UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER's THESIS No. 1196

EAGLER - Eliminating Assembly Gaps by Long Extending Reads

Luka Šterbić

Zagreb, December 2015

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

Zagreb, 14 October 2015

MASTER THESIS ASSIGNMENT No. 1196

Student: **Luka Šterbić (0036458744)**
Study: Computing
Profile: Computer Science

Title: **EAGLER - Eliminating Assembly Gaps by Long Read Extension**

Description:

The next generation sequencing methods that produce short and accurate reads dominates in genomics these days. However, using only these reads makes it difficult to correctly assemble whole genomes due to long repeating regions. Therefore many genomes remain unfinished and are divided in several disjoint components. Although a new generation of sequencers produces longer reads, these are usually error-prone making it necessary to find a solution that will correct them. The short accurate reads produced by the next generation sequencing methods are usually used for this purpose. However, some recent works show that it is possible to achieve the same goal using long-error prone reads exclusively. The goal of this work is to develop a tool that would correct existing genome drafts by connecting disjoint regions using only long reads. The solution should work with both PacBio and Oxford Nanopore reads. Synthetic datasets of varying complexity are to be constructed for extensive implementation testing. It is necessary to conduct performance testing on a real dataset as well. The solution should be appropriated for parallel architectures and implemented in C++. The code is to be documented using comments and should follow the Google C++ Style Guide when possible. The complete application should be hosted on Github under an OSI-approved licence.

Issue date: 16 October 2015
Submission date: 14 December 2015

Mentor:

Associate Professor Mile Šikić, PhD

Committee Chair:

Full Professor Siniša Srbljić, PhD

Committee Secretary:

Assistant Professor Tomislav Hrkać, PhD

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

Zagreb, 14. listopada 2015.

DIPLOMSKI ZADATAK br. 1196

Pristupnik: **Luka Šterbić (0036458744)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **EAGLER - Popravljanje već sastavljenih genoma koristeći dugačka očitanja**

Opis zadatka:

Metode sekvenciranja druge generacije koje proizvode kratka i točna očitanja trenutno dominiraju u genomici. Međutim, koristeći isključivo ta očitanja teško je ispravno sastaviti cijele genome uslijed dugačkih ponavljajućih regija, zbog čega mnogi genomi još uvijek nisu završeni te su podijeljeni u nekoliko odvojenih regija. Iako nova generacija uređaja za sekvenciranje proizvodi dugačka očitanja, ta očitanja imaju veliku pogrešku. Zato je nužno pronaći rješenje za njihovo popravljanje, za što se obično koriste kratka točna očitanja dobivena uređajima druge generacije. Međutim, u nekoliko novijih radova pokazano je da je moguće ostvariti isti cilj koristeći isključivo dugačka očitanja koja imaju veliku pogrešku. Cilj ovoga rada je razvoj alata koji će moći popraviti trenutno sastavljene genome spajanjem odvojenih regija, a za to će koristiti isključivo dugačka očitanja. Razvijeno rješenje treba podržavati očitanja pročitana PacBio i Oxford Nanopore uređajima za sekvenciranje. Potrebno je proizvesti nekoliko sintetskih testnih skupova podataka različite kompleksnosti i provesti iscrpno testiranje. Isto tako potrebno je testirati performanse i na stvarnim podacima. Rješenje mora biti prilagođeno paralelnoj arhitekturi i napisano u jeziku C++ . Programski kod je potrebno komentirati i pri pisanju pratiti stil opisan u Googleovom C++ vodiču. Kompletnu aplikaciju postaviti na GIT pod jednom od OSI-odobrenih licenci.

Zadatak uručen pristupniku: 16. listopada 2015.
Rok za predaju rada: 14. prosinca 2015.

Mentor:

Izv. prof. dr. sc. Mile Šikić

Predsjednik odbora za diplomski rad profila:

Prof. dr. sc. Siniša Srbljić

Djelovođa:

Doc. dr. sc. Tomislav Hrkać

CONTENTS

LIST OF FIGURES

List of Tables

1. Introduction

Next Generation Sequencing (NGS) has dominated the bioinformatics scene for the last couple of years and is rapidly becoming a cornerstone of clinical medicine. NGS methods are based on high throughput methods that produce short and very accurate reads. Reads are subsequently assembled into contigs that, on their part, have low error rates.

The limiting factor that has slowed down wider adoption of NGS technologies is the inability to reconstruct genomic regions with high repeat rates since a single read can span only up to 300 base pairs. This situation has led to the creation of countless highly accurate genome assemblies that are not yet finished. The sequence of these assemblies is usually close to error free but a genome that should be unified ends up divided in several disjoint components called contigs.

Current gap closure methods based on mate pair reads have a very heavy laboratory footprint and can take several months to complete. While such results are vital for scientific research, they are not relevant for clinical diagnosis because of the prolonged time frame.

There have been many attempts to address the gap closing issue, either with post-assembly tools or by de-novo assembly with longer reads. The scaffolder presented in this thesis is a post-assembly tool that aims to close gaps in unfinished NGS assemblies by using long reads. The input taken by the EAGLER scaffolder is a draft genome created by any modern assembler, such as SGA[1], ABySS[2] or SPAdes[3], and a set of long reads.

Currently, the state-of-the-art long reads sequencer is the Pacific Bioscineces' PacBio RS II[4, 5], while the latest addition to this field of science is the MinION MkI[6] sequencing device designed by Oxford Nanopore Technologies[7].

The MinION is a plug-and-play device slightly larger than a common USB memory that produces real time data. DNA molecules pass through the nanopore membranes of the device and generate a specific electric current. By correlating current wave forms to nucelotide K-mers using a base-calling software, the MinION reconstructs

the sequence of a molecule. The MinION is still an experimental piece of technology and has been commercially available only since May 2015.

Figure 1.1: MinION MkI device illustration[8].

Unlike short reads, long reads have an error rate anywhere from 15%, PacBio reads, up to 35%, 1D Nanopore reads. Since virtually all tools designed for short reads use exact matching algorithms they are unusable on error prone long reads. This fact has led to a recent boom in new algorithms and tools specifically created for long reads of either type, but a clear community accepted standard set of tools has yet to emerge.

A recent paper by Jain et al.[9] has given significant insights into the performance of the MinION sequencer. Maximum likelihood estimated rates for insertion, deletion and substitution have been calculated by using the expectation maximization algorithm. High quality 2D reads presented an average identity of 85% with error rates of 4.9%, 7.8% and 5.1% for insertion, deletion and substitution respectively.

Long reads technologies have already been proven useful in resolving genomic structures that are not detectable by short reads only assemblies. Ashton et al. in their paper[10] have given a concrete example of how Oxford Nanopore reads sequenced with a MinION device can identify genomic structures undetected by Illumina only assemblies. They have successfully resolved the position and structure of a bacterial antibiotic resistance island in the Salmonella Typhi H58 strain genome.

Several groups have developed hybrid assemblers that can take long reads as input, in addition to the standard short reads dataset. Antipov et al. recently released hybridSPAdes[11], a tool in the SPAdes[3] genome assembly framework. SPAdes constructs a de Bruijn graph from the input short reads and transforms it into an assembly graph. The assembly graph is a filtered and simplified instance of the previously constructed de Bruijn graph. Once the assembly graph is constructed, long reads are

mapped to the graph and used to resolve gaps in the assembly and ambiguous repeating regions. Gaps are closed by calculating the consensus of all the long reads that span a specific gap. Repeats are amended by using the exSPAnder[12] module of the SPAdes framework in a similar way as done with paired-end and mate-pair libraries.

One of the first publicly available scaffolder for Oxford Nanopore reads was published in August 2015 by Warren et. al. The LINKS[13] scaffolder proposed in the paper leverages the fact that Nanopore reads tend to have sequential errors and hence, sequential correct bases. K-mers of user-defined length are extracted from the given long reads over a sliding window. Two contigs are allowed to create a scaffold if sufficient K-mers derived from one read uniquely map to both contigs.

All the previously mentioned tools close gaps by having a single long read span the whole length of the void in the assembly. The extension-based approach introduced with the EAGLER scaffolder does not have this limitation. The EAGLER scaffolder can use any long read that maps to one edge a contig to iteratively extends that contig. Contig extensions are computed as the consensus of all long reads spanning that region. Once two contigs begin to overlap each others extensions they are merged together into a scaffold.

2. Extension Based Scaffolding

Scaffolding is in some regards very similar to sequence assembly. They both share the same goal of reducing the input dataset to a set of significant, disjoint and unique elements.

The input for a Next Generation Sequence assembler is a set of short reads, usually the output of an Illumina sequencing machine. The assembler seeks to combine the reads in meaningful elements, contigs, that span different areas of a genome. The sequence of a single contig can be the result of the combination of tens of thousands of reads each consisting of no more than 300 base pairs.

A scaffolder takes as input the contigs generated by an NGS assembler and an additional source of information. This secondary source can be either of:

- Illumina mate pair reads [14]
- Pacific Biosciences long reads [4]
- Oxford Nanopore long reads [7]

The goal of a scaffolder is to reduce the number of contigs by merging them together using the secondary information source to guide the process. The ideal output is a dataset that contains as many scaffolds as there are chromosome-like structures in the analyzed genome.

The most obvious difference between the two described tasks is the starting point of the process. The sequence assembly starts from a blank sheet of paper and has no initial reference point, hence the commonly used term de-novo assembly. Scaffolding on the other hand, has a very clear starting position since each contig can be considered as a reference point from which to start the process.

The extension based scaffolding concept presented in this thesis leverages exactly this notion. Since the input contigs are considered disjoint parts of the reference sequence, appending one single base to each side of a contig gets the scaffolder closer to its goal.

This single-base extension process is repeated until the originally disjoint contigs

begin to overlap. Once two extended contigs are overlapped with a sufficient level of significance they can be safely joined together in a single scaffold. The scaffolder iteratively tries to extend and join contigs until all contigs are joined in a single scaffold or there is not sufficient data to further extend the remaining contigs.

The pseudocode in Algorithm 1 gives a high level view of the main procedure for the EAGLER scaffolder. The algorithm takes as inputs the contigs from a draft genome and a set of long reads, either PacBio long reads or Oxford Nanopore 2D reads.

Algorithm 1 High level scaffolding algorithm (draft_genome, long_reads)

```
1:  extended_contigs ← [ ]
2:  sam_file ← align(draft_genome, long_reads)
3:  for contig in draft_genome do
4:      alignments ← fetch_alignments(sam_file, contig.id)
5:      extensions ← compute_extensions(alignments)
6:      left ← consensus(extensions.left)
7:      right ← consensus(extensions.right)
8:      extended_contigs.append(left + contig.seq + right)
9:  end for
10: scaffolds ← join_contigs(extended_contigs)
11: return scaffolds
```

A pairwise alignment between all contigs and reads is performed at the beginning of the algorithm as shown on line 2. Once the alignments are computed, each contig is processed independently.

All relevant alignments for a specific contig are extracted from the SAM file outputted by a sequence aligner. The overhanging part of each aligned read is classified either as a left or right extension as shown in Figure 2.1. The exact algorithm to calculate an extension given an alignment record will be presented in section 2.1.

