UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR THESIS No. 6900

DE NOVO DIPLOID ASSEMBLY USING THIRD-GENERATION SEQUENCING DATA

Roman Yatsukha

Zagreb,

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

BACHELOR THESIS No. 6900

DE NOVO DIPLOID ASSEMBLY USING THIRD-GENERATION SEQUENCING DATA

Roman Yatsukha

Zagreb,

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Zagreb, 13 March 2020

BACHELOR THESIS ASSIGNMENT No. 6900

Student: **Roman Yatsukha (0036507035)**

Study: Computing

Module: Computer Science

Mentor: prof. Mile Šikić

Title: **De Novo Diploid Assembly Using Third-Generation Sequencing Data**

Description:

Third-generation sequencing technologies facilitated the genome assembly problem significantly due to their ability to read considerably longer fragments than their predecessors. The drawback is the high error rate present in such fragments, although algorithms were adapted quickly to tolerate it. The majority of de novo assemblers were designed on smaller genomes and work well on larger eukaryotic organisms, but they create haploid representations of the genome regardless of the ploidy. The separation of genetic material from each parent results in knowledge of the complete genotype - all variant forms of all genes. Therefore, different methods should be applied in the assembly process to achieve better reconstructions. The main goal of this thesis is to adapt Raven, a de novo assembler for long erroneous sequencing data, for diploid organisms sequenced with third-generation sequencing technologies. The solution should be appropriated for parallel architectures and implemented in C++. The source code has to be well 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.

Submission date: 12 June 2020

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

ZAVRŠNI ZADATAK br. 6900

Pristupnik: **Roman Yatsukha (0036507035)**

Studij: Računarstvo

Modul: Računarska znanost

Mentor: prof. dr. sc. Mile Šikić

Zadatak: **De novo sastavljanje diploidnih genoma koristeći tehnologiju za očitanje DNA treće generacije**

Opis zadatka:

Treća generacija tehnologija za očitanje DNA je značajno olakšala sastavljanje genome uslijed mogućnosti očitanja značajno duljih fragmenata od prethodnih tehnologija. Nedostatak je visok postotak pogreške takvih fragmenata, iako su postojeći algoritmi brzo adaptirani da mogu tolerirati tu razinu pogreške. Većina metoda za de novo sastavljanje genoma je dizajnirana na manjim genomima i dobro radi na većim eurokariotskim organizmima, ali one kreiraju haploidne reprezentacije genoma neovisno o ploidnosti. Odvajanje genetičkog materijala svakog od roditelja rezultira poznavanjem ukupnog genotipa - svim prisutnih varijanti u svim genima. Stoga je potrebno primijeniti drugačije metode u procesu sastavljanja za postizanje bolje rekonstrukcije. Cilj ovoga rada je prilagoditi Raven, de novo metodu za sastavljanje genoma koja koristi dugačka greškovita očitanja, za diploide organizime čija je DNA očitana tehnologijama treće generacije. Rješenje mora biti pogodno za paralelnu arhitekturu i implementirano u jeziku C++. Izvorni kod treba biti iscrpno dokumentiran koristeći komentare i slijediti Google C++ Style Guide kada je to moguće. Cijeli programski proizvod potrebno je postaviti na GitHub pod jednu od OSI odabranih licenci.

Rok za predaju rada: 12. lipnja 2020.

Table of contents

1. Introduction **1**

2. Concepts **2**

3. Problem setup **3**

4. Data simulation **4**

5. Algorithms for fragment separation based on haplotype defining features **8**

5.1. SNP detection 8

5.2. Minimum fragment removal 11

5.2.1. Memoization 14

5.2.2. Odd cycles in undirected graphs 17

5.3. Fragment separation 19

6. Integrating changes into Raven **22**

7. Conclusion **24**

Literature **25**

1. Introduction

Third-generation sequencing (TGS) technologies have a multitude of advantages compared to previous technologies, providing longer reads and faster sequencing, while requiring less preparation of the sample. The drawback is the higher error rate compared to previous generations of sequencing technologies. The longer read lengths further enhance the capabilities of modern *de novo* assemblers enabling haplotype inference and whole chromosome phasing, however some additional computation is required to correct errors due to higher error rates [1, 2].

Even though haplotype assembly is more approachable due to TGS, only a small subset of existing *de novo* assemblers support it, with a few standalone tools also emerging [3].

