UNIVERSITY OF ZAGREB
**FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING**

GRADUATION THESIS num. 745

# Simplification of the Overlap Graph

Bruno Rahle

Zagreb, June 2014.

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

Zagreb, March 10th, 2014

# MASTER THESIS ASSIGNMENT No. 745

| | |
|---|---|
| Student: | **Bruno Rahle (0036448928)** |
| Study: | Computing |
| Profile: | Computer Science |

Title: **Simplification of the overlap graph**

Description:

The goal of this thesis is to simplify the overlap graph down to the non-redundant set of overlaps comprising the unitigs (a.k.a. string graph). Furthermore, perform error corrections by using trimming and "bubble popping" methods. Compare performances (running time, memory usage and accuracy) with the overlap phases of SGA and Minimus assemblers. This thesis should also explore the consequences of high error rates, non-uniform read lengths, and low coverage found with real data. The code should be extensively commented in order to make its debugging and maintenance easier. Provide detailed instructions for installation and usage. The code should be written in C/C++ for Linux.

Thesis submitting date: June 30th, 2014

Mentor:

[illegible]

Prof. Mile Šikić, PhD

Committee Chair:

[illegible]

Prof. Siniša Srbljić, PhD

Committee Secretary:

[illegible]

Prof. Tomislav Hrkać, PhD

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

Zagreb, 10. ožujka 2014.

# DIPLOMSKI ZADATAK br. 745

| | |
|---|---|
| Pristupnik: | **Bruno Rahle (0036448928)** |
| Studij: | Računarstvo |
| Profil: | Računarska znanost |

Zadatak: **Pojednostavljenje grafa preklapanja**

Opis zadatka:

Cilj ovoga rada je pojednostavljenje grafa preklapanja do razine neredundantnoga skupa preklapanja (graf znakovnoga niza). Nadalje, potrebno je provesti mjere ispravljanja pogreški metodama uklanjanja sporednih grana i "mjehura" u grafu. Usporediti performanse (vrijeme izvođenja, korištenje memorije i točnost) s fazama preklapanja SGA i Minimus alata za sastavljanje genoma. Dodatno je potrebno istražiti utjecaj visokih razina pogreške, neujednačenosti duljine očitanja i malog pokrivanja kod stvarnih podataka. Sav kod treba biti iscrpno komentiran u cilju olakšanja ispravljanja pogrešaka i održavanja. Upute za instalaciju i izvođenje moraju se nalaziti zajedno s kodom.

| | |
|---|---|
| Zadatak uručen pristupniku: | 14. ožujka 2014. |
| Rok za predaju rada: | 30. lipnja 2014. |

Mentor:

Mile Šikić

Doc. dr.sc. Mile Šikić

Predsjednik odbora za diplomski rad profila:

[illegible]

Prof. dr.sc. Siniša Srbljić

Djelovođa:

[illegible]

Doc. dr.sc. Tomislav Hrkać

*Firstly, I'd like to thank my parents and grandparents for support they have given me over the years.*

*I'd like to thank my mentor Mile Šikić for his help and for his patience.*

*In the end, honourable mentions go to Matija Osrečki and Anton Grbin for their help with the implementation.*

# CONTENTS

# LIST OF FIGURES

# 1. Introduction

Bioinformatics is a fairly new and exciting field that is advancing at a staggering rate. It is concentrated around creating algorithms and methods for efficient management of biological data. According to Hogeweg (2011), the term itself was coined in 1970 by the same author. Biological data that needs to be processed to obtain useful knowledge is very vast (only the human DNA has more than three billion nucleotides) so despite the advent of computer power and data processing algorithms we still have a long way to go until we can use it to improve our everyday life.

DNA holds the genetic information about all known living organisms, including humans. It is made from two sequences of nucleotides that are connected by the hydrogen bonds according to the base pairing rules. Most organisms (even amongst the same species) have different number and order of nucleotides in their DNA; in other words, (almost) every living organism has different DNA. Sequencing (finding the order of the nucleotides) of the human genome was the goal of the biggest bioinformatical project in history, The Human Genome Project. The project was started in 1984, but wasn't formal until 1990 when the US Department of Energy and the National Institutes of Health secured $3 billion in funding for the proposed 15 year project. Scientists from all around the world collaborated on the project which was declared complete in April 2003 and it was thought to have sequenced 99% of the human genome with 99.99% accuracy. According to Schmutz et al. (2004), 92% of the sampling is more than 99.99% accurate.

In the years that followed, the cost decreased and speed of sequencing increased rapidly. Today, just 11 years after the first genome was sequenced, the cost to sequence a human's DNA is around a thousand dollars. It is expected to fall to a few hundred dollars in the near future, and that is considered to be the point where it may become commonplace to have your DNA sequenced just as it is common to have your blood analysed. Various bioinformatical startups, like 23andMe, have been created with the idea of knowing a persons DNA in the core of their product.

There are two things that drive the cost of sequencing the DNA - the cost of the sequencers that generate the reads and the efficiency of the algorithms that combine those reads into the final DNA. In this thesis, we will explore certain algorithms that are used in DNA sequencing and how they can be implemented efficiently. Those algorithms revolve around building a string graph, which is a representation of the reads in a graph format where redundant information is simply ignored.

Chapter 2 will give a quick introduction to the biological, mathematical, and technological background of this thesis.

Chapter 3 describes how we deal with the enormous graph that is created by mutually overlapping all reads (and their reverse complements).

Chapter 4 will describe what a string graph is and how it is constructed.

Chapter 5 will describe how was the implementation developed, the technologies and frameworks that were used.

Chapter 6 will show the results of the C++ implementation that is provided besides this work.

Chapter 7 will offer final thoughts and discuss the importance of this thesis and potential future work.

# 2. Preliminaries

In this chapter basic knowledge required to understand the topic of this paper shall be covered. We will first give the biological introduction to the DNA and DNA sequencing problem, and after that the mathematical background shall be presented.

## 2.1. DNA

Deoxyribonucleic acid, otherwise known just as DNA, "is a molecule that encodes the genetic instructions used in the development and functioning of all known living organisms and many viruses." [Wikipedia (2014)].

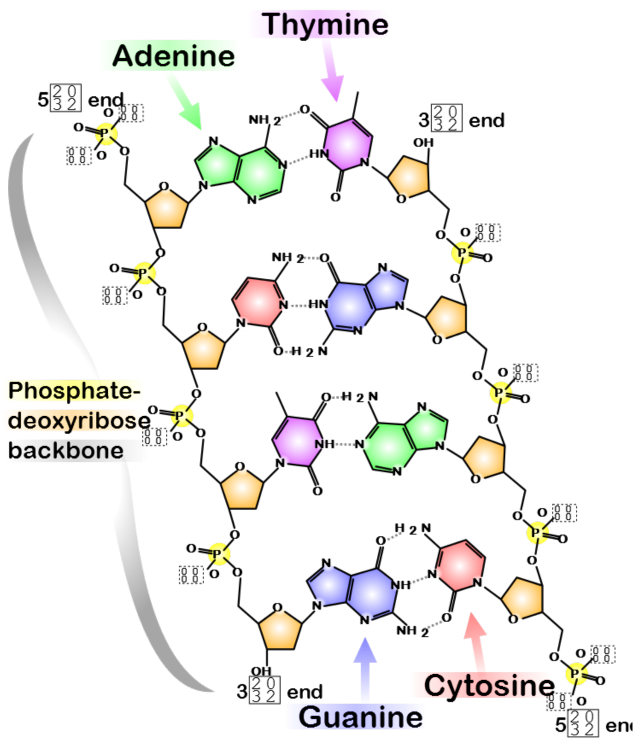
### 2.1.1. Chemical Structure

The DNA is considered to be a macromolecule, as it is made from smaller molecules. Two strands of biopolymers wrap around each other like a coil, forming a double helix. The smaller molecules, **nucleotides**, that go into making of each biopolymer consist of a nucleobase, deoxyribose and a phosphate group. The covalent bonds between the phosphate group from one nucletoide and deoxyribose from the next are responsible for forming DNA's backbone. The two nucleotides from separate strands are connected by the hydrogne bonds according to the base pairing rules (A-T and G-C). Because of the base pairing rules, knowing only one strand is enough to know the bases of the other polymer strand. DNA's chemical structure is depicted in Figure 2.1.

We are most interested in nucleobases and their order in the polymers. These are the nucleobases we can encounter:

- **Adenine**, which we will identify with capital letter A, and whose base pair is thymine
- **Cytosine**, which we will identify with capital letter C, and whose base pair is guanine

**Figure 2.1:** Chemical structure of the DNA [Ball (2013)]



- **Guanine**, which we will identify with capital letter G, and whose base pair is cytosine
- **Thymine**, which we will identify with capital letter T, and whose base pair is adenine

### 2.1.2. Biological Functions

The DNA is stored in every cell of the organism, usually in a number of chromosomes (1 to 630 pairs, according to Bowen (2014)). There are different types of chromosomes (linear, circular, etc.) and every chromosome stores different information in a different number of base pairs (ranging from 100,000 to 3,750,000,000, according to Paux et al. (2008) and Pellicer et al. (2010)).

The information DNA holds is grouped in genes, subsequences of the DNA that control how the features are inherited through generations. The whole gene is found completely on a single chromosome, but each chromosome holds many different genes and even more non-coding DNA seqeunces (for humans, 98% of the DNA is considered to be non-coding). The order of the nucleotide bases within a gene form a messenger RNA which will carry blueprints of one or more proteins to the ribosome.

### 2.1.3. DNA Sequencing

DNA sequencing is the process of determining the order of the nucleotide bases in the DNA of an organism. There is a multitude of methods used today, but the problem with all of them is that they can only sequence a limited number of bases. That sequence of bases that was identified by the sequencer is called a **read**. Not only is the length of a read fairly short (from 50 to 30,000) compared to the number of base pairs that make up the DNA, even the identified bases aren't guaranteed to be correct. What's more, usually the longer the read is, the higher the probability of a wrong base being identified. To sequence the whole of the DNA, we will obviously need more than one read. We will define **coverage** as the total number of reads multiplied by their average length and divided by the length of the reference genome, with that number usually being 10.

