Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Algorithms for dynamic assembly of nanopore reads

Diploma Thesis

2024

Bc. Jana Černíková

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Algorithms for dynamic assembly of nanopore reads

Diploma Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	Department of Applied Informatics
Supervisor:	doc. Mgr. Tomáš Vinař, PhD.

Bratislava, 2024
Bc. Jana Černíková

55568555

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta:	Bc. Jana Černíková
Študijný program:	informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor:	informatika
Typ záverečnej práce:	diplomová
Jazyk záverečnej práce:	anglický
Sekundárny jazyk:	slovenský

Názov: Algorithms for dynamic assembly of nanopore reads
Algoritmy pre dynamické zostavovanie nanopórových čítaní

Anotácia: Jednou z výhod nanopórového sekvenovanie je, že sekvenačné čítania sú produkované postupne počas celého sekvenačného behu a na základe ich priebežnej analýzy je možné prispôsobiť vlastnosti sekvenačného behu. Cieľom práce je preskúmať možnosti vytvorenia algoritmov, ktoré by dokázali skladať a dynamicky upravovať zostavenie sekvencií na základe sekvenačných čítaní tak, ako pribúdajú v reálnom čase pri nanopórovom sekvenovaní.

Vedúci:	doc. Mgr. Tomáš Vinař, PhD.
Katedra:	FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry:	doc. RNDr. Tatiana Jajcayová, PhD.
Dátum zadania:	15.12.2022

Dátum schválenia: 28.04.2023

prof. RNDr. Rastislav Kráľovič, PhD.
garant študijného programu

.... študent

.... vedúci práce

55568555

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Jana Černíková
Study programme:	Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Algorithms for dynamic assembly of nanopore reads

Annotation: One of the great advantages of nanopore sequencing is that the sequencing reads are produced continuously and it is possible to adjust parameters of the sequencing during the sequencing run. The goal of the thesis is to explore possibilities of designing algorithms that can perform and dynamically update partial sequence assembly from the sequencing reads in real time as they are produced by the nanopore sequencer.

Supervisor:	doc. Mgr. Tomáš Vinař, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	doc. RNDr. Tatiana Jajcayová, PhD.
Assigned:	15.12.2022
Approved:	28.04.2023

prof. RNDr. Rastislav Kráľovič, PhD.
Guarantor of Study Programme

.... Student

.... Supervisor

Acknowledgments: I would like to thank my supervisor doc. Mgr. Tomáš Vinař, PhD. and doc. Mgr. Broňa Brejová, PhD. for their patience, a lot of helpful advice and guidance. I am also thankful to the people from M-25 lab for their emotional support.

Abstrakt

Jednou z výhod nanopórového sekvenovania je, že sekvenované čítania sú dostupné v reálnom čase a je možné upravovať parametre sekvenovania počas behu na základe ich priebežnej analýzy. Preskúmali sme možnosti vytvorenia algoritmov ktoré by mohli byť použité na dynamické zostavovanie sekvencií zo sekvenovaných čítaní v reálnom čase a implementovali sme nástroj, ktorý uchováva reprezentatívnu vzorku z čítaní, dynamicky zostavuje čítania počas sekvenačného behu a zobrazuje výsledky z dostupných dát používateľovi v reálnom čase.

Kľúčové slová: zostavovanie genómov počas sekvenačného behu, nanopórové sekvenovanie, monitorovanie sekvenovania

Abstract

One of the advantages of nanopore sequencing is that the sequencing reads can be accessed as they are produced, and it is possible to adjust the parameters of the sequencing during the sequencing run. We have explored the possibilities of designing algorithms that can dynamically perform the sequence assembly from the sequencing reads in real time as they are produced by the nanopore sequencer and implemented a pipeline that maintains a representative sample of the data and dynamically assembles the reads to provide the information to the user in real time.

Keywords: real-time genome assembly, nanopore sequencing run, sequencing run monitoring

Contents

List of Figures

Introduction

Nanopore sequencing is a rapidly developing third-generation sequencing technology. One of the advantages of nanopore sequencing is that the sequencers commonly used for nanopore sequencing can output the sequenced data in real time as they are produced. This suggests a possibility of performing genome assembly dynamically from the reads produced in real-time during the sequencing run.

This thesis aimed to explore such options and design an algorithm that could be used for the dynamic assembly of nanopore reads. We have designed and implemented a real-time pipeline that maintains a representative sample from the available data and assembles the reads, providing the user with real-time results.

The first chapter describes the principles of genome assembly, the Miniasm assembler and the reasoning behind our approach. In the second chapter, we focus on the approaches for selecting a representative sample of the reads needed for the assembly. The third chapter describes the real-time pipeline we have implemented, and the fourth chapter discusses the results. The last chapter contains some additional notes about other options and future work.

Chapter 1

Assembly of nanopore sequencing reads

This chapter describes the methods and principles of genome assembly and introduces some commonly used algorithms and heuristics for genome assembly from nanopore data.

1.1 Problem definition

1.1.1 Sequencing and Oxford Nanopore Technologies

Sequencing is the process of determining the order of nucleotides in a DNA (or RNA) molecule. There are multiple steps between the preparation of the sequencing sample in the laboratory and the string of bases produced in the computer.

Sequencing technologies development over time is commonly divided into three generations. The development started with the oldest approaches requiring a lot of laboratory work, such as Sanger sequencing in the first generation; in the second generation, the sequencers producing short reads with high quality came (from which the Illumina sequencers are probably the most known). The third generation is the generation of nanopore technologies, mostly represented by the Oxford Nanopore technologies and Pacific Biosciences sequencing devices. [6]

One of the sequencing devices provided by Oxford Nanopore Technologies (ONT) [18] is MinION. Its main advantage is that it can produce sequenced reads in real-time during the sequencing run, which allows us to analyse the data as the run progresses. Such analysis can be helpful for deciding when to terminate the sequencing run.

As a final result of the nanopore sequencing process, short sequences of bases A, C, G or T are produced from the molecules traversing the nanopores. The short sequences are called *reads*.

The raw results of the nanopore sequencing process are the values of electric current measured when a molecule traverses the nanopore. A set of reads along with some additional information is produced from the raw data in the *basecalling* process. The additional information includes the quality of the bases, a name (ID) for each read, and the time when it was produced by the sequencer. Such set of reads is then processed bioinformatically to get the desired results.

One of the typical applications of sequencing is the use of the technology to determine the genome sequence of an organism. The bioinformatics term for determining the genome sequence from the reads is *genome assembly*. If the complete genome sequence is unknown and we have no other supporting information except the reads, we call it *de novo* genome assembly.

In our work, we want to analyse and assemble the data from the sequencing run in real time to determine when there is enough data for *de novo* genome assembly.

1.1.2 Nanopore sequencing reads

Even though the development of sequencing technologies (especially nanopore) has been progressing quickly in recent years, the maximal length of a read is limited. The exact length of the reads and frequency of sequencing errors depends on the sequencing technology. The third-generation sequencing technologies can produce reads of length up to 4 Mbp (the longest read reported by a MinION user to date). [18] However, the length of reads varies during the sequencing run, and the reads usually do not span the entire genome regions (i.e., we do not have a single read covering the whole chromosome of a yeast). The read lengths depend on multiple factors that may be hard to predict. For example, the distribution of read lengths of the reads from one nanopore sequencing run of *Saprochaete ingens* yeast (which we will use for testing later) is shown in Figure 1.1. As we can see, most of the reads have lengths less than 0.075 Mbp (75000 base pairs), which is far from the theoretical maximum. The reference genome of this yeast is 21.2 Mbp long. [7]

The length of the reads compared to the length of the genome is an important factor when considering the genome assembly problem. The assembly result also highly depends on the coverage of the genome by the sequencing reads, which we will discuss later.

1.1.3 Genome assembly as a problem in bioinformatics

Various definitions of genome assembly can be found in bioinformatics research papers. From the biological perspective, the goal is to determine the whole sequence of the genome of the sequenced organism.

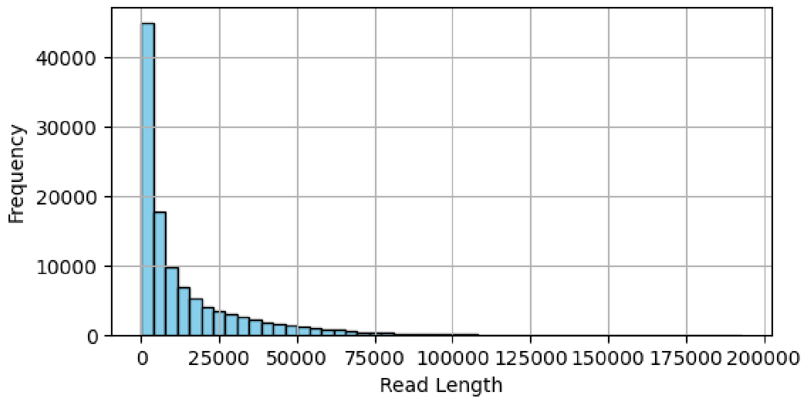


Figure 1.1: Read length distribution of reads from Saprochaete ingens sequencing run.

Because of the specific biological aspects that have to be considered, there is no straightforward way to define the problem of genome assembly as a problem in bioinformatics considering all the specifics of the task. Therefore, some extrapolation is necessary to define genome assembly as a theoretical problem.
The basic definitions we will need are the representation of nucleotides and reads:
Let $\Sigma = \{A, C, G, T\}$ be the alphabet of *nucleotides*. A DNA *sequence* or *read* is a string $s = a_1a_2...a_n$ over alphabet Σ. The *length of the sequence* or *length of the read* is the length of the string s.

1.1.3.1 Shortest common superstring

The first and probably the most straightforward way to define the genome assembly problem is the problem of the *shortest common superstring.*

Definition:(shortest common superstring) Given a set of reads S, find the shortest possible string g which contains all the reads from S as contiguous substrings.

Note that the minimality of the superstring is required. Otherwise a simple solution for any S would be a concatenation of all the reads from S one after the other, which is usually not the result the biologist is looking for.

However, this definition does not fully consider the biological aspects of the problem. For example, it is quite common that there are regions of the genome that consist of repetitive sequences (repeats). If there is no read spanning the whole repeat sequence, this definition will give us assembly without that part of the genome. Also, the definition does not consider sequencing errors, which are always present in the reads. They cause that the assembly defined this way can not be created correctly (without very precise error correction). Another problem is that the genomes are typically organized into chromosomes, so the result should not be a single sequence but a set of sequences for chromosomes.

Also, the problem of the shortest common superstring, formulated this way, is NP-hard. Some approximation algorithms exist for this task, but the properties that can be guaranteed by them are not sufficient for biological applications (we may, for example, find a string that is two times longer than the genome, and such information is not useful for the biologist). [5]

Since the straightforward formulation of the problem is NP-hard and the task of defining the problem is complex, multiple heuristic approaches were developed.

1.1.3.2 Overlap-layout-consensus

The most common and widely used approach to genome assembly is the overlap-layout-consensus (OLC) heuristic. As the name of this procedure suggests, it comprises three steps: overlap, layout, and consensus.

Overlap The aim of the overlap phase is to find overlaps between the given reads and form longer sequences (contigs) from them.[5] The overlaps can be found by aligning the set of reads onto themselves using a similarity search (alignment) tool such as Minimap (or Minimap2). The contig-building strategies differ in various implementations. Some of them will be discussed in more detail in the next sections of this chapter.

Layout In the layout phase, we are trying to determine the relative orientation of and position (distances) between the contigs. In this phase, we should address problems such as sequencing gaps or repetitive sequences, which were not resolved in the overlap phase. Sequencing gaps are the result of the sequencing process in which we commonly have no guarantee that the reads we get cover the whole genome – there may be some parts of the genome that are not covered by the reads we have. These are then replaced by the sequences of Ns, meaning that the nucleotide at the position is unknown to us, although we know there should be one.

Other issues can be caused by repetitions or ambiguous information from the overlap phase. Pair-end reads may be helpful in the process of resolving these issues – they are useful for connecting the contigs or determining the length of repetitions. Also, they are a lead for determining the relative orientation of the contigs. The result of the layout phase is a set of supercontigs. [5]

Consensus The last step of the genome assembly process is consensus. In the previous steps, we have built the supercontigs, which give us the information how all the reads can be assembled – but we focus on them mostly as some parts that have to be put together, not on a single-base level. In the consensus step, we want to polish the result, taking into account that we have more reads covering one position, and we can use this information. The mapped reads for a position may differ in a particular base, which may be caused either by a sequencing error or a polymorphism. However, knowing the possibilities for a particular base, we can choose one that is the most probable by some kind of consensus rule. [5]

1.1.3.3 De Bruijn graphs

Another commonly used approach to genome assembly are de Bruijn graphs. A de Bruin graph consists of vertices representing k-mers (short parts of the sequences of length k) and oriented edges representing the overlaps between the k-mers in the vertices. The k-mers are constructed from the reads by taking substrings of length k for each read of length n, starting at positions $1, 2, ..., n - k$ in the read.

In a simplified scenario, considering a graph for a single contig (that has only one component and meets the conditions for the existence of the Eulerian trail in a directed graph), the Eulerian trail in the de Bruin graph represents the genome assembly. According to known theoretical results, the problem of finding the Eulerian trail in a directed graph (if the trail exists) is solvable in $O(n + m)$ time, where n is the number of vertices and m is the number of edges.

With this approach, some information from the data is discarded, as we are splitting the sequence into fixed-length k-mers. However, it is a properly defined informatics problem that is also easily solvable.

The problem with this approach is that the de Bruijn graph constructed from the reads may not meet the properties for the existence of the Eulerian trail in a directed graph (or its component). Also, the results may contradict the read sequences we have, since we are discarding some information in the graph construction process. Furthermore, there is typically a number of contigs, and depending on the size of the k-mers (and multiple other factors), the components of the graph may not correspond to the contigs in the genome. Instead, we can have more components for a single contig, or a component that connects two contigs together (which can be hard to resolve).

The existence of the trail depends on the start and end vertex we are considering (unless there is an Eulerian cycle). The Eulerian trail from vertex u to v exists iff after we add the edge from v to u to the graph, then the graph is *strongly connected* (every vertex of the graph is *reachable* [1] from any other vertex), and the indegrees and

[1]Vertex v is reachable from w, iff a directed path from w to v exists

the outdegrees are equal for each vertex. [5]

If there is exactly one pair of start and end vertices for which the trail exits, we have some solution. However, there can be multiple such pairs, which means there are multiple Eulerian trails in the graph, and we do not know which one to choose. It can also happen that there is no such pair of vertices for which the trail exists. Those issues can be, in some cases, resolved by heuristic approaches that preprocess the graph in some way after it is created from the reads, or they can be addressed directly in the algorithm that looks for the trail.

1.2 Minimap and Miniasm

Miniasm is one of the most used tools for genome assembly in the bioinformatics field. It is indeed directly linked with Minimap2, which is used for computing the read mappings (alignments) that are the input for Miniasm.

In the overlap-layout-consensus assembly approach, which is also the case of the combination of Minimap and Miniasm, finding read mappings or overlaps is typically the first and crucial step of the process. It is also the most time-consuming step in assemblers implementing the OLC approach.

Most of the practically used algorithms for read mapping are heuristics because the non-heuristic approaches are too slow and space-inefficient for common applications. One example of such algorithm is the Smith-Waterman dynamic programming algorithm, which performs local sequence alignment and has quadratic time and space complexity. This makes it, without applying additional heuristics, unusable for large-scale problems like genome assembly.

In the next section, we will briefly introduce the evolution of alignment heuristics up to the time when Minimap2 was created. Then, we will discuss the algorithms of Minimap2 and Miniasm and how they utilize the ideas from the described similarity search tools.

1.2.1 Before Minimap

Before 2015, when Minimap was released, multiple similarity search tools were developed. We will briefly introduce some of them: BLAST, BLAT, DALIGNer, and MHAP.

BLAST BLAST and BLAT are some of the most popular sequence similarity search tools even these days. The core idea of the tools is that several high-scoring segments are found, and then a dynamic programming approach similar to the Smith-Waterman algorithm is performed on them to obtain an optimal alignment. [1] Basically, they hash k-mers at positions $1, w+1, 2w+1, ..$ of the target sequence using a k-mer hash function $h : \Sigma^k \rightarrow Z$ and store them into a hash table. Then, they use the same hash function on every k-mer in the query sequence and look for potential matches. [2] [15]

There are several types of BLAST (Basic Local Alignment Search Tool). It can compare all combinations of nucleotide or protein queries with nucleotide or protein databases. BLAT (BLAST-Like Alignment Tool) and BLAST are similar. The main difference is that while BLAST builds an index of the query sequence and then scans linearly through the database, BLAT builds an index of the database first and then scans linearly through the query. There is also a difference in how they can handle the hits that are close to each other (BLAT can handle more at once) and how they handle the intron sequences (there is a special code for introns in BLAT). Another difference is that BLAT needs an exact (or nearly exact) match to find a hit, while BLAST can also find more distant matches. In terms of speed, BLAT is faster. [9]

DALIGNer DALIGNer [17] builds lists of k-mers and uses a highly cache-efficient sorting and merging algorithm instead of a hash table.

It is designed with respect to the high error rates in the long reads. The DALIGNer algorithm uses highly optimized threaded radix sorts in order to obtain high speed as opposed to other commonly used structures such as the BWT index, which is has various applications in bioinformatics. The main drawback of the BWT index (or Suffix Arrays) highlighted by the authors is the cache incoherence of the structures. Therefore, they decided to replace it with their own sorting approach.

The authors of the algorithm also analyse the possibilities of *de novo* reconstruction of a genome and its accuracy based on the technology used.
They state that using the PacBio RS II sequencer, even though the error rate is relatively high ($\epsilon = 12 - 15\%$ error), if
(a) the reads are long enough
(b) without repetitive genome elements confounding assembling the reads
(c) given enough coverage
(d) the set of reads produced by the sequencer is a nearly Poisson sampling of the underlying genome
(e) errors are distributed randomly within the reads
then it is, in principle, possible to reconstruct the genome *de novo* at any level of accuracy.

The authors highlight the importance of looking for all the local alignments and not for overlaps only (alignments for which the suffix of one read is mapped to the prefix of the other) since this gives us more information we may be interested in to achieve better assemblies, especially for solving repeats or handling chimeric reads.

MHAP MHAP (MinHash Alignment Process) [3] is an algorithm for overlapping noisy long reads. It uses probabilistic locality-sensitive hashing.

The MHAP algorithm was used as a part of Celera Assembler, designed for high-noise single-molecule sequencing (PacBio RSII, Oxford Nanopore MinION). It enabled the researchers to assemble genomes of organisms such as Saccharomyces cerevisiae, Arabidopsis thaliana, and Drosophila melanogaster *de novo*, resulting in highly accurate and contiguous assemblies.

MHAP uses MinHash, a dimensionality reduction technique that was originally developed for determining the similarity of web pages. It allows a more compact representation of the sequencing reads. The principle of the MinHash technique is that it reduces a string into a small set of fingerprints called *sketches*. This approach was successfully applied in multiple fields – it can be used to determine the similarity of documents, images and sequences, or it can be used even for metagenomic clustering.

A sketch of a DNA sequence can be created from the k-mers in the sequence using a set of randomized hash functions. The functions are used to convert each of the k-mers into integer fingerprints. From those fingerprints, only the minimal one, called *min-mer*, is taken for each hash function. The set of these min-mers is the sketch of the sequence. Then, the Jaccard similarity of two sets of k-mers is estimated by computing the Hamming distance [2] of their sketches. The authors claim that the resulting estimate of Jaccard similarity is strongly correlated to the fraction of shared k-mers. Hence, the technique can be used for sequence similarity estimation. Moreover, as the sketches are small compared to the sequences, the technique is also computationally efficient.

1.2.2 Minimap

The author of Minimap [15], Heng Li, used the methods and principles he found advantageous in all the algorithms mentioned in the previous Section and developed his own algorithm for sequence similarity search – Minimap. For the genome assembly, the overlaps are the most important, but Minimap is designed not only as a read overlapper but also as a read-to-genome and genome-to-genome mapper, which extends its applications.

Minimap stores the k-mers in a hash table in an analogous way to BLAST or BLAT and MHAP. It also uses a sorting algorithm similar to the one in DALIGNer. The author adopted the idea of sketch from MHAP, but instead of using all the k-mers, he uses only *minimizers*, yielding to even more reduced representation. Minimizers are specially selected k-mers that satisfy a property that for two strings that have a significant exact match, at least one of the k-mers chosen from one will also be chosen

[2]number of elements at which they are different

from the other. [19] Minimizers can be generated by applying a special hash function to the k-mers and selecting the ones that have the minimal hashing value.

Minimap first computes the sets of minimizers. This set forms a *sketch* of the read. A set of minimizers for a string s is computed in close to $O(|s|)$ time in the average case. The next step is indexing. The minimizers of all target sequences are kept in a structure that serves as a hash table, with minimizers serving as keys. The values in the hash table are 64-bit integers; the sequence index, position of the minimizer and the strand are packed into the integer value. In fact, the 64-bit integers are not directly put in a hash table – instead, they are stored in an array. The array is sorted after the collection of all the minimizers and the hash table keeps only the intervals on the sorted array. This approach is more cache efficient and, therefore, faster than inserting the values directly into a hash table.

