Online 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 **June 14, 2019 19:18**

Abstract: A real-time assembly pipeline for Oxford Nanopore Technology (ONT) data is important for either better control of sequencing resources or quick turnaround of analyses. The previous approach from `npScarf` provided a greedy fast-response, auto-correct streaming algorithm for such task but was relatively prone to mis-assemblies. Here we present another real-time hybrid assembler, `npGraph`, that work on the assembly graph instead of the pre-assembled contigs. The results show improved accuracy while still minimize the required computational cost. The tool is available at https://github.com/hsnguyen/assembly.

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

Introduction

Streaming assembly methods have been proven to be useful in saving time and resources compared to the traditional batch algorithms with examples included *e.g.* `Faucet` [1] 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 DBG construction while waiting for the data being retrieved.

`npScarf`, on the other hand, works on the available short-read assembly to scaffold the contigs using nanopore sequencing which is well-known by the real-time property. The completion of genome assembly along with the sequencing run provides explicit benefits in term of resource control and turn-around time for analysis [2]. However, due to the greedy approach of a streaming algorithm, as well as being an alignment-based-only scaffolding mechanism, running the tool with default settings suffers from mis-assemblies [3, 4]. In many case, 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 execution, an assembly graph processing system is investigated as it would provide an additional high-quality source of linking information for the assembly operations.

After the construction of an 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` [5] or `Unicycler` [3]. 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` [6], 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 later approach 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.

However, the challenges to adapt this approach into a real-time mechanism are obvious and mainly come from the heavy path-finding task and the complication of self-improvement step which is critical to a streaming algorithm. A modified DFS algorithm and a voting system with accumulating scores calculation has been implemented to overcome these issues. This results in `npGraph`, an user-friendly tool with GUI that can traverse the assembly graph and scaffold its components in real-time as long as the nanopore sequencing process is running and continuously generating long reads.

Results

`npGraph` – a tool to resolve assembly graph in real-time

`npGraph`'s input consists of Illumina assembly graph resulted from running assembler, e.g. `SPAdes` [7], `Velvet` [8], `AbySS` [9] on Illumina short reads, together with long reads from third generation sequencing technology (Oxford Nanopore Technology, Pacbio). The long reads will be aligned with the contigs in the assembly graph to indicate longer paths that should be traversed. These local paths, given sufficient data, are expected to untangle the complicated graph and guide to the global Eulerian paths (or cycles if possible) that represent the entire genomic sequences. `npGraph` can

be invoked and fully function from the command-line interface. In addition, in order to aid the visualization of the assembly process, a GUI has been developed as well.