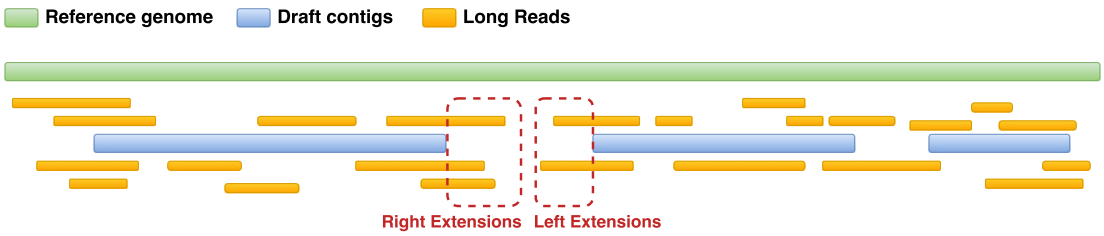

Figure 2.1: Representation of sequence types involved in the scaffolding process.

The consensus algorithm called on lines 6 and 7 takes as input a list of extensions and computes the consensus sequence. The different consensus algorithms supported by the scaffolder will be discussed later in this chapter and in the next one.

The original sequence of the contig is prefixed and suffixed by the generated consensus sequences to create an extended contig. After all contigs have been processed and their extended sequences have been collected to the *extended_contigs* list, the merging procedure is called. The algorithm employed by the *join_contigs* function to create the final scaffolds will be detailed in chapter 4.

2.1. Extracting Contig Extensions

This section presents the algorithm employed to extract the contig extensions given an alignment record. The algorithm takes as input a structure representing an alignment record from a SAM file and the contig that the read mapped to. The output consists of the left and right extending sequences, if they exist for the examined contig and alignment record pair.

Aligners tend to map overhanging sequences close to the end of the contigs, but often fail to align correctly the last few bases. This happens either as an imprecision of the alignment software or because of the presence of a mismatch or indel close to the ends of the contig. Both cases end up soft clipping the aligned read earlier than necessary.

Aligners describe an alignment using the CIGAR format. This format can be viewed as the sequence of operations an aligner has taken to align a query sequence to the given reference, starting from the beginning of the query. A CIGAR string is represented as a list of pairs where the first element is the number of bases that the second element, the operation, is applied to. Table 2.2 presents all CIGAR operations and the correspondent description.

Figure 2.2 shows an example of such behavior. An aligner can opt to soft clip the first 3 or 6 bases of the read, Cigar string 1 and 2 respectively. The margin concept is introduced here to define how far can the beginning of an alignment be to still consider it to extract an extension sequence.

The EAGLER scaffolder defines two types of margins: inner and outer. The first type, the inner margin, defines the maximum distance of the first match in the alignment from its closer edge of the contig. Reads that align within the inner margin are given the opportunity to immediately output an extension. Reads that align between the inner and outer margin are flagged as dropped and will not be used to output an

Position:				0	1	2	3	4	5	6	7	8	9	10	11	12
Contig:				A	T	C	A	A	C	C	T	A	A	G	A	G
Aligned read:	T	C	G	A	T	T	A	A	C	C	T	A	-	G	A	
Extension:	T	C	G													

Cigar string 1: 6S 6M 1I 2M

Cigar string 2: 3S 9M 1I 2M

Figure 2.2: Alignment margin issue example.

extension immediately, but will be reconsidered further down the scaffolding pipeline. More details on the usage of dropped reads will follow in chapter 3. If a read aligns after the outer margin, it is discarded completely and will not be considered for contig extension at any subsequent step in the pipeline.

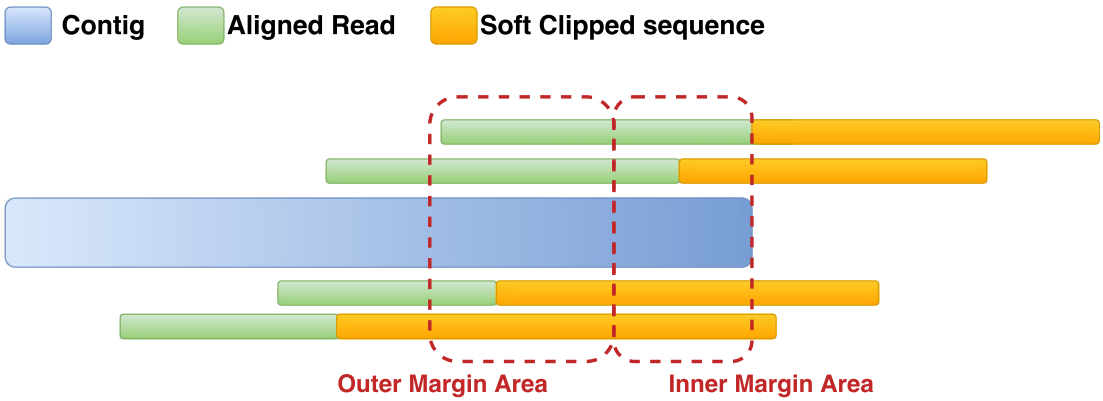

Figure 2.3: Inner and outer margins for a contig in the extension process.

Figure 2.3 illustrates the margin concept for the right end of a contig. The reads above the contig aligned within the inner margin and they will construct an extension. The first read below the contig has its last match between the inner and outer margin, hence it will be dropped and reconsidered for realignment in future steps of the pipeline. The last read mapped before the outer margin and is discarded.

The aligned read in Figure 2.2 has a left margin of 3 and 0 base pairs respectively for Cigar strings 1 and 2. Table 2.1 illustrates various possible outputs of the extension extraction procedure defined in Algorithm 2 depending on the values set for the inner and outer margins.

Algorithm 2 shows the computation needed to extract the extension sequences. The *extension* structure stores the extending sequence and a Boolean flag indicating

Table 2.1: Outputs for Algorithm 2 given inputs presented in Figure 2.2.

Margins	[0, 2]	[0, 5]	[5, 10]
Cigar string 1	read discarded	extension dropped	extension outputted
Cigar string 2	extension outputted	extension outputted	extension outputted

whether the extension is dropped or not.

To extract any extensions, the alignment record has to successfully map on the contig, this is check at line 6. The next *if* statement checks if the read is a left extension. To be a left extension, a read has to have a soft clipping at its beginning and its first match to the contig has to happen before the inner margin limit. The left extension is created by truncating the read sequence at the beginning of the contig. The position for this operation is calculated by subtracting the position of the first match from the length of the soft clipped head of the read.

Generating the right extension is slightly more complex because the typical SAM record does not expose the end position of the alignment as it does the begin position. The number of bases covered in the alignment has to be calculated by iterating through the Cigar string and is performed by the functions called at lines 17 and 18. CIGAR operations M, D, X and $=$ contribute to the length of the sequence used from the contig, while operations M, I, S, X and $=$ contribute to the length used by the read.

The sequence for the right extension is generated by taking the last *r_len* base pairs from the read sequence, where *r_len* is given as the subtraction of the right margin, as calculated at line 19, from the length of the soft clipped tail of the read.

Table 2.2: CIGAR string operations.

Operation	Description
=	Alignment Match
X	Alignment Mismatch
M	Match or mismatch
I	Insertion
D	Deletion
N	Skipped region of the reference
S	Soft clipped region of the query
H	Hard clipped region of the query
P	Padding

Algorithm 2 Extension extraction algorithm (contig, aln)

```
1: struct extension {
2:     string seq
3:     bool is_dropped
4: }
5: r_ext ← NULL
6: if aln.mapped then
7:     if aln.cigar[0].op == S and aln.beginPos <= outer_margin then
8:         l_len ← aln.cigar[0].count − aln.beginPos
9:         if l_len > 0 then
10:            l_seq ← reverse(aln.seq[0 : l_len])
11:            l_is_dropped ← aln.beginPos > inner_margin
12:            l_ext = extension(l_seq, l_is_dropped)
13:        end if
14:    end if
15:    if aln.cigar[−1].op == S then
16:        clip_len ← aln.cigar[−1].count
17:        used_read_len ← get_used_read_len(aln.cigar)
18:        used_contig_len ← get_used_contig_len(aln.cigar)
19:        margin ← contig.len − aln.beginPos − used_contig_len
20:        r_len ← clip_len − margin
21:        if r_len > 0 and maring <= outer_margin then
22:            r_seq ← aln.seq[used_read_len − r_len : aln.seq.len]
23:            r_is_dropped ← margin > inner_margin
24:            r_ext ← extension(r_seq, r_is_dropped)
25:        end if
26:    end if
27: end if
28: return l_ext, r_ext
```

2.2. The POA Algorithm

The EAGLER scaffolder supports two algorithms to compute the consensus of the extending sequences. The first algorithm is the well known Partial Order Alignment algorithm[15, 16] designed by Christopher Lee. The second algorithm used to compute the consensus extending sequence is a novel approach that combines majority vote calculations, local read error corrections and global read realignments. This algorithm will be discussed in great detail in chapter 3.

The POA algorithm uses a modified Smith-Waterman local alignment algorithm to calculate an alignment graph. The alignment graph is a directed acyclic graph (DAG) where each edge counts the number of sequences that aligned to the graph passing thought that edge.

The alignment graph is built iteratively by adding new sequences to it one by one. Each time the graph is edited, topological sorting is performed to generate a linear ordering of all nodes so that all predecessors for any given node appear in the sorted array before that node. Once the nodes have been sorted, the dynamic programming step is carried out by visiting the nodes in the computed order and all predecessors are considered for align and insert moves.

Extracting a consensus sequence from the graph can be reduced to the well known problem of finding the maximum weight path in the alignment DAG. Nodes are iterated in the reverse topological order so that each node is visited before all its predecessors. The score of each node is initially set to 0 and remains 0 for all nodes that have no outbound edges. The score of all other nodes is updated to maximize the sum of the weight of an outbound edge and the score of the node it points to. Once the node with the highest score has been found, the maximum weight path can be trivially computed by following the heaviest edges from each node.

The POA algorithm has proven to be an order of magnitude slower than the new consensus algorithm proposed in this thesis. The two consensus algorithms achieve similar identity values of the consensus sequence and both showed an indel bias towards insertions. In the case of the POA algorithm the bias is however drastically more pronounced, with deletions barely present in the consensus. The advantage of using the POA algorithm is in its ability to produce long consensus sequence in low coverage states for which the Global/Local Realign algorithm produces very short, although accurate, extensions.

3. Predicting the Next Base

This chapter will present the default algorithm used by the EAGLER scaffolder to compute the consensus sequence given a list of extensions. The algorithm leverages 3 key concepts:

- Majority voting
- Local error correction
- Full read realignment

A modified majority vote algorithm is used to iteratively produce each base of the consensus sequence starting from the edge of the contig. Every time a new base is pushed to the output and is considered finalized, all reads that did not contribute to the vote for that base are checked for errors such as indels and mismatches. The full read realignment is used to reset the state of the algorithm once the read errors have become too severe to correct them locally.

3.1. Simple Majority Vote

Algorithm 3 presents the basic version of the majority vote algorithm used to determine one base of the consensus sequence. The *extension* structure employed by the algorithm is an extended version of the structure defined in Algorithm 2. Each extension keeps track of its first base that has not yet been used to produce an output base in the *position* variable.

