Optical maps in guided genome assembly

Miika Leinonen

Helsinki September 9, 2019

UNIVERSITY OF HELSINKI

Master's Programme in Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Koulutusohjelma — Studieprogram — Study Programme Study Programme in Computer Science

Tekijä — Författare — Author
Miika Leinonen

Työn nimi — Arbetets titel — Title

Optical maps in guided genome assembly

Ohjaajat — Handledare — Supervisors

Leena Salmela and Veli Mäkinen

Työn laji — Arbetets art — Level Master's thesis	Aika — Datum — Month and year September 9, 2019	Sivumäärä — Sidoantal — Number of pages 41 pages + 0 appendices

Tiivistelmä — Referat — Abstract

With the introduction of DNA sequencing over 40 years ago, we have been able to take a peek at our genetic material. Even though we have had a long time to develop sequencing strategies further, we are still unable to read the whole genome in one go. Instead, we are able to gather smaller pieces of the genetic material, which we can then use to reconstruct the original genome with a process called genome assembly. As a result of the genome assembly we often obtain multiple long sequences representing different regions of the genome, which are called contigs. Even though a genome often consists of a few separate DNA molecules (chromosomes), the number of obtained contigs outnumbers them substantially, meaning our reconstruction of the genome is not perfect. The resulting contigs can afterwards be refined by ordering, orienting and scaffolding them using additional information about the genome, which is often done manually by hand. The assembly process can also be guided automatically with the additional information, and in this thesis we are introducing a method that utilizes optical maps to aid us assemble the genome more accurately. A noticeable improvement of this method is the unification of the contigs, i.e. we are left with fewer but longer contigs. We are using an existing genome assembler called Kermit, which is designed to accept genetic maps as auxiliary long range information. Our contribution is the development of an assembly pipeline that provides Kermit with similar kind of information via optical maps. The initial results of our experiments show that the proposed genome assembly scheme can take advantage of optical maps effectively already during the assembly process to guide the reconstruction of a genome.

ACM Computing Classification System (CCS):
Applied computing → Molecular sequence analysis
Theory of computing → Pattern matching

Avainsanat — Nyckelord — Keywords
genome assembly, optical map

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Thesis for the Algorithms study track

Contents

1 Introduction **1**

2 Background **2**

2.1 Genome 2

2.2 DNA sequencing 3

2.3 Genome assembly strategies 4

2.4 Supplementary long range data 7

3 Optical map guided genome assembly **9**

3.1 Data: reads and optical maps 9

3.2 Error correction 11

3.3 Pre-coloring contig assembly 14

3.3.1 Read aligning 14

3.3.2 Assembly graph and contig generation 17

3.4 Optical map mapping 20

3.5 Contig coloring 23

3.6 Read coloring 24

3.7 Post-coloring contig assembly 27

4 Experiments and results **29**

5 Conclusions **35**

References **40**

1 Introduction

Inspecting the genome can bring us a great number of benefits. For example, it gives us insight into different diseases allowing us to develop ways to combat them. Taking a look at a person's genetic material can help doctors to personalize medication for them. We can also explore the genetic diversity of a population of endangered species, which can help us come up with an effective conservation plan. The possibility to inspect and modify the genome also provides a host of applications in biological engineering. All in all, there are many ways to improve our, and other species' lives by exploring the contents of our genomes.

The first methods to read small parts of the genome were developed in the 1970s. Since then, more advanced technologies have emerged and we are able to read even longer sections of the genome faster than before. Unfortunately, we are unable to read the whole genome in one go; we only have the ability to collect shorter subsections of it, called reads. This leads to the problem of genome assembly, a way to automatically rebuild the genome from the smaller parts of it that are available to us.

There already exists a variety of genome assembler programs which take in reads and try to reconstruct the genome as well as possible. When working with a completely new genome, you might not have a relevant reference genome on hand. This kind of assembly is called *de novo* assembly. It is still possible to guide a *de novo* assembly through other means besides a ready genome we can refer to. In this research project we are utilizing one of such assemblers, called Kermit [WRS18].

Kermit is designed to accept long range information about the reads provided to it, so it can produce a more accurate representation of the genome. The long range information is given in the form of genetic maps. Other kinds of auxiliary data are also available, and in this thesis we will examine how optical maps can be used to guide genome assembly process. Typically optical maps have been used after the initial genome assembly is complete to refine the final reconstruction of the genome. Commonly this is done manually, which takes a lot of time and effort, and is susceptible to human error.

Automated systems that involve optical maps already during the assembly process have been researched and developed to some extent, but the results are still quite slim. As an example, AGORA [LGM+12] is an assembler program that can utilize optical maps, but it was only tested with error free reads of very small bacterial genomes. Nevertheless, AGORA got positive results from using optical maps. Another example program called KOOTA [ASP$^+$17] also takes advantage of optical maps during assembly. While KOOTA did not perform competitively compared to the other current assemblers, it demonstrated that optical maps can be used to improve specific phases of the assembly process.

In this thesis we propose a genome assembly pipeline that provides long range information through optical maps to guide the long read genome assembly of Kermit. We determined that the usage of optical maps during the assembly can produce higher quality contigs compared to a similar unguided *de novo* assembly. Our method is also applicable with larger genomes than bacterial ones. While the initial results suggest our pipeline can use optical maps efficiently during the assembly, there is still room for possible improvements which could present themselves after more experimentation. The results and ideas for future work are discussed further in Section 5.

2 Background

Before diving into our genome assembly strategy, some background information about the subject is required. First we will introduce the genome; what it is and what kind of structure it has. Next we will discuss how we can get information about the genome, and why genome assembly is needed. At the end, some genome assembly strategies utilized in our assembly pipeline are introduced.

2.1 Genome

The goal of genome assembly is to figure out the structure of the genetic material of an organism. The genome contains all the information on how the organism is built and maintained on cellular level. The genome can consist of a single or multiple *chromosomes* depending on the complexity of the species. A chromosome is a deoxyribonucleic acid (DNA) molecule. Some viruses have their genetic information encoded in ribonucleic acid (RNA) molecules instead of DNA molecules. Often chromosomes are linear molecules, but some species can also have circular chromosomes.

Regardless of the number of chromosomes or their shape, a DNA molecule is composed of two strands of small building blocks called nucleotides. The strands are intertwined together forming the characteristic double helix structure of a DNA molecule. There exists four different nucleotides, which are differentiated by their base components; adenine, cytosine, guanine and thymine. The bases, and the corresponding nucleotides, are simply denoted by the first letters, A, C, T and G. A strand of a DNA can thus be represented by a string of these letters. Each base of a strand is paired to another base of the other strand. Generally, there exists two kinds of pairings, adenine pairs with thymine and cytosine pairs with guanine. Because of this redundant structure, we can deduce the base sequence of a strand if the other strand is already known. Two paired strands of a chromosome are also called complementary strands. A DNA sequence length is measured in base pairs, meaning the number of base pairs in two complementary sequences. As a side note, an RNA molecule is similar to a DNA molecule, except it has only one strand, and thymine nucleotide is substituted with another one, called uracil, U. However, we are only focusing on assembling genomes with DNA in this research.

2.2 DNA sequencing

Figuring out the structure of the genome can give us very useful insight into different species and even individuals. DNA sequencing, the methods for discovering the nucleotide sequences of DNA molecules, got its start over 40 years ago in the 1970s. Two main methods defined this first generation of sequencing, Sanger sequencing [SC75] and Maxam-Gilbert sequencing [MG77]. These first generation technologies were very slow, and sequencing DNA molecules to collect data took a long time. The sequenced DNA segments known as *reads* were also quite short, less than 1000 bp (base pairs) long. For comparison, the human genome is longer than three billion base pairs, and even a simple single chromosome bacteria like *Escherichia coli* has a genome that is around five million bases long.

The second generation improved upon the first one by enabling massively parallel sequencing, raising the throughput dramatically. This is done by randomly fragmenting DNA molecules, and the fragments of multiple DNA molecule samples are then sequenced separately at the same time. This way a great number of reads can be acquired in a reasonable time. As a drawback, the length of the reads ended up being even shorter compared to the first generation methods, which the third generation sequencing technologies try to address. The reads obtained through them are considerably longer, the average length of the reads reaching over 10 kb (kilo base pairs) depending on the technology used. On the down-side, their error rate increases from the typical 0.01-0.10% of the first and second generation technologies to over 10%. The main error types in reads are insertions, deletions, and substitutions, which appear in the read sequences due to the inaccuracy of the current third generation technologies. A review by Kchouk et al. [KGE17] provides a nice summary of different sequencing technologies, their read lengths and error rates you can expect from them.

In our research we are using long reads obtained with third generation sequencers, but even in this case the read lengths are nowhere near the lengths of genomes. This is where genome assembly comes into play; the reads are used to reconstruct the genome they were sequenced from. The reason higher error rate long reads are used is because of repeating regions in genomes. If the reads are shorter than repeating regions, we cannot be sure how genome regions separated by them are connected to each other. Due to the necessity of longer reads, the higher error rate must be take into account and addressed appropriately. We did this by trying to correct the errors in our reads before anything else, which we will go through in more detail in Section 3.2.

The reads we used were generated by third generation Pacific Biosciences SMRT platform. *Arabidopsis thaliana* reads were actual real world reads, and an already assembled genome was used to evaluate our method. Additionally we produced simulated reads using already assembled *Escherichia coli* and *Saccharomyces cerevisiae* genomes. Simulated reads were generated using read simulation program simlord [SKR16], which mimics the PacBio sequencer errors.

2.3 Genome assembly strategies

Genome assembly is sometimes compared to the problem of solving a giant jigsaw puzzle, where sequenced reads are the puzzle pieces. The question now is, how would one start putting these pieces back together to form the original 'picture'. Normally you would find pieces that fit together and start combining them until all the pieces are connected. Genome assembly is performed in a similar manner; the core idea is to find reads that fit together. Due to the random nature of massively parallel shotgun sequencing, the reads come from all over the genome and are of varying length. We often collect a great number of reads, so that they together cover the genome multiple times over. For these reasons, the reads are bound to overlap with each other containing common regions of the genome. This helps us to glue connected reads together appropriately, resulting in a longer and longer DNA sequences, consensus regions called *contigs*, which represent the structure of the original genome.

The first approach that could come into one's mind is to just use this overlap information directly to rebuild the genome. These are the greedy *overlap methods*. The general idea is to compare all reads to each other, detecting all overlap candidates. Then, the highest quality overlap is used to combine two read sequences into a longer sequence. Next the second best overlap is used, and so forth. This is done until the overlap quality drops too low. The quality of the overlap is usually a combination of multiple things, like how long is the overlapping region and how good are the base-to-base matches in it. An example of a more complicated overlap finding tool will be introduced in Section 3.3.1.

The overlap methods are a good starting point, but their weakness becomes apparent when we consider the fact that often genomes contain repeating regions. When the overlapping region between two reads consists of a repeating pattern of the genome, they could be erroneously joined if the overlap quality is very good. The problem arises when these reads are in reality from totally different regions of the genome, but they just happened to contain the same repeating region just in the right way. A simple greedy algorithm is unable to take multiple good candidate overlaps into account and determining which reads to merge while considering the possibility of repeating regions. The more advanced graph based methods are better suited for handling possible repeating regions. They are still very much reliant on overlapping strings of characters, but they bring in the flexibility of graphs that are not so easily available to the greedy approaches.

The natural evolution of the overlap methods are *overlap graphs*. The basic idea is to represent the reads and their relationships (overlaps) with vertices and edges. One way to do this is to let the vertices of the graph represent the reads, which are then connected with edges based on the overlaps between the reads. The graph is traversed to find paths which form long sequences likely to appear in the original genome. In an ideal case there would be a single path for each chromosome in the genome. But because very few things in the real world are perfect, we must decide which of the many different paths in the graph are appropriate approximations for the genome. A common solution is to use just the unambiguous paths of the graph. An unambiguous path is non-branching, meaning all vertices other than the source and sink must have in and out-degree of one. The sequences spelled by these paths are called *unitigs*. Overlapping unitigs of the graph can then be used to calculate contigs as a consensus sequence using multiple sequence alignments. Consensus sequence and multiple sequence alignment will be explained further in the context of read correcting in Section 3.2.