In this work we discuss the problems of diploid assembly: haplotype assembly of diploid organisms, and algorithms used to achieve it. Furthermore we show concrete results by adapting Raven, a competitive open source *de novo* assembler [4, 5], to accommodate diploid assembly.

2. Concepts

This section explains some concepts that are required to fully understand this work. All concepts are explained in terms of a diploid organism.

A **chromosome** is a DNA molecule that comes in homologous pairs in diploid organisms. Each pair is inherited from a single parent. A **gene** is a subsequence in DNA that defines some trait in an organism, for example the color of eyes in humans. Since chromosomes come in pairs where each is inherited from a different parent, there can be more than one alternative for a gene. These are called **alleles**. If an allele is same on both chromosomes it is called homozygous, otherwise it is heterozygous. The size of alleles can range anywhere from a single base in DNA to a larger region. If an allele is just a single base in the DNA sequence, and has a sufficiently low percentage in population (for example about 0.5% possibility of variation between two individuals in a population) it can be classified as a **single-nucleotide polymorphism**, where single-nucleotide refers to the length: a single base/nucleotide in the DNA sequence. It is common to refer to single-nucleotide polymorphisms as SNP's or snips. Snips play a central role in inferring **haplotypes**. A haplotype is a set of all alleles inherited from a single parent. **Structural variations** happen when a part of a chromosome is deleted, inverted, duplicated, or even exchanged with a region from another chromosome [6]. They also contribute to organism's haplotype since they can be inherited from parents.

3. Problem setup

Third-generation sequencing produces reads/fragments of relatively large size when compared to previous sequencing technologies. In case of diploid organisms these reads can originate from any chromosome, but the information about which chromosome the read was taken from is lost, which complicates original haplotype reconstruction. Most modern *de novo* assemblers do not take into account the ploidity of an organism and treat input reads as if they all came from a single chromosome. The output is therefore just an alternation of alleles from different chromosomes [7].

There are different approaches to reconstructing the original haplotypes. Some *de novo* assemblers use additional information to achieve this. In case of the Trio-Canu assembler, the additional information is in form of short reads from both parents of an organism, and the input is in form of TGS reads. Having information about parents simplifies segregating child reads into two haplotypes, in a process known as "trio binning" [8]. Since this method requires a large amount of additional information, it may not be ideal for some cases. Another *de novo* assembler called Falcon infers haplotypes without any additional information. It uses "alignment tags" to construct an alignment graph that mostly represents a linear chain of bubbles, which are then decomposed into paths used to construct haplotypes [7]. Although this approach is more efficient in terms of input data, it is also less accurate than, for example, the approach used by Trio-Canu, as it fails to preserve information such as structural variants. For that reason the assembled haplotypes using this approach are also called "pseudo-haplotypes" [8].

Since the end goal of this work is to adapt Raven to output haplotypes for diploid organisms, we preserve its current approach of needing no additional information to carry out the assembly. We also opted to take a different approach to haplotype assembly than Falcon. Rather than constructing an alignment graph with ploidy in mind, and altering most of the assembly steps, the input reads will be segregated into two haplotype groups before the assembly, and the assembly itself will be carried out on these two groups separately. This has the consequence of requiring minimal changes to the existing assembly pipeline, as the groups will be created in a separate "preprocessing" step.

4. Data simulation

In order to effectively measure the efficiency of *de novo* diploid assembly, it is of interest to have full control over the input data. In order to accommodate this we must be able to control various haplotype defining features of the input reads such as SNP's and structural variations. We achieve this by writing a simple tool that simulates reads with wanted features using a reference genome. Used algorithms are described below.

In order to simulate reads, we set the number of reads, mean read length, and standard deviation of length. Then we pick a read at a random position in the reference genome and generate its length from the normal distribution.