The algorithm takes as input a list of extensions and returns a counter array. Each element of the counter array tracks the number of appearances of a specific base. Indexes 0 to 3 correspond to bases A, T, G and C respectively. The output base can be easily found by converting the index of the maximum value in the counter array using the previously mentioned mapping.

The algorithm iterates over all given extensions and increments the counter for the base at the current position. Some extensions may be too short to participate in the

majority vote so they have to be skipped. This case in checked by the *if* statement at line 8.

Algorithm 3 Simple majority vote (extensions)

```
1:  struct extension {
2:      string seq
3:      bool is_dropped
4:      int position
5:  }
6:  counter ← [0, 0, 0, 0]
7:  for ext in extensions do
8:      if ext.position < ext.seq.len then
9:          base ← ext.seq[ext.position]
10:         increment(counter, base)
11:     end if
12: end for
13: return counter
```

3.2. Modified Majority Vote

The modified majority vote described in Algorithm 4 uses the same *extension* structure and produces the same output as Algorithm 3 presented in the previous section. The two algorithms differ in the input arguments, the modified majority vote takes two additional arguments.

The first additional argument is an integer offset that is added to the position at which the output base is extracted from the extension sequence. The second additional argument is a function that takes as input a character representing a base and returns a Boolean. This function acts as a filter and controls whether an extension should be considered in the current majority vote based on the passed argument. The filtering mechanism is useful because it enables the calculation of the majority vote over a subset of extensions without the need to actually instantiate that subset in memory.

This custom majority vote algorithm that offers more granulated control over which extension may vote and what base is taken from eligible extensions is the main building block for the Confirming Majority Vote algorithm used by the EAGLER scaffolder to compute a single base of the extension sequence.

Algorithm 4 Modified majority vote (extensions, offset, is_ext_eligible)

```
1: counter ← [0, 0, 0, 0]
2: for ext in extensions do
3:     if ext.is_dropped then
4:         continue
5:     end if
6:     if ext.position + offset < ext.seq.len then
7:         base_check ← ext.seq[ext.position]
8:         base_output ← ext.seq[ext.position + offset]
9:         if is_ext_eligible(check_base) then
10:            increment(counter, base_output)
11:        end if
12:    end if
13: end for
14: return counter
```

3.3. Confirming Majority Vote

The confirming majority vote algorithm takes as input a list of extensions, usually one of the results of a call to the procedure defined in Algorithm 2.

The algorithm first of all computes a majority vote over the given extensions using the procedure defined in section 3.2. This majority vote is computed over all non-dropped extensions without offsetting the position stored in the *extension* structure.

To output a base, the algorithm needs to meet a preset minimum coverage. The coverage is calculated as the sum of the values in a *counter* array, i.e. the total number of extensions that contributed to a vote. Given sufficient coverage, the confirming part of the algorithm is executed.

A second majority vote, as defined in Algorithm 4, is calculated only over the subset of extensions that voted for the most frequent base in the first voting run, i.e. *output_base* from line 6. The position for the vote is also offset by $+1$ to target the first position after the one used in the previous majority vote computation.

Only if the confirming majority vote achieves a coverage of at least 60% of the minimum coverage, the output base is returned to the caller along with the resulting base for the confirming majority vote. The idea behind this safety check is that if there is sufficient confidence to output a base in the current iteration, there should also be a fairly high confidence in the next base to be pushed to the output considering only the

extensions that agreed on the first voted base.

The returned base at this point is considered finalized and will appear in the extended contig computed by the scaffolder. Both bases returned by a procedure call to Algorithm 5 will later be used to attempt to correct local errors in the extensions that did not vote on line with the majority. More on this topic will follow in the next section.

Algorithm 5 Confirming majority vote (extensions)

```
1:  function true_predicate(base)
2:      return true
3:  end function
4:  bases ← majority_vote(extensions, 0, true_predicate)
5:  if coverage(bases) >= min_coverage then
6:      output_base = output_bp(bases);
7:      function is_read_eligible(base)
8:          return base == output_base
9:      end function
10:     confirming_counter = majority_vote(extensions, 1, is_read_eligible)
11:     next_base = output_bp(confirming_counter)
12:     if coverage(confirming_counter) >= 0.6 * min_coverage then
13:         return output_base, next_base
14:     end if
15: end if
16: return NULL, NULL
```

3.4. Local Realignment

The local realignment procedure detailed in Algorithm 6 uses the two resulting bases from the confirming majority vote algorithm to correct single-base errors in the extensions that did not vote for the output base.

Four bases are mutually tested to determine the type of error correction to be performed for each extension. The first two, which are constant for all extensions, are the bases resulting from Algorithm 5. The third base is the base at the current position in the extension, while the fourth one is the one immediately after it. Because of this prerequisite, every extension that does not have at least 2 unused bases left is immediately dropped as shown on line 6.

The tests over the four mentioned bases attempt to classify which of the following operations should be performed: match, mismatch, insertion or deletion. The first performed test detects a match, i.e. the extension voted for the output base, and in that case the position for the given extension moves forward by $+1$.

A deletion is detected by comparing the current base in the extension to the second output base predicted by the confining majority vote. If these two bases match, the position for the examined extension is not edited since it will most likely contribute to the next majority vote.

When the majority voted next base and the next base in the extension coincide, but the extension did not vote for the output base, a mismatch is detected. In terms of position offsetting, this case is handled the same way as a match.

Algorithm 6 Local realignment (extensions, output_base, next_mv_base)

```
for ext in extensions do
    if ext.is_dropped then
        continue
    end if
    if ext.position >= ext.seq.len − 2 then
        ext.is_dropped ← true
        continue
    end if
    current_base ← ext.seq[ext.position]
    next_base ← ext.seq[ext.position + 1]
    if current_base == output_base then
        handle_operation(ext, MATCH)
    else if current_base == next_mv_base then
        handle_operation(ext, DELETION)
    else if next_base == next_mv_base then
        handle_operation(ext, MISMATCH)
    else if next_base == output_base then
        handle_operation(ext, INSERTION)
    else
        ext.is_dropped ← true
    end if
end for
```

The last test checks whether the output base matches the next base in the examined extension. Upon a positive test result, the operation is classified as an insertion and the position in the current extension is moved $+2$ positions forward.

In some cases all tests yield negative results and the algorithm fails to classify an operation. When this scenario is realized, the extension is marked as dropped and will not participate in the next output base voting round. The next section will present a method to recover dropped extensions when the scaffolder starts lacking coverage and cannot output the next extension base.

Figure 3.1 gives an example of 8 extensions contributing to a consensus sequence where all test cases are present. The green bases represent a match, the red one is a mismatch, while the orange one is an insertion. The blue colored bases indicate the positions for the next round of voting and in case of *Extension 5* it also indicates a deletion. *Extension 7* fails all tests defined in Algorithm 6 and is dropped, hence it does not have a position for the next voting round.

Extension 1:	A	G	C	T	T	T	T	C	A	T	C	T	G	A	C
Extension 2:	A	G	C	T	T	T	T	C	A	T	C	T	G	A	C
Extension 3:	A	G	C	T	T	T	T	C	G	T	C	T	G	A	C
Extension 4:	A	G	C	T	T	T	T	C	A	T	C	T	G	A	C
Extension 5:	A	G	C	T	T	T	T	C	T	C	T	G	A	C	T
Extension 6:	A	G	C	T	T	T	T	C	G	A	T	C	T	G	A
Extension 7:	A	G	C	T	T	T	T	C	G	G	C	T	G	A	C
Extension 8:	A	G	C	T	T	T	T	C	A	T	C	T	G	A	C
Consensus:	A	G	C	T	T	T	T	C	A	*	*	*	*	*	*

Figure 3.1: Error classification in the local realignment algorithm.

3.5. Global Realignment

The previous sections have explained in great detail the consensus sequence generation algorithm. The last issue left to address to complete the contig extension algorithm is the Global Realignment algorithm used to amend low coverage states.

The consensus sequence generation algorithm starts with a given coverage level, i.e. the number of reads that mapped within the inner margin. With each output base

the algorithm can only stay at the current coverage level or drop to a lower coverage. The Global Realignment is invoked when the scaffolder does not have sufficient coverage to output the next base or when it does not have sufficient coverage to confirm it, as seen in Algorithm 5.

The algorithm attempts to recover some coverage using dropped extensions. There are two sources of dropped extensions: out of margin reads and unclassified local errors. Out of margin reads are those reads that aligned to the contig between the inner and outer margin and were dropped during the extension extraction phase. Unclassified local errors also produce dropped extensions as shown in Algorithm 6.

The Global Realignment algorithm is similar to the first pairwise alignment between contigs and reads carried out at the very beginning of the scaffolding pipeline. The algorithm however operates on a single contig and on a reduced set of reads to boost performance.

The procedure takes as input an extended contig, the original set of long reads and the left and right extensions at the moment of insufficient coverage. The output consists of two lists, the right and left extensions respectively, obtained by realigning the dropped reads onto the extended contig.

The *extension* structure defined in Algorithm 3 is expanded to also include the ID of the read which the extension originated from. This integer ID is used to rapidly fetch the read associated to an extension from the input dataset as shown in lines 8 and 18.

The procedure iterates over all given extensions and creates a list of reads that have to be realigned, while non-dropped extensions are saved so that they can be propagated to the output. If the *realign_reads* list is not empty, the underlying aligner is invoked to align the selected reads to the extended contig.

The SAM file produced by the aligner is forwarded to the *compute_extensions* procedure defined in Algorithm 2 along with the contig that the alignment records refer to. The final output extension lists are generated by combining the extensions extracted from the SAM file and the previously saved extensions.

Algorithm 7 Global realignment (contig, reads, left_extensions, right_extensions)

```
realign_reads ← [ ]
keep_left_exts ← [ ]
keep_right_exts ← [ ]
for ext in left_extensions do
    if ext.is_dropped then
        read_id ← ext.read_id
        if read_id not in realign_reads then
            realign_reads.append(reads[read_id])
        end if
    else
        keep_left_exts.append(ext)
    end if
end for
for ext in right_extensions do
    if ext.is_dropped then
        read_id ← ext.read_id
        if read_id not in realign_reads then
            realign_reads.append(reads[read_id])
        end if
    else
        keep_right_exts.append(ext)
    end if
end for
if realign_reads.size > 0 then
    sam_file ← align(contig, reads)
    realn_left, realn_right ← compute_extensions(contig, sam_file.aln)
    left_extensions = keep_left_exts + realigned_left
    right_extensions = new_right_exts + realigned_right
end if
return left_extensions, right_extensions
```

4. The Scaffolder

Chapters 2 and 3 presented the building blocks for the extension based EAGLER scaffolder. This chapter will rework the generic extension based scaffolding procedure presented in Algorithm 1 to closely match the one used in the EAGLER C++ implementation.

The first step of the scaffolding pipeline is the pairwise alignment between all contigs from the input draft genome and all given long reads. Once the alignment is ready, each contig can be processed independently and its extended version can be collected in the *extended_contigs* list.

