SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Master's thesis no. 1182

De novo assembly using long error-prone reads

Mario Kostelac

Zagreb, February 2016.

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
MASTER THESIS COMMITTEE

Zagreb, 7 October 2015

MASTER THESIS ASSIGNMENT No. 1182

Student:	**Mario Kostelac (0036460349)**
Study:	Computing
Profile:	Computer Science

Title: **De Novo Assembly Using Long Error-prone Reads**

Description:

The problem of genome assembly has been around for more than 30 years. However it is still difficult to correctly assemble whole genomes due to short reads that cannot span repeating regions. Although a new generation of sequencers produces longer reads, these are usually error-prone, making it necessary to find a solution that will correct them. The short accurate reads produced by the next generation sequencing methods are usually used for this purpose. However, some recent works show that it is possible to achieve the same goal using long-error prone reads exclusively. The goal of this work is to develop a tool for genome assembly using such reads. The assembler should follow overlap-layout-consensus paradigm. As the name suggests, the implementation consists of the following: 1) Overlap phase, in which the overlap graph is built; 2) Layout phase, in which the graph is simplified and regionally linearized; 3) Consensus phase, in which the ambiguities from the layout phase are resolved. The implementation can use already developed overlap phase and integrate it into the solution. Every phase is to have an output in a standard community format. Synthetic datasets of varying complexity are to be constructed for extensive implementation testing. The solution should be appropriated for parallel architectures and implemented in C++. The code is to be documented using comments and should follow the Google C++ Style Guide when possible. The complete application should be hosted on Github under an OSI-approved licence.

Issue date:	16 October 2015
Submission date:	12 February 2016

Mentor:

[illegible]

Associate Professor Mile Šikić, PhD

Committee Chair:

[illegible]

Full Professor Siniša Srbljić, PhD

Committee Secretary:

[illegible]

Assistant Professor Tomislav Hrkać, PhD

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 7. listopada 2015.

DIPLOMSKI ZADATAK br. 1182

Pristupnik:	**Mario Kostelac (0036460349)**
Studij:	Računarstvo
Profil:	Računarska znanost

Zadatak: **De novo sastavljanje genoma koristeći dugačka očitanja s velikom pogreškom**

Opis zadatka:

Problem sastavljanja genome postoji dulje od 30 godina, no još uvijek je teško ispravno sastaviti cijele genome zbog kratkih očitanja koja ne mogu premostiti dugačke ponavljajuće regije. Iako nova generacija uređaja za sekvenciranje proizvodi dugačka očitanja, ta očitanja su osjetljiva na greške, zbog čega je potrebno pronaći rješenje za njihovo ispravljanje. Danas se u tu svrhu obično koriste kratka očitanja dobivena uređajima za sekvenciranje druge generacije. Međutim, u nekoliko novijih radova je pokazano da je moguće ostvariti isti cilj koristeći isključivo dugačka očitanja koja imaju veliku pogrešku. Cilj ovoga rada je izrada alata za sastavljanje genoma koji će koristiti samo ta očitanja. Alat je potrebno temeljiti na paradigmi preklapanje-razmještaj-konsenzus za sastavljanje genoma. Kao što samo ime sugerira implementacija se sastoji od: 1) Faze preklapanja u kojoj se izrađuje graf preklapanja; 2) Faze razmještaja u kojoj se graf pojednostavljuje i regionalno linearizira; 3) Faze konsenzusa u kojoj se razrješavaju nepreciznosti nastale u fazi razmještanja. Dozvoljeno je korištenje već razvijene, javno objavljene metode za fazu preklapanja i njena integracija u konačno rješenje. Svaka faza treba imati izlaz koji odgovara standardnim formatima. Potrebno je proizvesti nekoliko testnih skupova podataka različite kompleksnosti i provesti iscrpno testiranje. Rješenje mora biti prilagođeno paralelnoj arhitekturi i napisano u jeziku C++ . Programski kod je potrebno komentirati i pri pisanju pratiti stil opisan u Googleovom C++ vodiču. Kompletnu aplikaciju postaviti na GIT pod jednom od OSI-odobrenih licenci.

Zadatak uručen pristupniku:	16. listopada 2015.
Rok za predaju rada:	12. veljače 2016.

Mentor:

[illegible]

Izv. prof. dr. sc. Mile Šikić

Djelovođa:

[illegible]

Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za diplomski rad profila:

[illegible]

Prof. dr. sc. Siniša Srbljić

CONTENTS

1. Introduction

Understanding nature has always been a big desire of the human race. Observing, crafting theses, writing notes, observing again and again until the hypotheses are proven to be correct. That is just how it started. The whole story with observing the DNA started back in the 1866, when Gregor Mendel has written his Law of Heredity (Castle, 1903), observation on transferring different properties of plants across generations. As a matter of fact, at that point we did not know that the same mechanism of nature is accountable for transferring single's properties on their children; and that no living being can run away from Deoxyribonucleic acid (DNA).

Having an "image" of someone's DNA brings many benefits such as observation of mutations and detection of genes that could cause some diseases and illnesses. Fast forward to 2015; we know how that mechanism works and we know how to read the source of it - DNA (the process of "reading" the DNA is called sequencing), but the length of every fragment we get as output from the sequencing process (called read) is much shorter than the original DNA. We get it cut, in fragments, without any order or position in the original DNA. As years passed by, new sequencing techniques were developed so length and quality of reads depend on the platform used (Vaser, 2015). One of the hottest and most disruptive sequencing technologies in bioinformatics these days is MinION reader, produced in labs of Oxford Nanopore technologies. It produces very long reads (2 or 3 orders of magnitude longer than the previous generation), with a higher error rate as a tradeoff. Longer reads enable us to resolve short repeats in the DNA, but the higher noise in data introduces new problems which include the need for novel overlapping techniques.

The goal of this thesis is to present a modified assembly technique for such longer, erroneous reads, developed on the Faculty of Electrical Engineering and Computing in Zagreb, under the leadership of assoc. prof. Mile Šikić. Software presenting developed technique is called ra-integrate. It can be classified as a standard OLC *de novo* assembler without a consensus phase.

Chapter 2 (Preliminaries) will describe the sequencing technology used, basic concepts of assembling, different ways of doing it and different overlap types.

Chapter 3 (Methods) will try to describe the key methods used for building layout phase of our assembler.

In Chapter 4 (Implementation) we dissect the current state of the assembler implementation and specify the reasons for such architecture.

Chapter 5 (Results) shows results on two different datasets.

Chapter 6 brings the conclusion.

2. Preliminaries

This chapter will try to cover required knowledge to understand the rest of the thesis. It will start with describing current state of sequencing technology developed by Oxford Nanopore technologies (ONT). Afterwards, it will describe assembly process, general terms and types of assemblers and overlaps. Vast majority of important terms will be introduced in this chapter.

2.1. Oxford Nanopore

2.1.1. Technology

Oxford Nanopore is a sequencing technology developed by Oxford Nanopore Technologies. As its name says, it uses nano-scale pores (nanopores) as tunnels to drive DNA strands through (oxs). During the time DNA strands are passing through nanopores (as visible on Fig. 2.1), the device is passing ionic current through these nanopores and it is reading changes in current as a reaction to biological molecules passing through or nearby. These changes in current can be used to identify single molecules - single-molecule technology.

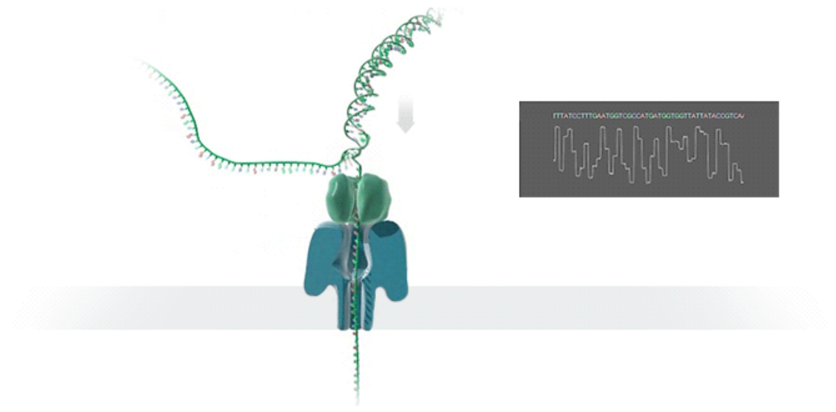

Figure 2.1: DNA strain being pulled through a nanopore (http://bit.ly/1X7FvxH)

Nanopores are protein-based (in the future they are going to be replaced by synthetic, solid-state nanopores, (sol)) and they are set in electrically resistant polymer membrane. As described, changes in current going through that membrane can be used to identify different molecules, they can be also used to identify different DNA bases - **a**denine, **g**uanine, **c**ytosine and **t**hymine (visible on Figure 2.2).

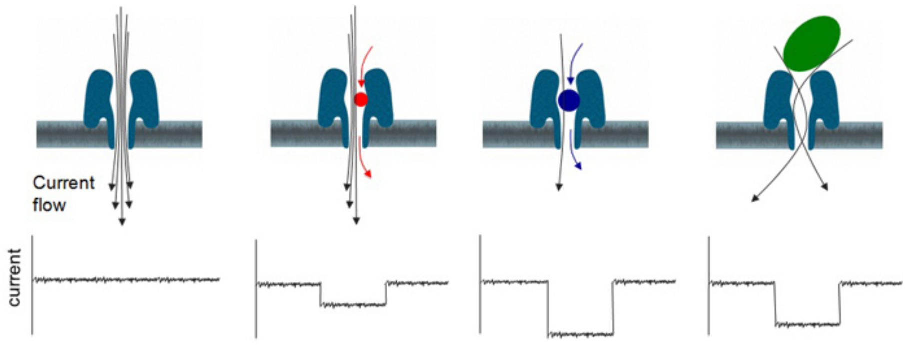

Figure 2.2: Different molecules are causing different current changes (http://bit.ly/1PWDhLD)

2.1.2. Disruptiveness

The value of described technology lies in three different properties - price, sequences length and portability. At the time of writing this thesis, listed price for the MKI set (sequencing machine + flow-cell) is ~1000$. Previous generation technologies cost at least one order of magnitude more.

MinION produces DNA sequences (produces as “streams to computer”) of variable lengths, usually longer than 1000b. Longest produced sequence is longer than 200,000 bases. Having an input set made of such a long DNA sequences solves a problem of short repeats (T. J. Treangen, 2011) and lowers the computational complexity of assembly.

MinION (device using Oxford Nanopore technology) set is very portable (smaller mobile phone sized) and connects through standard USB port, which puts that device almost on consumer market.

Huge drawback of this technology is the error rate higher than all other technologies. It can be even 40%, but range of error rate for 2d reads is usually 20-30% (Madoui et al., 2015).

PacBio technologies developed competitive single-molecule real-time technology, producing long-reads (not as long as Oxford Nanopore), with lower error-rate than Oxford Nanopore. Developed assembler should work equally good both with PacBio and MinION datasets.

2.2. Assembly

Once we connect MinION to our computer and it starts pulling DNA molecules through nanopores, streams of data are coming to a connected computer. The streamed data is actually current level over time which is being converted to strings made of C, G, A and T by software provided by ONT. That is how we get *reads* (from now on, reads are just strings of these four letters, sometimes with some associated attributes like quality or coverage) and point where computer science takes place. *Genome assembly* (referenced as assembly in latter parts of the thesis) is the process of reconstructing original genome from reads. Since reading technologies are imperfect and reads represent DNA in fragments, often wrong because of error rate, cut and without any meaningful order, a DNA strand has to be read multiple times in order to get more information and fill the gaps. That information overhead is usually called *coverage* - average number of reads per single DNA base.