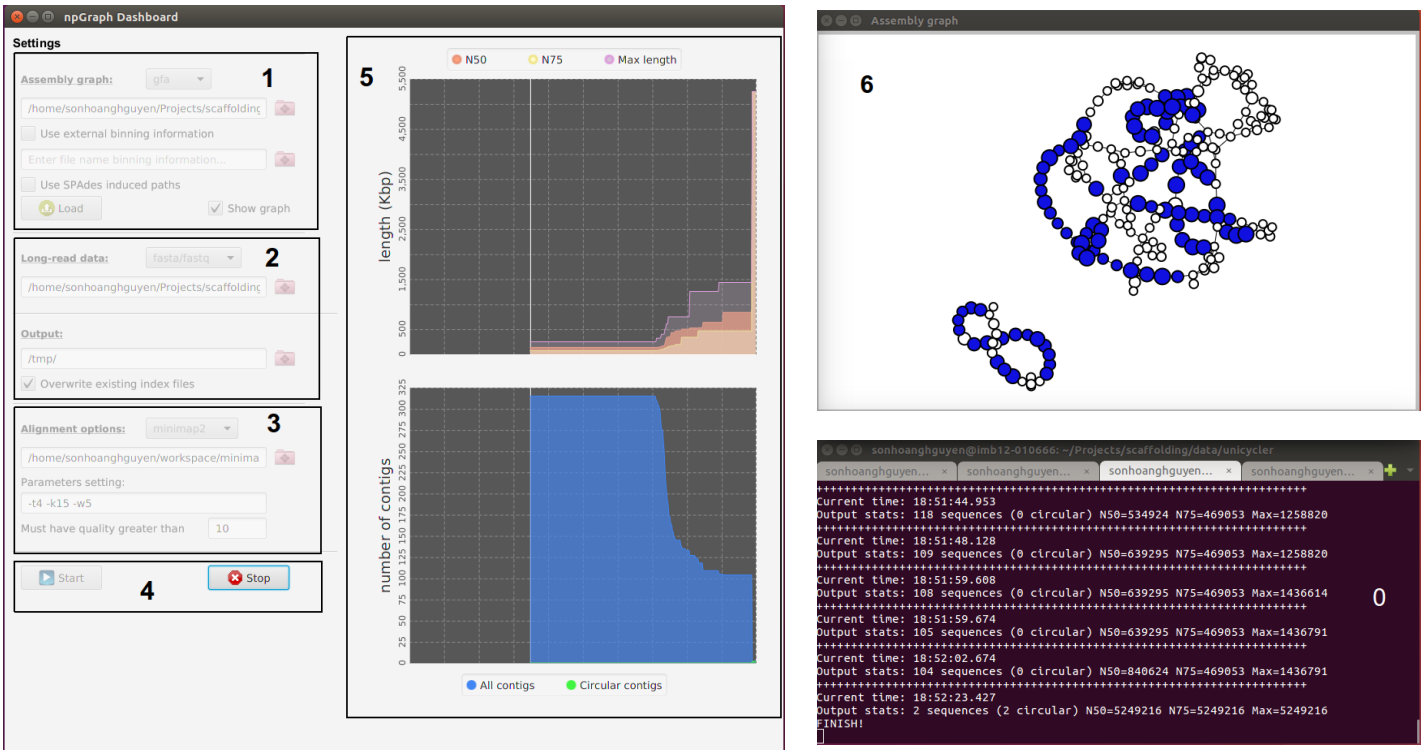


Figure 1: `npGraph` user interface including Console (**0**) and GUI components (**1**-**6**). The GUI consists of the Dashboard (**1**-**5**) and the Graph View (**6**). From the Dashboard there are 5 components as follow: **1** the assembly graph input field; **2** the long reads input field; **3**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

The GUI includes the dashboard for control the settings of the program and another pop-up window for a simple 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. The box numbered **1** on Figure 1 is designed for this task. Only after an assembly graph is loaded successfully, users can move to box **2** to specify the nanopore input data. Settings for an aligner (`BWA-MEM` or `minimap2`) in box **3** is required if the input is the raw sequences in FASTA/FASTQ format. Another option is to run the alignment independently and provide SAM/BAM input for the next stage of bridging and assembly. This stage is controlled by buttons in box **4**: the START button ignites the process while the STOP button can prematurely terminate it and output the assembly result till that moment. The plots from the right panel (**5**) depicts real-time statistics of the assembly contigs inferred from the graph. From the second window (**6**), the colored vertices imply unique contigs while the white ones involve either unspecified or repetitive elements. The number of different colors (other than white) indicates the amount of abundant groups being detected as population bins (*e.g.* chromosome versus different plasmids, or different bins in metagenomics).

A proper combination of command line and GUI can provide an useful streaming pipeline that copes well with MinION output data. The practice is similar to the previous developed pipelines [2, 10, 11] that allow the analysis to take place abreast to a nanopore sequencing run.

Hybrid assembly for synthetic data sets

To evaluate the performance of the method, `npGraph` was benchmarked against `SPAdes` with its hybrid assembly module [5], `npScarf` with/without assembly graph integrated, and Unicycler version 0.4.6 on the latter's testing data [3] . This data set were simulations of Illumina and MinION raw data, generated *in silico* based on random and available microbial references. Specifically, the synthetic Illumina data was generated by using a wrapper script of ART [12] that allows uniform coverage for circular genome sequencing with justifiable depths. PBSIM [13], on the other hand, was used to simulate the long reads.

There were three settings for each of the synthetic raw data (*good, medium, bad*) corresponding to the quality, yield and length of reads being generated. In the SGS simulation, the *bad* data consist of 100bp paired-end reads with low and uneven coverage (40X) leading to many dead ends in the assembly graph due to missing regions in the genome. The read length was 125bp in the *medium* setting with better depth distributions that could cover the genome better.