The hash table is then used in the mapping process. Given the hash table and the query sequence, the algorithm first collects the minimizer hits (minimizers for which there exists a matching hash value in the target sequence) and then clusters the hits that are collinear (mapped without gaps or mismatches) within a band of a given width. Such clustering is then used for identifying long collinear matches. The clustering procedure is inspired by Hough Transformation, a method commonly used in image processing for the detection of shapes such as lines, circles and ellipses.[20]

Minimap in its original version [13], with the latest release in 2015, was marked by its author as deprecated in December 2017 and was completely replaced by Minimap2 in all its applications.

1.2.3 Minimap2

Minimap2 [16] is a versatile alignment program for nucleotide sequences. It can be used as a read mapper, long-read overlapper or a full-genome aligner. The capabilities were extended and speed and memory usage were improved compared to Minimap.

Minimap2 uses a similar seed-chain-align procedure as Minimap. It collects minimizers of the reference sequences and stores them into a hash table where the hashes of the minimizers serve as keys and locations of minimizer copies are stored as values. Then, for each query sequence, Minimap takes minimizers of the query as *seeds* and finds sets of anchors (exact matches) that form the *chains*.

One of the key improvements made in Minimap2 compared to Minimap is the better accuracy of the chaining algorithm – the chaining score is calculated using dynamic programming and also a new heuristic was introduced to accelerate the chaining process. In the chaining process, Minimap2 introduces a new heuristic for finding optimal chaining scores and uses backtracking to avoid reusing anchors in multiple chains. Furthermore, the chains are split into two categories – primary and secondary, from which the former is important and considered in the next steps. The aim is to avoid significant overlaps caused by the repeats in the reference – ideally, each query segment should be mapped to only one place in the reference. However, the chains may have significant or complete overlaps due to the repetitions. The heuristic for the primary chain identification tries to find the primary chains that do not greatly overlap on the query, which reduces the number of sequences being mapped to multiple positions.

Another improvement is that number of minimizers is reduced using an approach that compresses homopolymers within sequences – the idea of homopolymer compression (HPC) is that the homopolymers in the sequences are contracted into a single base. For example, a string $s = GAATTTCCA$ will be contracted to $HPC(s) = GATCA$ and GAT will be the first 3-mer for HPC(s), instead of GAA for s. With this approach, they achieved higher sensitivity, using a smaller number of k-mers, which is a remarkable result.

Minimap2 can also produce base-level alignment and is able to align spliced sequences. For the base-level alignment, Minimap2 applies dynamic programming with two-piece affine gap scoring cost to extend from the ends of chains and to close regions between adjacent colinear matches (anchors) in the chains. Some additional heuristics were used for improvements in speed and accuracy. In the spliced alignment mode, Minimap2 distinguishes chaining gap cost for insertions to the reference and deletions from the reference.

Based on whether per-base alignment is requested or not, Minimap2 provides two different file formats of the alignments. The SAM format (or BAM, which is a binary variation of SAM) contains per-base alignment information. The PAF format is simpler and faster to compute – it contains only the basic information about the alignments: positions where the alignment starts and ends in each of the sequences, the length of the sequences, relative strand (orientation) of the sequences, number of matches and the length of the alignment block.

Both Minimap and Minimap2 are available as command-line tools. For the purpose of the genome assembly process, Minimap2 can be used for mapping a set reads of reads onto themselves. Such mapping is then used as the input for Miniasm which performs the assembly.

1.2.4 Miniasm

Miniasm [15] [12] is a de novo assembler built on the overlap-layout-consensus principle. However, it does not have the typical overlap-layout-consensus structure as described in Section 1.1.3.2, since it does not implement the consensus step. Miniasm takes read mappings produced by Minimap2 as an input and outputs the assembly graph in GFA format. Instead of the consensus step, Miniasm simply takes and concatenates pieces of reads into the final sequences called unitigs. Miniasm uses Minimap2's result from mapping a set of reads onto themselves as an input.

To understand the Miniasm assembly process further, we have to define the assembly graph. For the assembly graph definition, we also have to define the sequence overlap and assembly graph properties. The definitions are based on the work of Heng Li, the author of Minimap, Minimap2 and Miniasm. [15]

Definition (Sequence overlap): Let v, w be two DNA sequences. We say that v overlaps w if the suffix of v can be mapped to the prefix of w. We denote the overlap as $v \to w$ or (v, w).

Definition (Watson-Crick complete graph): Let V be a set of DNA sequences, let E be a set of overlaps between the sequences from V. Let $l : E \to R_+$ be the edge length function. Then graph $G = (V, E, l)$ without multi-edges is said to be *Watson-Crick complete* iff (i) $\forall v \in V : \overline{v} \in V$ and (ii)$\forall (v, w) \in E : (\overline{w}, \overline{v}) \in V$

Definition (containment-free graph): Let V be a set of DNA sequences, let E be a set of overlaps between the sequences from V. Let $l : E \to R_+$ be the edge length function. Then graph $G = (V, E, l)$ without multi-edges is said to be *containment-free* iff for any $v, w \in V$, the sequence v can not be mapped to a substring of w.

Definition (Assembly graph): Let V be a set of DNA sequences, let E be a set of overlaps between the sequences from V. Let $l : E \to R_+$ be the edge length function. Then graph $G = (V, E, l)$ is an assembly graph iff it is Watson-Crick complete and containment-free.

Miniasm algorithm operates in four steps: read filtering and trimming of the reads, assembly graph generation, graph cleaning, and unitig sequences generation.

The first filter is applied when reading the PAF file – only some of the mappings are stored. The mappings for which the whole sequence aligns to another with no overlap are marked as *contained* (see Figure 1.2) and they are discarded.

Next, the PAF mappings are filtered in two phases that mainly differ only in the way how strong the filter is (the first one is firm; in the second one, most of the reads are discarded). The filtering is guided by the read trimming process. The goal of the trimming is to remove artefacts such as untrimmed adapters or chimera that may be present in raw sequencing reads. The read-to-read mappings are examined and based on good mappings against other reads, per-base coverage is computed for each read using dynamic programming. [12] Then, bases outside the longest region of the read having coverage of three or more are trimmed. The trimmed mappings are then used in the next steps. If the trimmed region is too short, the mapping is discarded.

After the trimming step is done, each trimmed mapping is classified into one of the five groups: internal match, first read contained, second read contained, first to second read overlap or second to first read overlap (see Figure 1.2). Only the mappings from the *overlap* groups are added to the assembly graph. Also, for each pair of reads, only the longest overlap is taken, which ensures that no multi-edges will be created in the graph.

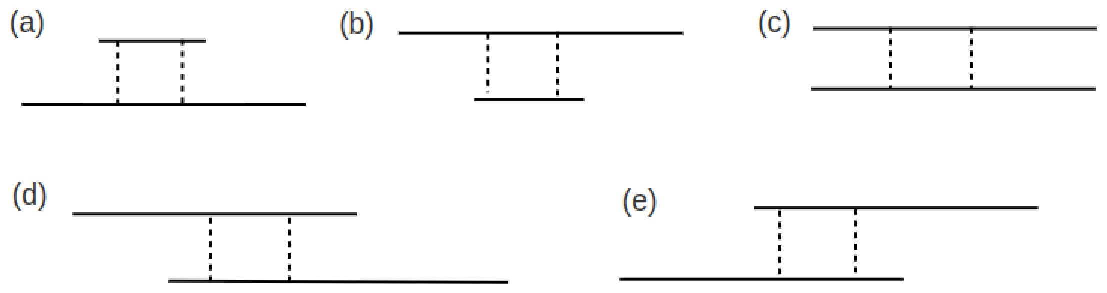


Figure 1.2: Mapping classification in Miniasm. (a) first contained (b) second contained (c) internal match (d) first to second read overlap (e) second to first read overlap. Only the overlaps ((d) and (e)) are considered when building the assembly graph.

In the graph cleaning phase, Miniasm removes the transitive edges in the graph, trims tipping unitigs composed of few reads and pops small bubbles in the graph. The bubble detection is based on Khan's topological sorting algorithm [8]. Multiple research groups have independently developed algorithms for the detection and removal of bubbles similar to the one used in Miniasm. [15]

Next, the unitig sequences are generated from the assembly graph. As the graph does not contain multi-edges, for a particular path, the sequence we are looking for is the concatenation of substrings in the vertices forming the path, taken in the order defined by the path. The paths Miniasm is looking for, called unitigs, are defined as follows:

Definition (unitig): Let $G = (V, E, l)$ be a transitively reduced assembly graph. Then unitig is a path $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_k$ such that $deg^+(v_i) = deg^- v_{i+1}$ and at least one of the following conditions is satisfied: (i) $v_1 = v_k$ or (ii) $deg^-(v_1) \neq 1$ and $deg^+(v_k) \neq 1$.

Simply said, the unitigs are the paths that can be unambiguously merged into one vertex without affecting the connectivity of the original graph – so there are no ambiguous paths to other vertices that would prevent generating a sequence. A simplified example of a transitively reduced assembly graph is in Figure 1.3. The graph with resulting merged vertices for the unitigs is in Figure 1.4.

The unitig sequences are returned to the user in GFA format. The format includes the information about the resulting sequences, but it also contains some additional information about the selected segments of the reads, their overlaps and paths in the graph. The GFA output from Miniasm can be directly converted to FASTA format, which contains only the resulting sequences (contigs).

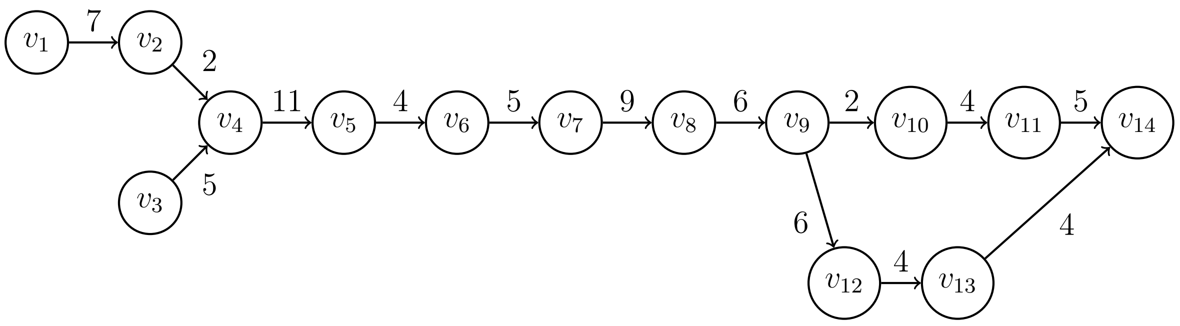


Figure 1.3: A simplified example of a transitively reduced assembly graph. The paths $v_1 \rightarrow v_2 =: utg1$, $v_4 \rightarrow v_5 \rightarrow v_6 \rightarrow v_7 \rightarrow v_8 \rightarrow v_9 =: utg2$, $v_{10} \rightarrow v_{11} =: utg3$, $v_{12} \rightarrow v_{13} =: utg4$ and also the vertices $v_3 =: utg5$ and $v_{14} =: utg6$ by itself are unitigs. Those are the maximal paths that can be unambiguously merged without affecting the connectivity of the graph. For example the path $utg3$ can not be extended to $v_9 \rightarrow v_{10} \rightarrow v_{11} \rightarrow v_{14}$. We can not include the vertex v_9 into the path, because $deg^+(v_9) = 2$ and $deg^-(v_{10}) = 1$, which does not satisfy the condition $deg^+(v_i) = deg^-(v_{i+1})$. Also, we can not prolong the path by adding the vertex v_{14}, because $deg^-(v_{14}) = 2 \neq deg^+(v_{11}) = 1$.

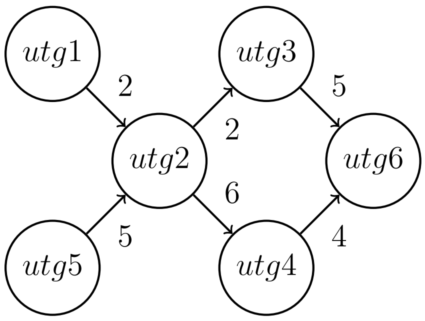


Figure 1.4: Merged paths of unitigs in a transitively reduced assembly graph corresponding to Figure 1.3

The consensus step is not implemented in Miniasm. Therefore, Miniasm does not correct sequencing errors and the error rate of the unitigs is the same as the error rate of the input reads.

1.3 Problem of real-time incremental assembly

Minimap2 and Miniasm expect the whole input to be given at once, after the sequencing is completed. In our work, we were looking for a way how to efficiently perform the genome assembly dynamically. After investigation of the algorithms of Minimap2 and Miniasm, we have concluded that modifying those algorithms to update the structures inside Minimap2 and Miniasm dynamically may not be a feasible approach.

For Minimap2, it might be possible to store the minimizers for the reads that were produced up to the given point of the sequencing run and compute only the minimizers for the new, unseen reads. Whether it is possible to extend the rest of the algorithm for dynamic assembly is unclear. The underlying data structures were not designed in a way that they could be extended dynamically. Therefore, we have decided not to investigate this option further.

For Miniasm, adding mappings to the assembly graph dynamically would be complicated, because in the early steps of the algorithm, Miniasm performs coverage-based filtering and trimming of the reads based on the coverage. Therefore, a dynamic extension of Miniasm would either require a potentially large amount of additional memory to store the additional information, or we would not get significant speed improvements with such extension. Also, Miniasm is highly optimized and fast enough for real-time assembly, and it is not the bottleneck of the data processing needed for the genome assembly. The time bottleneck of the Minimap2 and Miniasm assembly is the Minimap2 read alignment.

With the above in mind, we have decided to utilize Minimap2 and Miniasm in a pipeline which runs them as they are. However, running the analysis on the whole dataset would be too time-consuming if we would do that regularly – instead, we want to maintain a minimal representative sample of all the already seen reads from the sequencing run and use the sample for the assemblies along with new reads produced in the run.

Chapter 2

Sampling reads for efficient assembly

This chapter focuses on the approaches for selecting a representative sample of the reads needed for the assembly. First, we discuss the reasoning behind this approach and the biological and sequencing-specific aspects that we have to consider. Then we describe the sampling strategies we decided to implement in our pipeline.

2.1 Finding sufficient coverage for Miniasm assembly

When building an assembly, we do not need all of the data from the sequencing run. After achieving some coverage threshold for each contig, we can assemble most of the contigs from the data.

To determine the sufficient coverage for Miniasm with its default parameters for Oxford Nanopore reads, we have created the Miniasm assembly with multiple different fractions of data from the *Saprochaete ingens* yeast sequencing run. First, we have selected a random sample of 90% of the reads from the run (9/10 for the whole dataset), then 8/9 from the 90% random selection to form the 80% of all data in the dataset, and down to 10% in a similar way.

For each of these samples, we have assembled the reads using Minimap2 and Miniasm, calculated the N50 score [1] of the assembly compared to the reference genome, and average coverage of the reference genome. The resulting N50 scores, corresponding coverages and numbers of contigs are shown in Figure 2.1. Based on the N50 score and number of contigs, it seems that we can build most of the assembly from 40% of data, corresponding to approximately 30x coverage of the reference genome.

Even though this is only a rough estimate, it suggests that keeping a representative sample of reads for each contig of size smaller than the whole dataset may be sufficient

[1] the sequence length of the shortest contig we need for 50% of the reference length, if we sort the contig lengths in the assembly from longest to shortest and take the longest contigs

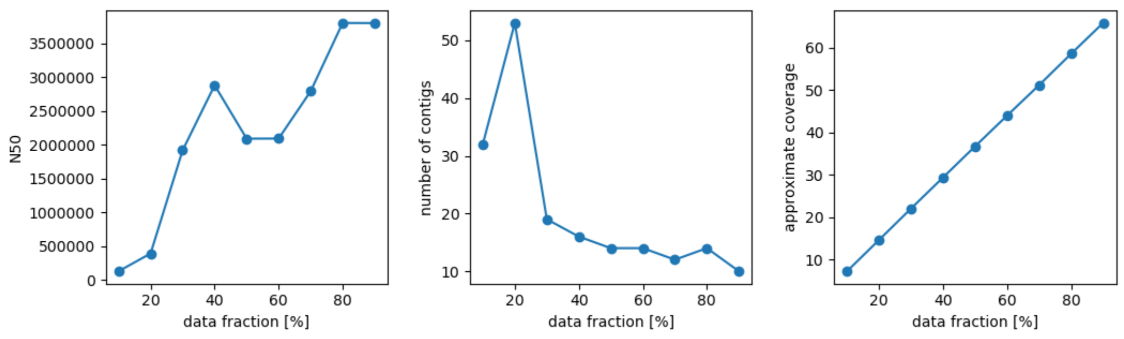


Figure 2.1: N50, number of contigs and reference coverage for assemblies of random samples of different sizes from the whole dataset

for preserving the most of the information for the assembly, since we can build most of the assembly using only a fraction of the data. Also, using a sample that is not chosen in a completely random way can help us to improve this result further.

2.2 The representative sample

2.2.1 The goal of the sampling process

Keeping a smaller sample of the data defined by some coverage threshold might be a plausible approach to minimize the amount of data that have to be processed multiple times in dynamic assembly process. The results from pre previous Section show that if we keep only a part of the reads, we can preserve most of the information for the assembly. However, taking a completely random sample from the data would cause that we will preserve the differences in coverage within the parts of the genome, and consequently leave out the data from the parts of the genome for which the coverage is low.

We want the sample to be representative in sense that we should keep the data that give us the most information. To do this, we should take into account the fact that the coverage may differ for different contigs in the reference genome. We may also want to prefer longer reads over the short ones, as they can be more informative for the assembly.

Since we are designing the pipeline for *de novo* assembly, we can not determine the coverage using a reference genome. Instead, we can only rely on the partial assembly we have up to the given point in the time of the sequencing run. The ultimate goal is to preserve the assembly we have with a smaller amount of data (sample) and to improve it further with the new data that will come later in the next iterations. This can be achieved by sampling the reads we already have for each contig in the assembly up to the given coverage threshold so that the coverage for each contig is sufficient.

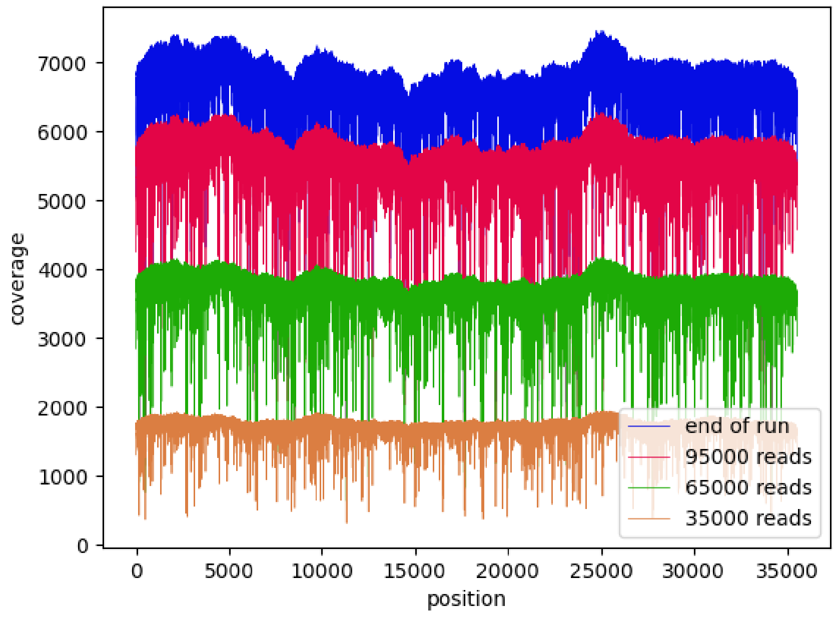


Figure 2.2: Coverage of mtDNA of *Saprochaete ingens* yeast reference genome. First 3500, 65000, 95000 and all of the reads from the sequencing run were aligned to the contig and per-base coverage values were calculated. The coverage is significantly higher than for the chromosomal contigs.

2.2.2 Contig coverages

It is essential to create a separate sample for each contig, not for the whole assembly, since the coverages between the contigs differ. The most significant difference can be seen when comparing chromosomal contigs and mitochondrial DNA, but it also varies between the chromosomal contigs. For example, in the data we have used for testing, the chromosomal contigs have around 60x coverage, and the mitochondrial contig has around 7000x. The coverages for different amounts of data during the run for the mitochondrial contig of *Saprochaete ingens* yeast are shown in Figure 2.2.

Since the mitochondrial contigs are usually significantly shorter (for the *Saprochaete ingens* yeast, the mtDNA contig has less than 0.2% of the genome length) compared to the chromosomal contigs, the impact of the reads from the mitochondrial contigs on the average coverage when it is calculated as a sum of alignment lengths divided by the assembly length for the whole assembly is not too high. However, in the early stages of the run, the difference between the number of reads for chromosomal contigs and for the mitochondrial contigs would cause issues if we did not separate them.