Over years, two main assembling approaches have been developed: *de novo* and *mapping* (there are some hybrid techniques that have properties both of de novo and mapping assembling techniques).

De novo assembling tries to reproduce (assemble) genomes purely from reads (and associated metadata like quality), without the access to *DNA reference* - string of C, T, A and Gs that is believed to be correct representation of a DNA strand of a single of particular specie.

Mapping assembling has the access to DNA reference of the same specie.

Fact that singles of same specie share large amount of genome material (by (hgp), every two human beings share 99% of genome material) allows us to compare difference between DNA reference of the specie (if it exists) and output of assembly as a success measure.

Assembler explained in this document is pure *de novo* assembler.

2.3. Overlap-layout-consensus assembly

The *Overlap-layout-consensus assembly* (shorter OLC) method was developed for the first generation of longer-read ($\sim 400 - 800bp$) sequencing technologies (Miller et al., 2010). It is based on mathematical concept of graph in order to use existing graph algorithms and simplify assembly process. Assembly process usually consists of three main phases:

Overlap - finding overlaps between reads

Layout - simplifying graph built from overlaps and linearizing graph as much as possible

Consensus - resolving ambiguities from layout phase and finishing the assembly.

Phases definitions are not very clear and assemblers usually have some phases in-between. PBcR pipeline (run), for example, runs meryl module that calculates k-mers (sequences of k consecutive bases) frequencies and decides which k-mers are bringing new information into dataset, and which of them are information overhead.

Nature of OLC definition allows very granular phase decoupling in such way that assembler (and each phase) is built of several modules. Having that, people also developed hybrid OLC assemblers (hyb) built of these modules, using different sequencing technologies in the same assembly process.

Chapter 3 describes methods used for implementing layout phase of *ra-integrate* assembler, assembler developed as part of this thesis.

2.4. Overlapper goal

Since we are not covering overlapper implementation in this thesis, we will just bring main overlapper goal in short. Overlapper is an algorithm (or program) that finds overlaps between given reads. It should report overlaps between reads that are adjacent in real genome (true positive) and it should not report overlaps if they are not adjacent (false positives). As described, *de novo* process does not have reference to the original genome (if it did, we would not need assembler at all) and there is no way to be sure whether some reads are really adjacent or not. It shifts the detection process from checking adjacency to similarity comparison and adjacency prediction. Over time, several techniques have been developed, but some of them are very ineffective when tackling high error-rate data like ONT (or PacBio) sequences. As part of this thesis we use *owler* module of GraphMap, overlapper leveraging modified seed technique, allowing errors on some positions (owl).

2.5. Overlap types

Like any other information, overlaps have to be modeled (described) somehow. We use different models in different stages of OLC pipeline. At the start of processing (right after *owler*) we use MHAP format (mha), which describes overlapper in a very simple and concise way. One overlap records looks like

```
[A ID] [B ID] [Jaccard score] [# shared min-mers]
    [0=A fwd, 1=A rc] [A start] [A end] [A length]
    [0=B fwd, 1=B rc] [B start] [B end] [B length]
```

Described in short, it brings IDs of reads, some scores that should tell us how good reads are overlapping and for each read:

- overlap start position (on read)
- overlap end position (on read)
- read length (redundant information if having set of reads)
- whether original read or its reverse complement is part of overlap.

Having overlaps defined in this form brings simplicity into handling input and output, but is a bad fit for string graph algorithms (Myers, 2005), or any graph representation derived from string graph. Because of that, we have developed a technique for transforming these overlaps into dovetail overlaps, described in Chapter 3.
"A 'dovetail overlap' is one where each fragment has exactly one of its ends in the alignment, and the alignment begins at one fragment end and continues until the other fragment end." (ove). In our case, *fragment* is just a read.

All dovetail overlaps are one of next four types:

1. suffix-suffix (a_{hang} positive, b_{hang} positive)

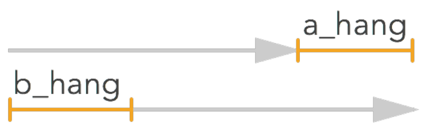

Figure 2.3: Suffix-suffix dovetail overlap.

2. prefix-prefix (a_{hang} negative, b_{hang} negative)

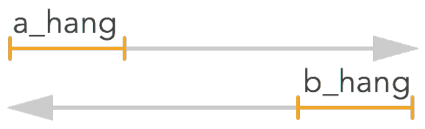

Figure 2.4: Prefix-prefix dovetail overlap.

3. suffix-prefix (a_{hang} positive, b_{hang} positive)

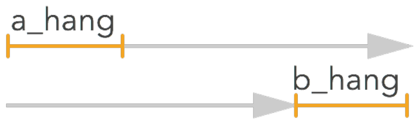

Figure 2.5: Suffix-prefix dovetail overlap.

4. prefix-suffix (a_{hang} negative, b_{hang} negative)

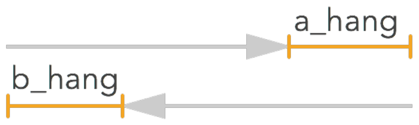

Figure 2.6: Prefix-suffix dovetail overlap.

Orientation left to right means that overlap is using read in the original form, while right to left means read is used as a reversed complement. a_{hang} is number of bases in overlap after read a, while b_{hang} is number of bases in overlap before read b.

All other overlaps can be transformed to one of presented types without any information loss.

Innie overlap is dovetail overlap which uses second read in form of a reversed complement (SS and PP).

3. Layout methods

This chapter covers most important methods used for implementing layout phase of *ra-integrate*. Methods are presented in the order of execution, from first filters to finding unitigs and contigs, giving a way for implementing very similar layout phase in any modern programming language.
General idea of all presented methods is to remove information overhead so resulting string graph is as simple as it can be, without losing structure of read DNA.

3.1. Converting overlaps to dovetail overlaps

Original string graph implementation requires overlap dataset to only consist of dovetail overlaps. Previous chapter clearly describes that we do not have such overlaps so conversion has to be done before we proceed to certain algorithms (string graph creation, filtering contained reads, filtering transitive overlaps).

Visually, it is a very simple algorithm shown on the image 3.1.

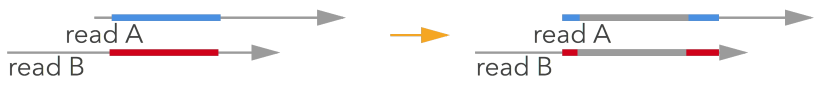

Figure 3.1: Dovetailing algorithm.

Figure 3.2 represents overlap before and after conversion to dovetail overlap. Colored parts on left side are representing original overlap, while colored parts on the right side are representing new parts of overlap, added by “dovetailing” algorithm.

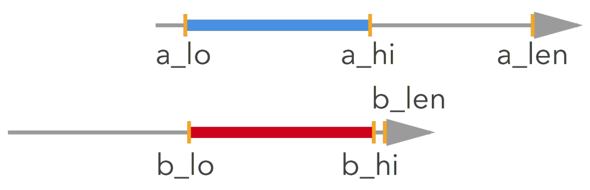

Figure 3.2: Overlap coordinates.

3.1.1. Input

Overlap start and end coordinates

3.1.2. Output

Hangs for dovetail overlaps.

3.1.3. Algorithm

Having coordinates defined as on Figure 3.2, algorithm implementation is pretty simple:

Algorithm 1 Algorithm for dovetailing overlaps

```
1: function CALCULATEFORCEDHANGS($a\_lo, a\_hi, a\_len, b\_lo, b\_hi, b\_len$)
2:     $hangs \leftarrow new\_pair$
3:     $hangs.first \leftarrow a\_lo - b\_lo$                    ▷ $a\_hang$
4:     $b\_after \leftarrow b\_len - b\_hi$          ▷ number of bases read $b$ has after overlap
5:     $a\_after \leftarrow a\_len - a\_hi$          ▷ number of bases read $a$ has after overlap
6:     $hangs.second \leftarrow b\_after - a\_after$             ▷ $b\_hang$
7:     return $hangs$
8: end function
```

3.1.4. Analysis

Time and memory complexity are both $O(1)$ (memory allocation and a few arithmetic operations)

The algorithm itself is very fast, but it can introduce an error if insertions or deletions happened during sequencing phase. One of next subchapters explains how to partially correct introduced error.

3.2. Filtering contained reads

Filtering contained reads is a method that removes all reads contained in some other read from input dataset. Like previous chapter describes, sequences in dataset are of different lengths, which makes following situation possible:

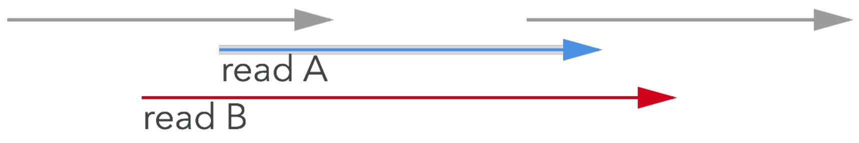

Figure 3.3: Read A is contained in read B.

Having that, read A can be removed without losing any information about genome since read B and coincident overlaps contain super-set of read A information. Removing read A is equivalent to removing all overlaps coincident with read A (grey on picture above). Overlaps of contained reads should not be removed permanently, though. Keeping information about them could be very useful in consensus phase.

3.2.1. Input

Array of dovetail overlaps.

3.2.2. Output

Array of overlaps coincident with non-containment reads.

3.2.3. Algorithm

Algorithm 2 Filtering overlaps coincident with contained reads

```
 1: function FILTERCONTAINED($overlaps$)
 2:     $contained\_reads \leftarrow new\_set$                          ▷ initialise set
 3:     for $o \leftarrow overlaps$ do
 4:         if $o.a\_hang \leq 0$ and $o.b\_hang \geq 0$ then
 5:             $contained\_reads \leftarrow o.a$    ▷ adds read $a$ to $contained\_reads$ set
 6:         else if $o.a\_hang \geq 0$ and $o.b\_hang \leq 0$ then
 7:             $contained\_reads \leftarrow o.b$    ▷ adds read $b$ to $contained\_reads$ set
 8:         end if
 9:     end for
10:     $idx \leftarrow 0$
11:     for $i \leftarrow 0, overlaps.size$ do
12:         $o \leftarrow overlaps[i]$
13:         if $o.a$ in $contained\_reads$ then
14:             continue
15:         end if
16:         if $o.b$ in $contained\_reads$ then
17:             continue
18:         end if
19:         $overlaps[idx] \leftarrow overlaps[i]$              ▷ relocate valid overlaps
20:         $idx \leftarrow idx + 1$
21:     end for
22:     $overlaps.resize(idx)$                         ▷ remove all other overlaps
23: end function
```

We check if the combination of hanging values shows that read is contained. If it is, we mark read as contained.

In second pass, we relocate overlaps coincident with non-contained reads to the start of overlaps array, increasing position for next overlap each time.

At the end, we truncate all overlaps located after position idx.

3.2.4. Analysis

Having this method implemented as part of layout pipeline is of utmost importance. It usually removes big portion of dataset, leaving the rest of the pipeline working with an order of magnitude smaller dataset (shown in Chapter 5); ideally without losing any information.

Time complexity of shown algorithm is $O(N)$, where N is number of given overlaps.

