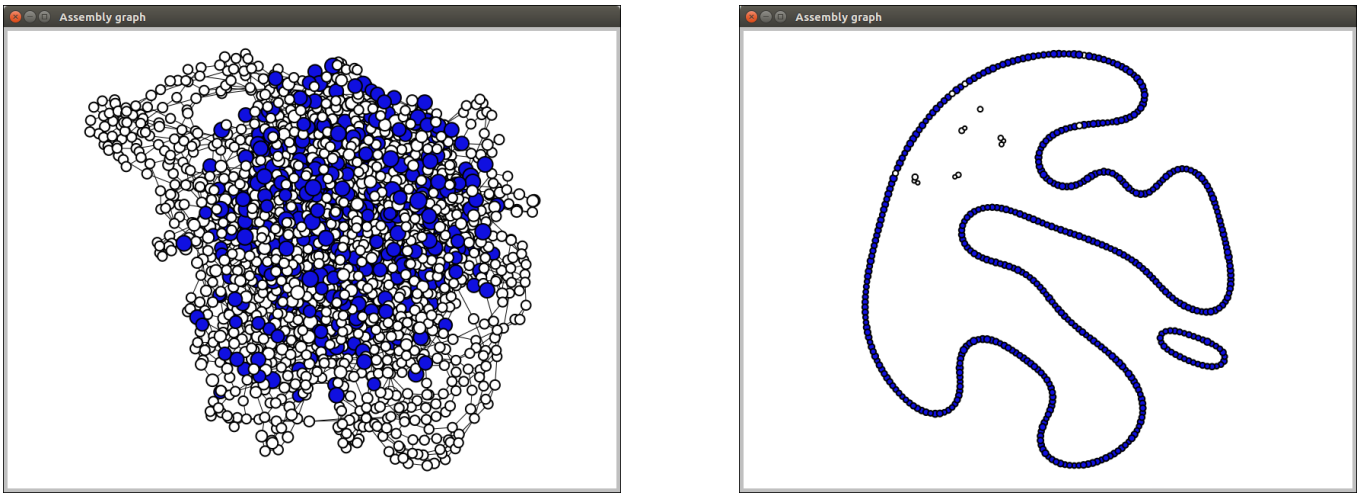
Basically, the algorithm keeps track of a stack that contains sets of candidate edges to discover. During the traversal, a variable d is updated as an estimation for the distance to the target. A hit is reported if the target node is reached with a reasonable distance *i.e.* close to zero, within a given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



(a) Initial graph

(b) Resolved graph

Figure 4: 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.

Figure 4 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`.

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 $p = \{v_0, e_1, v_1, \ldots, v_{k-1}, e_k, v_k\}$ of size k is considered as straight if and only if each of every edges along the path $e_i, \forall i = 1, \ldots, k$ 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 5. These edges are defined as branching edges which stop straight paths from further extending.

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.

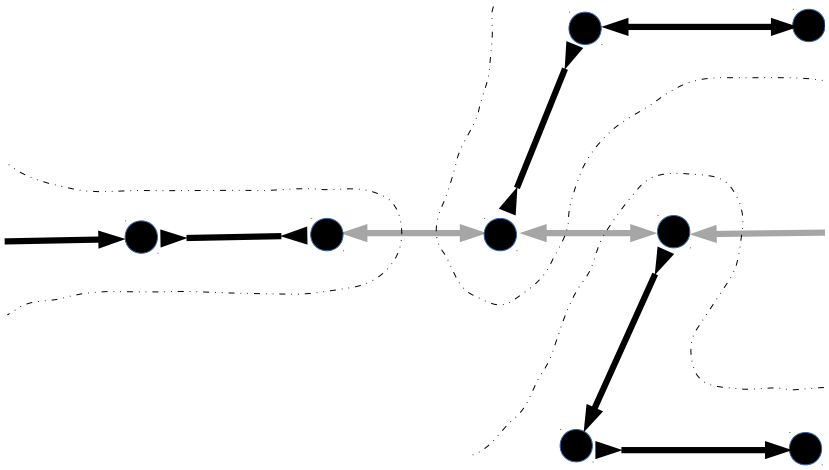


Figure 5: Example of graph decomposition into longest straight paths. Branching edges are masked out (shaded) leaving only straight paths (bold colored) to report. There would be 3 contigs extracted by traversing along the straight paths here.

References

[1] Bankevich A et al. (2012) SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing. *Journal of Computational Biology* 19(5):455–477.

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

[3] Simpson JT et al. (2009) ABySS: A parallel assembler for short read sequence data. *Genome Research* 19(6):1117–1123.

[4] Cao MD, Ganesamoorthy D, Cooper MA, Coin LJM (2016) Realtime analysis and visualization of MinION sequencing data with npReader. *Bioinformatics* 32(5):764–766.

[5] Cao MD et al. (2017) Scaffolding and completing genome assemblies in real-time with nanopore sequencing. *Nature Communications* 8:14515.

[6] Nguyen SH, Duarte TP, Coin LJ, Cao MD (2017) Real-time demultiplexing Nanopore barcoded sequencing data with npBarcode. *Bioinformatics* 33(24):3988–3990.

[7] Antipov D, Korobeynikov A, McLean JS, Pevzner PA (2016) hybridSPAdes: an algorithm for hybrid assembly of short and long reads. *Bioinformatics* 32(7):1009–1015.

[8] Wick RR, Judd LM, Gorrie CL, Holt KE (2017) Unicycler: resolving bacterial genome assemblies from short and long sequencing reads. *PLOS Computational Biology* 13(6):e1005595.

[9] Mikheenko A, Prjibelski A, Saveliev V, Antipov D, Gurevich A (2018) Versatile genome assembly evaluation with QUAST-LG. *Bioinformatics* 34(13):i142–i150.

[10] Li H (2013) Aligning sequence reads, clone sequences and assembly contigs with BWA-MEM. p. 3.

[11] Li H (2016) Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics* 32(14):2103–2110.

[12] Brudno M et al. (2003) Glocal alignment: finding rearrangements during alignment. *Bioinformatics* 19(suppl_1):i54–i62.

[13] George S et al. (2017) Resolving plasmid structures in Enterobacteriaceae using the MinION nanopore sequencer: assessment of MinION and MinION/Illumina hybrid data assembly approaches. *Microbial genomics* 3(8).

[14] Carattoli A et al. (2014) Plasmidfinder and pmlst: in silico detection and typing of plasmids. *Antimicrobial agents and chemotherapy* pp. AAC–02412.

[15] Potter RF, D'souza AW, Dantas G (2016) The rapid spread of carbapenem-resistant Enterobacteriaceae. *Drug Resistance Updates* 29:30–46.

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

[17] Giordano F et al. (2017) De novo yeast genome assemblies from MinION, PacBio and MiSeq platforms. *Scientific reports* 7(1):3935.

[18] Prjibelski AD et al. (2014) ExSPAnder: a universal repeat resolver for DNA fragment assembly. *Bioinformatics* 30(12):i293–i301.

[19] Ester M, Kriegel HP, Sander J, Xu X (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. (AAAI Press), pp. 226–231.

[20] Kullback S, Leibler RA (1951) On information and sufficiency. *The annals of mathematical statistics* 22(1):79–86.

[21] Dijkstra EW (1959) A note on two problems in connexion with graphs. *Numerische mathematik* 1(1):269–271.