

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

MASTER THESIS No. 1570

Scaffolding Assembled Genomes with Long Reads

Ivan Krpelnik

Zagreb, lipanj 2018.

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

Zagreb, 2 March 2018

MASTER THESIS ASSIGNMENT No. 1570

Student: Ivan Krpelnik (0036478618)
Study: Computing
Profile: Computer Science

Title: **Scaffolding Assembled Genomes with Long Reads**

Description:

Long erroneous reads of third generation sequencing opened the door to more contiguous genome assemblies. Recently, researchers have shown that this is also possible by avoiding read error correction before assembly despite the high error rates. The resulting assemblies, especially for larger genomes, are still often fragmented either due to unbridged repeats or false overlaps caused by errors in the data. The goal of this work is to develop a tool which will improve fragmented assemblies by connecting disjoint regions using error corrected long reads which are not fully contained in the assembly. This should be achieved by fully spanning gaps with long reads or iteratively expanding sequences with multiple sequence alignment on both of their ends or using a hierarchical approach which combines the two before mentioned methods. The solution should be appropriated for parallel architectures and implemented in C++. The source code has 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 license.

Issue date: 16 March 2018
Submission date: 29 June 2018

Mentor:



Associate Professor Mile Šikić, PhD

Committee Chair:



Full Professor Siniša Srblijić, PhD

Committee Secretary:



Assistant Professor Tomislav Hrkać, PhD

DIPLOMSKI ZADATAK br. 1570

Pristupnik: **Ivan Krpelnik (0036478618)**
Studij: Računarstvo
Profil: Računarska znanost

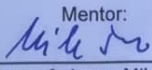
Zadatak: **Popunjavnaje rupa sastavljenih genoma pomoću dugačkih očitavanja**

Opis zadatka:


Dugačka greškovita očitavanja proizvedena trećom generacijom sekvenciranja omogućila su manju fragmentiranost sastavljenih genoma. Nedavno je pokazano da je sastavljanje isto moguće bez ispravljanja pogrešaka u očitanjima prije samog sastavljanja, usprkos velikom udjelu pogreške. Sastavljeni genomi, pogotovo onih od većih organizama, još uvijek su često fragmentirani i to zbog nepremošćenih ponavljajućih regija ili zbog lažnih preklapanja uzrokovanih pogreškama u podacima. Tema ovog rada je razvoj alata koji će poboljšati fragmentirane genome spajanjem nepovezanih regija pomoću ispravljenih očitavanja koja nisu u cijelosti sadržana u dotičnim genomima. Metoda koja to ostvaruje trebala bi ili premostiti rupe cijelim dugačkim očitanjima ili iterativno produljivati sekvence na oba kraja pomoću višestrukog poravnanja očitavanja ili koristeći hijerarhijski pristup koji kombinira prva dva pristupa. 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 GitHub pod jednu od OSI odobrenih licenci.

Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 29. lipnja 2018.

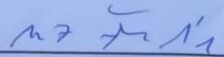
Mentor:


Izv. prof. dr. sc. Mile Šikić

Djelovođa:


Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za
diplomski rad profila:


Prof. dr. sc. Siniša Srblić

CONTENTS

1. Introduction	1
2. Scaffolding problem	2
3. Extension	4
3.1. Extension labeling	4
3.1.1. Extensions preprocessing	7
3.1.2. Circular contigs	7
3.2. Extension assembly	9
3.2.1. Partial order alignment	10
3.2.2. Extension assembly using spoa	10
3.2.3. Appending extensions	13
4. Bridging	15
4.1. Contig chains	18
4.2. Chains consuming	21
5. Implementation	23
5.1. Overview	23
5.2. External dependencies	23
5.2.1. Bioparser	23
5.2.2. SPOA	23
5.3. Code structure	24
6. Results	26
6.1. Artificial gaps	26
6.2. Pacbio bacterial datasets	30
7. Conclusion	34

List of Figures	35
List of Tables	36
Bibliography	37

1. Introduction

New sequencing methods are developed that greatly reduce the cost. The cost of the first sequencing and assembling of human genome, also known as the Human Genome Project, was up to 3 billion USD (7). Today, the cost dropped to just a few thousand USD. The drop in price allows more researches to build tools needed for sequence analysis and assemblies.

Next generation sequencing methods are high throughput methods that produce high short and accurate reads (8). Assembling these reads without the reference sequence, using *de novo* methods can lead to fragmented assemblies. Fragments will most likely have good quality, but because the reads are short, some regions might be impossible to assemble. For example, repeat regions that are longer than the reads would cause problems and fragmentation and there are tools being developed that help resolving these issues (1). Long reads assembling have these problems as well. False overlaps might cause fragmentation, since in the case when the assembler could assemble two different assemblies that it deems equal in quality, the assembler might stop connecting reads there as it has no way of telling which reads to choose next.

This thesis offers methods to bridge fragmented genomes with long reads. Fragments are extended using partial order alignment to create high quality extensions. When a read spans over two different fragments, those fragments are bridged into one fragment. The assemblies are made by the new fast *de novo* assemblers *Miniasm* (6) and *Rala* (9).

The overview of the chapters in this thesis is given below.

Chapter 2 introduces the problem of fragmented assemblies.

Chapter 3 describes the extension methods for the assembled fragments.

Chapter 4 describes the bridging method which assembles chained fragments.

Chapter 5 gives an overview of the implemented solution.

Chapter 6 shows the results of the implementation on artificial fragments of *e. coli* genome and real fragments produced by *Rala* assembler.

Chapter 7 gives a conclusion of this thesis.

2. Scaffolding problem

De novo assembly tools often cannot assemble the whole genome depending on its complexity. The data produced by this generation sequencing is prone to high error rates, so the resulting assemblies might be fragmented due to false overlaps. Another problem which is present in resulting assemblies is unbridged repeats - fragments (contigs) in the resulting assembly are often cut off at repeat regions.

An example of a fragmented assembly produced by *Rala* (?) is shown on the figure 2.1. The dotplot is created using the tool *GEPARD* (3).

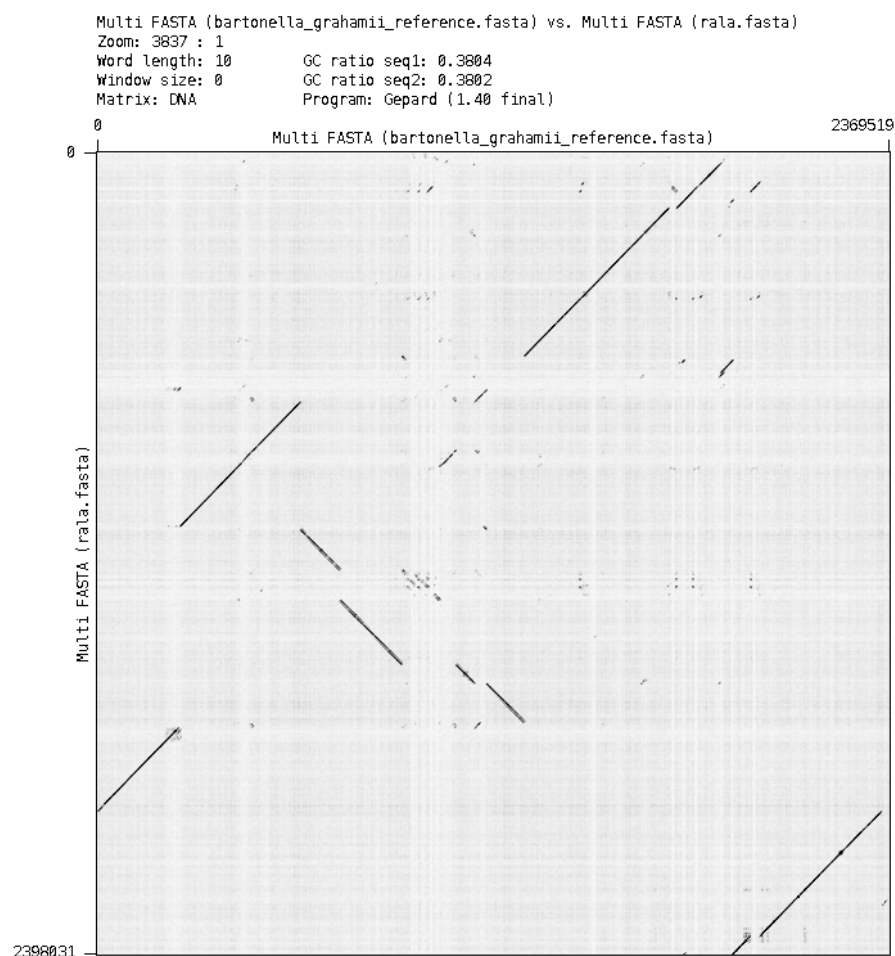


Figure 2.1: Rala assembly aligned to reference genome

The reference genome positions are on the abscissa and the assembly has its positions on the ordinate. It's evident that the assembly is fragmented compared to the reference and some of the produced contigs are in reverse complemented. Lines that are parallel to the main diagonal represent contigs which are on the same strand, meaning that the region on the abscissa and the region on the ordinate are aligned on the same strand. Lines that are perpendicular to the main diagonal represent contigs that are aligned with opposite strands - on the above figure, most of the contigs are in reverse complement of the reference sequence.

This paper offers a partial solution to the problem of fragmented assembly. There are two main ideas to solving the presented issue. The first is to bridge adjacent contigs using long reads that span over the missing region. For the regions that cannot be bridged yet, contigs are extended using reads that are mapped onto their suffix and prefix. This process is meant to be iterative, so the next iteration will try to bridge or extended the contigs from last iteration. The methods are described in greater detail in the following chapters.

3. Extension

The extension method aims to make current contigs longer by appending a consensus sequence of reads that are mapped to the contig's ends. Circular contigs and inner contigs in a chain of contigs are not extended. Recognition of circular contigs is described in the next section. Inner contigs in a chain are bridged by a single read that spans over the region between adjacent contigs. The quality of the bridged area will later be improved using *racon*.

3.1. Extension labeling

To find reads that extend assembled contigs, reads are first mapped using *minimap* onto contigs. Each read mapping made by *minimap* is labeled as a suffix, prefix or invalid extension. Extension of a contig is described with the following structure.

```
Extension {
    ExtensionType eType ,
    string strReadName ,
    string strTargetName ,
    ulong ulReadBegin ,
    ulong ulReadEnd ,
    ulong ulReadLength ,
    ulong ulTargetBegin ,
    ulong ulTargetEnd ,
    StrandType eStrand ,
    ulong ulTargetDelta
}
```

Listing 3.1: Extension structure

Part of the structure shown in 3.1 overlaps with the *paf* format produced by *minimap* and all of the members are populated from the *paf* line representing the extension. First of, *strReadName*, *strTargetName*, *ulReadBegin*, *ulReadEnd*, *ulReadLength*, *ulTargetBegin*, *ulTargetEnd* and *eStrand* are copied directly from the *paf* line for which this extension is made. Target and read positions represent the regions that are overlapping between the target sequence (contig) and the read. Extension type *eType* is determined based on these positions, sequence's length and the mapping strand and can be one of the following *SUFFIX*, *PREFIX*, *CONTAINED* or *INVALID*. When the type is determined as *SUFFIX* or *PREFIX*, the delta position to the closer ending on the target is cached in *ulTargetDelta*.

The following figure 3.1 shows an example of an extension that would be labeled as *INVALID*. The regions on the read that are not part of the overlap would not extend the contig as they are too short and the read's overhangs are almost the same length as the overlap. This read would be marked as *CONTAINED* if the overhangs were smaller compared to the overlap.