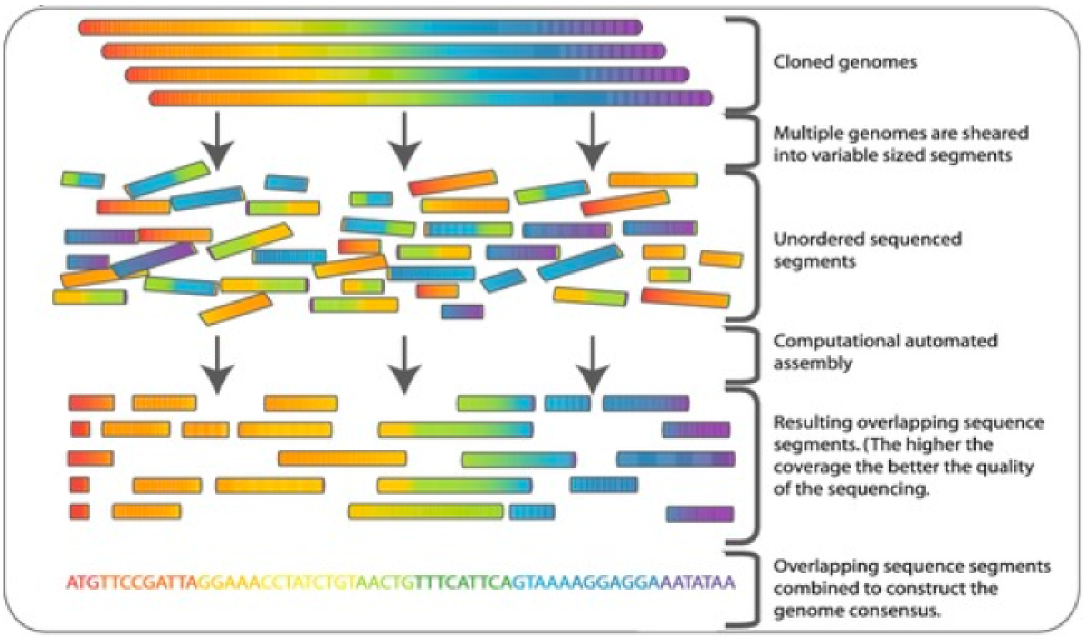
#### Shotgun sequencing

One of the approaches we can use when sequencing a large amount of reads is to sequence a random part of the DNA (instead of doing it in some kind of an order). That is the idea that lies behind shotgun sequencing - we do not care what part of the DNA are we reading, as long as we can more or less ensure that there is no bias in the sampling. Usually, shotgun sequencing methods work by multiplying the DNA, chopping it up into smaller pieces and then sequentially reading them. There are two approaches in chopping the DNA up into smaller pieces - hierarchical methods fist cut it into longer pieces which are then cut into smaller ones, while whole genome methods skip the first step and cut the macromolecule to small pieces right from the start. Most of the methods use both strands of the DNA to sequence it, but they do not know which strand are they sequencing at the time of the sequencing. Figure 2.2 shows the basic idea behind this kind of sequencing.

The single molecule real time sequencing is a whole genome shotgun sequencing method developed by Pacific Biosciences that usually produces reads of 5,500 to 8,500 base pair average length with 87% of single read accuracy. Because of it's cost and speed, it is currently one of the best sequencing methods, and, as such, test data for this work has been developed with it in mind.

To sum up, not only are we dealing with a huge amount of data, whose accuracy we cannot vouch for, we also don't know if a read's data is complemented because it may come from the other strand, not to mention the position of the read within a strand! Thankfully, this is where computers step in. Once this phase is completed, various

**Figure 2.2:** Whole genome shotgun sequencing [Commins (2011)]


Cloned genomes
Multiple genomes are sheared into variable sized segments
Unordered sequenced segments
Computational automated assembly
Resulting overlapping sequence segments. (The higher the coverage the better the quality of the sequencing.
ATGTTCCGATTAGGAAACCTATCTGTAACTGTTTCATTCAGTAAAAGGAGGAAATATAA
Overlapping sequence segments combined to construct the genome consensus.

algorithms are employed to assemble the genome information. When assembling the genome from that data, our algorithms usually have three steps: the overlap phase, where we find which reads share common information, the layout phase, where we try to arrange the reads into contigs - groups of reads that are in sequence, and the consensus phase where we try to make the final decision on how the genome looks like. This work is mostly centred around the layout phase.

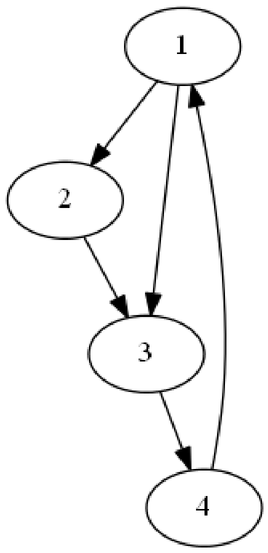
## 2.2. Graphs

In mathematics, there is a huge number of different definitions that are used for describing graphs. The following definitions shall be used in this thesis.

A **graph** $G$ is defined as a set of objects, called **vertices** and denoted as $V$, and connections between those objects, called **edges** and denoted as $E$. Every edge describes a connection between two vertices, let's call the two vertices $A$ and $B$. The edge $E_{AB}$ means that there is a connection from vertex $A$ to vertex $B$. We will only be dealing with **directed graphs**, in which there does not necessarily need to be an edge $E_{BA}$ if there is an edge $E_{BA}$. You can see an example of the graph on Figure 2.3

A **path** is a sequence of edges which connect a sequence of vertices. An example of a path from the Figure 2.3 would be "{1 -> 2 -> 3}". If there isn't an edge connecting two vertices, then they cannot be subsequential in a path, so "{1 -> 4}" wouldn't be

**Figure 2.3:** A sample graph



considered a valid path.

# 3. Layout

## 3.1. Overview

Layout is a phase of *De novo* sequencing where we take read and overlap data from the overlap step and remove redundant information. It turns out that the amount of data that is redundant is actually pretty big, and we can decrease the number of objects we are using by (several) orders of magnitude, as will be discussed in Chapter 6.

We will divide the layout phase into three parts:

1. **Removing contained reads** where we remove reads and all their associated overlaps that can be found in other reads.
2. **Removing transitive edges** where we remove overlaps that can be reconstructed from remaining edges. That is, if we have overlaps (edges) between (A,B), (A,C) and (B,C), we remove (A,C).
3. **Collapsing unique joins** where we connect reads to form contigs by repeatedly connecting reads and contigs who can only be with connected with one other read or contig.

The terms edge and overlap refer to the same thing in this chapter, so their usage is interchangeable.

### 3.1.1. Reverse Complement

The way today's DNA sequencers work, as described in Chapter 2, has an interesting consequence - two strands of the same DNA are read from different ends! For example, take a look at Figure 3.1. Read $R =$ "ACGTAC" can also be read as $R' =$ "GTACGT" and we will consider read $R'$ to be a reverse complement of read $R$ (and vice versa). More formaly, a reverse complement of a read $R$ is a read $R'$ in which the order of the nucleotides is reversed, and nucleotides are replaced with their base pairs.

**Figure 3.1:** Two DNA strands and the directions the sequencing machine sequences them

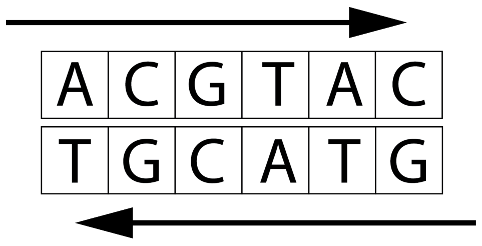
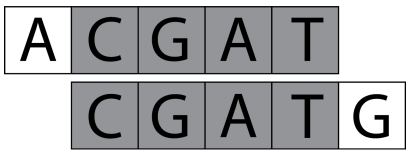


**Figure 3.2:** Regular dovetail overlap



Because the strands are identical (if we ignore the fact that the nucleotides are base pairs), we could end up with two separate DNA reconstructions - one for the "normal" direction, and the other for the reversed. Doing so would only decrease our accuracy, as both would be the same length but each would have lower coverage. Therefore, it would be opportune to only reconstruct one strand. The problem lies in the fact that the sequencing machine doesn't provide us with information about the strand the read comes from (so we don't know the orientation of the read either). First step in solving this problem came while we were overlapping the reads, and it was to consider each read to actually represent two reads - itself and it's reverse complement.

### 3.1.2. Types of Edges

The result that the previous phase of DNA sequencing gives us is a set of reads and overlaps between them. We will consider reads to be vertices and overlaps edges in a graph. Depending on the relative positions of the overlapping reads in the DNA, there can be different types of overlaps between them. One read may be entirely contained inside another read, which will form a **containment edge**. In any other case, the overlaps happen on the "edges" of the reads. Those form **dovetail edges**, and there are three different kinds of those edges.

The first, most obvious one, is called a **regular dovetail** edge. It is created when two reads from the same strand overlap, like in the Figure 3.2. We can decide that the one whose end is being covered comes before the one whose beginning is being covered. We shall also refer to this type of edges as "EB" edge.

**Figure 3.3:** Prefix dovetail overlap

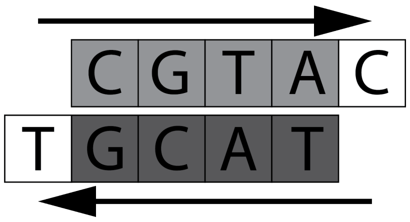
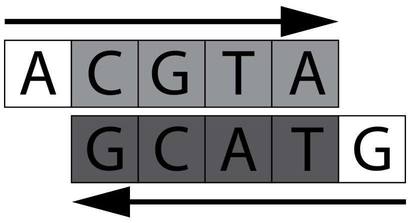


**Figure 3.4:** Suffix dovetail overlap



Other two types are less obvious - they occur because the sequencing machine is giving us data from both strands. We can have two situations here. Consider two reads $A$ = "CGTAC" and $B$ = "TACGT", like in figure Figure 3.3. Those reads are considered to form a **prefix dovetail** edge. If we take a reverse complement of one of the edges, say $B'$ = "ACGTA", we see that it's end is the same as the start of the

Similarly, if the beginning of the reverse complemented read is the same as the end of the other read, we get a **suffix dovetail** edge. That situation is illustrated in Figure 3.4, where $A$ = "ACGTA" and $B$ = "GTACG".

Table 3.1 shows us how we represent different edge type, with notation defined by Myers (1995).

## 3.2. Removing Contained Reads

This is the first step in reducing redundant information. If a read is contained inside a larger one, it is obvious that we can simply not use the read (and all it's overlaps)

**Table 3.1:** Taxonomy of overlap types

| **Edge name** | **Edge** |
|---|---|
| Containment | $A$ >==> $B$ |
| Regular Dovetail | $A$ >—> $B$ |
| Prefix Dovetail | $A$ <—> $B$ |
| Suffix Dovetail | $A$ >—< $B$ |

without losing much information.

### 3.2.1. Input

We are given a list reads and a list of overlaps between two reads.

### 3.2.2. Output

We need to return a list of overlaps without the reads which are contained in another read. In other words, only the overlaps between non-contained reads must be in the list.

### 3.2.3. Algorithm

The idea of the algorithm is very simple: we go through the list of all edges twice. First time we find all the reads that are contained within another read and mark them for deletion. The second time we are adding the overlap to the list we will return if none of two the reads were marked for deletion. Let $overlap.A$ be the first read of the overlap, and $overlap.B$ be the second read. Pseudocode for the function can be seen in Algorithm 1.