Also, at the start of the run, there are lot of reads that do not align to any assembled contig from the data up to that point, so we have to separate the unaligned reads from the reads that align to the already assembled contigs. We want to keep all the unaligned

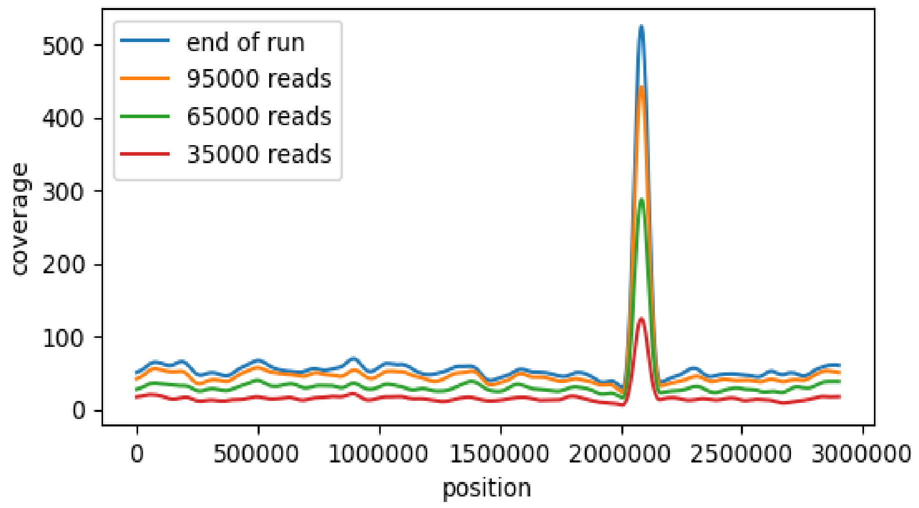


Figure 2.3: Coverage of contig4 of *Saprochaete ingens* yeast reference genome. First 3500, 65000, 95000 and all of the reads from the run were aligned to the contig, and double moving average of the coverage values was calculated with window size 50000. We can see the uneven coverage in this plot. The region around the 2.1-millionth position is an rDNA repeat.

reads, since the fact that they are unaligned means that we do not have enough data to build a contig from them. However, they have to belong somewhere. From the aligned reads, we may have more than necessary amount of data for the contigs, so we can sample the reads for each contig to a given coverage threshold.

2.2.3 Limitations of average coverage metric

The commonly used metric for evaluating the amount of data from the sequencing run is the average coverage. We are using the average coverage as a guiding metric when considering the amount of data we have for each assembled contig. However, the average does not incorporate all information that could be useful for efficient sampling, because the number of reads mapped to a position can vary even within a single contig.

An example of this is contig4 [7] of the *Saprochaete ingens* yeast, which we will later use for testing. We have computed the coverage using Minimap2 per-base alignment of all the reads in different stages of the run and plotted the moving average of the coverage values. The resulting plot is shown in Figure 2.3. We can observe that in the region around 2.1-millionth position, the coverage is significantly higher (500 at the end of run) than in the rest of the contig (less than 100). It is caused by the fact that there is a long rDNA repeat in this region.[7] For the other contigs, the variance in coverage is lower. The coverage plot for contig1 (Figure 2.4) is provided as an example.

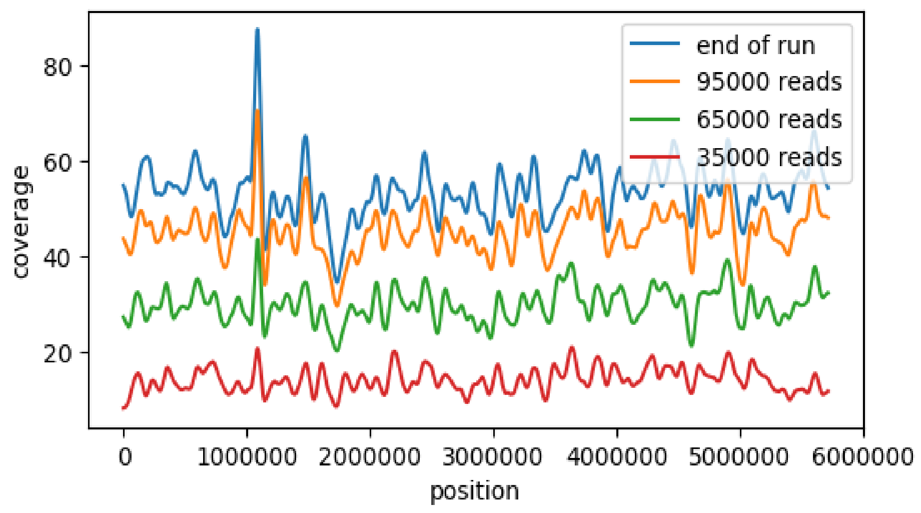


Figure 2.4: Coverage of contig1 of *Saprochaete ingens* yeast reference genome. First 3500, 65000, 95000 and all of the reads from the run were aligned to the contig, and double moving average of the coverage values was calculated with window size 50000. We can see that the coverage variance for this contig is lower than for contig4.

The use of the average coverage metric is a limitation of our approach. For the strategies to work reliably using the average coverage metric, we would need an assumption that for each contig, the coverage is evenly distributed across the contig. In some cases, this assumption is not met, and therefore, we can have lower than desired coverage in some regions of the contigs in which the reads are not distributed evenly. If the coverage threshold is high enough, or the regions with high coverage are short compared to the contig length, it may not be a big issue. If the threshold is low, it can cause that the contigs that have uneven coverage will not be fully assembled from our sample.

It is genome-specific how much the coverage varies and whether it is significant, but regions with different (too low or too high) coverages by the reads do cause issues when assembling genomes in general. To address this issue, we would have to consider the coverage in a per-base or at least per-region manner. It could be possible to identify the high-coverage regions in a more complex way, but since it would likely be too slow for real-time sampling, we did not implement such approach.

2.2.4 Sum of mapping lengths vs. read length

When calculating the coverage of a sequence by a given set of reads, a common and fast approach is to simply divide the sum of the lengths of all reads by the length of the given sequence. However, such approximation may be too far from reality for smaller assembled sequences. At the start of the run, when we do not have much data, this approach may overestimate the average coverage value and cause us to downsample some of the data we need.

There can also be a problem with per-contig coverage estimation caused by the chimeric reads. Some sequences may join and be sequenced as a single read. They often cause issues with assembly, since they can create unwanted bubbles and ambiguous paths in the assembly graph. These problems are resolved to some extent by Miniasm in its filtering and graph cleaning phase. However, in the sampling process, the chimeric reads in the correctly assembled contigs can cause issues when computing per-contig coverage considering read lengths only, because there may be chimeric reads that join the molecules from two different contigs. Such reads can be significantly longer than their alignments to different contigs. This issue is not that significant when considering coverage for the whole assembly – in that case, a read spanning multiple contigs is not a problem. However, it matters if we count the coverage for the contigs separately because, in this case, it can lead to overestimation of the coverage and downsampling of too many reads.

We want to avoid overestimation of the average coverage, which may lead to a sample with a smaller actual average coverage than we want. Too small coverage may cause that we will not be able to rebuild the assembly from the sample. Because of this, we are considering the sum of alignment lengths from Minimap2 for each read and contig combination instead of simply taking the read length. This reduces the overestimation of coverage and gives us a better estimate of the actual coverage.

Aligning the reads to the new assembly in each iteration is necessary because we have to identify the sets of reads for each contig. Therefore, we need the Minimap2 alignments of the reads to the assembly. We can use the same alignment to separate the reads to sets for contigs and to count alignment lengths, so considering the alignment lengths does not require an additional time compared to considering read lengths only.

2.2.5 Repetitive sequences

Repetitive sequences often cause problems in genome assemblies since resolving them from short reads is hard. For the sampling process, if there is a long repetitive sequence in the assembly, the reads spanning a part of the repeat can align to multiple positions in the repetitive region of the assembly. Therefore, Minimap2 with its default settings outputs a lot of alignments for such regions. Miniasm then considers each of these alignments in its coverage-based filtering step, so keeping too many short reads for the repetitive region is unnecessary. Instead, we should keep the longer reads from the region. The issue with highly covered repetitive regions can be partially avoided by not using the secondary alignments in the reads to assembly mapping for coverage computation – but even without the secondary alignments, the coverage is significantly higher for the repetitive regions.

The correct approach for sampling reads for repetitive regions is unclear. If we do not implement some additional heuristic for detecting and handling such regions (which may be hard to do in real-time and is a common problem in assemblers in general), the best option that intuitively seems to be reasonable, is to prefer long reads in the whole dataset. This way, we expect that for the repetitive regions, we will have a higher number of longer reads and lower coverage generated by the reads that align to multiple positions of a repeat. Also, this can overall improve the resulting assembly, since the longer reads can provide us more information about how the reads should be connected.

For this purpose, also using the sum of aligned lengths for each read instead of read lengths (as mentioned in the previous section) is more accurate, as it is more consistent with how Miniasm handles the coverage.

2.3 Sampling strategies

In order to monitor how sequence assembly evolves during the sequencing run, we have implemented a pipeline that iteratively processes the new reads from the run and maintains a sample of the reads seen up to the given point in the run. The exact structure of the pipeline will be discussed in more detail in Chapter 3. In this section, we describe the approaches (*sampling strategies*) for creating the sample from the reads in the run.

In all the strategies, we are creating the samples per contig due to reasons mentioned in Sections 2.2.1 and 2.2.2. At the end of the sampling process for each iteration, the sampled reads for the contigs are merged into a single sample. Since the contig sequences change in each iteration, keeping the reads separated among the contigs is not useful, unless we would compare the assemblies between the iterations.

It may be possible to try to align the old contigs to the new ones to avoid re-aligning the reads from the sample to the new assembly, since the assemblies between two iterations are similar. It could, however, introduce some further issues. Also it probably would not improve the running time a lot, since mapping sampled reads to an assembly does not take too much time, while mapping assemblies to each other and comparing the reads in the samples between iterations would require additional time. Therefore, we have decided not to implement such strategy.

Sometimes, Miniasm can generate two contigs that align to each other. This can cause the coverage in this region to be higher, which we do not want. This issue also has to be addressed.

2.3.1 Incremental sample

The first approach to sampling reads from the run we have implemented builds a sample during the run in a way that a read that is added to the sample set at any point of time stays in the sample set for the whole time. In other words, the sample is built incrementally as the run progresses.

Let r_i be the sum of the lengths of alignments of read i to given contig c. Let ℓ_c be the length of contig c and n_c be the number of reads mapped to the contig c. We can also denote the set of indices of mapped reads to contig c as M_c. We want the coverage at each position to be approximately at a given threshold t. Computing the coverage at each position in a deterministic way would be too slow, therefore, we have implemented a probabilistic sampling approach.

For each contig c, we want to select a sample of reads $S_c \subseteq M_c$, for which

$$\sum_{j \in S_c} r_j \approx t \cdot \ell_c \tag{2.1}$$

To achieve that the expected sum of alignment lengths for selected reads will be approximately $\ell_c \cdot t$, we are selecting each read i from M_c with probability

$$p_i = \min\left(\frac{\ell_c \cdot t \cdot r_i}{\sum_{j \in M_c} r_j^2}, 1\right)$$

Then, the expected sum of alignment lengths will be at most

$$\sum_i p_i r_i = \sum_{i \in M_c} \frac{\ell_c \cdot t \cdot r_i}{\sum_{j \in M_c} r_j^2} r_i = \frac{\ell_c \cdot t}{\sum_{j \in M_c} r_j^2} \sum_{i \in M_c} r_i^2 = \ell_c \cdot t$$

So the coverage generated by this model satisfies the condition (2.1).

Also, since the value of $\ell_c \cdot t$ divided by the sum of squares of alignment lengths is a constant for each contig, longer reads for the sample will have higher probability to be selected.

The reads are added in this way to the sample incrementally, so in the new iteration, we always have to consider the coverage that the assembled contig has from the reads sampled in previous iterations; it can be calculated from the alignment of the reads sampled in previous iterations to the new contigs. The coverage threshold is then lowered for each contig accordingly based on the coverage information, so that the new resulting coverage is not too high.

We also have to address the issue of reads aligning to multiple contigs, since we do not want to have higher coverage for the intersecting contigs. To solve this issue, we sample the reads for contigs in order from the longest to the shortest contig, and if there are reads mapped that were sampled for longer contigs, the desired coverage threshold for the contig is lowered accordingly.

The drawback of this sampling approach is that it can result in higher coverage in the parts of the contigs that were assembled earlier, because in each iteration, we are adding the reads randomly for each contig, and we do not consider the coverage distribution within the contig in the sampling process.

At the beginning of the run, when the contigs are short, we may in some cases downsample more reads, because some of the values for the fraction in the definition of p_i will be greater than 1, but the probabilities p_i for those reads will be set to 1. The expected value of the sum of the alignments is overestimated in that case. This effect will be investigated in more detail in Chapter 4.

2.3.2 Mapping new reads to assembly to detect overhangs

The sampling strategy from the previous Section can be further improved by some heuristics. If we already have a contig built from Miniasm that has sufficient coverage, it is not necessary to add new reads that align only to the inner part of that contig. Instead, we can use and sample only those new reads that extend beyond the ends of already assembled contigs (we will call them *overhangs*), since those are the reads that can give us new information.

However, it is not sufficient to only take the lowest amount of reads from which a contig can be built using Miniamp2 and Miniasm because by adding new reads, we would obtain higher coverage at the ends of the contig, and lower coverage somewhere in the middle. This would result in problems with the assembly in the following iterations. To address this problem, we have implemented a strategy that takes all aligned reads for each contig until it achieves coverage given by the threshold, and after that, we keep the new overhangs only, not all the new reads that align to the contig. Also, we limit the number of overhangs so that coverage of contig ends is only slightly higher than threshold. More precisely, we select the longest overhanging reads for the contig ends.

Compared to the approach from Section 2.3.1, this strategy can also reduce the coverage variance within a contig, because if there is an uneven coverage in the already assembled part of the contig, no more reads will be added to the region where the coverage is high. Also, the parts of the contigs assembled earlier that already satisfy the coverage threshold will not get any new reads; instead, the new reads will be added to the ends.

2.3.3 Reducing the bias towards the reads from the start of the run

In the strategies from Sections 2.3.1 and 2.3.2, the assembly is built mostly from the reads at the start of the run. Sometimes, the newer reads can give us some additional information, even though the coverage is sufficient.

One of the reasons for this is that it is known that during the nanopore sequencing run, the read lengths increase over time, and the read qualities decrease over time. To illustrate this, we have counted the average read lengths per file (500 reads each) for the data from the *Saprochaete ingens* sequencing run and plotted the result (Figure 2.5). The reads are slightly longer towards the end of the run, but there are also some points in the run, where the reads in a number of consecutive files are significantly shorter. The files with shorter reads are caused by the flow cell cleaning process during the sequencing run, which is done regularly in a fixed time intervals (approximately 1.5 hour).

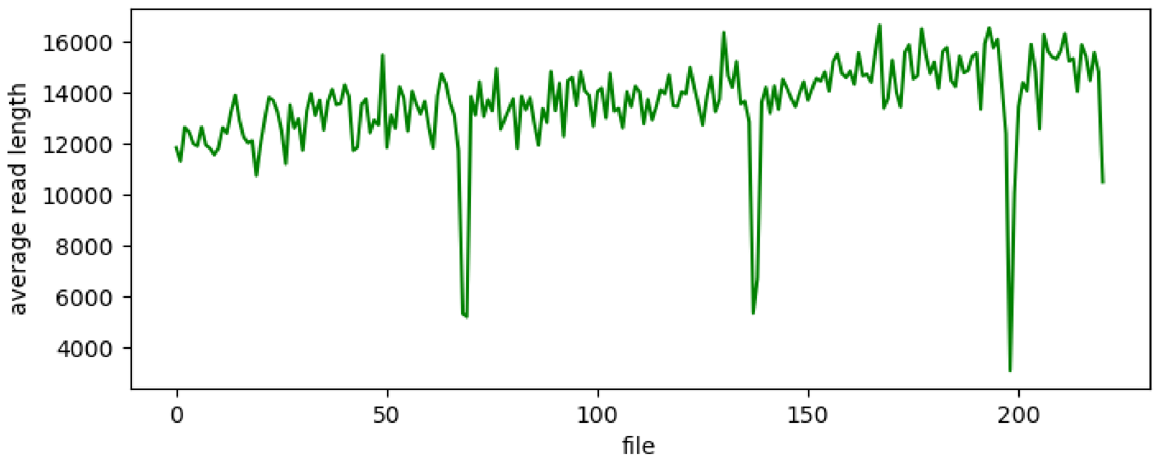


Figure 2.5: Read lengths at different stages of the nanopore sequencing run. Average read length for each file was computed. The values are plotted in order in which the files were produced in time, each file containing 500 reads.

With this strategy, we aim to form a less biased sample. To achieve this without causing the sample to grow over time even after we reach the coverage threshold, we have to allow replacement of the reads in the sample. Also, we want to ensure that the new reads have similar probability of being selected for the sample compared to the older ones. To do this, we assign a random number p from interval $\langle 0, 1\rangle$ to each new read. The value for the read stays the same during the whole run. Then, for each contig, we sort the reads aligned to the contig according to these values in descending order. We compute prefix sums of lengths of alignments for this sorted array and find the value of p_t for which, if we take all the reads for contig that have greater or equal

p, we will get the desired coverage. All the reads for the contig that have p greater than or equal to p_t are then added to the sample. The reads that were not selected for the sample are discarded and not considered in the subsequent iterations. We expect the thresholds for p values for contigs to increase over time, since we will keep only the reads with the highest values. This means that the reads which did have lower p would not get into the sample anyway, so we do not need to keep them.

Even though the reads are selected based on their random values, there is still some bias towards the reads from the start of the run, because the first partial assemblies were build from the first reads. However, this bias is reduced over time, since there are new reads in the sample in each iteration, forming a new assembly – in each iteration, the assembly is built using the new reads from the batch and the sampled reads. The exact ratio of how many new and old reads we use for assembly is influenced by the batch size and the coverage threshold.

The drawback of this strategy is that it may take more time than the strategies from previous sections, and it can be harder to implement some further possible time improvements of the pipeline on top of this strategy, since the sample can change gradually during the run. To use some information from the previous iteration, we would have to look for an intersection between the old and new sample, which can take some time. In the previous strategies, we have this information directly.

Overlapping contigs and chimeric reads

There may be intersections between the sets of reads for different contigs that have different threshold values. The intersections may be caused by chimeric reads between the contigs, or they can occur if Miniasm builds two contigs that map to each other, especially if the region has higher coverage. It can also be caused by the sequencing errors or by the chimeric reads within the contig that can cause ambiguous paths in the Miniasm assembly graph.

Since the p values are set per-read, they are not independent for different contigs. Therefore, it may not be necessary to check whether there is some coverage for the contig by the reads sampled for the contigs that were processed before. However, the assumption may be wrong in some cases.

Consider a long contig c for which we take all of the reads as its sample S_c, since it has low coverage (so the p_t threshold for this contig will be 0). For some other contig d, that is shorter than c and has higher coverage, we take some set of reads S_d, in which all the reads have the values of p above or equal to some threshold $q > 0$. Let S_{xd} be the set of reads mapped to contig d that have p value lower than q. Then, if $S_{xd} \cap S_c$ is non-empty, there will be some additional coverage for contig d from the reads in the intersection.

In case $S_d - S_c$ is non-empty, the additional coverage can be avoided by considering the coverage from longer contigs before sampling reads for the shorter ones. In some specific scenarios, when there are multiple contigs that align to each other, this may help to reduce the uneven coverage caused by those contigs. Also, this approach reduces the number of misassembled contigs, because the scenario described above is typical for them – they align to some other contig (have some reads with it in common), but they are also built from some other reads, that do not align to the correctly assembled contig.

We can modify the strategy to process the contigs in order from the longest to the shortest, and explicitly check for the coverage that we already have for a contig from the reads that were selected for the longer ones. Then we can lower the thresholds for the shorter contigs accordingly and sample only from the reads that were not used for the longer contigs in order to complete the smaller contig samples.

2.3.4 Preferring longer reads

We can also use a strategy similar to strategy from Section 2.3.3 with a different goal. If we use the sums of alignment lengths instead of random values of p, we can get a sample that favors the reads with longer alignment lengths. These values for reads can change over time, since the new contigs can be assembled, but this should not be a problem. Also, the values for the same reads may be different for different contigs, but not completely unrelated. This way we can get a sample consisting of reads with the longest mappings, which can give us the most useful information we can get from the data. Also, this approach is deterministic, which has some further benefits.

2.4 Summary

Considering the aspects of the data from the sequencing runs discussed in Sections 2.1 and 2.2, we have designed and implemented six sampling strategies.

For the strategies that build the sample incrementally, the strategy from Section 2.3.1 will be referred to as *strategy A*, the strategy from Section 2.3.2 will be denoted as *strategy A-map.* The strategies that aim to reduce the bias towards the start of the run will be labeled *strategy B-u* for the first strategy from Section 2.3.3 that relies on the assumption that the p values are not independent (it does not consider the coverage generated by the reads from longer contigs for the contig we are considering) and *strategy B-c* for the strategy that does not rely on the assumption that the p values are not independent, and explicitly checks the coverage from the already selected reads for the sample.

The modified versions of the *B-u and B-c* strategies that consider the mapping lengths instead of the randomly generated p values for each read (Section 2.3.4) will be denoted as *strategy D-u* and *strategy D-c*, similarly to *B-u* and *B-c.*

Table 2.1 provides a summary of the properties of the strategies.

