UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER's THESIS no. 1125

Scaffolding using long error-prone reads

Marko Čulinović

Zagreb, June 2015.

Umjesto ove stranice umetnite izvornik Vašeg rada.

Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`*.*

Firstly, I'd like to thank my family for the support they have given me over the years.

I'd also like to thank my mentor Mile Šikić for his help and constant motivation.

In the end, honourable mentions go to Luka Šterbić for his help with the implementation and to my brother Filip Čulinović for his help with writing this thesis in English.

Contents

1. Introduction **1**

2. Preliminaries **3**

2.1. Oxford Nanopore Technologies . . . 3
2.2. Alignment Algorithms and Tools . . . 4
2.2.1. The Smith-Waterman Algorithm . . . 5
2.2.2. Burrows-Wheeler Aligner . . . 8
2.2.3. SAM Format . . . 8
2.3. Graphs . . . 9
2.3.1. Depth-First Search . . . 10
2.3.2. Topological Sort . . . 13

3. Methods **15**

3.1. Finding possible extensions . . . 15
3.2. Majority vote . . . 16
3.2.1. Simple extension . . . 18
3.2.2. Local realignment . . . 18
3.2.3. Local realignment with global realignment of dropped reads . 21
3.3. POA . . . 21
3.3.1. POA algorithm . . . 21
3.3.2. Extension using POA consensus . . . 27
3.4. Scaffolding . . . 27

4. Implementation **28**

4.1. General overview . . . 28
4.2. External dependencies . . . 30
4.3. Code layout . . . 30
4.3.1. CPPPOA . . . 30

4.3.2. ONTscaffolder 30
4.4. Scripts 32

5. Results 33
5.1. Tools and methods for results evaluation 33
5.2. Data 33
5.3. *E. coli* PacBio reads results 35
5.3.1. One gap of length 1000bp 35
5.3.2. Four gaps of length 1000bp 35
5.3.3. One gap of length 5000bp 37
5.4. *E. coli* Oxford Nanopore reads results 37
5.4.1. One gap of length 1000bp 37
5.4.2. Four gaps of length 1000bp 38
5.5. Discussion 38

6. Conclusion 41

Bibliography 42

1. Introduction

Field of Bioinformatics is currently one of the most fast-growing scientific areas. Its real-world application is numerous. For example, it is used for identifying correlations between gene sequences and diseases, predicting protein structures from amino acid sequences, aiding in the design of novel drugs and tailoring treatments to individual patients based on their DNA sequences (pharmagenomics) [1].

Sequencing is a process of determining the order of individual nucleotid bases - Adenyne, Thymine, Cytosine, Guanine and Uracil - inside of the RNA or DNA chain. After sequencing the humane genome in 2003, modern biology produces as many new algorithms as any other fundamental realm of science. One of the the major challenges in bioinformatics is genome assembly. As I explain this process, I will also explain some basic terms in this field. Second generation of devices for genome sequencing (i.e. sequencers) produce short reads. Therefore, the biggest challenge is how to assemble these short reads into a unique sequence. Process of assembling short reads to create full-length, sometimes novel, sequences is called *de novo* assembly. In an ideal case, an assembled genome should have one sequence without gaps or unknown parts. This is not the case in practice, where an assembled genome is often represented with hierarchical data structure which maps reads into assumptive reconstruction of targeted genome. In this structure, reads given by sequencers are clustered into contigs, and afterwards, contigs into scaffolds. Contigs consist of sequence of mutually overlapped reads. Scaffolds define contigs order, orientation and gap size between them. This is called *draft* genome. [2]

Third generation of sequencers enabled production of longer reads as output. Oxford Nanopore Technologies device has a great new technology for genome sequencing which enables sequences practically unlimited in length. Currently the longest sequence was 180 000 base pairs (bp) long, contrary to Illumina, 2nd generation ($\approx$ 150 bp) and PacBio, 3rd generation ($\approx$ 5000 bp - 60000 bp) technologies. This is great, because the biggest problem with Illumina sequences is that they are quite short and genomes have repetitive parts which can be longer than that. During *de novo*

assembly these parts cannot be unambiguously solved. So the output of assembly is given as previously mentioned *draft* genome. Long reads do not have a problem with repeats, but they have other significant problem - enormous error rate. For example, PacBio reads have $\approx$15% error rate and Oxford Nanopore reads have error rate greater than 25% (even great as 40%). [2]

Currently there are a great number of *draft* genomes of multiple species/organisms which are assembled from Illumina reads and were never finished, therefore presented in form of sequence of contigs. Idea of this paper is to use Oxford Nanopore reads for scaffolding and to fill the gaps between contigs. With this approach some of unfinished genomes could be finalized.

This master thesis is a part of a collaborative project which is publicly available at https://github.com/mculinovic/ONTscaffolder.

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

Chapter 3 describes methods used to fill gaps in draft genomes. Two methods will be explained in detail.

Chapter 4 will describe the implementation development, tests executed and the technologies used to maintain and document code.

Chapter 5 will show results of the implementation on two datasets and three different simulated *draft* genome examples.

Chapter 6 will offer final thoughts, discussion, comparison of results and potential future work.

2. Preliminaries

In this chapter basic knowledge required to understand the topic of this thesis will be covered. Firstly, it will be explained how third generation Oxford Nanopore sequencers work and how they can achieve reads much longer than second generation sequencers. Afterwards, alignment algorithms will be introduced and differences between local and global alignment will be clarified. As a necessary example for POA algorithm (explanation in section 3.3), the Smith-Waterman algorithm for local alignment will be described. Burrows-Wheeler aligner (BWA), tool used for aligning reads to contigs/genomes will be presented as well. Another prerequisite for understanding POA algorithm is to understand what are graphs, and how algorithms (Depth-First Search and Topological Sort) modified for graphs work.

2.1. Oxford Nanopore Technologies

Oxford Nanopore Technologies is developing nanopore-based sensing technologies for the analysis of biological molecules like DNA or RNA. The technology is based on protein nanopore. At the core of a protein is a nano-scaled hollow tube. Oxford Nanopore designs and manufactures described nanopore structures for a range of applications. In nature, nanopores form holes in membranes. In Oxford Nanopore Technologies, nanopore is inserted into an highly electrically resistant membrane created by a synthetic polymer. By applying a potential across the membrane which is bathed in electric chemical solution, an ionic current can be generated through the nanopore. Single molecules that enter the nanopore or pass near its aperture cause characteristic disruptions in the current - this is known as a nanopore signal. By measuring that disruption it is possible to identify the molecule in question. For example, this system can be used to distinguish between the four standard DNA bases G, A, T and C, and modified bases. Using microchip fabrication techniques, Oxford Nanopore has developed devices that enable highly scalable arrays of this nanopore to be used for real-time sensing of biological molecules. The number of nanopores used can be scaled up to

-	A	T	G	C	T	T	A	G
T	A	T	C	C	-	T	A	G

$\underbrace{\qquad\qquad\qquad\qquad}_{\sum=3}$

Figure 2.1: Example of the *edit* distance. If value of every operation is 1, total distance between sequences is 3

G	T	C	G	T	G	A	A	C	C	T	A	A	G	A	G	C	C
G	T	C	G	-	-	-	-	C	-	-	A	-	-	-	-	C	C

Figure 2.2: Global alignment

industrial levels. [3]

2.2. Alignment Algorithms and Tools

Sequence alignment is often the first step in bioinformatics analysis. It can be interpreted as a way to transform one sequence into another. It is one of the oldest and most explored problems in bioinformatics. First measure from the aforementioned area is the one which explains how many operations are needed to transform one sequence into another. This measure is called Levenshtein distance[4], also known as *edit* distance. We differ between three possible operations on one element:

- Insertion means inserting one element into the first sequence so that it matches the second sequence, e.g. fih to fi**s**h
- Deletion means deleting one element from the first sequence so that it matches the second sequence, e.g. **s**hip to hip
- Substitution means changing one character of the first sequence to a character on matching position in another sequence, e.g. **w**ay to **s**ay

As an addition to calculating distance between sequences (Figure 2.1), bioinformaticians needed information about how alignment looks. First algorithm to solve this problem was the one presented by Needleman and Wunsch and was based on dynamic programming[5]. There are two types of alignments in bioinformatics - local and global.

Global alignments try to align every character in every sequence and are used when sequences are similar in length (Figure 2.2).

C T C A A

C T - A A

Figure 2.3: Local alignment example ([2]) between two sequences, s = ACCTAAGG and t = GGCTCAATCA.

Contrary, local alignments try to align regions of similarity within sequences (Figure 2.3). Example of a local alignment algorithm also based on dynamic programming is Smith-Waterman algorithm[6] and will be explained in detail in section 2.2.1.

In this thesis, the BWA ([7]) tool was used to align long reads to *draft* genome and it will be explained in section 2.2.2. Output of this software is presented in SAM format[8], which will be explained in detail in section 2.2.3.