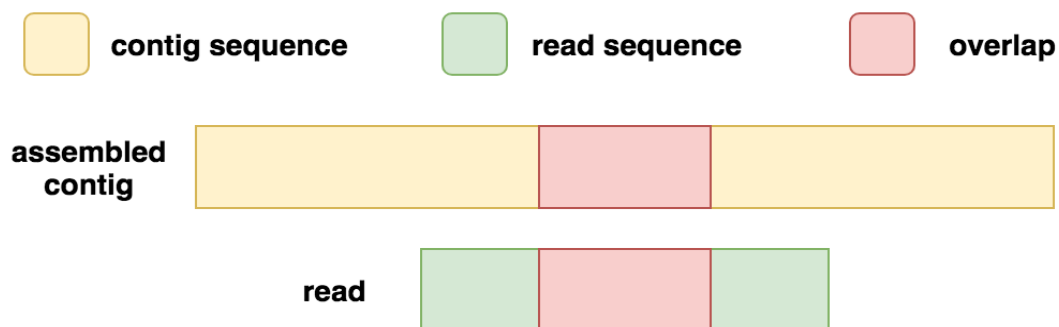


Figure 3.1: Invalid extension

Let's examine the other two types *SUFFIX* and *PREFIX*. Figures 3.2 and 3.3 show suffix and prefix extensions respectively. As it is seen on the figures, the reads would definitely extend the assembled contig. However, this condition alone is not enough to not label the extension as *INVALID*. If the overlapping area is not big enough compared to overhangs, than these extensions would also be labeled as *INVALID*.

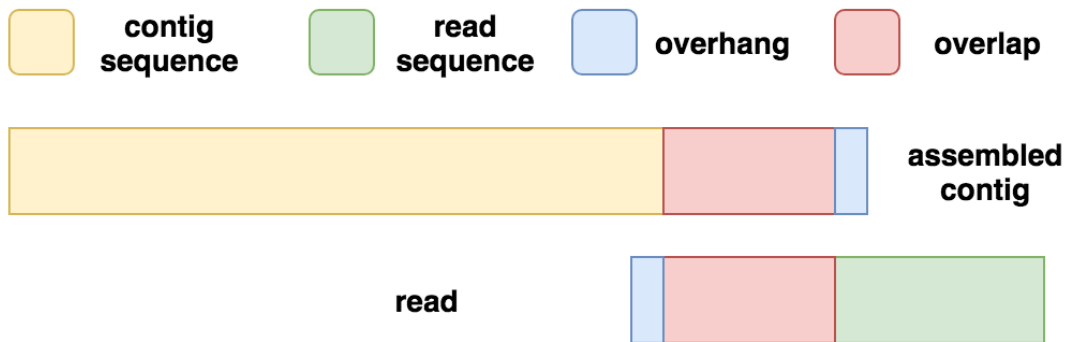


Figure 3.2: Suffix extension

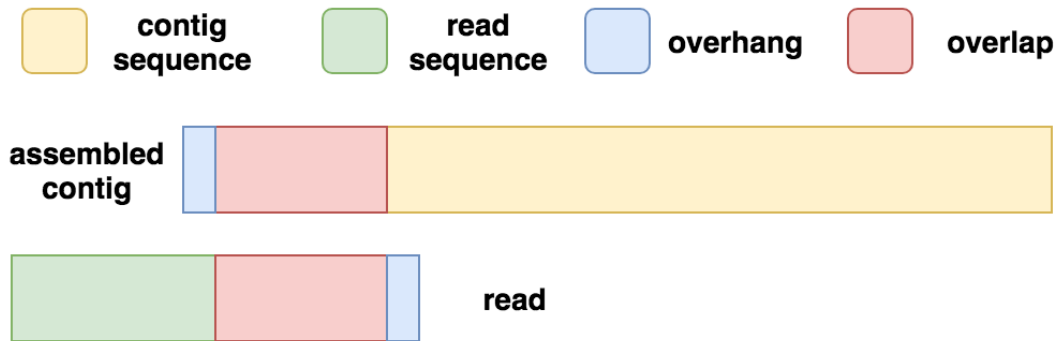


Figure 3.3: Prefix extension

A heuristic is made to discard some overlaps based on the ratio between overlap length and overhang length. Let's label the length of the read region shown in green color as *extensionLength*, the lengths of the overhangs *readOverhangLength* and *contigOverhangLength* and the lengths of the overlaps *readOverlapLength* and *contigOverlapLength*.

First, a maximums between overhang lengths and overlap lengths are determined.

$$\text{maxOverhang} = \text{max}(\text{readOverhangLength}, \text{contigOverhangLength})$$

$$\text{maxOverlap} = \text{max}(\text{readOverlapLength}, \text{contigOverlapLength})$$

Then, there are a two conditions that must be met for the extension to be valid. As mentioned, the read should extend the contig which translates to following expression, $\text{extensionLength} > \text{contigOverhangLength}$. The other condition to be satisfied is a heuristic to discard overlaps that are not meaningful, i.e. the overlap is too small

compared to overhangs.

$$\frac{\mathit{maxOverhang}}{\mathit{maxOverlaps}} < \mathit{Threshold} \quad (3.1)$$

The threshold used in the current implementation is 0.125.

3.1.1. Extensions preprocessing

Before building the actual extensions, for each contig 2 sets of extensions are built - *PREFIX* and *SUFFIX* sets. To ensure better quality extensions and reduce false positive rate, any read that was marked as *CONTAINED* will be removed.

Algorithm 1 shows how the extensions are filtered. First, a set of contained reads names is built and then, for each extension, we can check if the extension read was contained somewhere else.

Algorithm 1: Contained reads filtering

Input: *Extensions*

Output: *FilteredExtensions*

Filter \leftarrow {}

foreach *ext* \in *Extensions* **do**

if *ext*.*GetType()* == *CONTAINED* **then**
 | *Filter* \leftarrow *Filter* \cup {*ext*.*GetReadName()*}
 end

end

FilteredExtensions \leftarrow {}

foreach *ext* \in *Extensions* **do**

if *ext*.*GetReadName()* \notin *Filter* **then**
 | *FilteredExtensions* \leftarrow *FilteredExtensions* \cup *ext*
 end

end

return *FilteredExtensions*

3.1.2. Circular contigs

Some dna sequences are known to be circular, examples include plasmids and bacterial chromosomes on which the implementation is tested (2). The simple method to recognize circular contigs used in this paper relies on the fact that the same read should

map to both ends of the contig and it should be possible to make a bridge between the two ends. Figure 3.4 shows an example of a circular sequence and 3 different regions on which the sequence is broken and then straightened. All 3 straightened sequences essentially represent the same sequence.

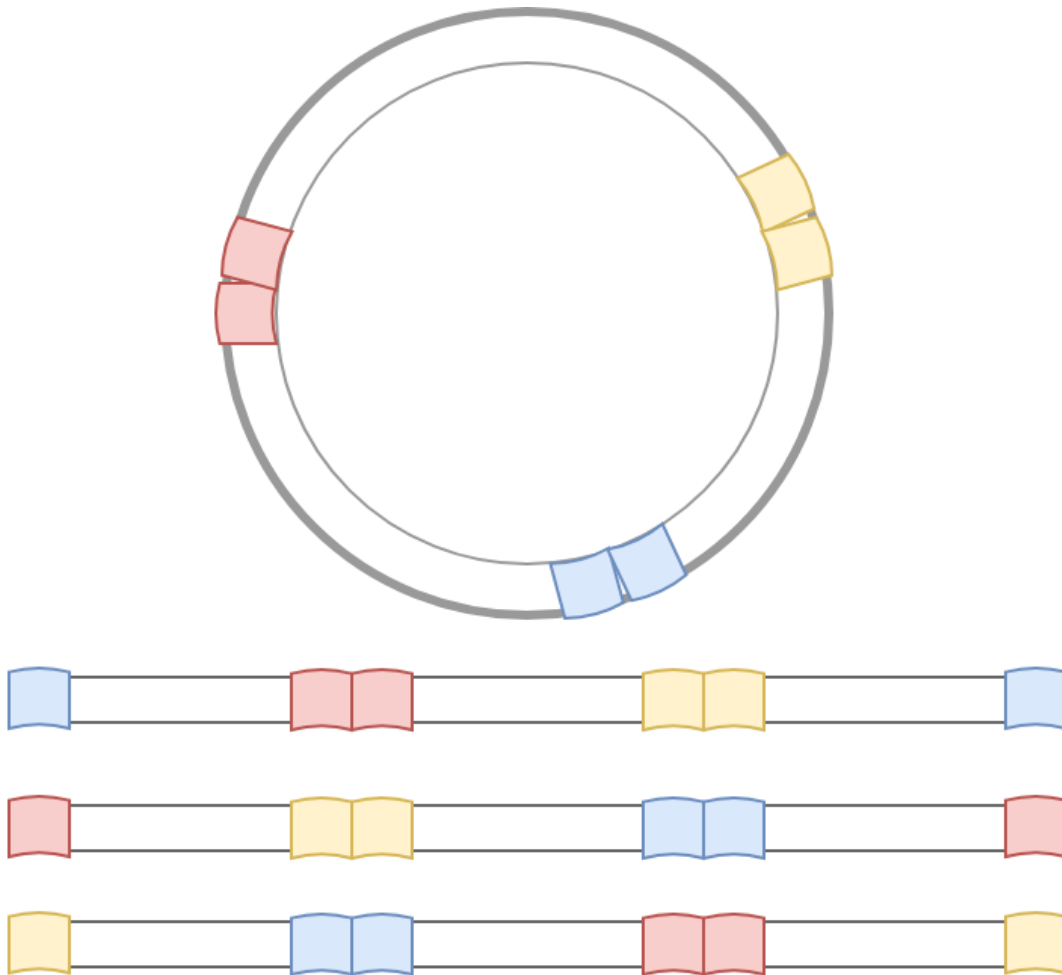


Figure 3.4: Circular sequence

There are two sets for each contig - suffix extension reads and prefix extension reads. What this check essentially does is an intersection of these two. If the intersection is not an empty set, the contig is circular. Chapter 4 explains this in greater detail.

Figure 3.5 shows an example of an assembled circular sequence. In the figure, it can be seen that a region at the beginning of the assembled sequence maps to the end region of the reference sequence. Since the prefix and the suffix of the reference can be bridged, this is a valid assembly.