Finally, the *good* option provided best data sets with 150bp of read length and 100X coverage. While the quality of SGS reads would determine the assembly graph nature, the nanopore data plays critical role in graph resolving. The three settings leveled up the depth (8X, 16X, 32X) and at the same time, average length (5Kbp, 10Kbp, 20Kbp) and maximum identities (90% ,95% ,98%) respectively. We only considered the *good* configurations of both platforms for the sequence data being tested. Also, since the other comparative tools do not support streaming assembly, there were only batch-mode runs being carried out and the reciprocal results were examined by QUAST 5.0.2 [14].

Table 1: Comparison of assemblies produced in batch-mode using `npGraph` and the comparative methods on 5 synthetic data sets taken from https://cloudstor.aarnet.edu.au/plus/index.php/s/dzRCaxLjpGpfKYW

Method	Assembly size (Mb)	#Contigs	N50 (Kp)	Mis-assemblies	Error (per 100 Kb)	Run times (CPU hrs)
random sequence with repeats						
SPAdes	3.928	226	40.5	0	0.00	0.95
SPAdes-Hybrid	4.109	3	4,000.0	0	0.85	1.196
Unicycler	4.110	3	4,000.0	0	0.47	6.783
npScarf	4.251	9	3,952.2	27	8.74	0.95 + 0.39
npScarf_wag	4.554	9	3,999.6	37	6.16	0.95 + 0.45
npGraph (bwa)	4.110	3	4,000.0	0	0.47	0.95 + 0.33
ngGraph (minimap2)	4.110	3	4,000.0	0	0.47	0.95 + 0.02
Mycobacterium tuberculosis H37Rv						
SPAdes	4.371	114	125.5	1	1.51	1.55
SPAdes-Hybrid	4.411	1	4,411.2	0	1.73	1.68
Unicycler	4.412	1	4,411.5	0	2.56	6.36
npScarf	4.446	4	4,389.9	12	11.41	1.55 + 0.78
npScarf_wag	4.408	1	4,407.6	2	7.01	1.55 + 0.79
npGraph (bwa)	4.411	1	4,411.6	0	7.28	1.55 + 0.63
ngGraph (minimap2)	4.411	1	4,411.4	0	7.01	1.55 + 0.02
E. coli O25b H4-ST131						
SPAdes	5.173	159	191.0	1	1.69	1.26
SPAdes-Hybrid	5.249	7	5,109.6	0	2.65	1.40
Unicycler	5.249	3	5,109.8	0	4.29	4.70
npScarf	5.354	7	5,087.5	14	29.12	1.26 + 0.78
npScarf_wag	5.413	7	5,108.1	6	30.29	1.26 + 0.78
npGraph (bwa)	5.252	3	5,112.3	0	16.37	1.26 + 0.66
ngGraph (minimap2)	5.250	3	5,111.1	0	14.61	1.26 + 0.03
Streptococcus suis BM407						
SPAdes	2.119	81	131.0	0	3.84	0.59
SPAdes-Hybrid	2.147	48	1,438.0	0	0.98	0.65
Unicycler	2.171	2	2,146.2	0	2.99	2.58
npScarf	2.220	4	2,120.0	9	97.20	0.59 + 0.31
npScarf_wag	2.245	4	2,128.3	3	89.64	0.59 + 0.31
npGraph (bwa)	2.167	6	2,146.7	0	26.77	0.59 + 0.21
ngGraph (minimap2)	2.167	6	2,146.2	0	22.53	0.59 + 0.01
Acinetobacter AB30						
SPAdes	4.134	265	42.5	0	3.23	0.95
SPAdes-Hybrid	4.287	49	3,308.0	0	5.04	1.84
Unicycler	4.333	1	4,333.0	1	6.95	5.27
npScarf	4.595	11	4,299.7	1	120.99	0.95 + 0.45
npScarf_wag	-	-	-	-	-	-
npGraph (bwa)	4.317	6	2,766.9	1	39.82	0.95 + 0.41
ngGraph (minimap2)	4.337	1	4,336.8	0	24.71	0.95 + 0.03

Table 1 shows evaluation results for 5 synthetic data sets, the output of the full run can be found in Supplementary Table 1. In the first column of applied methods, beside `Unicycler` and `hybridSPAdes`, the original scaffolder `npScarf` was included as well as `npScarf_wag` – its modified version with assembly graph integrated. On the other hand, `npGraph` can use 2 different aligners, `BWA-MEM` and `minimap2`, for its bridging phase thus both practices were included in this comparison.