2.2.1. The Smith-Waterman Algorithm

As was mentioned beforehand, the Smith-Waterman algorithm is a deterministic dynamic programming method for finding local alignment between sequences. When it comes to local alignment it is much better to maximize similarity than to minimize previously mentioned distance between sequences. This similarity measure can achieve positive and negative values, but we were interested only in positive similarity, because negative similarity means that there is no similarity between sequences, and it can be set to zero. So we can define two-dimensional dynamic programming similarity matrix (S) between two sequences (s and t) as follows:

$$S_{i,j} = \begin{cases} 0 & i = 0 \lor j = 0 \\ max \begin{cases} 0 \\ S_{i-1,j-1} + sim(s_i, t_j) \\ S_{i-1,j} + gap_cost \\ S_{i,j-1} + gap_cost \end{cases} & \text{otherwise} \end{cases} \tag{2.1}$$

In the similarity matrix (2.1) there is one function sim, and one parameter gap_cost. Function sim calculates if characters in sequences - s_i and t_j - are equal or not. A situation where characters are equal represents *match* operation and the function returns positive value, e.g. 4. If characters are not equal, we call this situation *mismatch* and the function returns negative value, e.g. -2. Parameter gap_cost represents aforementioned *deletion* or *insertion* operations, and it has negative value. We can distinguish

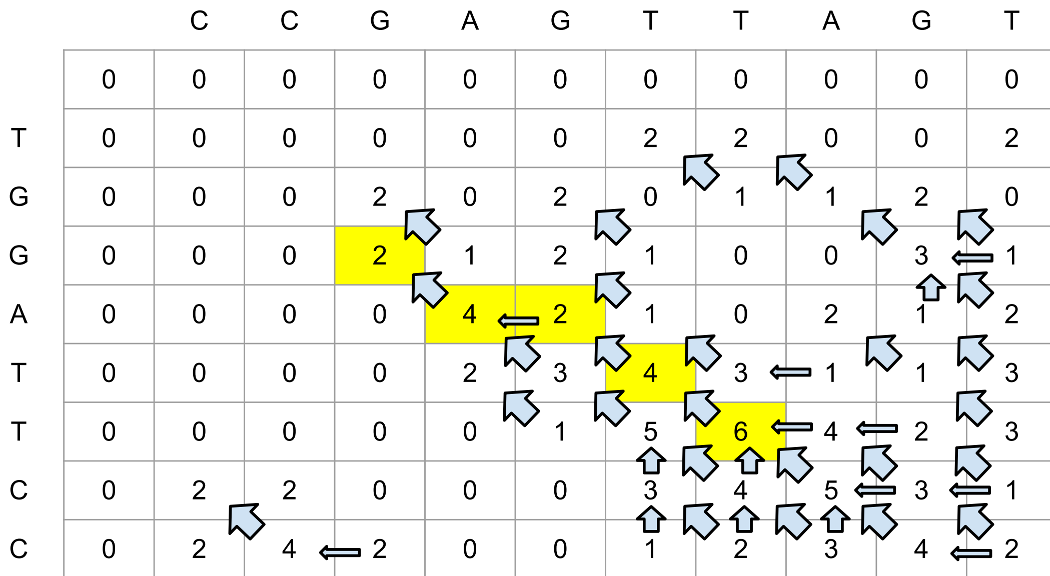

Figure 2.4: Smith-Waterman example ([2]). Values in the matrix are similarity values and arrows symbolize a backtracking matrix. Best alignment path is colored in yellow.

between opening gap cost and extending gap cost by penalizing gap extension less than opening a new gap.

Within the similarity matrix we only have information about maximum similarity between sequences. If we want to get actual alignment this information has to be stored in another two dimensional table. This is called a backtracking matrix (2.2).

$$B_{i,j} = \begin{cases} start & S_{i,j} = 0 \lor i = 0 \lor j = 0 \\ deletion & S_{i,j} = S_{i-1,j} + gap_cost \\ insertion & S_{i,j} = S_{i,j-1} + gap_cost \\ match \quad or \quad mismatch & \text{otherwise} \end{cases} \tag{2.2}$$

By using a backtracking matrix, we can reconstruct the alignment path. Reconstruction/backtracking starts from element with maximum similarity value in similarity matrix, and ends when an element with zero similarity value occurs, i.e. when value in backtracking matrix is $start$.

An example of how the Smith-Waterman Algorithm calculates alignment is given in figure 2.4. Operation values used in this example were $+2$ for $match$, -1 for $mismatch$ and -2 as gap_cost. Space and time complexity of the stated algorithm (Algorithm 1) is $O(mn)$ where m and n are lengths of sequences. The algorithm returns maximum similarity value and best alignment path found.

Algorithm 1 Smith-Waterman (sequence s, sequence t)

```
1: max_sim_value = 0
2: max_sim_position = (0, 0)
3: for i = 1 → |s| do
4:     S[i, 0] = 0
5:     B[i, 0] = start
6: end for
7: for i = 1 → |t| do
8:     S[0, i] = 0
9:     B[0, i] = start
10: end for
11: for i = 1 → |s| do
12:     for j = 1 → |t| do
13:         calculate S[i, j] as in equation 2.1
14:         calculate B[i, j] as in equation 2.2
15:         if S[i, j] > max_sim_value then
16:             max_sim_value = S[i, j]
17:             max_sim_position = (i, j)
18:         end if
19:     end for
20: end for
21: extract path p by backtracking matrix B starting from max_sim_position until
    value of element in B is start
22: return p, max_sim_value
```

2.2.2. Burrows-Wheeler Aligner

The BWA ([7]) is a software package, freely available at https:// github.com/lh3/bwa. In this thesis it is used to map long reads (PacBio and Oxford Nanopore) to draft genome. This is done by using the BWA-MEM algorithm, currently the latest and fastest algorithm in BWA package. Before being able to map reads to a genome, BWA first needs to construct FM index for reference genome. Alignment of reads to draft genome is performed in verbose mode - meaning that one read can be aligned to multiple places in genome. Furthermore, every command is done in parallel. Commands used in this thesis are:

- for indexing draft_genome:
  $bwa\,index$ <$draft_genome$>
- for indexing contig in global realignment method(3.2.3):
  $bwa\,index$ <$contig$>
- for aligning ONT reads to genome:
  $bwa\,mem$ -t $\$num_threads$ -x ont2d -Y <$draft_genome$> <$reads$>
- for aligning PacBio reads to genome:
  $bwa\,mem$ -t $\$num_threads$ -x pacbio -Y <$draft_genome$> <$reads$>
- for global realignment method (3.2.3):
  $bwa\,mem$ -t $\$num_threads$ -x ont2d/pacbio -Y <$contig$> <$reads$>

2.2.3. SAM Format

SAM Format is a text format for storing sequence data in a series of tab delimited ASCII columns. SAM stands for Sequence Alignment/Map format. It consists of optional header section which is followed by an alignment section. The alignment section consists of alignment records which have 11 mandatory fields. These are ([9]):

1. *QNAME* - The query/read name.
2. *FLAG* - The records flag.
3. *RNAME* - The reference sequence name.
4. *POS* - 1-based position on the reference (leftmost mapping position).
5. *MAPQ* - The mapping quality.
6. *CIGAR* - The CIGAR string of the alignment.

7. *RNEXT* - The reference name of the mate/next read.

8. *PNEXT* - The position of the mate/next read

9. *TLEN* - The observed length of the template.

10. *SEQ* - The query/read sequence.

11. *QUAL* - The ASCII PHRED-encoded base qualities.

These information about alignment were necessary for contig extension, especially fields POS and CIGAR. Cigar string consists of interlaced numbers and characters (regex *|([0-9]+[MIDNSHPX=])+). Possible operations in cigar string are:

- alignment match (M),
- insertion to reference (I),
- deletion from reference (D),
- skipped region from reference (N),
- soft clipping (S),
- hard clipping (H),
- padding (P),
- sequence match (=)
- or sequence mismatch (X).

Numbers before the operations give information about how many consecutive times the operation appears at this point in alignment. Usage of this information will be described in chapter 3.

2.3. Graphs

A graph is a set of vertices which are connected by edges. Formally, a graph G is a tuple (V, E) where V is nonempty finite set of vertices and E is a set (possibly empty) of edges. An edge is defined as every binomial subset of set V. We differ between directed and undirected graphs. Undirected graphs are those in which for every edge $e_1 = (u, v)$ there is an edge $e_2 = (v, u) = e_1$. Given an edge $e_1 = (u, v)$ in a directed graph, the vertex u is its *source*, and v is its *sink*, and edge $e_2 = (v, u) = e_1$ does not have to exist. The edge $e = (u, v)$ is called *outedge* of vertex u and *inedge* of vertex v. *Path* in a directed graph from vertex u to vertex v is a sequence of vertices

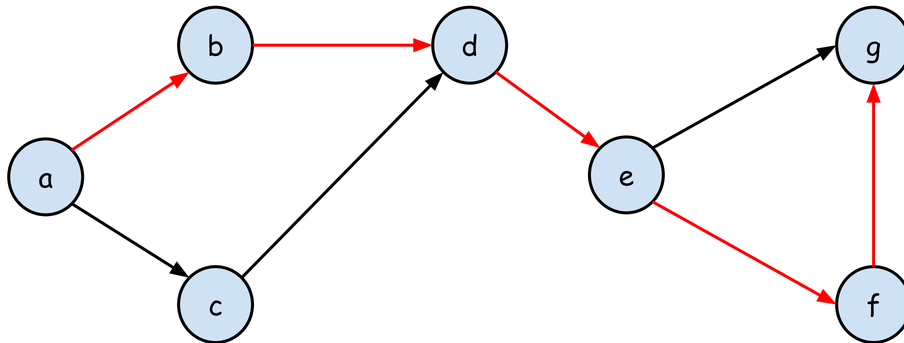

Figure 2.5: Directed acyclic graph. Length of a path colored in red $\langle a, b, d, e, f, g \rangle$ is 5.

$\langle v_0, v_1, \cdots, v_{n-1} \rangle$ where $v_0 = u, v_{n-1} = v$, and $(v_i, v_{i+1}) \in E$ for $i \in \{0, \cdots, n-2\}$. The sequence may consist of a single vertex. If there exists a path from u to v, v is said to be *reachable* from u and number of edges it traverses is the $length$ of a path. An example of a directed graph is given in figure 2.5, and one of the possible paths is colored in red. [10]

Computing reachable vertices (searching a graph) from other vertices is a fundamental operation. There are two methods for graph traversal and these are depth-first search (DFS) and breadth-first search (BFS). Both of this algorithms have linear time complexity, $O(|V| + |E|)$. For POA algorithm to work, nodes in a graph need to be topologically sorted (explained in section 2.3.2), and prerequisite for this is DFS, so I will explain this algorithm in detail in following section.