It is important to notice that every step of first loop is completely localised, interacting only with current element. That allows us to introduce trivial parallelisation and speed up the process.

The other pass, however, shares the access to idx value so it cannot be parallelised that way.

3.3. Tuning dovetail overlaps

The algorithm written for converting arbitrary overlaps to dovetail overlaps forces extended overlapped parts on reads to be of same length, which is rarely correct. Approximate overlap coordinates are good enough for most algorithms (filtering contained reads, building string graph), but can be misleading for algorithms that are quite sensitive to coordinate changes (like calculating overlap error rate).

The algorithm for tuning dovetail overlaps is the extension of dovetail conversion algorithm. After the initial conversion happens, algorithm tries to shorten/extend hanging parts of overlap.

Visually, it looks like moving blue dot left or right until an optimal solution is found; shown on Fig. 3.4. Red part is utilized as a whole.

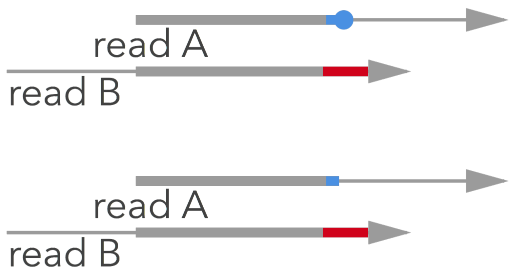

Figure 3.4: Representation of overlap "tuning".

The very same thing needs to be executed for the other end of the overlap (middle of read B, start of read A).

Optimal solution for this problem can be found using dynamic programming with algorithm very similar to the *Needleman–Wunsch* algorithm(Cammack et al.), but modified in a way it allows gaps at the end of a query (on the picture above read B i target and read A is query).

Given algorithm (space and time complexity $O(N^2)$) would be too slow for our use case so we use modified problem - we change overlap coordinates in order to optimize *edit distance* (edi) between overlapped parts; all using fast algorithm for edit distance calculation (Myers, 1999) implemented in `https://github.com/Martinsos/edlib`.

Downside of using algorithm that optimizes edit distance, instead of score in the NW algorithm, is eagerness to find shorter overlaps as a result.

3.3.1. Input

An overlap defined with start and end coordinates for both reads.

3.3.2. Output

Dovetail overlap with more accurate coordinates.

3.3.3. Algorithm

We calculate "rough" dovetail overlap with method from this chapter.

After that, depending on overlap type (SP, PS, SS, PP), we call helper method that tries to shrink given overlap on hanging ends.

Having positions tuned, we calculate hangs for new overlap, overlap error rate and original overlap error rate. Knowledge of error rates before and after dovetailing can help us detect overlaps that introduce new error during dovetailing process.

Algorithm 3 Tuning dovetail overlap coordinates

```
1: function GETTUNEDOVERLAP($overlap$)
2:     $rough\_hangs \leftarrow$ CALCULATEFORCEDHANGS($o.a\_lo, o.a\_hi,$
           $o.read\_a\_length, o.b\_lo, o.b\_hi, o.read\_b\_length$)
3:     $tmp \leftarrow$ OVERLAP($o.read\_a, rough\_hangs.first,$
           $o.read\_b, rough\_hangs.second, o.is\_innie$)        ▷ temporary overlap we use to detect overlap type
4:     if $tmp.is\_using\_suffix(a)$ and $tmp.is\_using\_prefix(b)$ then
5:         STRETCHSUFFIXPREFIXOVERLAP($o, \&new\_a\_lo, \&new\_a\_hi,$
               $\&new\_b\_lo, \&new\_b\_hi, \&added\_edit\_distance$)
6:     else if $tmp.is\_using\_prefix(a)$ and $tmp.is\_using\_suffix(b)$ then
7:         STRETCHPREFIXSUFFIXOVERLAP($o, \&new\_a\_lo, \&new\_a\_hi,$
               $\&new\_b\_lo, \&new\_b\_hi, \&added\_edit\_distance$)
8:     else if $tmp.is\_using\_prefix(a)$ and $tmp.is\_using\_prefix(b)$ then
9:         STRETCHPREFIXPREFIXOVERLAP($o, \&new\_a\_lo, \&new\_a\_hi,$
               $\&new\_b\_lo, \&new\_b\_hi, \&added\_edit\_distance$)
10:    else if $tmp.is\_using\_suffix(a)$ and $tmp.is\_using\_suffix(b)$ then
11:        STRETCHSUFFIXSUFFIXOVERLAP($o, \&new\_a\_lo, \&new\_a\_hi,$
               $\&new\_b\_lo, \&new\_b\_hi, \&added\_edit\_distance$)
12:    end if
13:    $orig\_err\_rate \leftarrow orig\_edit\_distance/o.length$
14:    $err\_rate \leftarrow (orig\_edit\_distance + added\_edit\_distance)/$
           $(0.5 * (new\_a\_hi - new\_a\_lo + new\_b\_hi - new\_b\_lo))$
15:    $real\_hangs \leftarrow$ CALCULATEFORCEDHANGS($new\_a\_lo, new\_a\_hi,$
           $o.read\_a\_length, new\_b\_lo, new\_b\_hi, o.read\_b\_length$)
16:    return OVERLAP($o.read\_a, real\_hangs.first,$
           $o.read\_b, real\_hangs.second, o.is\_innie$)        ▷ final overlap
17: end function
```

Method for fine tuning suffix-prefix overlap type looks like:

Algorithm 4 Calculating exact overlap edges

```
1: function STRETCHSUFFIXPREFIXOVERLAP($o, new\_a\_lo, new\_a\_hi,$
          $new\_b\_lo, *new\_b\_hi, *edit\_distance$)
2:     $query\_used\_bases \leftarrow -1$
3:
4:     $new\_a\_hi \leftarrow o.read\_a\_length$
5:     $target \leftarrow o.read\_b.sequence$
6:     $query \leftarrow o.read\_a.sequence$
7:     $x\_edit\_distance \leftarrow$ EDITDISTANCESHW($query,$
          $o.a\_hi, target, o.b\_hi, \&query\_used\_bases$)
8:     $new\_b\_hi \leftarrow o.b\_hi + query\_used\_bases$
9:
10:    $new\_b\_lo \leftarrow 0$
11:    $target \leftarrow$ REVERSE($o.read\_a.sequence.substr(0, o.a\_lo)$)
12:    $query \leftarrow$ REVERSE($o.read\_b.sequence.substr(0, o.b\_lo)$)
13:    $o\_edit\_distance \leftarrow$ EDITDISTANCESHW($query, 0,$
          $target, 0, \&query\_used\_bases$)
14:    $new\_a\_lo \leftarrow target.length - query\_used\_bases$
15:
16:    $edit\_distance \leftarrow x\_edit\_distance + o\_edit\_distance$
17: end function
```

Other called methods are omitted in order to keep the length of the thesis readable.

EditDistanceSHW, method that calculates the optimal position for an overlap end, has an interface is defined as:

```
int32_t editDistanceSHW(string& query, uint32_t query_lo,
    string target, uint32_t target_lo, int* query_best_end);
```

In words:

- *query_lo* is the position of the first character used in query
- *target_lo* is the position of the first character used in target
- *query_lo* and *target_lo* can be always set to 0, if preceded with extracting exact substrings.

Underlying code calls SHW mode of edlib library, which performs semi-global alignment. SHW mode does not penalize gap at the end of query.

Processing x part from the code above will use colored parts of strings, like shown on Fig. 3.5.

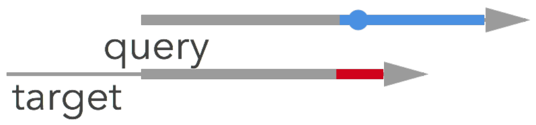

Figure 3.5: Extending/shrinking an overlap end during "tuning" phase.

$query\_best\_end$ is actually the position of blue dot that can be placed anywhere on the blue line.

Actual implementation can be found on `https://git.io/vz1Zw`.

3.3.4. Analysis

The time complexity is still $O(NxM)$ (where N and M are lengths of strings passed to SHW function), but the actual implementation is fast because it is using fast Myers algorithm ($NxM/64$ on 64-bit computers).

Unfortunately, edit distance is a bad choice for scoring metric because it always tries to shorten the overlap (except when we have a perfect overlap), which is almost of no use here.

This algorithm can be omitted from a pipeline without major effect on final results.

3.4. Filtering transitive overlaps

Another example of information overhead is transitive overlap. One of them is shown on Figure 3.6.

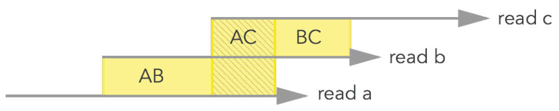

Figure 3.6: Overlap AC is transitive to AB and BC.

Overlap AC is transitive to overlaps AB and BC. Removing overlap AC will not break any walks in string graph so it can be safely removed. Although, existence of transitive overlap AC lowers the probability of overlaps AB and BC being falsely reported overlaps.

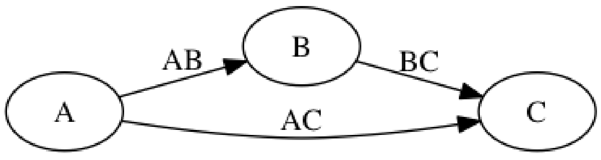

Figure 3.7: String graph representing transitive overlap shown on Fig. 3.6.

Removing transitive overlaps can be done before the string graph construction or as part of bubble popping algorithm (explained later in this chapter). Ra-integrate is not started from scratch and we have inherited code for removing transitive overlaps.

Detailed algorithm explanation and analysis can be found in (Rahle, 2014), where author explains his implementation of position based algorithm for transitive reduction (Simpson i Durbin, 2010), improved with merge-sort technique.

3.4.1. Input

List of dovetail overlaps.

3.4.2. Output

List of non-transitive dovetail overlaps.

3.4.3. Analysis

Time complexity is $O(MK)$, where M is number of all overlaps and K average number of overlaps per read. We expect number K to be approximately the same as coverage of dataset.

3.5. Filtering short overlaps

During development and initial testing of *ra-integrate*, it turned out to be very useful to filter overlaps that cover very small portion of reads. Hereof, we defined algorithm that removes all overlaps covering less than X% of any coincident read. This filter is the simplest filter and removes some amount of noise (example will be given in Chapter 5). Algorithm implementation is trivial and will not be explained. Implementation can be found on `https://git.io/vz1cL`.

3.6. Filtering erroneous overlaps

Assembling genomes from error-prone reads brings many challenges. Beside difficulties with finding any overlaps, it is also difficult to decide which overlaps are real and which are falsely reported. Since *owler* reports some number of false positive overlaps, we have to filter them out to ensure we are dealing with correct data as much as we can.

A simple technique we use here is removing all overlaps with error rate higher than X. X is a value provided to the algorithm and it needs to be chosen based on overlaps error rate distribution.

Error rate is calculated during “tuning” dovetail overlaps, defined as:
$error\_rate = overlap\_edit\_distance/overlap\_length$.

3.6.1. Input

List of overlaps, X.

3.6.2. Output

List of overlaps, without overlap with error rate higher than X.