The contig extension process is controlled by two Boolean flags, the *should_ext_left* and the *should_ext_right* one. The current contig is iteratively extended while either flag is set using the *mv_local_realign* function defined in Algorithm 9. This function combines the Confirming Majority Vote from Algorithm 5 and the local error correction from Algorithm 6 to produce one consensus extending sequence.

Once both the left and right consensus sequences have been computed, the extended contig can be constructed by prefixing the original contig with the left consensus and suffixing it with the right consensus sequence, as shown on line 17. The left consensus has to be reversed before attaching it to the contig because the extension extraction and consensus computation procedure treat all extensions as right extensions.

The maximum length by which a contig should be extended on each side is a scaffolder parameter saved in the *max_ext_len* variable. This limitation is introduced to avoid unnecessary work since an approximation of the gap sizes for the analyzed genome are usually known. There is no benefit in setting the maximum extension length to a value greater than the size of the largest gap.

The extension process for one side of the contig is stopped either when the current iteration does not produce a new output base or when the maximum extension length is reached. If either side of the contig are extended in the next iteration a global realignment is carried out as per Algorithm 7.

Algorithm 8 EAGLER Scaffolder (draft_genome, long_reads)

```
extended_contigs ← [ ]
sam_file ← align(draft_genome, long_reads)
for contig in draft_genome do
    extended_contig ← copy(contig)
    should_ext_left ← true
    should_ext_right ← true
    left_exts, right_exts ← compute_extensions(sam_file.aln)
    while should_ext_left or should_ext_right do
        if should_ext_left then
            left_consensus ← reverse(mv_local_realign(left_exts))
            should_ext_left ← left_consensus.seq.len > 0
        end if
        if should_ext_right then
            right_consensus ← mv_local_realign(right_exts)
            should_ext_right ← right_consensus.seq.len > 0
        end if
        extended_contig.seq ← left_consensus.seq + extended_contig.seq +
    right_consensus.seq
        if extended_contig.left_ext.seq.len > max_ext_length then
            should_ext_left ← false
        end if
        if extended_contig.right_ext.seq.len > max_ext_length then
            should_ext_right ← false
        end if
        if should_ext_left or should_ext_right then
            left_exts, right_exts ← global_realign(extended_contig,
    long_reads, left_exts, right_exts)
        end if
    end while
    extended_contigs.append(extended_contig)
end for
scaffolds ← join_contigs(extended_contigs)
return scaffolds
```

Algorithm 9 Confirming majority vote with local realignment (extensions)

```
extension ← ""
while true do
    output, next ← confirming_majority_vote(extensions)
    if output == NULL then
        break
    end if
    extension ← extension + output
    local_realign(extensions, output, next)
end while
return extension
```

4.1. The Contig Connection Algorithm

The last step in the scaffolding pipeline is joining overlapping contigs in the scaffolds that will be the final output of the pipeline.

Merging overlapping contigs may look like a simple task but actually presents quite a few challenges, the first one being the size of the contigs. The contigs produced by NGS assemblers can be very large and Megabase lengths are not uncommon.

Aligners built for mapping reads, either short or long, onto a reference don't usually cope well when given two contigs as input. For this reason, the Contig Connection algorithm cannot use the whole sequence of a contig to detect overlaps but rather uses relatively short representative sequences from the ends of the contigs. This sequences are termed anchors and combine the extended part of a contig with a preset chunk size of the original contig sequence as shown in Figure 4.1.

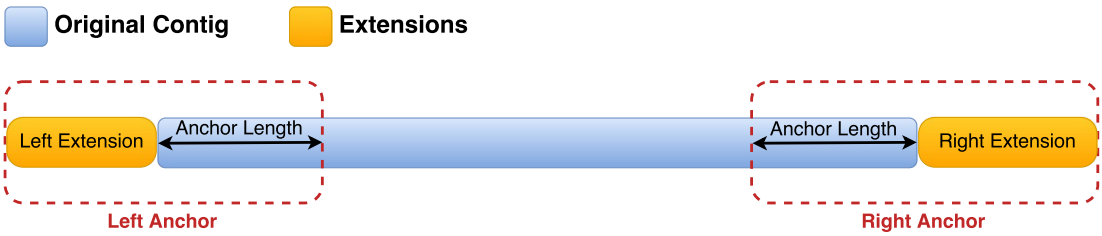

Figure 4.1: Anchors for an extended contig.

The aim of an anchor is to unequivocally identify the edge of a contig. By definition, the original contigs from a draft genome are not overlapping, at least not significantly. This fact leads to the conclusion that the only way for two extended contigs to

overlap is for their anchors to overlap. Hence, an expensive Megabase scale, contig to contig, alignment can be avoided and is replaced by a Kilobase scale, anchor to contig, alignment.

The anchor mechanism has however a drawback, the extended contigs have to meet a minimum length criterion. The extended contig has to be large enough to yield two non-overlapping anchors or otherwise the contig will not be merged with any other contigs, yielding a singleton scaffold.

Each anchor has its unique ID that is constructed by concatenating the strings "$|L$" or "$|R$" to the name of the contig.

Algorithm 10 Create contig anchors (extended_contigs)

```
anchors ← [ ]
for contig in extended_conitgs do
    left_len ← contig.left_ext.seq.len
    right_len ← contig.right_ext.seq.len
    if conitg.seq.len < 2 * anchor_len + left_len + right_len then
        continue
    end if
    left_anchor ← conitg.seq[0, left_len + anchor_len]
    right_anchor = contig.seq[−right_len − anochor_len : conitg.seq.len]
    left_id = contig.name + "|L"
    right_id = contig.name + "|R"
    anchors.append((left_id, left_anchor)))
    anchors.append((right_id, right_anchor)))
end for
return anchors
```

Algorithm 11 presents the pseudocode for the Contig Connection algorithm. The algorithm takes as input a list of extended contigs and outputs a list of disjoint scaffolds. To simplify the passage of arguments to other functions, the *connector* structure is defined.

The *connector* context tracks all contigs, scaffolds and anchors used in the Contig Connection algorithm. The components of the connector are listed as *collection* objects for simplicity, but the implementation also uses additional data structures to speed-up the computation and element retrieval. A set is used to store the names of all contigs that are part of scaffold, while free contigs are tracked in a map with the contig name as key. The scaffold that a contig is part of can be found in constant time by using a

hash map that uses contig IDs as keys and references to scaffolds as values.

The *create_scaffold* functions creates a new scaffold given a *connector* context. The function detects the first contig in the *connector* that is not part of a scaffold and constructs a new scaffold with the found contig at its core. The function will also append the newly created scaffold to the list of scaffolds saved in the *connector* context. In case there are no free contigs, the function will not be able to create a new scaffold and will return NULL.

The procedure iteratively calls the *connect_next* function which attempts to connect a free contig to the current scaffold. Details for this function are given in Algorithm 12. When the current scaffold cannot be merged with any more contigs, a new scaffold is created by picking one of the remaining free contigs. The iterative loop continues until all input contigs are part of a scaffold.

Algorithm 11 Join extended contigs (extended_contigs)

```
1:  struct connector {
2:      collection contigs
3:      collection scaffolds
4:      collection anchors
5:  }
6:  anchors ← create_anchors(extended_contigs)
7:  context ← connector(extended_contigs, [ ], anchors)
8:  current ← create_scaffold(context)
9:  while current ≠ NULL do
10:     found ← connect_next(context, current)
11:     if not found then
12:         current ← create_scaffold(context)
13:     end if
14: end while
15: if trim_circular_genome then
16:     for scaffold in context.scaffolds do
17:         correct_circular_scaffold(scaffold)
18:     end for
19: end if
20: return context.scaffolds
```

The *trim_circular_genome* is a setting of the scaffolder that controls whether the scaffolder should be checked for circularity. When working with circular genomes, a scaffold may start to circle on itself, i.e. the left and right ends of the scaffold may overlap. This phenomenon introduces redundant bases in the output and depending on the purpose of the scaffolding procedure may be wanted or not. When the mentioned flag is set, the ends of each scaffold will be checked for overlaps and redundant genomic material will be trimmed from the output.

The procedure given in Algorithm 12 takes as input the *connector* context and the current scaffold. The return value is a Boolean indicating whether the procedure successfully joined a contig to the scaffold.

A new contig is always concatenated to the right end of a scaffold, never to the left end. At first sight this may seem as an error since a free contig can indeed align to both sides of the scaffold but it is not the case. Contigs that map to the left of a particular scaffold will eventually become scaffolds themselves and the first scaffold will than map to the right of the newly formed scaffold. This merge right-only policy drastically simplifies the algorithm since it has to know how to handle only one case instead of two, very different, cases.

To evaluate prospective candidates for merging, all contig anchors are aligned to the last contig in the scaffold. Each alignment record is examined until a suitable one is found. The *should_connect* function is a preliminary test to check if a record is eligible. Various conditions are tested such as the mapping status and the contig state.

The contig name can be easily accessed through the anchor alignment record by removing the last two characters from the read name stored in the record. The test at line 8 checks if the candidate contig is already in the scaffold. The positive result of this test is a strong indication of a circular genome and that the current scaffold should not be extended further.

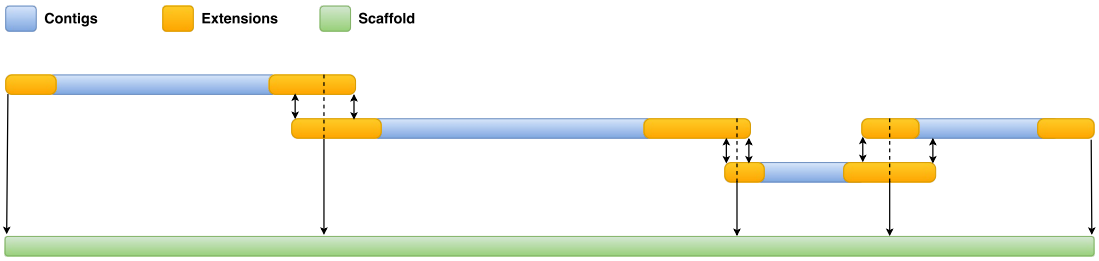

Figure 4.2: Contig and extension contributions to the sequence of a scaffold.