2.3.1. Depth-First Search

The depth-first search is a graph traversal method best suited for problems where we want to find any solution or to visit all nodes in a graph. The basic idea of the algorithm is to visit a vertex, then push all of the neighbor vertices to the stack. We repeat this procedure by finding next node so that we pop from stack, and then push all the nodes connected to that one onto the stack until all nodes are visited. The algorithm also memorizes which vertices are visited to ensure that every vertex is visited only once. Example of the algorithm execution is given in figure (2.6), and its pseudocode in Algorithm 2.

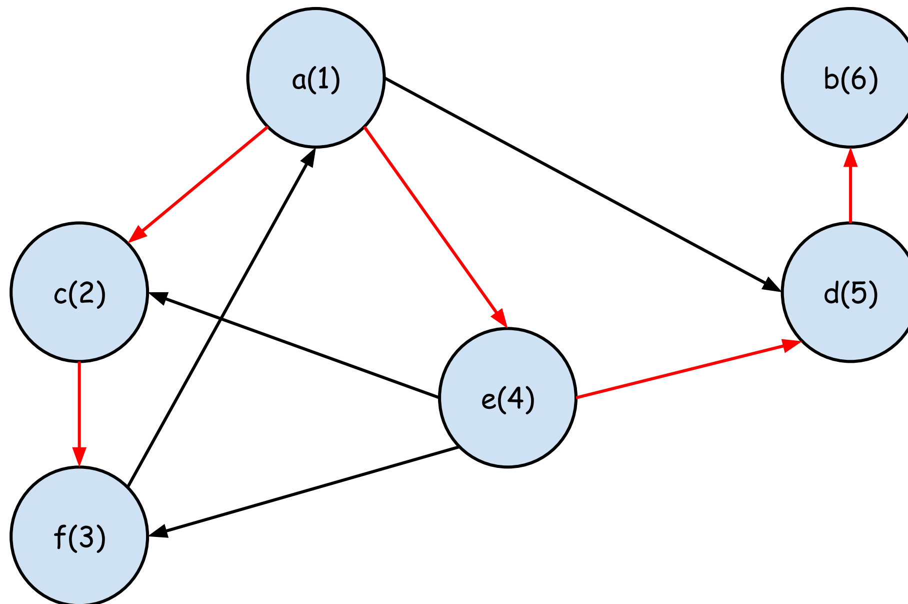

Figure 2.6: DFS traversal example. Edges used in traversal are colored red. Vertex order in traversal is given in brackets.

Algorithm 2 Depth-first search (vertex $start$)

1: set $visited$
2: stack s
3: $s.push(start)$
4: **while** s is not empty **do**
5: $top = s.top()$
6: $s.pop()$
7: **if** top not in visited **then**
8: check if this is the vertex we are searching for (or any other termination criteria)
9: $visited.insert(top)$
10: add all neighbors of top to stack
11: **end if**
12: **end while**

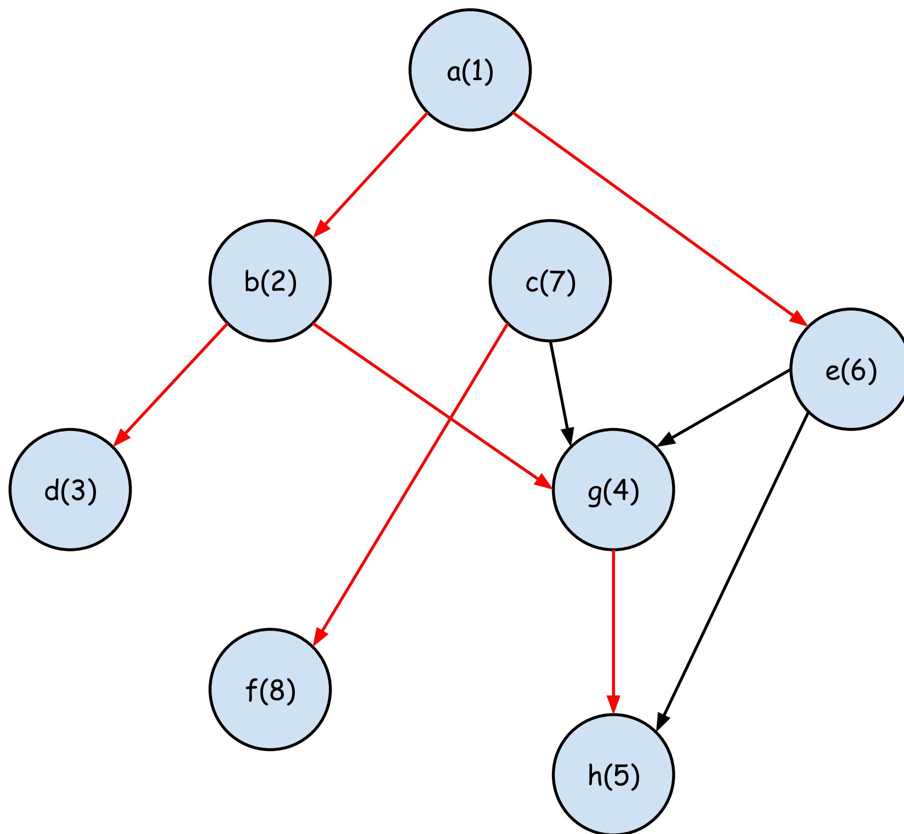

Figure 2.7: Topological sort example. Edges used in traversal are colored red. Vertex order in traversal is given in brackets.

Topological Order
c
f
a
e
b
g
h
d

Figure 2.8: Topological order of vertices in figure 2.7.

2.3.2. Topological Sort

Topological sort or topological ordering of a directed graph is a linear ordering of vertices such that every directed edge $e = (u, v)$ from vertex u to vertex v, u comes before v in ordering. Topological ordering of vertices is possible only if a graph is directed and acyclic (DAG). Any DAG has at least one topological ordering. [11] An example of topological ordering of a graph is given in figures 2.7 and 2.8.

One way to get a topological sort of a DAG is to run a modified depth-first search. When we finished visiting all neighbor vertices of some vertex, we push that vertex on the stack. We can get topological ordering of graph by popping vertices from the stack (Algorithm 3). Time complexity of this algorithm is linear, same as in DFS - $O(|V| + |E|)$.

Algorithm 3 Topological sort

```
1:  function DFS(vertex start)
2:      set started
3:      stack s
4:      s.push(start)
5:      while s is not empty do
6:          top = s.top()
7:          s.pop()
8:          if top in visited then
9:              continue
10:         end if
11:         if top in started then
12:             insert top in visited
13:             order.push(top)
14:             remove top from started
15:             continue
16:         end if
17:         insert top in started
18:         s.push(top)
19:         add all neighbors of top to stack
20:     end while
21: end function
22:
23: set visited
24: stack order
25: for vertex in vertices do
26:     if vertex not in visited then
27:         DFS(vertex, visited)
28:     end if
29: end for
```

3. Methods

In this chapter I will explain all methods used for contig extension. Firstly, I will explain how reads which are possible extensions of contig were found (3.1). Afterwards, I will explain how these reads were used for contig extension. Two methods were used for contig extension - a majority vote method with modifications (3.2) and a method using POA consensus algorithm for generating consensus sequence between possible extensions (3.3). Common property of all methods is that every contig was extended independently of one another. Finally, when contigs are extended they are merged into scaffolds or if possible into a genome as a whole which is described in section 3.4.

3.1. Finding possible extensions

Prerequisite for finding possible extensions for contigs was to align Oxford Nanopore or PacBio reads to draft genome. This was done by using the BWA as described in chapter 2.2.2. Information about this alignment was stored in .sam file. Next, records from .sam file were mapped to contigs in draft genome so reads aligned to specific contig, i.e. records about read alignment, could be easily accessed.

Every record mapped to contig needs to be checked if it is a possible extension. A record is suitable for extending contig if it is soft clipped and clipped part extends left of contig start, or if it is soft clipped and clipped part extends right of contig start. Additional criteria is introduced to check if alignment record, i.e. read, is feasible for contig extension and these are *inner* and *outer* margin. Reads whose alignment starts (left extension) or ends (right extension) within the *inner* margin is immediately suitable for extension. Reads whose alignment starts or ends within the *outer* margin are called *dropped* reads and are later used in global realignment method (3.2.3). An example of possible extensions is given in figure 3.1.

To define the start and the end read indices of extension (which is actually substring of a read), actual length of read used in alignment was calculated. Furthermore, actual length of contig part used in alignment was calculated. The contribution to

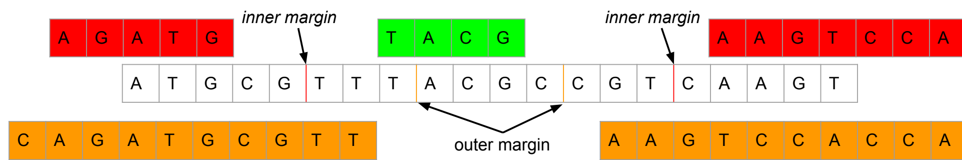

Figure 3.1: Sequence in white represents contig. Other sequences represent reads. Red reads are possible extensions of contig, orange reads are *dropped* reads and read in green is not suitable for contig extension.

read/sequence length is if operations in cigar string are alignment match, insertion to reference, soft clipping, sequence mismatch, or sequence match. The contribution to contig length is similar, but instead of insertion to reference, deletions from reference are counted.

Pseudocode of the described method is given in Algorithm 4.