3.6.3. Algorithm

Algorithm 5 Algorithm for filtering erroneous overlaps

```
 1: function FILTERERRONEOUSOVERLAPS($overlaps, x$)
 2:     $idx \leftarrow 0$
 3:     for $i \leftarrow 0, overlaps.size$ do
 4:         if $overlaps[i].error\_rate \geq X$ then
 5:             continue
 6:         end if
 7:         $overlaps[idx] \leftarrow overlaps[i]$
 8:         $idx \leftarrow idx + 1$
 9:     end for
10:     $overlaps.resize(idx)$                    ▷ Remove the rest of overlaps
11: end function
```

3.6.4. Analysis

Time complexity of written algorithm is $O(N)$, where N is number of overlaps. While the algorithm is simple, choosing right X can be quite difficult. For purposes of this thesis, we analysed datasets by hand and picked X for each dataset to get decent results. Better solution would be automated analysis of a dataset and choosing X that would remove noise (and ideally just noise).

3.7. Graph creation

So far we were doing fine without having any graph implementation. For following algorithms it is handy to have our data structured as a graph. For the purpose of this thesis, we are using modified implementation of Robert Vaser's graph (Vaser, 2015) because it was part of project we have inherited.

Different from String Graph implementation, every read is represented with one node (instead of one node for read start and one node for read end). Every overlap is represented with two edges: one for $A > B$ and another one for $B > A$.

Also:

- all vertices are stored in vertices collection
- all edges are stored in edges collection
- every edge has its pair edge (pair edges are built from same overlap)
- every edge is added to its source vertex, via $vertex.add\_edge$ method.

If we take an example from subchapter explaining transitive overlaps, our graph is shown on Fig. 3.8.

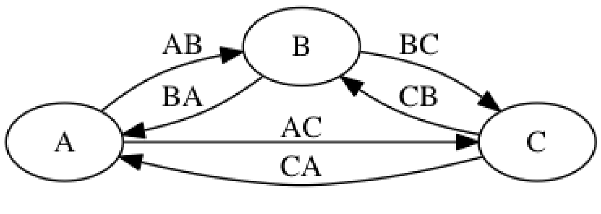

Figure 3.8: String graph example.

Internally, every node keeps two collections of edges, one for edges that use read's start ($edges\_b$) and another one that keeps edges that use read's end ($edges\_e$); for the sake of faster iteration.

3.7.1. Input

Vector of overlaps, vector of reads.

3.7.2. Output

String graph.

3.7.3. Algorithm

For the sake of simplicity, we assume reads are having ids $0, 1, 2, 3...$

Algorithm 6 Algorithm for creating a graph from overlaps

```
1:  function CREATEGRAPH($reads, overlaps$)
2:      $graph \leftarrow new\_graph$
3:      for $i \leftarrow 0, reads.size$ do
4:          $graph.vertices \leftarrow$ VERTEX($reads[i]$)            ▷ adds a new vertex
5:      end for
6:      for $i \leftarrow 0, overlaps.size$ do
7:          $o \leftarrow overlaps[i]$
8:          $edge\_a \leftarrow$ EDGE($graph.edges.size, o, o.a$)    ▷ $edge\_id, overlap, source$
9:          $graph.edges.push\_back(edge\_a)$
10:
11:         $edge\_b \leftarrow$ EDGE($graph.edges.size, o, o.b$)
12:         $graph.edges.push\_back(edge\_b)$
13:
14:         $graph.vertices[o.a].edges.push\_back(edge\_b)$    ▷ adds $edge\_b$ to vertex
    made from read $a$
15:         $graph.vertices[o.b].edges.push\_back(edge\_a)$
16:
17:         $edge\_a.pair \leftarrow edge\_b$                        ▷ link edges
18:         $edge\_b.pair \leftarrow edge\_a$
19:     end for
20:     return $graph$
21: end function
```

As seen in code, every edge accepts following arguments on construction: *edge_id*, *overlap*, *source* (destination can be calculated from overlap and source).

Actual implementation of add_edge:

Algorithm 7 Adding an edge to a vertex

```
1: function VERTEX.ADDEDGE($edge$)
2:     $using\_suffix \leftarrow$ EDGE.OVERLAP.IS_USING_SUFFIX($this.get\_id$)
3:     if $using\_suffix = true$ then
4:         $edges\_e$.PUSH_BACK($edge$)
5:     else
6:         $edges\_b$.PUSH_BACK($edge$)
7:     end if
8: end function
```

is_using_suffix returns true if overlap uses given reads suffix (considering original sequence even if read is used as reveresed complement).

3.7.4. Analysis

Given algorithm for graph creation executes within space and time complexity of $O(N + M)$, where N is number of nodes, and M is number of overlaps.

3.8. Graph simplification

Pseudo code below represents overall method for graph simplification. Non-obvious methods are explained in following subchapters.

Algorithm 8 High-level graph simplification algorithm

```
1: function SIMPLIFY($graph$)
2:     while graph changes do
3:         REMOVENODESWITHNOEDGES($graph$)
4:         TRIM($graph$)
5:         POPBUBBLES($graph$)
6:     end while
7: end function
```

3.9. Trimming (tips)

Due to errors in sequencing process, it is possible that our graph looks like on Fig. 3.9.

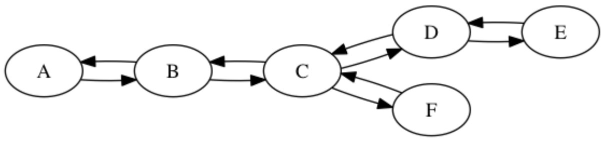

Figure 3.9: Example of tip in string graph (node F).

Node F is considered as a tip and it is usually very safe to remove it. We are omitting explanation of tip removal algorithm since we use inherited implementation from RA project. Original explanation can be found in (Vaser, 2015).

3.9.1. Input

String graph.

3.9.2. Output

String graph without tips.

3.10. Bubble popping

After having all previous steps done, majority of noise and redundancy should have been removed. However, it is possible that we have situation shown on image 3.10.

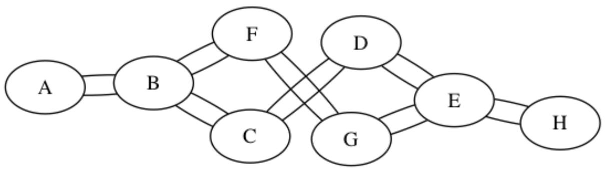

Figure 3.10: Simple bubble.

Structure made of walks $B - F - G - E$ and $B - C - D - E$ is called a bubble. It usually occurs because of sequencing errors and it is very possible that error-prone reads will lead to graphs with many bubbles.

Also, when dealing with error-prone reads without correction, another possible, more complex situation is shown on Fig. 3.11.

3.10.1. Input

String graph.

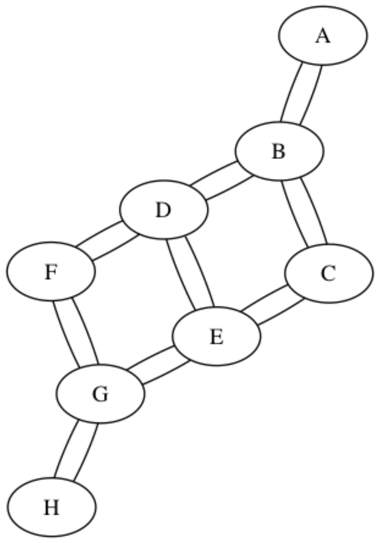

Figure 3.11: Complex bubble.

3.10.2. Output

String graph without bubbles.

3.10.3. Algorithm

The algorithm we have created is effective in removing both types of bubbles.

It tries to find a bubble starting from every node that has more than one outgoing edge (on first node example nodes B and E). Such node will be called *start node*. A bubble is considered to be found when all possible walks starting from *start node* can end in some other node, called *sink*.

Finding a bubble starts with creating a walk for every outgoing edge of *start node*. Using the second bubble example (Fig. 3.11) and having node B as *start node*, it creates walks $B - C$ and $B - D$. In every iteration each walk is extended by one edge, and forked if needed. All walks are placed in some collection so we can backtrack later.

In first iteration, walk $B - C$ is extended to $B - C - E$, while $B - D$ is forked and extended to $B - D - E$ and $B - D - F$; our walks are [$B - C - E$, $B - D - E$, $B - D - F$]. At the end of iteration we check if *sink* is found (it is not this case).

In second iteration we extend our walks to [$B - C - E - G$, $B - D - E - G$, $B - D - F - G$], respectively. Node G is *sink* because all walks can end in node G.

Detecting a *sink* is implemented by counting how many walks end in certain node. When N is current size of walks collection and N walks can end in node X, X is considered as the *sink*. It is heuristics that allows fast detection of bubbles where walks are not of same length (imagine having a way from G to E with a node in the middle), but is error prone to graphs with loops (a loop can end in node X multiple times and trick algorithm to decide it is a *sink*). We have not hit a problem on datasets we used using this algorithm and trade-off we made using this heuristics proved as a good one.

Algorithm has to be stopped at some point so we are introducing MAX_STEPS constant that terminates the algorithm if nothing is found after MAX_STEPS iterations. Having that, we do not have to tackle loops as separated edge case.

Once we detect node with more than one outgoing edge, we call an algorithm defined with following pseudo-code:

Algorithm 9 Finding bubbles in string graph

```
 1: function FINDBUBBLES(Node $start\_node$, bool $direction$)
 2:     $walks \leftarrow new\_list$
 3:     $ends\_in \leftarrow new\_hash$          ▷ $ends\_in[x]$ number of paths ending in $x$
 4:     if $direction = 1$ then
 5:         $edges \leftarrow start\_node.edges\_e$
 6:     else
 7:         $edges \leftarrow start\_node.edges\_b$
 8:     end if
 9:     $walks.push\_back($ WALK(start_node) )
10:     for $i \leftarrow 0, MAX\_STEPS$ do
11:         $dead\_walks \leftarrow 0$                    ▷ Reset dead walks counter
            ▷ Set loop size upfront so new walks are skipped in the same iteration when
    they are added
12:         $walks\_size \leftarrow walks.size$
13:         if $walks\_size > MAX\_WALKS$ then
14:             break
15:         end if
16:         for $j \leftarrow 0, walks\_size$ do
17:             $walk \leftarrow walks[j]$
18:             $extended\_walks \leftarrow$ EXTENDWALK($walk$)
19:             if $extended\_walks.size = 0$ then          ▷ Walk is dead end
20:                 $dead\_walks = dead\_walks + 1$
21:                 continue
22:             end if
23:             $walks[j] = new\_walks[0]$       ▷ replace old walk with extended one
24:             for $k \leftarrow 1, new\_walks.size$ do ▷ add walks created by fork to the end
25:                 $walks.push(new\_walks[k])$
26:             end for
```

```
27:             for $new\_walk \leftarrow new\_walks$ do
28:                 $ends\_in[new\_walk.head] ++$          ▷ Update counter for walk ends
29:                 if $ends\_in[new\_walk.head] = walks.size$ then
30:                     return $walks$                        ▷ Bubble is found
31:                 end if
32:             end for
33:         end for
34:         if $dead\_walk = walks\_size$ then
35:             break
36:         end if
37:     end for
38:     return []
39: end function
```

Function $extend\_walk$ returns extended walk; or walks if fork happened (*walk* is represented with a start node and following edges that form a walk). In example we have, for walk $B-C$ it returns $[B-C-E]$, while for $B-D$ returns $[B-D-E, B-D-F]$ (because D has 2 outgoing edges). If $extend\_walk$ can not extend given walk, it returns [] and that walk is marked as *dead walk* (lines 19-21).

Once we have all walks forming a bubble (function $popBubble$ returns array of walks), we shorten them to look like $start - ... -$ *sink*. In example we have above there is no need for that since all walks end in *sink* node - node G.