Algorithm 1 Read simulation

Input: reference genome, number of desired reads, mean length of reads, standard deviation of mean length

Output: simulated reads

1: **function** SIMULATEREADS($referenceGenome, n, l, d$)
2: $reads \leftarrow \varnothing$
3: **while** $len(reads) \leq n$ **do**
4: $start \leftarrow uniformRnd(0, len(referenceGenome) - 1)$
5: $end \leftarrow min(start + normalRnd(l, d), len(referenceGenome) - 1)$
6: $reads \leftarrow reads + subsequence(referenceGenome, start, end)$
7: **end while**
8: **return** $reads$
9: **end function**

Algorithm 1 demonstrates read simulation from a reference genome. $uniformRnd$ picks a number from the given range using an uniform distribution, while $normalRnd$ picks a number using the normal distribution and given mean and standard deviation. It is necessary to constrain the read length, as shown on line 5, to guard from the case in which a part of the read will go outside the reference genome. $subsequence$ takes a reference genome and returns a region defined by the second and third arguments,

which denote start and end.

Simulating SNP's is straightforward. We pick the number of SNP's and uniformly distribute them along the reference genome. We must also decide what is the alternative base for the SNP. For each read that contains some SNP's we apply the alternative bases with some chance, otherwise we just leave the read as it is.

Algorithm 2 SNP simulation

Input: (simulated) reads, reference genome, number of SNP's, the chance of applying SNP's

Output: (simulated) reads with applied SNP's

```
1:  function SIMULATESNIPS(reads, referenceGenome, n, p)
2:      snpReads ← ∅
3:      snips ← pickSnips(referenceGenome, n)
4:      for all read ∈ reads do
5:          containedSnips ← findContained(read, snips)
6:          if containedSnips ≠ ∅ ∧ uniformRnd(0, 1) < p then
7:              snpReads ← snpReads + applySnips(read, containedSnips)
8:          else
9:              snpReads ← snpReads + read
10:         end if
11:     end for
12:     return snpReads
13: end function
```

Function $pickSnips$ picks n uniformly distributed SNP's using the given reference genome. For each read we check if there are any SNP's that are contained within that read using $findContained$. If the read contains SNP's we apply them using $applySnips$ with a probability of p, otherwise we do not edit the read. When applying a SNP we edit the read so that we put the alternative base at the SNP location, instead of keeping the original base.

In addition to SNP's, structural variations also define haplotypes. We simulate these on the reference genome itself, before simulating reads, so that each read contains the structural variation.

To simulate deletion of parts of reference genome, we define number of deletions, as well as the mean size of the deleted regions, and standard deviation. This is similar to how the read simulation is carried out, but the regions are deleted from the reference genome, rather than copied.

Algorithm 3 Deletion simulation

Input: reference genome, number of deletions, mean length, standard deviation

Output: reference genome with some parts deleted

```
1: function SIMULATEDELETIONS(referenceGenome, n, l, d)
2:     modifiedGenome ← referenceGenome
3:     for i ← 1, n do
4:         start ← uniformRnd(0, len(modifiedGenome) − 1)
5:         end ← min(start + normalRnd(l, d), len(modifiedGenome) − 1)
6:         modifiedGenome ← deleteSubsequence(modifiedGenome, start, end)
7:     end for
8:     return modifiedGenome
9: end function
```

Similarly to the simulation of sequences, we must be vary of deletion regions going out of bounds, therefore we restrict the end of the region to be at most at the end of the modified genome. The hypothetical function $deleteSubsequence$ returns the given sequence without the region specified by $start$ and end parameters.

When simulating inversion, we pick the regions in a similar fashion, using some predefined mean length and deviation, the only difference is that the region is inverted rather than deleted.

Algorithm 4 Inversion simulation

Input: reference genome, number of deletions, mean length, standard deviation

Output: reference genome with some parts inverted

```
1: function SIMULATEINVERSIONS(referenceGenome, n, l, d)
2:     modifiedGenome ← referenceGenome
3:     for i ← 1, n do
4:         start ← uniformRnd(0, len(modifiedGenome) − 1)
5:         end ← min(start + normalRnd(l, d), len(modifiedGenome) − 1)
6:         modifiedGenome ← invertSubsequence(modifiedGenome, start, end)
7:     end for
8:     return modifiedGenome
9: end function
```

The function on line 6, $invertSubsequence$, takes a genome as its first parameter and inverts the region defined by $start$ and end parameters.

A more interesting structural variation is the copy-number variation (CNV). It ma-

nifests as a region which has been duplicated several times in a row meaning there is a region of chromosome which consists solely of a subsequence repeating itself. In order to simulate this, we pick a region in the reference genome, and insert it before itself (or after) multiple times.