Algorithm 4 Find possible extensions ($alignment_records$)

1: **for** $record$ in $alignment_records$ **do**
2:     $read_name = record.qName$
3:     $read = name_to_sequnce(read_name)$
4:     **if** $record$ is potential left extension **then**
5:         $left_soft_clipped_len = count(record.cigar[0])$
6:         $extension_length = left_soft_clipped_len - record.beginPos$
7:         **if** $record.beginPos < inner_margin$ **then**
8:             $extension = read.substring(0, extension_length)$
9:             $reverse(extension)$ // we are moving from right to left in extending
10:        **else**
11:            read is $dropped_read$
12:        **end if**
13:    **end if**

3.2. Majority vote

First idea that comes to mind is to extend contigs by using possible extension reads and extend contigs base by base. Next base added to contig is calculated by majority vote method amongst bases aligned to that position (section 3.2.1). Variations of this

Finding possible extensions continued

```
14:     if record is potential right extension then
15:         used_read_size = 0
16:         used_contig_size = 0
17:         for element e in record.cigar do
18:             if e contributes to sequence length then
19:                 used_read_size+ = count(e)
20:             end if
21:             if e contributes to contig length then
22:                 used_contig_size+ = count(e)
23:             end if
24:             right_soft_clipped_len = count(record.cigar[cigar_len − 1])
25:             used_read_size− = right_soft_clipped_len
26:             (margin = contig_length − (record.beginPos + used_contig_size)
27:             if margin > outer_margin then
28:                 continue
29:             end if
30:             if margin > inner_margin then
31:                 read is dropped_read
32:             else
33:                 extension_length = right_soft_clipped_len − margin
34:                 start = used_read_size + (right_soft_clipped_len −
    extension_length)
35:                 extension = read.substring(start, extension_length)
36:             end if
37:         end for
38:     end if
39: end for
```

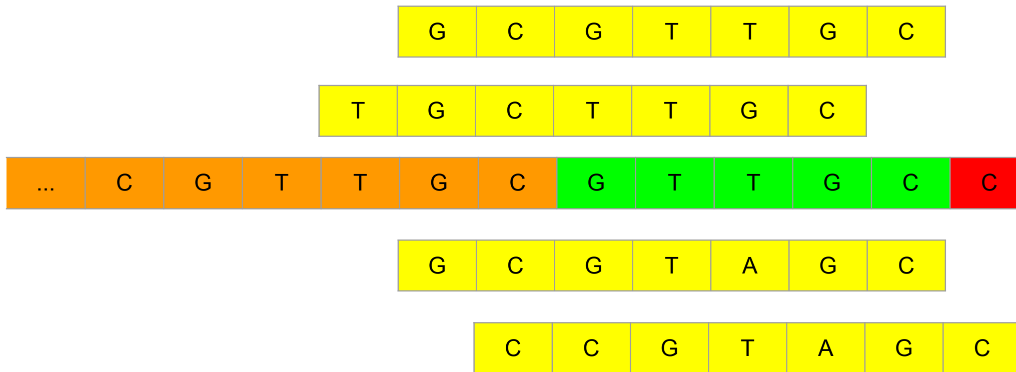

Figure 3.2: Example of contig extension using simple majority vote method. Possible extension reads are colored in yellow. Sequence colored in multiple colors is contig - orange represents original contig, bases in green are extended part of contig, and base in red is discarded because reads coverage is too low.

methods are local realignment of possible extension reads (section 3.2.2) and upgraded version of mentioned method that uses *bwa* for global realignment of dropped reads (section 3.2.3).

3.2.1. Simple extension

Simple extension using majority vote is pretty straightforward. Take all possible extension reads for contig, calculate next base for extension using majority vote and proceed until number of reads (coverage) is greater than some defined value (Figure 3.2).

3.2.2. Local realignment

Variation of simple extension majority vote method is to locally realign reads which bases at current extension position are not the same as base calculated with majority vote. Local realignment "looks" one move ahead, and checks if one of the alignment operations (match, mismatch, deletion, insertion) could correct read alignment to (extended) contig. "Looking ahead" is done by calculating the next base by majority vote, but only reads with correct base at current position are considered eligible for counting. Minimum coverage for the next base is determined as 0.6 * *MIN_COVERAGE* (line 22 in Algorithm 5) and is subject to changes if necessary for obtaining better results. Depending on local realignment operation do following:

– if operation is match move one base forward in sequence,
– if operation is mismatch move one base forward in sequence,
– if operation is insertion move two bases forward in sequence,
– if operation is deletion stay at current base in sequnce,
– otherwise extension read is marked as dropped.

Pseudocode of this method which finds contig extension is given in Algorithm 5. Every extension read memorizes for itself which is the current index for majority vote examination (variable $current_read_position$ in algorithm).

Algorithm 5 Majority vote with local realignment($extension_reads$)

```
1:  function COUNT_BASES(extension_reads)
2:      coverage = 0
3:      majority_vote_base = null
4:      for read in extension_reads do
5:          if read is not dropped then
6:              coverage+ = 1
7:              curr_base = read[current_read_position]
8:              update counter of curr_base
9:              update majority_vote_base if counter is new maximum
10:         end if
11:     end for
12:     return majority_vote_base, coverage
13: end function
14: while true do
15:     contig_extension = ””
16:     output_base, coverage = COUNT_BASES (extension_reads)
17:     if coverage < MIN_COVERAGE then
18:         break
19:     end if
20:     next_output, next_coverage = COUNT_BASES (extension_reads) with
21:     offset 1 // continued from line before
22:     if next_coverage < 0.6× MIN_COVERAGE then
23:         break
24:     end if
25:     contig_extension.push_back(output_base)
```

Majority vote with local realignment continued

26: **for** $read$ in $extension_reads$ **do**
27: $curr_base = read[current_read_position]$
28: $next_base = read[current_read_position + 1]$
29: **if** $curr_base == output_base$ **then**
30: operation is a match
31: **else if** $curr_base == next_output$ **then**
32: operation is a deletion
33: **else if** $next_base == next_output$ **then**
34: operation is a mismatch
35: **else if** $next_base == output_base$ **then**
36: operation is a deletion
37: **else**
38: drop read
39: **end if**
40: **end for**
41: **end while**
42: **return** $contig_extension$

3.2.3. Local realignment with global realignment of dropped reads

How can local realignment method work better? Answer is somehow intuitive. We can see that in the algorithm, all reads which are possible extensions, but are not precise at current position and cannot be locally realigned are marked as dropped reads. With that approach loss of data (coverage) is too fast, so these reads should be globally realigned to extended contig using *bwa*.

This process is iterative - in each step, after contig extension is found using local realignment method ($get_extension()$ function call in Algorithm 6), dropped reads are globally realigned and new possible extensions of already extended contig are found. If the coverage of left and right possible extensions are both below the minimum coverage, contig cannot be extended anymore and process is stopped. Pseudocode of this method is given in Algorithm 6. In the implementation other stopping criteria were added if maximum extension length of contig is specified.

3.3. POA

Another approach was tried out for contig extension which is based on multiple sequence alignment. All possible extension reads were mutually aligned and consensus sequence of alignment was taken as contig extension. The algorithm used for this was the Partial Order Alignment algorithm ([12]) which is based on using partial order graphs for multiple sequence alignment. After partial order multiple sequence alignment graph is constructed, consensus can be easily extracted as maximum weight path in graph ([13]). These methods are described in section 3.3.1, and how they were used for contig extension is described in section 3.3.2.

3.3.1. POA algorithm

As mentioned in chapter 2.2 we differ between local and global alignments. POA algorithm is based on the Smith-Waterman algorithm for local pairwise (between only two sequences) alignment. We know that two sequences can have different local alignment and which one is the best depends on match / mismatch / gap scoring scheme used. Example of this is given in figure 3.3 where different alignments of two sequences, "CGCGAAAAGGCC" and "CGCGTTTTGGCC", are presented.

This is not a problem for pairwise alignment, but for multiple sequence alignment these ambiguities begin to distort the eventual result. So how can we present alignment

Algorithm 6 Extend contig - global realignment method ($contig$)

1: **vector** $left_extensions, right_extensions$
2: $left_extensions, right_extensions = find_possible_extensions()$
3: $should_ext_left = should_ext_right =$ **true**
4: **while** $should_ext_left$ **or** $should_ext_right$ **do**
5: $left_extension = right_extension =$ ""
6: **if** $should_ext_left$ **then**
7: $left_extension = get_extension(left_extensions)$
8: $should_ext_left = \neg left_extension.isEmpty()$
9: **end if**
10: **if** $should_ext_right$ **then**
11: $right_extension = get_extension(right_extensions)$
12: $should_ext_right = \neg right_extension.isEmpty()$
13: **end if**
14: // construct extended contig
15: $contig = left_extension + contig + right_extension$
16: **if** there are not any dropped reads **then**
17: **break**
18: **end if**
19: align dropped reads to contig using *bwa*
20: $left_extensions, right_extensions = find_possible_extensions()$
21: **if** $left_extensions.size() <$ MIN_COVERAGE **or**
22: $right_extensions.size() <$ MIN_COVERAGE **then**
23: **break**
24: **end if**
25: **end while**
26: **return** $contig$

C G C G - - - - A A A A G G C C
C G C G T T T T - - - - G G C C

(a) Alignment 1.

C G C G A A A A - - - - G G C C
C G C G - - - - T T T T G G C C

(b) Alignment 2.

C G C G - A A - A A - - G G C C
C G C G T - - T - - T T G G C C

(c) Alignment 3.

C G C G - A - A - A - A G G C C
C G C G T - T - T - T - G G C C

(d) Alignment 4.