When two contigs are merged, they both contribute to the scaffold sequence as shown in Figure 4.2. Each contig propagates to the end sequence its original non-

Algorithm 12 Connect next (connector, current_scaffold)

1: $current\_contig \leftarrow current\_scaffold.last\_contig$
2: $sam\_file \leftarrow align(current\_contig,\ connector.anchors)$
3: **for** $record$ **in** $sam\_file.aln$ **do**
4: **if not** $should\_connect(connector,\ current\_contig,\ record)$ **then**
5: **continue**
6: **end if**
7: $next\_id \leftarrow record.name[0:-2]$
8: **if** $is\_in\_scaffold(next\_id,\ current\_scaffold)$ **then**
9: **break**
10: **end if**
11: $next\_scaffold \leftarrow scaffold\_for\_contig\_id(connector,\ next\_id)$
12: **if** $next\_scaffold \neq NULL$ **and not** $is\_edge(scaffold,\ next\_id)$ **then**
13: **continue**
14: **end if**
15: $next\_contig \leftarrow find\_contig(connector,\ next\_id)$
16: **if** $record.complement$ **then**
17: $next\_contig \leftarrow reverse\_complement(next\_contig)$
18: **end if**
19: $merge\_start \leftarrow max(current\_contig.right\_ext\_pos,\ record.beginPos)$
20: $right\_ext\_len \leftarrow current\_contig.seq.len - merge\_start$
21: $next\_start \leftarrow min(right\_ext\_len,\ next\_contig.left\_ext.seq.len)$
22: $merge\_end \leftarrow next\_start + record.beginPos$
23: $merge\_len = merge\_end - merge\_start$
24: $add\_contig(scaffold,\ merge\_start + merge\_len/2,\ next\_start - merge\_len/2)$
25: **if** $next\_scaffold \neq NULL$ **then**
26: $merge(connector,\ scaffold,\ next\_scaffold)$
27: **end if**
28: **return true**
29: **end for**
30: **return false**

extended part which, as a result of an NGS assembler, is considered to be without errors. The gap between two contigs is filled by their extensions in such a way that both contigs contribute equally to the end sequence of the scaffold. The arithmetic to compute this contribution boundaries can be seen on lines 19 through 24.

At the end, if the merged contig was part of a previously formed scaffold, all contigs from that scaffold are also merged into the current scaffold.

5. Implementation Details

The EAGLER scaffolder is implemented in C++ and uses the C++11 language standard. The tool should be compatible with most UNIX flavors and has been successfully tested on the following operating systems:

- Mac OS X El Capitan 10.11.1
- Mac OS X Yosemite 10.10.3
- Ubuntu 14.04 LTS

As of version v1.0.1, available at `https://github.com/mculinovic/EAGLER/releases/tag/v1.0.1`, the implementation presents the metric displayed in Table 5.1. The implementation has 4 top-level dependencies as shown in Figure 5.1.

Table 5.1: Implementation metrics obtained by running cloc[17] in the root of the project before initializing the submodules.

Language	Files	Lines of code
C++ (.cpp)	11	1339
C++ (.h)	11	492
Python	7	342
Shell	3	176
Make	3	49
Sum	35	2398

The following libraries are needed at compile time by the EAGLER scaffolder:

- SeqAn[18]
- CPPPOA[19]

SeqAn is an efficient C++ library for sequence analysis. The EAGLER scaffolder uses it mainly for the input and output of standard bioinformatics formats such as FASTA files containing reads and contigs, and SAM files created by aligners. The

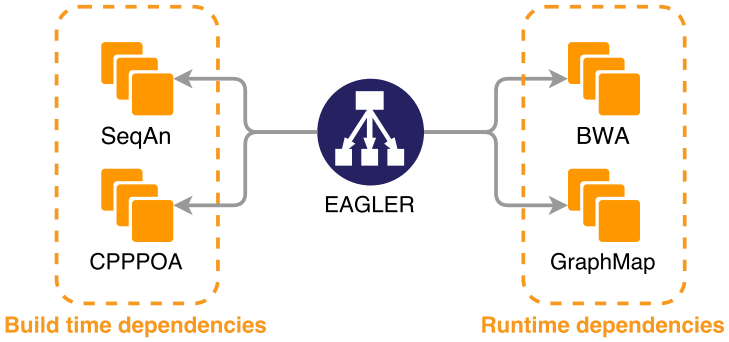

Figure 5.1: A high level dependency diagram for EAGLER scaffolder.

implementation of the Partial Order Alignment algorithm from the CPPPOA library is used to provide an alternative to the default Global/Local Realign algorithm. Both libraries will be automatically downloaded, configured and built by entering the installation commands (see 5.1) in a terminal window.

The scaffolder currently supports 2 aligners and their executables are required at runtime:

- BWA[20]
- GraphMap[21]

The end user is responsible for installing the aligners on the host system and making sure that the executable for the selected aligner is reachable from his 'PATH' environment variable.

The Burrows-Wheeler Aligner is the de facto industry standard for aligning long reads to a reference genome and is the default aligner used by the EAGLER scaffolder. Support for a second aligner has been provided for testing and comparison purposes, as well as to reduce the workload required to add support for another aligner in future releases.

The GraphMap aligner was selected for this purpose as one of the most promising novelties in its field. GraphMap is an optional requirement and the scaffolder can run without it being installed on the host system, as long as the command line flags are not explicitly enabling it. See Table 5.2 for details.

The scaffolder relies on an aligner abstraction instead of the concrete implementation for a specific aligner. Any tool that can satisfy the minimum interface defined by the *Aligner* abstract class from the 'src/aligners/aligner.h' file is compatible with the EAGLER scaffolder. The most important constraint being the possibility to mark multiple significant alignments as primary.

Adding support for a new aligner requires implementing a wrapper to the terminal

interface of the aligner realized as a class inheriting the previously mentioned *Aligner* base class. A new command line option would also be necessary to trigger the activation of the aligner.

The only part of the code aware of all concrete aligner implementations is the static method 'init' from the 'src/aligners/aligner.cpp' file that is used to instantiate the desired aligner at the beginning of the scaffolding pipeline. The 'init' method is the last element that would have to be edited in order to support a new alignment.

The provided dependency graph in Figure 5.2 shows how the core of the EAGLER scaffolder depends exclusively on the abstract aligner defined in the header file 'aligners/aligner.h' and each concrete aligner has its own isolated dependency graph. The included header files from the standard library, such as *string* or *vector*, are not represented in the graph to make it more readable.

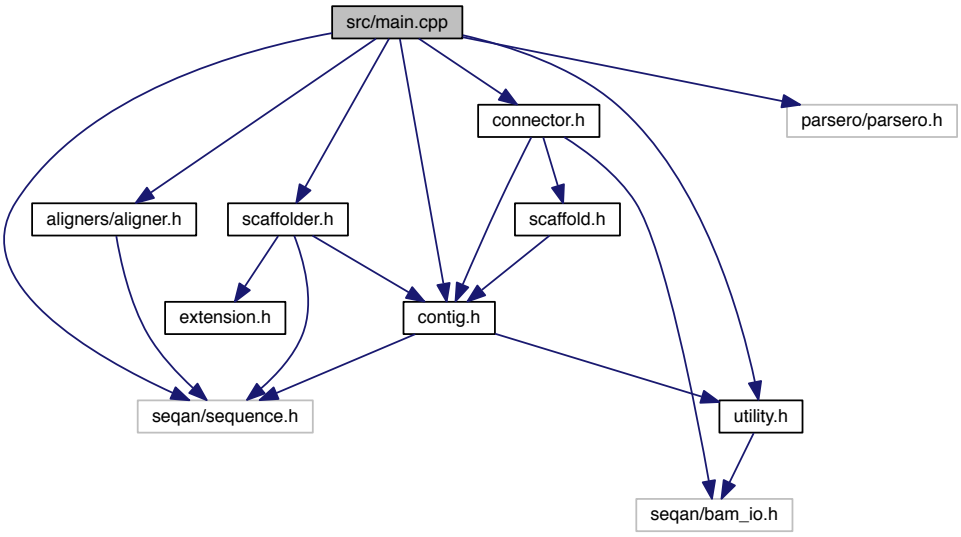

Figure 5.2: Include graph for the main file of the EAGLER scaffolder.

5.1. Building the executable

To be able to build the scaffolder, the executables for the programs listed below must be reachable from the 'PATH' variable of the user's shell:

- git[22]
- g++[23]
- GNU Make[24]
- Doxygen[25]

The Version Control System used to track the development of the project is Linus Torvalds' git, while the build system in charge of creating all the necessary files to run the scaffolder is GNU Make. The g++ compiler is used to compile and link the C++ source code. The version of g++ should be 4.9.0 or higher to correctly compile the C++11 standard code. The Doxygen package is used to create the project documentation in HTML and PDF formats, and is optional in case the user does not require the code manual.

To build the EAGLER scaffolder, the following commands are to be used from the folder where the tool will be installed:

```
git clone https://github.com/mculinovic/EAGLER.git
cd EAGLER/
git checkout v1.0.1
git submodule update --init --recursive
make
```

The first two commands will download the source code of the project from it's remote host and position the user in the root folder of the project. The third command, 'git checkout v1.0.1', is used to activate the specific version 1.0.1 that is discussed in this thesis. The fourth command initializes the project submodules, SeqAn and CPPPOA, and fetches the required files from their remote repositories.

The last command will initiate the build process for the scaffolder executable and will trigger the make process for the submodules as needed. Running the 'make' command without arguments will build the release version of the tool as the binary file './release/eagler'.

To build the debug version of the tool use:

```
make debug
```

To build both the debug and release versions use:

```
make all
```

Once the release version has been built the following command may be used to install the EAGLER scaffolder in the '/usr/local/bin' shared location. By doing this, the executable becomes available to the user throughout the UNIX shell and third-party applications can easily locate it.

```
make install
```

Depending on the operating system and user type, this command might require super-user privileges in order to run correctly. In that case the user will be prompted automatically to enter his password in the terminal, there is no need to prepend 'sudo' to the 'make' command.

To delete all files generated during the build process, both for the debug and release versions, use:

```
make clean
```

The previous command, however, will not remove the files created by the install command. To remove those files, use the command listed below. The same level of privileges as for using the install command will be required.

```
make uninstall
```

5.2. The Documentation

The source code comments of the scaffolder are written to be compatible with the Doxygen documentation generator. To successfully generate the documentation, the 'doxygen' executable must be reachable from the user's 'PATH'. The Graphviz[26] open source graph visualization software must also be present on the host system to render the graphs present in the documentation.

To create the documentation in HTML and LaTeX format, run the following command from the root of the scaffolder:

```
make docs
```

The HTML documentation is placed in './docs/html' and can be viewed by pointing any web browser at 'docs/html/index.html'.

The LaTeX documentation is placed in 'docs/latex' and needs compiling before it can be viewed. The PDF documentation is obtainable by compiling the generated LaTeX code with the provided makefile.

Use the following commands from the root of the project to access the PDF version of the documentation:

```
cd docs/latex/
make
open refman.pdf
```

5.3. Running the scaffolder

The EALGER scaffolder expects 3 arguments from the command line and supports the options detailed in Table 5.2. To run the scaffolder, use the following command format:

```
./release/eagler [options] <draft_genome.fasta> <long_reads.fasta>
    <output_prefix/output_dir>
```

The 3 mandatory arguments are detailed as follows:

draft_genome.fasta

Path to the FASTA file containing the draft genome created by an NGS assembly pipeline such as SGA or Spades.

long_reads.fasta

Path to the FASTA file containing long reads, PacBio or Nanopore, to be used in the scaffolding process.

output_prefix/output_dir

The prefix to be added to the output files or the directory where the scaffolder will store the results. If the given argument is an existing directory, than the results will be placed in that directory with default file names. If the argument is not a valid directory, it will be treated as a prefix that will be prepended to the name of each output file stored in the current working directory.

The settings for the default configuration, i.e. no command line options, expecting PacBio reads as input, will use the BWA aligner and the Global/Local Realign to compute the consensus of the contig extensions. The implementation will also automatically detect the number of hardware threads supported by the system and adjust accordingly. A detailed view of all the default settings is given in Table 5.3.