Algorithm 5 CNV simulation

Input: reference genome, number of CNV's, mean length, standard deviation of the length, mean number of duplications, standard deviation of the duplication number

Output: reference genome with applied CNV's

1: **function** SIMULATECNV($referenceGenome, n, l, d, nCNV, dCNV$)
2: $modifiedGenome \leftarrow referenceGenome$
3: **for** $i \leftarrow 1, n$ **do**
4: $start \leftarrow uniformRnd(0, len(modifiedGenome) - 1)$
5: $end \leftarrow min(start + normalRnd(l, d), len(modifiedGenome) - 1)$
6: $CNVRegion \leftarrow subsequence(modifiedGenome, start, end)$
7: **for** $j \leftarrow 1, normalRnd(nCNV, dCNV)$ **do**
8: $modifiedGenome \leftarrow insert(modifiedGenome, CNVRegion, start)$
9: **end for**
10: **end for**
11: **return** $modifiedGenome$
12: **end function**

In line 6 we obtain the region that will be repeatedly duplicated, and then duplicate it according to some random number generated using given mean and standard deviation. As is its name, $insert$ takes the region and inserts it in the modified genome at the given $start$ location.

Although these simulations are naive, as they do not take into account the very specific nature of SNP's and structural variations, nor do they take into account the specifics of the sequencing technology itself, they provide a good starting point for verifying the correctness of the algorithms we will introduce later in this work.

5. Algorithms for fragment separation based on haplotype defining features

Due to advances in sequencing technologies, the last two decades have seen an increased interest in haplotype assembly, yielding some algorithms with practical applications. These algorithms mostly work around SNP's, since they are the most potent source of genetic variation, and are relatively easy to detect as opposed to detecting structural variations [9].

5.1. SNP detection

In order to detect SNP's in the reads, we need to align the reads. This is done so that we can traverse the reads column by column and detect contentious locations where we can call an SNP. An example of such alignment is shown below. Each row represents a fragment, gaps are represented with a dash ('-').

```
1 CATAAAAGAACGTAGGTCGCCCGTCCGTAACCTGTC--------------
2 ---------ATAAAGGCAGTCGCTCTGTAAGCTGTCGATTCACCGGAAAG
3 ---ATCAAAGAACGTGTAGCCTGTCCGTAATCTAGCGCAT----------
4 CGTAAATAGGTAATGATTATCATTACATATC-------------------
5 -----------GTCGCTAGAGGCATCGTGAGTCGCTTCCGTACCGCAA--
6 CCGTAACCTTCATCGGATCACCGGAAAGGACCCGTAAAT-----------
```

Listing 5.1: Example of fragment alignment

To support this requirement, we use a SIMD-accelerated, partial-order alignment algorithm called spoa, which is a submodule of racon, a standalone consensus module [10] that is used by Raven itself.

Once we have fragments aligned, it is intuitively clear how the SNP's would be found, and this is described in more detailed using the listing below.

```
0 |  0     1  2
1 |CGTAGGTCG----
2 |-ATACGGCGA---
3 |--TACGCCGGT--
4 |---AGGTCGGTG-
5 |----GGACGATGA
```

Listing 5.2: Fragment alignment with points of interest colored and enumerated

By using this simplified example we can derive some general rules about how to conclude if some location holds an SNP. Location 0 is our first candidate for an SNP since we have differing bases in two fragments. However, due to relatively high error rates in third-generation sequencing technologies we can not conclude that this is indeed an SNP. We only have two bases, and one could have been easily been flipped due to a sequencing error, meaning there originally was not an SNP at this position in the chromosome. This brings us to our first criteria, minimal coverage depth. By saying that we have to have at least n bases to make conclusions about that location we somewhat limit the impact of sequencing errors. For this concrete example we will pick 4 as the minimal number of bases. This is a somewhat arbitrary value, and should be carefully picked when working with real data, such that it fits somewhere below the average coverage depth, but not too much so that sequencing errors do not become a significant factor.

Location 1 is our next SNP candidate. It has a coverage depth of 5, which is above our minimum coverage depth. However, we can see that we can not simply pick two alternating bases that would form an SNP, since there are four bases with similar relative ratio. Using this we can derive another rule. We can require a minimum ratio of the two most frequently alternating bases. For example we might require that the most frequent base takes up at least 50% of all bases, and that the second base is frequent for at least 30% of bases. In this example we can discard this position based on that criteria, since the most frequent base 'T' contributes by 40%, and all the other bases contribute by 20%.