	Section (1)	**deterministic (2)**	**sample changes (3)**	**cons. longer contigs cov. (4)**	**overlaps only after thr. (5)**	**all reads until thr. (6)**
A	2.3.1	No (probability based on the alignment length)	incrementally (reads added to the sample can not be replaced)	Yes	No	No
A-map	2.3.2	No (probability based on the alignment length)	incrementally (reads added to the sample can not be replaced)	Yes	Yes	No
B-u	2.3.3	No (random choice of reads)	randomly after the threshold is reached	No	No	Yes
B-c	2.3.3	No (random choice of reads)	randomly after the threshold is reached	Yes	No	Yes
D-u	2.3.4	Yes (reads with the longest alignments)	if there are new, longer reads, they replace shorter	No	No	Yes
D-c	2.3.4	Yes (reads with the longest alignments)	if there are new, longer reads, they replace shorter	Yes	No	Yes

Table 2.1: Properties of sampling strategies.
(1) Section in which the strategy is described,
(2) whether the strategy is deterministic,
(3) how the sample changes during the run,
(4) whether it considers the coverage from the reads sampled for longer contigs when sampling for a contig
(5) whether the strategy considers overlaps only (Yes) or all the reads (No) from the batch after the threshold is reached,
(6) whether all reads from the run are stored until the threshold is reached.

Chapter 3

Building a real-time assembly pipeline

This chapter discusses the principle, structure, and implementation of our real-time assembly pipeline.

3.1 Pipeline structure and components

The pipeline iteratively processes the data as they are produced during the sequencing run. In each iteration, we process a new set of reads (batch) from the run, and we use the new information we have from them for the assembly along with a selection of reads (sample) from the previous iterations.

The goal is to provide the user with real-time information about whether there is enough data from the sequencing run for the genome assembly. At the start of the run, when we have only a small amount of data, we can build the assembly from all the available data in real time. However, as the run progresses, the amount of sequenced data grows, and assembling all the available data from the run would take too much time. Therefore, we want to limit the amount of data we have to process in each iteration, which can be achieved using the sampling strategies from Chapter 2.

3.1.1 Reads in the pipeline iteration

The reads processed in an iteration can be divided into three main categories: *new reads from the batch* (N), *reads previously included in the sample* (S), and *previously unmapped reads* (U).

During the iteration, we first compute a genome assembly using the reads included in the N, S and U sets. Using the iteration assembly, we produce two *new* sets of reads, that will be used in the next iterations:

- *The iteration sample* (S_{new}): Reads selected as a representative sample for the contigs in assembly for the current iteration. These include a subset of reads from N, S, and U.

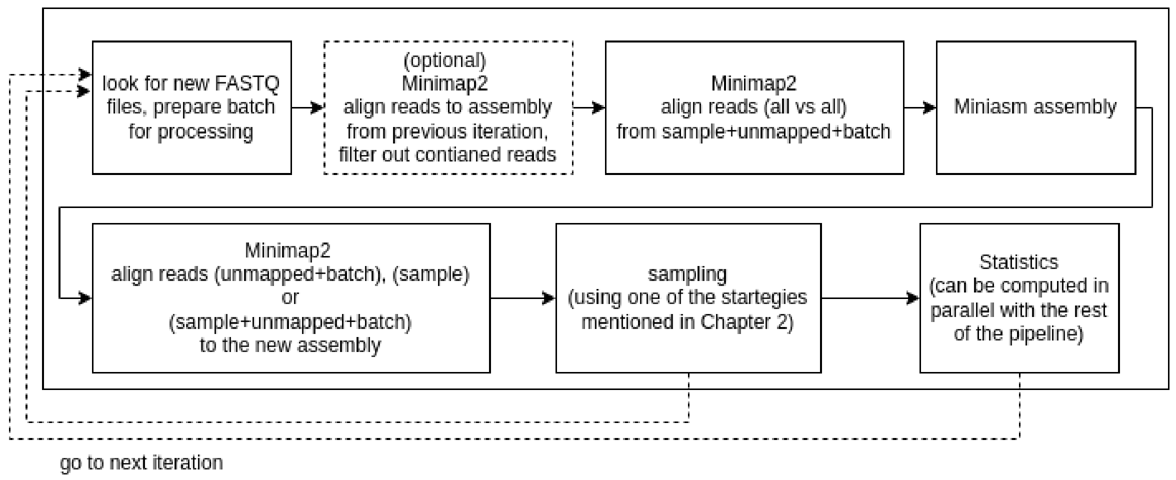

look for new FASTQ files, prepare batch for processing
(optional) Minimap2 align reads to assembly from previous iteration, filter out contianed reads
Minimap2 align reads (all vs all) from sample+unmapped+batch
Miniasm assembly
Minimap2 align reads (unmapped+batch), (sample) or (sample+unmapped+batch) to the new assembly
sampling (using one of the startegies mentioned in Chapter 2)
Statistics (can be computed in parallel with the rest of the pipeline)
go to next iteration

Figure 3.1: An iteration of the real-time pipeline. There are some minor differences in the exact implementation of the components that depend on the sampling strategy that is used for the run.

- *The iteration unmapped reads* (U_{new}): Reads from the current iteration that did not map to any contig in the assembly for the current iteration. Any reads from N, U, or even from S may end up in this set. If there are reads from S, it means that some contig that was assembled in the previous iteration was not assembled in the current iteration. This can happen, since we may sometimes leave out a read that was important for the assembly and not include it in the sample. This is a drawback of the probabilistic or heuristic sampling approaches.

In the subsequent iteration, the new set S_{new} of sampled reads form this iteration will be used as set S, and the new set U_{new} of unmapped reads will be used as set U. The S and U read sets are dynamically updated by the pipeline in this way during the sequencing run. The purpose of keeping the unmapped reads separated from the sampled reads is that depending on the sampling strategy used, they may be handled differently in the sampling step.

For each iteration, the assemblies and list of read IDs for the reads in the sample are stored, so that they can be used for statistics computation and further analysis.

3.1.2 The pipeline components

The pipeline consists of several components that repeat in each iteration: batch preparation, (optional) read filtering, Minimap2 alignment and Miniasm assembly, Minimap2 alignment for sampling, the sampling process and statistics computation. The overview of the pipeline iteration is in Figure 3.1.

Batch preparation At the start of the iteration, the pipeline first loads the new reads for the iteration (set N from Section 3.1.1) – the new batch is prepared for processing. This is done by linking the specified number of files (that were not seen by the pipeline up to this point) from the input directory to the pipeline's working directory. At the end of the iteration, the links are cleared.

Mapping reads to assembly After the batch is prepared, if we are using the sampling strategy *A-map* from Chapter 2, the reads from the set N are aligned to the previous iteration assembly and only the reads that align to the ends of the contigs are kept (the strategy aims to preserve the existing assembly with the sample, and it considers only overhangs for the contigs as the new reads – hence, we want to keep only the new reads that extend beyond the ends of the already assembled contigs, and filter out the rest). The alignment is done by Minimap2. The resulting PAF file is parsed in the pipeline, and only the overhangs are kept for further processing of the batch.

For all the strategies except *A-map*, the set N of reads from the batch proceeds unchanged to the next step – the reads from the batch are directly used in Minimap2 and Miniasm assembly.

Minimap2 and Miniasm assembly We are building the assembly in two steps. First, we align the reads onto themselves using Minimap2. For the alignment, all the reads from the sets N, S and U are used (except for the A-map sampling strategy, for which only a subset of N is used instead of N). Then the assembly is created using Miniasm. Before the alignment, a temporary file with reads is created by merging the three sets of the reads, since both Minimap2 and Miniasm expect a single file with reads in FASTQ or FASTA format as an input. The result from Minimap2 is a set of alignments in PAF format that can be directly used as the input for Miniasm. Miniasm outputs the assembly in GFA format, so we have to convert it to FASTA format for further processing.

Aligning reads to assembly and sampling After the new assembly in FASTA format is obtained, we have to align the reads from which the assembly was built to the assembly. This is done either separately for the sample from the previous iteration and for the unmapped and new reads (for the strategies A and *A-map* from Chapter 2), or in one Minimap2 run for all the reads from which the assembly was built (for the rest of the strategies). Then, the PAF files are parsed and sampling is performed by one of the sampling strategies. The sampling step is crucial for the real-time performance, since it ensures that the amount of data between the iterations will not grow after we reach a certain coverage threshold.

Statistics When assembly and sampling is finished, we compute the statistics that the user can view in real time during the sequencing run. The statistics can be computed either sequentially or in parallel with the rest of the pipeline (this is a user-configurable option). However, computing statistics in parallel may require too many resources, since it will cause multiple instances of Minimap2 to run simultaneously.

3.2 Pipeline output for an iteration

At the end of the pipeline iteration, we have an assembly, the new sample set S (sample of the reads aligned to the assembly that will be used in the next iteration), the new set of unmapped reads U, and we compute the statistics that are provided to the user in real-time during the sequencing run.

3.2.1 Statistics

In the general case, when the reference genome for the data we are assembling is unknown, we can compute statistics such as assembly length, number of reads, number and length of the assembled contigs, and contig coverages. The contig lengths (Figure 3.2) and the average coverages for contigs (Figure 3.3) are provided to the user as bar plots. An example of the summary numeric statistics for an iteration follows.

processed files: 190
processed reads:95000
num of files in input directory:223
catch-up:85.20%
run time:
start time: 2024-04-16 23:45:24.083492
end time: 2024-04-16 23:50:18.570348
duration: 0:04:54.486856
stats end: 2024-04-16 23:53:48.263965
stats duration: 0:03:29.693617

sample/assembly statistics:
# contigs: 9
# subsampled reads: 39141
# unmapped reads: 7314
sum length of contigs: 20609398
sum length of reads: 663134420
avg coverage (crude) [sum of read lengths/assembly length]: 32.18
N50 (raw assembly): 2640274 (*only if reference genome is given*)

We are computing average coverages per contig, but we also want to show a per-base coverage plot to the user (an example is in Figure 3.4). One option would be to compute the per-base coverage using Minimap2 in BAM format. However, in that case, Minimap2 completes the per-base alignments precisely using dynamic programming, and the computation is too slow for real-time results. Plotting such results also does not give an easily readable information to the user, since there are typically large

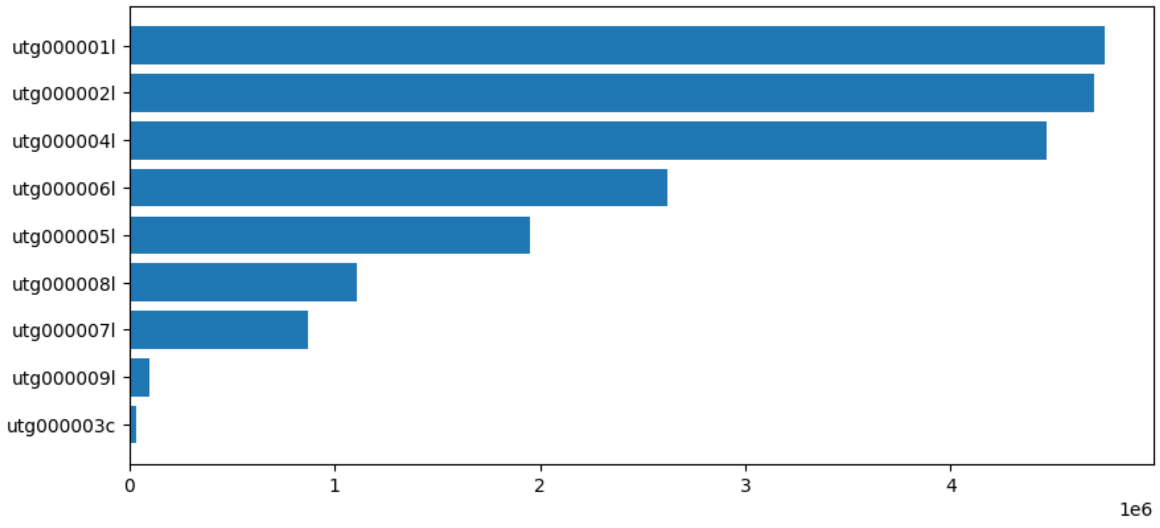


Figure 3.2: Assembly contig lengths plot.

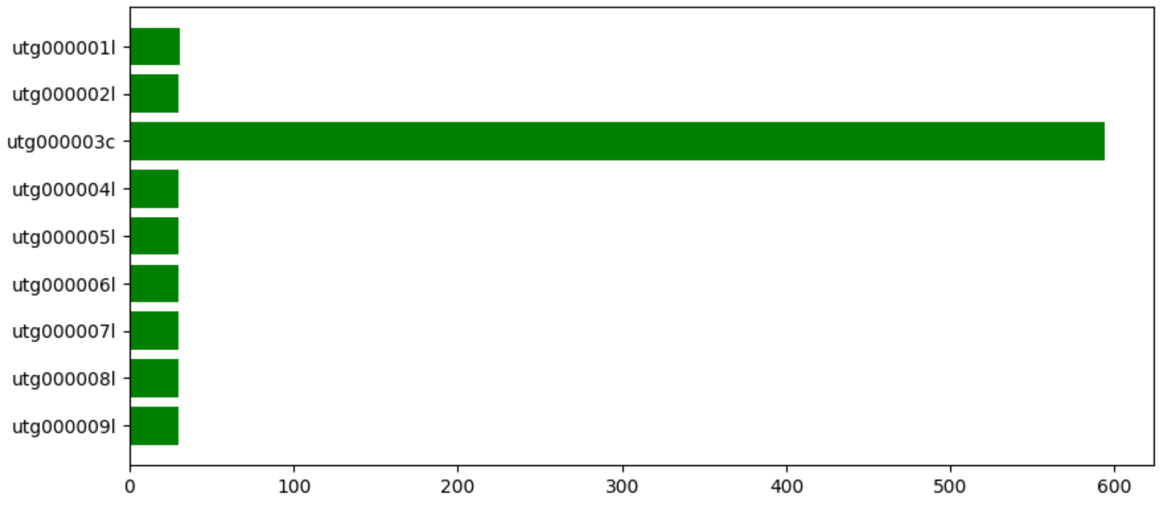


Figure 3.3: Average coverage of the assembly contigs by the reads in the sample.

differences between the coverage values on per-base level. Therefore, we are using the PAF alignments only, and we are computing the coverage based on the mapped regions from the PAF file.

If the reference genome is given, the pipeline can also compute the statistics using the reference. In case the user has enough resources to run two instances of Minimap2 simultaneously, the reference-based statistics can be computed in parallel with the statistics without the reference genome, since they are independent (otherwise, when we are limited by the resources, the statistics can also be computed sequentially). In addition to the statistics without the reference genome, if the reference is given, we are computing the N50 score of the assembly with respect to the reference genome.

To get more precise information about the assembly fragmentation compared to the reference genome, we align the assembly to the reference using Minimap2 and create a dot plot (Figure 3.5). We also compute the reference coverage by the reads from the sample (using Minimap2 with its result in the PAF format) and plot the results for each reference contig (similarly to the way how we are computing the coverage for iteration assembly). At the bottom of the graphs for the reference contigs, the regions of assembled contigs that align to the reference contigs are plotted as rectangles, indicating the fragmentation of the assembly compared to the reference genome (see Figure 3.6). Also, a BED file with the assembly to reference alignment information is stored for each iteration and can be used for further analysis.

In the statistics implementation, we use the matplotlib Python library. Since matplotlib is not thread-safe, it was necessary to handle the plotting as critical sections and implement mutual exclusion accordingly. Even with this limitation, some time can be saved by the parallelism if we want to compute both the statistics with and without reference genome in real-time, as the plotting sections are not the time bottleneck of the statistics threads.

The first level of parallelization option for the user is the parallel run of the assembly-only and reference-based statistics. It is also possible to configure the pipeline to run the statistics for the previous iteration in parallel with the next iteration. If the user has enough resources to run multiple Minimap2 instances simultaneously, a lot of time can be saved by the parallel run, allowing a resource-time trade-off.

The pipeline also has the option to run the statistics with BAM alignments to get more precise statistics and plots. However, this version of statistics is not applicable in real-time scenario, because it is too slow.

3.3 Time efficiency

In the early stages of the sequencing run, when we have only a small amount of data, the output from the pipeline is equivalent to the direct computation of the assembly and the statistics from all of the available data.

However, after we reach a certain coverage threshold for the assembly, we work only with a constant amount of data for each iteration: (a) the data from the previous iterations downsampled to the given coverage threshold and (b) the new data from the batch that were created within a constant time during the sequencing run.

Therefore, we can achieve real-time performance even in the latter stages of the sequencing run by setting the pipeline parameters (batch size and coverage threshold) in a way that the time needed for the computations in the pipeline iteration will not exceed the sequencing time for a batch. The effect of the different choices of the parameters will be examined in more detail in Chapter 4.

3.4 Technical details of pipeline implementation

We have implemented the described real-time assembly pipeline in Python. The implementation consists of

- a main script that loads the user-configurable parameters and creates the *PipelineRunner* class to initialize and run the pipeline
- class *PipelineRunner* and its subclasses that manage the different pipeline workflows (those are sampling-strategy specific)
- class *FileTracker* that manages the batching process
- class *ReadSubset* and its subclasses that implement the different sampling strategies from Chapter 2
- class *Minimap*, which is a wrapper class for running Minimap2 in different modes (all vs. all reads alignment, alignment of reads to assembly/reference genome) and different types of outputs (PAF or BAM alignments)
- class *Miniasm*, which is a wrapper class for Miniasm. Its purpose is to run Miniasm using the alignments from Miniamp2 and the corresponding set of reads in FASTQ format. It also converts assembly from the GFA to the FASTA format, and saves the assembly for each iteration.
- separate classes for computing statistics – those can compute the statistics either with or without the information from the reference genome, and there are various parallelization options

There is also a script that runs a simple server that can be used for displaying the pipeline statistics result for each iteration (in HTML format) to the user so that they can navigate between the results as the run progresses.

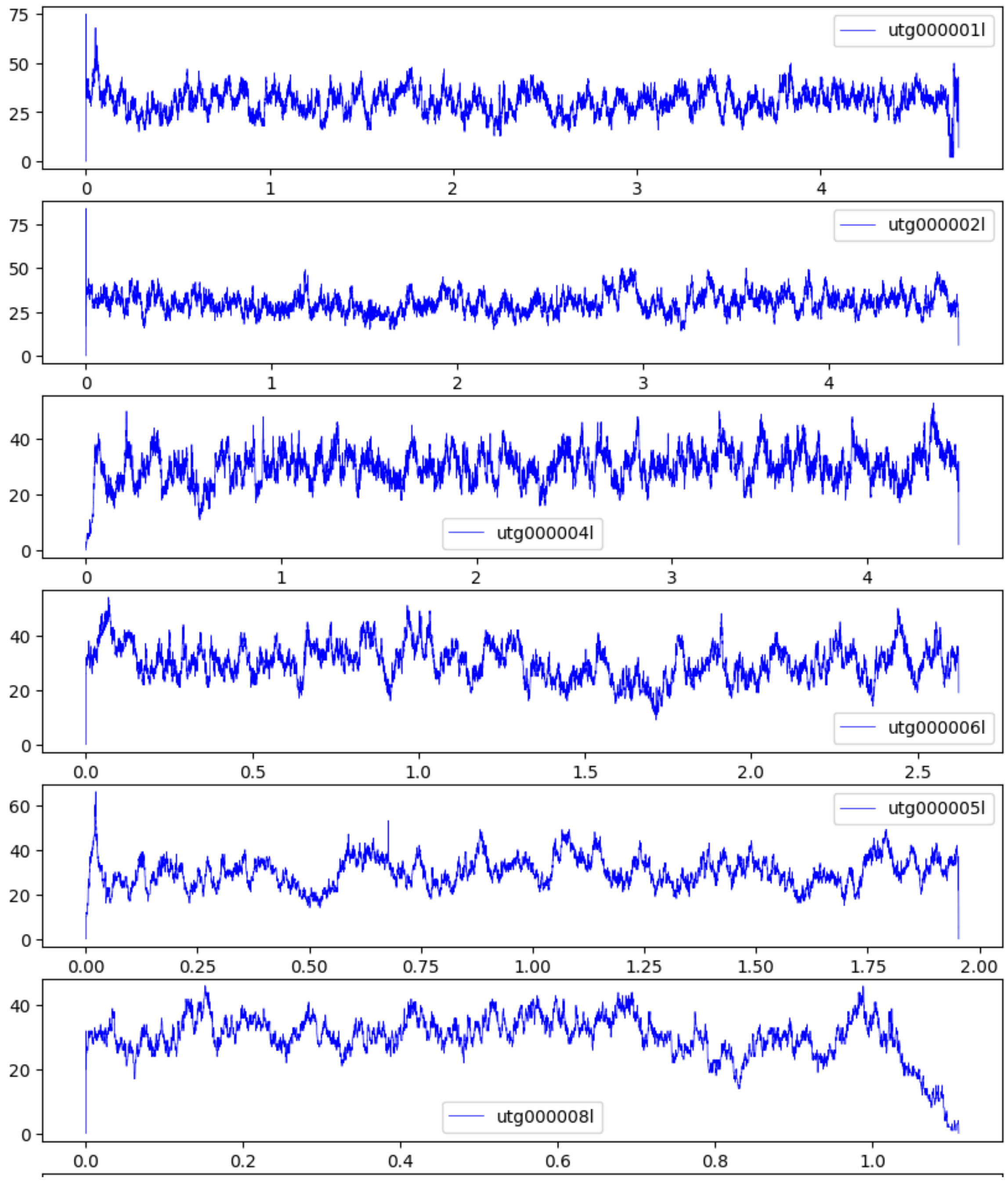


Figure 3.4: Per-base coverage of the assembly contigs by the reads in the sample counted based on PAF alignments. The contigs are sorted by length; only the first 6 (out of 9) are in this figure. On the x-axis is the position within the contig in millions, and on the y-axis is the coverage.