The genome assembly paradigm described above is called *OLC*, which is an abbreviation of the three different phases; overlap, layout and consensus. First the overlaps are found, then the graph is built, and finally the contigs are constructed as consensus sequences. In the last step consensus sequences are carefully calculated to give us the final products of the assembly. The two genome assembly programs we made use of follow this OLC paradigm loosely, skipping the consensus step. Instead, unitigs found during layout step are presented as the final product.

The overlap graphs can become enormous and too complex to build consensus sequences quickly. For this reason a polished version of the overlap graph was developed by Myers [Mye05]. These *string graphs* are obtained from overlap graphs by removing contained read vertices and transitive edges. Our chosen genome assemblers use string graphs, and how they are constructed is discussed in more detail in Section 3.3.2.

Searching for overlaps can be very time consuming with a large number of reads, because you need to compare all reads to each other, at least in the naive implementation. Another graph type used in genome assembly, called *de Bruijn graphs* [PTW01], circumvents this step entirely. A de Bruijn graph is constructed by finding all distinct *k-mers* within your set of reads, and using them as the vertices. A k-mer is simply a substring of length k. Adjacent k-mers in a read overlap so that the $k-1$ -suffix of the previous k-mer is the same as the $k-1$ -prefix of the following one. For example, all 4-mers of sequence 'AGGCTTAC' in order are 'AGGC', 'GGCT', 'GCTT', 'CTTA' and 'TTAC'. If two k-mers in the graph appear back-to-back in a single read, they are connected with an edge.

Again, finding paths within this graph produces longer sequences that are likely present in the genome. Path finding can be handled by trying to search for an *Eulerian path*, a path that uses each edge once. This might not be the most accurate approach, since repeating regions can cause there to be multiple different path options. Another way to handle this is to deploy the same unitig strategy of just finding the unambiguous paths. The de Bruijn graph will be introduced again in Section 3.2 when we talk about the error correction.

2.4 Supplementary long range data

The reads are the minimum needed information about the genome for its reconstruction. However, the bare base sequences of the reads are not the only information we can acquire about the genome. For example, in mate pair sequencing reads are generated so that they appear in pairs, and the approximate distance between the reads in a pair is known. This kind of information about the respective location between the reads can be used to enhance genome assembly process.

In the case of Kermit assembler [WRS18], the location information is provided via *genetic linkage maps*, also known as *genetic maps* or *linkage maps*. As the name suggests, these maps contain information on how different pieces of genetic code are linked with each other. The genome of a sexually reproducing organism is a combination of its parents' genomes. For each chromosome, both parents' corresponding chromosomes are split randomly into small pieces and again recombined randomly to form the child's genome. If two sequences of DNA are close to each other in one parent's chromosome, they are more likely to be inherited together. This is because the closer the two sequences are, the greater the chance that a split does not happen between them, causing them to stay in a same DNA piece in the random recombination phase. In linkage map terms this means that these two DNA regions have a strong link.

What linkage maps contain specifically, is information about *genetic markers* and their relative positions. Genetic markers can be already known *genes* or other pieces of DNA, like *single nucleotide variations*, SNVs. A gene is a basic unit of inheritance, and they can contain the instructions to build a protein or have some other specialized function. SNVs are variations in DNA that appear in a big enough part of a population. For example, most individuals could have nucleotide G in a specific location of their genome, but a notable portion of the population would have nucleotide A in that position instead. Links between genetic markers do not represent their actual distance in a chromosome, but the frequency these markers are inherited together. Still, markers physically close to each other have strong links and markers far away from each other have weaker links. Markers in different chromosomes are not physically connected, and cannot be inherited together as a part of the same DNA sequence, so they remain unlinked.

Kermit uses linkage maps to guide its genome assembly. Our new proposal is to provide location information using *optical maps* instead of linkage maps. Optical map is a set of lengths of subsequent DNA sequence fragments. A DNA sequence

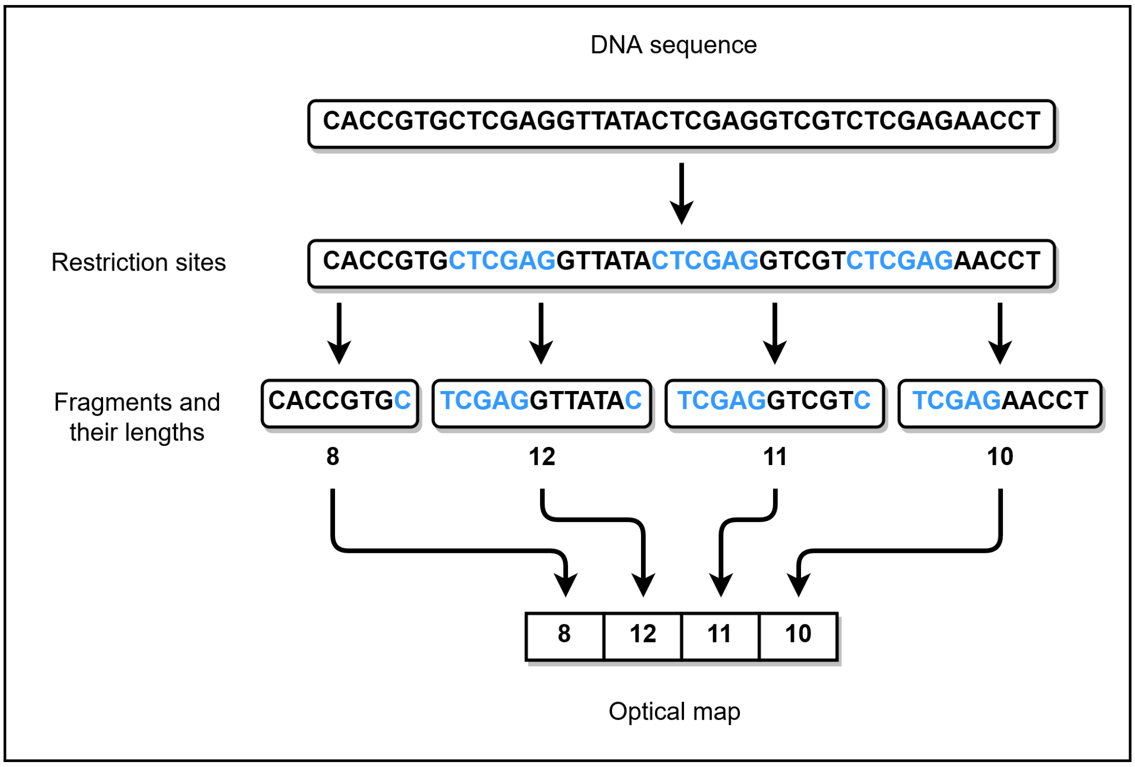
DNA sequence
CACCGTGCTCGAGGTTATACTCGAGGTCGTCTCGAGAACCT
Restriction sites
CACCGTGCTCGAGGTTATACTCGAGGTCGTCTCGAGAACCT
Fragments and their lengths
CACCGTGC
TCGAGGTTATAC
TCGAGGTCGTC
TCGAGAACCT
8
12
11
10
8
12
11
10
Optical map

Figure 1: Example DNA sequence and its optical map corresponding to XhoI restriction enzyme. XhoI recognizes 'CTCGAG' sites and cleaves the sequence after the first C-nucleotides. The lengths of the resulting fragments are then measured to compose the optical map.

can be cut at specific places, *restriction sites*, using *restriction enzymes*. Applying restriction enzyme to a DNA molecule causes it to split into fragments at the corresponding restriction sites. For example, enzyme XhoI recognizes nucleotide sequence 'CTCGAG', and splits the DNA molecule after the first C-nucleotide. This process leaves us with a number of consecutive DNA fragments, whose order is known, and their length can be measured. The measured lengths put together in order gives us an optical map of the DNA sequence. A simplified example of a DNA sequence and its optical map can be seen in Figure 1. The optical map of a genome we want to assemble is usually generated this way in a laboratory environment. In the case we have access to the DNA sequence itself, we can use *in silico* digestion. This means that we can just fragmentize the sequence computationally by finding each instance of a substring corresponding to a restriction enzyme and splitting it at these sites. For example, optical maps for reads and contigs can be acquired this way since their sequences are known.

3 Optical map guided genome assembly

Kermit [WRS18] assembler uses the relative location information of reads to produce contigs that represent the reference genome more accurately. The location information is expressed by coloring the reads, which will be explained in more detail later. Our approach is to get the location information by approximating which part of the reference genome each read represents using optical maps. Our first attempt was to use optical maps of the reads and the reference genome, and map the read optical maps to the genome optical map. Unfortunately due to the short lengths of the reads, their optical maps had very few fragments on average and were impossible to map to the reference reliably. This meant we could not get location information for many of the reads, and they were left uncolored. This seemed to happen even when the reads were fragmented using multiple restriction sites. We tried creating optical maps using three different restriction sites corresponding to enzymes XhoI, EagI and NheI. It appeared that longer sequences were needed for the optical map mapping to be successful.

In our next approach we used the corrected reads directly with a *de novo* assembler to acquire *pre-coloring contigs*, contigs that are assembled using non-colored reads. After this optical maps of the pre-coloring contigs are created and mapped to the reference genome optical map. With optical map alignments we are able to approximate the locations of the pre-coloring contigs within the reference genome and color them accordingly. Most of the contigs could be colored because they are much longer and their optical maps have more fragments compared to the reads. Next the reads are aligned to the pre-coloring contigs. Alignments contain information on how the reads and contigs overlap, like which contig a read aligns with and where in the contig the overlap starts and ends. With this information and colored pre-coloring contigs the reads can be colored. After this the colored reads are ready to be given to Kermit assembler as input. Kermit outputs new *post-coloring contigs*, which are the final product of our guided genome assembly, hopefully representing the reference genome more accurately than the pre-coloring contigs. The whole assembly workflow can be seen in Figure 2.

3.1 Data: reads and optical maps

To assemble a genome, a set of reads covering it is needed. The reads can be produced with sequencing technologies discussed in Section 2.2. A *de novo* assembler

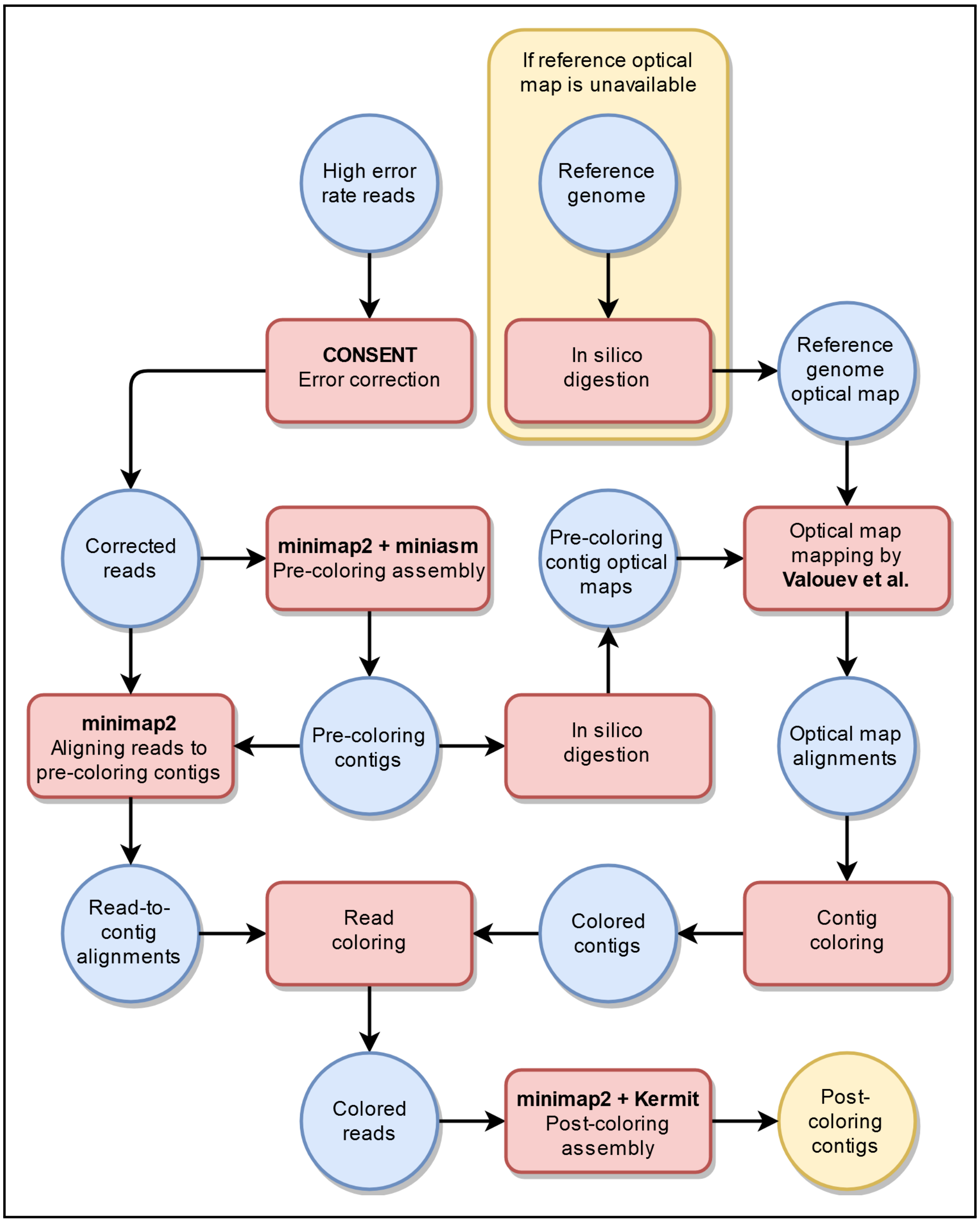
If reference optical map is unavailable
High error rate reads
Reference genome
CONSENT Error correction
In silico digestion
Reference genome optical map
Corrected reads
minimap2 + miniasm Pre-coloring assembly
Pre-coloring contig optical maps
Optical map mapping by Valouev et al.
minimap2 Aligning reads to pre-coloring contigs
Pre-coloring contigs
In silico digestion
Optical map alignments
Read-to-contig alignments
Read coloring
Colored contigs
Contig coloring
Colored reads
minimap2 + Kermit Post-coloring assembly
Post-coloring contigs