In general, the graph integrated version of `npScarf` improved the assembly results in terms of mis-assemblies and error reduction while virtually consuming similar resources compared to the original version. The only exception where

the number of mis-assemblies being increased was the simulation of a random sequence with many repeats. There were 10 more mistakes detected using the later version `npScarf_wag`. However, with additional investigations, we found that the number of mis-assemblies on the true positive circular sequences (3 from the reference) has been significant reduced by applying assembly graph for `npScarf`. The errors mostly came from redundant sequences output from the software due to the failure in estimation of contig multiplicity. Even though using assembly graph for gap fillings, there were no changes in the way to determine if a contig is unique or not from `npScarf_wag`. It still relied on the length and coverage statistics, *i.e. Astats* [15] to find anchors that were critical for the backbone construction of the assembly. Redundant path findings for false positive replicons consequently returned additional wrong translocations in the final contigs which were reported by QUAST. Other than that, the method had successfully produced better assemblies than the original. Regarding *Mycobacterium tuberculosis* H37Rv, the number of mis-assembled breakpoints had been trimmed down from 12 to 2, while the number of final contigs had reduced from 4 to only one as in the reference. Results in cases of *E. coli* O25b H4ST131 and *Streptococcus suis* BM407 also showed enhancements in terms of those categories as well as N50 statistics. There was improvements considering the nucleotide errors (mismatches and indels) as well from aforementioned data sets, except for the *E. coli* when slightly more mismatches had been detected. However, these errors can be corrected by running polishing tools with the raw Illumina data afterward.

For *Acinetobacter* AB30 synthetic data, it was an deficiency for `npScarf_wag` in traversing the graph to find candidate paths for a bridge of long distance due to particular large search space. The exhaustive, naive DFS implementation for this version of `npScarf` required a lot of memory to traverse a complex assembly graph that usually exceed a normal desktop's capacity. This issue has been fixed in `npGraph` when Algorithm 1 was used on the definitive graph. This resulted in completed runs of the assembly process for all data sets with the similar number of mis-assemblies compared to the best figures in this category. As shown in Table 1 and 1, the assembly graph based methods offered significant improvements when compared to `npScarf`. Not only because of the clear drops with respect to mis-assemblies and errors, but it was also reflected by the number of final contigs and their N50 as well.

To align the long reads to the assembly graph components, either `BWA-MEM` [16] or `minimap2` [17] was invoked in `npGraph`. The former option was inherited from `npScarf` pipeline with the intact parameters while the latter was used with the recommended settings (`-k15 -w5`) for the best sensitivity working on MinION data. Even though, `BWA-MEM` normally reported more hits than `minimap2` but at the same time, was responsible for more false positive alignments. For instance, regarding the last data set from Table 1, the assembly of `npGraph` using `BWA-MEM` was suffered from the ambiguous alignments thus more fragmented than the other counterparts. On the other hand, referring to more complicated graphs from *Acinetobacter* A1 and the yeast *Saccharomyces cerevisiae* S288c from Table 1, the number of mis-assemblies from using `minimap2` were increased due to the lacks of appropriate alignments to support accurate bridging process. However, under almost circumstances, using either aligners would result in final assemblies with similar qualities. Furthermore, in terms of running time and resources required, `minimap2` proved to be the best option. The total CPU hours had been trimmed down drastically with the new aligner, making `npGraph` the fastest hybrid assembler available. This feature is certainly more favoured to a real-time assembly as well. As the consequence, as long as `minimap2` is expected to replace `BWA-MEM` for long-read sequencing data alignment, it would likewise become the main aligner for `npGraph` pipeline in the future.