This brings us to location 2. The minimum coverage depth criteria is satisfied, since we have 4 bases. The 'A' takes up 50% of all the bases, and so does the 'G'. Since the most frequent base is at least 50% frequent, and the second base is at least 30% frequent, we can conclude that this location is indeed an SNP. This separates the 4 fragments into two haplotypes. Fragments 2 and 5 would belong to one haplotype,

while the fragments 3 and 4 would belong to the other haplotype. This is fairly straightforward conclusion since there is only one SNP in this example. However, this is hardly a representative real world example, and separating fragments is a much harder problem than this example makes it out to be, NP-hard to be precise [9].

The above rules are shown in algorithm 6. For the sake of simplicity the alignment matrix is column major.

Algorithm 6 SNP detection

Input: aligned reads, minimum coverage depth, minimum frequency of the primary base from 0 to 1, minimum frequency of the secondary base from 0 to 1

Output: detected snips

1: **function** FINDSNP($alignedReads, minCov, freqPrimary, freqSecondary$)
2: $snips \leftarrow \varnothing$
3: **for** $column \leftarrow 1, len(alignedReads)$ **do**
4: $bases \leftarrow countBases(alignedReads(column))$
5: $coverage \leftarrow sum(bases)$
6: $primary \leftarrow primaryBase(bases)$
7: $secondary \leftarrow secondaryBase(bases)$
8: **if** $coverage \geq minCov$
$\wedge\ count(primary) \geq freqPrimary \cdot coverage$
$\wedge\ count(secondary) \geq freqSecondary \cdot coverage$ **then**
9: $snips \leftarrow snips + (column, base(primary), base(secondary))$
10: **end if**
11: **end for**
12: **return** $snips$
13: **end function**

Function $countBases$ goes through the given column, and counts valid bases, returning a structure holding necessary information. The coverage is then calculated using the sum of all base counts, and the $primary$ and $secondary$ base information is extracted using helper functions. If the coverage is sufficiently high, and the frequency of primary and secondary bases is above specified thresholds, we say that the current location is an SNP and save information about it's position, and primary and secondary base.

To conclude this section, and pave the way for the next section, we will take a look at a more problematic placement of SNP's in fragments.

```
     0  1     2
GCTAGCTAGGATCGATCA----
-CTAGCTACGATCGTTCAG---
--TAGATACGATCGTTCAGG--
---AGATACGATCGATCAGGC-
----GCTAGGATCGATCAGGCC
```

Listing 5.3: Conflicting SNP's

In listing 5.3 we can see 5 fragments that contain 3 SNP's. If we choose to separate the fragments based on the SNP at location 0, we would separate fragments 1, 2, and 5 into the first group, while the fragments 3 and 4 would be in another group. However, this conflicts with the SNP at location 1, since if we would separate based on this SNP, we would put fragments 1 and 2 into separate groups, as opposed to putting them in the same group.

There are two way of dealing with SNP conflicts. The first is to remove the minimum amount of fragments such that there are no conflicts, while the second revolves around removing the minimum amount of SNP's to achieve the same goal. These problems are formulated as minimum fragment removal (MFR) and minimum SNP removal (MSR) respectively [9]. In the following section we will explain and show an implementation of an MFR algorithm.

5.2. Minimum fragment removal

Once we are able to properly detect SNP's in our data, we can construct an SNP matrix [9]. This matrix is essential for building a conflict graph [11], which will later be the basis of the MFR algorithm. The construction itself is very basic. We associate rows with fragments, and columns with SNP's. For each SNP and fragment, we mark the cell in the matrix with 2 (or any number other than zero) if the fragment does not hold the particular SNP, otherwise we mark the cell with 1 or with -1, depending on which base the cell holds. The choice of which base gets marked with 1/-1 does not matter, it is only necessary that the choice is consistent for the same SNP across different fragments. These rules are arbitrary, and it is possible to use any marking system as long as you can detected conflicts, and differentiate them from the case when a fragment does not hold an SNP. The SNP matrix for listing 5.3 is shown below.