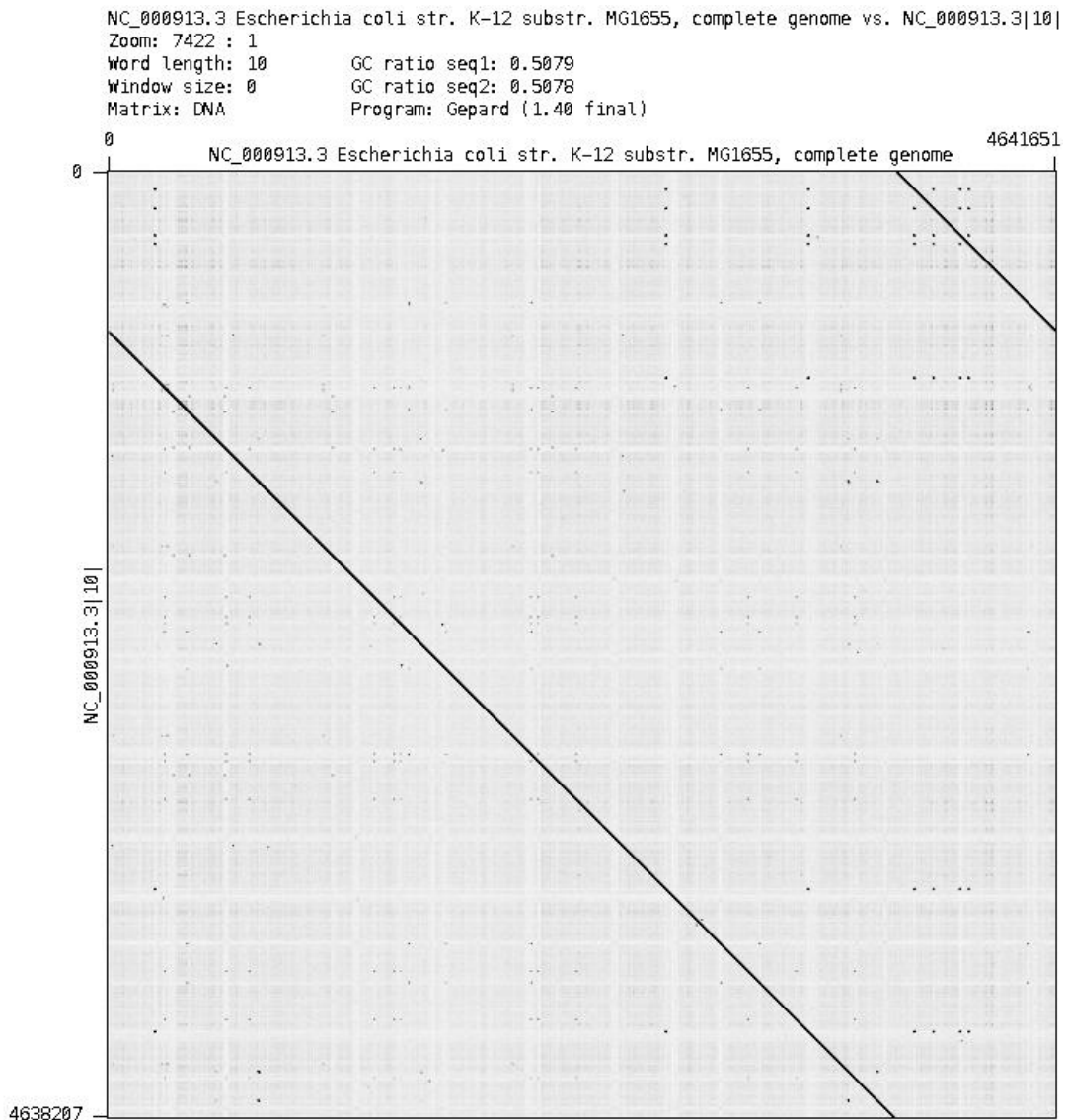


Figure 3.5: Circular sequence assembly

3.2. Extension assembly

For each contig and contig's end that is not bridged to another, an extension is built if there are valid extension reads on that end. The extension sequence is built using *spoa* which is an implementation of partial order alignment. All the reads that are mapped on the end for extension are aligned into the POA graph. The extension is then appended or prepended to the contig.

3.2.1. Partial order alignment

The problem of extension assembly can be solved as multiple sequence alignment and then generating a consensus sequence of the alignment (4). Two sequences of length L can be optimally aligned using dynamic programming in $O(L^2)$ and the algorithm can be extended for N sequences, having the time complexity of $O(L^N)$. While doing the alignment in $O(L^2)$ is acceptable for smaller sequences, the exponential time for larger number of sequences would not be practical. This problem can be solved more efficiently using partial order graphs, although there is no guarantee that the algorithm will find the optimal alignment.

The implementation used in this paper is SIMD POA *spoa* available at <https://github.com/rvaser/spoa>. *Spoa* offers multiple alignment modes - local (Smith-Waterman), global (Needleman-Wunsch) and semi-global of which local alignment is used here.

3.2.2. Extension assembly using *spoa*

Extensions are assembled using *spoa* as shown by algorithm 2. The extension is assembled from reads and a part of the contig. First, a maximum delta is found in the extensions and then the contig is cut off so only the part on which the reads are mapped is left together with the overhang. Since the order in which sequences are put in the POA graph matters, first sequence that is put in is the contig for which the extension is made as sort of a bias. The extensions are sorted by the overlap position on the contig sequence, so the reads that are aligned deeper on the contig come first. For each read, the overhang part is cut off and the rest is aligned with the graph. Finally, when the consensus is generated, it gets truncated depending on the coverage.

Algorithm 2: Extension using spoa

Input: *Extensions*, *IdToRead*, *strContigSeq*, *ExtensionType*

Output: *ulDelta*, *strConsensus*

engine \leftarrow *spoaCreateAlignmentEngine*()

graph \leftarrow *spoaCreateGraph*()

alignment \leftarrow *engine.align*(*strExt*, *graph*)

graph.addAlignment(*alignment*, *strExt*)

ulDelta \leftarrow *GetMaxDelta*(*Extensions*)

if *ExtensionType* == *SUFFIX* **then**

| *strContigSeq* \leftarrow *strContigSeq.substr*(0, *len*(*strContigSeq*) - *ulDelta*)

end

else

| *strContigSeq* \leftarrow *strContigSeq.substr*(*ulDelta*, *len*(*strContigSeq*))

end

alignment \leftarrow *engine.align*(*strExt*, *graph*)

graph.addAlignment(*alignment*, *strExt*)

foreach *ext* \in *Extensions* **do**

| *ulBegin* \leftarrow *ext.GetBeginPos*()

| *ulEnd* \leftarrow *ext.GetEndPos*()

| *seqRead* \leftarrow *IdToRead.find*(*ext.GetReadName*())

| *strExt* \leftarrow *seqRead.GetData*()*.substr*(*ulBegin*, *ulEnd*)

| **if** *ext.IsReverseComplement*() **then**

| | *strExt* \leftarrow *ReverseComplement*(*strExt*)

| **end**

| *alignment* \leftarrow *engine.align*(*strExt*, *graph*)

| *graph.addAlignment*(*alignment*, *strExt*)

end

strConsensus, *coverage* \leftarrow *graph.generateConsensus*(*coverage*)

TruncateConsensus(*strConsensus*, *coverage*, *len*(*Extensions*) + 1)

return *ulDelta*, *strConsensus*

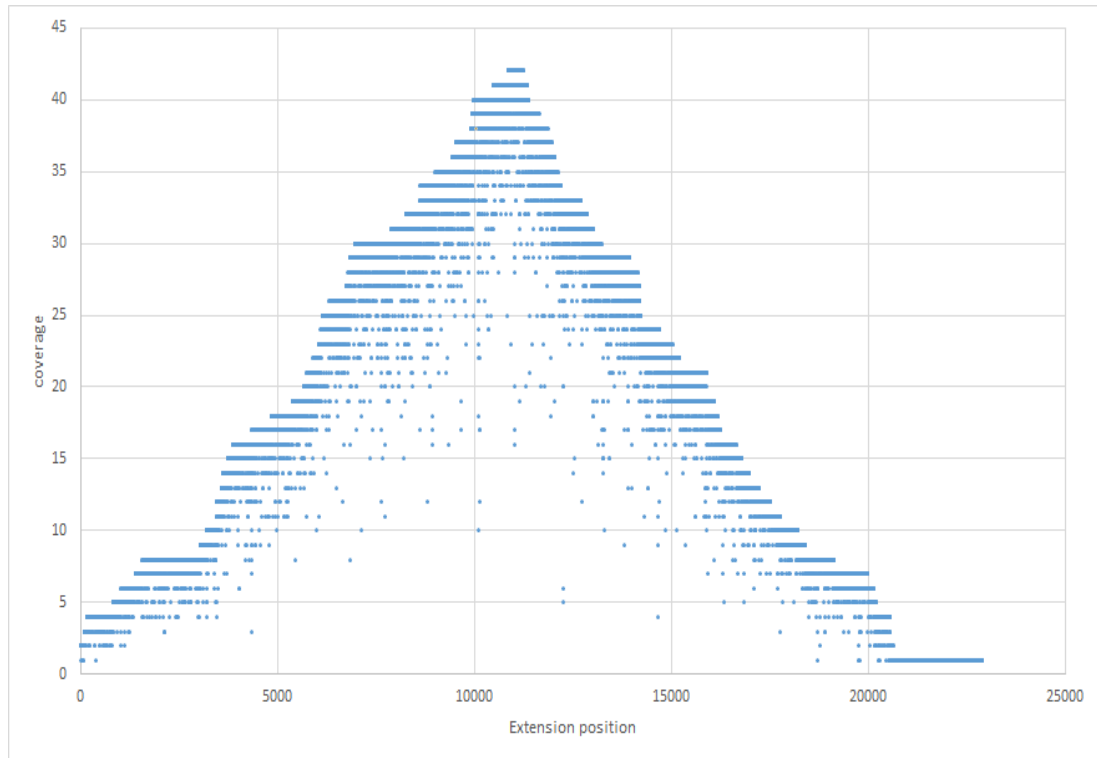


Figure 3.6: Dot plot representation of extension coverage produced by *spoa*

Figure 3.6 shows an example of a coverage graph for a generated extension on a suffix using *spoa*. When generating a consensus sequence, *spoa* returns a vector that holds coverage of each base in the consensus sequence. Figure 3.6 shows a dot plot representation of this vector. As this is an extension on a suffix of a contig, this consensus sequence would be appended to the contig, meaning that the beginning of the consensus sequence overlaps with the contig. Because of this, the beginning is considered to be accurate and the coverage of this part doesn't matter as much because the suffix of the contig is aligned in that region and the contig is considered accurate. The ending region of the extension isn't covered by contig's suffix, so when the coverage drops towards the end, the ending region of the generated sequence will not have high quality. Because of this, the ending should be cut off. This is shown in algorithm 3 which uses half maximum coverage as a threshold. So, when the coverage drops below the threshold, the rest of the extension is cut off.

The same applies for an extension generated on a prefix with only difference being that the extension has to be cut off somewhere at the beginning. This is because the prefix extension is prepended to the contig, meaning that the end of the extension overlaps with the prefix of the contig.

Algorithm 3: Extension consensus truncating

Input: *strConsensus*, *coverage*, *ulMaxCoverage*

Output: *strConsensus*

$Threshold \leftarrow ulMaxCoverage/2$

$ulPos \leftarrow len(coverage) - 1$

while $coverage[ulPos] < Threshold$ **do**
| $ulPos \leftarrow ulPos - 1$

end

return $strConsensus.substr(0, ulPos)$

3.2.3. Appending extensions

The extension produced by algorithm 2 is prepended or appended to the contig. Extension is done after bridging, since the strategy for bridging is greedy - as soon as it's possible to bridge 2 contigs, it is done so. Reads could be mapped again to the newly assembled contig and then the contigs could be extended using the extensions made from the new mapping data. Instead of repeating the mapping process, it is possible to use the old mapping data.

Positions of extensions on the original contigs are not necessarily the same on the assembled contig, but the delta from the contig's end, for which the extension is made, stays the same. This is because the contig ends used for extension are not used when bridging and they remain unchanged ends of marginal contigs. When extending a contig assembled by bridging we are effectively extending its marginal contigs. Before calling the extension method for the marginal contig, the unused end of the marginal contig should be determined - this is the extension type, either *PREFIX* or *SUFFIX*.

As shown in the figure 3.7, the extension is built as a consensus sequence from the read overlap regions and read extensions regions.