Next step is to choose a *base walk*. Base walk is a walk we believe to be the most correct walk that truly represents that part of genome. As metric of correctness we use total walk coverage, defined as sum of read coverages on that walk.

After choosing the base walk, we have to check if it is a bubble that represents one part of DNA strand or genome really has such structure (diploid genomes). For that purpose, we use new metric $difference = edit\_distance/length$ - between every walk and base walk, having defined maximum error that we allow - Y. Maximum allowed error should depend on dataset properties and expected error rate. If $difference > Y$, we mark all edges on that walk for removal.

Edge e can be removed from a graph once we are sure that we wanted to remove all the walks that could use edge e.

Actual code is quite complicated so we are bringing simplified pseudo-code. Real implementation can be found on `https://git.io/vgce0`.

Algorithm 10 Popping a bubble from given walks

```
1: function POPBUBBLE([]walk $bubble\_walks$, double $Y$)
2:     $best\_coverage \leftarrow 0$
3:     for $walk \leftarrow bubble\_walks$ do                          ▷ Find the base walk
4:         if $coverage(walk) > best\_coverage$ then
5:             $best\_coverage \leftarrow coverage(walk)$
6:             $base\_walk \leftarrow walk$
7:         end if
8:     end for
9:     for $walk \leftarrow bubble\_walks$ do                          ▷ Find the base walk
10:        if $walk = base\_walk$ then
11:            continue
12:        end if
13:        if DIFFERENCE($walk, main\_walk$) $> Y$ then
14:            continue
15:        end if
16:        for $edge \leftarrow walks.edges$ do
17:            MARKFORREMOVAL($edge$)
18:            if REMOVEDFROMALLPATHS($edge$) then
19:                REMOVEEDGE($edge$)
20:            end if
21:        end for
22:    end for
23: end function
```

Considering code complexity, bubble popping turned out to be the most complex method.

In chapter about filtering transitive overlaps we noted that the same goal can be done with bubble popping algorithm. From Fig. 3.7 it is very obvious that every transitive overlap (with two other overlaps it is transitive to) would form a bubble, but the algorithm for removing bubbles is far slower in detection and removal than the algorithm dedicated just for transitive overlaps removal.

3.10.4. Analysis

As said, bubble popping is the most complex algorithm in layout phase, time complexity wise, memory complexity wise and implementation complexity wise. Since we use MAX_STEPS parameter, we limit global number of expanded paths with MAX_WALKS argument. Still, complexity for finding a potential bubble is $O(N * MAX\_WALKS)$, where N stands for number of nodes in a graph. Once potential bubble is found, we have to compare each walk to base walk. That comparison takes $O(MAX\_WALKS * M^2)$ (one comparison per path), where M is maximal path length (base-wise, not node-wise length).
However, algorithm proved to be very efficient and it pops tens of bubbles in datasets used, as shown in Chapter 5.

3.11. Extracting unitigs

Once we have done everything we can to remove noise and linearise graph into a single walk, next step is to extract high-fidelity sequences - unitigs. As seen in Chapter 5, it is common that previous steps untangled the graph in a way that makes some unitigs pretty obvious. Extracting a part of graph could give us image like shown on Figure 3.12.

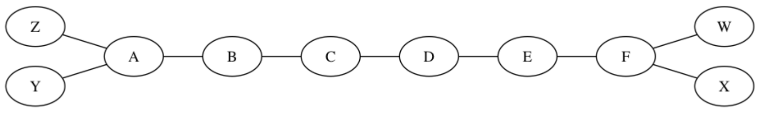

Figure 3.12: Obvious unitig.

With assumption that we have removed all false positive reported overlaps from our dataset, sequence created from walk $A - B - C - D - E - F$ is high-fidelity because there are no forks on mentioned walk. Therefore, walk $A-B-C-D-E-F$ makes an unitig.

Also, there are some less obvious potential unitigs. Celera assembler (available at `http://wgs-assembler.sourceforge.net/wiki/index.php/PBcR`) is using best overlap graph, graph where some forks are resolved if two nodes in a fork are "best buddies" to each other. In that case, fork is linearised and best buddies become part of the same unitig.

Two nodes A and B are best buddies if overlap with another node is the longest overlap for that side of node.

3.11.1. Input

String graph.

3.11.2. Output

List of unitigs, defined as start read and overlaps.

3.11.3. Algorithm

Algorithm is quite simple. At the very start, every node is its own unitig. Iteration over nodes checks if node is already part of some known unitig, and if not, it starts building a new one. Process goes in both directions.

At the end, every node is part of one and only one unitig.

Algorithm 11 Extracting unitigs

```
1: function EXTRACTUNITIGS($graph$)
2:     $visited\_nodes \leftarrow new\_hash$
3:     $unitigs \leftarrow new\_list$
4:     for $node \leftarrow graph.nodes$ do
5:         if $node$ in $visited\_nodes$ then
6:             continue
7:         end if
8:         $edges \leftarrow new\_list$
9:         GETEDGES($edges, visited\_nodes, node, DIRECTION\_LEFT$)
10:        REVERSE($edges$)    ▷ Reverse edges because we will return unitig going in other direction
11:        GETEDGES($edges, visited\_nodes, node, DIRECTION\_RIGHT$)
12:        $unitigs.add(edges.first.src, edges)$
13:    end for
14:    return $unitigs$
15: end function
```

3.11.4. Analysis

Time complexity of written algorithm is $O(N)$, where N is number of nodes. Algorithm visits every node exactly once. Memory complexity of the algorithm is also $O(N)$, since every node will become part of one and only one unitig.

```
16: function GETEDGES($dst\_edges, visited\_nodes, start, start\_direction$)
17:     $use\_suffix \leftarrow start\_direction$
18:     $curr\_node \leftarrow start$
19:     while $true$ do
20:         $visited\_nodes.add(curr\_node)$
21:         $edge \leftarrow curr\_node.best\_edge(use\_suffix)$
22:         if $edge = null$ then
23:             continue
24:         end if
25:         if $edge.overlap.is\_innie$ then     ▷ change direction if overlap is SS or PP
26:             $use\_suffix \leftarrow 1 - use\_suffix$
27:         end if
28:         $next \leftarrow edge.dst$
29:         if $next.best\_edge(1 - use\_suffix).overlap()! = edge.overlap$ then
30:             break                            ▷ Break if curr and next are not best buddies
31:         end if
32:         $edges.push\_back($EDGE$(edge))$
33:         if $next$ in $visited\_nodes$ then
34:             break                            ▷ If read is already part of some other unitig
35:         end if
36:         $curr\_note \leftarrow next$
37:     end while
38: end function
```

3.12. Extracting contigs

Contigs are sequences that are wild guesses of genome final image. Usual procedure would be to create unitig graph (graph made from unitigs and edges between them) and extract unitigs as walks through that graph. Unfortunately, during work on this thesis there was not enough time to develop such a method so we are bringing modified high-level explanation of modified method inherited from Ra project. Described method does not have any sense of unitigs and it works exclusively on string graph.

3.12.1. Input

String graph.

3.12.2. Output

Contig, defined as start node and edges.

3.12.3. Algorithm

Algorithm 12 Extracting contigs

```
1: function EXTRACTCONTIG($graph$)
2:     $start\_candidates \leftarrow$ FINDSTARTCANDIDATES($graph$)
3:     for $candidate \leftarrow start\_candidates$ do
4:         if $longest\_walk = nil || walk.length > longest\_walk.length$ then
5:             $longest\_walk \leftarrow walk$
6:         end if
7:     end for
8:     return $longest\_walk$
9: end function
```

Function $find\_start\_candidates$ returns up to $MAX\_START\_CANDIDATES$ start candidates. Only forks and tips are considered as good start candidates. Also, there is a case when genome is circular and we get perfect assembly before this step. In that case every node is a good start candidate.

Function $find\_longest\_walk$ is exhaustively seeking for the longest walk under following boundaries:

– every node is visited at most once in a walk

- if there is a fork, quality of used overlap has to be inside certain boundaries, derived from best overlap quality ($1 - QUALITY\_THRESHOLD$)
- for the sake of speed and feasibility, algorithm stops after MAX_FORKS number of forks

If last boundary was not there, function $find\_longest\_walk$ would try to find every possible walk in the graph in order to return the longest one. Since complexity of this problem has exponential nature, we added the last boundary to limit the execution time (the algorithm complexity stayed the same).

Original implementation can be found on `https://git.io/vz1Xy`.

3.12.4. Analysis

Time complexity of algorithm is exponential, since it is trying to find the longest walk in a graph.

3.13. General layout algorithm

At the end of this chapter, we are bringing pseudo-code of high-level algorithm:

Algorithm 13 High-level layout algorithm

```
1: function LAYOUT
2:     $reads \leftarrow$ READREADS
3:     $overlaps \leftarrow$ READOVERLAPS
4:     CONVERTTODOVETAIL($overlaps$)
5:     FILTERCONTAINERREADS($overlaps$)
6:     TUNEDOVETAILOVERLAPS($overlaps$)
7:     FILTERTRANSITIVEOVERLAPS($overlaps$)
8:     FILTERSHORTOVERLAPS($overlaps$)
9:     FILTERERRONEOUSOVERLAPS($overlaps$)
10:    $graph \leftarrow$ CREATEGRAPH($reads, overlaps$)
11:    SIMPLIFYGRAPH($graph$)
12:    WRITEUNITIGS($graph, file$)
13:    WRITECONTIGS($graph, file$)
14: end function
```

4. Implementation

This chapter covers rough overview of implementation decomposition, used technologies, brief story about tangled project history and instructions for running assembly process yourself.

4.1. Project history

History of this project is pretty tangled, but never left rooms and minds of FER. Layout development started with (Rahle, 2014). It consisted just of filtering contained reads and transitive overlaps. After some time Marko Čulinović implemented basic algorithms for trimming and bubble popping, basis for assembling more complicated genomes and dealing with non-perfect datasets. That project was just a rough skeleton of what we have today; code available at `https://git.io/vz1Mm`. Fixing few bugs led us to position where we were able to assemble smaller parts of HIV, but nothing was properly tested and validated. Robert Vaser decided to rewrite the whole project to get his RNA assembler up and running and published his work on `https://git.io/vz1Mz`. This project, made for the purpose of this thesis, is a fork of ra project, coupled with some runner scripts and GraphMap overlapper (*owler* module) into ra-intergrate project. 4 projects and 5 repositories later, inheriting some technical debt from all previous projects, located on `https://git.io/vz1yu` (v0.9), we have an assembler able to assemble Escherichia coli genome in one contig.

4.2. Used technologies

Most of the codebase is written in C++ for the sake of speed and freedom it gives. Some runner scripts are written in Ruby[1] (main script wrapping entire process of assembling

[1] `https://www.ruby-lang.org/en/`

a genome).

We used GoogleTest [2] as unit testing framework.

GraphViz [3] is external dependency used for plotting graphs.

Since project grew over time, we needed something for automated building so we chose Make [4] tool and Travis CI [5] for making sure that every build passes all defined tests.

Docker [6] helps us to provide a "package" for the project and make sure that one does not need to install all dependencies and pollute their environment.

During development, we used Gepard (Krumsiek, 2007) tool to visually check whether assemblies are correct.