Figure 2: Assembly workflow.

can assemble a genome with reads only, but for our guided assembly with Kermit we still need the optical maps of the contigs and the reference genome. Optical maps for the contigs are obtained using *in silico* digestion, and optical map for the reference genome is obtained in a same way or in a laboratory with real DNA molecules and restriction enzymes as discussed in Section 2.4.

3.2 Error correction

We are using third generation long reads that have a high error rate, which causes problems in the assembly process making correcting the reads an important first step. Correcting not only improves *de novo* assembly results, but in our approach it also helps the guided assembly by improving the quality of the *in silico* optical maps. Insertions, deletions and substitutions affect the appearing cut sites, and correcting these errors causes lost cut sites to be restored and false cut sites to disappear.

There are generally two types of error correction approaches for long reads; hybrid and non-hybrid methods. Hybrid methods require short and highly accurate reads in addition to the long reads that we want to correct. The non-hybrid methods work with just long reads, and we decided to take this non-hybrid route to make things more simple. In the hybrid approach, one would of course need two different sets of reads produced by two different sequencing methods.

We decided to use a fairly recent error correction program known as CONSENT [MML$^+$19]. According to the authors CONSENT performs well compared to the other state-of-the-art self-correction methods. One weakness of CONSENT is that its running time is noticeably longer in comparison. Still, we decided this was acceptable because the throughput and resulting error rate were good. Practical reasons were also present, as this was the program that we could easily run on all our data sets without any problems.

CONSENT combines multiple state of the art correction techniques. As the first step it finds overlaps between the reads. By default, CONSENT uses minimap2 alignment program [Li18] to do this. The overlaps are represented with *alignment piles*, which were originally proposed in Daccord correction program [TM17]. Each read is given its own alignment pile. An alignment pile for *template read* A is represented as a set of tuples $(A, R, Ab, Ae, Rb, Re, S)$ where R is a read aligning with A. Ab and Ae are the start and end positions of the alignment in A. Rb and Re are the start and end positions in the aligning read R. S is 1 if A aligns with the

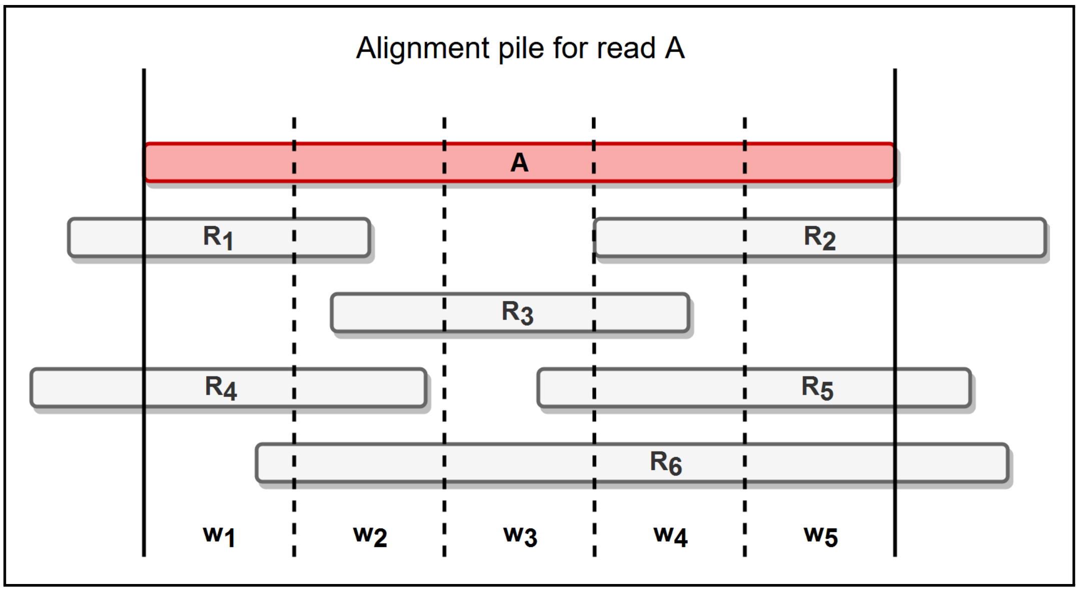

Figure 3: Alignment pile for template read A and its windows. For example, if minimum window coverage is set to four, windows w_1 and w_3 are not corrected by CONSENT. The template read itself is included in coverage counting.

reverse complement of R, otherwise it is 0. All this information is available from the output of minimap2. While it does not necessarily contain the accurate nucleotide level alignment information, the information about the alignment locations is a good starting point. The alignment method of minimap2 is discussed further in Section 3.3.1.

Another idea inspired by Daccord is the usage of alignment windows. Instead of correcting the read as a whole, CONSENT splits it into smaller windows of a set length L. The correcting job is thus split into smaller subproblems that are easier to solve. CONSENT tries to correct a window if its coverage in the alignment pile is high enough, to ensure enough information for correction is available. Figure 3 shows an example picture of a template read and its alignment pile split into five windows.

To correct a window, its consensus is calculated using multiple sequence alignment (MSA) strategy implementation POAv2 [LGS02]. The data structure that is used to hold the alignment information of the multiple sequences is a directed acyclic graph. The graph starts as the sequence of the template read of the window. Each other sequence in that window is then one by one aligned with the graph and the result of the alignment is then added to the graph. This is done with a generalization of the Smith-Waterman algorithm [SW81]. The consensus sequence is extracted from the graph using a voting system, where the bases are chosen by majority vote. If there is a tie, the base from the template read takes priority.

Even though the reads were already split into smaller windows, the resulting sequences are still quite long causing the calculation of the alignments to take a long time. Windows are divided further into smaller colinear segments to help with this problem. These sections are found by searching for an ordered set of k-mers $K = [a_1, a_2, ..., a_{n-1}, a_n]$. These k-mers are called *anchors*, and they must be unique within the sequences. Additionally, they must appear in order in all sequences, even though not all of them must appear in every sequence. Final restriction is that every consecutive pair of the anchors (a_i, a_{i+1}) must be present in multiple different sequences a sufficient number of times. All sequences within a window are then split into smaller segments divided by the anchors. Once all suitable anchors are found, the consensus sequences for the segments between them are produced separately. For a consensus sequence between consecutive anchors a_i and a_{i+1} only sequences that contain both of them are considered when building the MSA graph.

After consensus sequences between anchors are generated, they need to be glued together to get the consensus sequence S for the whole window. Suppose we have a set of anchors $[a_1, a_2, ..., a_n]$, and the consensus sequence between two anchors is $consensus(a_i, a_{i+1})$. Let the consensus sequence for the sequence before the first anchor a_1 be $consensus(prefix)$, and in the same fashion $consensus(suffix)$ for the sequence after the last anchor be a_n. S is now gotten by concatenating the anchors and the results of these smaller consensus problems, like so;

$$\begin{aligned} S = {} & consensus(prefix) + a_1 + consensus(a_1, a_2) + a_2 + ... \\ & + a_{n-1} + consensus(a_{n-1}, a_n) + a_n + consensus(suffix) \end{aligned}$$

Some errors may persist in the consensus sequences of the windows, which CONSENT aims to polish afterwards. A local de Bruijn graph is constructed using specific k-mers of the window. The used k-mers are small and solid, to avoid errors in the graph as much as possible. A solid k-mer is one that appears a sufficient number of times leading us to believe it very likely appears in the original sequence. Short k-mers are used because a single sequencing error affects more and more consecutive k-mers as the value of k increases. After building the de Bruijn graph for a window using the short and solid k-mers, CONSENT searches the consensus sequence of that window for regions that only consist of weak (non-solid) k-mers that are surrounded by few solid k-mers on both sides. These surrounding k-mers are called *anchor k-mers*. The regions surrounded by these anchors are then polished by finding a path in the de Bruijn graph that starts with one of the left side anchor k-mers, and ends with a right side anchor k-mer. If a path like this is found, the weak section is replaced by this path. If no such path exists, the section is left unpolished.

After all windows have had their consensus calculated and polished, it is time to use them to correct the template read. CONSENT does this by aligning the consensus sequences back to the regions in the template read where they originated from using an optimized Smith-Waterman algorithm. Alignment is made faster by forcing a consensus sequence s_i to align to the template read sequence near window w_i. The consensus sequences can end up being longer than the size of the window they are generated for, which causes the aligned consensus sequence to overlap with the consensus sequences that are aligned to the neighboring windows. In these overlapping parts of the alignments, the consensus sequence with most solid k-mers is used over the other one. Finally, the corrected read is obtained by replacing the original read with the concatenation of its aligned window consensus sequences. Figure 4 depicts the steps CONSENT takes to correct the reads.

3.3 Pre-coloring contig assembly

The pre-coloring assembler of our choice is miniasm [Li16], which Kermit assembler is also based on. Before miniasm assembly can be run, we need to use an alignment program that maps reads to each other. A natural choice for this is minimap2 [Li18], a successor to the minimap mapping program introduced in the miniasm article.

3.3.1 Read aligning

Minimap was originally developed as a fast DNA aligner, which later became minimap2 with added functionality, like base-level alignments and support for multiple types of input. Minimap2 uses a common *seed-chain-align* procedure. First specific markers are found in the reference sequences, which are stored in an index. Next same kind of markers (seeds) are searched from the query sequence, and matches (anchors) between the markers of the reference and query sequence are found. Sets of matching colinear markers (chains) are used to determine the likely alignments with the help of base-level alignments.