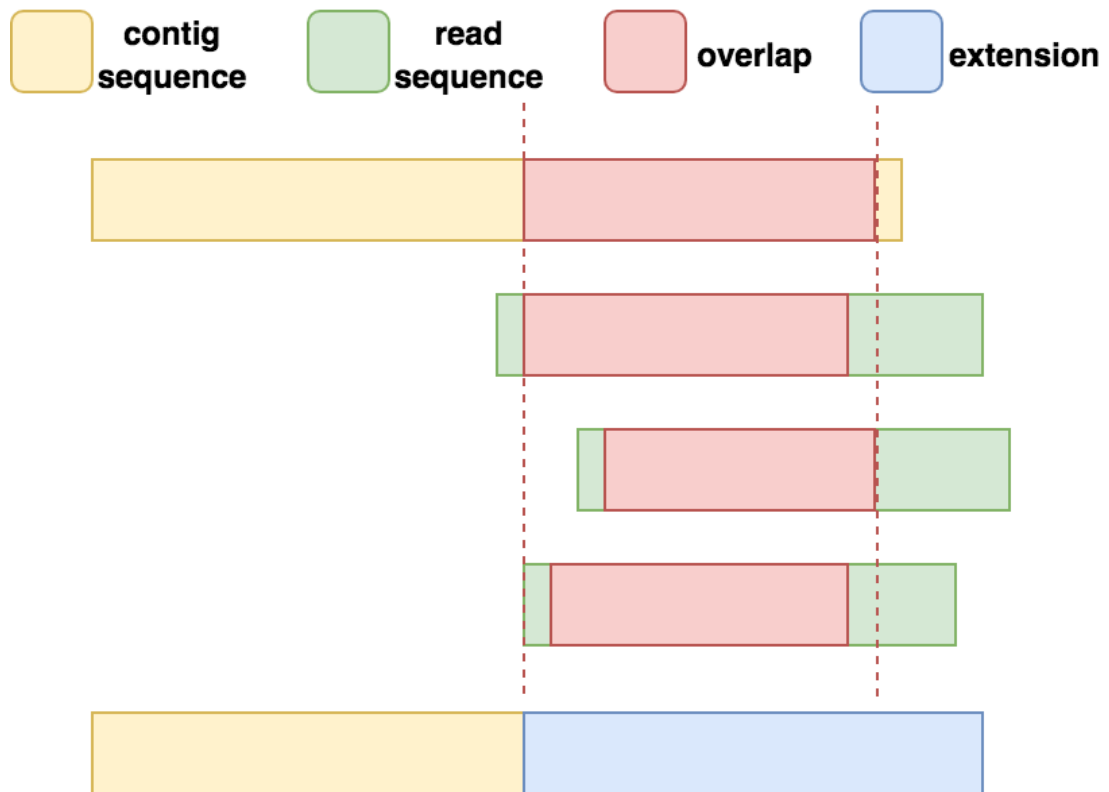


Figure 3.7: Extension appending

Depending on how the assembled contigs were bridged, some adjustments to the generated extension should be done. Marginal contigs might be bridged as reverse complements which means the generated extension should also be in reverse complement since the extension is generated relative to the contig in its original strand.

4. Bridging

Bridging is done between contigs that share extension reads. When the extension sets share the same reads and certain conditions are made, a bridge can be assembled between the two contigs. It is also possible that the extension sets of the same contig share reads in which case the contig is declared as circular.

For the two contigs for which the intersection of their extensions sets is not an empty set, it is possible to build a bridge if a read can be placed between the two contigs in the same strand with one of its ends being overlapped with the first contig and the other end with the second contig. This is shown on the figure 4.1. The reads orientation is denoted with the arrow and its two ends with letters *A* and *B*.

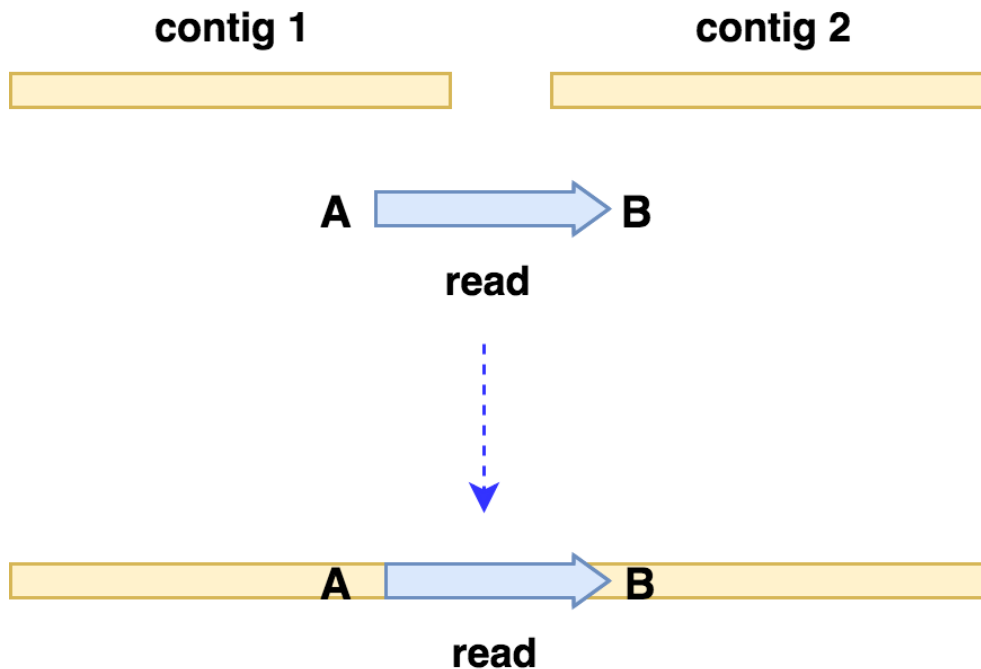


Figure 4.1: Bridge read

The two contigs are bridged with the read so that the regions where the read is mapped are cut off and the read is placed in between. The same is shown on a real

example on figures 4.2 and 4.3. Figure 4.2 shows 2 contigs which are not bridged with their ends barely overlapping compared to the reference sequence. The same region is shown again on figure 4.3, but this time the two contigs are bridged by a read into a single contig.

Depending on the read quality, the bridged region might not have more errors than the resto of the contig sequence. This issue can be resolved by polishing.

