Algorithms for Big Data Problems in de Novo Genome Assembly

Anand Srivastav, Axel Wedemeyer(✉), Christian Schielke, and Jan Schiemann

Kiel University, Kiel, Germany
{srivastav,wedemeyer,schielke,schiemann}@math.uni-kiel.de

Abstract. De novo genome assembly is a fundamental task in life sciences. It is mostly a typical big data problem with sometimes billions of reads, a big puzzle in which the genome is hidden. Memory and time efficient algorithms are sought, preferably to run even on desktops in labs. In this chapter we address some algorithmic problems related to genome assembly. We first present an algorithm which heavily reduces the size of input data, but with no essential compromize on the assembly quality. In such and many other algorithms in bioinformatics the counting of k-mers is a botleneck. We discuss counting in external memory. The construction of large parts of the genome, called contigs, can be modelled as the longest path problem or the Euler tour problem in some graphs build on reads or k-mers. We present a linear time streaming algorithm for constructing long paths in undirected graphs, and a streaming algorithm for the Euler tour problem with optimal one-pass complexity.

Keywords: De novo genome assembly · Data reduction · Euler tour · Semi-streaming longest path · External memory counting

1 Reduction of Input Data in Genome Assembly

Sequencing is a chemical and physical process in which DNA is 'crushed' into very small parts ('fragments') which are 'read' to strings called reads, containing information of the sequence of nucleotides. Reads are of limited length and contain errors (Fig. 1).

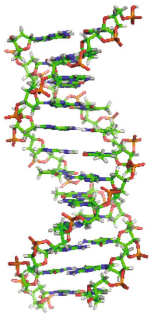
Sequencing of big genomes and other samples is a computationally challenging recent trend for two main reasons:

– sequencing became much cheaper (price decreased by more than 100.000× since year 2000), so researchers can afford to create much bigger data sets than ever before (Fig. 2)
– it was discovered that most bacteria (90%–99%) can't be cultivated, so metagenomic sequencing is (nearly) the only way to assess them .

1.1 Reads, Coverage and Assembly

A **read** is a string over the alphabet $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}, \mathsf{N}\}$ where A, C, G, T are the four nuclobases and N is a place holder for an unknown nucleotide. The maximal read length depends on the sequencing technology used. For the Illumina sequencing technology the read length initially was 34 and now can be as high as 300. Other sequencers allow

Biochemical molecule $\to \Sigma^* = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}, \mathsf{N}\}^*$



```
@EAS18:1:1:1:1:971:0/1
NGTCTTTTATAACCAGTTCACCTATAGCAAAGACGCTATTCATACCGAAGGCGTCATT
+
B`bbbb`bb`_abaXU\Za\ab_X_`Xa_R[X\a_b_YX`a__X_b`YH\T`b`_\_]
@EAS18:1:1:1:1:971:0/2
NAAANCGGCAANTTAANNNTAACGCCCATAATGCCGATAAACCCCCATTTAAAAAANT
+
B_YaBbaaaTWBa^P`BBBb[aa`a__P_Z`aab`Z]__^\``a_\[abb[IZPPbBa
```

Fig. 1. A shortened paired (Illumina) read

for longer reads at the price of a higher error rate (and higher costs). Illumina produces substitution errors with an error rate of roughly 1%. In most cases, so called paired reads are generated where in a first step pieces of DNA of a known length are produced which are than sequenced from both sides (e.g.: a paired read with a read length of 150 contains one string over Σ for the first 150 nucleotides and a second string over Σ for the last 150 nucleotides).

The sequencer also outputs so-called phred scores quantifying the error probability of each nucleotide read (Quality $Q = -10\log_{10} P$ where P is the error probability).

Genome assembly (or just assembly) is the task to reconstruct the complete genome of the sequenced species using the reads only (*de novo assembly*) or the reads and a reference genome (*mapping* or *reference based assembly*). It's like a puzzle with millions of small parts, unknown overlaps and a lot of the parts containing errors.

In our work we focus on de novo assembly, or just assembly for the rest of this chapter.

For a sequencing data set, the **coverage** of a genomic position A is the number of reads in the data set which contain A. The coverage of the whole data set is the average over the coverages of all genomic positions. The empirical 'optimal' coverage for a de novo assembly is about 20 *at every position*. A coverage higher than 20 means redundant data. Some sequencing protocols, especially single cell MDA (multiple displacement amplification), produce read sets with an extreme uneven coverage distribution. Metagenomic data sets may have an uneven coverage distribution, too, when both abundant and rare species are sequenced. Given a string σ over the nucleotide alphabet, a k**–mer** is a sub-string of σ of length k.

```
GTCTTTTATAAC
GTCTTT
 TCTTTT
  CTTTTA
   TTTTAT
    TTTATA
     TTATAA
      TATAAC
```

the 6–mers of a string

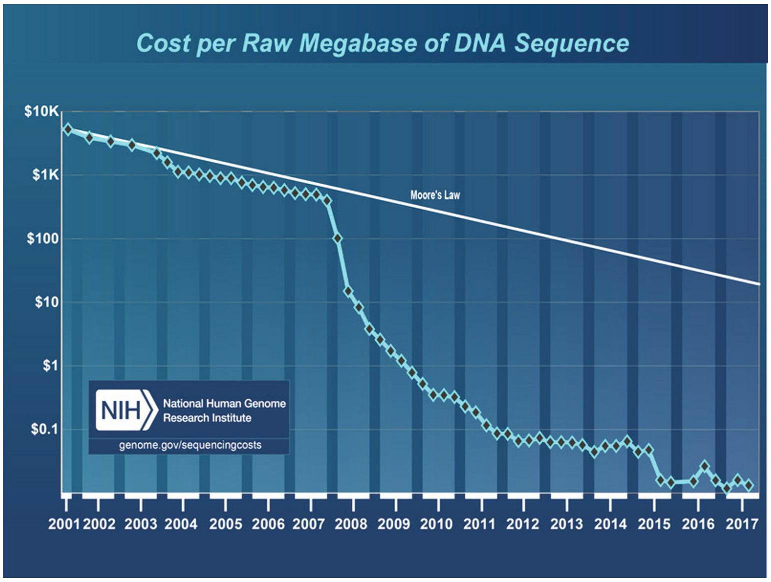


Fig. 2. The cost of sequencing a human genome, source: NIH

Most bacteria can't be cultivated in the lab. Therefore, it is not possible to create a homogeneous sample of thousands or millions of equal cells as in a 'normal' sequencing setting.

As a consequence, **single cell** sequencing protocols, like the multiple displacement amplification (MDA) have been developed which are able to amplify the genome of a single bacterial cell. A drawback of these methods is a strong amplification bias (called 'Preferential amplification' and 'Allelic dropout') between different regions of the genome, meaning that the coverage of some regions of the genome might overshot 100.000X, while other regions are not covered at all.

A **metagenome**, introduced by [14], is, according to wiktionary, '*All the genetic material present in an environmental sample, consisting of the genomes of many individual organisms*'. In other words, in a metagenomic experiment, you are interested in

– all the genes/DNA
– of everything living
– at a specific location

The experiment is conducted by collecting a sample from the desired environment, isolating the DNA from it and sequencing it with a Next Generation Sequencing (NGS) system. There are three different types of metagenomic experiments with different goals:

– *phylogenetic profiling:* based upon the 16S ribosomal RNA found in the sample, reconstruct which families of bacteria live in the probed environment (and how abundant they are). Basis: each (bacterial) cell has ribosomes. The coding genes for these essential proteins are widely conserved (which makes it possible to identify these genes), but they also include less conserved regions which differ between different families or even species.