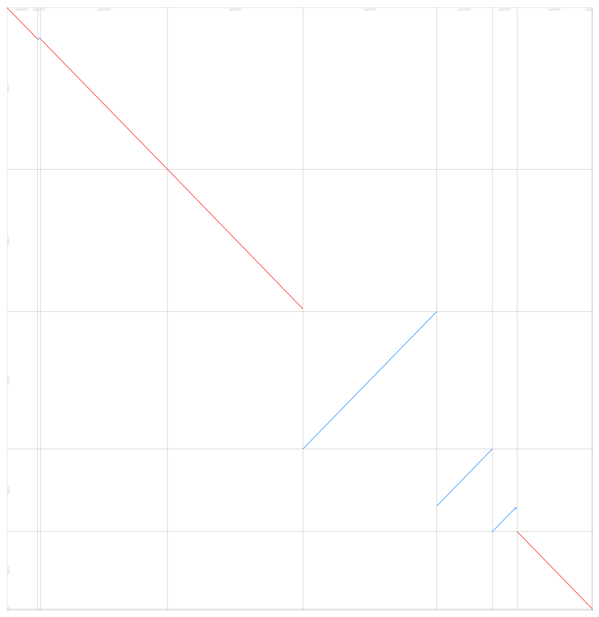


Figure 3.5: Assembly to reference alignment dotplot created from the PAF alignment using minidot (which is a part of the Miniasm repository) [12]. On the x-axis, there are the contigs from the assembly; on the y-axis are the contigs from the reference genome and the lines correspond to the aligned segments of the contigs.

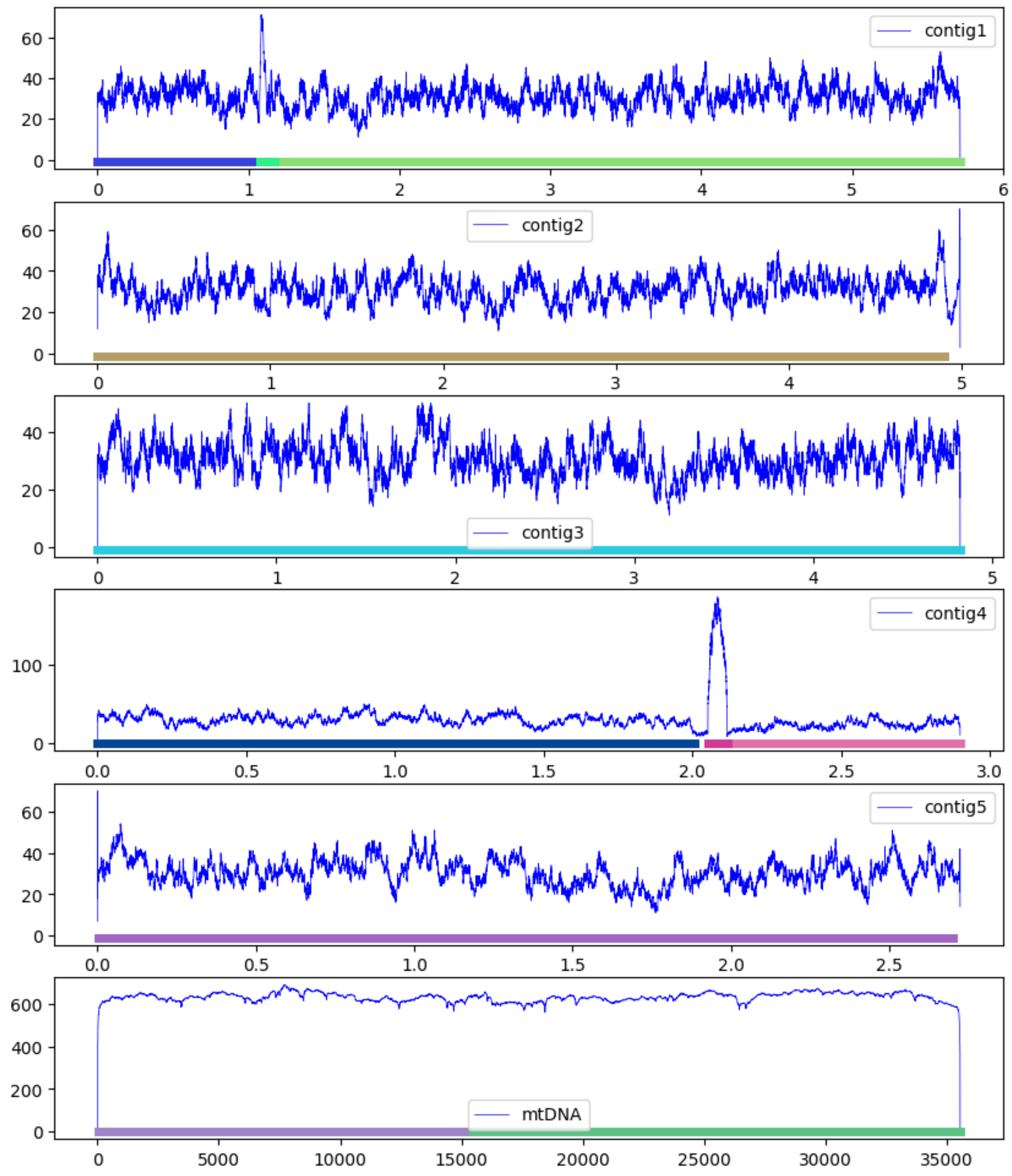


Figure 3.6: Per-base coverage of the reference genome by the reads from the sample. At the bottom of the graphs, there are alignments of the contigs from the assembly to the reference genome.

Chapter 4

Results

In this chapter, we will discuss the results of our real-time assembly pipeline implementation with focus on the quality of the resulting assembly and the running time.

We have run a set of experiments using our implementation of the real-time assembly pipeline utilizing the sampling strategies described in Chapter 2. We have tested the sampling strategies with different coverage thresholds given to the pipeline on the nanopore data from the nanopore sequencing run to evaluate how the choice of the strategy and the coverage threshold affects the quality of the assembly and the running time of the pipeline.

The time and the quality of the assembly is also affected by the size of the batch. For most of our measurements (Sections 4.3, 4.4 and 4.5), we have used a batch size of 10 files with 500 reads in each file. In Section 4.6 we present results for batch of size 20 and coverage threshold of 30x for comparison.

4.1 The data and the reference assembly

We have evaluated our implemented pipeline on the data from the *Saprochaete ingens* yeast MinION (ONT) sequencing run performed by Hodorová et al. [7].

The reference genome created using long reads from MinION (ONT) combined with short reads from Illumina sequencing has five chromosomal contigs and one mitochondrial contig [7]. The lengths of the contigs in the reference genome are in Table 4.1.

The Miniasm assembly on the whole dataset (all the reads we have) results in eight contigs, with lengths shown in Table 4.2. The order of the reads influences the result of Minimap2, and subsequently the result of Miniasm is also different. Thus, we sorted the reads for the assembly by their sequencing time, in the same order they were presented to the pipeline as an input. The N50 score of the Miniasm assembly

contig name	length [bp]
contig1	5714510
contig2	4992828
contig3	4821795
contig4	2900717
contig5	2723818
mtDNA	35540

Table 4.1: Lengths of *Saprochaete ingens* reference genome contigs.

contig name	length [bp]
utg000001l	5571831
utg000002l	4856367
utg000006l	4690952
utg000008l	2727673
utg000003l	2650865
utg000004l	131920
utg000007l	86677
utg000005c	47368

Table 4.2: Lengths of *Saprochaete ingens* contigs from Miniasm assembly on the whole dataset.

with respect to the reference genome is 4690952.

4.2 Assembly evaluation metrics

The quality of the assembly compared to the reference genome can be evaluated in different ways. However, there is no universal approach or metric that could serve as an indicator of whether an assembly is correct or wrong or how different it is from the reference genome. Instead, multiple numeric values are usually computed for comparison. Apart from them, the bioinformatician or a biologist can manually check the parts that differ and investigate the reasons behind the differences.

One commonly used numeric statistic is the N50 score, also mentioned in Section 2.1. It represents the sequence length of the shortest contig from the assembly needed to cover 50% of the reference genome. To calculate the N50 score, we sort the contig lengths in the assembly from longest to shortest and identify the length of the shortest contig for which its length summed with the lengths of the longer contigs is greater than or equal to 50% of the reference genome length.

Maybe a more informative variation of the N50 score is the corrected N50 (corrN50) score metric. The difference between N50 and corrN50 is that for corrN50 we consider the lengths of the alignments of the contigs to the reference genome instead of the whole contig lengths. This way, we are not using the contigs or parts of the contigs that do not align to the assembly, which can give us more accurate information because the misassembled contigs or regions are omitted from the score calculation.

The limitation of the N50 or corrected N50 score is that it is not that informative if half of the genome is assembled correctly and the rest of the assembly is very fragmented. Therefore, sometimes multiple variations of this statistic are computed, for example the N90 score defined in a similar way as N50, but we are looking for the length of the contig for which 90% of the reference genome length will be achieved.

Another useful statistic is the portion of the reference genome length that is covered by the contigs from the assembly. If the percentage of the reference genome covered by the assembly is high and the N50 (or other N score) is low, it suggests that we have preserved most of the information for the assembly compared to the reference genome (the reads for the assembly cover most of the reference genome), but the assembly is more fragmented compared to higher N50 scores (some parts of the reference genome contigs are not connected in the assembly).

Additional statistics include the coverage of the reference genome or the assembly by the reads. However, after the first few iterations, in our case the coverage corresponds to the threshold given as a parameter for the pipeline, so this information is not useful for comparison of various sampling strategies.

4.3 Determining sufficient coverage threshold

One of the questions we wanted to answer was the sufficient coverage for the representative sample – the coverage threshold value for the pipeline run that gives us enough data for the assembly. To answer this question, we can look at the best results achieved with any of our sampling strategies between the pipeline iterations. We have computed the maximum corrected N50 scores and percentages of reference genome covered by the assembly in each iteration: for each iteration i, we take the maximum of the scores in iteration i from six runs, one for each of the sampling strategies. With such maximum scores computed for runs with multiple coverage threshold values (six runs for each threshold), we can evaluate whether each coverage threshold is sufficient for any of the strategies. We have tested the coverage thresholds 10, 20, 30, 40 and 10000; the purpose of the last one is to define the best case scenario where no reads will be discarded due to threshold limitations.

The comparison of the results for the different coverages is shown in Figures 4.1 (using the corrected N50 score metric) and 4.2 (considering the percentage of reference genome covered by the assembly contigs). Note that the maximum possible corrected N50 score for the assembly is 4992828, as we can see from Table 4.1.

From the Figures 4.1 and 4.2, we can see that coverage threshold 10 is too low, yielding assemblies that are significantly more fragmented compared to the results achievable with higher coverage thresholds. Also, some parts of the assembly are entirely missing in this case. None of the strategies could build more than 95% of the assembly with this threshold, and for the strategy *A-map* (see Table 2.1 for strategy

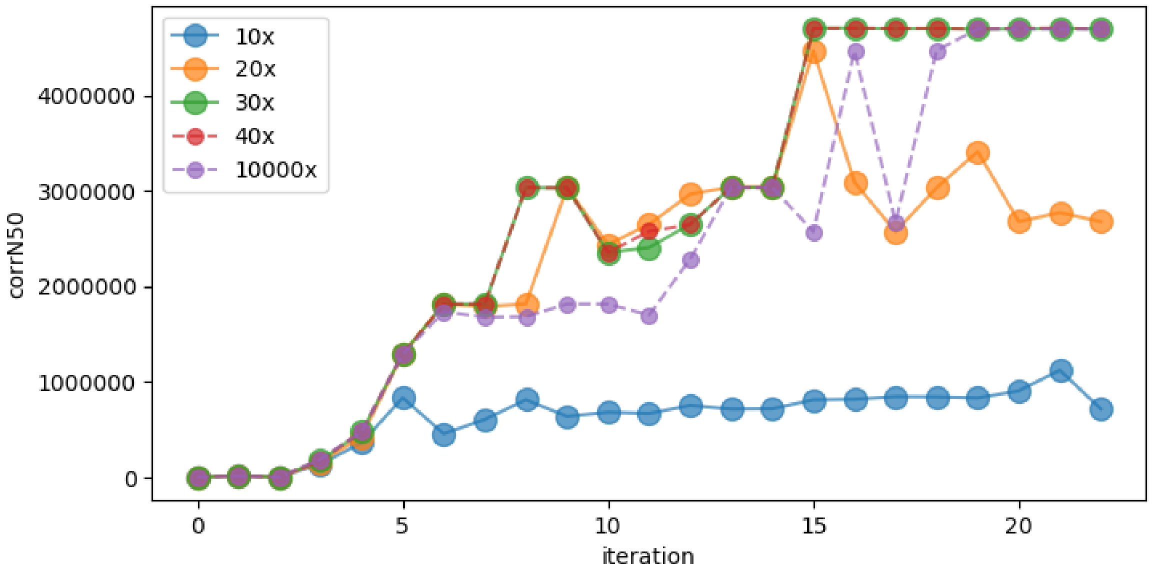


Figure 4.1: Maximum corrected N50 score achieved by the sampling strategies for different target coverages; in each iteration, a batch containing 5000 reads is added as new reads.

properties), even more than 10% of the reference genome is missing with the coverage threshold 10. The differences between the values for the sampling strategies will be discussed in more detail later.

Figure 4.2 shows that the portion of reference genome covered by the iteration assemblies generally increases with further iterations. However, for the corrected N50 score (Figure 4.1), we can sometimes observe drops between iterations, especially for the higher lengths of assembled contigs. This is because a significant change in the score can be caused by removing a small portion of information and the longer the contigs, the more significant the difference between the N50 scores for the subsequent iterations it can cause. We will inspect this in more detail in the next Section.

Our observations in Figures 4.1 and 4.2 also indicate that the coverage threshold 20 and above should be sufficient to build an assembly that is close to the reference genome.

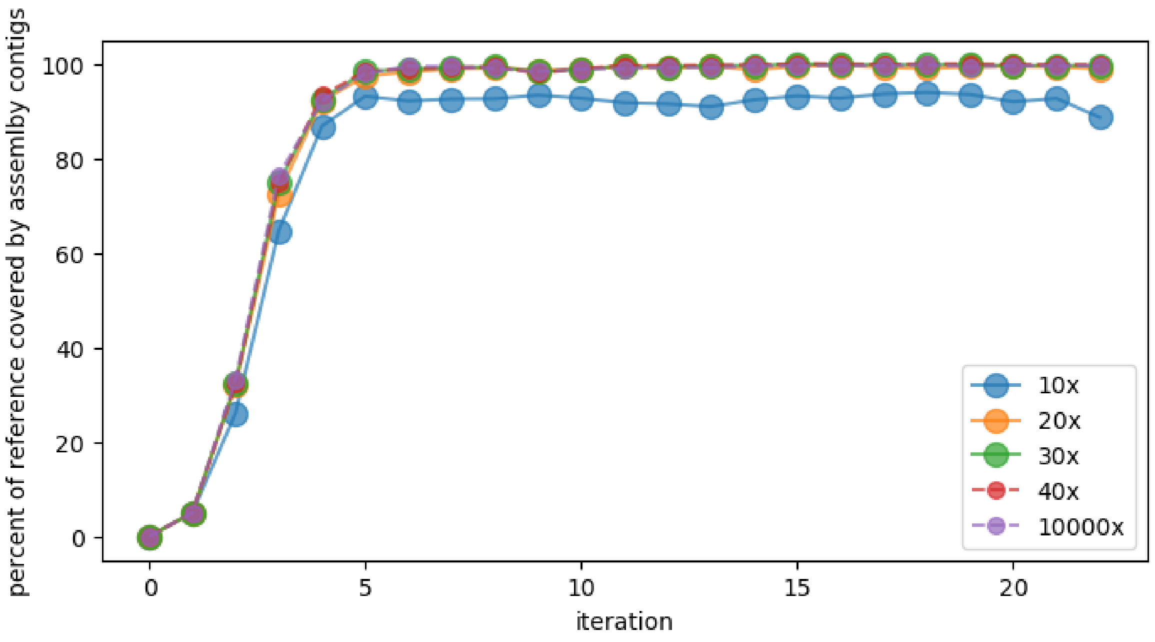


Figure 4.2: The maximum portion of the reference genome covered by assembly achieved by any of the strategies for different coverages for each iteration. In each iteration, the percentage of the reference genome covered by the iteration assembly was calculated for each of the strategies (from six runs with the same target coverage threshold), and the maximum of the six values for the strategies was taken. Such values were computed separately for each target coverage threshold.

4.4 The quality of the assembly

To compare the assemblies between the sampling strategies, we can look at the maximum corrected N50 score and percentage of the reference covered by the assembly that was achieved for each strategy-coverage combination. Each strategy-coverage combination run has multiple iterations, for which the scores were computed individually. Then, the maximum of the values for the iterations was taken as the maximum value for the strategy-coverage combination – so we have computed the best scores that each of the strategy-coverage combination has achieved in any of its iterations. The percentages were computed based on the Minimap2 alignments of the assembly to the reference genome in PAF format (they were not computed with per-base accuracy, and small insertions/deletions compared to the reference genome were ignored). The corrected N50 scores are shown in the heatmap in Figure 4.3, and the percentages are in Figure 4.4.

With the B-c strategy, we have achieved the highest coverage percentages. However, the differences are not very significant (see Figure 4.4). For coverage 20x and higher, we can get an assembly that aligns to more than 99% of the reference genome. This indicates that the B-c strategy is successful at preserving most of the information from

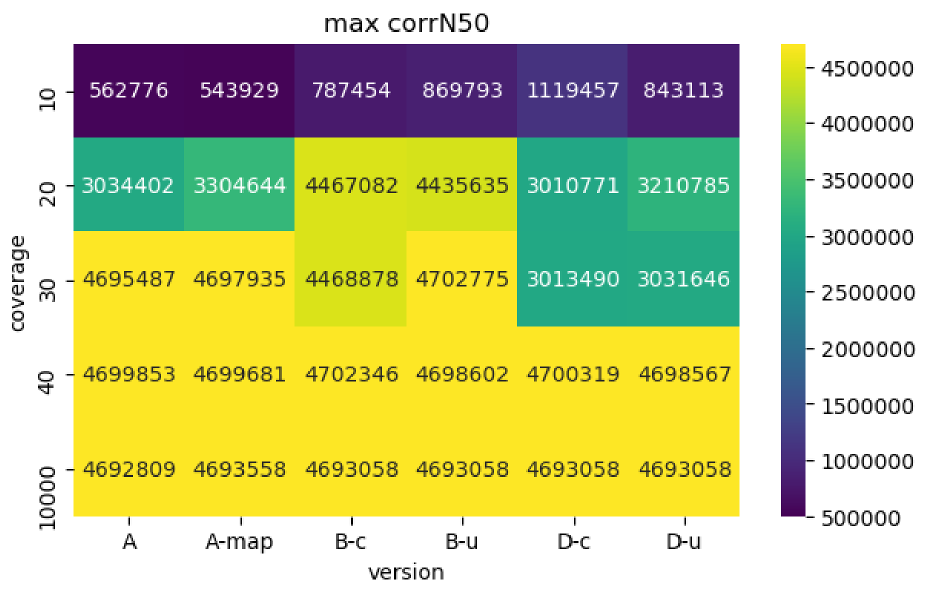


Figure 4.3: Maximum N50 score achieved for each coverage-strategy combination.

sequencing reads. Closer investigation revealed that the missing parts of the assembly are located around the 2.1-millionth position in contig4 in the reference genome, for which we could see a peak in the coverage by all the reads in Figure 2.3. This is caused by the fact that there is an rDNA repeat in the region, as stated by Hodorová et al. [7].

We can see how the assembly contiguity evaluated using corrected N50 score differs for the strategies on the heatmap in Figure 4.3. For threshold 10, we got the best result with the D-c strategy. With the B-c and B-u strategies, we got the best results in terms of assembly contiguity for coverage threshold 20. For 30x coverage, we have achieved best scores with the A and B strategies. In runs with 40x thresholds, the differences between maximum corrN50 score between the strategies are smaller compared to lower thresholds. We can also notice that the scores for 40x are better than in cases when we do not sample the reads – this indicates that sampling of the reads resulted in a better assembly. Interestingly, we have also achieved better N50 score (the N50 score is necessarily greater or equal to corrN50) compared to assembly on the full dataset with the reads sorted by their sequencing time (4690952).