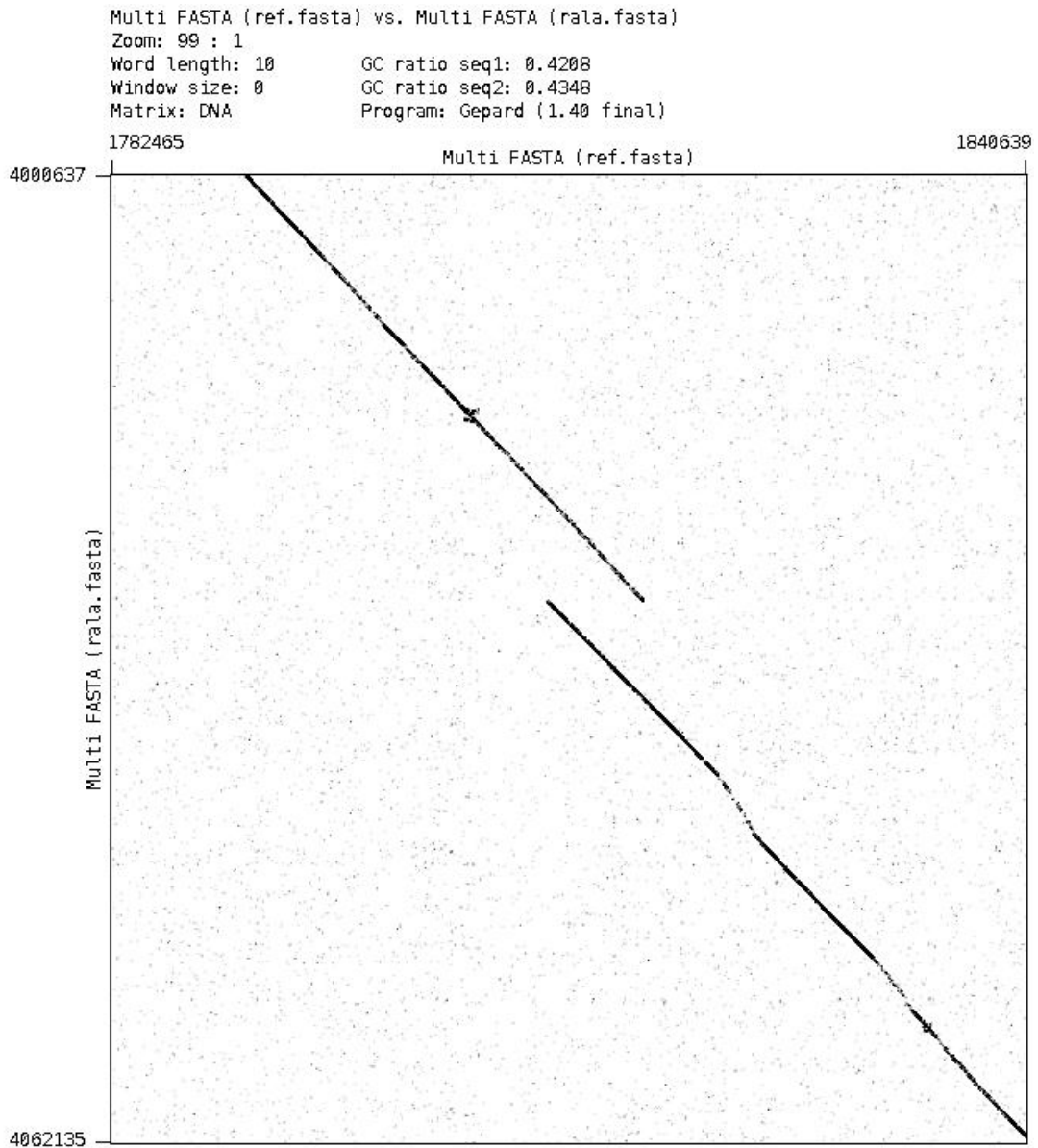


Figure 4.2: Unbridged contigs

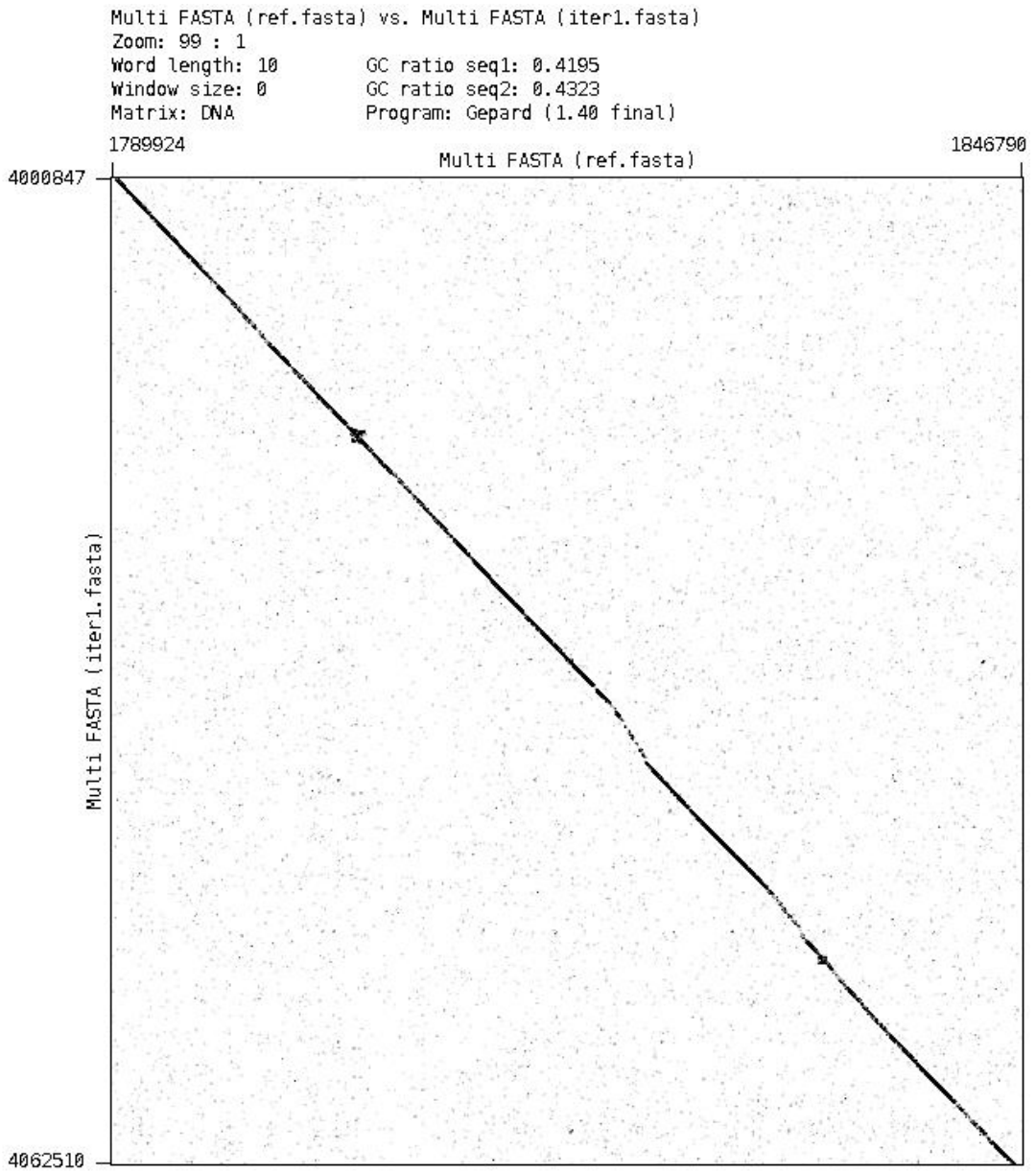


Figure 4.3: Bridged contigs

4.1. Contig chains

A graph is built where contigs are nodes and edges are bridges between contigs. For each two contigs, there is an edge if there is an intersection between their extension sets. Chains of contigs are then extracted from the graph for contigs to be bridged.

Let ext_A be an extension made on contig A and ext_B be an extension of the same read on contig B . Based on two extension properties $eType$ and $eStrand$, it can be determined if the read can be used as a bridge. One of the following two conditions must be satisfied.

1. $ext_A.eType \neq ext_B.eType \wedge ext_A.eStrand = ext_B.eStrand$
2. $ext_A.eType = ext_B.eType \wedge ext_A.eStrand \neq ext_B.eStrand$

This can be seen in the figure 4.4. In the picture, contig A is extended on its suffix. Contig B can be bridged onto contig A only if B is extended on its prefix by the read in original strand or on its suffix in reverse complement.

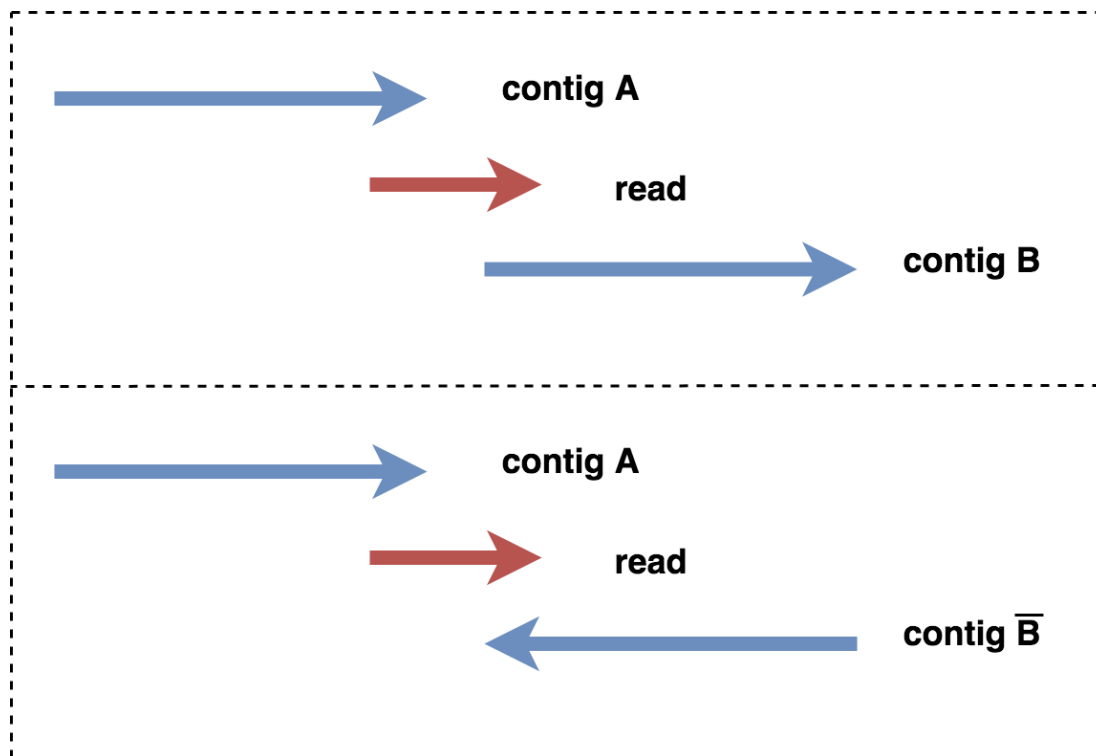


Figure 4.4: Bridge cases

Let's define the bridge graph by describing the edges more precisely. An edge is defined by the two contigs, extension types for both contigs and bridge extension set and it only exists if the aforementioned conditions are satisfied. Bridge extension set is the intersection of the extension sets of the two contigs for which the bridge is made. The read, from the bridge extension set, which will be used for bridging is taken at random.

Figure 4.5 shows an example of a bridge graph. For each contig there is a node with 2 exits, one for bridges on *SUFFIX* and one for bridges on *PREFIX*. Each edge is labeled with a number which denotes how many reads can bridge the two contigs. *Contig5* doesn't have any bridges to other contigs, so it would not be bridged, but rather extended on both sides if there are reads that map into its extensions. The graph also shows an example in which two contigs are bridged on the same prefix of *Contig1*. The goal is to create chains of contigs which will be bridged sequentially. Some of the edges will have to be cut, so that there is at most one bridge on some prefix or suffix.

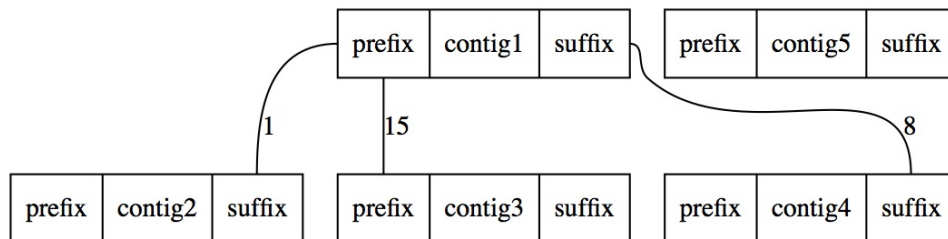


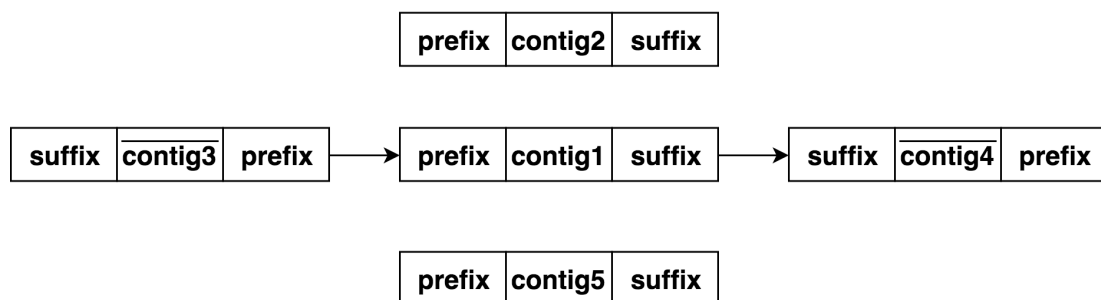
Figure 4.5: Bridge graph

To create chains, algorithm 4 is used. A greedy approach is used to select the edges for final chains. First, all the edges are sorted in descending order by the size of the bridge extension set - the number of reads that bridge the region. Edges are then selected greedily, each edge that connects two contigs on ends that were not used yet is selected in the final list of edges.

Algorithm 4: Edges selection

Input: *Edges***Output:** *SelectedEdges**sort(edges)**SelectedEdges* \leftarrow []*Used* \leftarrow {}**foreach** *edge* \in *Edges* **do** **if** (*edge.ctg1*, *edge.type1*) \notin *Used* \wedge (*edge.ctg2*, *edge.type2*) \notin *Used* **then** *Used* \leftarrow *Used* \cup {(*edge.ctg1*, *edge.type1*), (*edge.ctg2*, *edge.type2*)} *SelectedEdges.append(edge)* **end****end****return** *SelectedEdges*

The selected chains for the graph shown in figure 4.5 is shown by figure 4.6. The edge between *contig2* and *contig1* was cut off, since its weight is lower than the edge between *contig3* and *contig1*. The figure also shows a possible configuration in which the chains could be assembled. Here *contig3* and *contig4* are bridged onto *contig1* in reverse complement. In the extension step, the newly assembled contig will be extended on *contig3* suffix and *contig4* prefix. *Contig2* and *contig5* will be extended on both ends.

**Figure 4.6:** Chains graph

4.2. Chains consuming

Starting on a random node, the contig should be bridged along the chain on both its prefix and suffix. To simplify bridging, contigs that start the chains could be found and then the bridging process would go one way, depending on which end of the contig is the chain attached. Doing this, bridging would be simplified since contigs and bridge reads would always be appended to the last contig and there would be no need for prepending. But, finding the start of the chain would require some preprocessing before the chains are bridged. The chain might also be circular in which case, again a random contig would have to be selected.

The key observation here is that extending on contig's prefix is actually extending on contig's suffix in reverse complement. This is shown on figure 4.7. In the figure, contigs that are on the right would be appended to the contigs on their left. To use the one sided algorithm for bridging on both sides, it is enough to make a reverse complement of a contig before calling it for the other side.

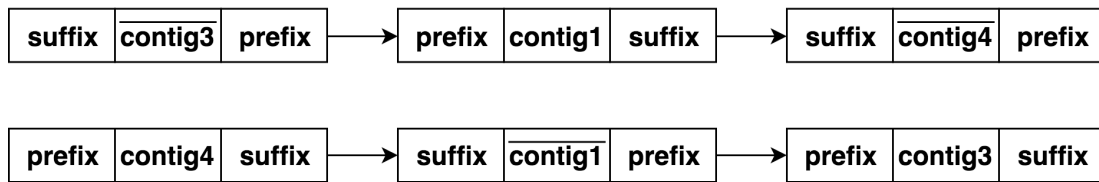


Figure 4.7: Bridging on a random contig

Algorithm 5 shows how the contigs in a chain are bridged by appending. When bridging on prefix, this method input should be reverse complement of the current contig. It can be seen that each read used as a bridge is appended, as well as the next contig in the chain. Contigs are cut off using *CutOff* method which removes *ulTargetDelta* (distance between the end of the contig and the overlap starting position) bases from the end at which the contig is bridged. *CalculateBridge* will cut off the bridging read so it can be inserted between the two contigs without overhangs. Finally, before appending the next contig, if the next contig is bridged on its suffix with the current contig, then the contig should be in reverse complement.