– *directed/guided assembly of specific genes:* based upon some known variants of a gene (or even whole genomes), all existing variants in a specific environment are to be assembled.
– *de novo assembly of all species in the sample:* all genomes of all species in the probed environment are to be assembled using the output of the sequencer only.

In our work, we focus on de novo assembly.

The main problem of metagenome assembly is non-uniform coverage: some species in the sample are much more abundant than others. The goal is to assemble all their genomes. The following issues may arise:

– to be able to assemble the less abundant species in the sample, a high number of reads have to be generated ($\rightarrow$ high coverage sequencing).
– the huge input files force the assembler programs to use huge amounts of RAM and running time. For bigger projects, even $1TB$ of RAM might not be enough.
– for the assembler, its often hard to tell whether a rare sequence belongs to a rare species or whether it is a sequencing error.

1.2 The Bignorm Algorithm

The basic idea of read filtering is to remove reads from a single cell or metagenome data set without losing information, and in this way to reduce the size of the problem, possibly escaping the 'big data curse'. This is possible if only those reads which have overlapping genomic regions with high coverage are removed. A good read filter should remove as many reads as possible, without lowering the coverage of the sequenced genome below the desired threshold at any position and without increasing the error rate of the data set.

Highly memory efficient algorithms are sought to solve this problem. Brown et al. invented an algorithm named *Diginorm* [1] for read filtering that rejects or accepts reads based on the abundance of their *k*–mers. The name *Diginorm* is a short form for *digital normalization*: the goal is to normalize the coverage over all loci, using a computer algorithm after sequencing. The idea is to remove those reads from the input which mainly consist of *k*–mers that have already been observed many times in other reads. Diginorm processes reads one by one, splits them into *k*–mers, and counts these *k*–mers. In order to save RAM, Diginorm does not keep track of those numbers exactly, but instead keeps appropriate estimates using the count-min sketch CMS [4]. A read is accepted if the median of its *k*–mer counts is below a fixed threshold, usually 20. It was demonstrated that successful assemblies are still possible after Diginorm removed high amount of the data.

Diginorm is a pioneering work. However, the following points, which are important from the biological or computational point of view, are not covered by Diginorm. We have included them in our algorithm called Bignorm [29 SPP]:

(i) we incorporate the important phred quality score into the decision whether to accept or to reject a read, using a quality threshold. This allows a tuning of the filtering process towards high-quality assemblies by using different thresholds.

(ii) when deciding whether to accept or to reject a read, we do a detailed analysis of the numbers in the count vectors. Diginorm merely considers their medians.

(iii) we offer a better handling of the N case, that is, when the sequencing machine could not decide for a particular nucleotide. Diginorm simply converts all N to A, which can lead to false k–mer counts.

(iv) we provide a substantially faster implementation. For example, we include fast hashing functions (see [10,30]) for counting k–mers through the count-min sketch data structure (CMS), and we use the C programming language and OpenMP.

Let us fix the following parameters:

– N-*count threshold* $N_0 \in \mathbb{N}$, which is 10 by default;
– *quality threshold* $Q_0 \in \mathbb{Z}$, which is 20 by default;
– *rarity threshold* $c_0 \in \mathbb{N}$, which is 3 by default;
– *abundance threshold* $c_1 \in \mathbb{N}$, which is 20 by default;
– *contribution threshold* $B \in \mathbb{N}$, which is 3 by default.

When our algorithm has to decide whether to accept or reject a read $i \in \mathbb{N}$, it performs the following steps: If the number of N symbols counted over all read positions is larger than N_0, the read is rejected. Otherwise, those parts of the read having phred scores of or above Q_0 are converted into a vector H of *high-quality k–mers*.

Using the CMS, it is then checked how many times these k–mers have been seen in the accepted reads so far (function $\widehat{c}(\mu)$) and two counters hold the results:

$$b_0 := |\{\mu \in H\,;\, \widehat{c}(\mu) < c_0\}|,$$
$$b_1 := |\{\mu \in H\,;\, c_0 \leq \widehat{c}(\mu) < c_1\}|$$

Note that the frequencies are determined via CMS counters and do not consider the position p at which the k–mer is found in the read string. The read is accepted if and only if at least one of the following conditions is met:

$$b_0 > k, \tag{1}$$
$$\sum_{s=1}^{m(i)} b_1 \geq B. \tag{2}$$

The motivation for condition (1) is as follows. According to [15], most errors of the Illumina sequencing platform are single substitution errors and the probability of appearance of an erroneous k–mer in the genome, caused by an incorrect reading of a nucleotide, is quite low. Thus, k–mers produced by single substitution errors are likely to have very small counter values in the CMS (less than c_0 times) and can be considered as rare k–mers. One such error can only effect at most k k–mers. So if we count more than k rare k–mers, they most likely are not a result of one single substitution error. If we assume that the probability of multiple single substitution errors in a read is smaller than the probability of error-free rare k–mers, we should accept this read.

Condition (2) says that in the read, there are enough (namely at least B) k–mers where each of them appears too frequently to be a read error (CMS counters at least c_0), but not that abundant that it should be considered redundant (CMS counters less than c_1).

Algorithm 1: Bignorm

```
Input: fastq–files as produced by an (Illumina–)sequencer
Result: filtered fastq–files
Parameter: k–mer size $k$
Parameter: quality threshold $Q_0$
Parameter: rarity threshold $c_0$
Parameter: abundance threshold $c_1$
Parameter: contribution threshold B
Parameter: N-count threshold $N_0$
1  begin
2      initCMS ()
3      foreach read r in input do
4          if count of N in $r < N_0$ then
5              $b_0 = 0$
6              $b_1 = 0$
7              foreach canonical k–mer $\kappa$ in r do
8                  if min (phred_scores ($\kappa$)) $\geq Q_0$ then
9                      $t_\kappa = getCount(\kappa)$
10                     if $t_\kappa < c_0$ then
11                         $b_0 += 1$
12                     else if $t_\kappa \leq c_1$ then
13                         $b_1 += 1$
14             if $b_0 > k$ OR $b_1 \geq B$ then
15                 Add r to Output
16                 foreach canonical k–mer $\kappa$ in r do
17                     $incCount(\kappa)$
```

Results for Single-Cell Assemblies. We tested Bignorm on 13 bacterial single-cell data sets and were able to remove up to 90% of the reads without significant loss of the assembly quality. Some results (median of all samples) (Fig. 3):

Measurement	Filtered/Unfiltered (%)
Read count	2.85
Run time SPAdes Assembler	3.57
Largest Contig	97.56
N50	90.84
Mean Phred Score	103.00

Bignorm heavily cuts away redundant reads (mean, Fig. 4, left-hand side) but is careful in critical regions (P10, Fig. 4, right-hand side).