4.3. Supported input and output formats

Ra-integrate supports fasta and fastq files for defining reads. Since GraphMap writes results in MHAP format, we read overlaps from MHAP format (hidden in wrapper scripts). Output is written in several files

- unitig sequences (*fasta*)
- contig sequences (*fasta*)
- assembly (AFG [7])
- string graph layout (dot language [8])

4.4. Modules

Ra-integrate (wrapper project) consists of two projects: GraphMap and Ra (forked). Ra project again is decoupled into more than few modules:

unitigger implements bubble popping, trimming, extracting unitigs and contigs

overlap2dot transforms overlaps from depot into dot definition of assembly graph

zoom (debug) takes overlaps and returns neighbourhood of given size for given node

[2] https://github.com/google/googletest
[3] http://www.graphviz.org/
[4] https://en.wikipedia.org/wiki/Make_(software)
[5] https://travis-ci.org/
[6] https://www.docker.com/
[7] http://amos.sourceforge.net/wiki/index.php/Message_Types
[8] http://www.graphviz.org/doc/info/lang.html

filter_contained filters overlaps of contained reads from given set

filter_transitive filters transitive overlaps

widen_overlaps converts MHAP overlaps into dovetail overlaps

filter_erroneous_overlaps filters all overlaps shorter than X% or with error rate higher than Y%

depot import/export tool for centralised depot, used by other components

fill_read_coverage analyzes overlaps in order to calculate coverage for each read

4.5. Stats

Some interesting stats from the project:

- 6 people involved
- 480 commits
- 135,396 lines added, 22,308 lines removed during work (GitHub metrics)
- numerous bugs fixed (unfortunately, we have not tracked them)
- 22,834 lines of code in total (GoogleTest not included).

4.6. Installation and running

The shorter this subchapter is, the better job is done.

Requirements:

- Some distribution of Linux
- Ruby 2.2 or newer
- Make
- g++ (4.8 or later)
- GraphViz (*for getting a string graph representation at the end of assembly*)

You can get and install ra-integrate by typing following command:

```
git clone --recursive \
    https://github.com/mariokostelac/ra-integrate.git
make
```

Once you have your reads ready in *fasta* or *fastq* format, run the assembly process with:

```
./scripts run reads.fasta -s specs_file
```

Defining $specs\_file$ is completely optional. All settings that can be modified in specs file can be found in readme file of ra-integrate project itself.

5. Testing and results

5.1. Hardware and software

All tests were run on Assembly - one of FER computers, whose characteristics are:

1. Hardware
   - Architecture: $x86\_64$
   - Number of CPUs: 2
   - CPU model name: $Intel(R)\ Xeon(R)\ CPU\ E5645$
   - Cores per CPU: 6
   - CPU GHz: 2.40
   - RAM: $296GB$
2. Software
   - OS: $Ubuntu\ 14.04.2\ LTS$
   - Kernel: $3.13.0\text{-}71\text{-}generic$
   - ra-integrate: $v0.9$

All measurements are times extracted from log files created by forwarding stderr stream to a file. Provided times rely on system clock and they are (un)reliable as the unix time tool itself, standard for measurements like this one. Despite of Assembly's shared nature, for the purpose of fairness and correctness, all tests were run when there was no other users, just few background services running (their CPU and memory consumption are not comparable to ra-integrate consumption).

5.2. Datasets

For the purpose of demonstrating effectiveness of implemented methods, we are using two datasets, both results of sequencing samples of Escherichia coli. They are representative examples of datasets from two competitive companies. First dataset showcases very long, single-molecule PacBio data with a sample of a E. coli K12 MG1655 strain. Dataset can be downloaded from `https://git.io/vz1Sn`. Mean length of the sequences is 3549bp and coverage 19.96x.

To show effectiveness on Oxford Nanopore datasets, we decided to use Oxford Nanopore MinION dataset from (Loman et al., 2015). Converted dataset can be found on `http://www.cbcb.umd.edu/software/PBcR/data/sampleDataOxford.tar.gz`, while original set is available on `http://www.ebi.ac.uk/ena/data/view/ERP007108`. Dataset consists of 2D reads prepared with R7.3 chemistry. Mean coverage is 30x.

Sequence length distributions are shown on images 5.1 and 5.2.

Reference sequence can be found on `http://www.ncbi.nlm.nih.gov/nuccore/U00096.2`.

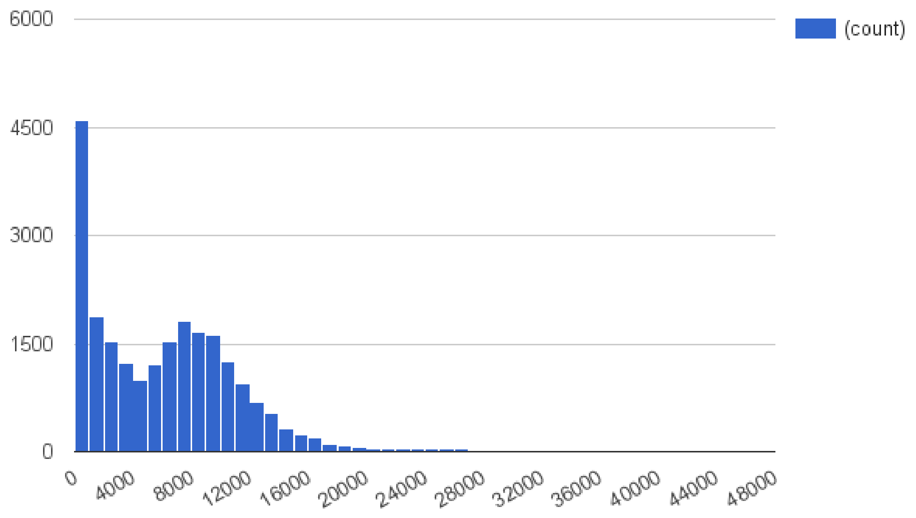

Figure 5.1: Sequence length distribution - MinION dataset.

5.3. Dot plot as a verification method

"In bioinformatics a dot plot is a graphical method that allows the comparison of two biological sequences and identify regions of close similarity between them. It is a type of recurrence plot." (dot).

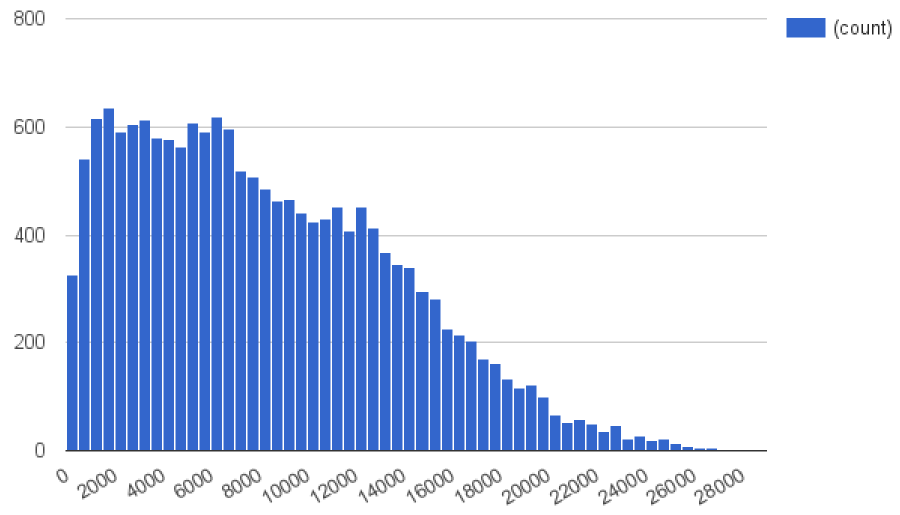

Figure 5.2: Sequence length distribution - PacBio dataset.

It organizes sequences (a and b) in two dimensional space, one sequence per axis; one on x-axis, one on y-axis. When two parts of sequences match, a point is drawn on that position. The simplest example would be drawing a picture of similarity matrix A, where

$$A[x, y] = \begin{cases} 1, & \text{if } a[x] = b[y] \\ 0, & \text{otherwise} \end{cases} \tag{5.1}$$

Position x, y would be black if $A[x, y] = 1$, otherwise it would stay white.

It is very obvious that identical sequences would form a diagonal line starting from $(0, 0)$ and ending in (x, x) (example shown on Fig. 5.3). Drawing dot plot of random sequences looks just like a noise on white (shown on Fig. 5.4). Dot plot of sequences $AAAA$ and $AAAA$ would be black $4x4$ image (because every position matches every position).

Drawing dot plots for comparisons of real biological sequences gets more difficult. Calculating exact matrix A is taking $O(NxM)$ time, which is usually too slow. Also, it is very convenient to match sequences a and b, but also a and reversed complement of b.

Gepard tool has lower-complexity matching implemented and is drawing both comparisons on the same graph (Krumsiek, 2007).

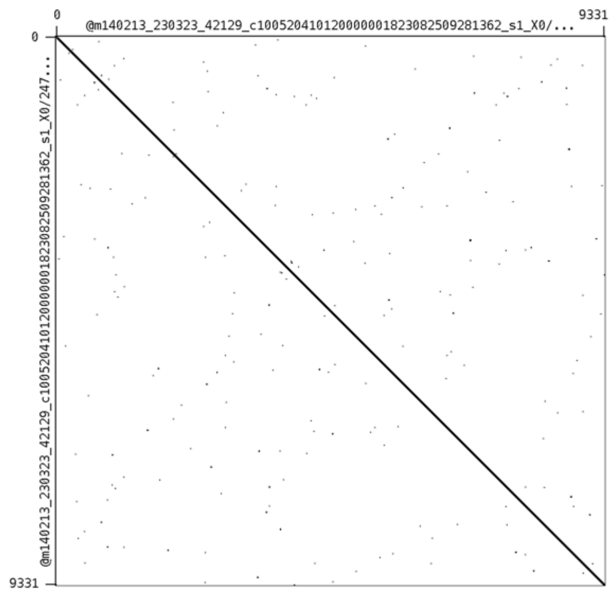

Figure 5.3: Dot plot of identical sequences.

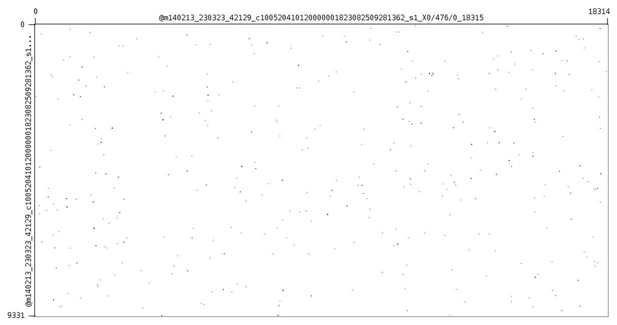

Figure 5.4: Dot plot of randomly picked sequences.

In this thesis we used dot plot as verification method. If dot plot looked good, we considered assembly process successful. Bad side of using dot plot as verification method is need for manual action.

5.4. MinION E. coli dataset - default parameters

Overall assembly was finished in 7 minutes and 46 seconds. From the table 5.1 it is visible that biggest part of time is spent on calculating overlaps with *owler* (6 minutes and 7 seconds).

The whole pipeline used 12 threads, which is the number that equals to number of physical cores on the computer we have used for testing.

We are bringing some numbers to get a feeling of which stages reduced dataset size by what portion:

Table 5.1: Dissection of assembler run-time on MinION dataset.