**Algorithm 1** Removing containment edges

```
function RemoveContainment($reads$, $overlaps$)
    $contained \leftarrow \{\}$
    for all $overlap \in overlaps$ do
        if $overlap.type =$ Containment then
            $contained \leftarrow contained \cup overlap.B$
        end if
    end for
    $noverlaps \leftarrow \{\}$
    for all $overlap \in overlaps$ do
        if not $(overlap.A \in contained$ and $overlap.B \in contained)$ then
            $noverlaps \leftarrow noverlaps \cup overlap$
        end if
    end for
    return $noverlaps$
end function
```

**Figure 3.5:** Before removing contained reads

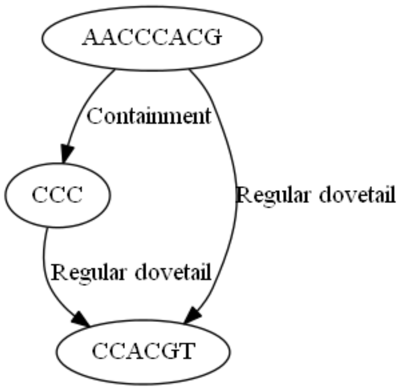
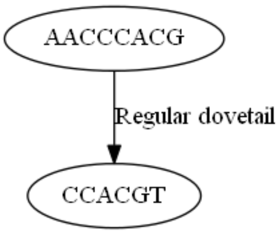


**Figure 3.6:** After removing contained reads



### 3.2.4. Example

Say we are given reads $A =$ "AACCCACG", $B =$ "CCC", and $C =$ "CCACGT" and overlaps $A$ >==> $B$, $B$ >—> $C$, $A$ >—> $C$. In this case, it is obvious that read $B$ is contained inside read $A$. So we will remove the read $B$ and both of it's overlaps. Figures 3.5 and 3.6 depict the situation before and after.

### 3.2.5. Analysis

Using a lookup table, we can implement marking a read for deletion and checking if it is marked in $O(1)$ time. The memory requirements for that, however, are equal to $O(N)$ (linear in the number of reads). We also need additional $O(M)$ to keep the new list of overlaps.

In total, we need $O(N + M)$ and $O(M)$ time for this algorithm.

## 3.3. Merge Sort's Merging Algorithm

One of the main pillars that allowed us to remove the transitive edges in both time-and space-efficient manner was the idea that lies behind this algorithm.

We will now describe a linear time algorithm that solves the problem of merging two sorted sequences into a sorted sequence that holds all objects.

### 3.3.1. Input

We are given two lists ($A$ and $B$) of sorted objects. Without loosing generality, we can assume that the sort is ascending.

### 3.3.2. Output

One sorted list ($C$) of objects that contains all elements from both lists of objects.

### 3.3.3. Algorithm

We start with two iterators pointing to the beginning of the lists. We then compare the elements that the iterators are pointing to, place the smaller one in the resulting list $C$, and increase the value of that iterator (the one pointing to the smaller object). If we've reached the end of the list, we then sequentially place the remaining items from other list at the end of the list $C$. Otherwise, we repeat the process.

The resulting list $C$ will, therefore, be constructed in such a way that it is always sorted. Because of that, we won't need to sort the list after the algorithm is over and we keep time complexity linear. The algorithm's pseudo code is given in Algorithm 2.

### 3.3.4. Example

Let's say we are given two lists of integers, $A = \{2, 3\}$ and $B = \{1, 4\}$, and we need to merge them. The list $C = \{\}$.

We compare the two elements $A_0 = 2$ and $B_0 = 1$ and we see that $B_0 < A_1$. We pick the one from $B$ so we get $C = \{1\}$ and $j = 1$, while $i$ stays $0$.

We compare the two elements $A_0 = 2$ and $B_1 = 4$ and we see that $A_0 < B_1$. We pick the one from $A$ so we get $C = \{1, 2\}$ and $i = 1$, while $j$ stays $1$.

We compare the two elements $A_1 = 3$ and $B_1 = 4$ and we see that $A_1 < B_1$. We pick the one from $A$ so we get $C = \{1, 2, 3\}$ and $i = 2$, while $j$ stays $1$.

We are now out of elements in $A$ so we have to pick one from $B$. We get $C = \{1, 2, 3, 4\}$ and $j = 2$, while $i$ stays $2$.

**Algorithm 2** Merge Sort's Merging Algorithm

```
function MERGE($A$, $B$)
    $i \leftarrow 0$
    $j \leftarrow 0$
    $k \leftarrow 0$
    $C \leftarrow$ array of size $|A| + |B|$
    while $i < |A|$ or $j < |B|$ do
        if $i = |A|$ or $(j < |B|$ and $B_j < A_i)$ then
            $C_k \leftarrow B_j$
            $j \leftarrow j + 1$
        else
            $C_k \leftarrow A_i$
            $i \leftarrow i + 1$
        end if
        $k \leftarrow k + 1$
    end while
    return $C$
end function
```

### 3.3.5. Analysis

The time complexity of this algorithm is $O(|A| + |B|)$, as our loop does that many iterations. The space complexity is $O(|A| + |B|)$, but in reality we use double the memory.

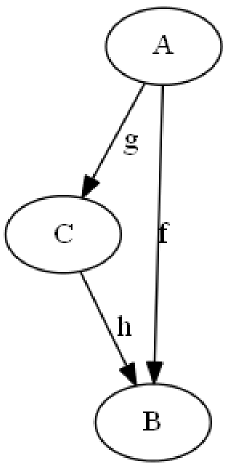
### 3.3.6. Additional Notes

There is a version that uses constant additional memory, but as we only need to iterate over the list once (and not keep it in memory!), it won't be explained here. You can look into Sedgewick (1998) for help.

## 3.4. Removing Transitive Edges

In this step we need to remove all transitive edges. In general, an edge between $A$ and $B$ is considered to be transitive if and only if there exists an edge between $A$ and $C$ and one between $C$ and $B$. That situation is depicted in Figure 3.7.

**Figure 3.7:** Transitive edge is the edge $f$, between $A$ and $B$



In our case, however, we will impose a few additional rules to consider an edge transitive. That is because edges in our graph aren't simple and have additional information besides their direction, like we described in Subsection 3.1.2.

Let's consider nodes $A$, $B$ and $C$, and edges $f$, $g$ and $h$ between them ($f$ is connecting $A$ and $C$, $g$ is connecting $A$ and $B$ and $h$ is connecting $B$ and $C$), like in Figure 3.7. For edge $f$ to be considered transitive, ends of the reads that connect them must be consistent. Let us use $f.suffix_A$ to denote which end of the read $A$ overlap $f$ is using (the only two values could be either "begin" or "end"). Figure 3.8 shows what would could be suffixes for the graph from Figure 3.7. For the edge to be transitive, the following must hold:

$$f.suffix_A = g.suffix_A$$

$$f.suffix_B = h.suffix_B$$

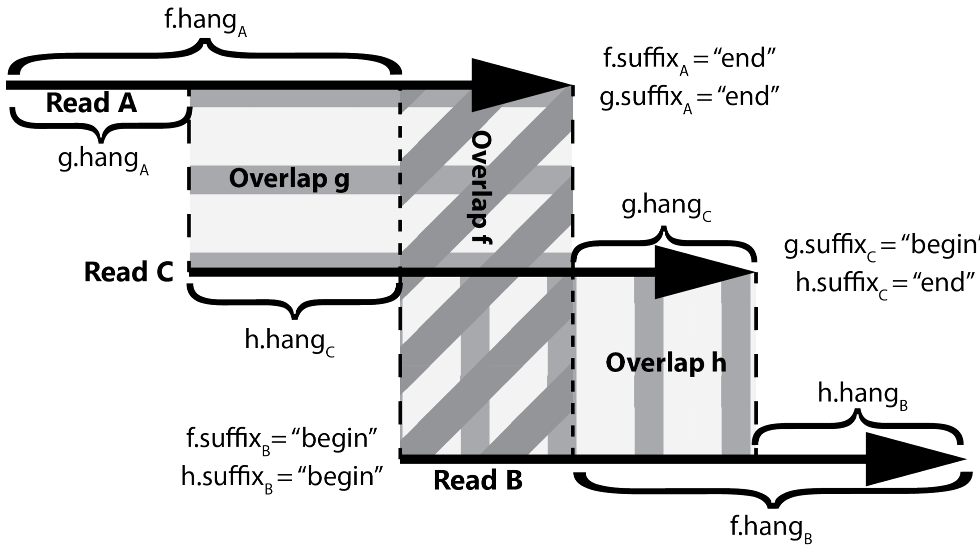
$$g.suffix_C \neq h.suffix_C$$

In other words, the end of read $A$ that edges $f$ and $g$ are using must be the same; same holds true for read $B$ and edges $f$ and $h$; however, edges $g$ and $h$ must use the opposite ends of the read $C$.

It also must hold that the overlaps refer to the same data. Since comparing the data they hold can prove to be costly, Myers (1995) proposed a position-based heuristic that works fine in practice. Let us first define $f.hang_A$ to be the length of the **remaining** (or hanging) part of read $A$ that isn't contained in overlap $f$, and $f.length$ be the length of read $f$. Figure 3.8 again shows what are the hanging parts of each edge for the graph from Figure 3.7. We will then have the following formulas:

$$g.hang_A + h.hang_C \in f.hang_A \pm (\epsilon \cdot f.length + \alpha)$$

**Figure 3.8:** Image showing what are certain parts of overlaps and reads are called



$$g.hang_C + h.hang_B \in f.hang_B \pm (\epsilon \cdot f.length + \alpha)$$

All of the rules are implemented in Algorithm 3. Function isTransitive, when given three overlaps, will check if the first one of them is transitive.

### 3.4.1. Input

We are given a list reads and a list of overlaps between them (without contained reads).

### 3.4.2. Output

We need to return a list of overlaps without the transitive overlaps.

### 3.4.3. Algorithm

We start by iterating through each overlap and test if the current edge is a transitive one. We do that by checking if there exists a third vertex between the two that that has overlaps with both of them. The easiest way to do that is to use a modified version of merge sort's merging algorithm on read's sorted adjacency lists to find the reads that are adjacent to both of our currently observed reads. Then we just call the previously defined `isTransitive` function to verify if those overlaps really make one of the edges to be a transitive one. If the edge is transitive, we mark it for deletion. In the end, we simply remove the marked edges from the list of all overlaps.

**Algorithm 3** Check if an edge is transitive

```
function isEqual($x$, $y$, $\epsilon$)
    return $y \leq x + \epsilon$ and $x \leq y + \epsilon$
end function
function isTransitive($o1$, $o2$, $o3$)                ▷ $o1$, $o2$, $o3$ are three overlaps
    $A \leftarrow o1.A$
    $B \leftarrow o1.B$
    if $A = o2.B$ then
        $C \leftarrow o2.A$
    else
        $C \leftarrow o2.B$
    end if
    if $o2.suffix_C = o3.suffix_C$ then
        return false
    end if
    if $o1.suffix_A \neq o2.suffix_A$ or $o1.suffix_B \neq o3.suffix_B$ then
        return false
    end if
    if not isEqual($o2.hang_A + o3.hang_C$, $o1.hang_A$, $\epsilon * o1.length + \alpha$) then
        return false
    end if
    if not isEqual($o2.hang_C + o3.hang_B$, $o1.hang_B$, $\epsilon * o1.length + \alpha$) then
        return false
    end if
    return true
end function
```

**Algorithm 4** Removing transitive overlaps

```
1: function REMOVETRANSITIVE($reads$, $overlaps$)
2:     $noverlaps \leftarrow \{\}$
3:     $marked \leftarrow \{\}$
4:     for all $f \in overlaps$ do
5:         for all $g, h \in$ mutual overlaps of $f.A$ and $f.B$ do
6:             if ISTRANSITIVE($f$, $g$, $h$) then
7:                 $marked \leftarrow marked \cup f$
8:             end if
9:         end for
10:    end for
11:    for all $overlap \in overlaps$ do
12:        if not $overlap \in marked$ then
13:            $noverlaps \leftarrow noverlaps \cup overlap$
14:        end if
15:    end for
16:    return $noverlaps$
17: end function
```

### 3.4.4. Example

Consider the example from Figure 3.7. Here we should remove the edge $f$ and that will leave us with the edges $g$ and $h$, but all three reads stay and are not removed. We can see how the graph looks like after removing the transitive edges on Figure 3.9.