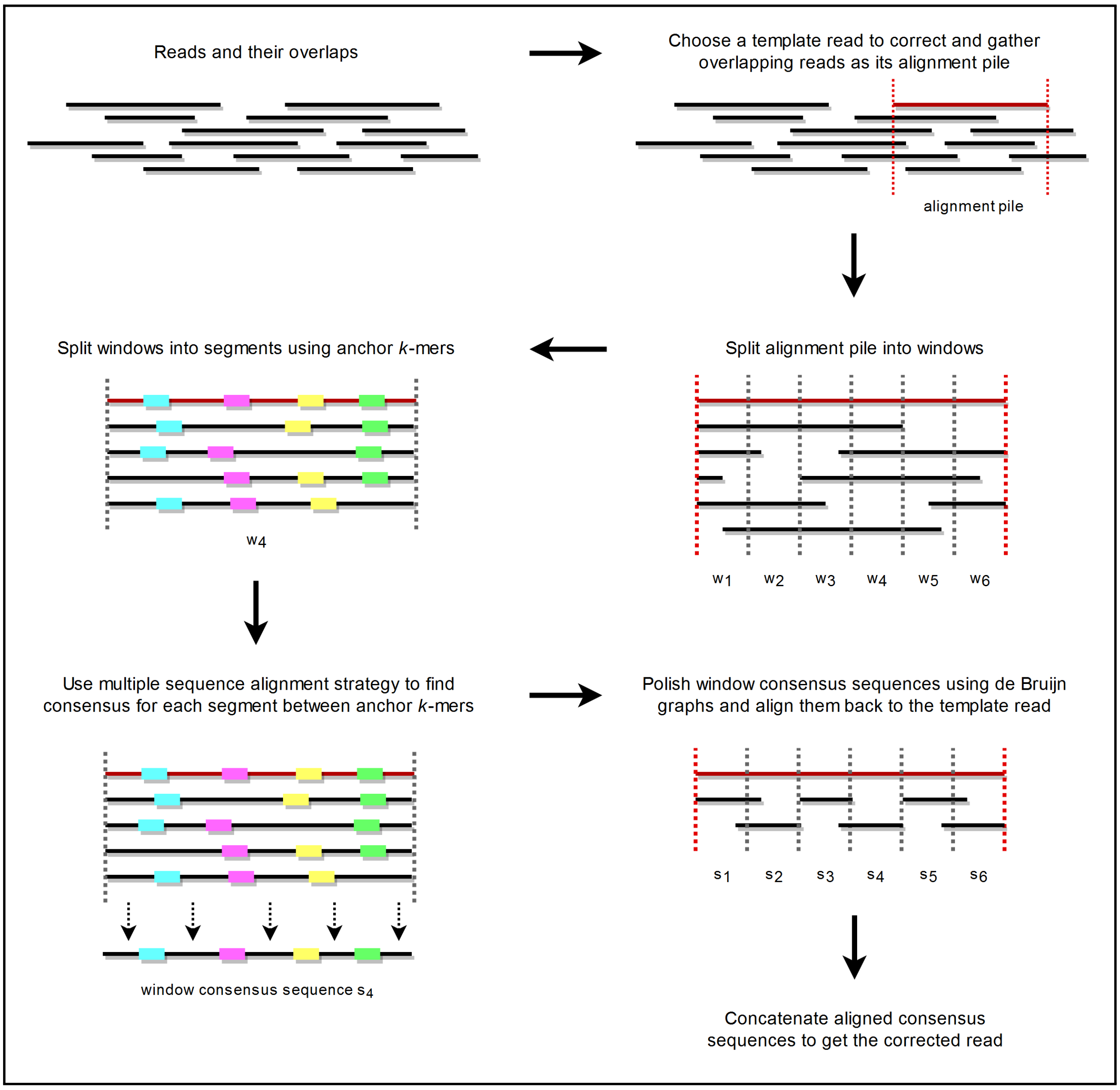
Reads and their overlaps
Choose a template read to correct and gather overlapping reads as its alignment pile
alignment pile
Split alignment pile into windows
w1 w2 w3 w4 w5 w6
Split windows into segments using anchor k-mers
w4
Use multiple sequence alignment strategy to find consensus for each segment between anchor k-mers
window consensus sequence s4
Polish window consensus sequences using de Bruijn graphs and align them back to the template read
s1 s2 s3 s4 s5 s6
Concatenate aligned consensus sequences to get the corrected read

Figure 4: CONSENT read correction steps.

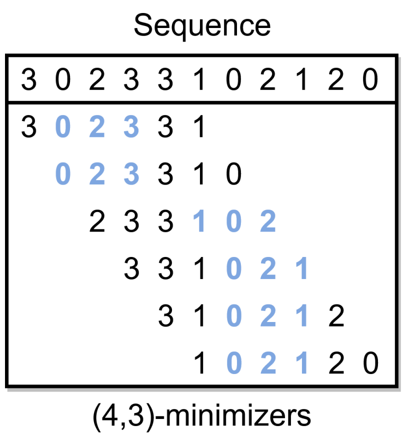

Figure 5: Sequence and its (4,3)-minimizers. There are six windows each having four 3-mers, with the minimizer 3-mer colored blue. For the sake of clarity the characters in this example are integers and the 'hash value' is the number they form. The first window '302331' contains four 3-mers: '302', '023', '233' and '331'. '023' is the smallest value among these 3-mers, so it is chosen as the first minimizer.

Finding all k-mers (with large enough k) in the sequences and using them as markers to match the sequences would be a decent choice, if there were not so many of them. The exponential growth of the number of possible k-mers makes it difficult to be able to store them all in RAM at the same time, making this approach also inefficient time-wise. Instead of using all appearing k-mers, minimap2 finds a smaller subset of them, called *minimizers* [RHH+04]. In simple terms, a minimizer is the smallest k-mer in its surrounding area. By definition (w, k)-minimizer is the smallest k-mer within w consecutive k-mers based on the method used to compare their values. Typically the k-mers are run through a hash function, and the k-mer with the smallest hash value is chosen as the minimizer. The first minimizer is chosen from k-mers starting from indexes $\{1, 2, \dots, w-1, w\}$, the second starting from indexes $\{2, 3, \dots, w, w+1\}$ and so on. Figure 5 shows a small example of a sequence and its minimizers. As we can see from this example, a minimizer k-mer is often the minimizer of several consecutive windows, greatly reducing the number of different k-mers that need to be stored.

The found minimizers are run through a hash function, which are stored as index keys, with its occurrence locations in the sequences as the value. When we try to find an alignment for a query sequence, its minimizers are also calculated. The minimizers are then compared to the stored minimizers of the reference sequences to find long colinear chains of matching regions. Chains indicate a potential alignment, and on top of that minimap2 can do a more rigorous base-level calculations to determine the optimal alignments to the reference sequence to produce as alignment outputs.

3.3.2 Assembly graph and contig generation

Miniasm generates an assembly (overlap) graph based on the alignments received from minimap2. Before generating the graph, miniasm trims reads to reduce the number of artifacts, like adapters and chimeras. An adapter is a short nucleotide sequence that may attach to the ends of the reads as part of the sequencing process. Chimeras are reads that originate from multiple different DNA sequences from different positions of the genome which have merged together. To get rid of these artifacts, miniasm calculates the per-base coverage between aligned reads. Per-base coverage simply means how many times a specific nucleotide of the genome has been sequenced in total among all reads. For each read, miniasm finds and keeps the longest continuous region where all nucleotides have the coverage of three or more, and discards the rest.

The assembly graph is built using the mappings between the trimmed reads. Each mapping is classified as one of the following cases, which are also presented in Figure 6;

- Internal match
- Contained mapping
- Proper mapping

For the following examples let us suppose we have read v of length l_v that has a mapping with read w of length l_w. Let b_v be the beginning position of the mapping in v, and b_w in w. Similarly let the end positions be e_v and e_w.

Internal match happens when the mapping region length is too short to be considered trustworthy. If mapping region is either

- shorter than 1000 bp **or**
- the overhang region (overlapping region between reads that is not part of the mapping region) is greater than 0.8 times the mapping region

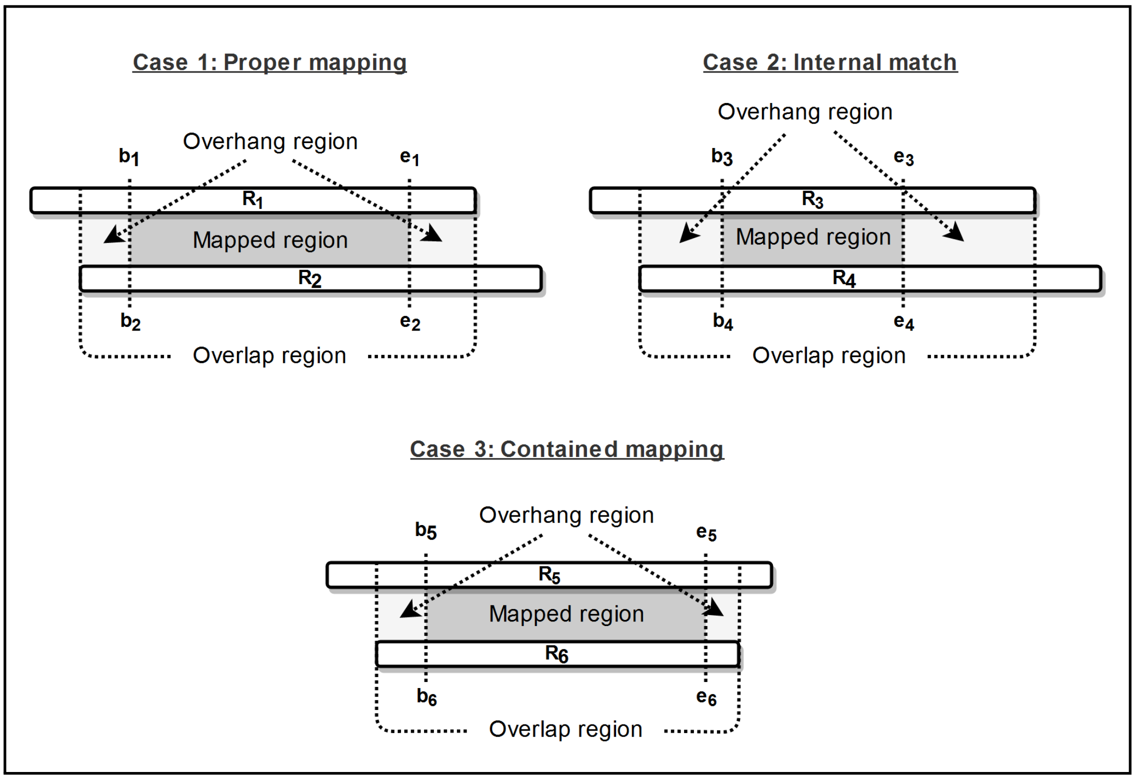

Figure 6: Miniasm mapping cases used to determine which overlaps are used to build the assembly graph. **Case 1:** Proper mapping. Mapped region is sufficiently long and a read is not contained within the other. Mapping is used to build overlap graph. **Case 2:** Internal match. Mapped region is too short to reliably determine if the reads really overlap. Mapping is not used to build overlap graph. **Case 3:** Contained mapping. Read R_6 is contained within read R_5. Read R_6 does not provide more useful information so it is discarded.

miniasm does not use the mapping in building the graph. Length of the overhang region is defined as $OverhangLen = min(b_v, b_w) + min(l_v - e_v, l_w - e_w)$ and length of the mapping as $MappingLen = max(e_v - b_v, e_w - b_w)$. By default miniasm uses $r = 0.8$ as maximum overhang to mapping ratio and $MaxOverhangLen = 1000$ as the maximum overhang length. Using these definitions mapping is considered as an internal match if $OverhangLen > min(MaxOverhangLen, r * MappingLen)$.

Contained mapping happens when a read is wholly contained within another read, making it redundant. Minimap drops these reads from the graph completely. Read v is contained within w if $b_v \leq b_w$ and $l_v - e_v \leq l_w - e_w$. This is also true the other way around, if $b_w \leq b_v$ and $l_w - e_w \leq l_v - e_v$, then read w is contained within v.

Proper mapping is a mapping that is used to build the assembly graph. These are all the mappings that have a long enough mapping region not to fall in internal matches, and do not feature contained reads. Specifically, if the mapping between v and w does not fall in the two previous categories, then the mapping is a proper mapping from v to w $(v \to w)$ if $b_v > b_w$, and mapping $w \to v$ if $b_w > b_v$.

The trimmed non-contained reads are used as the vertices of the graph, and the proper mappings are added to it as edges. Suppose we found a proper mapping $v \to w$. This means we would add an edge between v and w, and the length of the edge would be the length of the beginning section of v that does not overlap with w. The length of the edge is defined as $len(v \to w) = b_v - b_w$. If two reads happen to have multiple mappings between them, only the longest one is used. Because the strands of the reads are unknown, their complements must be added to the graph as well. This also means that each edge should have a complement edge. The length of the complement edge from read $\bar{w}$ to $\bar{v}$ is defined as $len(\bar{w} \to \bar{v}) = (l_w - e_w) - (l_v - e_v)$. These steps make a graph $G = (V, E)$ *Watson-Crick complete*, because the requirements for it; $\forall v \in V, \bar{v} \in V$ and $\forall v \to w \in E, \bar{w} \to \bar{v} \in E$, are fulfilled. Discarding reads that are contained makes the graph also *containment free*. These two properties make the graph a suitable string graph for the assembly.