Algorithm 5: Chaining

Input: $Edges, currStrand, currContig$

Output: $chain$

$chain = currContig.GetData()$

while $HasNextEdge(edges, currStrand, currContig)$ **do**
 $edge \leftarrow GetNextEdge(edges, currStrand, currContig)$
 $bridgeRead \leftarrow edge.GetBridgeRead()$
 $currExt \leftarrow FindExtension(currContig, bridgeRead)$
 $CutOff(result, currExt.eType, currExt.ulTargetDelta)$
 if $currExt.eStrand \neq currStrand$ **then**
 $ReverseComplement(bridgeRead)$
 end
 $nextContig \leftarrow edge.GetNextContig()$
 $nextExt \leftarrow FindExtension(nextContig, bridgeRead)$
 $bridge \leftarrow CalculateBridge(currExt, nextExt, bridgeRead)$
 $chain.append(bridge)$
 $CutOff(nextContig, nextExt.eType, nextExt.ulTargetDelta)$
 if $nextExt.eType == SUFFIX$ **then**
 $ReverseComplement(nextContig)$
 $currStrand \leftarrow -$
 end
 else
 $currStrand \leftarrow +$
 end
 $chain.append(nextContig.GetData())$
 $currContig \leftarrow nextContig$

end

return $chain$

5. Implementation

This chapter gives an overview of this thesis' implementation. It explains the external dependencies, the code structure and gives instructions on how to clone and use the project.

5.1. Overview

The project is written in C++ under the C++14 standard. The algorithms are implemented in the *ezra* namespace as a possible extension for *Ra* - rapid assembler available on <https://github.com/rvaser/ra>.

The project is available at <https://gitlab.com/Krpa/ezra>.

5.2. External dependencies

External dependencies are under *vendor/* directory and will be downloaded automatically if *--recursive* flag is passed when cloning the repository.

5.2.1. Bioparser

Bioparser is a C++ implementation of parsers for formats FASTA, FASTQ, MHAP, PAF and SAM. *Bioparser* is written by Robert Vaser and it is available on <https://github.com/rvaser/bioparser>. The program developed in this thesis needs to read FASTA or FASTQ files which store contigs and reads and PAF file which contains mapping data.

5.2.2. SPOA

SPOA is a C++ SIMD (Single instruction, multiple data) (10) implementation of the partial order alignment algorithm described in (4) and (5) written by Robert Vaser.

The project is available on <https://github.com/rvaser/spoa>. It supports multiple alignment modes local (Smith-Waterman), global (Needleman-Wunsch) and semi-global alignment (overlap) of which local alignment is used in this thesis when generating extensions.

5.3. Code structure

To use the developed library, the header file *Bridger.h* should be included. Figure 5.1 shows the dependency tree. Each component is described below:

- *Bridger.cpp / Bridger.h* - wrapper around the implemented algorithms. It holds the data about reads, contigs and mappings read from provided files and it offers methods to execute the implemented algorithms.
- *Graph.cpp / Graph.h* - contains the graph edge definition and implementation of graph algorithms as described in chapters 3 and 4.
- *Parser.cpp / Parser.h* - wrapper around *Bioparser*, does some preprocessing to exclude contained reads from the extension sets.
- *Extension.cpp / Extension.h* - wrapper around the extensions. This is used when reading PAF mapping data with *Bioparser*. Extension is created for each line in the PAF file with the appropriate extension type. Extensions are filtered in the preprocessing phase and later used by graph algorithms.
- *Sequence.cpp / Sequence.h* - wrapper around sequence data, used when reading FASTA/FASTQ files with *Bioparser*.
- *Types.h* - contains declaration of maps that are used to store sequences and extensions.

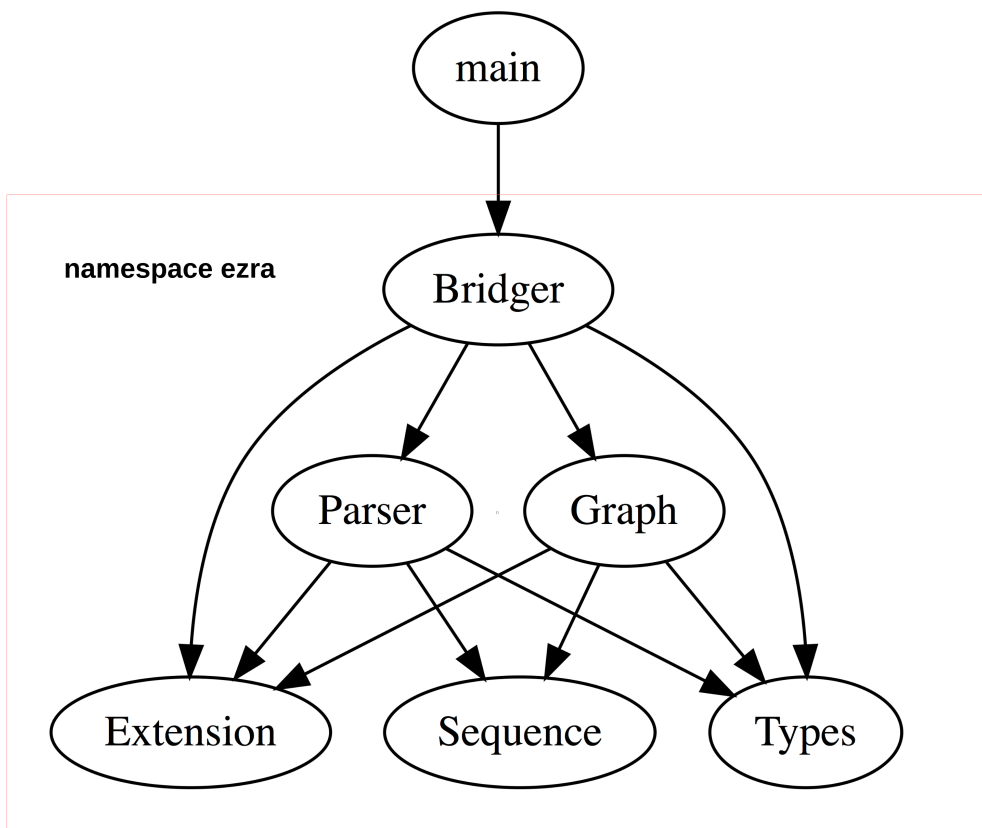


Figure 5.1: Dependency graph

6. Results

Ezra was tested on multiple bacterial datasets. First, few artificial datasets were created by cutting the e. coli reference genome to get the fragmented assembly. These fragments are easily connected since the fragments don't have errors. The second dataset consists of multiple Pacific Biosciences next generation sequencing bacterial datasets for which some of the reference sequences are known and some are not. The source of all PacBio datasets is Sanger institute <https://www.sanger.ac.uk/>.

6.1. Artificial gaps

The gaps are generated by randomly cutting the reference genome of Escherichia Coli. The Python script which does the cutting, creates random length gaps in a provided interval. Some of the generated contigs then reversed and complemented.

Number of reads	Median read length	Average read length
25483	9341	9755

Table 6.1: Reads statistics

Figure 6.1 shows an example where the reference genome was cut into 21 contig of which some are in reverse complement. The figure shows how these contigs align to the reference genome.

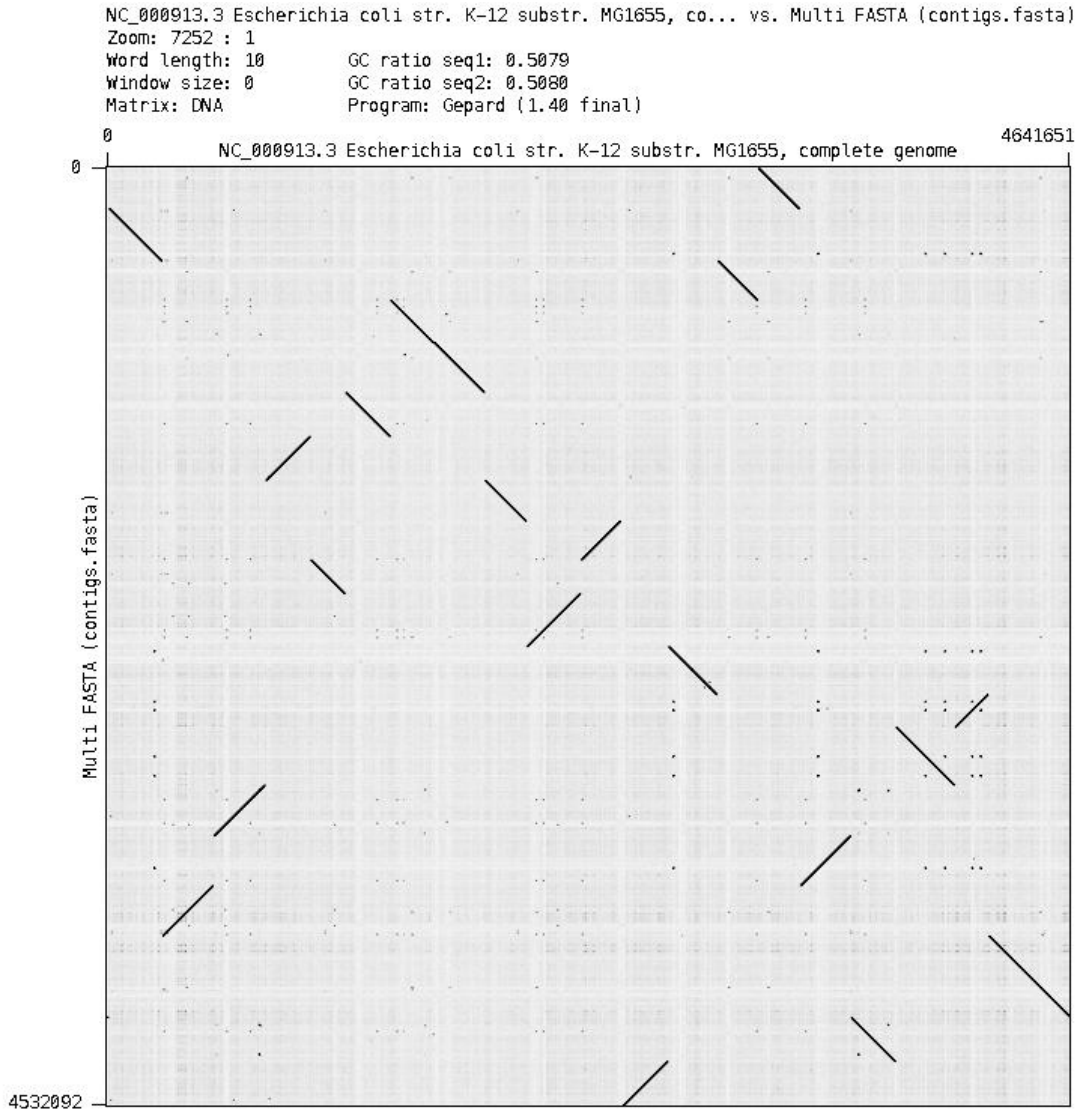


Figure 6.1: 21 contigs

Figure 6.2 shows all the contigs bridged together. Bridging was done in one iteration, since the gaps between contigs were small enough that they could be bridged by a single read. Reads statistics are shown by table 6.1. The gaps between the contigs are between 3000 and 6000 bases, while the average read length is a lot bigger.