A larger-scale example of removing transitive edges can be found in Chapter 6.

### 3.4.5. Analysis

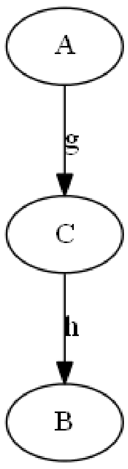
The time complexity of the sorting step is $O(NK \log K)$, where $N$ is the number of reads, and $K$ is the average number of overlaps each read has. Considering that $K$ is a fairly low number, this isn't much. We are also using additional $O(M)$ memory, where $M$ is the number of overlaps.

The second step has the time complexity of $O(MK)$. We are using additional $O(M)$ memory to store whether the edge is transitive or not.

The final step has the time complexity of $O(M)$, as we just iterate through a list.

In total, that would put the time complexity to $O(NK \log K + MK) = O(MK)$. The memory requirements are linear in the number of overlaps, $O(M)$.

**Figure 3.9:** Transitive edge removed from the graph from the Figure 3.7



## 3.5. Collapsing unique joins

In this step we create contigs, which are a sequence of reads that we know must come one after another. Since those reads may be mutually overlapping, there might still be uncertainties on how to assemble them. However, that is what the next step (consensus) is responsible for, so we won't discuss it here further. If you are interested, you can look into Sommer et al. (2007), Anson i Myers (1997) and Rausch et al. (2009) for further info.

We start by declaring all reads to be contigs of just one read. We shall define $A.prefix\_degree$ to be the number of edges that overlap the beginning of the contig $A$, and, similarly, $A.suffix\_degree$ to be the number of edges that overlap the ending of the contig $A$. We shall also define $A.degree_f$ to be the degree corresponding to the degree of the appropriate end of the contig $A$ in use by the edge $f$.

We then collapse all edges $f$ that have $f.A.degree_f = 1$ and $f.B.degree_f = 1$, by combining their respective contigs into one. In other words, we collapse all edges $f$ when they are the only edge overlapping both reads on whichever side of the read the overlap $f$ is using. We have to make sure we the relative order of the reads within a contig is preserved, as well as if we should actually be using their reversed complement in the reconstruction.

Myers (1995) reports that the number of contigs created in this way is fairly small (orders of magnitude smaller) when compared to the number of reads and this work confirms similar results, as is described in more detail in Chapter 6.

### 3.5.1. Input

We are given a list of reads and overlaps between them (without contained reads and without transitive edges).

### 3.5.2. Output

We need to return a list of contigs and edges between them.

### 3.5.3. Algorithm

We start by converting all reads into contigs. We then iteratie through all overlaps to find the overlaps that can be used to join two contigs. The problem is that the overlaps use reads and not contigs - so we need to find a way to quickly map a read to the contig that contains it. The data structure should also support the operation of joining two different contigs into one, as that is the operation we will perform after we find which contigs to use. It turns out that disjoint set data structure, also known as union find, can do just that in amortized constant time with just linear memory. Operation $disjoint\_set.find(x)$ finds the group (in our case, contig) object $x$ belongs to, and operation $disjoint\_set.union(x, y)$ unites groups $x$ and $y$ and returns which of the groups was larger before the join, as that is group that "survived" the operation. The detailed description of the data structure can be found in Sedgewick (1998). We also have to join the contig objects, which means we need to update the list of reads that the contig consists of. For complexities sake, we will always connect the shorter list onto the bigger one, like it is done in union find.

In the end we just need to remove all contigs whose size is 0 from the resulting set. The pseudocode of this method is outlined in Algorithm 5.

### 3.5.4. Example

Consider a situation depicted on Figure 3.10, where we see how a graph might look like right before we form contigs. The prefix degree of read (or 1-read contig) $A$ is $0$, reads $B$, $D$, $E$ and $F$ have prefix degree of $1$ and read $C$ has prefix degree $2$. Suffix degrees are $0$ for read $E$, $1$ for reads $A$, $B$, $D$ and $E$, and $C$ has suffix degree $2$.

As we test each edge, we see that only edges $A$ >–> $B$, $B$ >–> $C$ and $D$ >–> $F$ are can be collapsed and thus we form three contigs: $A$, $B$, and $C$ forming the first one, $D$ and $F$ forming the second, and $E$ being the sole read in the third contig. Note that the contig $DF$ is most probably a repeat sequence. Figure 3.11 shows the resulting graph.

**Algorithm 5** Collapsing unique joins

```
1:  function CollapseUniqueJoins($reads$, $overlaps$)
2:      $disjoint\_set \leftarrow$ new DisjointSet of size $|reads|$
3:      $contigs \leftarrow reads$                         ▷ Convert all reads into 1-read contigs
4:      for all $overlap \in overlaps$ do                          ▷ Collapse edges
5:          if $overlap.A.degree_{overlap} = 1$ and $overlap.B.degree_{overlap} = 1$ then
6:              $contig1 \leftarrow disjoint\_set.find(overlap.A)$
7:              $contig2 \leftarrow disjoint\_set.find(overlap.B)$
8:              $larger \leftarrow disjoint\_set.union(contig1, contig2)$
9:              if $contig1 = larger$ then
10:                 $contigs_{contig1}.join(contig2)$
11:             else
12:                 $contigs_{contig2}.join(contig1)$
13:             end if
14:         end if
15:     end for
16:     for all $contig \in contigs$ do                        ▷ Erase empty contigs
17:         if $contig.size = 0$ then
18:             $contigs \leftarrow contigs \setminus contig$
19:         end if
20:     end for
21:     return $contigs$
22: end function
```

**Figure 3.10:** An example graph before forming contigs. Assume all edges are regular dovetail edges where arrow is pointing towards the second read.

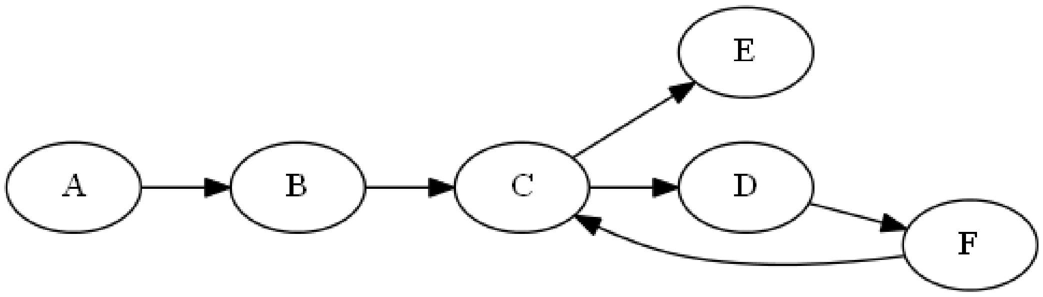
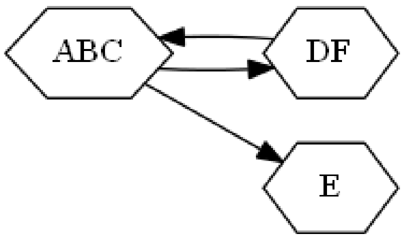


**Figure 3.11:** An example contig graph formed from the graph on Figure 3.10



### 3.5.5. Analysis

The time complexity of disjoint set is, as previously noted, approximately equal to amortized $O(1)$. We will do $3M$ (with $M$ being the number of overlaps) queries to the data structure, so we will spend in total $O(M)$ time in it. The memory requirement of that data structure is $O(N)$, where $N$ is the number of reads.

The contig set will have at most $O(N)$ elements, and all sets together will use a total of $O(N)$ memory to store the list of reads that they contain. In the worst case (if we always connect contigs of the same size), a read will change the contig it is in $\log_2 N$ times, so all the contig join operations may in total take $O(N \log N)$ time. Note that, because of the underlying data structure used to store the reads in a contig (`std::deque`), we may allocate some additional memory that won't be used, but the total should still stay $O(N)$.

In total we use additional $O(M + N)$ memory and $O(M + N \log N)$ time.

# 4. String Graph

## 4.1. Overview

The idea of the string graph is to provide a representation of the genome data in a way that can easily be used in reconstruction of said genome. So far we have been dealing with graphs whose building blocks were reads, overlaps and contigs. A string graph is fairly similar to the one we have been using, but the edges are a bit different, so the string graph also has some different properties.

All reads shall be made vertices of the string graph. Unlike the graph from Chapter 3, each overlap $o$ between reads $A$ and $B$ creates two edges - one from read $A$ to read $B$ and one from read $B$ to read $A$. Edges also have two additional properties, a label and a type.

The edge $f$ from $A$ to $B$ has a **label** which is equal to the hanging (remaining) part of the read $B$, and the label of the edge $g$ in the opposite direction is, of course, equal to hanging part of the read $A$. If the overlap is reverse complemented, we also reverse complement the labels.

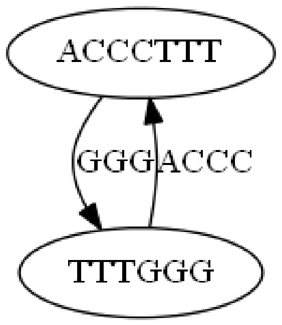
We shall define the **type** of edge $f$ to be equal to the suffix of read $A$ in overlap $o$, i.e. $f.type = o.suffix_A$, and, similarly, $g.type = o.suffix_B$. Recall that $suffix$ property can only be one of two values - "begin" or "end".

As Figure 4.1 depicts, reads $A =$ "ACCCTTT" and $B =$ "TTTGGG" would form two edges $f = E_{AB}$ and $g = E_{BA}$, whose labels would be "GGG" and "CCCA" respectively, while $f.type =$ "end" and $g.type =$ "begin".

One of the best properties of a string graph is that one can easily assemble the reads into genome: if we start with read $A$, the assembly of reads $A$ and $B$ is equal to the read $A$ concatenated with the label of edge $f = E_{AB}$. We can assemble the whole genome by going through all the vertices of the graph, but we must take care that we use different types of edges to enter and exit the graph.

If we managed to make just a single chain of vertices, that would mean that assembly is very simple - we just need to traverse it once and that is our assembly. Most of

**Figure 4.1:** An example of a string graph



the times, that is unfortunately not the case, but we do get long, chain-like paths. A chain-like path is a path where all the vertices except the first and last have only two neighbours. Those chain-like paths are actually contigs.

More often then not, the DNA is repetitive in many places. Those repeats are very long - sometimes even hundreds of thousands of nucleotide bases long and they need not be one after another. A string graph has the property that all repeats of the same data are collapsed in a single chain-like sequence. To know how many times a chain-like sequence is repeated, we need to compare the coverage of it with the coverage of other parts of the graph. We need to include all the removed reads and edges, though.

This definition is very similar to one found in Simpson i Durbin (2010), where the author builds upon the work done by Myers (2005), which is considered to be the first mention of the string graph. In that (older) paper, Myers describes how to construct a string graph using a completely new algorithm. He also, however, notes that one can also use the algorithms from Myers (1995) to remove the contained reads and transitive edges and then build the string graph from there. Considering we had an efficient working implementation of those algorithms, we have decided to build from there instead of working form ground up.