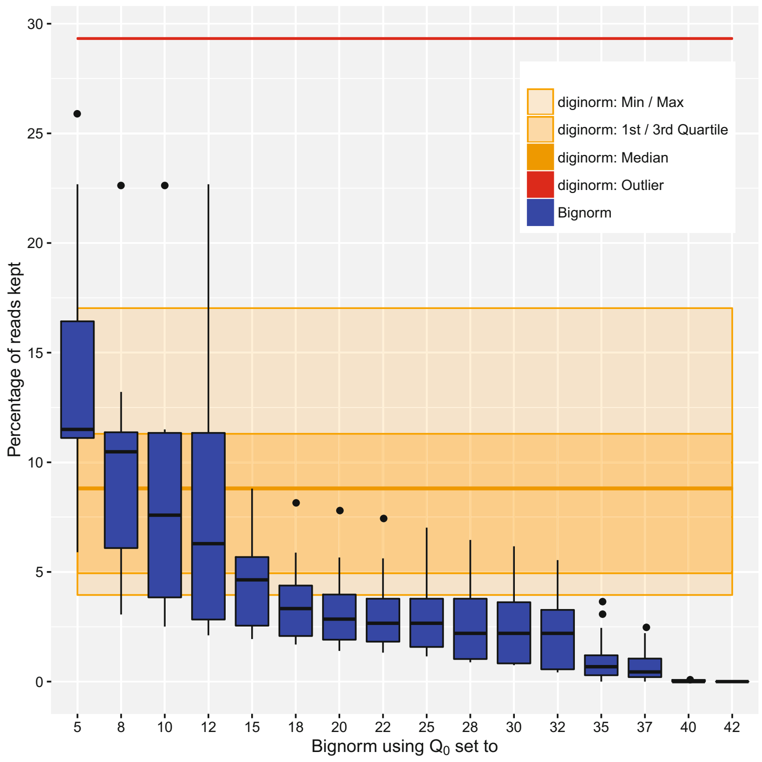


Fig. 3. Reads kept

Results for Metagenomic Assemblies. We tested Bignorm on metagenomic data sets. For data sets with reads of length about 250 base pairs, the results are quite promising and stable. Compared to the single cell case, the results are not that impressive, but compared to the State–of–the–art approach of *sub-sampling* data sets which are too big to be assembled on the given hardware (this means a certain proportion of reads is selected randomly), we could show that by read filtering it is possible to get results which are nearly as good as those of assembling the complete data set, using about the same amount of RAM and in run time as using the sub-sampling approach. The following table gives an impression on the results:

	Raw	Filtered	Sub-sampled (3x)
Largest Contig	2183	1731	1356 $\pm$ 143
Total length	385282	358036	136552 $\pm$ 8406
Genome fraction (%)	15.0	14.0	5.4 $\pm$ 0.3
Predicted genes	689	648	262 $\pm$ 12
RAM needed (GB)	212	100	96 $\pm$ 0.6
Run time (h)	151	52	52 $\pm$2

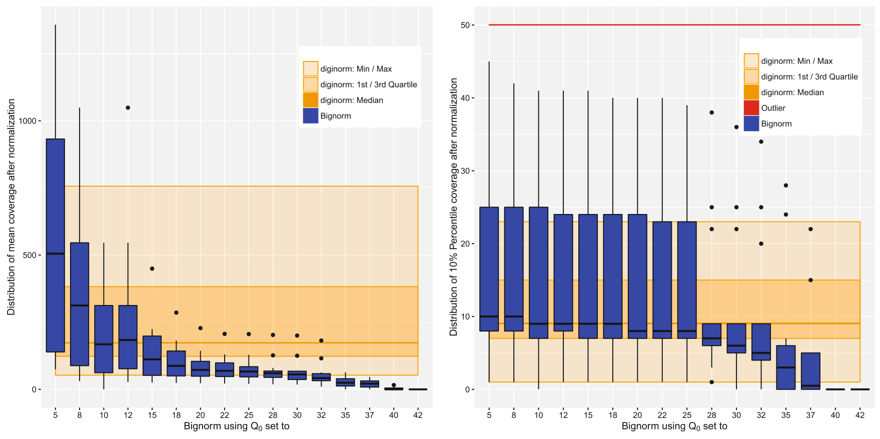


Fig. 4. Coverage: mean and critical region

2 Counting *k*–mers in External Memory (EM)

Many bioinformatics algorithms (e.g., assemblers, error correctors, read normalization) are based on *k*–mers, and that requires to count them (mostly for $21 \leq k \leq 127$). As bioinformatics data sets are growing much faster than RAM sizes, new computational models are needed. (We could show that hash–based counting, which is state of the art in current software, will produce $\mathcal{O}(n^2)$ hash table dumps when the number of different *k*–mers is much bigger than the number of slots in the hash table.)

Some examples of recent *k*–mer counting algorithms are:

– `jellyfish (2)` [19]: the standard, *hash-based k*–mer counter
– `dsk` [26]: the first *EM-based* counter
– `kmc (2/3)` [8,9,17]: the state–of–the–art *EM-based* counter.
– `bloomfish` [12]: *MPI-based Map–Reduce* framework for counting
– `squeakr` [21]: based on *counting quotient filter* (a probabilistic data structure)
– `turtle` [27]: using a *Bloomfilter* and *sort–and–compact* algorithm

We need some notations:

– All strings are based on the biological alphabet $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$.
– So the base set for a *k*–mer is $\mathscr{M} := \Sigma^k$ and $m := |\mathscr{M}| = 4^k$.
– The input of a *k*–mer counter is $\eta \in \underbrace{\mathscr{M} \times \mathscr{M} \times \cdots \times \mathscr{M}}_{n\ times} = \mathscr{M}^n$.
– Denote by $\mathscr{C} := \{p \in \mathscr{M} \mid \exists_{i \leq n} : p = \eta_i\}$ the set of *k*–mers occurring in the input at least once, $c := |\mathscr{C}|$.
– Let R be the size (in bytes) of the RAM available.
– Let B the number of bytes needed to count each element of $\mathscr{C}$.

2.1 Counting in RAM

The **straightforward algorithm** for small values of k, counting can be done in RAM. If $m \le R/B$, the following $\mathcal{O}(n+m)$ algorithm can be used:

```
Algorithm 2: trivial counting
Input: fastq–files as produced by an (Illumina–)sequencer
Result: k–mers of input and their number of occurrence in the input
1 begin
2     initialize $c_0^{m-1} = 0$
3     foreach canonical k–mer $\kappa$ in input do
4         $c_\kappa = c_\kappa + 1$
5     for $i \leftarrow 0$ to $(m-1)$ do
6         if $c_i > 0$ then
7             output $i, c_i$
```

For $k = 19$ and one Byte per counter, $4^{13} \approx 275GB$ of RAM is needed.

Hash–Based Counting. Most state–of–the–art k–mer counting programs are based on hash algorithms using open addressing:

– Hash table with h entries of size $B + \lceil \frac{\log_2 m}{8} \rceil = B + \lceil \frac{\log_2 4^k}{8} \rceil = B + \lceil \frac{k}{4} \rceil \rightarrow h(B + \lceil \frac{k}{4} \rceil) \le R$
– For hash size $h \gg c$ the time complexity is $\mathcal{O}(n+h)$. But if $h \approx c$ the run time may increase to $\mathcal{O}(nh)$ and if $c > h$, the program will fail (or dump to external memory)
– Most existing programs will dump the full hash tables and merge them afterwards — for bigger data sets, this merging phase may need days and terabytes of external memory. The runtime depends linearly on the expected value $\mathbb{E}[d(h,n)]$ of the number of hash table dumps. We can show the following formula for the expected value.

Theorem 1 (Gallus, Srivastav, Wedemeyer 2021). *Counting a set of n elements of a population with K different, normally distributed types using a hash table of size h, the expected value of hash table dumps $d(h,n)$ is*

$$\mathbb{E}[d(h,n)] = n \log_{(1-\frac{h}{c_n})} \left(1 - \frac{1}{c_n}\right), \tag{3}$$

where $c_n = K(1-(1-\frac{1}{K})^n)$ gives the number of different (normally distributed) types in a set of size n.

This formula is the basis for further quantifying the log-term in (3). If one can show that this log-term behaves linearly or sublinearly in n in case of including singletons in the the set of k-mers, it would match experimental observations. In fact, a constant portion of the k–mers can be assumed as sequencing errors of which each occures exactly once.