It is noteworthy to discuss in more details the errors addressed in the above assembly methods. This figure measured the total mismatches and indels per 100kpb from the assembly sequences when mapping to the reference. As expected from hybrid assemblies where Illumina sequencing data were used as the main building blocks, the figures were hardly bigger than 100 (equivalent to 0.1% error rate) for almost every case. In addition, the indels errors, which mainly caused by TGS data, 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. From all the hybrid assemblers, `hybridSPAdes` reported results with highest fidelity. This meant that the performance its decision-rule algorithm `exSPAnder` [6] was the most accurate amongst all path finding methods. As the trade-off, there were fewer connections satisfying its quality threshold, resulting in the fragmented assemblies in cases of *Streptococcus suis* or *Acinetobacter* samples (Table 1 and 1). `Unicycler`, which employs an algorithm based on semi-global (or glocal) alignments [18] with the consensus long reads, returned the second best reliable and at the same time, closest-to-complete results overall. `npScarf`, on the other hand, exploited the long reads for the gap filling thus inherited the high error rates from them. By integrating the assembly graph for the task, the errors were reduced in general (random sequences, *M. tuberculosis*, *S.suis* from Table 1) but not completely since the mis-placed contigs were still not resolved in other circumstances. `npGraph` significantly reduced the errors compared to `npScarf`, however the figures were still higher than the those of the best counterparts. This implied a more robust decision making system is needed in `npGraph`'s real-time path finding module for even better output's accuracy.

Hybrid assembly for real data sets

Several sequencing data sets of actual bacterial samples [19] were used in this scenario. The data included both Illumina paired-end and MinION sequencing based-call data for each sample. Unlike previous settings, rather than the default output, the optimal `SPAdes` assembly graph detected by `Unicycler` were used for `npGraph` algorithm as well. This step had been proved to be useful in selecting the best graph available, amongst `SPAdes` runs with ranging k-mer values, that

have dead-ends and number of contigs minimized [3]. Likewise, as mentioned before, the quality of the initial assembly graph would considerably influence the final results of `npGraph`.

Due to the lack of available reference genomes, fewer statistics were 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 shows the benchmark results of `npGraph` (using `minimap2`) against `Unicycler` 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)
Citrobacter freundii	**5,029,534**	**5,029,486**	Chromosome
CAV1374	**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	4,806,666|2	4,858,438|2	Chromosome
CAV1411	**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	6,153,947|5	**6,155,762**	Chromosome
CAV1015	**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	-

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 100Kpb) were classified as IncFIB while the other two mid-size replicons, 85.6Kbp and 43.6Kpb, 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 more challenging as the chromosomal DNA sequence was 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 14Kpb 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` for the final contig of length 11.6Kbp where it failed to combine two smaller sequences into one.