## 4.2. Construction

### 4.2.1. Input

We are given a list of reads and overlaps between them (without contained reads and without transitive edges).

### 4.2.2. Output

We need to return a string graph made from the given reads and overlaps.

### 4.2.3. Algorithm

To construct the string graph, when we have already removed the contained reads and transitive edges, we just need to follow the procedure described in Section 4.1. Therefore, we first iterate through all reads and make a corresponding vertex in a string graph. After that, we just need to iterate through all the overlaps and make the two edges - one from $A$ to $B$ and the other from $B$ to $A$.

One important implementation detail to note is that we do not need to make another copy of the hanging part for each of the labels. We can just store a pointer to the string and store a flag that tells us if we should really reverse complement it.

The pseudo code for the method described above can be found in Algorithm 6.

### 4.2.4. Example

Let's take Figure 4.2 as an example. We start with converting all reads (vertices in the graph) to vertices in the string graph. Then we just need to follow the rules and convert all the overlaps into edges in the string graph. In the end, we get Figure 4.3.

### 4.2.5. Analysis

As we are just doing two for loops and calling constant-time functions, the time required to create the string graph is equal to $O(N + M)$, where $N$ is the number of reads and $M$ is the number of overlaps.

Additional memory needed is also $O(N + M)$, as we need to store all vertices and edges in the string graph. Because we are only storing pointers for edge labels, we don't need much additional memory to store label data.

**Algorithm 6** Constructing a string graph

```
1: function ADDEDGES($overlap$, $edges$)
2:     $edge \leftarrow$ new Edge
3:     $edge.from \leftarrow overlap.A$
4:     $edge.to \leftarrow overlap.B$
5:     if $overlap.isReverseComplemented$ then
6:         $edge.label \leftarrow$ REVERSECOMPLEMENT($overlap.hang_B$)
7:     else
8:         $edge.label \leftarrow overlap.hang_B$
9:     end if
10:    $edge.type = overlap.A.suffix_{overlap}$
11:    $edges \leftarrow edges \cup edge$
12:    $edge.from \leftarrow overlap.B$
13:    $edge.to \leftarrow overlap.A$
14:    if $overlap.isReverseComplemented$ then
15:        $edge.label \leftarrow$ REVERSECOMPLEMENT($overlap.hang_A$)
16:    else
17:        $edge.label \leftarrow overlap.hang_A$
18:    end if
19:    $edge.type = overlap.B.suffix_{overlap}$
20:    $edges \leftarrow edges \cup edge$
21: end function
22: function MAKESTRINGGRAPH($reads$, $overlaps$)
23:     $graph \leftarrow$ new StringGraph
24:     for all $read \in reads$ do
25:         $graph.vertices \leftarrow graph.vertices \cup read$
26:     end for
27:     for all $overlap \in overlaps$ do
28:         ADDEDGES($overlap$, $graph.edges$)
29:     end for
30:     return $graph$
31: end function
```

**Figure 4.2:** An example of an overlap graph

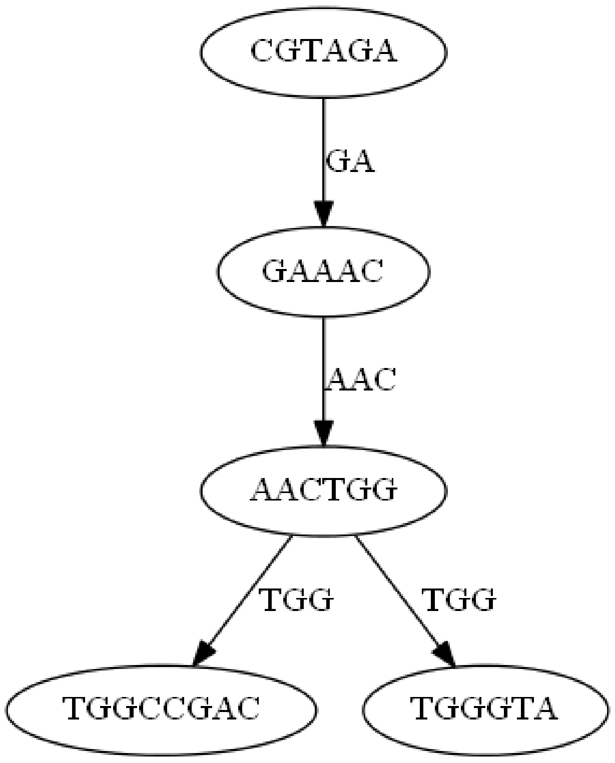
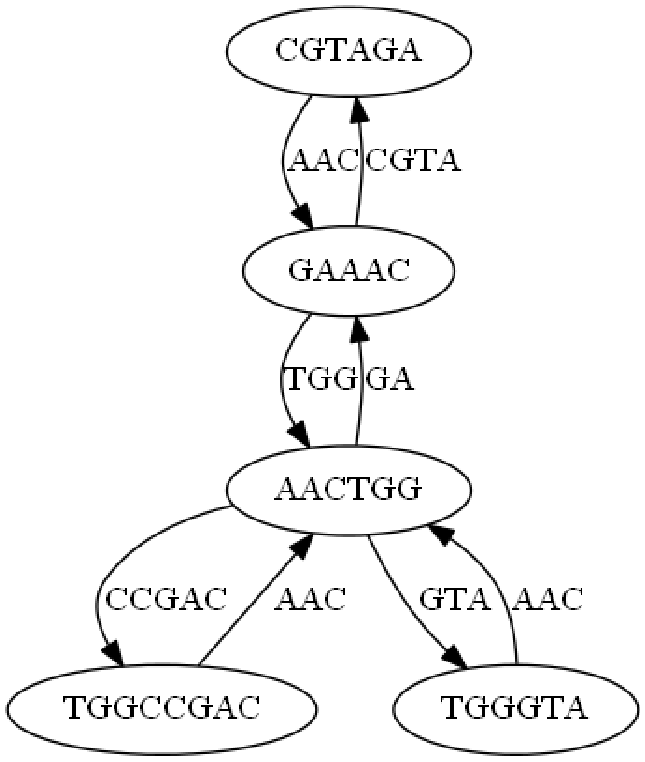


**Figure 4.3:** A string graph made from overlap graph on Figure 4.2



# 5. Implementation

## 5.1. Overview

The implementation of this thesis is built upon Osrecki (2014). For that work, Osrečki implements the overlap phase in a C++ program and this solution wraps around his implementation.

### 5.1.1. Code layout

All of the source code is found in folder `src/`, which is divided by different projects. The overlap step, and all it's relevant sources are found in `src/overlap/` directory, this implementation is found in `src/layout/` and tests are in `src/test/` directory.

There is a `Makefile` in the root of the project, which, when invoked, compiles the code and prepares it for debugging and analysis with valgrind. The executables and object files are placed in the `bin/` directory.

Some small genome samples are found in `sample/` directory, whereas `config/` directory holds different configuration files, primarily for Arcanist.

## 5.2. C++11

C++ is a general purpose programming language first developed in 1979 by Bjarne Stroustrup of Bell Labs, which started as C with classes. Besides the standards, Stroustrup et al. (1995) is considered to be the bible of the language. New standard, called C++ 2011, has been released in 2011 and is finally finding it's way to the developers. It offers multiple improvements over base language, ranging from range based for loops and multi-threading support to lambda functions.

There exist multiple different compiler suits, like the GNU G++, Digital Mars C++ compiler, Clang, and Visual C++ compiler, and all of them have different advantages.

For compilation of this implementation, GNU G++ 4.8.1. was used as it is readily available the selected Linux operating system.

Several of the new functionality has been instrumental in the design of this implementation. Shared, unique and other types of pointers are very useful for a programming language without a garbage collector, to decrease the risk of memory leaks. Range-based for loops also helped decrease the amount of boilerplate in algorithms, so they are much easier to both write and read.

## 5.3. Git

Git is a source control system created by Linus Torvalds in 2005 for Linux kernel development. Because of it's speed, distributed properties, and complete openness, it is one of the most popular revision control systems that exist today, despite having very little user friendliness.

There are many git repository hosts on the Internet - with Github and Bitbucket being the most popular ones. The repository for this implementation is hosted by Bitbucket, as it provides free private repositories which is not common for other providers.

## 5.4. Testing

### 5.4.1. Unit tests

Unit tests are small test that test just a certain feature of a software. For example, a unit test could test if our function that returns an object's name works as expected. They are the easiest way to test the code during development of the implementation.

For this thesis, a custom test framework has been made. A test needed to implement the following simple interface:

```
class UnitiggingTest {
 public:
  UnitiggingTest();
  virtual ~UnitiggingTest();
  virtual bool run()=0;
  overlap::Read* makeRead(const char* data);
};
```

All what was left to do was to register the test with the unit test runner, and the framework would then run it and inform us of the results and time taken to complete the test.

```
test::UnitiggingTestRunner ut;
ut.addTest(new test::UnitiggingIsTransitiveTest());
ut.run();
```

Of course, if the test needed access to private functions or members of a class, it needed to be declared as a friend to the class.

### 5.4.2. Valgrind

Valgrind is an open source GPL licensed command line tool for any kind of executables that tracks memory usage and can report most instances of memory leaks, invalid memory accesses and other memory issues. It was developed by Julian Seward and named after the entrance to Valhalla in Norse mythology [Seward et al. (2013)].

Valgrind comes with a call-graph generating profiler callgrind, which allows us to profile the code to find the bottlenecks, and with a heap profiler massif, which allows us to analyse the heap memory and who and why is using it. The default tool keeps track of all the allocated and accessed memory locations, so it can check if the memory referenced has been initialized and freed properly. The downside, however, is that using it slows the program by a factor of 4 or 5. What makes Valgrind great is that is fairly easy to use - to test an executable one simply needs to write `valgrind` in front of the command that is under test. Valgrind then outputs the issues it has found on the standard out, as can be seen on Figure 5.1.

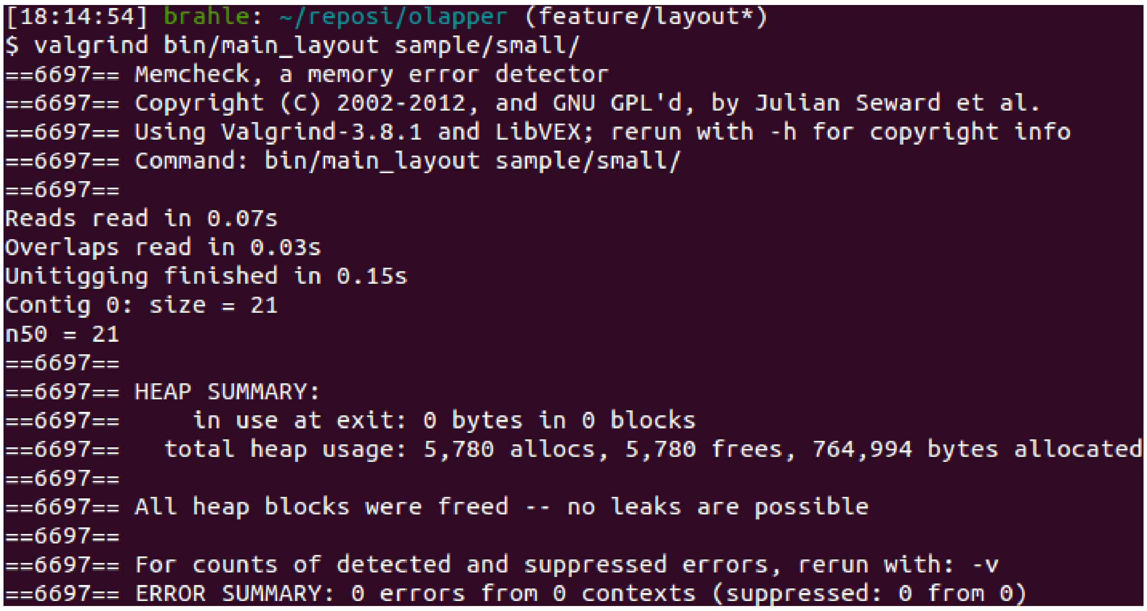
This implementation has been been tested in Valgrind on each test case, and Valgrind has consistently reported the absence of any memory related issues.