2.2 Counting in External Memory

kmc3 is the presently leading program using the external memory model. It works as follows:

- the input is parsed into $k - x$–mers (a combination of up to 3 k–mers)
- they are split into a prefix and a suffix, the suffixes are written to one temporary file per prefix
- each temporary file is loaded one by one into RAM, sorted (radix sort, library *radul*)
- the sorted $k - x$–mers are unified and counted
- written to a pair of special binary files (one index-file for the prefixes and one with the suffixes and the counts)

Drawback of kmc3: The output files of kmc3 are not completely sorted (due to the introduction of $k - x$–mers in kmc2). Therefore,

- they need to be loaded into RAM completely for read out
- exporting to other formats takes more time than counting
- no compression is in place (although the suffix–files are highly compressible)

As a result, even though kmc3 is the fastest EM k–mer counter available (and the fastest k–mer counter overall under RAM restriction), it is not the perfect choice to be used as a counting module for an EM assembler.

Based on STXXL 1.4.1 [5], in 2018 Christopher Nehls [20] from Kiel University developed a k–mer counter called **xsc** which uses a sorting based approach:

- generate k–mers from input
- sort the k–mers (using the STXXL EM sorter)
- count the k–mers

For $k \leq 32$, xsc outperformed jellyfish and was at least competitive to dsk, but kmc3 was always faster. For $k > 33$ (using uint128 and uint256 classes), xsc was not competitive to the existing counters. The main bottleneck of xsc is the overloaded relational operator (operator<).

2.3 Counting Using a Bloomfilter

Roy et.al. [27] stated that *more than 50% of all k–mers in a sequencing data set may be singletons* — which are not of interest as they were probably introduced by errors. To utilise this, their k–mer counter Turtle uses a upstream Bloomfilter to save space and time in a sorting based approach named ‘sort–and–compact’.

We developed a program which combines the ideas of kmc and the usage of a Bloomfilter, experiments show that the cost of running a bloomfilter is higher than the savings (Fig. 5). What is wrong? Say, we have 100 k–mers,

- 50 singletons (occurring once)
- 50 ‘good’ k–mers occurring 100$\times$ on average

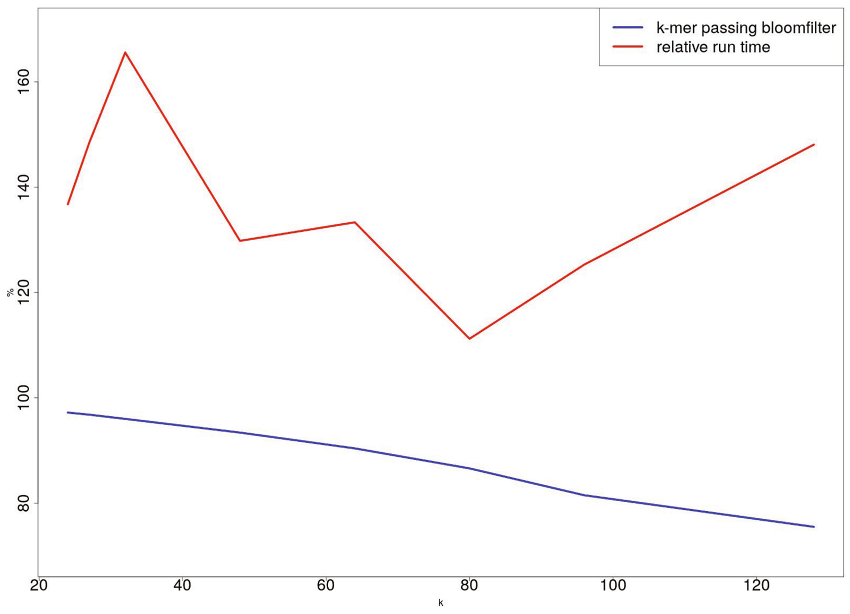


Fig. 5. Comparison of run times using or not using a bloomfilter

Our input contains 5050 k–mers, the bloomfilter removes $100 \rightarrow \approx 2\%$ of the input, not enough to compensate for the running time of the bloomfilter.

Our Current Approach. We have developed the following algorithm which combines sorting and kmc. Experiments are ongoing work:

```
  Input: fastq–files as produced by an (Illumina-)sequencer
  Result: k–mers of input and their number of occurrence in the input
1 begin
2     foreach canonical k–mer $\kappa$ in input do
3         split $\kappa$ into prefix and suffix
4         use turtle–like sort–and–compact per prefix
5         if an array is full then
6             dump the sorted array to EM
7     merge arrays
```

3 A Streaming Algorithm for the Longest Path Problem

In de novo genome assembly, finding a large genome sequence called contig is the fundamental problem. It can be understood as computing a very long path in the associated graph, for example the de Bruijn graph ([3]). Unfortunately, computing the longest path in a graph is an NP-hard problem and the situation is even more worse if the graph is very large. In this chapter, we present a new algorithm for computing a long path, which is surprisingly competitive with RAM-based algorithms.

Graph streaming is a very efficient concept to handle big graphs, where the number of edges is far too large for computations in the main memory. The semi-streaming model was introduced by Feigenbaum et al. [11], and can be briefly described as follows:

In the semi-streaming model, the algorithm is allowed to use at most $\mathcal{O}(n \cdot \text{polylog}(n))$ bits of RAM where n is the number of vertices of the input graph. Because of this restriction, dense graphs where the number of edges is in the order of $\omega n \cdot \text{polylog}(n)$, cannot be processed entirely in RAM. Instead, the edges are presented in a stream where the edges are in no particular order. Typically, it is desired to call only a small number of passes (over the input stream).

3.1 Our Tree-Based Algorithm

We give a streaming algorithm for the longest path problem in undirected graphs with a proven per-edge processing time of $\mathcal{O}(n)$ published in the proceedings of the European Symposium on Algorithms in 2016 [16 SPP]. Our algorithm works in two phases, which we outline here briefly and explain in detail in Sect. 3.1. In the first phase, global information on the graph is gathered in form of a constant number of spanning trees $T_1, \ldots, T_\tau$. This is possible in the streaming model since roughly speaking, for a spanning tree we can "take edges as they come". A spanning tree can be constructed in just one pass—we however use multiple passes and limit the maximum degree during the first passes in order to favor path-like structures and avoid clusters of edges. Experiments clearly indicate that this degree-limiting is essential for solution quality. The spanning trees fit into RAM, since we consider τ as constant (we will in fact have $\tau = 1$ or $\tau = 2$ in the experiments). After construction of the τ trees, they are merged into one graph U by taking the union of their edges. Then we use standard algorithms to determine a long path P in U, isolate P, and finally add enough edges around P to obtain a tree T.

Then, in the second phase, we conduct further passes during which we test if the exchange of single edges of T can improve the longest path in it. (A longest path in a tree can be found by conducting DFS two times [2]; the length of a longest path in a tree is its diameter.) The main challenge in the second phase is to quickly determine which edges should be exchanged. We show that this decision can be made in linear time, hence yielding a per-edge processing time of $\mathcal{O}(n)$.

For a set X, we write x unif X to express that x is drawn uniformly at random from X.

An example run of the Algorithm is shown in Fig. 6.