Miniasm cleans the assembly graph afterwards by removing *transitive edges* [Mye05], *small bubbles* [ZB08] and *tips*. Edge $v \to u$ is called a transitive edge if the graph also has edges $v \to w$ and $w \to u$. If reads corresponding to vertices v and u overlap, the transitive edge can be removed without affecting which contigs can be produced. A bubble is an acyclic directed subgraph, where the sink and source vertices have at least two paths connecting them, and they are the only vertices that are connected to the rest of the graph. Bubbles may appear because of missing overlaps or haplotype variants. Collapsing bubbles makes the graph simpler, thus creating more unified contigs at the expense of some possibly lost information. A tip is a vertex with at least one incoming edge but no outgoing edges. They can appear due to missing overlaps or artifacts so the removal of short tips is considered justified.

After graph is cleaned, miniasm finds all unitigs in it. The exact definition of a unitig is that it is a path in an assembly graph, where the in-degree and out-degree of all vertices is 1, except the first vertex can have in-degree that is not 1, and the last vertex can have out-degree that is not 1. A unitig can also be a circular non-branching path. A unitig is formed by taking the reads corresponding to the vertices in a path and concatenating them so that the overlapping sequence between consecutive reads is included only once. For example, suppose we found a path containing reads $[v_1, v_2, v_3, ..., v_{n-1}, v_n]$. The unitig U obtained from this path can be expressed as

$$\begin{aligned} U = v_1[1, len(v_1 \to v_2)] \circ v_2[1, len(v_2 \to v_2)] \circ v_3[1, len(v_3 \to v_4)] \circ ... \\ \circ\, v_{n-2}[1, len(v_{n-2} \to v_{n-1})] \circ v_{n-1}[1, len(v_{n-1} \to v_n)] \circ v_n \end{aligned}$$

where $\circ$ is the concatenation operation.

Unitig is a special case contig; it is a consensus sequence that has no gaps. A contig can also be formed by taking a branching path, unlike unitigs. We will be referring to the results of the assemblies as contigs in the following, even though they could also be referred by the more descriptive term.

3.4 Optical map mapping

The relative location information between reads is expressed by coloring them with the help of optical maps. The optical maps of the reads often have very few fragments and they are not suitable for reliable mapping. On the other hand, optical maps of pre-coloring contigs are easier to map since they are considerably longer leading to more convincing mappings. Therefore we do the read coloring via colored contigs. This may result in some additional lost or even false information, because we are relying on the correctness of the pre-coloring contigs produced by miniasm [Li16].

Originally Kermit [WRS18] was designed to get the color information with the help of genetic maps. A genetic map is a set of markers, which are divided into bins. The bins are ordered, which means that if two markers are in different bins, we know for sure which marker appears before the other in the genome, while nothing can be said about the relative order of markers within a single bin. Bins are assigned colors, represented as integers starting from 0. If the integer, in other words a color, of a bin is smaller than another bin's, we know that all markers within it appear before the markers in the other one. Markers in a bin are given the color of the bin they reside in. In addition to genetic maps, Kermit requires a draft assembly of the genome to color reads. It first maps reads to the draft assembly, and then finds which markers in the genetic map the reads overlap with. Finally, reads are colored with the colors of these overlapping markers. Kermit only requires the knowledge of the smallest and biggest color integers, so the reads can be colored just with those.

Our new approach is reminiscent of the initial Kermit implementation. Our pre-coloring contigs from the unguided miniasm assembly are colored, and the reads are aligned to them. Each read is then colored with the colors of the overlapping contig segments. We color contigs using optical map mapping, which is done based on matching fragments sequences. Ideally, optical maps would map to each other exactly fragment to fragment. In other words, the query optical map would be a sub optical map of the target optical map. In real world applications this very rarely happens. When creating optical maps of the contigs computationally, sequencing errors, insertions and deletions, can respectively lengthen or shorten the resulting fragments. Additionally, these errors with substitutions can remove and create cut sites, causing intended fragments to falsely merge and split. On the other hand, when optical maps are created through restriction digestion, errors characteristic to these technologies are inevitably going to occur. We chose to perform optical map mapping with mapping tool by Valouev et al. [VLL+06], which takes errors occurring in restriction digestion into consideration. As suitable real optical maps were hard to find, our experimental data does not actually contain them, which this mapping tool is specialized in. We assumed it would still perform well enough with our computationally generated optical maps, which our experimental results seemed to confirm. Since the mapping tool by Valouev et al. [VLL+06] was not given a name, we will refer to it as VM from this point onward.

VM uses a statistical model to calculate mapping scores. This model corresponds to the different types of errors and their likelihoods that occur in restriction digestion. According to the authors of VM, there are four types of errors associated with optical maps which are described in the following; missing and false cuts, missing fragments, sizing errors and chimeric reads. Missing cuts happen when the restriction enzyme does not cut the molecule at the cut sites it is supposed to. False cut sites appear when the molecule happens to split at a random point. After the molecule has been cut, the fragments are attached to a glass surface for measuring. Some fragments can be too small, and they do not attach properly, causing some fragments to fall off and go missing. The lengths of the fragments are measured by attaching fluorescent markings to the DNA. The length of the fragments is estimated by measuring their fluorescence intensity. Due to this not hundred percent accurate estimation, the lengths of the fragments are just that, approximations. Chimeric reads happen when mapped molecules cross i.e. combine randomly. This creates false optical maps consisting of fragments from completely different locations of the genome. VM does not address this error type, which should not be a problem for our use case, since most of the optical maps are obtained via *in silico* digestion.

VM tries to take these errors into consideration by building a model to find the best matching regions between query and target optical maps. The number of cuts in an optical map is modeled as a Poisson distribution, which takes into account the distribution of the actual cuts and the distribution of the false cuts. The discrepancy between the measured and actual length of a fragment is modeled as a normal distribution. Two versions of this length error model are used, one for short fragments and the other for longer ones. The lengths of the fragments are modeled with an exponential distribution. These distributions are combined to build the whole model, and likelihood ratios for optical maps are derived from it. The likelihoods are used to calculate the score for how well a set of fragments map to another set of fragments, i.e. how likely it is that these two optical maps represent the same genomic region.

VM can handle two kinds of alignments, for fitting and overlapping. The likelihood functions differ accordingly between these two cases. In overlap alignment the alignment between two optical maps is partial, meaning the tails of the optical maps can be left unaligned. Fit alignment handles the cases where the query optical map is assumed to fully align with the target optical map. It is reasonable to assume fit alignment suits our needs better, since we are mapping mapping shorter contig optical maps to the chromosome optical maps of the reference genome.

In VM fit alignments are defined as sets of ordered cut site pairs (i_0, j_0), (i_1, j_1), ...,(i_d, j_d). Cut sites i_x, where x is the pair index, are from the set of query optical map cut sites Q and cut sites j_x are from the set of target optical map cut sites T. The first cut sites are at the beginning of the optical maps and the last cut sites are at the end of the optical maps. For this reason i_0 is always 0 because the first cut site of the query optical map is always present in the alignment. Similarly, the last cut site i_d is always equal to the length of the query optical map, since the last cut site must be present too. The remaining cut sites in the pairwise alignment are decided using the likelihood score. Simply put, the cut site pairs between query and target optical maps are chosen so that the resulting likelihood score is maximized. The query optical map can map anywhere in any of the target optical maps, so an efficient dynamic programming algorithm is used. It calculates the best possible fit alignments $X(i, j)$ where i and j are the rightmost cut sites in their corresponding optical maps, where $i \in Q$ and $j \in T$. Using the stored best scores for all different alignments (i, j) it is easy to find the alignment that maps the full query optical map to the target optical map.

VM calculates two scores, the likelihood score used in their model called s-score, and an heuristic score proposed in [WSK84] called t-score. The s-score is shown to work noticeably better with real world optical maps, while with simulated optical the improvements were only minor. For this reason only s-scores were considered in our experiments.

3.5 Contig coloring

The output of VM [VLL+06] tells us which cuts of the query optical map map to which cuts of the reference optical map. Another way to interpret this is to consider it as fragments mapping to each other. The blocks of fragments between contig and reference optical maps also map to each other similarly to the cut sites. We have two ways to color the contigs, either color the cuts or color the fragments. We decided to do the coloring through fragments, but using cut sites is also a valid approach.

Fragments of the optical maps are numbered from 0 to $n - 1$ where n is the number of fragments in that optical map. To separate the fragments of different chromosome optical maps, we add $k * s$ to the number of the fragment, where k is the number of the chromosome and s is an appropriate power of ten, so that the colors from a chromosome do not overflow into the colors of the next one. This way we can guarantee that each color represents a unique fragment, while it is easy to tell which chromosome the fragment belongs to and what is the position of that fragment within the chromosome. Fragment numbers of the reference optical map represent the colors in our coloring scheme. Fragments of the pre-coloring contig optical maps are colored based on their mapping with reference optical map fragments. There are four fragment mapping cases listed below:

1. one to one
2. one to many
3. many to one
4. many to many

In the first case, query fragment is colored with the matching target fragment color. In the second case, the query fragment is shattered into as many fragments as there are target fragments. The new fragment lengths are relative to the target fragment sizes, but their total length is that of the query fragment. These new fragments replace the old fragment, and they are colored with the colors of their respective target fragments. In the third case, all query fragments are colored with the color of the one target fragment. The fourth case is similar to the second case, but now the total length of the new fragments is the total length of all query fragments. Figure 7 illustrates these coloring rules further.

All these coloring cases can be generalized into a single coloring rule. Suppose N contig fragments with total length n map to M reference fragments with total length m. Create new fragments by taking the M reference fragments and dividing their lengths by m and multiplying by n. Keep the colors the same as in the original M reference fragments and replace the original N contig fragments with these new fragments.

VM gives at most one mapping for each query optical map. These mappings could be used directly to color the contigs, but one could also consider the quality of the mapping before deciding which of them are reliable enough to use. We tested two different approaches here, coloring all contigs that could be mapped, or set some score threshold that the mapping needed to exceed to be used in coloring. We decided to experiment with two different thresholds, s-score = 0 (color all mapped contigs) and s-score = 15 (color contigs that mapped 'well'). These were chosen arbitrarily just to see how applying thresholds would affect the coloring and assembly in general. The differences can be seen in Section 4.

3.6 Read coloring

The last step in our data processing is the coloring of the corrected reads. We start by aligning the reads to the pre-coloring contigs. This is done with minimap2

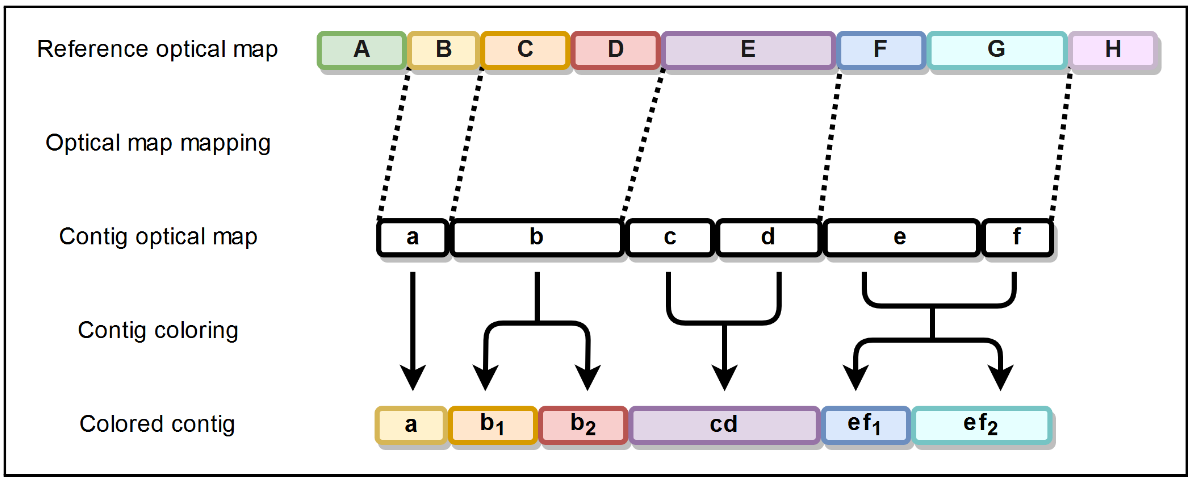

Figure 7: Contig coloring example illustrating four different fragment mapping cases. **Case 1:** Contig fragment a maps to reference fragment B, and is colored with its color. **Case 2:** Contig fragment b maps to reference fragments C and D. It is split into two fragments b_1 and b_2 whose total length is equal to the length of b, keeping the proportions of C and D. Fragment b_1 is colored with the color of fragment C, and b_2 with the color of D. **Case 3:** Contig fragments c and d map to reference fragment E. Fragments c and d are merged together into one fragment cd which is colored with the color of fragment E. **Case 4:** Contig fragments e and f map to reference fragments F and G. Fragments e and f are transformed into two fragments ef_1 and ef_2 with the total length of e and f, keeping the proportions of fragments F and G. Fragment ef_1 is colored with the color of fragment F, and ef_2 with the color of G.