### 5.4.3. Phabricator

Phabricator is a web based tool that allows developers and product managers to quickly and efficiently collaborate on a project. It is being actively developed by Evan Priestly, who first started working on while he was in Facebook and has since open sourced it and started working full time on it.

Amongst a slew of its features, the code review has proven to be very important when working on a collaborative project. It allows for all of the involved people to

**Figure 5.1:** Using Valgrind



```
[18:14:54] brahle: ~/reposi/olapper (feature/layout*)
$ valgrind bin/main_layout sample/small/
==6697== Memcheck, a memory error detector
==6697== Copyright (C) 2002-2012, and GNU GPL'd, by Julian Seward et al.
==6697== Using Valgrind-3.8.1 and LibVEX; rerun with -h for copyright info
==6697== Command: bin/main_layout sample/small/
==6697==
Reads read in 0.07s
Overlaps read in 0.03s
Unitigging finished in 0.15s
Contig 0: size = 21
n50 = 21
==6697==
==6697== HEAP SUMMARY:
==6697==     in use at exit: 0 bytes in 0 blocks
==6697==   total heap usage: 5,780 allocs, 5,780 frees, 764,994 bytes allocated
==6697==
==6697== All heap blocks were freed -- no leaks are possible
==6697==
==6697== For counts of detected and suppressed errors, rerun with: -v
==6697== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

know what others are working on. The automated unit testing ensures that the changes that introduce bugs are minimal, and linters can help make sure the code is written in the same style across all files. Example of the interface can be found on Figure 5.2.

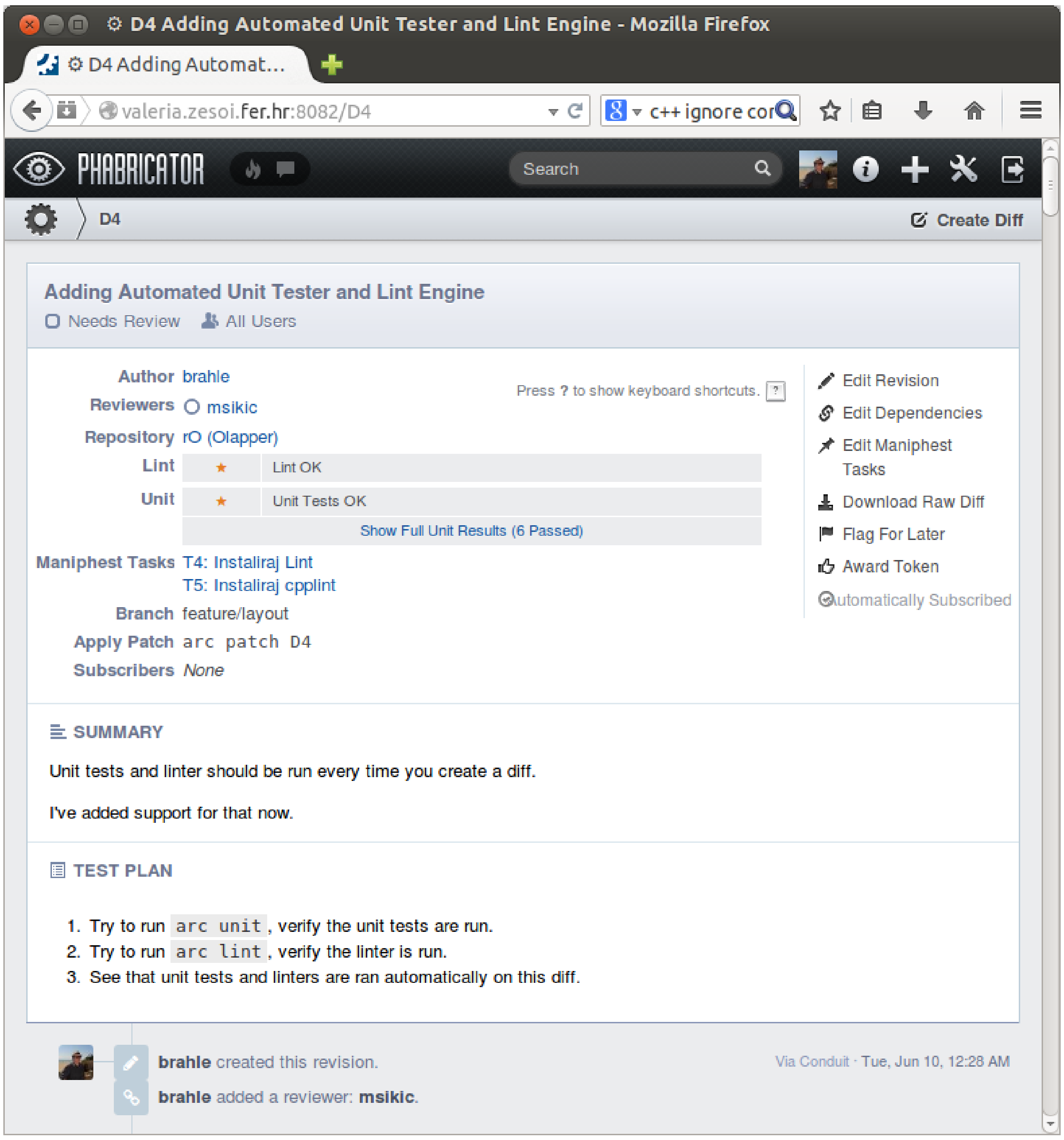
The command line interface for Phabricator is called Arcanist, with short name being arc. It offers a vast array of tools that help make code review as easy and pleasant as possible. To submit the last revision for review, all one needs to do is write the following commands

```
$ git commit -a -m 'Example commit'
$ arc diff
```

and complete the form that is presented. Arcanist would then automatically run lint and unit tests on the changed files. One could also invoke test automatically like so:

```
$ arc lint
$ arc unit
```

**Figure 5.2:** Phabricator' code review interface



Adding Automated Unit Tester and Lint Engine

Needs Review  All Users

| | |
|---|---|
| Author | brahle |
| Reviewers | msikic |
| Repository | rO (Olapper) |
| Lint | ★ Lint OK |
| Unit | ★ Unit Tests OK |
| | Show Full Unit Results (6 Passed) |
| Maniphest Tasks | T4: Instaliraj Lint<br>T5: Instaliraj cpplint |
| Branch | feature/layout |
| Apply Patch | `arc patch D4` |
| Subscribers | *None* |

SUMMARY

Unit tests and linter should be run every time you create a diff.

I've added support for that now.

TEST PLAN

1. Try to run `arc unit`, verify the unit tests are run.
2. Try to run `arc lint`, verify the linter is run.
3. See that unit tests and linters are ran automatically on this diff.

**brahle** created this revision. Via Conduit · Tue, Jun 10, 12:28 AM

**brahle** added a reviewer: **msikic**.

# 6. Results

In this chapter we will describe the results that we have created using the C++ implementation of the algorithms from the previous section. We will also compare them to other known works in the same field.

## 6.1. Measurement

### 6.1.1. Hardware

The computer used to get the results can be found in Table 6.1. Because the author has access to different machines, running different architectures, it proved to be opportune to use a virtual machine so work can easily be shared across different platforms. The tests were run on the fastest computer available to the author.

### 6.1.2. Process

Time measurements on the implementation were done in different ways. For this implementation, we have measured both the system time using UNIX's `/usr/bin/time`

**Table 6.1:** Testing Computer Tech Specs

| | |
|---:|---|
| Host OS | Windows 8.1 Pro 64-bit |
| Virtual OS | Ubuntu 13.10 32-bit |
| VM Player | VMware(R) Player 6.0.1 |
| CPU | Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz (8 CPUs), 3.5GHz |
| Total Memory | 16 GB RAM |
| VM Memory | 3.9 GB RAM |
| Hard disk #1 | 228.9 GB (KINGSTON SVP200S3240G) |
| Hard disk #2 | 953.5 GB (WDC WD10EARX-00N0YB0) |

and the time taken for every step of the algorithm using the `clock()` utility from C++ header `ctime`. The example usage of the latter looks similar to the following:

```
clock_t start = clock();
execute_function();
clock_t end = clock();
printf("Time taken: %.2lfs", static_cast<double>(end - start) /
 CLOCKS_PER_SEC);
```

Other software's time was measured using their own reporting (if available) or using the `/usr/bin/time`. All times that are going to be presented in the following sections are averages from three executions, except for those programs whose execution took longer than 10 minutes where we only run it once.

Peak memory is reported by running the program through Valgrind's tool massif, which was briefly explained in Section 5.4.2. As it slows down program's execution, the time wasn't measured while running it.

## 6.2. Part of *Escherichia coli*

This test case was made from the first 35,000 nucleotide bases of *Escherichia coli*. The data was acquired along with the code of Readsim [Schmid et al. (2006)], a program that was used to simulate reads. By default, the distribution of lengths is not uniform, but we can specify the average length, which is fairly similar to the situation found in real data. We have included a test on the data with a normal distribution of read lengths. We have also included a test with low coverage.

Table 6.2 shows the results we got when running our program on the generated data. If the reads are of similar sizes, like it is the case with the first example, we have very few contained reads, however, the number of transitive edges is very large. In the case of non-uniform reads, we have a pretty huge amount of contained reads, but the number of transitive edges is still relatively big. When we are dealing with data where coverage is low, we have a much smaller amount of reads to work with, hence the small number of overlaps. Because of that, we get a larger number of smaller contigs.

We can see the string graph of the test case with default data (non-uniform reads of length 2000, coverage of 10) created after we removed the contained reads on Figure 6.1 and the one created after removing transitive edges on Figure 6.2. Obviously, the former is just there for the illustration, as we would always be using the latter.

**Table 6.2:** Test case information for part of *E. coli*

| | | | |
|---:|:---:|:---:|:---:|
| Size of genome | 35 KB | 35 KB | 35 KB |
| Number of reads | 142 | 167 | 39 |
| Approximate coverage | 10 | 10 | 3 |
| Average read length | 2000 (normal) | 2000 | 2000 |
| Total size of all reads | 280 KB | 290 KB | 63 KB |
| Number of overlaps | 1,173 | 1,323 | 50 |
| Time to find the overlaps | 1s | 1s | 0s |
| Total time | 0.01s | 0.01s | 0.00s |
| Graph pruning time | 0.00s | 0.00s | 0.00s |
| String graph construction time | 0.00s | 0.00s | 0.00s |
| Peak memory usage | 563.7 KB | 559.3 KB | 183.0 KB |
| Contained reads count | 1 | 138 | 18 |
| Contained reads percentage | 0.70% | 82.63% | 46.15% |
| Remaining edges count | 1,149 | 56 | 9 |
| Remaining edges percentage | 97.95% | 4.23% | 18.00% |
| Transitive edges count | 1,009 | 36 | 1 |
| Transitive edges percentage | 87,82% | 64.29% | 11.11% |
| Found contigs | 1 | 1 | 3 |
| n50 of the contig set | 141 | 21 | 7 |
| Minimus time | 0s | 0s | 0s |
| Minimus Found contigs | 1 | 1 | 5 |
| Minimus n50 of the contig set | 141 | 31 | 6 |

**Figure 6.1:** A string graph of *Escherichia coli* after removing contained reads

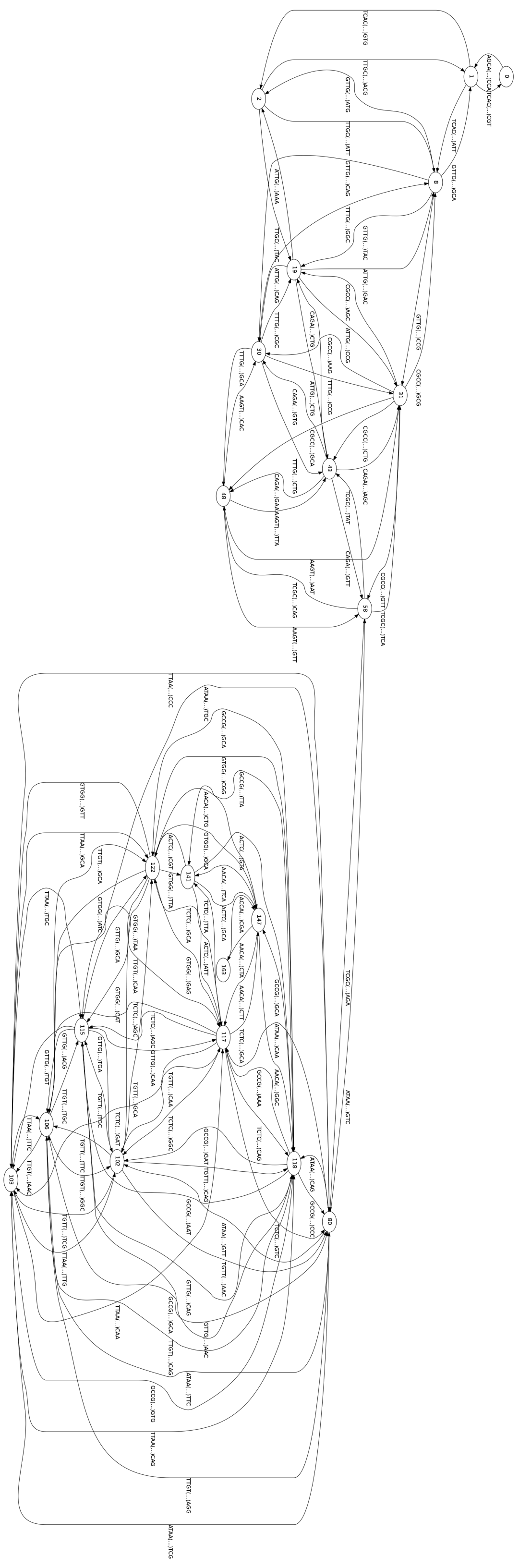
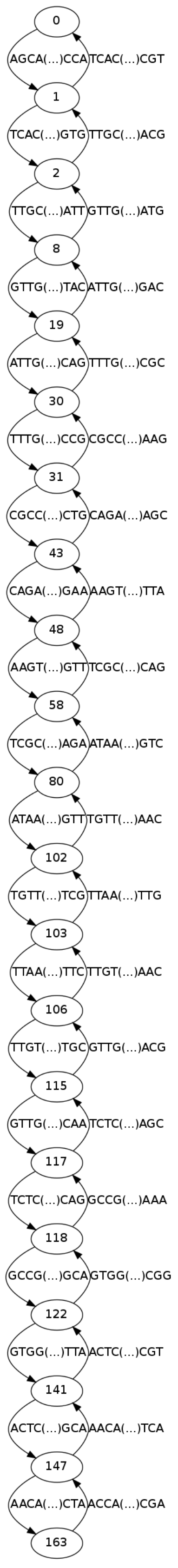


**Figure 6.2:** A string graph of *Escherichia coli* after removing transitive edges



## 6.3. Full *Escherichia coli*

This is the full genome of *Escherichia coli*, which contains around 4.7 million nucleotide bases. We have tested our implementation on three different test cases, with different lengths of reads - 500, 2000 and 10000. In all cases our program did the task it is designed to solve in under 2 seconds, with the actual algorithmic work being at most half a second (the rest was spent on IO). All test cases were generated using Readsim, and the error rates were set to be equal to the ones Pacific Biosciences claims for their sequencer.

The results are displayed in Table 6.3. We see that, as we increase the number of reads, we decrease the number of contigs and the number of reads in each contig decreases. The increased memory consumption occurs when we have a larger number of longer contigs because of the inefficient implementation and using a `std::deque` instead of a custom data structure.

Figure 6.3 shows how memory usage looks on the test case where average read length is 2000. The first part, where we slowly increase our memory is the input phase. The memory used to store reads is represented with the color red, and overlaps with color yellow. Orange coloured is the memory used by the `std::deque` when creating contigs. String graph is comparatively so small that it got bundled in the "other data" category and is shown in transparent red.

Figure 6.4 shows where our algorithm spends most time, again on the same test case where average read length is 2000. It is dominated by three rectangle-like shapes, first two of which are dedicated to input and the last one to construction of the labels that is used in output. That means we spend most CPU cycles doing work that is related to input and output.

Minimus produces comparable results within the same time frame. It's step of finding contigs, which is doing work similar to our implementation exhibits a bit slower performance for the first two cases (with shorter reads). Minimus measures time for each step with a resolution of a second, so we cannot really compare speed on the third test case. The slight difference of the results can be explained by different constants, most glaring one being the expected percentage of errors in a read.

**Table 6.3:** Test case information for full *E. coli*

| | | | |
|---:|:---:|:---:|:---:|
| Size of genome | 4.7 MB | 4.7 MB | 4.7 MB |
| Number of reads | 94,611 | 23,406 | 4,601 |
| Approximate coverage | 10 | 10 | 10 |
| Average read length | 500 | 2,000 | 10,000 |
| Total size of all reads | 49.8 MB | 45.4 MB | 42.6 MB |
| Number of overlaps | 900,414 | 249,674 | 45,988 |
| Time to find the overlaps | 9m 27s | 25m 52s | 2h 14m 25s |
| Total time | 1.52s | 0.64s | 0.11s |
| Graph pruning time | 0.41s | 0.12s | 0.01s |
| String graph construction time | 0.15s | 0.03s | 0.01s |
| Peak memory usage | 159.7 MB | 75.0 MB | 49.1 MB |
| Contained reads count | 78,370 | 21,103 | 4,103 |
| Contained reads percentage | 82.83% | 88.46% | 98.18% |
| Remaining edges count | 24,308 | 6,396 | 1,304 |
| Remaining edges percentage | 2.70% | 2.56% | 2.84% |
| Transitive edges count | 14,948 | 4,009 | 817 |
| Transitive edges percentage | 61.49% | 62.69% | 62.65% |
| Found contigs | 117 | 43 | 13 |
| n50 of the contig set | 131 | 96 | 71 |
| Minimus time | 10s | 2s | 0s |
| Minimus contigs found | 97 | 47 | 11 |
| Minimus n50 of the contig set | 140 | 87 | 75 |

**Figure 6.3:** A massif graph showing heap usage when running on the full *Escherichia coli* test case with the average read length of 2000

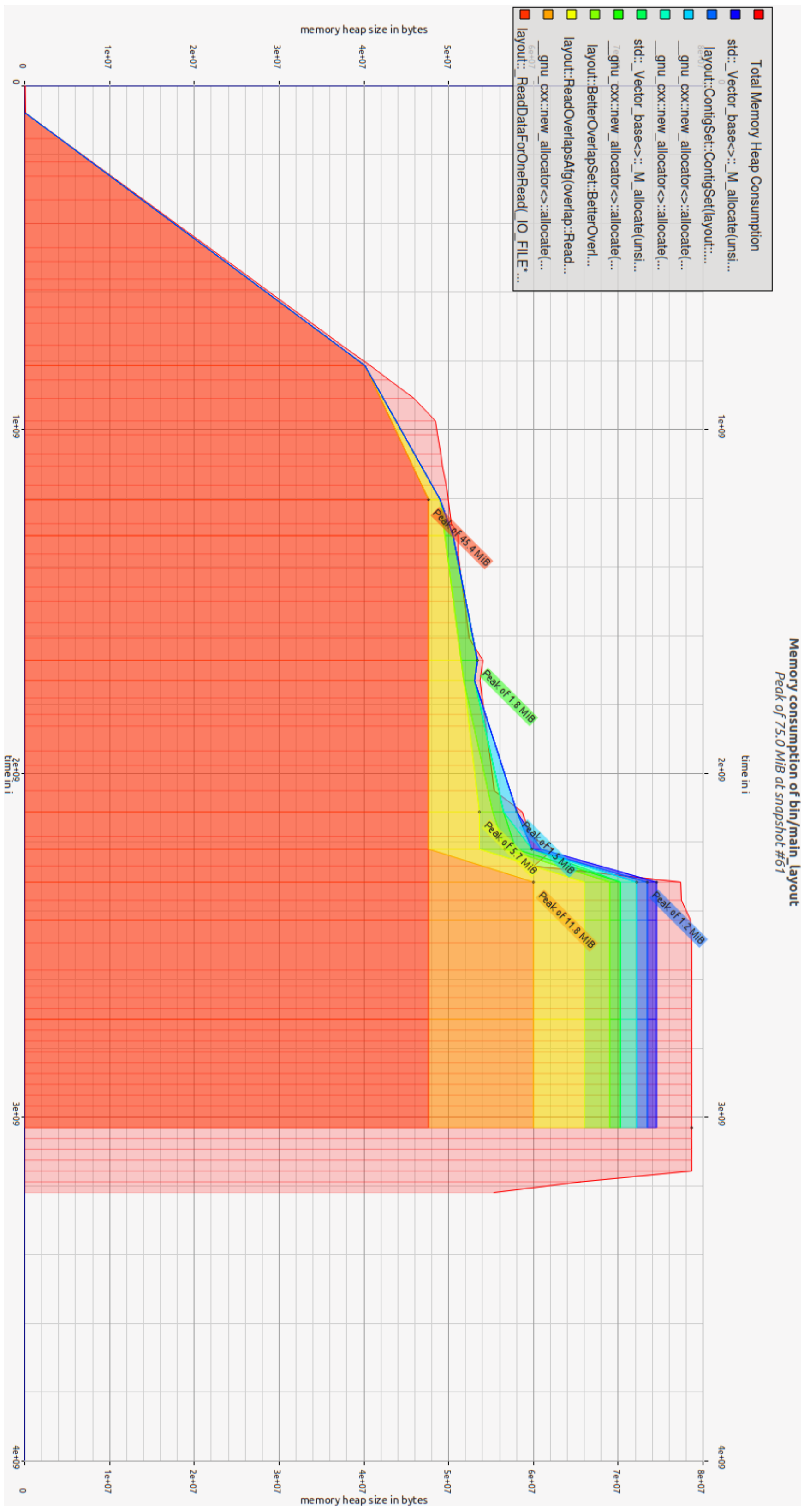
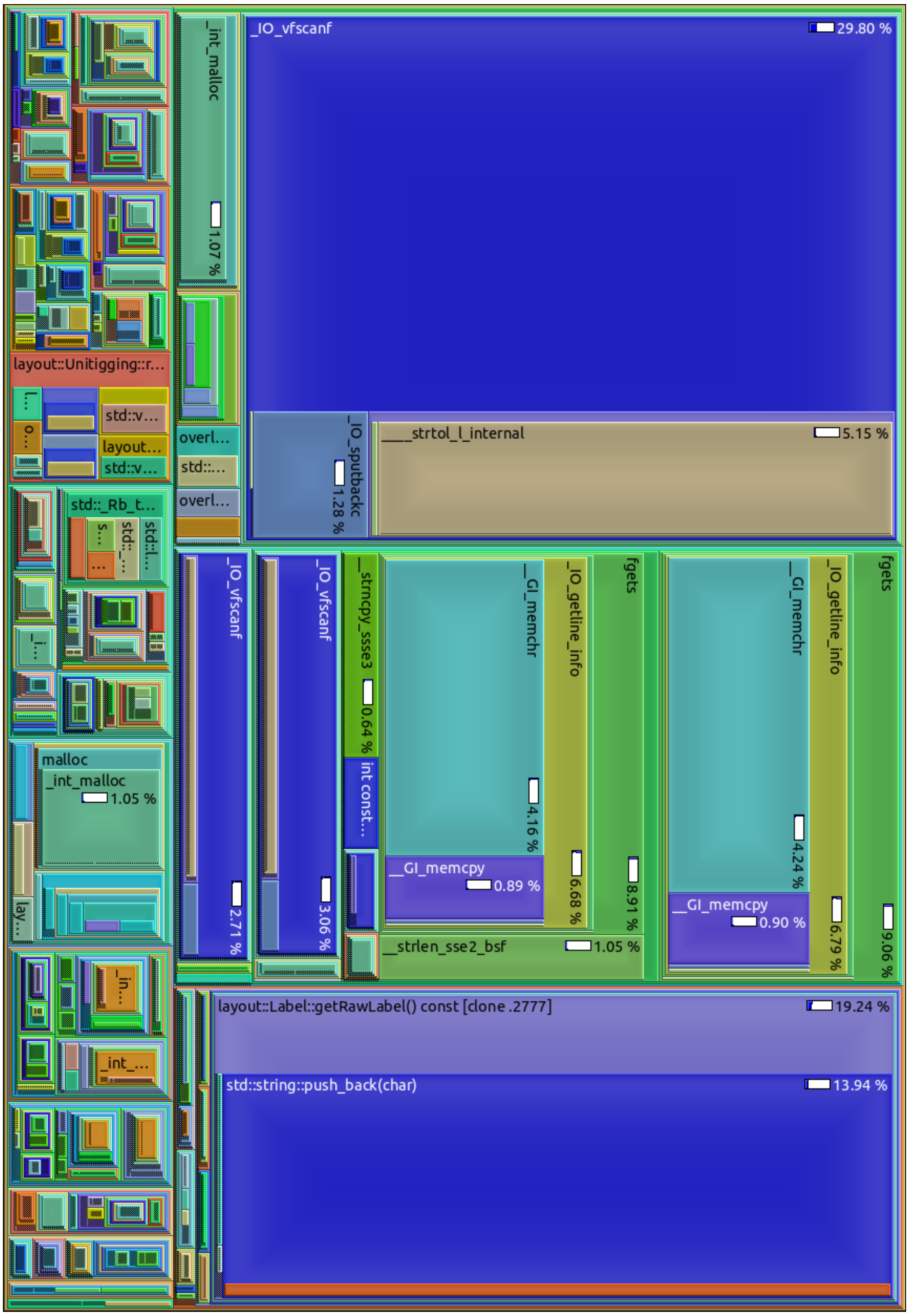