3.2 Linear Complexity of the Streaming Algorithm

If the cycle C is of length $\Omega(n)$, then a naive implementation requires $\Omega(n^2)$ to find an edge e' to remove (temporarily remove each edge on the cycle and invoke the Dijkstra algorithm). However, we have:

Theorem 2 (Kliemann, Schielke, Srivastav 2016). *Phase 2 can be implemented with per-edge processing time $\mathcal{O}(n)$.*

Algorithm 3: Streaming Phase 1: Spanning Tree Construction

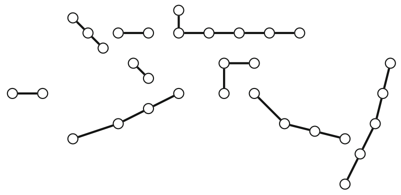
```
Input: connected graph $G = (V, E)$ as a stream of edges, parameter $\tau$,
       degree limit sequence $D = (D_1 \dots D_{q_1})$
Output: spanning tree of $G$
1 foreach $i = 1, \dots, \tau$ do
2     $T_i := (V, \emptyset)$
3     SpanningTree($T_i$)
4 $U := (V, \bigcup_{i=1}^{\tau} E(T_i))$
5 find a long path $P$ in $U$ using Warnsdorf's algorithm
6 $T := (V, E(P))$
7 SpanningTree($T$)
8 return $T$
```

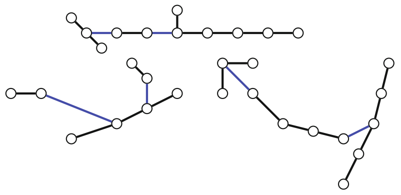
Procedure SpanningTree(T)

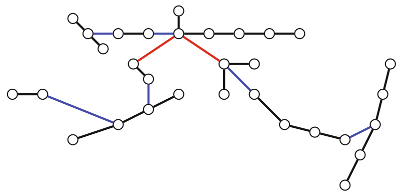
```
Input: forest $T$ on $V$, possibly empty
Output: spanning tree on $V$
1 $r =_{\text{unif}} [m]$
2 fast-forward the stream to position $r$
3 for $p = 1, \dots, q_1$ do
4     while not at the end of the stream do
5         get next edge $vw$ from the stream
6         if $T + vw$ is cycle-free and $\max\{\deg_T(v), \deg_T(w)\} < D_p$ then $T := T + vw$
7         if $|T| = n - 1$ then break
8     rewind the stream to its beginning
```

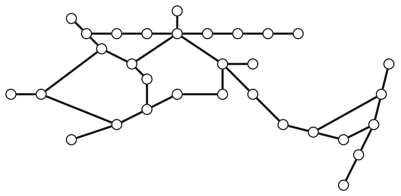
Algorithm 4: Streaming Phase 2: Improvement

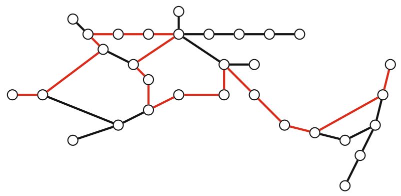
```
Input: connected graph $G$ as a stream of edges, spanning tree $T$, pass limit $q_2$
Output: a (long) path in $G$
1 compute longest path $P$ in $T$ with Dijkstra algorithm
2 for $q_2$ times do
3     rewind the stream to its beginning
4     while not at the end of the stream do
5         get next edge $e = vw$ from stream
6         if $v \in V(P)$ and $w \in V(P)$ then discard and continue with next iteration
7         $T' := T + e$
8         compute fundamental cycle $C$ in $T'$
9         $\ell^* := \max_{f \in E(C) \setminus \{e\}} \ell(T' - f)$
10        if $\ell^* > |P|$ then
11            pick any $e'$ from the set $\{f \in E(C) \setminus \{e\} : \ell(T' - f) = \ell^*\}$
12            $T := T' - e'$
13            update $P$ with longest path in $T$
14 return $P$
```

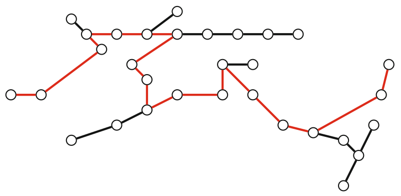

a: Degree Limit $D = 2$

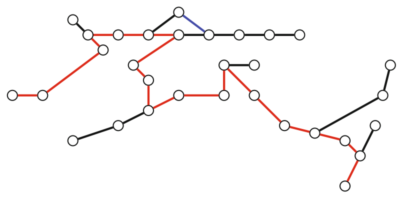

b: Degree Limit $D = 3$

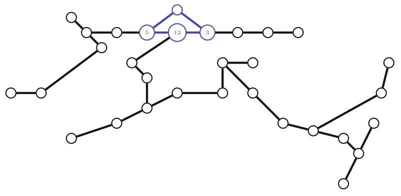

c: Degrees Unlimited


d: Union of Trees

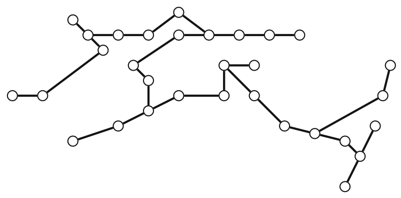

e: Path Found by Warnsdorf's Algorithm


f: New Spanning Tree Built around Path


g: Added Edge from Stream



h: Depths of Trees Extending from Fundamental Cycle


i: Remove Edge from the Cycle

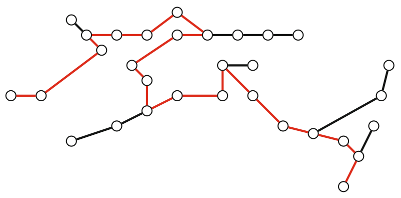

j: New Longest Path

Fig. 6. Example run of the algorithm's steps.

Proof. An $\mathcal{O}(n)$ bound is clear for all lines of Algorithm 4, except Line 9 and Line 11. Denote

$$\ell' := \max_{f \in E(C) \setminus \{e\}} \max\{|P| : P \text{ is path in } T' - f \text{ and } e \in E(P)\}$$

and let $R' \subseteq E(C) \setminus \{e\}$ be the set of edges where this maximum is attained. Then the following implications hold: $\ell' \leq |P| \implies \ell^* \leq |P|$ and $\ell' > |P| \implies \ell' = \ell^*$. This is because if a longest path in $T' - f$ is supposed to be longer than P, it must use e (since otherwise it would be a path in T). Hence it suffices to determine ℓ', and if $\ell' > |P|$, to find an element of R'.

Denote $C = (v_i, \ldots, v_k)$ the fundamental cycle for some $k \in \mathbb{N}$ written so that $e = v_1 v_k$. When computing ℓ', we can restrict to paths in T' of the form

$$(\ldots, v_s, v_{s-1}, \ldots, v_1, v_k, v_{k-1}, \ldots, v_t, \ldots) \tag{4}$$

for $1 \leq s < t \leq k$, where v_s is the first and v_t is the last common vertex, respectively, of the path and C. For each i, let T_i be the connected component of v_i in $T - E(C)$, i.e., T_i is the part of T that is reachable from v_i without using the edges of C. Denote $\ell(T_i)$ the length of a longest path in T_i that starts at v_i and denote $c_i := \ell(T_i) + i - 1$ and $a_i := \ell(T_i) + k - i$. Then a longest path entering C at v_s and leaving it at v_t, as in (4), has length exactly $c_s + a_t$. Hence we have to determine a pair (s,t) such that $c_s + a_t$ is maximum (this maximum value is ℓ'); we call such a pair an *optimal pair*. If the so determined value ℓ' is not greater than $|P|$, then nothing further has to be done (the edge e cannot give an improvement). Otherwise, having constructed our optimal pair (s,t), we pick an arbitrary edge (e.g., uniformly at random) from $\{v_i v_{i+1} : s \leq i < t\}$, which are the edges between v_s and v_t on C. We show that the following algorithm computes the value ℓ' and an optimal pair in $\mathcal{O}(n)$.

```
1  compute $c_1, \ldots, c_{k-1}$ and $a_2, a_k$ using DFS
2  $M := 0; L := 0$
3  for $i = 1, \ldots, k-1$ do
4      if $c_i > M$ then
5          $M := c_i$
6          $s := i$
7      if $M + a_{i+1} > L$ then
8          $L := M + a_{i+1}$
9          $t := i + 1$
10 return $(s,t)$
```

The total of computations in Line 1 can be done by DFS in $\mathcal{O}(n)$, and the loop in $\mathcal{O}(k) \leq \mathcal{O}(n)$. We prove that the final (s,t) is optimal. For fixed t, the best possible length $c_s + c_t$ is obtained if t is combined with an $s < t$ where $c_s \geq c_j$ for all $j < t$. In the algorithm, for each t (when $t = i + 1$ in the loop) we combine a_t with the maximum $\max_{j<t} c_j$ (stored in the variable M). Thus, when the algorithm terminates, $L = \ell'$ and $c_s + c_t = \ell'$.