Usage instructions can be accessed through the terminal once the EAGLER scaffolder is installed on the host system by using the following command:

```
    ./release/eagler -h
```

Table 5.2: EAGLER command line options available as of v1.0.1.

Option	Type	Description
-c	INT	Set the minimum coverage to output an extension base
-g	FLAG	Use GraphMap aligner instead of the default BWA aligner
-k	FLAG	Disable circular genome trimming
-m	INT,INT	Set the inner and outer margins in base pairs
-p	FLAG	Use the POA consensus algorithm instead of the default G/L Realign algorithm
-s	INT	Set the maximum extension size in base pairs
-t	INT	Set the number of parallel threads to be used by the aligner
-v	FLAG	Print the version of the scaffolder and exits
-x	ENUM	Set the input reads type [pacbio, ont]

Table 5.3: Default settings for the EAGLER scaffolder.

Setting	Default Value
Read Type	PacBio
Aligner	BWA
Consensus Algorithm	G/L Realign
Minimum Coverage	5
Inner Margin	5 BP
Outer Margin	15 BP
Maximum Extension Length	1000 BP
Circular Genome Trimming	Enabled
Parallel Threads	Physical Cores

Usage Example 1

```
./release/eagler -x pacbio -t 16 draft.fasta reads.fasta output_dir/
```

The above command will run the scaffolder over the draft genome 'draft.fasta' using 16 parallel threads. The input for this example is a set of PacBio long reads from the 'reads.fasta' file, the type of input reads is set by the '-x' option. The output of the scaffolder will consist of 3 files stored in the 'output_dir' directory as show in Table 5.4.

Table 5.4: Output files created by running the first EAGLER usage example.

Output File	Content
output_dir/contigs.fasta	Contigs from the draft genome extended by the scaffolder
output_dir/extensions.fasta	Left and right extensions for each contig in the draft
output_dir/scaffolds.fasta	Final scaffolds created by merging overlapping extended contigs

Usage Example 2

```
./release/eagler -g -p -x ont draft.fasta ont_reads.fasta example_2
```

The second usage example will run the scaffolder over the draft genome 'draft.fasta' using as many parallel threads as there are cores on the host machine. In this case the input is a set of Oxford Nanopore 2D reads stored in the 'ont_reads.fasta' file and the GraphMap aligner will be used to map the reads onto the draft genome. The extension will be computed by the POA algorithm, '-p' option, instead of the default Global/Local Realign algorithm. The output of the scaffolder will consist of 3 files stored in the current working directory as shown in Table 5.5.

Table 5.5: Output files created by running the second EAGLER usage example.

Output File	Content
example_2.contigs.fasta	Contigs from the draft genome extended by the scaffolder
example_2.extensions.fasta	Left and right extensions for each contig in the draft
example_2.scaffolds.fasta	Final scaffolds created by merging overlapping extended contigs

Usage Example 3

```
./release/eagler -k -m 10,25 -s 3000 draft.fasta pacbio_reads.fasta ex3_out/
```

The last usage example will run the scaffolder over the draft genome 'draft.fasta' and will take as input a set of PacBio reads. The inner and outer margins are set to 10 and 25 bases respectively by the '-m' option. To avoid overextension, the maximum extension length for both sides of each contig is set to 3000 base pairs by the '-s' option. The '-k' option will ensure that the scaffolds are not checked for circularity. The output of the scaffolder will consist of 3 files stored in the 'ex3_out' directory following the same patter shown in Table 5.4.

6. Test results

6.1. PacBio Reads

The EAGLER scaffolder has been tested on a simulated draft genome created by cutting the Escherichia Coli[27] draft genome into contigs. The E. Coli genome used in the test is of strain K-12 and substrain MG1655.

The EAGLER scaffolder provides utility scripts to cut gaps in a genome, either deterministically or at random. The dataset used in this section was obtained by creating 3 random gaps of size between 1000 and 2000 base pair in the E. Coli reference sequence.

The dataset can be reproduced by using the *genome2contigs* script over the E. Coli reference sequence. This Python script is shipped with the scaffolder and can be run as shown below. The first argument of the script points to the FASTA file containing the reference. The second one points to the file where the created draft genome will be stored, while each subsequent argument defines a gap in the genome. The script will truncate the reference sequence at the given intervals where the reference is a 0-indexed array and the given intervals are inclusive.

```
python3 genome2contigs.py reference.fasta draft_1.fasta 304857-306035
    1291637-1293139 2887649-2888922
python3 genome2contigs.py reference.fasta draft_2.fasta 516970-518104
    1581180-1582840 2810892-2812204
python3 genome2contigs.py reference.fasta draft_3.fasta 200362-201591
    2197812-2199227 3370974-3372487
```

The long reads used for this test are part of a publicly available dataset provided by Pacific Biosciences on their Github page[28]. The dataset contains 75152 raw, unfiltered reads and has a declared coverage of 19.96x. The N50 measure for the dataset is 5900, while the longest read has a length of 19416 base pairs. Figure 6.1 shows the distribution of the lengths of the reads present in the dataset.

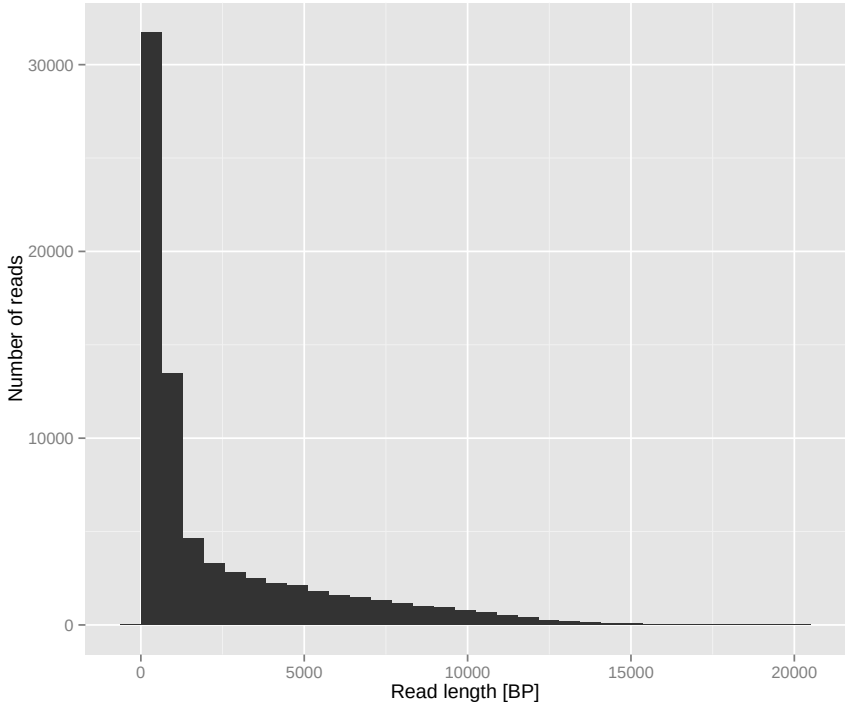

Figure 6.1: Read length distribution for the PacBio test dataset[28].

Each draft genome is run through the scaffolding pipeline and independently analyzed with the provided script *extension_analysis*. The scaffolder settings are the default ones, except for the maximum extension length set to 2500 base pairs. The machine that has been used for this test is a server with 24 Intel Xeon E5645 processors running at 2.40 GHz. Time and space consumption data is available in Table 6.1. The measurements have been acquired by using standard UNIX command line tools *time* and *ps*.

Table 6.1: Time and space consumption of the scaffolder during the PacBio test.

	Draft genome 1	Draft genome 2	Draft genome 3	Average
User time	7m 31.921s	7m 4.599s	7m 56.279s	7m 30.930s
Used memory	2167 MB	2079 MB	2029 MB	2091.67 MB

The script outputs a table like format with various statistics computed using the alignment of each extensions to the reference genome. The output is reproduced in Table 6.3 through Table 6.5.

The first column of the table defines the extension that is being analyzed where the name is obtained by suffixing the contig name with "L" or "R" for a left or right

extension respectively. Total counts for matches, mismatches, insertion, deletions and indels are given in columns MTCH, MISM, I, D and I+D respectively. The last column presents the length of the extension in base pairs.

The second column, probably the most representative one for this analysis, shows the percentage identity of the extension sequence. The ID is calculated as the number of matches divided by the sum of matches, mismatches and deletions. The identity value represents a measure of how closely the extension reproduces the bases from the reference sequence.

Two additional statistics are provided for indel analysis: IND and |I-D|. The IND column defines the percentage of an extension sequence that is an insertion or deletion, while |I-D| is the net divergence in length that the extension has towards the reference sequence.

One extension is missing from Table 6.5 because it was too short to produce a significant alignment to the reference genome, which is crucial for the employed analysis tool to work correctly. The missing extension is *gi|545778205|gb|U00096.3|3|R* with a length of only 12 BP.

Table 6.2 presents the average value of each statistical measure calculated over all the generated draft genomes.

Table 6.2: Average statistical values for the PacBio test.

Statistic	Average Value
Identity	98.961%
Matches	1231.0 BP
Mismatches	7.0 BP
Insertions	16.174 BP
Deletions	6.217 BP
Total indels	22.391 BP
Length divergence	10.13 BP
Indels rate	1.987%
Sequence length	1258.043 BP

Table 6.3: Extension analysis for draft genome 1.

Extension	ID	MTCH	MISM	I	D	I+D	|I-D|	IND	LEN
gi|545778205|gb|U00096.3|0|L	100.00%	63	0	2	0	2	2	2.77%	72
gi|545778205|gb|U00096.3|0|R	99.04%	2169	13	23	8	31	15	1.40%	2205
gi|545778205|gb|U00096.3|1|L	97.67%	924	12	23	10	33	13	3.42%	964
gi|545778205|gb|U00096.3|1|R	99.60%	1001	4	10	0	10	10	0.98%	1015
gi|545778205|gb|U00096.3|2|L	97.48%	271	5	8	2	10	6	3.44%	290
gi|545778205|gb|U00096.3|2|R	98.99%	1778	13	14	5	19	9	1.05%	1805
gi|545778205|gb|U00096.3|3|L	98.92%	184	1	1	1	2	0	1.02%	195
gi|545778205|gb|U00096.3|3|R	99.53%	2346	6	28	5	33	23	1.38%	2380

Table 6.4: Extension analysis for draft genome 2.

Extension	ID	MTCH	MISM	I	D	I+D	|I-D|	IND	LEN
gi|545778205|gb|U00096.3|0|L	100.00%	63	0	2	0	2	2	2.77%	72
gi|545778205|gb|U00096.3|0|R	99.55%	1119	1	2	4	6	2	0.53%	1129
gi|545778205|gb|U00096.3|1|L	98.40%	925	8	9	7	16	2	1.68%	949
gi|545778205|gb|U00096.3|1|R	98.90%	2158	15	22	9	31	13	1.41%	2195
gi|545778205|gb|U00096.3|2|L	98.98%	1663	7	27	10	37	17	2.17%	1700
gi|545778205|gb|U00096.3|2|R	99.29%	1120	3	6	5	11	1	0.97%	1131
gi|545778205|gb|U00096.3|3|L	97.53%	1345	5	43	29	72	14	5.14%	1399
gi|545778205|gb|U00096.3|3|R	99.53%	2346	6	28	5	33	23	1.38%	2380

Table 6.5: Extension analysis for draft genome 3.