Figure 3.3: Example of different local alignments

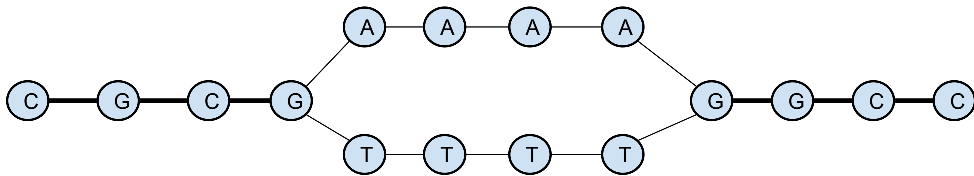

Figure 3.4: Example of partial order alignment graph between sequences CGCGAAAAGGCC and CGCGTTTTGGCC.

between sequences differently, and more clearly and unambiguously, additionally giving us a chance to extract all possible alignments from one view/layout? Layout structure that naturally comes to mind is graph. [14]

We can see in figure 3.4 that in partial order alignment graph one specific base can have multiple successors (e.g. base G before the fork) or predecessors (e.g. base G after the fork). Similarity to alignment strings is that there is a directional order of vertices in graph, each vertex has zero or more predecessors and successors and there is no repetition or doubling back - therefore, the graph is Directed Acyclic Graph (DAG). [14]

Next question that arises is how to align sequence to an already constructed graph. This actually is not very different from the Smith-Waterman algorithm described in section 2.2.1. In classical Smith-Waterman dynamic programming matrix base has

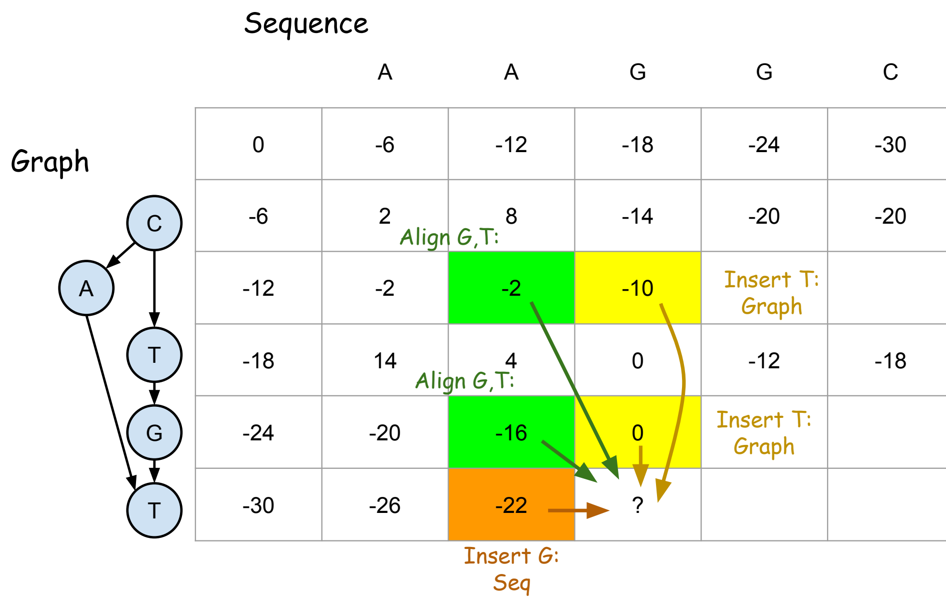

Figure 3.5: Sequence to graph alignment example ([14]). Align move means match or mismatch operation, vertical insertion moves are insertions from graph to sequence, and horizontal insertion moves are insertions from sequence to graph.

only one predecessor in sequence.

In sequence to graph alignment, base can have multiple predecessors in graph so there are more insert and align (match/mismatch) moves being considered (Figure 3.5). Prerequisite for this alignment to work is that vertices in the graph are topologically sorted (described in section 2.3.2), because dynamic programming methods require all previous states scores to be calculated before calculating the current state score.

Afterwards, when the best alignment between sequence and graph is calculated using dynamic programming method described earlier, this sequence has to be incorporated into a partial order graph. I will explain this on a simple example from [14]. Lets suppose that we have alignment as in in figure 3.6. When incorporating alignment into graph, we do not want to lose information about alignment operations. We should keep track about insertions (insert new vertex to graph), matches and mismatches (introduce new type of edge, which connects bases/vertices which are aligned to each other). So, for example in figure 3.6, we get a graph like one in figure 3.7. We can see that exactly those vertices which represent bases that are in mismatch are connected with dashed lines (edge).

According to [14], following steps are taken for incorporating aligned sequence to graph, and are presented in Algorithm 7. It is important to mention that weight of an

```
C  G  A  T  T  A  C  G
|  |  .  |  |  |  .
C  G  C  T  T  A  T  -
```

Figure 3.6: Calculated alignment between two sequences([14]). Vertical lines in second row represent match operations, and dots represent mismatches.

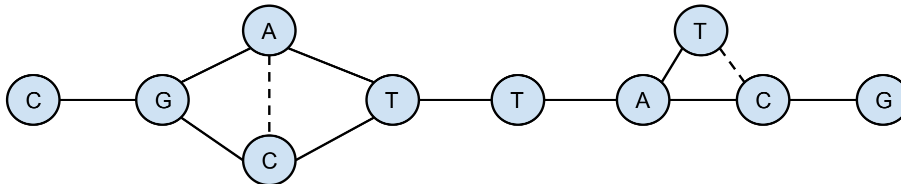

Figure 3.7: Incorporating sequence to graph example ([14]). Full lines are left to right directed edges and dashed lines connect vertices (bases) that are aligned to each other, but are mismatches.

edge is the number of sequences that include this edge.

Every sequence incorporated into a graph can be easily extracted because of its starting point. Afterwards, you only need to follow the edges labeled with this sequence through the graph. Once the sequence is incorporated, new topological sort of nodes is generated and another alignment can be performed. [14]

Generating a consensus sequence from a graph is actually a problem of finding maximum weight path between two vertices in DAG, and this can be solved in $O(|V|+|E|)$ time, which is linear in the size of the input. Firstly, we have to topologically sort the reverse graph, so all of the vertices are ordered in such a way that no node is visited before all of its children are visited. This can be done by topologically sorting the original graph, and then just reverse the vertex order. Next step is to label all the vertices with the weight of highest-weight path starting with that vertex. This is again a dynamic programming problem – first we set scores of all vertices to zero, and then visit them in previously mentioned order. With all the prerequisites satisified, we have two conditions for every vertex (scores of vertices are marked as $d(v)$).

- If vertex has no outgoing edges, weight of path starting at this vertex remanes zero.
- Otherwise, for each edge $e = (u, v)$ leaving the current vertex u, we compute value $d(v) + weight(u, v)$ and set $d(u)$ to maximum of these values.

Algorithm 7 Incorporate aligned sequence to graph

```
1:  Create new "starting point" start for this sequence in graph
2:  Set previous position prev to start
3:  for each sequence base b in calculated alignment do
4:      if b is not aligned to a vertex a in graph G(V, E) or
5:       (if it is but neither the aligned vertex a ∈ V nor
6:       any v ∈ V it is aligned to has the same base) then
7:          New vertex n is created with the base b,
8:            and is selected as current vertex curr
9:          n is aligned to the aligned vertex a if any
10:           and all of the "aligned-to" vertices are updated to align to n.
11:     else
12:         That vertex u ∈ V with the same base is selected as curr.
13:     end if
14:     if Edge e = (prev, curr) does not already exist then
15:         add new edge e = (prev, curr) to G
16:     end if
17:     add current sequence label to e
18:     prev = curr
19: end for
```

Finally, we only need to memorize which vertex has maximum score and follow edges from it to obtain maximum weight path in graph.

3.3.2. Extension using POA consensus

Using POA generated consesus to extend contigs is actually a very simple idea. Use *find possible extensions* method to get possible extension reads, create substring of defined length for each extension and pass these substrings to the POA algorithm for generating consensus of these sequences. Afterwards, just append the consensus sequences for left and right extensions to contig. Pseudocode of this method is given in Algorithm 8.

Algorithm 8 Extend contig - POA method ($contig$)

1: **vector** $left_extensions, right_extensions$
2: $left_extensions, right_extensions = find_possible_extensions()$
3: create substrings(trim extensions) of defined length from left and right extensions
4: $left_extension = poa_consensus(left_extensions)$
5: $right_extension = poa_consensus(right_extensions)$
6: $contig = left_extension + contig + right_extension$
7: **return** $contig$

3.4. Scaffolding

The intuitive way to merge contigs into scaffolds is to find overlaps between them if they exist, but this is not easily achievable because extensions have potentially high error rate and contigs sequences are very long. Therefore, anchors were created from contigs. Anchors are subsequences of user defined length at the start and the end of the contig. It is necessary that the anchors do not cover only previously found extension part of contig, but also a part of contig which precedes the extension, because it is assumed that this part is 100% correct.

Next, those anchors are aligned using *bwa* to each contig. Lets suppose we have contig 1 and contig 2 and anchor of contig 2 is aligned at the end or start of contig 1. These two contigs are merged. This is repeated until merging is not possible anymore. Result can be multiple scaffolds or genome as a whole.

4. Implementation

Two independent projects were developed in the scope of this master thesis. First one implements Partial Order Alignment algorithm as a library which provides programming interface for generating consensus, and it is publicly available at *https://github.com/mculinovic/cpppoa*. Second project is a collaborative project, which implements methods for contig extension and scaffolding described in chapter 3. It is implemented as a console application, and it is publicly available at *https://github.com/mculinovic/ONTscaffolder*.

4.1. General overview

Both projects were implemented in C++ programming language and the code was written according to Google C++ Style guide with some exceptions where those were unavoidable. Additionally, code is documented according to Doxygen conventions, so that documentation can be easily accessible both in pdf and html format. For automatic compilation of both debug and release versions and Doxygen documentation generation Makefiles were written. As version control tool, git was used, and git repositories were hosted at *http://github.com*. In *ONTscaffolder* project, scripts for preparing data and result analysis were written in Python and for automatic testing and program execution in bash (4.4). Both projects have detailed installation and usage instructions written in *readme* files.

To download the implemented POA library, *git clone* the repository and type *make* for compilation. After command is executed, the static library file *libcpppoa.a* will be generated and public header *poa.hpp* will be available in the *include/cpppoa* directory. To use *cpppoa* in your project include *poa.hpp* in your source code and provide it with a library file *libcpppoa.a* when linking the executable file. *CPPPOA* source code statistics obtained using *cloc* tool are presented in figure 4.1.