The changes in scores between iterations are important, since the per iteration results are the information that is given to the user in real time. While we would expect the scores will improve over time, Figure 4.1 shows that this is not necessarily the case. This is caused by the fact that the assemblies are always built from the sample and unmapped reads from the previous iteration, along with the new data from the batch. These can include new reads that contradict the previous assembly, for example chimeric reads. The new reads that introduce ambiguity may cause worsening of the assembly.

The sample selection process in the A and B strategies includes randomness. In

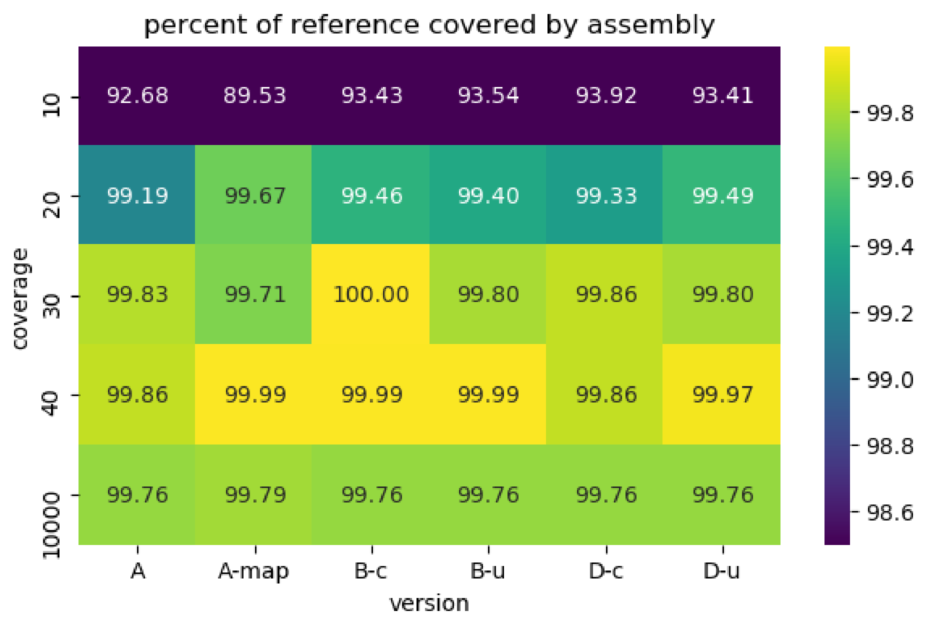


Figure 4.4: Maximum percentage of reference covered by assembly achieved for each coverage-strategy combination. The heatmap range was scaled to start at 98.5 for better visualization, since the values for 10x coverage are low compared to the others.

the A strategies, the sample is built incrementally, so the reads included in the sample once are never discarded. Instead, new reads for the assembled contigs are rejected if the coverage target is reached for the contigs. Therefore, the assembly N50 score can drop between the iteration in which there are some important reads from the batch and the next one, because the reads from the batch may not be included in the sample for the next iteration. In the B strategies, reads included in the sample may later be replaced with different reads. This may sometimes lead to replacement of a read that is key to the assembly contiguity with some other, less informative read, causing assembly quality to worsen.

The D strategies have an advantage in terms that they are deterministic. The sample does not change randomly during the run, instead, the sampling process is guided by the lengths of the mappings, generally preferring longer reads. However, it can happen that some long reads will not improve the assembly, but will break it instead. For example, a long chimeric read that joins two sequences that are not joined in the genome, can cause an ambiguity in the assembly graph, resulting in a contig split into two parts. We are trying to avoid favoring the chimeric reads from two different contigs by considering the alignment lengths for a contig instead of the whole read lengths – this can help us to prevent merging two contigs that should not be merged. However, the chimeras that consist of two reads for a single contig are still preferred over the other reads by this approach – and those are the reads that cause the breaks in the contigs.

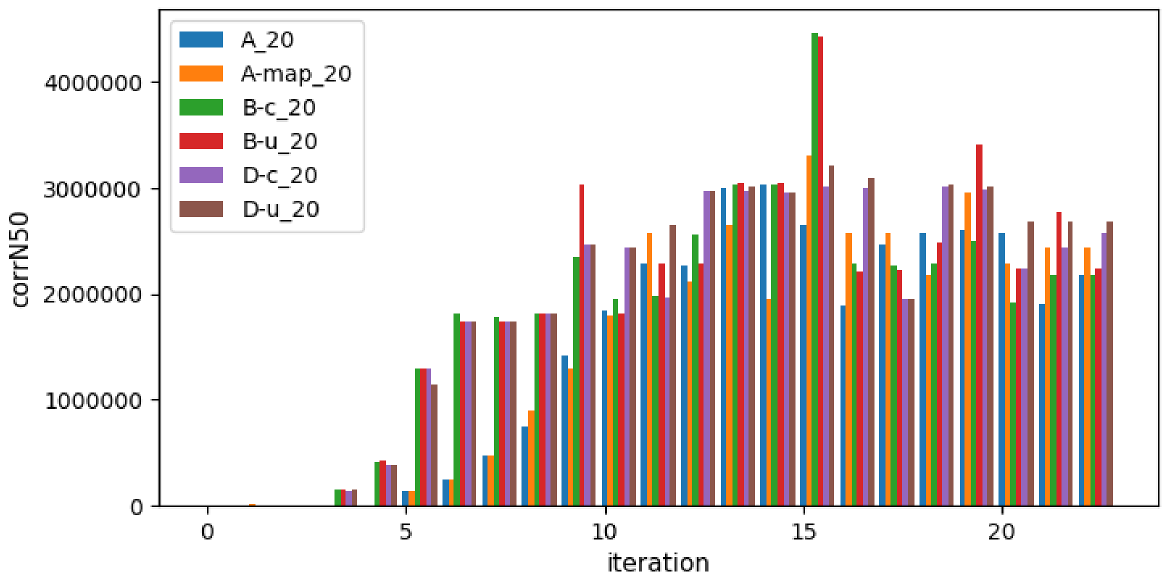


Figure 4.5: Corrected N50 score of the assemblies for the runs with 20x coverage threshold (per iteration).

The mentioned differences between the strategies are significant especially if the coverage threshold is low compared to the amount of coverage that is available from the batches for the partial assemblies. In this case, a large fraction of the sample may change between the iterations. Figure 4.5 shows the per-iteration results for the 20x coverage threshold runs with 5000 reads batch size. The high score for the B-strategies was achieved only in a single iteration, later decreasing as the set of reads for the assembly has changed (not all the reads from the 15th iteration batch were included in the sample, new reads from the 16th iteration batch were added, and also some reads previously included in the sample were replaced). In multiple iterations, the scores for the D-strategies are better than the scores for A and B strategies. The observations suggest that with the B-strategies we can achieve better results in some cases, but the results are highly affected by the randomness of the strategies and are therefore hard to predict. This difference is significant especially if the coverage threshold is low.

For comparison, we can also look at the corrected N50 scores per iteration plot for coverage threshold 30x (Figure 4.6). Naturally, we would expect the scores to be better than for 20x coverage; however, this is not always the case. For example, in the 15th iteration, the B-c strategy has lower score than in the 20x case. In contrast, the A-map strategy has higher scores in multiple iterations. It is because in this case, we are preserving the sample after the threshold is reached, and if any new reads are added, they are always at the ends of the contigs.

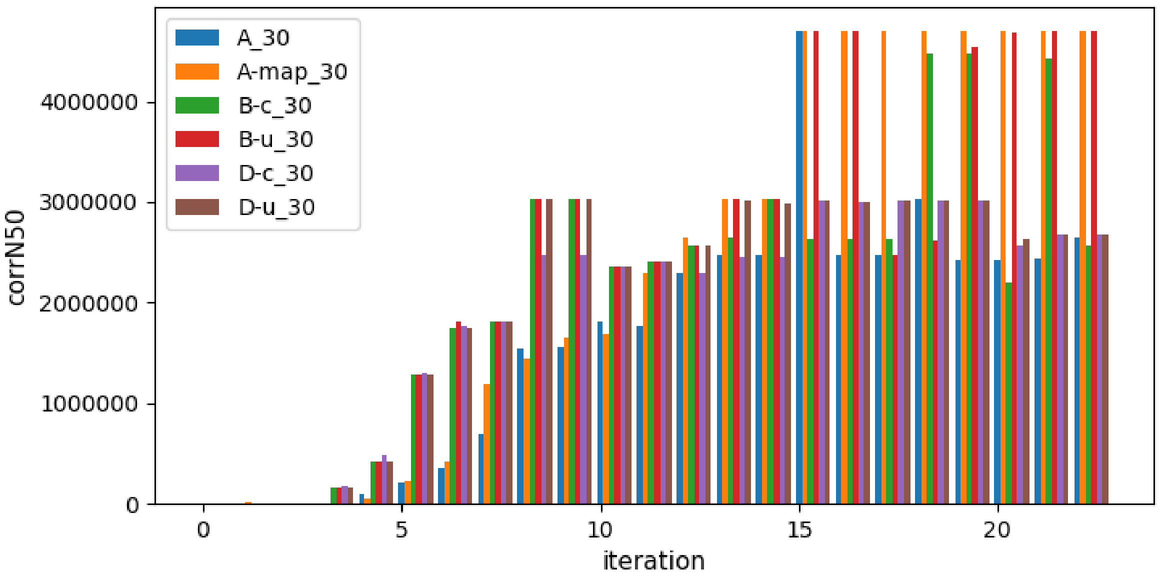


Figure 4.6: Corrected N50 score of the assemblies for the runs with 30x coverage threshold (per iteration).

For the A-strategy, if a longer contig is created in the iteration, it may not have sufficient coverage at the point where two contigs were joined. Therefore, new reads for the contig are sampled to satisfy the threshold. However, the sampling process for the A-strategy does not ensure that the reads will be added to the joining position of the previously separate contigs, in contrast to the A-map strategy which uses the overlaps only. This is the reason why we observe higher score for the A strategy in the 15th iteration, but it drops in the next one, compared to the A-map score that remains unchanged.

Another difference is that the new reads from the batch can contradict the assembly from the previous iteration for the A-strategy, resulting in a contig split into two parts, whereas for the A-map strategy, this should not happen often, because it uses overlaps only (if there are ambiguities caused by the overlaps, the differences are at the ends of the contigs only). Contigs that are assembled differently then influence the sampling process, which can result in new reads being added if the coverage for the contig was uneven, possibly preserving the contradicting information.

We can further investigate the differences in detail by looking at the assemblies in each iteration individually. Figure 4.7 shows the assembly fragments sorted by their alignment lengths to the reference genome, demonstrating further issues not captured by the corrected N50 score.

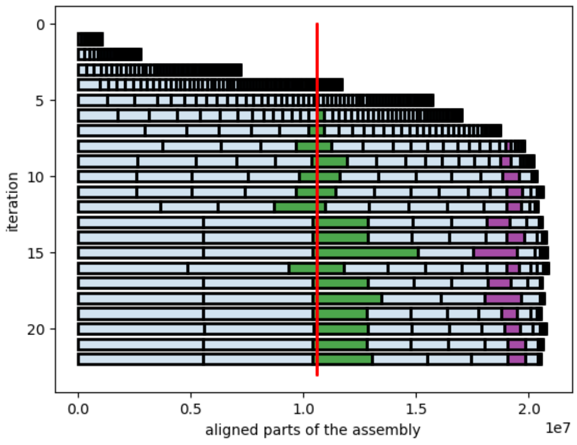
For example, we can see more detailed difference between the assemblies for the A-strategy in Figure 4.7 (a). In the 16th iteration, the assembly is more fragmented compared to the 15th iteration, where we have three long contigs. In the 16th iteration, the contigs were assembled differently, and some other reads may have been added to the sample, which also influenced the assemblies in the following iterations. The longest alignments correspond to a part of the contig1. In the 16th iteration, a chimeric read caused the contig1 to break into two parts. Such changes are hard to predict and are even more significant, if the read set changes a lot.

In the plot for the A-map strategy (Figure 4.7 (b)), we can see that the first three contigs of the assembly did not change a lot after the 15th iteration. This illustrates the main advantage of this strategy – it preserves the information that we already have. The other strategies focus more on improving the assembly using the reads from new iterations, while some information from the previous iterations may be lost in this process.

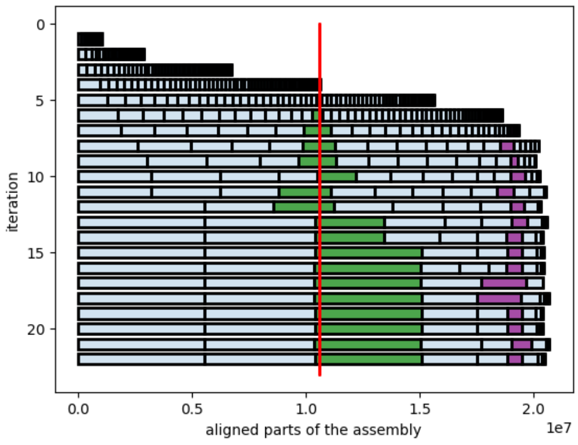
In some cases, we actually prefer assemblies with lower scores. For example, we can see that for the B-c and B-u strategies, the assemblies have better corrected N50 score than D-c and D-u, but for the D strategies, there is a smaller number of alignments on the right side of the red line in the plots, which is not captured by the N50 score – the number of short contigs for the D-strategies is smaller, while they cover the same fraction of the reference as the contigs for other strategies.

Some contigs are harder to assemble than others, because they have more complex structure. The ability of assembling such contigs is also important when considering the quality of the assembly. Whether this is or is not visible in the N50 scores depends on the lengths of those contigs in the reference genome. Therefore, also some other variations of N-scores are commonly used. For example, the corrected N90 scores, for which the contigs are marked in purple color in the plots, would be higher for the D-strategies. Also, there is a number of factors that affect the assemblies. A small change in the sample (for example, if we omit a single read) or in the order of the reads (if the reads are given in a specific order, it may either improve or worsen the assembly) can cause a large difference in the contiguity of the assembly.

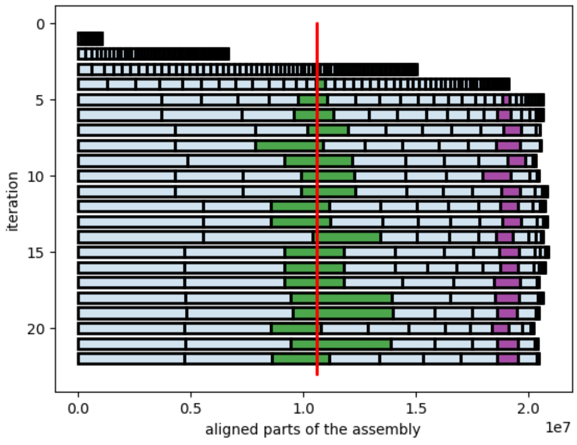
The percentages of reference covered by the assembly correspond to the ones in Figure 4.2 for strategies B and D. The difference between the strategies can be seen in Figure 4.8. For the A-strategies, the values are lower until the $\frac{l_c \cdot t}{\sum_{j \in M_c} r_j^2}$ ratio defined in Section 2.3.1 is sufficient for each contig, because in case the ratio is too low, the A and A-map strategy samples are smaller than the ones for B and D strategies. This is caused by the way how we have chosen the probability of a read being selected for the sample. In some way, it guides the sampling process towards more sampling if the assembly created in the iteration is less fragmented (the contig are longer) compared to the iterations where the assembly is very fragmented and the contigs are short. For higher coverage thresholds, this effect is lower, since we are multiplying the contig



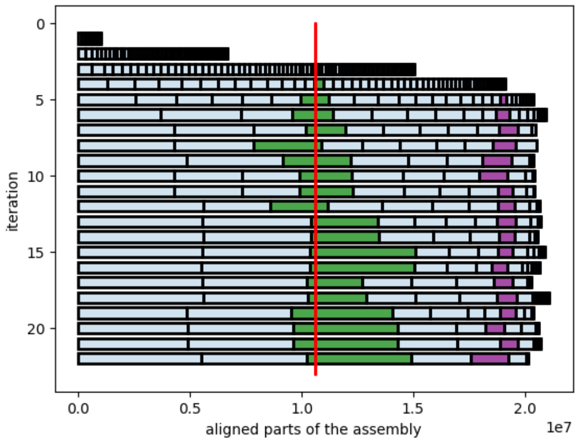
(a) A



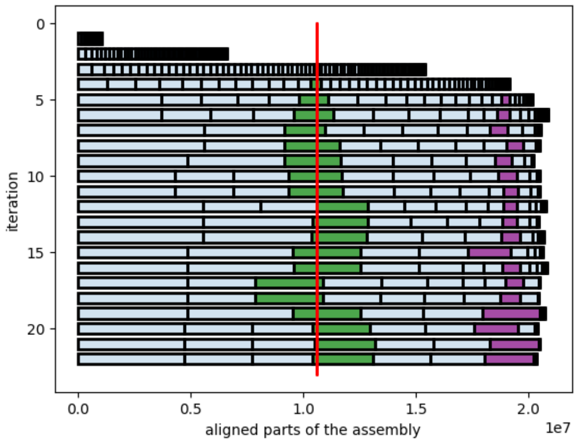
(b) A-map



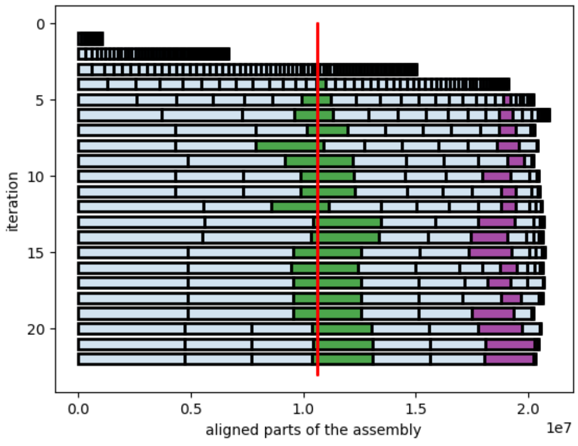
(c) B-c



(d) B-u



(e) D-c



(f) D-u

Figure 4.7: Assemblies for iterations in runs with 30x coverage threshold, corrected N50 and N90 score visualization. The red line corresponds to half of the reference length. The rectangles in each row are the lengths of alignments of the contigs from assembly to the reference genome for one iteration. The green rectangles correspond to corrected N50 score, the purple rectangles correspond to corrected N90 score.

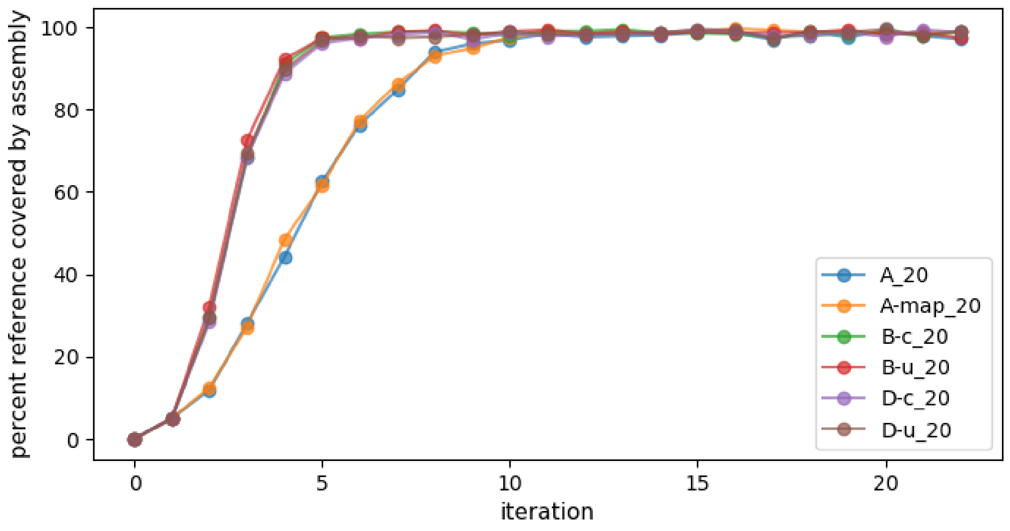


Figure 4.8: Percentages of reference genome length covered by assembly for 20x coverage threshold. We can see that the values for the A and A-map strategies are lower until the 10th iteration.

length by the threshold in the numerator of the fraction. This also explains why in the first half of the run, the B and D strategies give us better corrected N50 scores than the A strategies in Figures 4.5 and 4.6.

In Figure 4.9 we can see how the number of contigs evolves during the run for the strategies. After the fifth iteration, the number of contigs has dropped significantly for the B and D strategies, and two iterations later for the A strategies. At the end of the run, the number of the contigs for the D strategies was lower compared to the number of the contigs for the rest of the strategies. The number of the contigs in the last five iterations for the B-strategies varies in range from eight to 15, but is within range of eight to ten for the D-strategies. For the A strategies, the minimal value in the last five iterations is eight and maximum is 13. In context with the fact that the percentages of reference covered by the assembly are more than 99% for those iterations for all of the strategies, we can conclude that the assemblies that have lower number of contigs cover the assembly but are less fragmented.