Extension	ID	MTCH	MISM	I	D	I+D	|I-D|	IND	LEN
gi|545778205|gb|U00096.3|0|L	100.00%	63	0	2	0	2	2	2.77%	72
gi|545778205|gb|U00096.3|0|R	99.35%	1544	8	21	2	23	19	1.45%	1586
gi|545778205|gb|U00096.3|1|L	99.46%	372	2	6	0	6	6	1.53%	390
gi|545778205|gb|U00096.3|1|R	98.35%	2453	23	36	18	54	18	2.14%	2512
gi|545778205|gb|U00096.3|2|L	97.28%	895	13	16	12	28	4	3.03%	924
gi|545778205|gb|U00096.3|2|R	98.64%	1165	10	15	6	21	9	1.76%	1190
gi|545778205|gb|U00096.3|3|R	99.53%	2346	6	28	5	33	23	1.38%	2380

The standard output of the command used to scaffold the first dataset is reproduced in the following listing.

```
./release/eagler -t 8 -s 2500 rand_test/draft_1.fasta pacbio_reads.fasta rand_test/
    rand_1
[INPUT] Reading draft genome: rand_test/draft_1.fasta
[INPUT] Reading long reads: pacbio_reads.fasta
[ALIGNER] Initializing bwa aligner...
[ALIGNER] Creating index...
[ALIGNER] Aligning reads to draft genome using 8 threads...
[ALIGNER] Creating alignments map...
[EXTENDER] Contig extension algorithm: Local/Global Realign
[EXTENDER] Starting extension procedure for contig [1/4]: gi|545778205|gb|U00096
    .3|0|
        Left extension: 72 BP
        Right extension: 2205 BP
        Extended contig length: 307134 BP
[EXTENDER] Starting extension procedure for contig [2/4]: gi|545778205|gb|U00096
    .3|1|
        Left extension: 964 BP
        Right extension: 1015 BP
        Extended contig length: 987580 BP
[EXTENDER] Starting extension procedure for contig [3/4]: gi|545778205|gb|U00096
    .3|2|
        Left extension: 290 BP
        Right extension: 1805 BP
        Extended contig length: 1596604 BP
[EXTENDER] Starting extension procedure for contig [4/4]: gi|545778205|gb|U00096
    .3|3|
        Left extension: 195 BP
        Right extension: 2380 BP
        Extended contig length: 1755304 BP
[CONNECTOR] Attempting to connect extended contigs...
        Writing contig anchors to file...
        Created scaffold with base contig: gi|545778205|gb|U00096.3|3|
                Remaining free contigs: 3
                Connecting contig: gi|545778205|gb|U00096.3|0|
                Connecting contig: gi|545778205|gb|U00096.3|1|
        Created scaffold with base contig: gi|545778205|gb|U00096.3|2|
                Remaining free contigs: 0
                Connecting contig: gi|545778205|gb|U00096.3|3|
        Correcting circular genome scaffolds...
                Examining scaffold [1/1]... UNTOUCHED
[OUTPUT] Writing extended contigs to file: rand_test/rand_1.contigs.fasta
[OUTPUT] Writing extensions to file: rand_test/rand_1.extensions.fasta
[OUTPUT] Writing scaffolds to file: rand_test/rand_1.scaffolds.fasta
        Preparing scaffold 0 with length: 4641494
```

6.2. Nanopore Reads

The test for the Oxford Nanopore reads is conducted over the same strain of Escherichia Coli as the test for PacBio reads. In this case however, the draft genome is obtained by assembling Illumina reads rather than simulating assembly gaps.

The draft genome is obtained by running the SPAdes[3] assembler, version 3.6.1, over an Illumina dataset[29] with coverage of 51.6x. The resulting contigs are passed to the EAGLER scaffolder along with a Nanopore dataset. The scaffolder uses the default configuration with the maximum extension length set to 2500 base pairs.

The Nanopore dataset used in this test is based on Nick Loman's E. Coli dataset[30]. The reads used as input consist of only two dimensional reads from the *pass* folder of the *MAP006-1* run and have an approximated 40x coverage and an N50 of 8836. In total, there are 29635 Nanopore reads with a maximum length of 48834 base pairs.

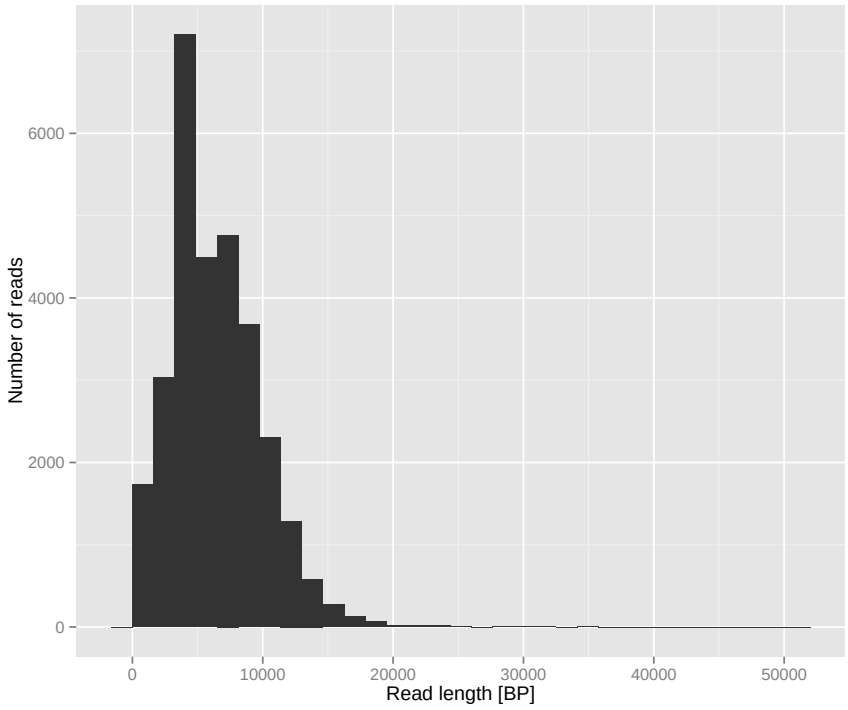

Figure 6.2: Read length distribution for the Nanopore test dataset[30].

The scaffolding process lasted 21.6 minutes and used at most 1589 MB of memory. Once the scaffolds have been constructed, the results are analyzed using the command line tool QUAST[31]. QUAST is a tool for the assessment of the quality of genome assemblies at contig or scaffold level. For the purpose of this test, QUAST is run over the SPAdes assembled draft genome as well as over the scaffolds created by EAGLER.

Table 6.6: QUAST results for SPAdes contigs and EAGLER Scaffolds.

	SPAdes Conitgs	EAGLER Scaffolds
Contigs (>= 0 bp)	121	11
Contigs (>= 1000 bp)	35	11
Total length (>= 0 bp)	4620199	4601091
Total length (>= 1000 bp)	4603244	4601091
Largest contig	667854	1814628
Total length	4606074	4601091
Reference length	4641652	4641652
GC Content	50.77%	50.80%
N50	216199	1107163
Misassemblies	4	4
Local misassemblies	14	21
Mismatches	326	358
Indels	41	206
Short indels	37	198
Long indels	4	8
Indels Length	122	524

From the results shown in Table 6.6 it is clear that EAGLER drastically reduced the number of contigs from 121 to 11. The longest contig after the scaffolding is 1.81 MBP long, compared to a maximum length of 667 KBP in the assembly.

As expected, the N50 is also significantly higher. To clarify, the N50 is a statistical measure defined as the length for which the set of all contigs of that length or longer contains at least half of the total lengths of all contigs. It can be also pictured as the center of mass of the distribution of contig lengths.

SPAdes introduced some error in the contigs it outputted in the form of 4 relocations. This errors obviously propagate to the output of the scaffolder since it assumes the input draft genome to be correct. The errors can however be amplified when contigs misassembled by SPAdes are merged with other contigs to form a scaffold.

These error amplification is very pronounced in the number of long indels and the average indel length. The number of mismatches has seen a 9.81% increase from the assembly to the scaffolds, which is in line with the mismatch rate seen in the previous section. Short indels are however caused by errors in the extension consensus sequence. These errors are usually single-base ones and are a consequence of the high

error rate in the long reads used for the extension process.

The rate of errors introduced in the final product of the scaffolder is however significantly lower than the error rate of a single read. This has been proven in the PacBio test case from the previous section where a single sequence has an average error rate of 15%, while the outputted consensus sequences averaged an identity of almost 99% and an indel rate of less than 2%.

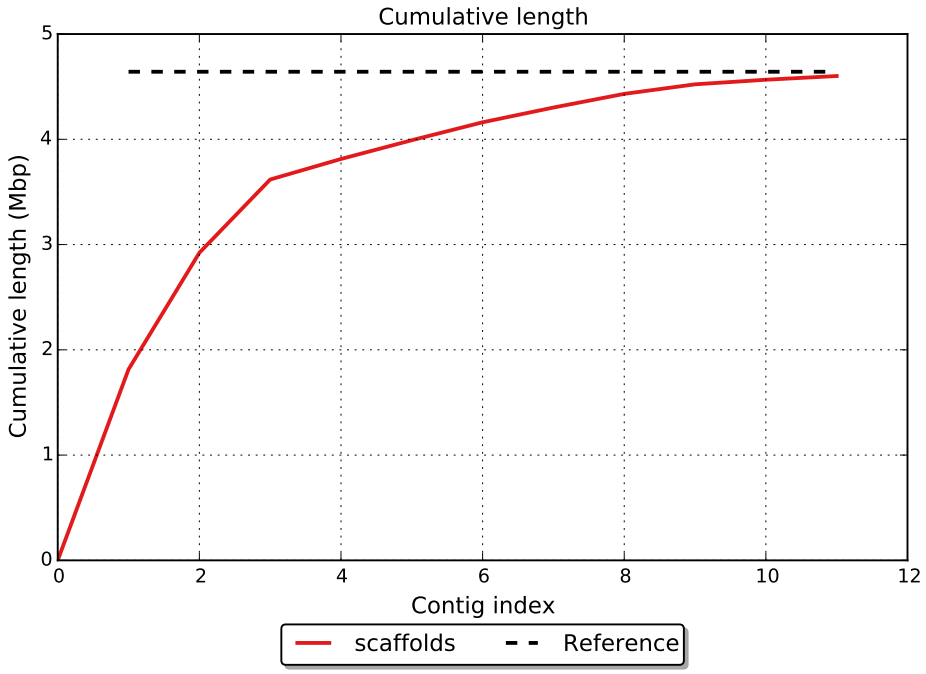

Figure 6.3: Cumulative length of the scaffolds created by the EAGLER scaffolder.

7. Conclusion

NGS assemblers condense short, low error rate reads into contigs but often fail to reconstruct the entire genome. EAGLER is a post assembly tool with the goal to close gaps left open by modern sequence assemblers.