Corollary 1. *Our streaming algorithm (with the two phases as in Algorithm 3 and Algorithm 4) can be implemented with a per-edge processing time of $\mathcal{O}(n)$.*

We turn to the memory requirement. Denote by b the amount of RAM required to store one vertex or one pointer (e.g., $b = 32bit$ or $b = 64bit$) and call $n \cdot b$ one *unit*.

Theorem 3. *Our streaming algorithm (with the two phases as in Algorithm 3 and Algorithm 4) conducts at most $2q_1 + q_2$ passes. Moreover, the algorithm can be implemented such that the RAM requirement is at most $(\max\{4\tau, 2\tau+4\} \cdot n + c) \cdot b$ with a constant c.*

The proof can be found in [16 SPP].

An **experimental study** was conducted on randomly generated instances with different structure, including ones created with the generator for hyperbolic geometric random graphs [18 SPP]. Different variants of our streaming algorithm are compared with four RAM algorithms: Warnsdorf and Pohl-Warnsdorf (two related classical heuristics [23,24]), Pongrácz (a recently published heuristic [25]), and a simple randomized DFS. Experiments show that although we never do more than 11 passes, results delivered by our algorithm are competitive. We deliver at least 71% of the best result delivered by any of the tested RAM algorithms, with the exception of preferential attachment graphs. By considering low percentiles, we observe a similar quality without any restriction on the graph class. This is a good result also in absolute terms, since we observe that for each graph class and set of parameters, there is one algorithm that on average gives a path of length $0.84 \cdot n$, i.e., 84% of a Hamilton path. On some graph classes, we outperform any of the tested RAM algorithms, which makes our algorithm interesting even outside of the streaming setting.

4 An One Pass Streaming Algorithm for Computing the Euler Tour in Graphs

Large genome sequences (contigs) can be computed in de novo genome assembly with so-called de Bruijn graphs on k-mers ([3,22]). Such graphs are directed. For very large graphs, the computation of an Euler tour cannot be done with known RAM-based algorithms and techniques like semi-streaming or external memory algorithms are sought. In this chapter, we present a survey on our optimal one-pass streaming algorithm for computing an Euler tour in an undirected graph. Our algorithm might be helpful to design a semi-streaming algorithm to compute Euler tours in a directed graph, which is an open problem.

Let G be a graph on n nodes and m edges given in the form of a data stream. We study the problem of finding an Euler tour in G. We present a survey on the first one-pass streaming algorithm computing an Euler tour of G in the form of an edge successor function with only $\mathcal{O}(n \log(n))$ RAM based on our paper [13 SPP]. The memory requirement is optimal for this setting according to Sun and Woodruff [28].

4.1 The W-Streaming Model and a Lower Bound

The *W-streaming model* was introduced by Demetrescu et al. [7]. It is a relaxation of the classical streaming model. At each pass, an output stream is written, which becomes the input stream of the next pass. For an Euler tour the successor of each edge in the tour is uniquely defined by its successor function, say δ. Then the output stream has the following form, where the edges are unordered.

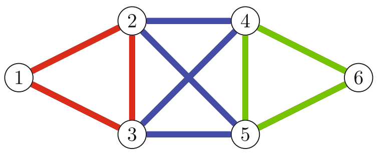
...	e	$\delta(e)$	$\delta(\delta(e))$	...

Finding an Euler tour in trees in W-streaming has been studied in multiple papers (e.g., [6]), but the general Euler tour problem has hardly been considered in a streaming model. There are some general results for transferring PRAM algorithms to the W-streaming model. In general, lower bounds for the complexity of streaming algorithms are hard to prove. Interestingly, Sun and Woodruff [28] showed that even a one-pass streaming algorithm for verifying whether a graph is Eulerian needs $\Omega(n\log(n))$ RAM, and this amount of RAM is also required for a one pass streaming algorithm for finding an Euler tour.

4.2 The Problem of Cycle Merging

The Euler tour problem in the RAM model can be easily solved by computing edge-disjoint cycles and merging them. We will see, why this is a problem with limited RAM. A *cycle* is a closed walk on the edges of G such that every node is visited at most once. The following result is well-known in graph theory.

Theorem 4. *If a graph with m edges contains an Euler Tour, it can be decomposed into at most $\frac{m}{3}$ pairwise edge-disjoint cycles.*



In fact, this can be accomplished in one pass.

Theorem 5. *During the pass, the edges from the input-stream can be ordered in form of a sequence of edge-disjoint cycles.*

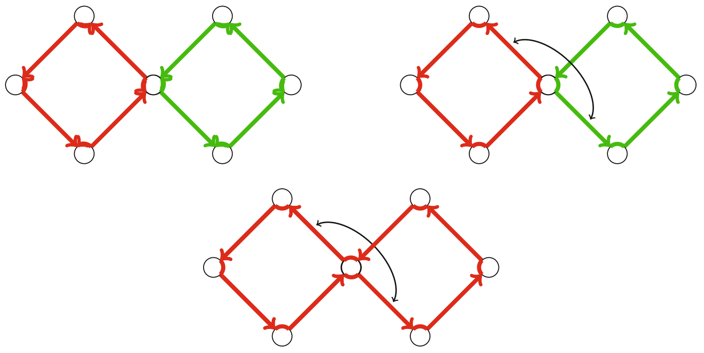
Proof. 1. Start with $T := \emptyset$
2. While T is cycle-free, add edges from the input stream to T
3. When a cycle occurs in T, store it and delete all its edges from T. Go to Step 2.

At every time, T contains at most n edges.

If $T \neq \emptyset$ at the end, there are some nodes of odd degree, thus G does not contain an Euler Tour.

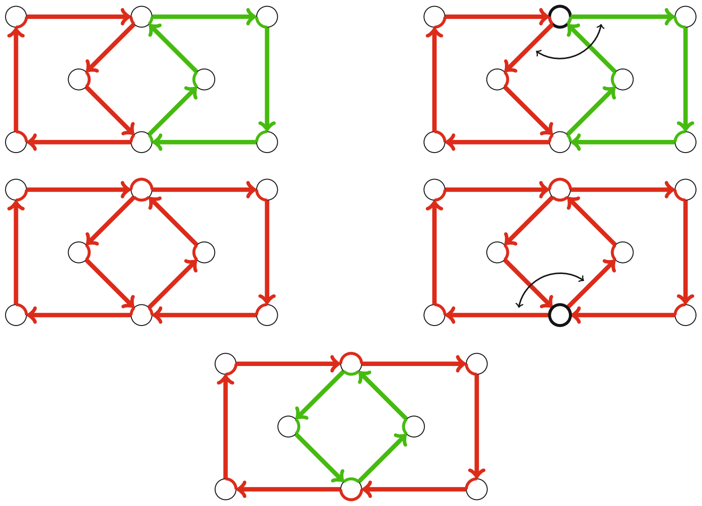
Obviously and unfortunately, we cannot store all the cycles in the semi-streaming model. The challenge is to merge cycles, when they are appearing with respect to the memory limitation of $\mathcal{O}(nlog(n))$. We will use the notion of tours or subtours for cycles, too.