Phase	Duration
GraphMap owler	6 minutes, 7 seconds
Filling depot with reads	0 minutes, 3 seconds
Filling depot with overlaps	0 minutes, 6 seconds
Calculating read coverage	0 minutes, 4 seconds
Filtering contained reads	0 minutes, 4 seconds
Creating dovetail overlaps	0 minutes, 30 seconds
Filtering erroneous reads	0 minutes, 4 seconds
Filtering transitive overlaps	0 minutes, 4 seconds
Unitigger	0 minutes, 36 seconds
Drawing graphs	0 minutes, 36 seconds

– layout process started with 470954 overlaps.

– 98.56% overlaps were filtered due to being coincident with contained reads

– 111 overlaps (1.64%) were filtered because they did not look like reads that could be transformed to dovetail overlaps (at least one detected end was more than 10% of overlap length far from read ends)

– 20 overlaps were filtered because they were detected as overlaps coincident with contained reads; after being converted to dovetail form

– 827 overlaps (12.44%) were filtered because they covered less than 15% of coincident read length

– 55 overlaps were filtered because their error rate was higher than 40%

– 71.04% (of remaining 5764) overlaps were detected as transitive overlaps

– 79 bubbles detected and removed

– 3 tips removed

Since no consensus step is implemented and no error correction on reads is done, there is no much sense in high-accuracy sequence difference calculation (error will stay the same as reads error rate).

From the Fig. 5.5 it is clear that reference and assembled genomes are matching pretty well. Assembled genome is covering whole reference genome. Visual check does not point to any misassemblies and genome lengths are of insignificant difference ($< 1\%$).

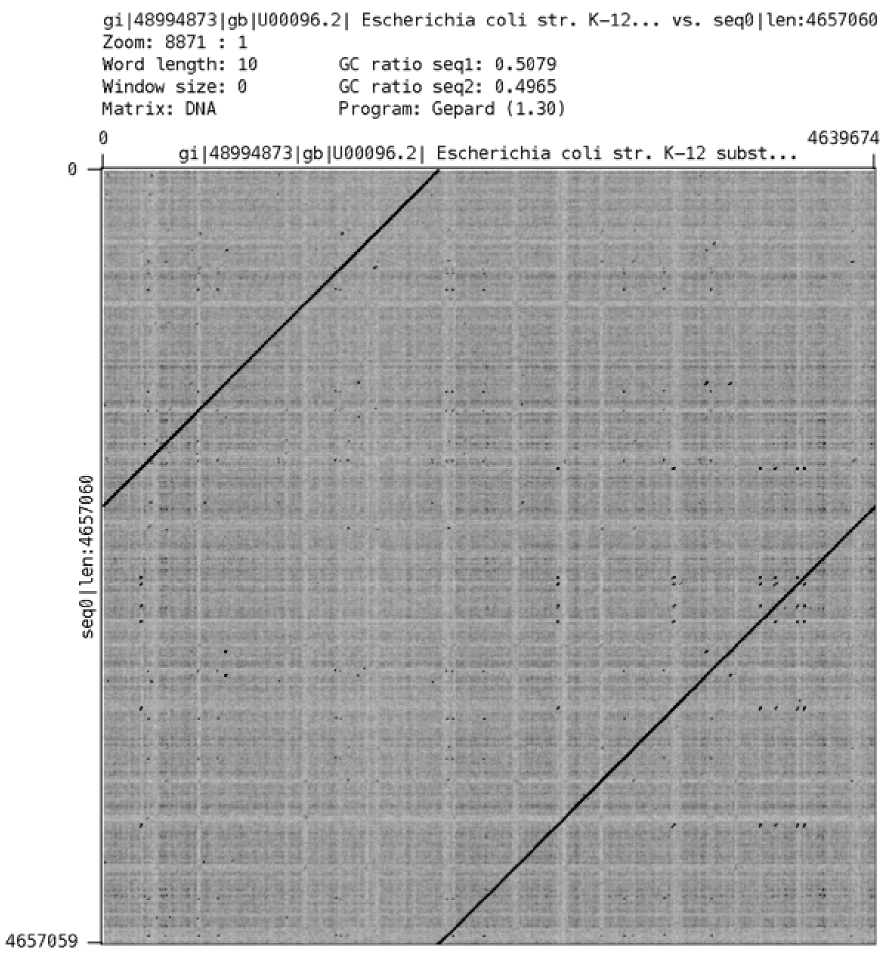

Figure 5.5: Alignment between E.coli reference genome ($x - axis$) and assembly result ($y - axis$).

Archive containing all the data (including depot and run logs) can be downloaded on `http://mariokostelac.com/thesis/ox.html`.

5.5. Pacbio E. coli dataset - default parameters

Overall assembly lasted for 10 minutes and 51 seconds. Distribution of times spent in each phase is very similar to the ones in the first dataset and it can be extracted from logs.

As run on previous dataset, maximum 12 threads were used for parts that are using leverage of multi-core systems.

We are bringing some numbers to get a feeling of which stages reduced dataset size by what portion:

- layout process started with 382668 overlaps

- 97.82% overlaps were filtered due to being coincident with contained reads
- 154 overlaps (1.85%) were filtered because they did not look like reads that could be transformed to dovetail overlaps (at least one detected end was more than 10% of overlap length far from read ends)
- 8 overlaps were filtered because they were detected as overlaps coincident with contained reads; after being converted to dovetail form
- 1389 overlaps (16.98%) were filtered because they covered less than 15% of coincident read length
- 66 overlaps were filtered because their error rate was higher than 40%
- 69.49% (of remaining 6723) overlaps were detected as transitive overlaps
- 82 bubbles detected and removed
- 246 tips removed

At the end, 23 unitigs were found, all part of one connected graph component. Extracted contig is about right length (4,349,490bp), but Fig. 5.6 shows that it is broken on several places.

Archive containing all the data (including depot and run logs) can be downloaded on `http://mariokostelac.com/thesis/pb_default.html`.

5.6. Pacbio E. coli dataset - modified quality_threshold parameter

As seen in previous subchapter, assembly results for PacBio dataset were not satisfying so we tried to modify some parameters to get better results. This time we reused calculated overlaps (from last run) because we have not changed any parameters that could affect overlapping phase.

The only parameter we changed is $quality\_threshold$. It affects the contig extraction algorithm by deciding whether some overlap (if coming from fork) should be considered as next one - based on its quality (as defined in Chapter 4). Default value is 1.00 and it implies that the only factor voting for next overlap in fork is actually maximum length reached from the fork.

After setting that parameter to 0.6, we restricted the choice of overlaps to the ones whose quality is at most 60% less than the best overlap in certain fork.

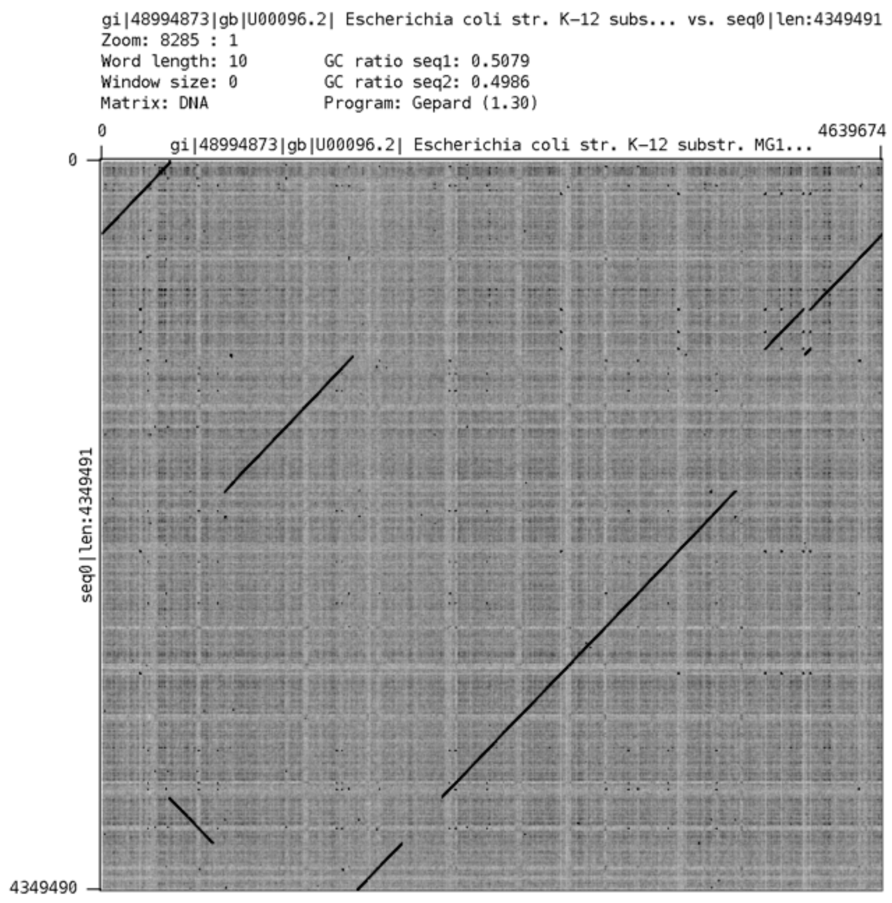

Figure 5.6: Alignment between E.coli reference genome ($x - axis$) and assembly result ($y - axis$); PacBio dataset, $quality\_threshold = 1.0$.

Fig. 5.7 shows that extracted contig is quite better, which implies that assembler did not lose too much information about the contig; it just was not able to extract the contig because of the noise in remaining data. It is not perfect (has one big break), but it is far better than result in previous chapter.

Archive containing all the data (including depot and run logs) can be downloaded on `http://mariokostelac.com/thesis/pb_quality.html`.

5.7. Comparison with $minasm$

In November of 2015 Heng Li started the $miniasm$ project (`https://git.io/vgcWL`). The idea behind the project is very similar to the work presented in this thesis - assembling genomes from long error-prone reads without correcting them upfront.

The implementation is quite fast and results seem promising. We have runned $minasm$ with default parameters from readme file.

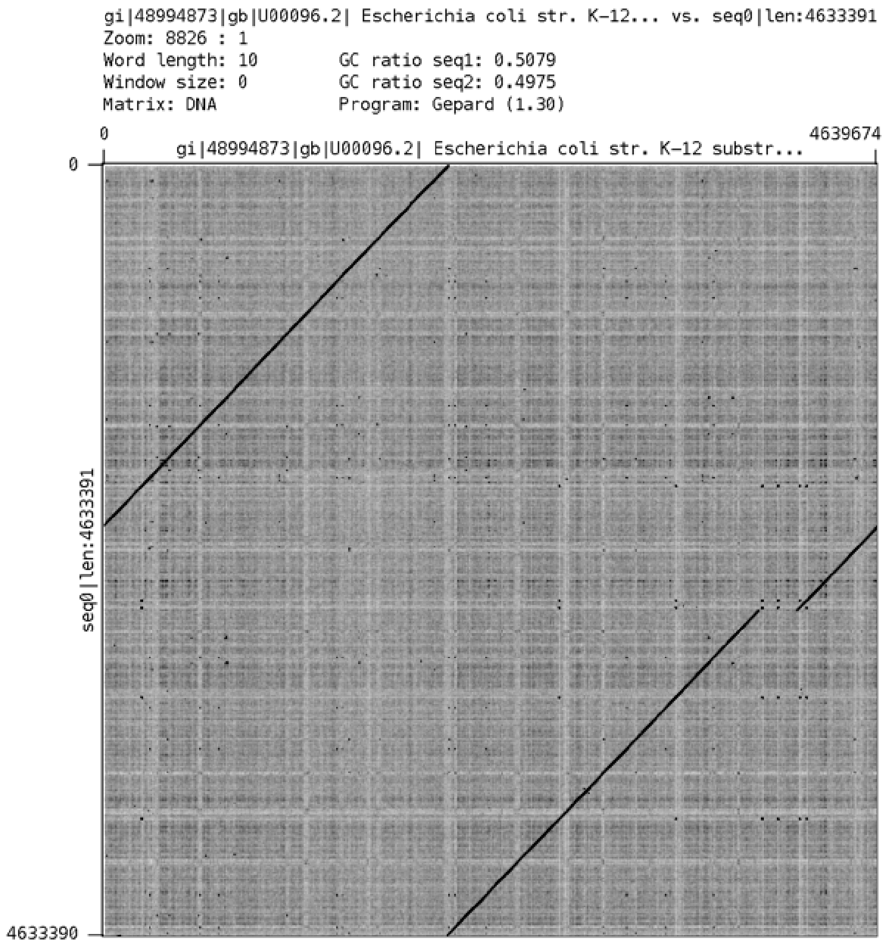

Figure 5.7: Alignment between E.coli reference genome ($x - axis$) and assembly result ($y - axis$); PacBio dataset, $quality\_threshold = 0.6$.