[Li18] to keep our aligning processes consistent, since this program is also used in pre-coloring contig assembly with miniasm [Li16], read correcting with CONSENT [MML$^+$19], and the post-coloring contig assembly with Kermit [WRS18].

Minimap2 can produce multiple possible alignments for a single read. If this should happen, the intuitive solution is to use the 'best' alignment for coloring. We used a simple heuristic for this choice; take the alignment with the greatest number of matching bases in its aligning segment with the target contig. The start and end positions of an alignment in a contig are known, which are used to determine which colors of the contig the alignment covers. For example, suppose the aligning section in the contig starts from position i and ends at position j. We begin to add the lengths of the contig fragments together one by one. The first fragment n, for which the sum of the lengths of the fragments $[0, n]$ is greater than i is chosen as the starting fragment i.e. starting color. Similarly, the first fragment m, for which the sum of the lengths of the fragment $[0, m]$ is greater than j, is used as the ending fragment i.e. ending color. As Kermit only uses the start and end colors, this coloring process suffices.

Even if the main idea of the coloring is quite simple, few choices regarding it must be made. First, which of the alignments is chosen to be used as basis for the coloring. We already mentioned using the matching bases as a quality measurement for the alignment. In some cases, we could be unable to color the contig that aligns best with a read. We might want to look into some of the other aligning contigs if this should happen. Also we need to consider if even the best alignment is good enough to do reliable read coloring.

We implemented three different ways to choose the used alignment. First approach is to only consider the best aligning contig. If it is not colored, then the read is also left uncolored. Second method is to choose the best aligning contig among all the aligning colored contigs. If none of the aligning contigs are colored, then the read is left uncolored. The third approach is similar to the second one, but if the best aligning colored contig is not 'good enough', the read is again left uncolored. We decided that if the number of matching bases is at least 80% of the total length of the read, the alignment would be reasonably strong to be accepted.

Next, we deliberated if the beginning and end parts of the reads that are left outside of the aligning section should be used to extend the coloring beyond the aligning section in the contig. For example, suppose the alignment in contig starts again a position i and ends at position j, but we also know that there is a sequence of length

a in the beginning of the aligning section of the read, and a sequence of length b at the end of aligning sequence of the read. The question is, would it be beneficial to adjust the starting position of the aligning section in the contig to start at $i - a$ and to end at $j + b$ to possibly extend read colors. Figure 8 shows how extending could affect the coloring.

During experiments, in most cases it appeared that the aligning section of the read was not remarkably shorter compared to the whole length of the read. This means the coloring would likely stay the same with or without this extension, or maybe be one lower at the start and one greater at the end. If the extensions would greatly affect the coloring, then it would suggest that the aligning section would be short, which should not be possible because we are requiring it to be strong in the first place. Nevertheless, we experimented with multiple different coloring options to figure out which is the optimal one, and it will be discussed more in Sections 4 and 5.

After the reads are colored, some of the available colors can be completely unused by the reads. In other words, none of the read alignments overlapped with the fragments corresponding to the missing colors. To be clear, colors between the start and end colors of a read are also considered as used. It is possible that two reads are actually physically close to each other even though there is a color gap between them due to inaccurate alignments for example. This might affect the performance of Kermit negatively, so we decided to shift the colors so that there were no missing colors. For example, suppose the reads used colors $\{1, 2, 4, 7, 8, 10\}$. These colors would then be mapped to colors $\{1, 2, 3, 4, 5, 6\}$ in their respective orders, $1 \to 1$, $2 \to 2$, $4 \to 3$, $7 \to 4$, $8 \to 5$, $10 \to 6$. Now the already colored reads would be recolored according to these mappings. They would still keep their respective order without having unused colors. Colors were only shifted within a single chromosome's colors i.e. the $s * k$ term in the colors was left untouched; only the color numbers which come from fragments were adjusted. We ran experiments with and without this adjustment to determine its effects.

3.7 Post-coloring contig assembly

After all the needed data is gathered and processed to get corrected and colored reads, we can start building post-coloring contigs. Different assembly methods were introduced in Section 2.3, and our chosen assembler Kermit [WRS18] is a modification of the miniasm [Li16] assembler, which was already used to get our pre-coloring

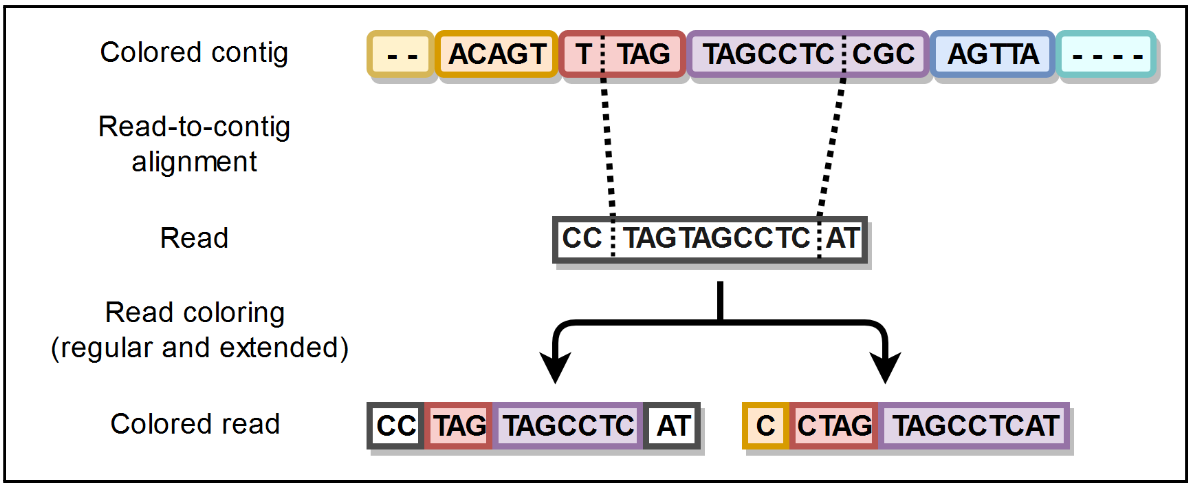

Figure 8: Simplified example of the two different cases of read coloring when a reasonably strong alignment with a colored contig is found. **Case 1:** Left hand side coloring is based on the aligning segment only. **Case 2:** Right hand side coloring is based on the aligning segment extended with the lengths of the non-aligning sections of the read.

contigs in Section 3.3.2. These two assemblers work very similarly, the difference being that during the layout step Kermit cleans the assembly graph based on the given colorings.

Kermit starts by building an overlap graph with the help of minimap2 [Li18] alignments. The same read-to-read alignments that were used during pre-coloring assembly can be used here. Each vertex of the graph represents a sequence, and we know which read is responsible for it. Because of this, all the vertices are colored with all the colors of the reads that were used to form it, which leads to a colored overlap graph. A vertex is given the starting and end colors of a read and with all the colors between them.

As it was with miniasm, a unitig is again defined as a maximal non-branching path in the overlap graph. Since the vertices are colored, the resulting paths are also colored. This coloring information can be used to alter the paths that are used to build the unitig. An edge from vertex v_i to vertex v_{i+1} means the corresponding sequences overlap, and can be merged together to be used as a part of a unitig. Some of the connections are bound to be erroneous, which are detected with the help of the colors. Suppose an edge (v_i, v_{i+1}) exists in the graph, but the color number of vertex v_i was larger than that of vertex v_{i+1}. The color information suggests that v_i should appear after v_{i+1}, but the edge says they are connected together in the reverse order. We want to trust the coloring information more than the edge, and the edge is removed completely from the graph. Even if the order of the colors is correct, their distance can cause suspicion. If the colors are very far apart, we can deduce the vertices should not be connected with an edge, and it is also removed from the graph. These kind of edges are called inconsistent edges. A consistent edge (v_i, v_{i+1}) would be an edge so that at least one of the colors of vertex v_{i+1} is equal or exactly one greater than at least one of the colors in vertex v_i. By default Kermit only allows the colors to differ at most by one, but this restriction can be adjusted by the user. All edges that do not follow this requirement are discarded. This way we cannot take a path with a huge gap or where some of the sequences are in wrong order.

Some of the reads might be left uncolored due to difficulties and uncertainness during the coloring step. An edge that is connected to a uncolored vertex would be automatically removed, but Kermit alleviates this problem of uncolored vertices by allowing the colors to propagate. An uncolored vertex is assigned all colors of the vertices that are reachable from it by travelling only through uncolored vertices. There is a limit how far the colors should propagate, since allowing the colors to flow as far as possible would most likely lead to incorrect colorings. By default Kermit allows the colors to propagate through five vertices, but again this can be changed to the user's liking. If propagated colors have some missing colors between them, the vertex is deleted completely from the graph because the propagated coloring is not coherent which makes us suspicious of its correctness.

After inconsistent edges are removed, Kermit starts looking for the unitigs. Instead of finding all maximal non-branching paths, we automatically find all maximal non-branching *rainbow paths*. A rainbow path is like a normal path, but all the colors must appear in increasing order and only once. Kermit loosens this condition by allowing consecutive vertices to use the same color on a path. The removal of the inconsistent edges guarantees that all paths we find are also rainbow paths. Kermit outputs these maximal non-branching rainbow path unitigs as our final assembly product.

4 Experiments and results

We used three different sets of reads to test our guided genome assembly pipeline. One set contained reads of *A.thaliana* [1] obtained through sequencing, and the two

[1] https://www.ncbi.nlm.nih.gov/genome/4?genome_assembly_id=454618

Data sets	*S.cerevisiae*	*C.elegans*	*A.thaliana*
Reference genome accession number	NC_001133 - NC_001148	NC_003279 - NC_003284	LR215052 - LR215056
Reference genome length [bp]	12 071 326	100 272 607	118 057 531
Number of chromosomes	16	6	5
Average chromosome length [bp]	754 458	16 712 101	23 611 506
Number of reads	58 806	488 554	213 113
Total length of reads [bp]	481 497 241	4 013 936 122	4 787 076 012
Average read length [bp]	8 188	8 216	22 463

Table 1: Information about the data used in our experiments. Mitochondrial and chloroplast DNA are excluded.

others were simulated reads of *S.cerevisiae* [2] and *C.elegans* [3]. *A.thaliana* reads were real world PacBio reads [4], while the simulated reads were generated based on already assembled genomes using simlord [SKR16], which mimics the error pattern of PacBio sequencing.

Each read set had a 40-time genome coverage. *A.thaliana* reads were chosen from the larger set of reads from longest to shortest until the coverage requirement was fulfilled. Information about the reference genomes and data sets can be seen in Table 1.

The high error rate reads were first corrected with CONSENT [MML+19], and the results can be found in Table 2. Read correction was by far the most time consuming step of the pipeline, taking three whole days to correct the *A.thaliana* read set. Even though the total length of the read sets of *C.elegans* and *A.thaliana* were similar, correcting the real world read set took more than three times longer compared to the simulated one. This could be the result of the shorter length of the simulated reads, or the simulated reads are just overall easier to correct e.g. simlord does not simulate ambiguous nucleotides which do appear in the real world reads.