**Figure 6.4:** A callgrind graph showing time spent in parts of the code when running on the full *Escherichia coli* test case with the average read length of 2000



# 7. Conclusion

For this thesis, we have created an efficient implementation that solves the layout problem. Methods that are used to simplify the overlap graph without losing much information have been described and implemented in the C++ programming language.

Other parts of the genome assembly are slower, both in theoretical terms of the big-Oh notation, and in practical terms of program runtime. As an example, finding overlaps for the full *E. coli* took 25.5 minutes, while the step of simplifying the graph takes less than a second, and is even then dominated by input and output. It is the opinion of the author that it might be better to focus work on improving other parts of the genome assembly, as this is not the place where one should look to find the biggest wins in terms of speed.

Nonetheless we can still improve the performance of this work. The easiest way to do so is to find the bottlenecks - and it seems that currently those are found in the input and output sector. We can approach those in a few different ways: one is to use lower level functions to do the actual work, the other is to use a different format and the third is to throw hardware towards the problem and use hard disks with lower latency and faster read speeds (like solid-state drives). Most of the input done is already done with fairly low level C functions, so that wouldn't improve the speed too much. Changing the disk drive can also only help to a certain extent.

A different format, however, would probably be the way to go. As the current format is a plain-text file, that, for the vast majority of its contents, is using 8 bits to represent one of the four characters (A, C, G and T), the speed could theoretically be improved 4 times if we started using a binary format simply because the file size would decrease approximately 4 times. A binary file is also much easier to parse, so that would help improve the performance even further.

The other issue we have is the "hidden" memory requirements we have when we create contigs. The offender is the `std::deque` data structure, which allocates a lot of unused memory as it grows - which will be the case if the coverage is large or the reads are short. To solve the issue, we could implement our own data structure, which would be similar to a doubly linked list, but with the additional requirement of being able to quickly reverse the contents. Additionally, we could also delete the contigs as soon as they are joined with another contig, and not at the end of the step.

# BIBLIOGRAPHY

Eric L Anson i Eugene W Myers. Realigner: a program for refining dna sequence multi-alignments. *Journal of Computational Biology*, 4(3):369–383, 1997.

Madeleine Price Ball. Dna chemical structure, 2013. URL `http://commons.wikimedia.org/wiki/File:DNA_chemical_structure.svg`.

R. A. Bowen. The extremes in chromosome number, 06 2014. URL `http://arbl.cvmbs.colostate.edu/hbooks/genetics/medgen/basics/minmax_chromos.html`.

Toft C. Fares M. A. Commins, J. Whole genome shotgun sequencing versus hierarchical shotgun sequencing, 11 2011. URL `http://en.wikipedia.org/wiki/Shotgun_sequencing#mediaviewer/File:Whole_genome_shotgun_sequencing_versus_Hierarchical_shotgun_sequencing.png`.

Paulien Hogeweg. The roots of bioinformatics in theoretical biology. *PLoS computational biology*, 7(3):e1002021, 2011.

Eugene W. Myers. Toward simplifying and accurately formulating fragment assembly. *Journal of Computational Biology*, Summer(2(2)):275–90, 1995. URL `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.9658&rep=rep1&type=pdf`.

Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(suppl 2):ii79–ii85, 2005. doi: 10.1093/bioinformatics/bti1114. URL `http://bioinformatics.oxfordjournals.org/content/21/suppl_2/ii79.abstract`.

Matija Osrecki. Otkrivanje preklapajucih dna ocitanja. Magistarski rad, Faculty of Electrical Engineering and Computing, 2014.

Etienne Paux, Pierre Sourdille, Jérôme Salse, Cyrille Saintenac, Frédéric Choulet, Philippe Leroy, Abraham Korol, Monika Michalak, Shahryar Kianian, Wolfgang Spielmeyer, et al. A physical map of the 1-gigabase bread wheat chromosome 3b. *science*, 322(5898):101–104, 2008.

Jaume Pellicer, Michael F Fay, i Ilia J Leitch. The largest eukaryotic genome of them all? *Botanical Journal of the Linnean Society*, 164(1):10–15, 2010.

Tobias Rausch, Sergey Koren, Gennady Denisov, David Weese, Anne-Katrin Emde, Andreas Döring, i Knut Reinert. A consistency-based consensus algorithm for de novo and reference-guided sequence assembly of short reads. *Bioinformatics*, 25(9): 1118–1124, 2009.

R Schmid, SC Schuster, M Steel, i D Huson. Readsim - a simulator for sanger and 454 sequencing. *submitted*, 1000(2000):3000, 2006.

Jeremy Schmutz, Jeremy Wheeler, Jane Grimwood, Mark Dickson, Joan Yang, Chenier Caoile, Eva Bajorek, Stacey Black, Yee Man Chan, Mirian Denys, et al. Quality assessment of the human genome sequence. *Nature*, 429(6990):365–368, 2004.

Robert Sedgewick. *Algorithms in c++, parts 1-4 (fundamental algorithms, data structures, sorting, searching)*. Addison-Wesley, 1998.

Julian Seward, Cerion Armour-Brown, Christian Borntraeger, Jeremy Fitzhardinge, Tom Huges, Petar Jovanovic, Dejan Jevtic, Florian Krohm, Carl Love, Maynard Johnson, Paul Mackerras, Dirk Mueller, Nicholas Nethercote, Bart Van Assche, Robert Walsh, Philippe Waroquiers, i Josef Weidendorfer. Valgrind faq, 2013. URL `http://www.valgrind.org/docs/manual/faq.html`.

Jared T Simpson i Richard Durbin. Efficient construction of an assembly string graph using the fm-index. *Bioinformatics*, 26(12):i367–i373, 2010.

Daniel D Sommer, Arthur L Delcher, Steven L Salzberg, i Mihai Pop. Minimus: a fast, lightweight genome assembler. *BMC bioinformatics*, 8(1):64, 2007.

Bjarne Stroustrup et al. *The C++ programming language*. Pearson Education India, 1995.

Wikipedia. Dna, 06 2014. URL `http://en.wikipedia.org/wiki/DNA`.

**Simplification of the Overlap Graph**

**Abstract**

Today, a lot of different DNA assembly methods exist and are being actively developed. In this thesis, we show how to efficiently prune unnecessary reads and overlaps from those supplied to us in the process of DNA assembly. We show algorithms for removing contained reads, transitive edges, collapsing unique joins and creating a string graph. A C++ implementation has also been provided and tested, with results presented in this work.

**Keywords:** Layout,DNA assembly,String graph,C++

**Pojednostavljenje grafa preklapanja**

**Sažetak**

U današnje doba, postoji mnogo različitih metoda sekvenciranja DNK koje se i dalje razvijaju. U ovome radu, prikazujemo kako efikasno odvojiti nepotrebna očitanja i preklapanja iz skupa očitanja i preklapanja koje dobijemo u procesu sekvenciranja DNK. Prikazujemo algoritme za micanje sadržanih očitanja, tranzitivnih bridova, spajanje jedinstvenih spojki i stvaranje grafa niza znakova. Napravljena je implementacija u C++-u i rezultati testiranja su prikazani u ovome radu.

**Ključne riječi:** Layout,slaganje DNK,Graf nizova znakova,C++