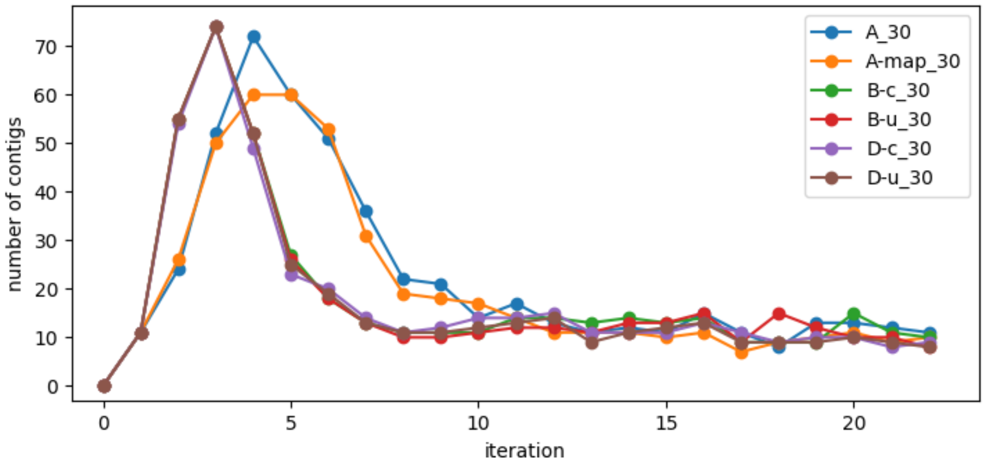


Figure 4.9: Number of contigs per iteration for 30x threshold runs. We can see that the number of contig drops significantly around fifth iteration for all of the strategies.

4.5 Running time

We have measured the running time for each sampling strategy and coverage threshold combination. Minimap2 runs were limited to a maximum of eight threads each. The statistics were computed sequentially with the rest of the pipeline, but statistics computed based on the reference genome ran in parallel with the statistics without reference (they take approximately the same time in most of the iterations). If two Minimap2 instances in the statistics threads run simultaneously (this usually happens if we are computing the statistics with and without reference in parallel), such run can use up to 16 threads and the corresponding amount of RAM needed for Miniamp2 in peak.

The amount of RAM needed for Minimap2 depends on multiple factors, mainly on the number and the length of the sequences. We have measured that for the all-vs-all alignment of all the reads for the Saproecheate ingens yeast in our data, Minimap2 used a little more than 16GB RAM in peak. For the alignment of all the reads to the reference genome (which is a superset of the task done in statistics), the peak RAM usage was around 1.2 GB. The user can influence the amount of RAM that Minimap2 uses by configuring its parameters. However, it is not possible to explicitly limit the amount of RAM for Minimap2 to a given threshold, since it is hard to predict. The amount of memory depends on the character of the data and multiple other factors [14].

In case the user does not want to compute the reference-based statistics (which is the case if we are monitoring a sequencing run for *de novo* assembly), the requirements on computational resources are lower – there would be at most one Minimap2 instance running at a time (except for the case, when the user decides to run the statistics

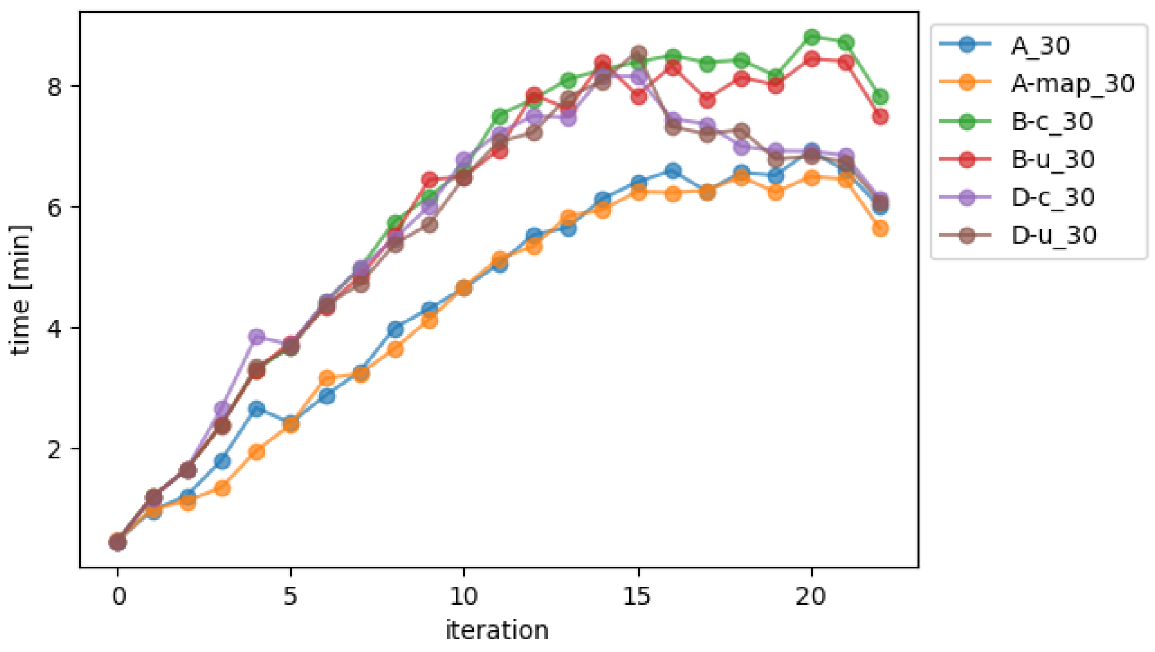


Figure 4.10: Running times per iteration for different sampling strategies with coverage threshold 30x.

in parallel with the rest of the pipeline, which is an option). For the sequential run, the peak RAM needed is the RAM used by Miniamp2 all-vs-all alignment for a set of reads in an iteration, and the maximum number of threads (by default) is nine: eight threads for Minimap2 during the Minimap2 alignment and one thread (which is idle during the Minimap2 alignment) for the rest of the pipeline. The running time in case we would not compute the reference-based statistics would be approximately the same compared to our run – maybe even a little faster, since some minor time delays in our parallel statistics threads run compared to the sequential case can be caused by waiting for the matplotlib lock that is held by the second statistics thread that computes the reference-based statistics.

The running times are different for each sampling strategy and coverage combination; for smaller coverage thresholds, the times are lower. To compare the times and analyse the differences between the strategies, we will use the 30x coverage threshold (Figure 4.10). For each strategy, the time for the iterations grows linearly until the desired coverage threshold is reached. After that, the time required for an iteration is (almost) constant: it does not depend on the amount of previously seen data, which is crucial. As we can see from the plot, the times for the A strategies are lower than the times for the B and D strategies.

The A-map strategy can be faster than A for smaller coverage thresholds (or, more precisely, it is faster if the difference between the threshold value and the coverage of the assembly by the reads from the batch is higher). In that case, we can save some additional time with the A-map strategy compared to the others, since it reduces the time needed for all-vs-all reads mapping. Otherwise, A-map may be slower than the A strategy, because we are spending time by the overlap filtering while not too many new reads have to be filtered out (and it would be more efficient to just include them in the all vs all mapping – which is exactly what the A strategy does).

For the B strategies, we can see a small difference between B-c and B-u. This may be partially caused by the randomness of the read selection process, but also by the additional computation needed for the B-c strategy compared to B-u, since it counts the coverages from the longer contigs.

If we look at the difference between D-c and D-u, it is smaller than for B-c and B-u, even though the strategies work on the same principle. However, compared to the B strategies, the D strategies are deterministic, so the difference is not influenced by the random choice of the reads. We can also see that the time for the D strategies has dropped after the 15th iteration. D strategies strongly prefer longer reads over the short ones, while still maintaining the desired coverage threshold. The difference in time is caused by having a smaller number of longer reads, which means faster sampling and Minimap2 alignment compared to the case if we have many short reads.

Basecalling The input to our pipeline are basecalled data. Therefore, we also have to consider the time needed for basecalling when evaluating real-time performance. In order to compare our results with the sequencing run speed, we have to sum the time spent by basecalling with the pipeline running times. In each iteration, the new data for the batch have to be basecalled. We have estimated the basecalling time by running the GPU version of the Guppy basecaller by ONT [18] (with parameter *num_ callers* set to 2 and *cpu_ threads_ per_ caller* set to 4) on a portion of our data (16000 randomly selected reads). The running time was approximately 66 seconds and the number of bases in the resulting file was 56 Mbp. The resulting estimate is that the Guppy basecaller can process approximately 850 000 bases per second. To estimate basecalling time for a batch, we have selected ten random files from the run with the total number of approximately 69.6 Mbp. Based on our measurements, we estimate the running time for basecalling of a single batch to approximately 82 seconds. The basecalling time of a batch does not depend on previous iterations in any way; rather, it depends on the number and lengths of the *new* reads in the batch. Therefore, for simplicity, we will consider the 82 seconds as a basecalling time constant for each batch. If the user does not have a GPU, there are multiple options. We have also calculated an estimate for basecalling with deepnano-blitz [4], which runs on CPU. With the *network-type 56* deepnano-blitz model, a batch can be basecalled in 98 seconds on CPU using 8 threads.

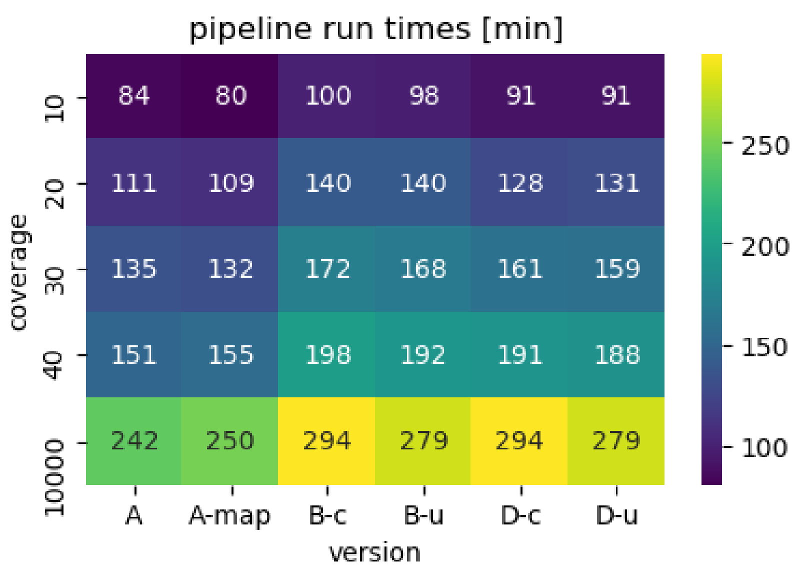


Figure 4.11: Total running times of the pipeline including basecalling in minutes.

Figure 4.11 shows the total running times in minutes including estimated basecalling time with the Guppy basecaller (30.8 min), for all combinations of strategies and coverage thresholds. As mentioned earlier, the A-map strategy is a little faster than A for lower coverage thresholds (10, 20 and 30), but slower for the greater thresholds. The B-strategies are the slowest, and the D strategies are in the middle between the A and B strategies.

In case of our data set, with sequencing time of 313 minutes in total, all strategies including the straightforward strategy with no sampling would be fast enough to provide real-time monitoring. However, Figure 4.12 shows that for longer runs, we would not be able to provide the results in real time. The coverage threshold also influences the time delay between the time of the data availability for a batch during the run and the time when the analysis of the batch is completed (for higher thresholds, the delays are longer, because we are processing a larger amount of data).

Until we reach the threshold (for any sampling strategy), the time for an iteration grows linearly. After the threshold is reached, the time for an iteration becomes nearly constant, and hence, the cumulative running time after the threshold is reached is linear. The slope of the resulting linear function depends on the coverage. Figure 4.13 demonstrates this on the B-c strategy, which is the slowest strategy we have.

The importance of the sampling threshold is to ensure that at some point, the set of reads to be processed in each iteration will reach its potential maximum and will only grow if the assembly will change dramatically.

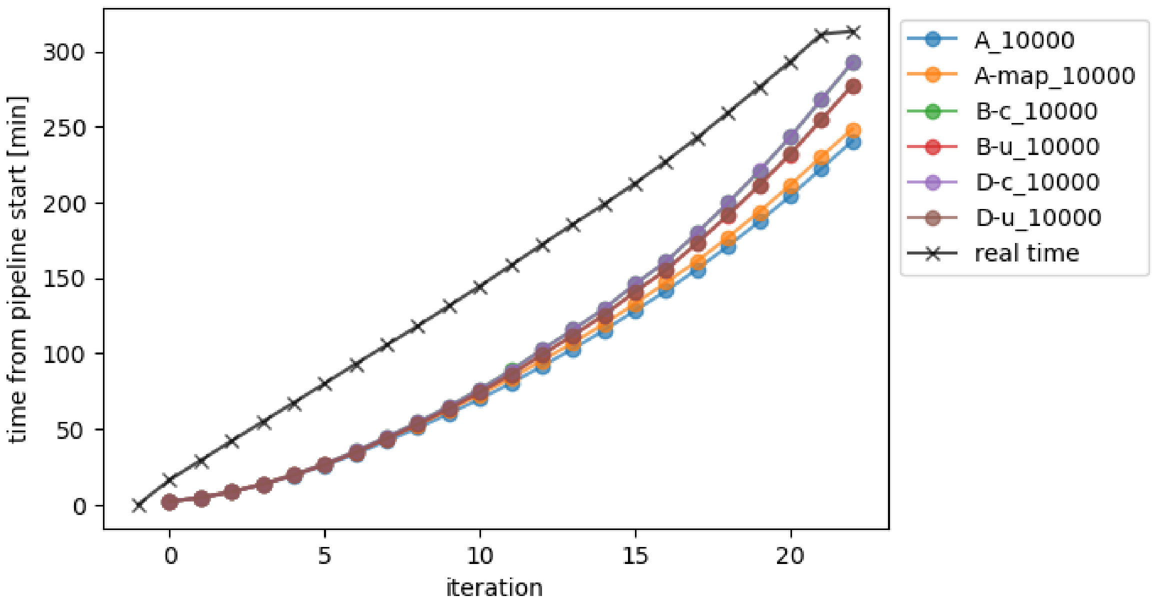


Figure 4.12: Cumulative running times of the pipeline for coverage threshold 10000 including basecalling (per iteration). Since the time for iterations, in this case, grows linearly, the cumulative time trend is a power function. For a longer run, we would not get the results in real time. The times for B-u are almost identical to the times for D-u, similarly for B-c and D-c.

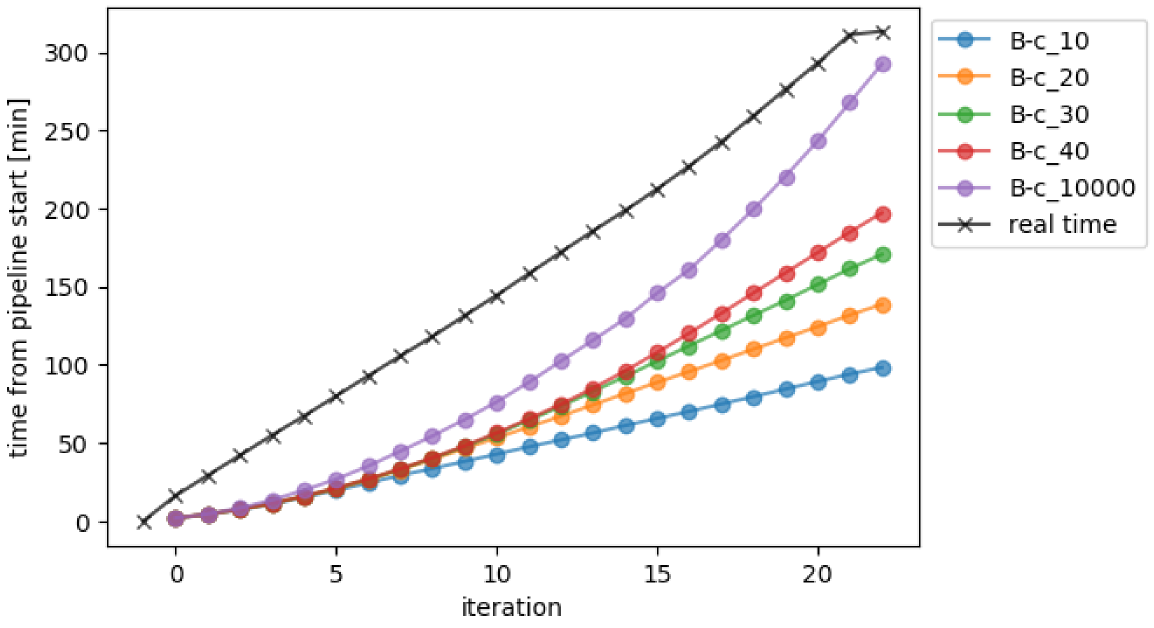


Figure 4.13: Cumulative running times of the B-c sampling strategy for different coverage thresholds including basecalling (per iteration). We can see that the trends are linear after the thresholds for each coverage are reached.

4.6 Batch size

In the previous Sections, there were results achieved with the batch size of 10 files, each containing 500 reads. Since the batch size influences the running time and the results, we have also measured the times and evaluated the scores for batch size 20 and 30x coverage threshold.

Figure 4.14 shows the running times for each of the 12 iterations (indexed from 0). We can see that after the threshold was reached, the times for an iteration are higher compared to times for batch size 10 in Figure 4.10. However, the differences are less than 20% of the batch size 10 run times, so increasing the batch size can be helpful if the user wants to run the pipeline faster and does not need the output frequently. The total running time, including base calling approximation (calculated similarly as for batch size 10), was highest for the B-c strategy (118 minutes), followed by B-u (116 minutes), and the fastest were A-map and A (approximately 95 minutes).

The corrected N50 scores are shown in Figure 4.15. We can see that in this case, the A and D strategies perform better than the B strategies. Compared to the results for batch size 10, the results for the D strategies have improved, because with a bigger batch, we are adding more coverage to the data that Miniasm can use in its filtering. This is advantageous if there are long reads in the sample, since they have a better chance of being covered by the reads in the batch and used as whole sequences (not only as fragments with higher coverage) in the assembly graph in Miniasm. Additionally, the samples are selected from bigger data sets. Another factor compared to the batch size 10 runs is that using a larger batch influences the order in which we give the reads to Minimap and Miniasm, which can have some impact on the results. Also, the assemblies that are built in the iterations at the beginning of the run are less fragmented compared to the smaller batches case, which allows us to create a more representative sample earlier in the run.

The detailed overview of the assembled contigs is shown in Figure 4.17. We can see that with the bigger batch size, the assemblies for A and D strategies are less fragmented compared to the B strategies. Figure 4.16 shows the numbers of assembled contigs per iteration. If we compare them to the runs with smaller batch size, we can see that for the runs with larger batches, there is a smaller difference between the A strategies and the others. Also, we can see that the numbers of contigs at the end of the run are smallest for the D strategies, similarly to the batch size 10 runs.

If we look at the percentages of the reference genome covered by the assembly, we will see a similar trend as for batch size 10. For the completeness of the analysis, the plot is provided in Figure 4.18. After the fourth iteration, the assemblies already cover more than 98% of the reference genome for all of the strategies. The maximum values achieved in the iterations for all of the strategies are more than 99.5% for each.

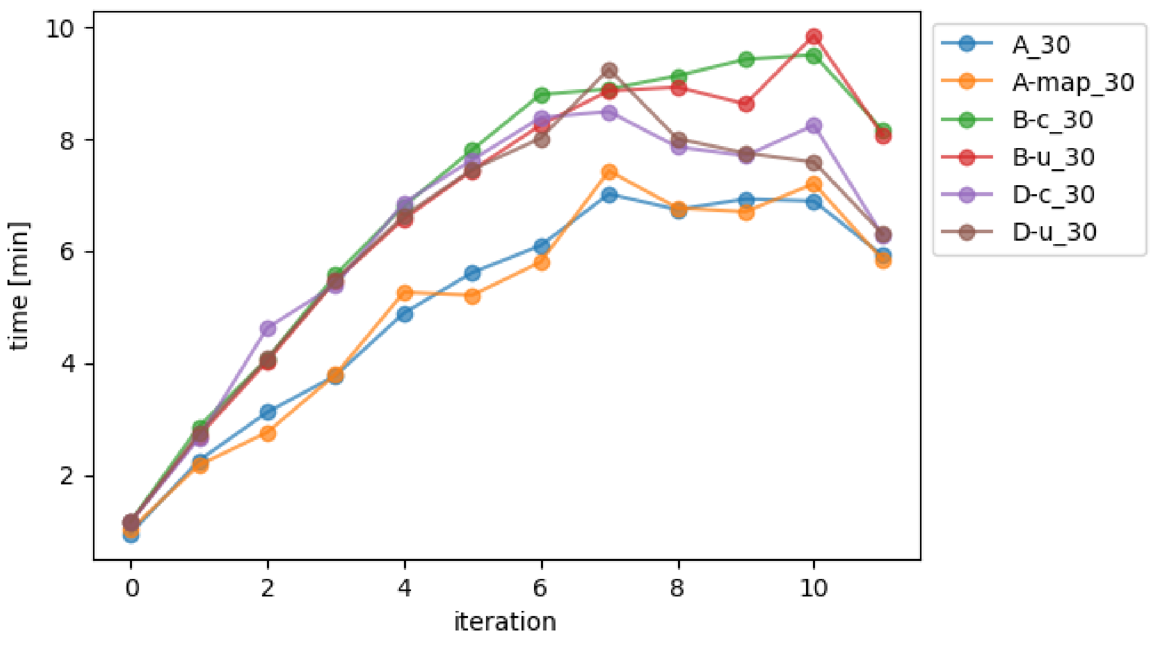


Figure 4.14: Running times per iteration for the different sampling strategies for coverage threshold 30x and batch size 20.