CONSENT was not the fastest correction tool available, but it was the one which worked with all our data sets. Additionally, the time consumption of the other steps of the pipeline was almost negligible in comparison, so using a correction algorithm that was slightly faster would not have made much of a difference. The time distribution between other steps of the pipeline can be seen in Table 6. CONSENT was

[2]https://www.ncbi.nlm.nih.gov/genome/15?genome_assembly_id=22535

[3]https://www.ncbi.nlm.nih.gov/genome/41?genome_assembly_id=43998

[4]https://downloads.pacbcloud.com/public/SequelData/ArabidopsisDemoData/

Error correction	*S.cerevisiae*	*C.elegans*	*A.thaliana*
Number of original reads	58 806	488 554	213 113
Number of corrected reads	58 552	486 460	209 206
Original reads total length [Kbp]	481 497	4 013 936	4 787 076
Corrected reads total length [Kbp]	458 484	3 826 773	4 096 015
Avg. original read length [bp]	8 188	8 216	22 463
Avg. corrected read length [bp]	7 830	7 867	19 579
Time [hh:mm:ss]	04:13:58	21:16:15	76:33:52
Max. resident set size [KB]	6 691 920	9 243 716	23 260 072

Table 2: CONSENT error correction results.

Pre-coloring contig assembly	*S.cerevisiae*	*C.elegans*	*A.thaliana*
Number of chromosomes	16	6	5
Number of contigs	26	111	539
Reference genome length (bp)	12 071 326	100 286 401	119 668 634
Contigs total length (bp)	12 049 591	100 429 921	133 817 935
Time: minimap2 [hh:mm:ss]	00:02:37	00:23:18	03:02:23
Max. resident set size (KB): minimap2	4 636 912	13 554 764	14 669 152
Time: miniasm [hh:mm:ss]	00:00:07	00:01:47	00:20:44
Max. resident set size [KB]: miniasm	114 978	2 132 968	14 160 656
Time: awk [hh:mm:ss]	00:00:01	00:00:02	00:00:04
Max. resident set size [KB]: awk	8 352	70 636	61 992

Table 3: Pre-coloring contig results using minimap2 read aligning and miniasm assembly.

executed on a machine with 12 cores (2.2 GHz) and 64 GB memory, while all the other steps were run on a machine with 6 cores (3.6 GHz) and 16 GB memory.

After reads were corrected, pre-coloring contigs were assembled. First minimap2 [Li18] was used to calculate read-to-read alignments, which were then used to produce contigs with miniasm [Li16]. Results of the pre-coloring contig assembly can be seen in Table 3.

Next optical maps of the pre-coloring contigs and the reference genome were generated. The optical map for the genome of *A.thaliana* was also produced computationally, since the optical map for it was not easily available. In fact, none of our experiments were done purely on real world data, which is something that could be further explored in the future. The optical maps were generated emulating enzyme XhoI, which recognizes restriction site 'CTCGAG'. After the optical maps of the contigs and reference were ready, they were mapped to each other using VM [VLL+06].

The following contig and read coloring are steps where we were more involved, since they were specific to our pipeline. First, the contigs were colored using VM mappings, as explained in Section 3.5. VM outputs a single mapping position for each contig optical map in the genome optical maps. Not all the mappings are necessarily correct, so we have to make a distinction between mappings we can trust and mappings that are dubious. We decided to simply use the s-score calculated by VM to determine which mappings could be used. We set the s-score threshold arbitrarily at 15, which seemed promising value after looking at the initial mapping scores. It is certainly possible that a correct mapping gets a lower s-score than 15, due to the short lengths of some contig optical maps. For this reason, we also ran test runs where all the mappings were accepted regardless of their s-scores.

After the contigs are colored and reads are aligned with them using minimap2, read coloring can begin. Minimap2 can produce multiple alignments for each read, and we must choose which one to use. We came up with three different ways to decide this. First, we can just use the longest alignment for coloring; if the contig with the longest alignment was uncolored, the read would also be left uncolored. Second coloring scheme is to find a colored contig with the longest alignment. We also wanted to try to use the quality information of the alignments to our advantage. The third coloring option looks for the longest alignment with a colored contig that also has $t * len(r)$ matching bases, where $t = 0.8$ and $len(r)$ is the length of the read.

Two more coloring options were discussed previously in Section 3.6, extended and adjusted coloring. We also run the genome assembly pipeline with and without these to determine which setup would give the most promising results. All in all, the coloring options we have are the following:

- Contig coloring with minimum s-score threshold at 0 or 15.
- Read coloring using only the best alignment, the best colored alignment or the best colored alignment with at least $t * len(r)$ matching bases.
- Read coloring using just the alignment region colors or extending the coloring region according to the read overhang region lengths.
- Leaving the colored reads as they are or adjusting them so that there are no

gaps between all used colors.

In total this gives us $2 * 3 * 2 * 2 = 24$ different combinations to color the reads. On top of that, we wanted to know how the color propagation of Kermit [WRS18] would be affected by them, so Kermit assembly was run with and without color propagation (with default propagation depth and tolerance parameters -d=5 and -b=1). This brings the total number of different assemblies to 48. We did not want to assemble all three genomes this many times, so *C.elegans* was chosen as the test subject to determine which is the optimal combination to assemble the other two. The results of these 48 assemblies are in Table 4. QUAST [GSVT13] (Quality Assessment Tool for Genome Assemblies) version 5.0.2 was used to estimate the quality of the resulting post-coloring contigs.

It is not obvious which of the coloring options is the best, as none of them has all the best qualities. We decided that NGA50 score gives a good overview of the overall quality of the assembled contigs. NGA50 is calculated by first aligning contigs to the reference genome. The contigs are split into blocks that align to the reference properly if they do not align as a whole. Next we set the blocks in order from longest to shortest and find the middle point i.e. where the longer blocks on the left have total length equal or greater than 50% of the total length of the reference genome. NGA50 is the length of the block at the middle point. There were four cases where the top NGA50 score of 3028 Kb was achieved. Another good quality is low number of misassemblies. Among these four assemblies two had lower score than the other two, 8 misassemblies. In fact these two had the exact same number of contigs and contig lengths, meaning that the effect of color extension was negligible. Nevertheless we decided to include the extension step, leaving us with the following assembly options:

- No minimum s-score in contig coloring
- Read is colored only based on the best aligning contig
- Read colors extended beyond the aligning region
- Read colors adjusted so that used colors do not contain gaps

The two other genomes for *S.cerevisiae* and *A.thaliana* were also assembled using these options. The results for all three of the assemblies can be seen in Table 5.

***C.elegans* pre-coloring contig results using miniasm**

Minimum s-score	Read coloring	Extended coloring	Adjusted coloring	Number of contigs	Number of >50Kb contigs	Length of contigs [Kb]	Length of >50Kb contigs [Kb]	Mis-assemblies	NGA50 [Kb]
-	-	-	-	111	75	100 430	99 770	11	2 656

***C.elegans* post-coloring contig results using Kermit with color propagation**

Minimum s-score	Read coloring	Extended coloring	Adjusted coloring	Number of contigs	Number of >50Kb contigs	Length of contigs [Kb]	Length of >50Kb contigs [Kb]	Mis-assemblies	NGA50 [Kb]
0	Only best	No	No	123	59	97 709	96 104	6	2 868
0	Best colored	No	No	162	92	97 408	95 621	6	1 741
0	Best hi-scoring	No	No	150	60	98 341	96 204	8	2 868
0	Only best	Yes	No	123	59	97 709	96 104	6	2 868
0	Best colored	Yes	No	162	92	97 408	95 621	6	1 741
0	Best hi-scoring	Yes	No	151	60	98 368	96 208	8	2 868
15	Only best	No	No	119	58	97 562	96 031	6	2 868
15	Best colored	No	No	157	92	97 274	95 621	6	1 741
15	Best hi-scoring	No	No	151	59	98 320	96 190	8	2 868
15	Only best	Yes	No	119	58	97 562	96 031	6	2 868
15	Best colored	Yes	No	157	92	97 274	95 621	6	1 741
15	Best hi-scoring	Yes	No	152	59	98 347	96 193	10	2 868
0	Only best	No	Yes	119	55	97 713	96 108	7	2 868
0	Best colored	No	Yes	162	91	97 434	95 634	6	1 741
0	Best hi-scoring	No	Yes	148	57	98 375	96 227	8	2 868
0	Only best	Yes	Yes	119	55	97 713	96 108	7	2 868
0	Best colored	Yes	Yes	162	91	97 434	95 634	6	1 741
0	Best hi-scoring	Yes	Yes	152	58	98 421	96 219	8	2 868
15	Only best	No	Yes	115	54	97 566	96 035	7	2 868
15	Best colored	No	Yes	157	91	97 300	95 634	6	1 741
15	Best hi-scoring	No	Yes	149	56	98 354	96 212	10	2 868
15	Only best	Yes	Yes	115	54	97 566	96 035	7	2 868
15	Best colored	Yes	Yes	157	91	97 300	95 634	6	1 741
15	Best hi-scoring	Yes	Yes	153	57	98 399	96 204	10	2 868

***C.elegans* post-coloring contig results using Kermit without color propagation**

Minimum s-score	Read coloring	Extended coloring	Adjusted coloring	Number of contigs	Number of >50Kb contigs	Length of contigs [Kb]	Length of >50Kb contigs [Kb]	Mis-assemblies	NGA50 [Kb]
0	Only best	No	No	93	62	100 192	99 640	8	2 868
0	Best colored	No	No	92	63	100 175	99 680	11	2 683
0	Best hi-scoring	No	No	131	85	100 022	98 875	10	2 868
0	Only best	Yes	No	93	62	100 192	99 639	8	2 868
0	Best colored	Yes	No	92	63	100 175	99 680	11	2 683
0	Best hi-scoring	Yes	No	130	85	100 019	98 888	10	2 868
15	Only best	No	No	102	63	100 414	99 680	10	2 868
15	Best colored	No	No	96	63	100 270	99 676	12	2 683
15	Best hi-scoring	No	No	131	85	99 982	98 863	12	2 868
15	Only best	Yes	No	102	63	100 414	99 680	10	2 868
15	Best colored	Yes	No	96	63	100 270	99 676	12	2 683
15	Best hi-scoring	Yes	No	130	85	99 979	98 876	12	2 868
0	Only best	No	Yes	94	61	100 215	99 637	8	**3 028**
0	Best colored	No	Yes	92	62	100 200	99 694	11	2 683
0	Best hi-scoring	No	Yes	133	84	100 077	98 893	10	2 868
0	Only best	Yes	Yes	94	61	100 215	99 637	8	**3 028**
0	Best colored	Yes	Yes	92	62	100 200	99 694	11	2 683
0	Best hi-scoring	Yes	Yes	133	84	100 086	98 902	10	2 868
15	Only best	No	Yes	103	62	100 436	99 678	10	**3 028**
15	Best colored	No	Yes	96	62	100 296	99 689	12	2 683
15	Best hi-scoring	No	Yes	133	84	100 037	98 881	12	2 868
15	Only best	Yes	Yes	103	62	100 436	99 678	10	**3 028**
15	Best colored	Yes	Yes	96	62	100 296	99 689	12	2 683
15	Best hi-scoring	Yes	Yes	133	84	100 046	98 890	12	2 868

Table 4: Comparison of *C.elegans* post-coloring contig assembly results with different read coloring schemes. Results obtained by running Quality Assessment Tool for Genome Assemblies, QUAST 5.0.2, on the contigs and reference genome with '--large' argument.

***S.cerevisiae* contig assembly results**						
	Number of contigs	Number of >50Kb contigs	Length of contigs [Kb]	Length of >50Kb contigs [Kb]	Mis-assemblies	NGA50 [Kb]
Miniasm	26	21	12 050	11 986	6	664
Kermit	24	18	12 029	11 956	6	776

***C.elegans* contig assembly results**						
	Number of contigs	Number of >50Kb contigs	Length of contigs [Kb]	Length of >50Kb contigs [Kb]	Mis-assemblies	NGA50 [Kb]
Miniasm	111	75	100 430	99 770	11	2 656
Kermit	94	61	100 215	99 637	8	3 028

***A.thaliana* contig assembly results**						
	Number of contigs	Number of >50Kb contigs	Length of contigs [Kb]	Length of >50Kb contigs [Kb]	Mis-assemblies	NGA50 [Kb]
Miniasm	539	114	133 818	119 877	58	1 442
Kermit	276	105	125 297	119 817	63	1 648

Table 5: Pre- and post-coloring contig assembly results using miniasm and Kermit.