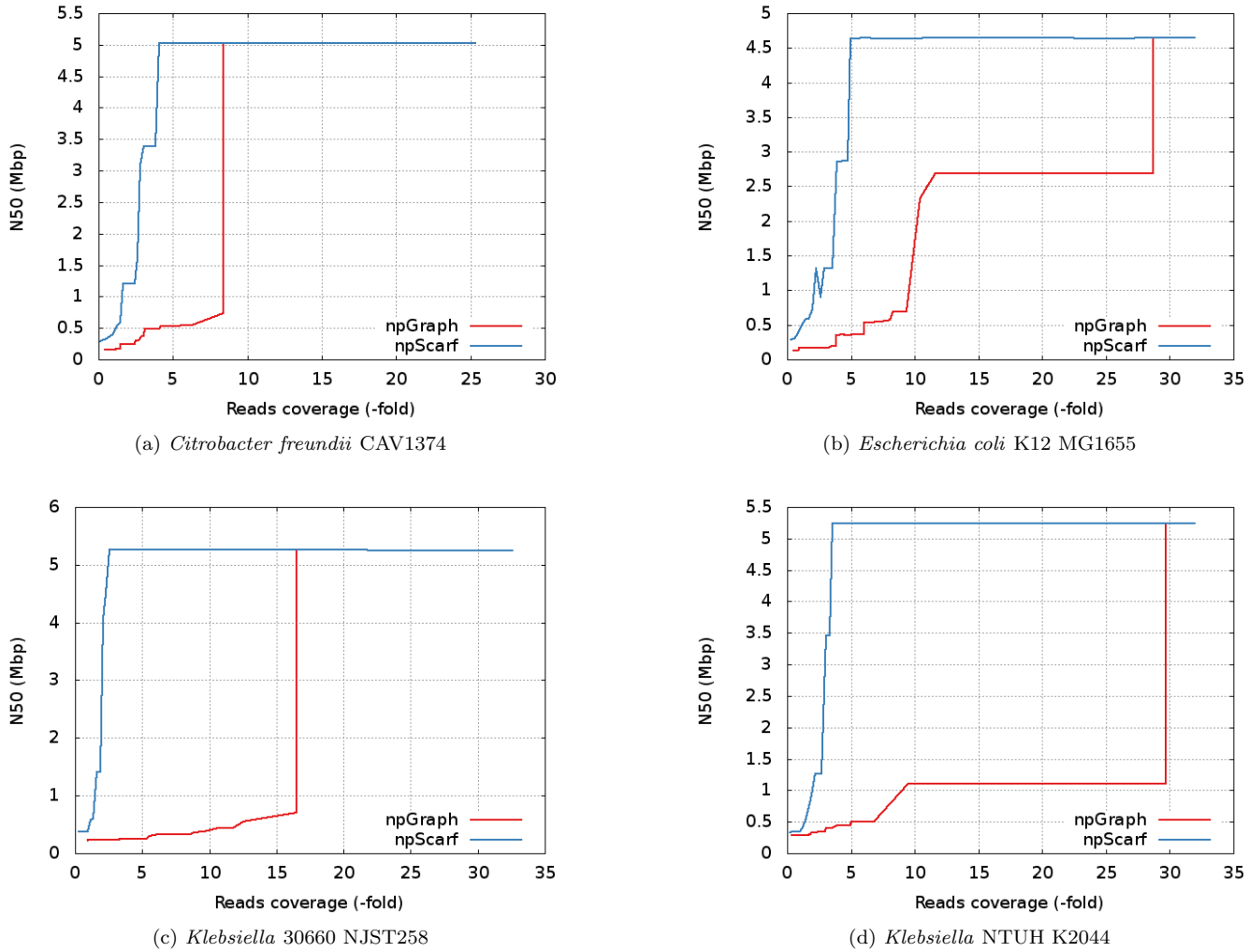
In addition to what presented in Table 2, dot plots for the pair-wise alignments between the assembly contigs were generated and can be found in Appendix Figure 1. Interestingly, beside all other agreements, there was a structural difference using two methods for *E. cloacae* CAV1411 genome assembly. 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 [21]), 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`.

Real-time mode hybrid assembly

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. `npScarf_wag` basically scaffolds the pre-assembly contigs in the same manner with the original version thus was not discussed here.