We can also notice that the difference between the A strategies and the others is lower in this case, compared to the runs with batch size 10.

The results suggest that larger batch sizes can shorten the running time and improve assembly quality. However, they also cause longer delays between the time when reads are produced and when the results are available, which introduces a time-quality trade-off in the choice of the batch size. The resulting percentages of assembly covered by the sample are similar for the different batch sizes, but the assemblies are more fragmented with the smaller batch size.

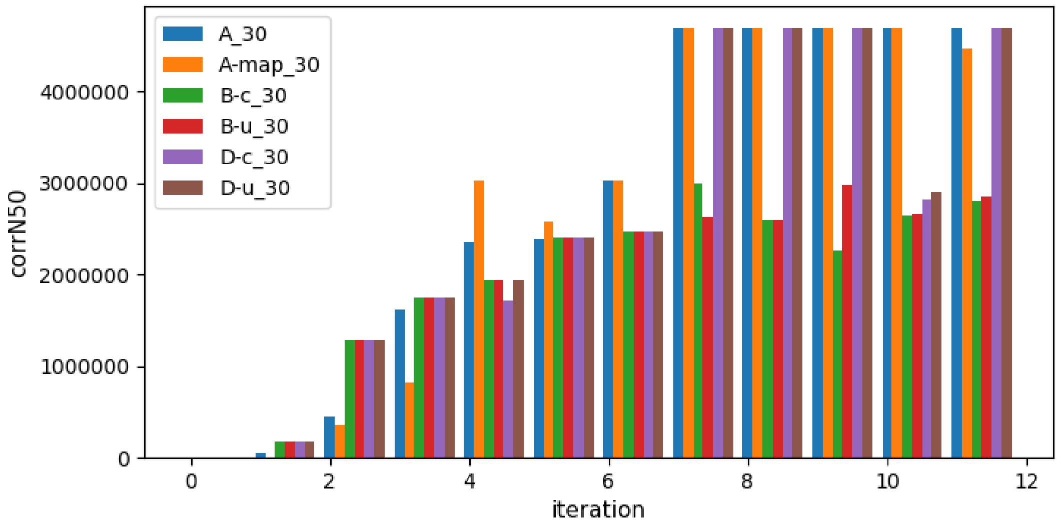


Figure 4.15: Corrected N50 score of the assemblies for the runs with 30x coverage threshold and batch size 20 (per iteration).

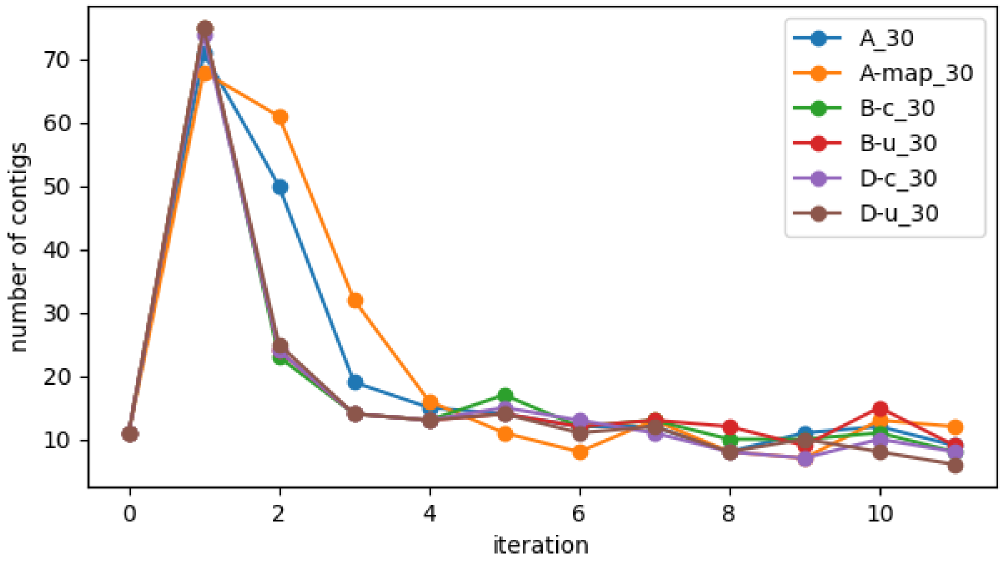
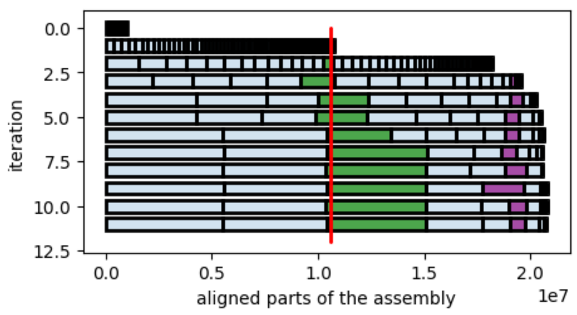
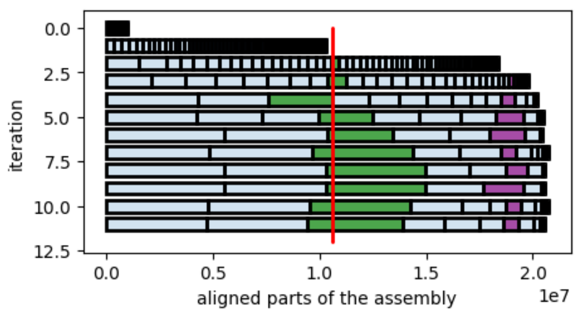


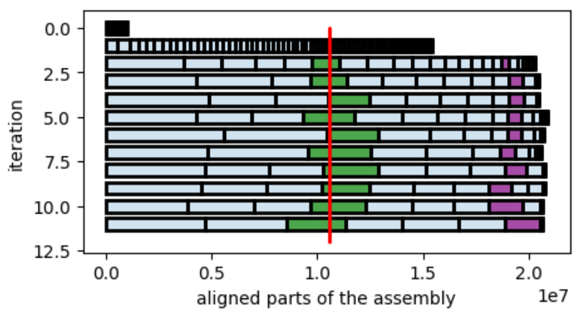
Figure 4.16: Number of contigs per iteration for 30x threshold runs with batch size 20.



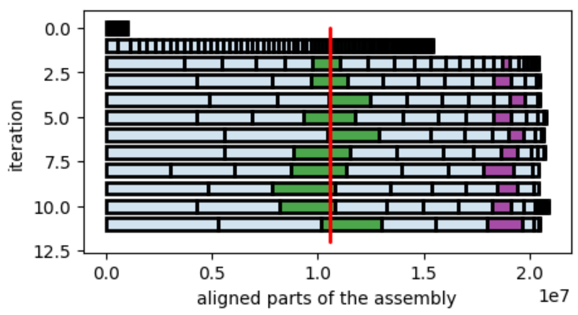
(a) A



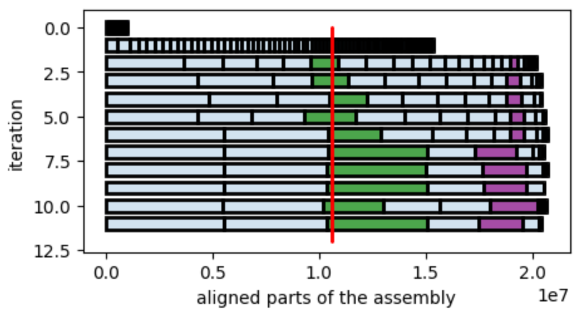
(b) A-map



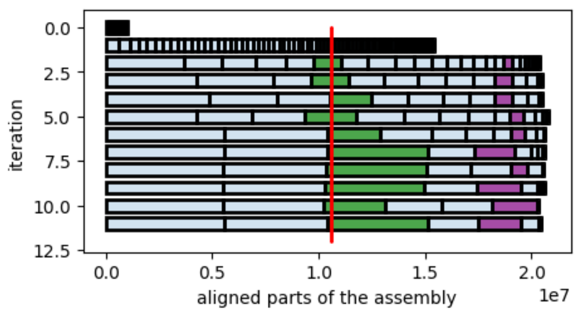
(c) B-c



(d) B-u



(e) D-c



(f) D-u

Figure 4.17: Assemblies for iterations in runs with 30x coverage threshold and batch size of 20 files, corrected N50 and N90 score visualization. The red line corresponds to half of the reference length. The rectangles in each row are the lengths of alignments of the contigs from assembly to the reference genome for one iteration. The green rectangles correspond to corrected N50 score; the purple rectangles correspond to corrected N90 score.

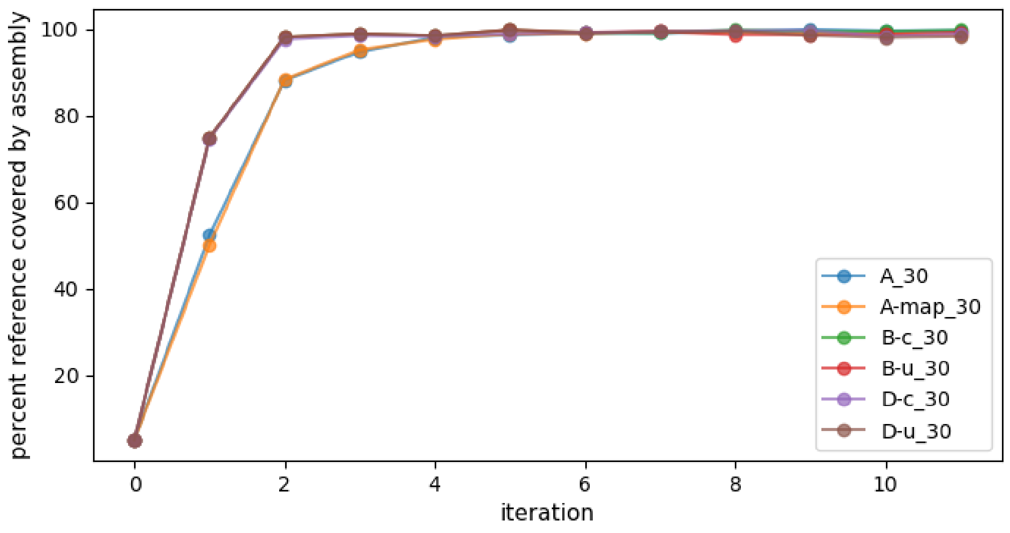


Figure 4.18: Percentages of reference genome length covered by assembly for 30x coverage threshold and batch size 20 (per iteration). We can see that the trend is similar to the trend in runs with batch size 10.

Chapter 5

Future work

We have built a real-time pipeline for genome assembly of the nanopore sequencing reads. It uses Minimap2 and Miniasm for the overlap and assembly tasks. They are used as they are, without any significant modifications. This chapter discusses some possible improvements and further options for future work.

5.1 Using the reads selected by Miniasm

Using the knowledge of the Miniasm algorithm, we could further improve the quality of the assembly. The pipeline repository contains a commented and slightly modified version of Miniasm. The purpose of our modification is that it can output both the GFA result and the list of reads Miniasm directly uses for the assembly.

Using those reads only instead of the sample is not plausible because of the Miniasm's coverage-based filtering step. The problem is that the subset of reads that passes the coverage-based filtering would not pass the filtering phase for the second time without the additional reads that were removed in the filtering phase. We can not remove this step, because it is a crucial part of the Miniasm algorithm – the parts of the reads that pass the filtering are directly used in the assembly graph and in the resulting assembly. However, it may be helpful to ensure that the reads used for the assembly will be included in the sample.

5.2 The order of the reads

From the analysis of our pipeline results, it seems that it may be beneficial to investigate the impact of the different orderings of the reads for Minimap2. In some cases, the different order of the reads can result in different mappings and a better assembly, since the result of Minimap2 depends on the order of the reads.

For example, when we ran Minimap2 on all the reads from the run in the order they were given by the basecaller, the number of alignments in the PAF file was slightly lower (65594967) than in case when the reads were ordered by time (65595019). As expected, also the resulting assembly from Miniasm was different in the second case (8 contigs from Table 4.2), compared to the first case (9 contigs, three of them short, and the long contigs shorter than the ones from the assembly with time-sorted reads).

If we sort the reads by length and give them in this order to Minimap2, the number of alignments is even higher (65904235), and the resulting assembly has eight contigs with lengths 5569095, 4861634, 4692711, 2784991, 2655155, 131920, 86677 and 47368 for the contig that Miniasm marked as circular. The first five contigs in order by length are longer than the contigs in Table 4.2 for the case when the reads were sorted by their sequencing time.

Both Minimap2 and Miniasm are deterministic, but the order of the reads matters. This fact can be considered and used (not only) in the dynamic assembly process. It is questionable, whether some specific order of the reads is better in general, since we have tested this on one dataset only. Further investigation would be necessary to examine and compare the resulting assemblies on multiple datasets and properly reason why some order may be better than others in general in context with Minimap2 heuristics and the use of the Minimap2 result in Miniasm, if some read order should be preferred.

5.3 Speeding up Minimap2 all-vs-all alignment

To further improve the running time of the pipeline, we could use the result from Minimap2 all-vs-all read alignment from the previous iteration to build a new one. If we have an alignment of set S_i from the i-th iteration, we can re-use it for building a new alignment for set $S_{(i+1)}$ for $(i+1)$-th iteration. If $S_i \subseteq S_{(i+1)}$, we already have the alignments of reads from S_i to S_i. Then we can align the reads from set $T := S_{(i+1)} - S_i$ to $S_{(i+1)}$, reads from $S_{(i+1)}$ to T and reads from T to T and merge the outputs with S_i vs S_i alignment. If $S_i \nsubseteq S_{(i+1)}$ we can do a similar procedure with $S_i \cap S_{(i+1)}$ using a filtered output from Minimap2 for the i-th iteration. Some further improvements could also be possible if we would save the minimizers for the reads from Minimap2 to use them in the next iteration. However, there are some other factors that should be considered for this approach, some of them related to the previous section.

5.4 Different assemblers

The sampling strategies we have proposed are not limited to the Miniasm assembler. We have decided to use Miniasm, because, in this case, the overlap phase is strictly separated, as opposed to other assemblers. Also, other assemblers typically implement the consensus or error correction step, which is slow and unnecessary for our task. It could be possible to use some other assemblers for this purpose as well, but it might be necessary to modify their source code to exclude some unnecessary computing.

An example of such assemblers is Flye [10] or Canu [11]. They both use Minimap2 for alignments in their code, so to use our pipeline with the assemblers, we can remove the all-vs-all Minimap2 alignment from our pipeline. Also, we would have to remove the consensus step from the source code of the assembler, because it is too slow and not that important for the real-time assembly.

5.5 Limitations

The contiguity of the assembly is influenced by various factors that are hard to predict. A small change in the set of the reads or even in their order can cause that the contig that was built from a smaller subset of reads will not be built from its superset or the same set if we provide the reads to the assembler ordered in a slightly different way. The heuristics used by Minimap2 and Miniasm (and also by other commonly used assemblers, since they use Minimap2 for overlaps) are not very robust. The causes of specific differences between the results can be investigated, but such investigations do not give us much new information, since they can not be generalized. Some issues can be addressed, but they are hard to resolve in general (for example, how to handle the chimeric reads). Therefore, the information about the contiguity of the assembly in the real-time monitoring has to be interpreted with respect to this limitation. For this purpose, we may want to provide the information about the best assembly achieved in the run to the user, instead of the per-iteration results. However, also the definition of the *best* assembly is unclear, since the problem of genome assembly itself is not properly defined in theory – it is a result of multiple heuristic approaches applied to the sequencing data.

5.6 User interface

The pipeline creates an HTML file for each iteration. It contains the numeric statistics and plots. The pipeline repository contains a script that runs a simple HTTP server that provides a convenient way to display the results in HTML format over the network. The user can navigate between the results from different iterations using the links in the HTML files. Since creating the plots in matplotlib might be time-consuming for large amounts of data, it would be beneficial to implement the user interface using some web framework as a separate module, saving some time in the pipeline and providing more options for the user.

Conclusion

We have investigated the possibilities of designing algorithms that can perform and dynamically update the partial sequencing assembly from the sequencing reads in real time. However, the underlying data structures of the algorithms are highly optimized, the data processing in those algorithms depends on various heuristics that are necessary for correct assembly, and they were not designed to be extensible dynamically.

Therefore, we have designed, implemented, and tested an approach that maintains and dynamically updates a representative sample of the sequenced reads during the sequencing run. The amount of data varies for different contigs in the genome, so we consider this information when selecting the sample. With our sampling approach, we can limit the amount of data processed in each iteration, which ensures that the time and resource requirements for an iteration remain constant after the assembly covers most of the genome of the sequenced organism and a certain coverage of the assembly is reached. This allows the assembly and analysis of the data in real time, while the new data from the run can further improve the assembly. We have shown that if we maintain the sample based on per-contig coverage information, the resulting assembly is similar in the most commonly used metrics compared to the Miniasm assembly on the whole dataset. Since the assembly depends not only on the set of the reads in the sample but also on their order and other factors, the comparison had some limitations.

We have tested our implementation on small (yeast) genomes. With some additional improvements mentioned in the last chapter, the pipeline could also be used for larger genomes, which is a subject of future work.

Bibliography

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J. Mol. Biol.*, 215(3):403–410, October 1990.

[2] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.*, 25(17):3389–3402, September 1997.

[3] Konstantin Berlin, Sergey Koren, Chen-Shan Chin, James P Drake, Jane M Landolin, and Adam M Phillippy. Assembling large genomes with single-molecule sequencing and locality-sensitive hashing. *Nat. Biotechnol.*, 33(6):623–630, June 2015.

[4] Vladimír Boža, Peter Perešíni, Broňa Brejová, and Tomáš Vinař. DeepNano-blitz: a fast base caller for MinION nanopore sequencers. *Bioinformatics*, 36(14):4191–4192, 05 2020.

[5] Broňa Brejová and Tomáš Vinař. *Metódy v bioinformatike.* Knižničné a edičné centrum, Fakulta matematiky, fyziky a informatiky, Univerzita Komenského, 2011, 2015. `https://compbio.fmph.uniba.sk/vyuka/mbi/images/e/e1/Skripta-2015-10-01.pdf`.

[6] James M. Heather and Benjamin Chain. The sequence of sequencers: The history of sequencing dna. *Genomics*, 107(1):1–8, January 2016.

[7] Viktória Hodorová, Hana Lichancová, Stanislav Zubenko, Karolina Sienkiewicz, Sarah Mae U Penir, Philipp Afanasyev, Dominic Boceck, Sarah Bonnin, Siras Hakobyan, Urszula Smyczynska, Erik Zhivkoplias, Maryna Zlatohurska, Eugeniusz Tralle, Alina Frolova, Leszek P Pryszcz, Broňa Brejová, Tomáš Vinař, and Jozef Nosek. Genome sequence of the yeast saprochaete ingens CBS 517.90. *Microbiol. Resour. Announc.*, 8(50), December 2019.

[8] A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, November 1962.

[9] W. James Kent. Blat—the blast-like alignment tool. *Genome Research*, 12(4):656–664, March 2002.

[10] Mikhail Kolmogorov, Jeffrey Yuan, Yu Lin, and Pavel A. Pevzner. Assembly of long, error-prone reads using repeat graphs. *Nature Biotechnology*, 37(5):540–546, Apr 2019.

[11] Sergey Koren, Brian P. Walenz, Konstantin Berlin, Jason R. Miller, Nicholas H. Bergman, and Adam M. Phillippy. Canu: Scalable and accurate long-read assembly via adaptive k-mer weighting and repeat separation. *Genome Research*, 27(5):722–736, Mar 2017.

[12] Heng Li. Miniasm GitHub repository. `https://github.com/lh3/miniasm`.

[13] Heng Li. Minimap GitHub repository. `https://github.com/lh3/minimap`.

[14] Heng Li. Minimap2 GitHub repository. `https://github.com/lh3/minimap2`.

[15] Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 03 2016.

[16] Heng Li. Minimap2: Pairwise alignment for nucleotide sequences. *Bioinformatics (Oxford, England)*, Sep 2018.

[17] Gene Myers. Efficient local alignment discovery amongst noisy long reads. In Dan Brown and Burkhard Morgenstern, editors, *Algorithms in Bioinformatics*, pages 52–67, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[18] Oxford Nanopore Technologies. Oxford Nanopore Technologies website. `https://nanoporetech.com`.

[19] Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, December 2004.

[20] Ivan Sovic, Mile Sikic, Andreas Wilm, Shannon Nicole Fenlon, Swaine Chen, and Niranjan Nagarajan. Fast and sensitive mapping of error-prone nanopore sequencing reads with graphmap. *bioRxiv*, 2015.

Appendix A: Source code

This thesis includes an electronic attachment containing the source code of the dynamic assembly pipeline, which is also available as a GitHub repository:
`https://github.com/janka000/dynamic-assembly-pipeline`