To install the *ONTscaffolder*, *git clone* the repository, locate yourself inside the *ONTscaffolder* folder, run command *git submodule update –init –recursive*, and type

```
      17 text files.
      17 unique files.
     165 files ignored.

http://cloc.sourceforge.net v 1.60  T=0.10 s (131.3 files/s, 13533.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                              5            112             89            504
C/C++ Header                     5            107            332            160
make                             3             11              0             25
-------------------------------------------------------------------------------
SUM:                            13            230            421            689
-------------------------------------------------------------------------------
```

Figure 4.1: CPPPOA source code statistics obtained using cloc.

```
      42 text files.
      42 unique files.
      51 files ignored.

http://cloc.sourceforge.net v 1.60  T=0.19 s (162.0 files/s, 16247.5 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                              9            347            125           1077
C/C++ Header                     9            219            195            398
Python                           6            111             33            293
Bourne Shell                     4             51             23            170
make                             3             18              1             48
-------------------------------------------------------------------------------
SUM:                            31            746            377           1986
-------------------------------------------------------------------------------
```

Figure 4.2: ONTscaffolder source code statistics obtained using cloc.

make afterwards. Running the *make* command without arguments will build the release version of the tool as the binary file *scaffolder* in *release* directory. To run the tool please use *run.sh* script because it will automatically delete temporary files and folders. If you prefer doing it manually or you want to inspect files created during program execution you can directly run *scaffolder*. *ONTscaffolder* source code statistics obtained using *cloc* tool are presented in figure 4.2.

Both projects should be compatible with most UNIX flavors and have been successfully tested on operating systems Max OS X 10.10.3. and Ubuntu 14.04 LTS. Requirements to build and run both projects are following:

– g++(4.8.2 or higher)

– GNU Make

– Burrows-Wheeler Aligner (0.7.12 or higher) - *ONTscaffolder* only

– Doxygen (optional)

4.2. External dependencies

The *ONTscaffolder* project depents directly on 2 libraries, one of which is *cpppoa*. Other dependency is SeqAn Library ([15]), which was used for easier input and output handling, especially reading and parsing alignments file in .sam format. Seqan is an open source C++ library of efficient algorithms and data structures for analysis of sequences with focus on biological data.

4.3. Code layout

The code is structured into multiple namespaces and classes. In the next sections, I will describe what specific files are used for.

4.3.1. CPPPOA

CPPPOA code structure is organized in multiple classes (*poa.cpp* dependencies can be seen in figure 4.3) as follows:

- *alignment.hpp / alignment.cpp* - class is used for calculating local alignment between sequence and graph using modified Smith-Waterman algorithm.
- *edge.hpp / edge.cpp* - class represents directed edge in Partial Order Graph.
- *node.hpp / node.cpp* - class represents vertex in Partial Order Graph.
- *graph.hpp / graph.cpp* - Partial Order Graph implementation, provides methods for incorporating sequence alignment into graph, and extracting consensus sequence from it.
- *poa.hpp / poa.cpp* - library programming interface and its implementation. Provides method for generating consensus sequence from vector of strings.

4.3.2. ONTscaffolder

ONTscaffolder code structure is organized in multiple classes and namespaces (*main.cpp* dependencies can be ssen in figure 4.4) as follows:

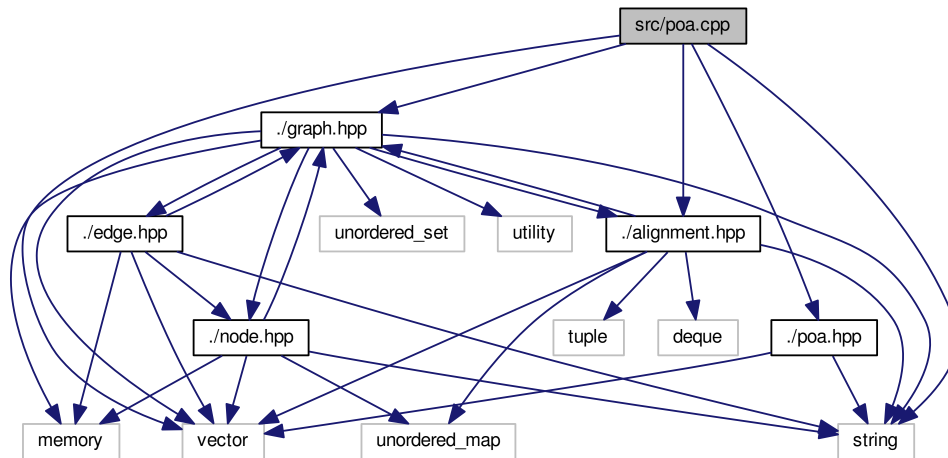

Figure 4.3: Public header implementation - *poa.cpp* - dependencies.

- *bases.h / bases.cpp* - namespace *bases* and class *BasesCounter*. Provides methods for calculating majority vote base and coverage at specific position in extension.
- *bwa.h / bwa.cpp* - namespace *aligner*. Provides wrapper functions for system calls to *bwa* tool.
- *connector.h / connector.cpp* - class *Connector* which is used for merging contigs into scaffolds.
- *contig.h / contig.cpp* - class *Contig* which represents contig sequence. Provides easy access to left and right extension of contig.
- *extension.h / extension.cpp* - class *Extension* which is used for representing extension read sequence during local realignment process. Provides functionality for positioning in sequence based on alignment operations.
- *scaffold.h / scaffold.cpp* - class *Scaffold* which consists of multiple *Contigs* and memorizes contributions of each contig into scaffold sequence.
- *scaffolder.h / scaffolder.cpp* - namespace *scaffolder* which implements method for finding possible extensions and all of the methods for contig extension.
- *utility.h / utility.cpp* - namespace *utility*; various utility functions (wrappers for reading specific file formats, shell commands execution, exception throwing etc.)

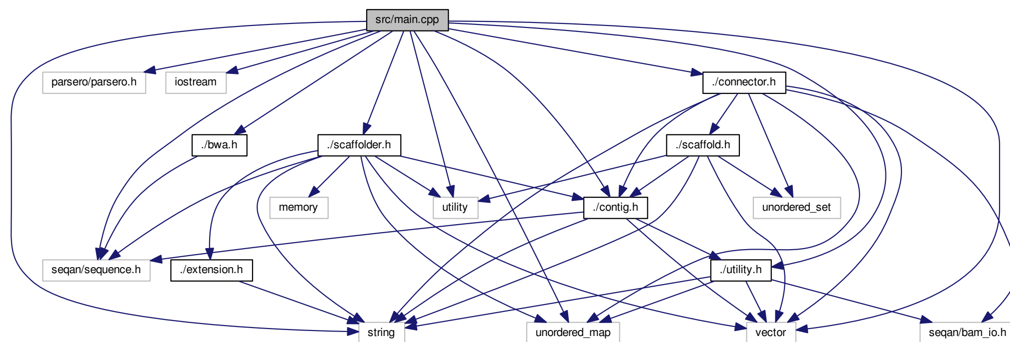

Figure 4.4: Scaffolder main program - *main.cpp* - dependencies.

- *main.cpp* - entry point and main program of project.

4.4. Scripts

Multiple scripts were implemented for various functions. Here is the list of available scripts:

1. Python scripts (for detailed usage instructions run script with –help option)
   - **genome2contigs** - cuts a reference genome into multiple contigs
   - **reverse_complement** - performs the reverse complement operation over sequences in FASTA file
   - **extension_analysis** - runs various statistics on contig extensions produced by the scaffolder

2. Bash scripts
   - **run.sh** - script used for running scaffolder. It will automatically detect the number of hardware threads supported by system and delete temporary files created
   - **run_tests.sh** - script used for comparing results of extension methods (POA and global realignment). It invokes extension_analysis script for statistics.

5. Results

5.1. Tools and methods for results evaluation

All measurements were conducted on the following hardware and software configuration:

1. **Hardware**
   - Architecture: x86_64
   - Number of CPUs: 2
   - Model name: Intel(R) Xeon(R) CPU E5645
   - Cores per CPU: 6
   - CPU GHz: 2.40
   - RAM: 96GB
2. **Software**
   - Ubuntu 14.04.2 LTS

Time measurements were performed by the tool *time*, which is available as a standard part of the *linux shell*. Testing was automated by using *run_tests.sh* script which enables the execution of both global realignment and POA consensus extension methods one after another on the same input. Results analysis was done by using *extension_analysis.py* script which was called from *run_tests.sh*. Simulated draft genomes were created from *Escherichia coli full genome* with usage of *genome2contigs.py* script.

5.2. Data

Two datasets of reads were used for contig extension. The first one is *Escherichia coli* resequencing dataset, which showcases PacBios extremely long-read, single molecule

data with a sample of the *E. coli* K12 MG1655 strain. Some of the key metrics are that mean mapped read length is 3549bp and coverage is 19.96x. This dataset can be downloaded from *https://github.com/PacificBiosciences/DevNet/wiki/E-coli-K12-MG1655-Resequencing*. The second dataset is Bacterial whole genome read data from the Oxford Nanopore Technologies MinION nanopore sequencer ([16]). Both High and Normal Quality Two-Directions (2D) reads from this dataset were used. For the simulation the draft genomes *Escherichia coli str. K-12 substr. MG1655, complete genome* was used. From this genome three different draft genomes were created:

- The first one has one gap of length 1000 starting from base 3000000, i.e. draft genome consists of 2 contigs.
- The second one has four gaps of length 1000 starting from bases 1000000, 2000000, 3000000 and 4000000 respetively, i.e. draft genome consists of 5 contigs.
- The third one has one gap of length 5000 starting from base 3000000, i.e. draft genome consists of 2 contigs.