(a) *Citrobacter freundii* CAV1374

(b) *Escherichia coli* K12 MG1655

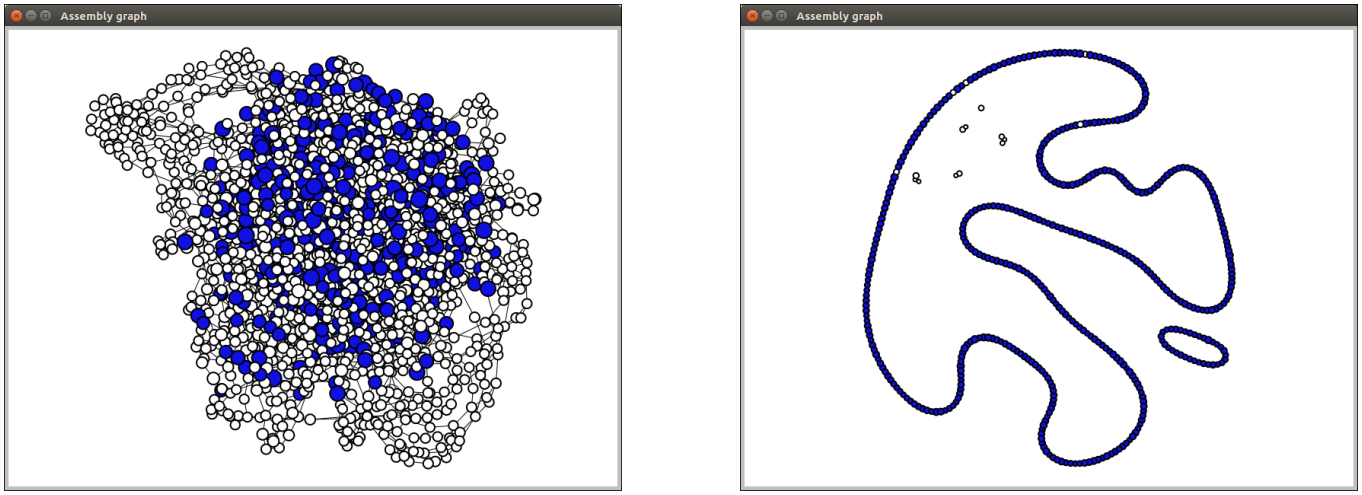
(c) *Klebsiella* 30660 NJST258

(d) *Klebsiella* NTUH K2044

Figure 2: N50 statistics of real-time scaffolding by `npScarf` versus `npGraph`.

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 implemented a more 'conservative' approach of bridge construction with at least 3 supporting long-reads for each to prevent any potential mis-bridging. 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. Once a unique path has been determined, the bridge can be formed to connect the fragments together into longer sequences.

Figure 3 shows an example of the real-time graph resolving process being displayed on GUI. The result graph, after cleaning, would only report the significant connected components that represents the final contigs. Smaller fragments,



(a) Initial graph (b) Resolved graph

Figure 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.

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`.

Conclusions

Assembly graph is the data structure describing the assembly process at a lower level of more details. The current chapter brings this informative knowledge to the original `npScarf`'s mechanism in two ways, either partly for the gap-filling module only (`npScarf_wag`) or completely as the building block of the assembly construction method (`npGraph`).

While the first approach showed improvements in terms of base's accuracy for the final results, it had limited ability in rectifying the mis-assemblies from the original version due to false positive alignments at contig level. The issue can be tackled by applying the assembly graph completely as in `npGraph`. With a more conservative bridging method applied, the new method may consume more data to confidently construct the assembly but at the same time, the number of mis-located fragments has reduced significantly. Furthermore, users can monitor the assembly in an interactive way providing the assembly GUI.

Compared to the other hybrid assemblers of similar methodology, such as `hybridSPAdes` and `Unicycler`, there are still rooms for further development of `npGraph`. More comprehensive pre-processing step is needed for a better input graph possible since it would affect the completeness of final results. Importantly, a robust real-time voting system should be implemented to be able to detect the most probable path amongst all candidates in an efficient way. This would alleviate the data consumption of the `npGraph` while at the same time, maintain high confident and accurate scaffolding.

Methods

Basically the work flow of `npGraph` consists of 3 main modules: abundance binning, graph resolving and assembly output. The first step is to bin the contigs into different groups of *population* based on their coverage. Each population would include contigs of the similar abundance in the final assembly sequences, *e.g.* chromosome , plasmids, or even particular species genome in a metagenomics community. The binning phase would assist to differentiate between repetitive contigs and unique ones. By using this information, 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.

Initial binning contigs into groups of abundances

Each contig is represented as a node in the assembly graph and the edge between two nodes indicate their overlap (link) properties. This step is to cluster the longest nodes (longer than 10Kbp) into different sets, namely *core* groups, based on their coverage values.

DBSCAN clustering algorithm [22] is applied for this task. The rationale is to approximate a coverage value of a significant contig (which consists of 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 [23], or relative entropy, of two Poisson distributions.

Assume 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

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

Coverage re-estimation

Due to the possible divergence of sequencing coverage relative to the real abundance of sequences, especially the short ones, an optimization step is implemented to alleviate this issue. The coverage measure of nodes (which represent contigs) are spread throughout the graph via edges that connect them for comparison and calibration. The assignment of coverage to edges is also helpful to identify multiplicity in later step.

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 Newton or gradient 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.

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 variances of coverage values. For that reason, they are only used as augmented information to calculate candidate paths' score in the next step.

Building bridges in real-time

Bridge is the data structure designed for tracking the possible connections between two unique contigs (anchor nodes in the assembly graph). Here we take advantage of the graph topology and nodes' multiplicity information to employ a dynamic bridging mechanism. This procedure considers the dynamic changing of multiplicity property for each node, meaning that a n-times repetitive node can become a unique node at certain time point when its $(n-1)$ occurrences are identified and assigned in appropriate unique paths.