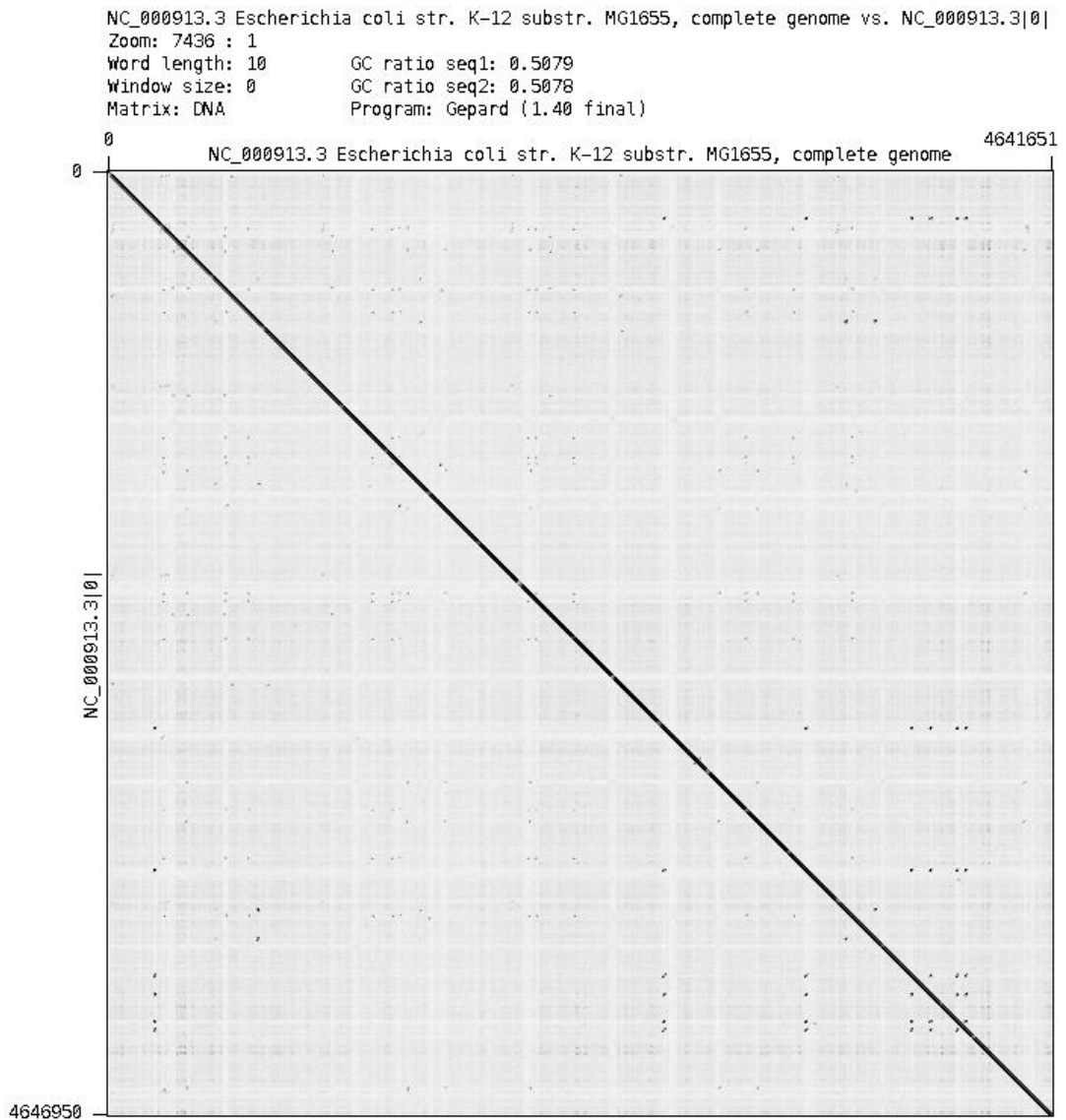


Figure 6.2: Bridged contigs

Finally figure 6.3 shows a gap closed by a read. The region has lower quality, since it is closed by one read without generating a consensus from all of the reads that span over. This bridge can be further improved by polishing.

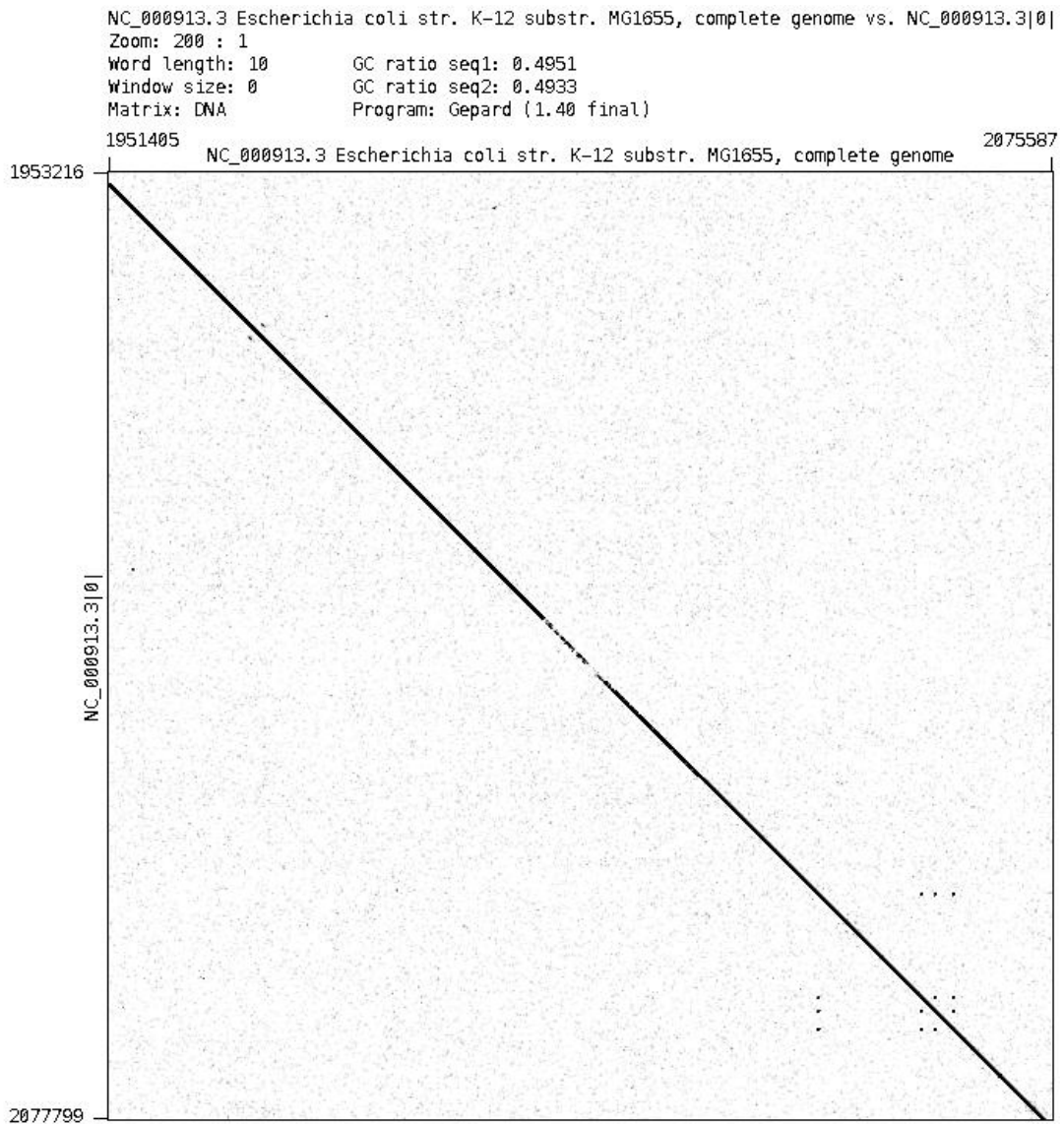


Figure 6.3: Bridged region

Previous figures are an example of a dataset that tested the bridging algorithm. Extensions were tested in a similar manner. Datasets which had gaps longer than reads were generated. Such dataset would then be consumed by Ezra and the generated extension were checked against the gaps. After enough iterations, gaps would become small enough to bridge them.

6.2. Pacbio bacterial datasets

Table 6.2 shows how Ezra reduced the number of contigs outputted by Rala. In most cases, the number of contigs was reduced. All of the datasets for which the reference genome was present were checked by aligning the contigs to the reference genome. In all of the cases contigs bridged by Ezra were true positives. The last column shows whether a reference genome was present for a certain dataset, + denoting presence of a reference genome.

As shown by the table 6.2, most of the datasets needed only one iteration to connect the contigs. This means that bridges were formed without extensions in most cases.

For one of the datasets, NCTC10183, ezra produced one contig and although the assembled sequence wasn't compared to a reference genome, interestingly the assembled sequence was circular and was aligned to assembly generated by miniasm.

Another interesting case was dataset NCTC8251. The contigs assembled by rala were overlapping. But the ends of the contigs had too many errors. After cutting off the ends, contigs were connected after two ezra iterations. This leads to an interesting idea for an upgrade to the tool where ends of contigs could be cut off based on their quality - for example, if there aren't any extensions mapping there.

Dataset	Rala contigs	Ezra contigs	Reference present	Iterations
NCTC11801	7	3	+	1
NCTC13482	4	3	+	1
NCTC8531	3	3	+	1
NCTC13277	3	1	+	1
NCTC8251	4	2	+	1
NCTC2611	7	3	+	2
NCTC11951	6	2	+	1
NCTC12860	6	4	+	1
NCTC10183	3	1	-	1
NCTC11180	4	3	-	1
NCTC11995	3	2	+	2
NCTC10696	9	4	-	1
NCTC7944	2	1	+	1
NCTC13769	2	1	-	1

Table 6.2: Contigs statistics

Table 6.3 shows lengths of longest contigs for each dataset that had a reference genome. This shows that even though some datasets had multiple contigs assembled, for some, the longest contig would cover the whole or most of the reference genome.

Dataset	Rala	Ezra	Reference
NCTC11801	1572304	4712531	4538470
NCTC13482	1863978	2286880	2172484
NCTC8531	2352906	2353897	2810675
NCTC13277	1889553	3092960	2821361
NCTC8251	1572304	2304818	2199877
NCTC2611	1866888	5672131	6296436
NCTC11951	724431	1124249	1641481
NCTC12860	1102801	1102801	2341328
NCTC11995	1286594	2184421	2271840
NCTC7944	1521734	2773590	2675240

Table 6.3: Max contig lengths

Assemblies are often cut around repeat regions. Figure 6.4 shows an example of an unbridged repeat region which was then successfully bridged by Ezra as shown in figure 6.5. The dataset in this example is NCTC7944.