The merging of two tours at one node is easy. We just flip edges in canonical way and get the new tour:



Similarly, one can merge several tours at one common node.

The problematic case is the simultaneous merging at two nodes. There is an example.



Unfortunately, the result of this merging is two tours, and the merging failed. A problem only occurs if the cycle shares more than one node with an already existing tour. In this case, we have to make sure that edge-swapping is performed at exactly one of these nodes. Every node belongs to at most one tour at a time, thus all nodes of a tour can get the same label.

4.3 The W-Streaming Algorithm and Its Analysis

We proceed to the pseudo-code statement of our streaming algorithm.

```
Algorithm 5: EULER-TOUR
input  : Undirected graph $G = (V,E)$, edge by edge on a stream $S$
output: Euler tour for $G$, i.e. a successor function $\delta^*$, if there is one
1  $c := 0$;  $F := \emptyset$;  $E_{\text{int}} := \emptyset$;  for every $v \in V$: $s(v) := 0, t(v) := 0$
2  for every edge $e$ on $S$ do
3      $E_{\text{int}} := E_{\text{int}} \cup \{e\}$
4      if $G_{\text{int}} = (V, E_{\text{int}})$ contains a cycle $C$ then
5          node MERGE-CYCLE ($C$)
6  if $E_{\text{int}} = \emptyset$ then
7      ERROR: At least one node with odd degree exists
8  if there exist $u,v$ with $t(u) \neq t(v) \neq 0$ then
9      ERROR: Graph is not connected
10 WRITE-F
```

```
Procedure Merge-Cycle
input  : Ordered directed cycle $C = (v_1, \dots, v_k)$ of length $k$
1  NEW-NODES
2  CHOOSE-NODES
3  WRITE
4  MERGE
5  UPDATE
6  for every edge $e \in C$ do
7      delete $e$ from $E_{\text{int}}$
```

The output stream is a successor function, i.e. e_1, $\delta(e_1)$, e_2, $\delta(e_2)$, … For $a,b,c \in V$ with (a,b); $(b,c) \in \vec{E}$ the triple (a,b,c) represents the successor function $(a,b) \to \delta((a,b)) = (b,c)$. So, edge (b,c) is the successor of edge (a,b). The output stream is *not* necessarily an ordered trail!

The main result is the following theorem [13 SPP].

Theorem 6 (Glazik, Schiemann, Srivastav, 2017). *There exists an one-pass W-Streaming algorithm with* $\mathrm{O}(n \log n)$ *RAM that outputs an Euler tour on the input graph G (if G contains an Euler tour).*

We sketch the proof. Let δ be a successor function. Equivalence classes: $e \in E$: $[e]_\delta = \{f \in E;\ \underbrace{e \equiv_\delta f}_{\exists k: \delta^k(e) = f}\ \}$ We identify the successor function with equivalence classes on $\vec{E}$.

Lemma 1 (Algebraic Representation [13 SPP], Lemma 1). *Let δ be a bijective successor function on a directed graph $\vec{G} = (V, \vec{E})$. Then $\equiv_\delta$ is an equivalence relation on $\vec{E}$.*

Lemma 2. *Let $\vec{G} = (V, \vec{E})$ be a directed graph with bijective successor function δ and the related equivalence relation $\equiv_\delta$. Then we have:*

(i) Let $e \in \vec{E}$ and $k_1, k_2 \in \mathbb{N}$ with $k_1 \neq k_2$ and $\delta^{k_1}(e) = \delta^{k_2}(e)$. Then $|k_1 - k_2| \geq |[e]_\delta|$.
(ii) For any $e \in \vec{E}$ we have $\delta^{|[e]_\delta|}(e) = e$.

Proof. (i): $F_s : \vec{E} \to \vec{E}$, $s \in \mathbb{N}$, $F_s(e') = \delta^{s(k_1-k_2)}(e')$.

- $\delta^{k_2}(e)$ fixpoint of F_s.
- $M := \{\delta^\ell(e); k_2 \leq \ell < k_1\}$, $|M| \leq k_1 - k_2$
- $[e]_\delta \subseteq M$ by fixpoint property of F_s

The assumption $k_1 - k_2 < |[e]_\delta|$ implies $|M| < |[e]_\delta| \leq |M| \to$ *contradiction*

(ii): $r := |[e]_\delta|$. Lets assume for a moment that $\delta^r(e_0) \neq e_0$ for some e_0.

- $M := \{\delta^\ell(e_0); 1 \leq \ell \leq r\} \subseteq [e_0]_\delta$

Case 1: $e_0 \in M$. Then

$$\delta^0(e_0) = e_0 = \delta^\ell(e_0) \text{ for some } \ell < r.$$

By (i): $\ell - 0 \geq r \to$ *contradiction*

Case 2: $e \notin M$. Then $|M| < |[e]_\delta|$. By the pigeonhole principle, there exist $1 \leq k_1, k_2 \leq |[e]_\delta|$ with $\delta^{k_1}(e) = \delta^{k_2}(e)$ in contradiction to (i).

Further, a structured theorem is needed. For an edge $e = (v, w)$ let $e_{(1)} := v$, $e_{(2)} := w$.

Theorem 7 (Successor function generates Euler tour [13 SPP], Theorem 3). *Let $= \vec{G}(V, E)$ be a directed graph with bijective successor function δ such that $e \equiv_\delta e'$ for all $e, e' \in \vec{E}$. Then δ is the successor function of an Euler tour for G.*

Let δ^0 be the successor function of an edge disjoint cycle decomposition of G. The algorithm computes a sequence of successor functions $\delta_0^* = \delta^0, \delta_1^*, \ldots, \delta_N^* := \delta^*$

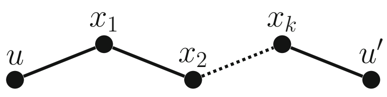
Theorem 8. *If G is Eulerian, δ^* determines an Euler tour on G.*

The following lemma is the backbone of the proof and requires substantial work.

Lemma 3 ([13 SPP], Lemma 9). *Let $k \in \{0, \ldots, N\}$. Then, δ_k^* is bijective and for any $(u, v), (u', v') \in R^*(E)$, we have*

(i) If $(u, v), (u', v')$ are processed edges, then $(u, v) \equiv_{\delta_k^*} (u', v') \Leftrightarrow t_k(u) = t_k(u')$.
(ii) If (u, v) is a processed edge, then $t_k(u) = t_k(v)$.
(iii) If $t_k(u) = 0$, then $(u, v) \equiv_{\delta_k^*} (u', v') \Leftrightarrow (u, v) \equiv_\delta (u', v')$.

Proof (Proof of Theorem 8). We show: If δ^* is bijective and $e \equiv_{\delta^*} e'$ for all e, e', then δ^* is an Euler tour by Theorem 3. Then, by Lemma 3, $\delta^* = \delta_N^*$ is bijective. For the second property let $e, e' \in E$, $e = (u, v)$ and $e' = (u', v')$. We show $e \equiv_{\delta^*} e'$. Now, there exists an u–u'–path P in G because G is Eulerian. Let $P = u\, x_1\, x_2 \ldots x_k\, u'$ such a path.