Other than `npScarf`, a bridge in `npGraph` has several completion levels. A bridge is only created with at least one unique contig as an end, if so it has level 1 of completion. If both ends are identified, the level is 2. The number is greater than that only if paths connecting two ends are found. A bridge is known as fully complete (level 4) if there is only one unique linking path left. Given a bridge with 2 ends, a path finding algorithm (described in next section) is invoked to find all candidate paths. 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 best ones are voted for the next round.

Whenever a bridge becomes complete thanks to the voting system, the assembly graph is *transformed* or *reduced* by replacing its unique path by an 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.

Path finding algorithm

Algorithm 1: Pseudo-code for finding paths connecting a bridge with 2 ends.

Data: Assembly graph $G\{V, E\}$
Input: Bridge $B = (\overrightarrow{v_1}, \overrightarrow{v_2})$ with two ending unique bidirected nodes $\overrightarrow{v_1}, \overrightarrow{v_2}$
Output: Set of candidate paths P connecting B

```
1  begin
2      d:=B.length()                          // length of the bridge or the distance between 2 ending nodes
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
```

In `npScarf` with assembly graph, the path finding algorithm is the original DFS (depth first search) which becomes computationally expensive when the traversing depth increases. For `npGraph`, we implement a modified stack-based version utilizing Dijkstra's shortest path finding algorithm [24] to reduce the search space.

Algorithm 1 demonstrates the path finding module in general. 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.

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.

It is worth to mention 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* DBG-derived assembly graph, the value of an edge is $-k$.

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 present a contig for the result. 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 every edge along the path, $e_i, \forall i = 1, \ldots, k$, must be the only option to traverse from either v_{i-1} or v_i following the transition rule. To decompose the graph, we can just 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.

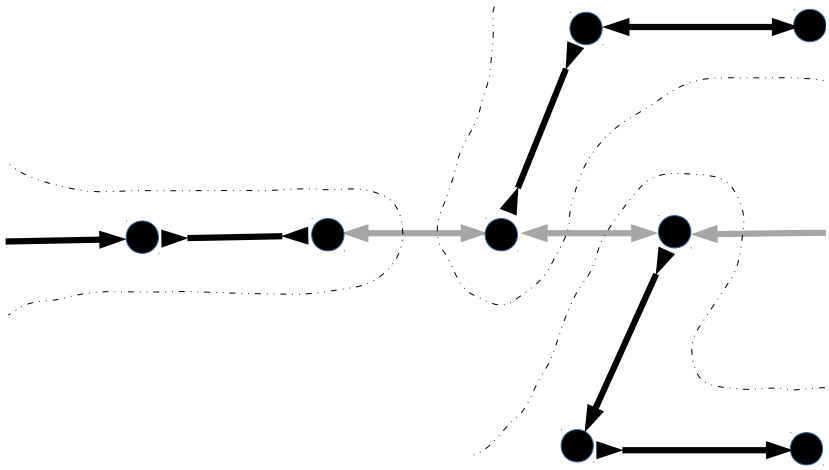


Figure 4: Example of graph decomposition into longest straight paths. Branching edges are masked out (shaded) leaving only straight paths (bold colored) to report. There would be 3 contigs extracted by traversing along the straight paths here.

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.

The final assembly output contains files in both FASTA and GFA v1 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 un-masked graph such as nodes, links and potential paths between them.

References

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

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

[3] 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.

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

[5] 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.

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

[7] 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.

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

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

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

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

[12] Huang W, Li L, Myers JR, Marth GT (2012) ART: a Next-generation Sequencing Read Simulator. *Bioinformatics* 28(4):593–594.

[13] Ono Y, Asai K, Hamada M (2012) PBSIM: PacBio Reads Simulator – Toward Accurate Genome Assembly. *Bioinformatics*.

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

[15] Myers EW et al. (2000) A Whole-Genome Assembly of Drosophila. *Science* 287(5461):2196–2204.

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

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

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

[19] 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).

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

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

[22] 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.

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

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