Running $miniasm$ on ONT dataset lasted for 17 seconds. Alignment with reference genome is shown on Figure 5.8.

Running $miniasm$ on PacBio dataset lasted for 15 seconds. Alignment with reference genome is shown on Figure 5.9.

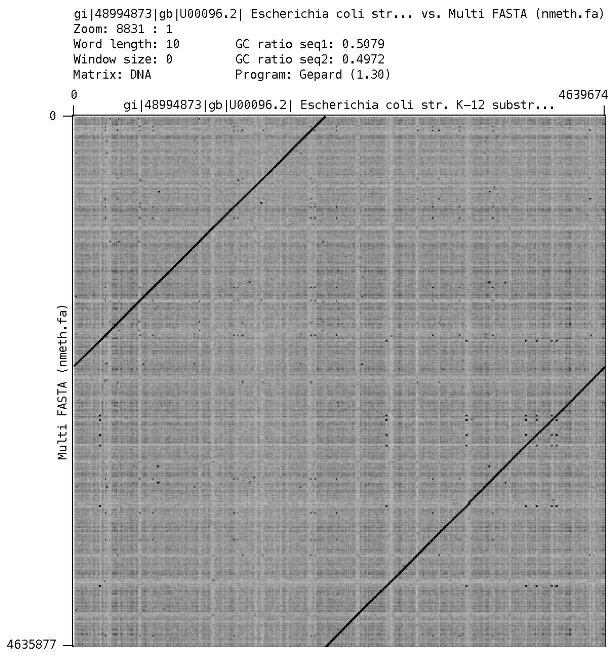

Figure 5.8: Alignment between E.coli reference genome ($x - axis$) and $miniasm$ assembly result ($y - axis$); ONT dataset

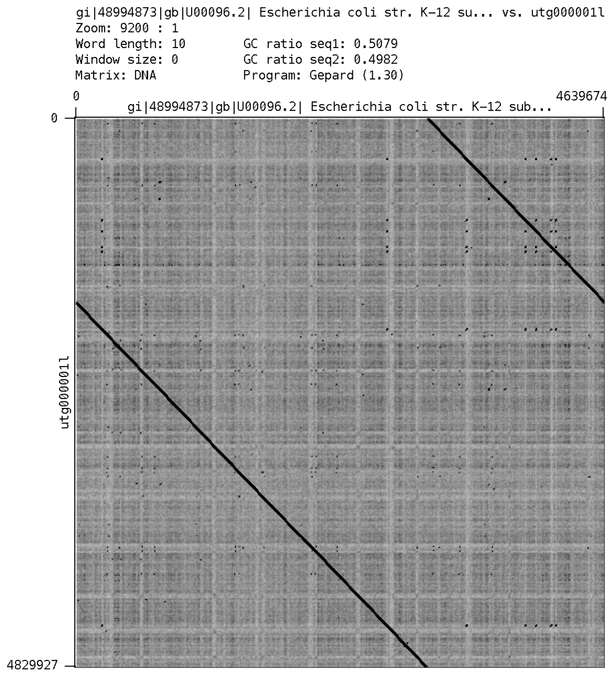

Figure 5.9: Alignment between E.coli reference genome ($x - axis$) and $miniasm$ assembly result ($y - axis$); PacBio dataset

6. Conclusion

At the time when this thesis work has started, there were no published papers or drafts on assembling genomes from long, error-prone reads without error-correction upfront. Error-correction is a very expensive process and, while it is feasible solution for smaller genomes like E.coli, it does not scale well for species with longer DNA strands. This thesis proves that it is possible to assemble genomes exclusively from long, error-prone reads, without any read correction upfront.

Performance-wise, the assembler developed for this thesis can be a lot faster than it is. Most parts are not multi-threaded, big portion of time is spent on IO operation (because of pipeline architecture) and project itself is carrying technical debt of four projects before.

Correctness of assembly can be improved, too. We have shown that even a simple technique can help with filtering noise from dataset, but second dataset (PacBio E.coli) showed that few bad overlaps can lead to extraction of completely faulty contigs.

The thesis has also shown that same base algorithms can be used for assembling genomes from long reads, error-prone reads.

Several areas could lead to qualitative improvements on final results. First area is filtering erroneous reads, based on coverage or overall coincident overlaps error rate. That method would clear sources of erroneous overlaps in very early phase. Second area for improvement is extraction logic. Current logic is greedy and favors length over quality. Even with $quality\_threshold$ factor included, it has its flaws. Better approach would be extracting unitigs and using them as starting point for new graph - unitig graph. Contigs could be just walks in that new graph.

The $miniasm$ project shows that the whole process can be much faster. Unfortunately, we did not have enough time to compare algorithm differences, but this thesis and $miniasm$ indicate that assembling genomes from long error-prone reads is possible.

BIBLIOGRAPHY

Dot plot. https://en.wikipedia.org/wiki/Dot_plot_(bioinformatics). Accessed: 2016-01-25.

Edit distance. https://en.wikipedia.org/wiki/Edit_distance. Accessed: 2016-01-24.

Human genome project. https://en.wikipedia.org/wiki/Human_Genome_Project. Accessed: 2016-01-24.

Hybrid genome assembly. https://en.wikipedia.org/wiki/Hybrid_genome_assembly. Accessed: 2016-01-24.

Mhap output specification. http://mhap.readthedocs.org/en/stable/quickstart.html#output. Accessed: 2016-01-24.

Overlaps specifications. http://wgs-assembler.sourceforge.net/wiki/index.php/Overlaps. Accessed: 2016-01-24.

Graphmap owler. https://github.com/isovic/graphmap/blob/master/overlap.md. Accessed: 2016-01-24.

Dna: nanopore sequencing. https://nanoporetech.com/applications/dna-nanopore-sequencing. Accessed: 2016-01-24.

RunCA: Dissection. http://wgs-assembler.sourceforge.net/wiki/index.php/RunCA_Dissection. Accessed: 2016-01-24.

Solid-state nanopores. https://nanoporetech.com/science-technology/introduction-to-nanopore-sensing/solid-state-nanopores. Accessed: 2016-01-24.

Richard Cammack, Teresa Atwood, Peter Campbell, Howard Parish, Anthony Smith, Frank Vella, i John Stirling. Needleman–wunsch alignment algorithm. URL //www.oxfordreference.com/10.1093/acref/9780198529170.001.0001/acref-9780198529170-e-13384.

W. E. Castle. Mendel's law of heredity. *Science*, 18(456):396–406, September 1903. ISSN 1095-9203. URL http://science.sciencemag.org/content/18/456/396.

Rattei Krumsiek, Arnold. Gepard: a rapid and sensitive tool for creating dotplots on genome scale. *Bioinformatics*, (23):1026–1028, 2007. URL http://bioinformatics.oxfordjournals.org/content/23/8/1026.full#ref-list-1.

Nicholas James Loman, Joshua Quick, i Jared T Simpson. A complete bacterial genome assembled de novo using only nanopore sequencing data. *bioRxiv*, 2015. doi: 10.1101/015552. URL http://biorxiv.org/content/early/2015/02/20/015552.

Mohammed-Amin Madoui, Stefan Engelen, Corinne Cruaud, Caroline Belser, Laurie Bertrand, Adriana Alberti, Arnaud Lemainque, Patrick Wincker, i Jean-Marc Aury. Genome assembly using nanopore-guided long and error-free dna reads. *BMC Genomics*, 16(1):1–11, 2015. ISSN 1471-2164. doi: 10.1186/s12864-015-1519-z. URL http://dx.doi.org/10.1186/s12864-015-1519-z.

Jason R. Miller, Sergey Koren, i Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, Jun 2010. ISSN 0888-7543. doi: 10.1016/j.ygeno.2010.03.001. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2874646/. 20211242[pmid].

E. W. Myers. The fragment assembly string graph. *Bioinformatics*, (21):79–86, 2005. URL http://bioinformatics.oxfordjournals.org/content/21/suppl_2/ii79.abstract.

Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *J. ACM*, 46(3):395–415, Svibanj 1999. ISSN 0004-5411. doi: 10.1145/316542.316550. URL http://doi.acm.org/10.1145/316542.316550.

Bruno Rahle. Simplification of the overlap graph. Magistarski rad, Fakultet elektrotehnike i računarstva, Sveučilište u Zagrebu, 2014. URL `https://bib.irb.hr/prikazi-rad?rad=773757`.

J. T. Simpson i R. Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 12(26):367–373, 2010. URL `http://bioinformatics.oxfordjournals.org/content/26/12/i367.full#cited-by`.

S. L. Salzberg T. J. Treangen. Repetitive dna and next-generation sequencing: computational challenges and solutions. *Nature Reviews Genetics*, (13/1): 36–46, November 2011. URL `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3324860/`.

Robert Vaser. De novo transcriptome assembly. Magistarski rad, Fakultet elektrotehnike i računarstva, Sveučilište u Zagrebu, 2015. URL `https://bib.irb.hr/prikazi-rad?rad=773702`.

De novo sastavljanje genoma koristeći dugačka očitanja s velikom pogreškom

Sažetak

Sastavljanje genoma koristeći dugačka očitanja s velikom pogreškom (bez ispravljanja očitanja) je obećavajuća metoda. Diplomski rad pokazuje da se, ako se koriste zajedno s jednostavnim metodama za uklanjanje šuma iz podataka, metode korištene za sastavljanje genoma koristeći kratka očitanja mogu koristiti i za sastavljanje genoma koristeći dugačka očitanja. Implementirane metode su testirane na dva skupa podataka E. coli (jedan tvrtke PacBio i jedan tvrtke Oxford Nanopore Technologies). Rezultati su uspoređeni s rezultatima *miniasm* assemblera.

Ključne riječi: De novo sastavljanje genoma, PacBio, Oxford Nanopore technologies, faza razmještaja.

De novo assembly using long error-prone reads

Abstract

Assembling genomes from long error-prone reads without error correction seems like a promising method. The thesis shows that layout methods used for assembling genomes from short reads can be utilized for assembling from long error-prone reads, if coupled with new methods for eliminating the noise from a dataset. Methods are validated with two datasets of Escherichia coli (PacBio and Oxford Nanopore Technologies). Results are compared with results of the *miniasm* assembler.

Keywords: De novo assembly, PacBio, Oxford Nanopore Technologies, layout.