By Lemma 3 (ii), label t_N propagates through P:

$$\begin{aligned} & t_N(u) = t_N(x_1) = t_N(x_2) = \cdots = t_N(x_k) = t_N(u') \\ \Rightarrow\quad & t_N(u) = t_N(u') \\ \underset{\text{Lemma 3(i)}}{\Rightarrow}\quad & e \equiv_{\delta_N^*} e' \end{aligned}$$

In future work we may investigate other routing problems and applications for streaming algorithms using Euler tours.

References

1. Brown, C.T., Howe, A., Zhang, Q., Pyrkosz, A.B., Brom, T.H.: A reference-free algorithm for computational normalization of shotgun sequencing data, pp. 1–18. ArXiv e-prints (2012). https://arxiv.org/abs/1203.4802
2. Bulterman, R.W., van der Sommen, F.W., Zwaan, G., Verhoeff, T., van Gasteren, A.J.M., Feijen, W.H.J.: On computing a longest path in a tree. Inf. Process. Lett. **81**(2), 93–96 (2002). https://doi.org/10.1016/S0020-0190(01)00198-3
3. Compeau Phillip, E.C., Pevzner Pavel, A., Tesler, G.: How to apply de Bruijn graphs to genome assembly. Nat. Biotechnol. **29**(11), 987–991 (2011). https://doi.org/10.1038/nbt.2023
4. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithms **55**(1), 58–75 (2005). https://doi.org/10.1016/j.jalgor.2003.12.001
5. Dementiev, R., Kettner, L., Sanders, P.: STXXL: standard template library for XXL data sets. Softw. Pract. Exp. **38**(6), 589–637 (2008). https://doi.org/10.1002/spe.844
6. Demetrescu, C., Escoffier, B., Moruz, G., Ribichini, A.: Adapting parallel algorithms to the w-stream model, with applications to graph problems. Theor. Comput. Sci. **411**(44–46), 3994–4004 (2010). https://doi.org/10.1016/j.tcs.2010.08.030
7. Demetrescu, C., Finocchi, I., Ribichini, A.: Trading off space for passes in graph streaming problems. ACM Trans. Algorithms **6**(1), 6:1-6:17 (2009). https://doi.org/10.1145/1644015.1644021
8. Deorowicz, S., Debudaj-Grabysz, A., Grabowski, S.: Disk-based k-mer counting on a PC. BMC Bioinform. **14**, 160 (2013). https://doi.org/10.1186/1471-2105-14-160
9. Deorowicz, S., Kokot, M., Grabowski, S., Debudaj-Grabysz, A.: KMC 2: fast and resource-frugal k-mer counting. Bioinformatics **31**(10), 1569–1576 (2015). https://doi.org/10.1093/bioinformatics/btv022
10. Dietzfelbinger, M., Hagerup, T., Katajainen, J., Penttonen, M.: A reliable randomized algorithm for the closest-pair problem. J. Algorithms **25**(1), 19–51 (1997). https://doi.org/10.1006/jagm.1997.0873
11. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: On graph problems in a semi-streaming model. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 531–543. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27836-8_46
12. Gao, T., et al.: Bloomfish: a highly scalable distributed k-mer counting framework. In: IEEE ICPADS 2017, pp. 170–179. IEEE Computer Society (2017). https://doi.org/10.1109/ICPADS.2017.00033

13 SPP. Glazik, C., Schiemann, J., Srivastav, A.: Finding Euler tours in one pass in the W-streaming model with O(n log(n)) RAM. CoRR abs/1710.04091 (2017). Theory of Computing Systems 2022, 23 p. Springer. https://doi.org/10.1007/s00224-022-10077-w

14. Handelsman, J., Rondon, M.R., Brady, S.F., Clardy, J., Goodman, R.M.: Molecular biological access to the chemistry of unknown soil microbes: a new frontier for natural products. Chem. Biol. **5**(10), R245-9 (1998). https://doi.org/10.1016/s1074-5521(98)90108-9
15. Kelley, D.R., Schatz, M.C., Salzberg, S.L.: Quake: quality-aware detection and correction of sequencing errors. Genome Biol. **11**(11), 1–13 (2010). https://doi.org/10.1186/gb-2010-11-11-r116

16 SPP. Kliemann, L., Schielke, C., Srivastav, A.: A streaming algorithm for the undirected longest path problem. In: ESA, pp. 56:1–56:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.ESA.2016.56

17. Kokot, M., Dlugosz, M., Deorowicz, S.: KMC 3: counting and manipulating k-mer statistics. Bioinformatics **33**(17), 2759–2761 (2017). https://doi.org/10.1093/bioinformatics/btx304

18 SPP. von Looz, M., Staudt, C.L., Meyerhenke, H., Prutkin, R.: Fast generation of dynamic complex networks with underlying hyperbolic geometry. CoRR abs/1501.03545 (2015). http://arxiv.org/abs/1501.03545

19. Marçais, G., Kingsford, C.: A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. Bioinformatics **27**(6), 764–770 (2011). https://doi.org/10.1093/bioinformatics/btr011
20. Nehls, C.: Effizientes sortier-basiertes Zählen von k-meren im externen Speicher. Mathematisches Seminar, Universität zu Kiel, Masterarbeit (2018)
21. Pandey, P., Bender, M.A., Johnson, R., Patro, R.: Squeakr: an exact and approximate k-mer counting system. Bioinformatics **34**(4), 568–575 (2018). https://doi.org/10.1093/bioinformatics/btx636
22. Pevzner, P.A., Tang, H., Waterman, M.S.: A new approach to fragment assembly in DNA sequencing. In: RECOMB, pp. 256–267. ACM (2001). https://doi.org/10.1145/369133.369230
23. Pohl, I.: A method for finding Hamilton paths and knight's tours. Commun. ACM **10**(7), 446–449 (1967). https://doi.org/10.1145/363427.363463
24. Pohl, I., Stockmeyer, L.: Pohl-Warnsdorf - revisited. In: Proceedings of the ISC 2004 (2004). https://users.soe.ucsc.edu/~pohl/Papers/Pohl_Stockmeyer_full.pdf
25. Pongrácz, L.L.: A greedy approximation algorithm for the longest path problem in undirected graphs. CoRR abs/1209.2503 (2012). withdrawn
26. Rizk, G., Lavenier, D., Chikhi, R.: DSK: k-mer counting with very low memory usage. Bioinformatics **29**(5), 652–653 (2013). https://doi.org/10.1093/bioinformatics/btt020
27. Roy, R.S., Bhattacharya, D., Schliep, A.: Turtle: identifying frequent k-mers with cache-efficient algorithms. Bioinformatics **30**(14), 1950–1957 (2014). https://doi.org/10.1093/bioinformatics/btu132
28. Sun, X., Woodruff, D.P.: Tight bounds for graph problems in insertion streams. In: APPROX-RANDOM, pp. 435–448. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.APPROX-RANDOM.2015.435

29 SPP. Wedemeyer, A., Kliemann, L., Srivastav, A., Schielke, C., Reusch, T.B., Rosenstiel, P.: An improved filtering algorithm for big read datasets and its application to single-cell assembly. BMC Bioinform. **18**(1), 324 (2017). https://doi.org/10.1186/s12859-017-1724-7

30. Wölfel, P.: Über die Komplexität der Multiplikation in eingeschränkten Branchingprogrammmodellen. Ph.D. thesis, Technical University of Dortmund, Germany (2003). http://hdl.handle.net/2003/2539