Our assembly pipeline consists of the following steps: pre-coloring contig assembly, optical map generation and mapping, read-to-contig aligning, contig and read coloring, and post-coloring contig assembly. Table 6 has the time and memory consumptions of each of these steps. Keep in mind that the read correcting was done on a different mahchine with more memory compared to the other steps, and the time and memory consumption of that can be seen in Table 2.

5 Conclusions

We have presented a new genome assembly pipeline to utilize optical maps automatically during the assembly process. First, the long reads are corrected with CONSENT [MML$^+$19]. The corrected reads are then given to miniasm [Li16] assembler to produce initial pre-coloring contigs. Next, optical maps of the contigs are generated computationally, and mapped with the optical maps of the reference genome chromosomes using VM [VLL$^+$06]. The contigs are also aligned with the corrected reads with minimap2 [Li18]. The optical map mapping data is then used with the read-to-contig alignments to approximate the origin locations of the reads within the genome. Location approximations are represented by coloring the reads. Finally these colored reads are given to Kermit [WRS18] assembler to produce the post-coloring contigs as our final assembly product.

***S.cerevisiae* assembly memory and time usage**			
	Memory [KB]	Time [hh:mm:ss]	Time [%]
Read-to-read alignment (minimap2)	4 636 912	00:02:37	80.9
Pre-coloring assembly (miniasm+awk)	114 978	00:00:08	4.1
Optical map mapping (create + mapping)	39 488	00:00:05	2.6
Read-to-contig alignment (minimap2)	602 060	00:00:14	7.2
Read coloring (contig+read coloring)	102 704	00:00:03	1.5
Post-coloring assembly (Kermit+awk)	267 348	00:00:07	3.6
Maximum memory / Total time	4 636 912	00:03:14	100.00
***C.elegans* assembly memory and time usage**			
	Memory [KB]	Time [hh:mm:ss]	Time [%]
Read-to-read alignment (minimap2)	13 554 764	00:23:18	68.3
Pre-coloring assembly (miniasm+awk)	2 132 968	00:01:49	5.3
Optical map mapping (create + mapping)	635 336	00:03:35	10.5
Read-to-contig alignment (minimap2)	1 955 148	00:01:59	5.8
Read coloring (contig+read coloring)	1 575 096	00:01:44	5.1
Post-coloring assembly (Kermit+awk)	2 132 976	00:01:41	4.9
Maximum memory / Total time	13 554 764	00:34:06	100.00
***A.thaliana* assembly memory and time usage**			
	Memory [KB]	Time [hh:mm:ss]	Time [%]
Read-to-read alignment (minimap2)	14 669 152	03:02:23	76.9
Pre-coloring assembly (miniasm+awk)	14 160 656	00:20:48	8.8
Optical map mapping (create + mapping)	913 976	00:07:43	3.3
Read-to-contigs alignment (minimap2)	2 393 556	00:07:12	3.0
Read coloring (contig+read coloring)	4 068 052	00:00:38	0.3
Post-coloring assembly (Kermit+awk)	14 682 172	00:18:28	7.8
Maximum memory / Total time	14 682 172	03:57:12	100.00

Table 6: Memory and time usage statistics of the genome assemblies.

Our optical map guided genome assembly produced better contigs with the simulated data, compared to the unguided version i.e. miniasm pre-coloring assembly. NGA50 statistic improved in both cases, and the number of contigs decreased, suggesting that we were able to connect sequences that would have been left separated otherwise. This means that in some cases we were able to extend non-branching paths in the graph by deleting spurious edges. Total length of the contigs did not drop a considerable amount, so the original genome is still covered well. With *S.cerevisiae* the total length got slightly further away from the reference length, but with *C.elegans* we actually got closer to the reference length. Misassemblies stayed the same or even dropped, which of course is a positive result.

With the real *A.thaliana* reads the number of total contigs dropped dramatically and their total length dropped closer to the total length of the reference. To be fair, the reference genome was not a fully completed assembly of the ler-0 *A.thaliana*, so we are unable to say if this decrease in length is a positive or negative thing. We can also see that more misassemblies were made with Kermit. On the other hand, NGA50 statistic got better. Due to the incomplete reference it is hard to say definitively if our approach improved the assembly with real reads, but overall the end results are still promising.

If we look at the time and memory consumption of our assembly pipeline, we can see that they do not increase dramatically compared to the unguided approach. The most memory intensive tasks seemed to be the read-to-read mapping and the assemblies themselves, which are of course required in both guided and unguided approaches. The most time consuming step was again the mandatory read aligning. In the worst case with *C.elegans* assembly, the usage of read coloring and Kermit assembly was responsible for 26.4% of the total used time.

Even though the performance regarding time and memory did not suffer too much, we still need to remember that before everything else all the reads were corrected with CONSENT. Looking at Table 2 it becomes very obvious that this was actually the step that was responsible for the majority of the time consumption, and also required a great amount of RAM. Read correction needed to be done on a machine with higher memory capacity because CONSENT would not finish with *A.thaliana* data set otherwise. An interesting experiment would be to try to correct reads with another correcting program and see if a more effective one could provide as good or even better results. Additionally, we could even try to move away from correcting the reads. Miniasm, and by necessity Kermit, were originally designed to work with high error rate reads. It would be interesting to see how our pipeline worked with non-corrected reads with a polishing program run at the end as a separate step, as it was done in the Kermit paper. The *in silico* optical maps would of course have more errors in them too, but due to the sheer length of the pre-coloring contigs this might not be that big of a problem.

The coloring step is also something that could be improved by running more experiments to figure out the optimal settings. The read coloring option with greatest impact seemed to be the method of choosing which read-to-contig alignment to use. Being very strict led to fewer reads being colored, which in turn caused more edges to be dropped from Kermit assembly graph. Fewer edges seemed to create more new non-branching paths than fragmenting existing non-branching paths when color propagation was allowed. On the other hand, being generous with coloring seemed more advantageous when propagation was not allowed in comparison. Overall the most promising combination seemed to be having no propagation and being strict with read coloring, but being more forgiving with contig coloring. This is of course if we consider NGA50 and the number of misassemblies to be the most important factors. Read color adjusting was beneficial most of the time, while extending colorings very rarely even had any impact. Finding the optimal way to handle colorings proved to be a difficult task, and finding the right balance needs more experimentation.

It is possible that tuning the coloring options with simulated reads was a bad choice, and it should have been done using the real reads which represent the actual use case much better. A more thorough assessment of the read coloring qualities could also be tried in the simulated case. For each read we know for sure where from the original genome they originate from. This could be used to estimate how many of the reads were colored 'correctly'. But again this is only possible for the simulated reads, which would not help that much if they behave very differently compared to the real reads.

Regarding the used data, the *A.thaliana* reads were chosen from two sets of reads, favoring longer reads over shorter ones. This could be a serious mistake if there happens to be a systematic discrimination towards specific regions of the genome getting only short reads. This way we could end up with a biased data set. However, the assembly length coincides with the reference length, so this kind of bias seems not to be a serious concern. Nevertheless, another experiment with a different set of real reads (maybe even from another species) would definitively bring more insight into the performance of our assembly pipeline. Also, the XhoI restriction enzyme was chosen arbitrarily for the optical map generation. It is possible that different species behave better with different enzyme cut sites, e.g. produce more fragments to make mapping more accurate. It would be interesting to find out how much the chosen enzyme affects the quality of the post-coloring contigs.

We are using a lot of different programs throughout our pipeline. We did not perform these experiments with alternative programs, which could also be a worthwhile investigation. It is definitively possible that there exists other ready tools that would perform better with our pipeline.

We aimed to construct a pipeline that utilizes optical maps in the assembly, instead of the more common way of using them to orient and scaffold the generated contigs separate from the assembly. Because we were unable to use read optical maps, we had to perform the initial assembly to allow us color the used reads. One could claim that we are just running a regular *de novo* assembly during the pre-coloring contig assembly, and then improving the resulting contigs by running a similar guided post-coloring assembly with the help of optical maps. We would still argue that instead we have a two part assembly and optical maps are incorporated during the latter half, where the whole genome assembly restarts creating completely new contigs.

Using optical maps during assembly is an area that has not seen a lot of research, but interest in it has been growing in recent years. A few example programs that utilize optical maps during genome assembly can alredy be found, like the ones mentioned in Section 1, AGORA and KOOTA. KOOTA was not very competitive as an assembler compared to the other state-of-the-art assemblers, having a weaker N50 score (N50 is like NGA50 without the alignment to a reference genome). This was due to the emphasis of the research being on using optical maps to simplify de Bruijn graphs in the genome assembly, not developing a sophisticated traversal algorithm for it. AGORA assembler also utilized optical maps with de Bruijn graphs, and it performed quite well as an assembler, but it was only tested with error free reads of short bacterial genomes.

Our new Kermit pipeline is able to handle more complex genomes even with real world reads and perform competitively to its unguided counterpart assembler miniasm. In summary, the usage of optical maps seems promising even during genome assembly, and more research into this subject is warranted. We have shown that our proposed assembly scheme can be a viable option, and could still be improved with further development.

References

ASP^{+}17 Bahar Alipanahi, Leena Salmela, Simon J. Puglisi, Martin Muggli, and Christina Boucher. Disentangled long-read de bruijn graphs via optical maps. In Russell Schwartz and Knut Reinert, editors, *17th International Workshop on Algorithms in Bioinformatics, WABI 2017*, Leibniz International Proceedings in Informatics, pages 1:1–1:14, Germany, 2017. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

GSVT13 Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. QUAST: quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 02 2013.

KGE17 Mehdi Kchouk, Jean-Francois Gibrat, and Mourad Elloumi. Generations of sequencing technologies: From first to next generation. *Biology and Medicine*, 09, 01 2017.

LGM^{+}12 Henry Lin, Steve Goldstein, Lee Mendelowitz, Shiguo Zhou, Joshua Wetzel, David Schwartz, and Mihai Pop. Agora: Assembly guided by optical restriction alignment. *BMC bioinformatics*, 13:189, 08 2012.

LGS02 Christopher Lee, Catherine Grasso, and Mark F. Sharlow. Multiple sequence alignment using partial order graphs . *Bioinformatics*, 18(3):452–464, 03 2002.

Li16 Heng Li. Minimap and miniasm: fast mapping and de novo assembly for noisy long sequences. *Bioinformatics*, 32(14):2103–2110, 2016.

Li18 Heng Li. Minimap2: pairwise alignment for nucleotide sequences. *Bioinformatics*, 34(18):3094–3100, 05 2018.

MG77 A M Maxam and W Gilbert. A new method for sequencing dna. *Proceedings of the National Academy of Sciences*, 74(2):560–564, 1977.

MML^{+}19 Pierre Morisse, Camille Marchet, Antoine Limasset, Thierry Lecroq, and Arnaud Lefebvre. Consent: Scalable self-correction of long reads with multiple sequence alignment. *bioRxiv*, 2019.

Mye05 Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(2):79–85, January 2005.

PTW01 Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An eulerian path approach to dna fragment assembly. *Proceedings of the national academy of sciences*, 98(17):9748–9753, 2001.

RHH+04 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, 2004.

SC75 F. Sanger and A.R. Coulson. A rapid method for determining sequences in dna by primed synthesis with dna polymerase. *Journal of Molecular Biology*, 94(3):441 – 448, 1975.

SKR16 Bianca K Stöcker, Johannes Köster, and Sven Rahmann. Simlord: simulation of long read data. *Bioinformatics*, 32(17):2704–2706, 2016.

SW81 T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

TM17 German Tischler and Eugene W. Myers. Non hybrid long read consensus using local de bruijn graph assembly. *bioRxiv*, 2017.

VLL+06 Anton Valouev, Lei Li, Yu-Chi Liu, David C. Schwartz, Yi Yang, Yu Zhang, and Michael S. Waterman. Alignment of optical maps. *Journal of Computational Biology*, 13(2):442–462, 2006. PMID: 16597251.

WRS18 Riku Walve, Pasi Rastas, and Leena Salmela. Kermit: Guided Long Read Assembly using Coloured Overlap Graphs. In Laxmi Parida and Esko Ukkonen, editors, *18th International Workshop on Algorithms in Bioinformatics (WABI 2018)*, volume 113 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:11, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

WSK84 M. S. Waterman, T. F. Smith, and H. L. Katcher. Algorithms for restriction map comparisons. *Nucleic acids research*, 12(1 Pt 1):237–242, Jan 1984.

ZB08 Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.