The scaffolding algorithm proposed in Chapter 4 uses a modified implementation of the Majority Vote Algorithm in combination with local single-base error correction and global long read realignment.

Chapter 6 has demonstrated that the proposed tool can create high quality assemblies using PacBio long reads. An average identity of almost 99% has been achieved with a modest 20x coverage of the input dataset. The scaffolder has shown a small bias in the indel type distribution, preferring insertions to deletions. However, the net divergence in length of the extensions compared to the reference is almost negligible. On the average, it has been observed that 8.12 base pairs are in excess for each 1000 outputted consensus bases.

The tool has also proven to be useful when given Oxford Nanopore long reads. Nanopore 2D reads have a higher, but comparable, error rate to PacBio reads but tend to have an error profile that is much more troublesome. Errors in the PacBio reads are almost uniformly dispersed throughout the sequence, while Nanopore reads have very concentrated errors that often span through 3-6 adjacent bases. In the presented test case, the scaffolder has been able to reduce the number of contigs from the 121 created by SPAdes to 11 contigs with the average length of 418281 BP.

The future development of the EAGLER scaffolder will move towards 3 goals: an iterative variant of the scaffolder, a parallel implementation of the extension phase and the support for external tools for the contig connection process.

An iterative variant of the scaffolding model proposed in this thesis may prove useful in closing especially long gaps. Since a full pairwise alignment between the reads and contigs is performed only once at the begging of the pipeline, the scaffolder may eventually run out of genomic material to extend the contigs with. While multiple all-to-all alignments are not a feasible solution in terms of performance, running the

whole pipeline multiple times scales linearly with the number of iterations and should produce comparable results.

A this point in time, the EAGLER scaffolder uses multiple threads only for the alignment process, which is the computationally most intense part of the pipeline. Full multithread support is certainly an objective for future releases but it prospects multiple challenges. The contig extension algorithm is natively parallel because the extension procedure for each specific contig is independent. The challenge however is posed by the side effects of the aligner calls issued by the extension process of each contig.

All modern aligners use temporary files created in the current directory or in the directory of the reference FASTA file. These files are used to store indexes, logs or partial results of the alignments. While parallelizing a single alignment job is usually as simple as setting the appropriate option flag, running multiple jobs is not as straightforward because of the multitude of temporary files that could clash with each other. The only way to support multiple concurrent jobs would be to sandbox each one in its unique temporary folder so that concurrent jobs share no temporary files.

Connecting extended contigs may seem like a simple task, but in reality it is not far from the complexity of a Overlap-Layout-Consensus sequence assembler with a minimal Layout phase. For this reason, the EAGLER scaffolder will probably migrate to an abstraction for the contig connection phase with the goal to support multiple 3rd party solutions in a similar fashion as it has been done with the aligner abstraction.

BIBLIOGRAPHY

[1] J. T. Simpson and R. Durbin, "Efficient de novo assembly of large genomes using compressed data structures," *Genome Res.*, vol. 22, no. 3, pp. 549–556, March 2012.

[2] J. T. Simpson, K. Wong, S. D. Jackman, J. E. Schein, S. J. Jones, and I. Birol, "ABySS: a parallel assembler for short read sequence data," *Genome Res."*, vol. 19, no. 6, pp. 1117–1123, June 2009.

[3] A. Bankevich, S. Nurk, D. Antipov, A. A. Gurevich, M. Dvorkin, A. S. Kulikov, V. M. Lesin, S. I. Nikolenko, S. Pham, A. D. Prjibelski, A. V. Pyshkin, A. V. Sirotkin, N. Vyahhi, G. Tesler, M. A. Alekseyev, and P. A. Pevzner, "SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing," *Journal of Computational Biology*, vol. 19, no. 5, pp. 455–477, 2012. [Online]. Available: http://dx.doi.org/10.1089/cmb.2012.0021

[4] "Pacific Biosciences," accessed November 28, 2015. [Online]. Available: http://www.pacb.com

[5] Pacific Biosciences, "The original long-read sequencer," accessed December 9, 2015. [Online]. Available: http://www.pacb.com/products-and-services/pacbio-systems/rsii/

[6] Oxford Nanopore Technologies, "MinION MkI: portable, real-time biological analyses," accessed December 9, 2015. [Online]. Available: https://www.nanoporetech.com/products-services/minion-mki

[7] "Oxford Nanopore Technologies," accessed November 28, 2015. [Online]. Available: https://www.nanoporetech.com

[8] Oxford Nanopore Technologies, "How it works: the MinION for nanopore DNA," accessed December 9, 2015. [Online]. Available: https://nanoporetech.com/science-technology/how-it-works

[9] M. Jain, I. T. Fiddes, K. H. Miga, H. E. Olsen, B. Paten, and M. Akeson, "Improved data analysis for the MinION nanopore sequencer," *Nature Methods*, vol. 12, no. 4, pp. 351–356, 04 2015. [Online]. Available: http://dx.doi.org/10.1038/nmeth.3290

[10] P. M. Ashton, S. Nair, T. Dallman, S. Rubino, W. Rabsch, S. Mwaigwisya, J. Wain, and J. O'Grady, "MinION nanopore sequencing identifies the position and structure of a bacterial antibiotic resistance island," *Nature Biotechnology*, vol. 33, no. 3, pp. 296–300, 03 2015. [Online]. Available: http://dx.doi.org/10.1038/nbt.3103

[11] D. Antipov, A. Korobeynikov, J. S. McLean, and P. A. Pevzner, "hybridSPAdes: an algorithm for hybrid assembly of short and long reads," *Bioinformatics*, November 2015.

[12] A. D. Prjibelski, I. Vasilinetc, A. Bankevich, A. Gurevich, T. Krivosheeva, S. Nurk, S. Pham, A. Korobeynikov, A. Lapidus, and P. A. Pevzner, "ExSPAnder: a universal repeat resolver for DNA fragment assembly," *Bioinformatics*, vol. 30, no. 12, pp. 293–301, June 2014.

[13] R. L. Warren, B. P. Vandervalk, S. J. Jones, and I. Birol, "LINKS: Scaffolding genome assemblies with kilobase-long nanopore reads," *bioRxiv*, 2015.

[14] "Mate Pair Sequencing," accessed November 28, 2015. [Online]. Available: http://www.illumina.com/technology/next-generation-sequencing/mate-pair-sequencing_assay.html

[15] 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

[16] 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

[17] A. Danial, "cloc - Counts Lines of Code," accessed November 25, 2015. [Online]. Available: http://github.com/AlDanial/cloc/

[18] A. Doring, D. Weese, T. Rausch, and K. Reinert, "SeqAn An efficient, generic C++ library for sequence analysis," *BMC Bioinformatics*, vol. 9, no. 1, p. 11, 2008. [Online]. Available: http://www.biomedcentral.com/1471-2105/9/11

[19] M. Čulinović, "CPPPOA," 2015, accessed November 25, 2015. [Online]. Available: https://github.com/mculinovic/cpppoa

[20] H. Li and R. Durbin, "Fast and accurate long-read alignment with Burrows-Wheeler transform," *Bioinformatics*, vol. 26, no. 5, pp. 589–595, March 2010.

[21] I. Sovic, M. Sikic, A. Wilm, S. N. Fenlon, S. Chen, and N. Nagarajan, "Fast and sensitive mapping of error-prone nanopore sequencing reads with GraphMap," *bioRxiv*, 2015.

[22] L. Torvalds and J. C. Hamano, "git - the stupid content tracker," accessed November 25, 2015. [Online]. Available: https://git-scm.com

[23] R. M. Stallman and GCC Developer Community, *Using The Gnu Compiler Collection*. Free Software Foundation, 2015. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/

[24] *GNU Make*. Free Software Foundation, 2015, accessed November 25, 2015. [Online]. Available: http://www.gnu.org/software/make/

[25] D. van Heesch, "Doxygen - Generate Documentation from Source Code," accessed November 25, 2015. [Online]. Available: http://www.stack.nl/~dimitri/doxygen/

[26] AT&T Labs Research and Contributors, "Graphviz," accessed November 25, 2015. [Online]. Available: http://graphviz.org

[27] "Escherichia coli str. K-12 substr. MG1655, complete genome," accessed December 6, 2015. [Online]. Available: http://www.ncbi.nlm.nih.gov/nuccore/U00096.3

[28] Pacific Biosciences, "E coli K12 MG1655 Resequencing," accessed December 8, 2015. [Online]. Available: https://github.com/PacificBiosciences/DevNet/wiki/E-coli-K12-MG1655-Resequencing

[29] Broad Institute, "Escherichia Coli str. K-12 substr. MG1655 Illumina Reads," accessed December 8, 2015. [Online]. Available: http://www.ncbi.nlm.nih.gov/sra/SRX131033

[30] N. Loman, "First SQK MAP 006 experiment," accessed December 8, 2015. [Online]. Available: http://lab.loman.net/2015/09/24/first-sqk-map-006-experiment/

[31] A. Gurevich, V. Saveliev, N. Vyahhi, and G. Tesler, “QUAST: quality assessment tool for genome assemblies,” *Bioinformatics*, vol. 29, no. 8, pp. 1072–1075, 2013. [Online]. Available: http://bioinformatics.oxfordjournals.org/content/29/8/1072.abstract

EAGLER - Eliminating Assembly Gaps by Long Extending Reads

Abstract

Next Generation Sequencing (NGS) is rapidly becoming a cornerstone of clinical medicine. The limiting factor of NGS technologies is the inability to reconstruct genomic regions with high repeat rates since a single read can span only up to 300 base pairs. The EAGLER scaffolder is a post-assembly tool aimed at closing gaps in the draft assemblies created by NGS assemblers such as SGA, ABySS and SPAdes. The scaffolder takes as input a draft genome and a set of long reads, either PacBio or Nanopore, and outputs a set of scaffolds. It has shown very promising results on the Escherichia Coli K12 MG1655 genome achieving an average identity of 98.96% using PacBio long reads with a 20x coverage.

Keywords: Scaffolding, NGS, Long Reads, PacBio, Oxford Nanopore

EAGLER - Popravljanje već sastavljenih genoma koristeći duga očitanja

Sažetak

Metode sekvenciranja druge generacije (NGS) ubrzanim tempom postaju svakodnevica kliničke medicine. Glavni nedostatak NGS tehnologija je nemogućnost rekonstrukcije dugačkih ponavljajućih regija genoma s obzirom na to da je svako pojedino očitanje dugo najviše 300 baza. EAGLER je alat namijenjen zatvaranju rupa u već sastavljenim genomima. Takvi genomi su rezultat modernih asemblera kao što su SGA, ABySS i SPAdes. Alat prima na ulaz sastavljeni genom i skup dugačkih očitanja, PacBio ili Nanopore, te kao rezultat vraća skup scaffolda. Rezultati testova na genomu Escherichia Coli K12 MG1655 pokazali su se vrlo obećavajućima. Koristeći PacBio očitanja pokrivenosti 20x postigao se prosječni identitet sekvence od 98.96%.

Ključne riječi: Sastavljanje genoma, Metode sekvenciranje druge generacije, Dugačka očitanja, PacBio, Oxford Nanopore