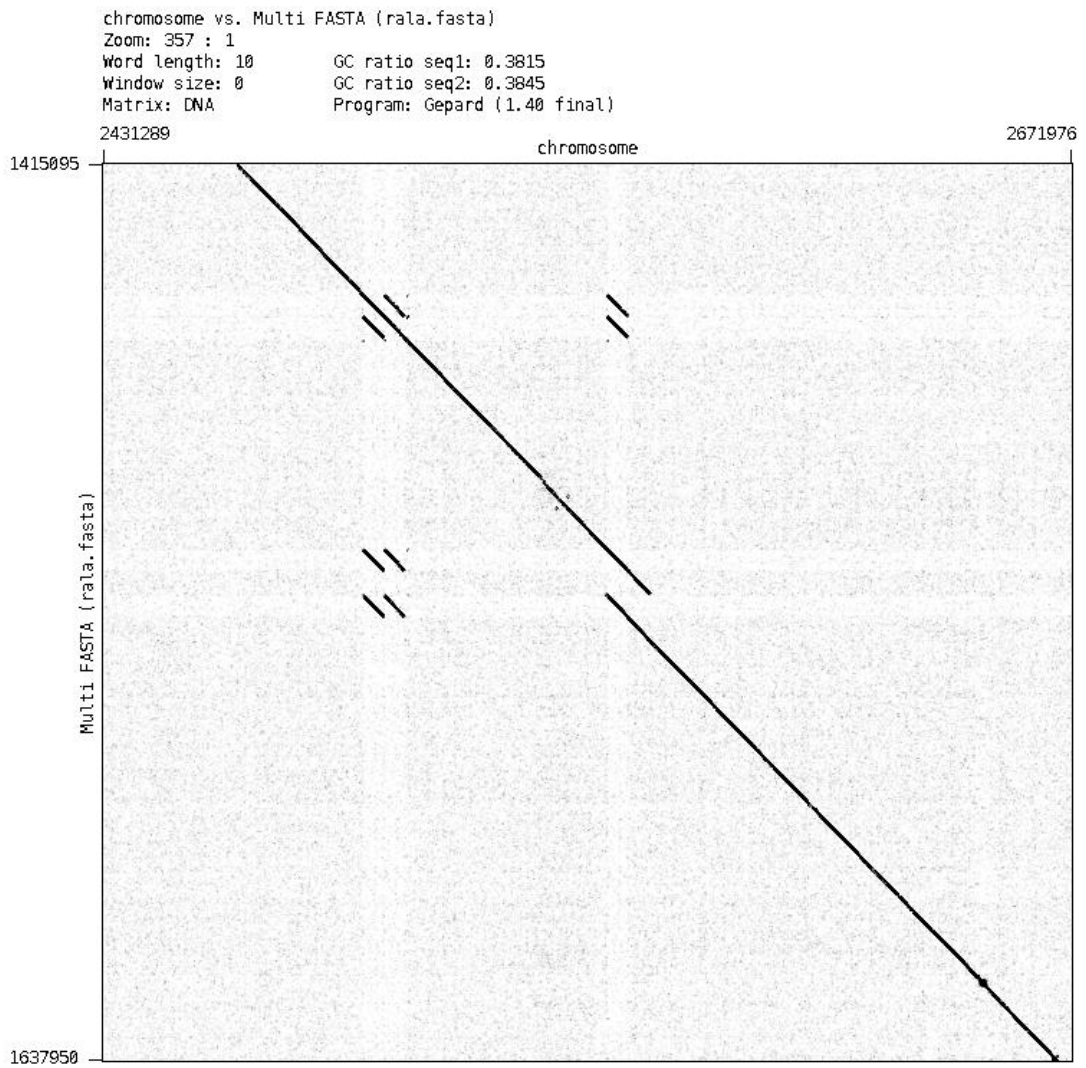


Figure 6.4: Unbridged repeat region

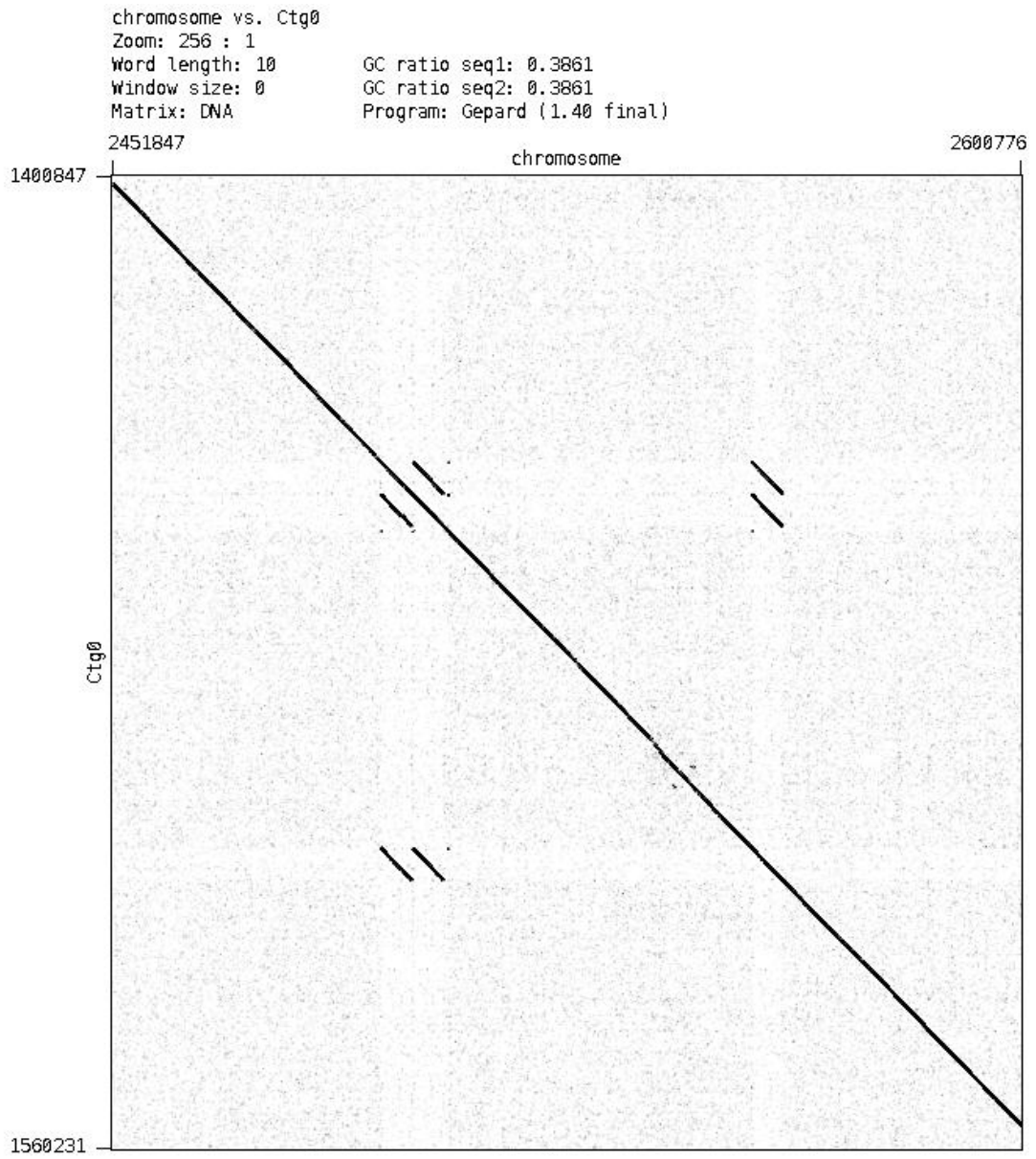


Figure 6.5: Bridged repeat region

7. Conclusion

The tool that was created in this thesis aims to connect fragmented genomes produced by de novo assemblers. As shown in the results chapter, developed methods were proved to be able to do that. Not all of the fragments were successfully connected to cover the whole reference genome, but these methods bring us a step closer to producing whole assemblies.

It is shown that gap regions can be bridged with long reads when they span over the region. The extension method also proved to be useful when the gaps were too big to be bridged in which case the tool ran for multiple iterations. Using *spoa* the extension methods built high quality extensions.

Some interesting ideas to explore include cutting off the ends of fragments and then generating extensions to create better quality ends. This is shown on one of the datasets which after cutting was bridged successfully.

LIST OF FIGURES

2.1. Rala assembly aligned to reference genome	2
3.1. Invalid extension	5
3.2. Suffix extension	6
3.3. Prefix extension	6
3.4. Circular sequence	8
3.5. Circular sequence assembly	9
3.6. Dot plot representation of extension coverage produced by spoa . . .	12
3.7. Extension appending	14
4.1. Bridge read	15
4.2. Unbridged contigs	16
4.3. Bridged contigs	17
4.4. Bridge cases	18
4.5. Bridge graph	19
4.6. Chains graph	20
4.7. Bridging on a random contig	21
5.1. Dependency graph	25
6.1. 21 contigs	27
6.2. Bridged contigs	28
6.3. Bridged region	29
6.4. Unbridged repeat region	32
6.5. Bridged repeat region	33

LIST OF TABLES

6.1. Reads statistics	26
6.2. Contigs statistics	30
6.3. Max contig lengths	31

BIBLIOGRAPHY

- [1] Pavel A Pevzner, Paul A Pevzner, Haixu Tang, i Glenn Tesler. De novo repeat classification and fragment assembly. 14:1786–96, 10 2004.
- [2] Samuel Karlin, Luciano Brocchieri, Allan Campbell, Martha Cyert, i Jan Mrázek. Genomic and proteomic comparisons between bacterial and archaeal genomes and related comparisons with the yeast and fly genomes. *Proceedings of the National Academy of Sciences*, 102(20):7309–7314, 2005. ISSN 0027-8424. doi: 10.1073/pnas.0502314102. URL <http://www.pnas.org/content/102/20/7309>.
- [3] Jan Krumsiek, Roland Arnold, i Thomas Rattei. Gepard: a rapid and sensitive tool for creating dotplots on genome scale. *Bioinformatics*, 23(8):1026–1028, 2007. doi: 10.1093/bioinformatics/btm039. URL <http://dx.doi.org/10.1093/bioinformatics/btm039>.
- [4] Christopher Lee. Generating consensus sequences from partial order multiple sequence alignment graphs. 19:999–1008, 06 2003.
- [5] Christopher Lee, Catherine Grasso, i Mark F Sharlow. Multiple sequence alignment using partial order graphs. 18:452–64, 04 2002.
- [6] Heng Li. Miniasm. <https://github.com/lh3/miniasm>.
- [7] NHGRI. The cost of sequencing a human genome. <https://www.genome.gov/sequencingcosts/>.
- [8] Ji Hanlee Shendure Jay. Next-generation dna sequencing. *Nat Biotech*, 26, 2008.
- [9] Robert Vaser. rala. <https://github.com/rvaser/rala>.
- [10] Wikipedia. Simd. <https://en.wikipedia.org/wiki/SIMD>.

Scaffolding Assembled Genomes with Long Reads

Abstract

In this thesis, a tool for contig extension and gap closing was implemented in C++. The tool is named Ezra and could be used for extension and bridging by de novo assembler Rala. The extension and bridging methods are meant to be run in iterations, repeating the process while it produces valid extensions and bridges.

The tool was tested on artificial fragments made by randomly cutting *e. coli* genome and on fragmented assembly produced by Rala on Pacbio bacterial datasets. The results show that this method is valid for reducing the number of contigs produced by Rala. The source code is available at <https://gitlab.com/Krpa/ezra>.

Keywords: Scaffolding, Assembly, Assemblies, Genome, Long reads

Popunjavanje rupa sastavljenih genoma pomoću dugačkih očitavanja

Sažetak

U ovom radu implementiran je alat za produljivanje i povezivanje sastavljenih slijedova u programskom jeziku C++. Alat je nazvan Ezra i potencijalno bi se mogao koristiti za produljivanje i povezivanje u alatu za de novo sastavljanje Rala. Metode produljivanja i povezivanja bi se trebale izvršavati iterativno, ponavljajući proces sve dok ima valjanih produljenja i povezivanja.

Alat je testiran na umjetno generiranim slijedovima napravljenih nasumičnim rezanjem genoma *e. coli* i na rascjepanim slijedovima sastavljenim alatom Rala nad Pacbio bakterijskim podatkovnim skupovima. Rezultati testiranja pokazuju da je ova metoda valjana za povezivanje slijedova sastavljenih Ralom. Izvorni kod dostupan je na <https://gitlab.com/Krpa/ezra>.

Ključne riječi: Povezivanje fragmenata, Sastavljanje genoma, Genom, Dugačka očitavanja