For filling these gaps PacBio (section 5.3) and ONT (section 5.4) read datasets were used. For analysis of contig extensions correctness, they were aligned to reference genome using *bwa*. Tables with following columns were created from alignment files:

NAME - name of the contig extension, it consists of original contig name plus additional suffix *L* or *R* meaning left or right contig extension respectively

ID - identity of alignment - number of matches divided by (number of matches + number of mismatches)

MTCH - number of matches

MISM - number of mismatches

I - number of insertions

D - number of deletions

I + D - number of insertions plus number of deletions

|I - D| - absolute value of number of insertions minus number of deletions

LEN - length of extension

All results are discussed in section 5.5.

Table 5.1: Aligned extensions for dataset with one gap of length 1000bp. Original contig names are *gi|545778205|gb|U00096.3|0|* and *gi|545778205|gb|U00096.3|1|*. Extensions are obtained by POA consensus method on PacBio reads.

NAME	ID (%)	MTCH	MISM	I	D	I + D	|I - D|	LEN
gi|545778205|gb|U00096.3|0|L	99.90	1031	1	107	0	107	107	2028
gi|545778205|gb|U00096.3|0|L	99.62	796	2	89	1	90	88	887
gi|545778205|gb|U00096.3|0|R	99.90	976	0	100	1	101	99	1168
gi|545778205|gb|U00096.3|1|L	99.49	975	5	167	0	167	167	1660
gi|545778205|gb|U00096.3|1|L	84.02	610	85	93	31	124	62	788
gi|545778205|gb|U00096.3|1|R	99.90	1011	1	47	0	47	47	1062

Table 5.2: Aligned extensions for dataset with one gap of length 1000bp. Original contig names are *gi|545778205|gb|U00096.3|0|* and *gi|545778205|gb|U00096.3|1|*. Extensions are obtained by global realignment method on PacBio reads.

NAME	ID(%)	MTCH	MISM	I	D	I + D	|I - D|	LEN
gi|545778205|gb|U00096.3|0|L	98.74	1018	5	6	8	14	2	1029
gi|545778205|gb|U00096.3|0|R	99.39	976	2	7	4	11	3	986
gi|545778205|gb|U00096.3|1|L	98.31	349	3	2	3	5	1	355
gi|545778205|gb|U00096.3|1|R	99.60	994	3	7	1	8	6	1004

5.3. *E. coli* PacBio reads results

5.3.1. One gap of length 1000bp

Extending contigs using global realignment method lasted 1 minute and 56.729 seconds. Results of this method are shown in table 5.2. Extending contigs using POA consensus method lasted 1 minute and 19.839 seconds. Results of this method are shown in table 5.1.

5.3.2. Four gaps of length 1000bp

Extending contigs using global realignment method lasted 2 minutes and 9.031 seconds. Results of this method are shown in table 5.4. Extending contigs using POA consensus method lasted 4 minutes and 17.743 seconds. Results of this method are shown in table 5.3.

Table 5.3: Aligned extensions for dataset with four gaps of length 1000bp. Original contig names are *gi|545778205|gb|U00096.3|0|*, *gi|545778205|gb|U00096.3|1|*, *gi|545778205|gb|U00096.3|2|* and *gi|545778205|gb|U00096.3|3|*. Extensions are obtained by POA consensus method on PacBio reads.

NAME	ID(%)	MTCH	MISM	I	D	I + D	|I - D|	LEN
gi|545778205|gb|U00096.3|0|L	99.90	1031	1	107	0	107	107	2028
gi|545778205|gb|U00096.3|0|L	99.62	796	2	89	1	90	88	887
gi|545778205|gb|U00096.3|0|R	100.00	1022	0	113	0	113	113	1138
gi|545778205|gb|U00096.3|1|L	99.79	1443	0	322	3	325	319	1771
gi|545778205|gb|U00096.3|1|R	99.25	1060	3	104	5	109	99	1238
gi|545778205|gb|U00096.3|2|L	98.88	1062	8	110	4	114	106	1351
gi|545778205|gb|U00096.3|2|R	99.90	977	0	100	1	101	99	1169
gi|545778205|gb|U00096.3|3|L	99.49	974	5	165	0	165	165	1654
gi|545778205|gb|U00096.3|3|L	83.88	609	85	92	32	124	60	786
gi|545778205|gb|U00096.3|3|R	100.00	987	0	52	0	52	52	1111
gi|545778205|gb|U00096.3|4|L	99.11	1000	9	162	0	162	162	1352
gi|545778205|gb|U00096.3|4|L	98.55	68	0	1	1	2	0	69
gi|545778205|gb|U00096.3|4|R	99.90	1011	1	47	0	47	47	1062

Table 5.4: Aligned extensions for dataset with four gaps of length 1000bp. Original contig names are *gi|545778205|gb|U00096.3|0|*, *gi|545778205|gb|U00096.3|1|*, *gi|545778205|gb|U00096.3|2|* and *gi|545778205|gb|U00096.3|3|*. Extensions are obtained by global realignment method on PacBio reads.

NAME	ID(%)	MTCH	MISM	I	D	I + D	|I - D|	LEN
gi|545778205|gb|U00096.3|0|L	98.74	1018	5	6	8	14	2	1029
gi|545778205|gb|U00096.3|0|R	99.50	1002	0	8	5	13	3	1010
gi|545778205|gb|U00096.3|1|L	99.25	926	3	10	4	14	6	939
gi|545778205|gb|U00096.3|1|R	100.00	1006	0	3	0	3	3	1010
gi|545778205|gb|U00096.3|2|L	99.30	995	5	6	2	8	4	1006
gi|545778205|gb|U00096.3|2|R	99.39	976	2	8	4	12	4	987
gi|545778205|gb|U00096.3|3|L	98.31	348	3	2	3	5	1	354
gi|545778205|gb|U00096.3|3|R	99.34	1048	4	8	3	11	5	1060
gi|545778205|gb|U00096.3|4|L	99.61	1012	0	10	4	14	6	1025
gi|545778205|gb|U00096.3|4|R	99.60	994	3	7	1	8	6	1004

Table 5.5: Aligned extensions for dataset with one gap of length 5000bp. Original contig names are *gi|545778205|gb|U00096.3|0|* and *gi|545778205|gb|U00096.3|1|*. Extensions are obtained by POA consensus method on PacBio reads.

NAME	ID(%)	MTCH	MISM	I	D	I + D	|I - D|	LEN
gi|545778205|gb|U00096.3|0|L	95.24	3261	157	795	6	801	789	9378
gi|545778205|gb|U00096.3|0|L	93.57	2068	133	607	9	616	598	2808
gi|545778205|gb|U00096.3|0|L	95.19	1485	64	299	11	310	288	1848
gi|545778205|gb|U00096.3|0|L	89.53	1000	84	84	33	117	51	1168
gi|545778205|gb|U00096.3|0|R	99.33	4869	26	1089	7	1096	1082	5984
gi|545778205|gb|U00096.3|1|L	97.36	9364	224	1987	30	2017	1957	11973
gi|545778205|gb|U00096.3|1|R	99.47	3938	17	492	4	496	488	5471
gi|545778205|gb|U00096.3|1|R	87.72	736	89	184	14	198	170	1009

Table 5.6: Aligned extensions for dataset with one gap of length 5000bp. Original contig names are *gi|545778205|gb|U00096.3|0|* and *gi|545778205|gb|U00096.3|1|*. Extensions are obtained by global realignment method on PacBio reads.

NAME	ID(%)	MTCH	MISM	I	D	I + D	|I - D|	LEN
gi|545778205|gb|U00096.3|0|R	99.39	976	2	8	4	12	4	987
gi|545778205|gb|U00096.3|1|R	99.49	2345	8	27	4	31	23	2380

5.3.3. One gap of length 5000bp

Extending contigs using global realignment method lasted 2 minutes and 43.506 seconds. Results of this method are shown in table 5.6. Extending contigs using POA consensus method lasted 31 minutes and 27.064 seconds. Results of this method are shown in table 5.5.

5.4. *E. coli* Oxford Nanopore reads results

5.4.1. One gap of length 1000bp

Extending contigs using global realignment method lasted 1 minute and 20.954 seconds. Results of this method are shown in table 5.8. Extending contigs using POA consensus method lasted 2 minutes and 5.504 seconds. Results of this method are shown in table 5.7.

Table 5.7: Aligned extensions for dataset with one gap of length 1000bp. Original contig names are *gi|545778205|gb|U00096.3|0|* and *gi|545778205|gb|U00096.3|1|*. Extensions are obtained by POA consensus method on ONT reads.

NAME	ID(%)	MTCH	MISM	I	D	I + D	|I - D|	LEN
gi|545778205|gb|U00096.3|0|L	96.28	1035	24	226	16	242	210	1304
gi|545778205|gb|U00096.3|0|R	98.14	1057	11	250	9	259	241	1323
gi|545778205|gb|U00096.3|1|L	94.56	1181	46	260	22	282	238	1496
gi|545778205|gb|U00096.3|1|R	95.66	1057	30	234	18	252	216	1322

Table 5.8: Aligned extensions for dataset with one gap of length 1000bp. Original contig names are *gi|545778205|gb|U00096.3|0|* and *gi|545778205|gb|U00096.3|1|*. Extensions are obtained by global realignment method on ONT reads.

NAME	ID(%)	MTCH	MISM	I	D	I + D	|I - D|	LEN
gi|545778205|gb|U00096.3|0|L	86.32	82	5	2	8	10	6	90
gi|545778205|gb|U00096.3|0|R	90.37	197	11	5	10	15	5	213
gi|545778205|gb|U00096.3|1|R	92.53	223	7	6	11	17	5	236

5.4.2. Four gaps of length 1000bp

Extending contigs using global realignment method lasted 1 minute and 56.915 seconds. Results of this method are shown in table 5.10. Extending contigs using POA consensus method lasted 3 minutes and 46.526 seconds. Results of this method are shown in table 5.9.

5.5. Discussion

Before the result analysis, I just want to highlight the situation where one extension is aligned multiple times to the referent genome. This is happening because of relatively small size of extensions and therefore only the best (primary) alignment will be considered in results comparison, although they are all written in result tables.

I will start by comparing results for the simplest problem - one gap of length 1000. When looking at tables 5.1 and 5.2, the first thing which arises is that length of POA extensions are longer than gap size. This is because of how this algorithm works - takes sequence of specific length, but does not put boundary on consensus sequence generated from them. Next, it can be seen that POA method has greater identity score than global realign method, but number of insertions and deletions is greater by an

Table 5.9: Aligned extensions for dataset with four gaps of length 1000bp. Original contig names are *gi|545778205|gb|U00096.3|0|*, *gi|545778205|gb|U00096.3|1|*, *gi|545778205|gb|U00096.3|2|* and *gi|545778205|gb|U00096.3|3|*. Extensions are obtained by POA consensus method on ONT reads.

NAME	ID(%)	MTCH	MISM	I	D	I + D	|I - D|	LEN
gi|545778205|gb|U00096.3|0|L	96.28	1035	24	226	16	242	210	1304
gi|545778205|gb|U00096.3|0|R	93.27	1067	57	203	20	223	183	1356
gi|545778205|gb|U00096.3|1|L	95.39	1014	44	283	5	288	278	1344
gi|545778205|gb|U00096.3|1|R	97.01	1039	23	208	9	217	199	1327
gi|545778205|gb|U00096.3|2|L	97.58	1009	16	248	9	257	239	1300
gi|545778205|gb|U00096.3|2|R	97.96	1055	15	250	7	257	243	1325
gi|545778205|gb|U00096.3|3|L	94.72	1184	44	263	22	285	241	1500
gi|545778205|gb|U00096.3|3|R	95.14	1038	36	179	17	196	162	1253
gi|545778205|gb|U00096.3|4|L	91.47	1126	63	215	42	257	173	1437
gi|545778205|gb|U00096.3|4|R	95.66	1057	30	234	18	252	216	1322

Table 5.10: Aligned extensions for dataset with four gaps of length 1000bp. Original contig names are *gi|545778205|gb|U00096.3|0|*, *gi|545778205|gb|U00096.3|1|*, *gi|545778205|gb|U00096.3|2|* and *gi|545778205|gb|U00096.3|3|*. Extensions are obtained by global realignment method on ONT reads.

NAME	ID(%)	MTCH	MISM	I	D	I + D	|I - D|	LEN
gi|545778205|gb|U00096.3|0|L	86.32	82	5	2	8	10	6	90
gi|545778205|gb|U00096.3|1|L	98.63	72	0	6	1	7	5	79
gi|545778205|gb|U00096.3|1|R	90.06	299	15	2	18	20	16	316
gi|545778205|gb|U00096.3|2|R	90.41	198	11	5	10	15	5	214
gi|545778205|gb|U00096.3|3|R	94.21	114	5	0	2	2	2	123
gi|545778205|gb|U00096.3|4|L	91.38	53	2	0	3	3	3	60
gi|545778205|gb|U00096.3|4|R	92.53	223	7	6	11	17	5	236

order of magnitude. Shortage of global realign method is that it highly depends on dataset coverage. Outcome of this problem is that some contigs cannot be extended in length approximate to gap size, because the coverage is simply too low. From comparing results obtained from PacBio reads to ONT reads in tables 5.7 and 5.8, multiple conclusions can be made. The first one is that the identity score is obviously lower when obtained with ONT reads and this is expected because of higher error rate in these reads. The second thing to be seen is that global realigned did not succeed in extending contigs to gap length, because of the same reason as with PacBio reads - low read coverage at this position in contig. This stands out here because in general coverage of ONT reads dataset is lower than PacBio reads dataset. Similar conclusions can be made from observing results from tables 5.3, 5.4, 5.9 and 5.10.

When observing results for problem with one gap of length 5000, it can be seen that POA method generates a lot of alignments because of great number of insertions which can "fix" generated extension, but a good thing is that length of extensions is scaled to gap size as was expected from this method. Contrary, global realign method could not extend to this length for the same reasons as before - low coverage and high error rate. Even more so, some extensions could not even be aligned to reference genome (see table 5.6). What is promising in this method is that number of insertions, deletions and mismatches remains very small in comparison to POA method. Because of these results obtained from PacBio reads, I did not run algorithms on ONT reads for this gap size, because it is expected that because of even lower coverage method results would be similar as in previous examples.

6. Conclusion

Scaffolding using long error-prone reads is a very promising method for completion of draft genomes. In this master thesis two methods were proposed for filling gaps between contigs in draft genomes. Both methods are based on extending contigs on their left and right ends. Firstly, reads which are possible contig extensions are found. The first method uses these reads so that it "predicts" alignment operation one move ahead and combines this approach with global realignment of reads which were not correct at a specific position in the extension. The second method uses possible extension reads and creates from them a consensus sequence using the POA algorithm. After contigs were extended, they were, if possible, merged into scaffolds or even a whole genome. Test results are promising, especially the global realignment method. This method depends highly on reads coverage and reads error-rate, but Oxford Nanopore Techonologies are showing encouraging progress in this specific area. The POA method depends on these same things but is lacking the ending point of extension where coverage becomes too low, therefore many insertions and deletions appear when aligning consensus sequence to reference genome.

For accomplishing better results, few methods can be tried out. For example, using other tool instead of *bwa* for alignment. If this tool is more sensitive and accurate, only correct possible extensions of contig would be found and error probability would decrease. Next, there are lot of parameters in the implemented algorithm - some of them are inner and outer margin, minimum coverage for extension, anchor size used for contig merging etc. Refinement of these parameters, various testing and adjustment of these parameters would certainly help to obtain better results. Additionally, with all these proposals it is also possible to restart the program on extended contigs obtained from the last program execution and repeat that process iteratively. This could solve the problem on lowering coverage in global realignment when gaps are too long.

Bibliography

[1] E. Britannica, "bioinformatics," accessed May 26, 2015. [Online]. Available: http://www.britannica.com/EBchecked/topic/1334661/bioinformatics

[2] M. Šikić and M. Domazet-Lošo, *Bioinformatika - Skripta*, 2013.

[3] O. N. Technologies, "Introduction to nanopore sensing," accessed May 29, 2015. [Online]. Available: https://nanoporetech.com/science-technology/introduction-to-nanopore-sensing/introduction-to-nanopore-sensing

[4] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 8, no. 10, pp. 707–710, February 1966.

[5] S. Needleman and C. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of molecular biology*, vol. 3, no. 48, pp. 443–453, March 1970. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/5420325

[6] T. Smith and M. Waterman, "Identification of common molecular subsequences," *Journal of Molecular Biology*, vol. 1, no. 147, pp. 195–197, March 1981. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/7265238

[7] L. H. and D. R., "Fast and accurate long-read alignment with burrows-wheeler transform," *Bioinformatics*, 2010, [PMID: 20080505].

[8] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and . G. P. D. P. Subgroup, "The sequence alignment/map format and samtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, 2009. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/25/16/2078.abstract

[9] T. S. F. S. W. Group, *Sequence Alignment/Map Format Specification*, May 2015,

accessed June 7, 2015. [Online]. Available: http://samtools.github.io/hts-specs/SAMv1.pdf

[10] A. Aziz, A. Prakash, and T.-H. Lee, *Elements of programming interviews: The insider's guide*, October 2012.

[11] Wikipedia, "Topological sorting - wikipedia, the free encyclopedia," accessed June 9, 2015. [Online]. Available: http://en.wikipedia.org/wiki/Topological_sorting

[12] C. Lee, C. Grasso, and M. F. Sharlow, "Multiple sequence alignment using partial order graphs," *Bioinformatics*, vol. 18, no. 3, pp. 452–464, 2002. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/18/3/452.abstract

[13] C. Lee, "Generating consensus sequences from partial order multiple sequence alignment graphs," *Bioinformatics*, vol. 19, no. 8, pp. 999–1008, 2003. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/19/8/999.abstract

[14] J. Dursi, "Understanding partial order alignment for multiple sequence alignment," May 2015, online; accessed June 16, 2015. [Online]. Available: http://simpsonlab.github.io/2015/05/01/understanding-poa/

[15] A. Döring, D. Weese, T. Rausch, and K. Reinert, "SeqAn An efficient, generic C++ library for sequence analysis," *BMC Bioinformatics*, vol. 9, no. 11, 2008.

[16] J. L. N. Quick, "Bacterial whole-genome read data from the oxford nanopore technologies minion™ nanopore sequencer." 2014. [Online]. Available: http://dx.doi.org/10.5524/100102

Scaffolding using long error-prone reads

Abstract

Scaffolding using long error-prone reads is very promising method for completion of draft genomes. Two methods were proposed for extending contigs in draft genomes. First one is based on local and global realignment of possible extension reads and second one uses POA algorithm for generating consensus sequence from these reads. Program was tested on two different types of long reads, PacBio and Oxford Nanopore Technologies. Project is implemented in C++ programming language.

Keywords: Scaffolding, local and global realignment, POA, PacBio, Oxford Nanopore

Popravljanje postojećih genoma koristeći dugačka očitanja s velikom greškom

Sažetak

Popravljanje postojećih genoma koristeći dugačka očitanja s velikom greškom obećavajući je pristup za popunjavanje rupa u nedovršenim genomima. Dvije metode predložene su za produljivanje kontiga. Prva se temelji na lokalnom i globalnom poravnanju očitanja koja su moguća produljenja kontiga, a druga koristi POA algoritam za generiranje konsenzus sekvence iz tih očitanja. Rad metoda testiran je na dva različita skupa podataka dugačkih očitanja - PacBio i Oxford Nanopore Technologies očitanja. Projekt je implementiran u programskom jeziku C++.

Ključne riječi: Skafold, lokalno i globalno poravnanje, POA, PacBio, Oxford Nanopore