	S1	S2	S3
F1	1	1	1
F2	1	-1	-1
F3	-1	-1	-1
F4	-1	-1	1
F5	1	1	1

Figure 5.1: SNP matrix for listing 5.3

Since all SNP's are present in all fragments, there are no 2's in the matrix. To find a conflict between two fragments, we add the two rows column by column, and if the sum is equal to zero, the fragments are in conflict. For example if we want to find out if F3 and F4 are in conflict, for each SNP we sum the corresponding cells in the two fragments, and if the sum is 0 then fragments are in conflict. This occurs for the last SNP: S3. Using this algorithm we can construct a fragment conflict graph. Each fragment is a vertex in the graph, and each connection is a conflict. An alternative approach of adding only the fragments that conflict as vertices in the graph is perhaps even better, since it avoids creating many single fragment components. The conflict graph based on figure 5.1 is shown below.

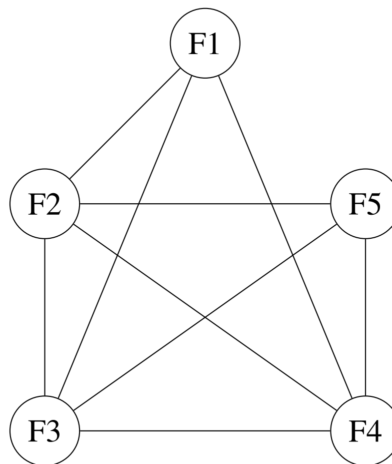

Figure 5.2: Fragment conflict graph based on figure 5.1

As we can see from this graph, every fragment is in conflict with all the other fragments, except F1 and F5. Fragment conflict graph is the basis of the minimum fragment removal problem. We ask what is the minimum number of fragments we can remove from the graph to make it bipartite. If the graph is bipartite then we can

trivially separate all the fragments into two haplotype groups, i.e. the only conflicts are between fragments belonging to different haplotypes. The MFR problem is a NP-hard problem, and in 2002 R. Lippert proposed a solution based on the branch and bound algorithm [11].

Rather than finding one optimal solution, the algorithm goes further, and finds all the fragments that are present in every optimal solution. This subset of fragments is obviously smaller, or at most equal to the number of fragments present in one optimal solution, so this may be a drawback. However, if we choose to take only one optimal solution the resulting haplotype separation might not bring meaningful results that hold for all optimal solutions. For this reason it is beneficial to take the intersection of all optimal solutions, as to not infer conclusions on a single (possibly randomly) picked optimum [11]. Even though this is Lippert's observation, it might be possible that the intersection of all optimal solutions is too small, and reduces the average coverage too much, which might mean that the overall assembly quality degrades significantly. As with most things this should be measured using real world data.

Algorithm 7 Finding the optimal solution to MFR

Input: fragment conflict graph, removed vertices, bound

Output: minimum number of fragments to remove to make the graph bipartite

1: **function** OPTIMA(g, r, b)
2:     **if** $len(r) \geq b$ **then**
3:         **return** ∞
4:     **else**
5:         $c \leftarrow oddCycle(g - r)$
6:         **if** $c \neq \varnothing$ **then**
7:             $n \leftarrow b$
8:             **for all** $v \in c$ **do**
9:                 $n \leftarrow min(n, optima(g, r + v, b))$
10:             **end for**
11:             **return** n
12:         **else**
13:             **return** $len(r)$
14:         **end if**
15:     **end if**
16: **end function**

The $optima$ function finds the minimum number of fragments that have to be re-

moved in order for the graph to be bipartite. It does so by finding odd cycles in the graph (which means the graph is not bipartite [11]) and breaking them up by removing nodes in the cycle. Once a cycle has been broken, the function recurses with the node removed from the graph (i.e. added to the set of removed vertices) to check for additional cycles. Although in our specific case it was unused, the function supports bounding using some heuristic using the parameter b. If the set of removed vertices grows too large, the search for solution will end.

Once we are able to find out how many removed vertices the optimal solution has, we can easily find the intersection of all optimal solutions.

Algorithm 8 Fragment intersection

Input: fragment conflict graph

Output: fragment conflict graph reduced to fragments found in all optimal bipartite configurations

```
1: function FRAGMENTINTESECTION(g)
2:     d ← ∅
3:     s ← optima(g, ∅, ∞)
4:     for all v ∈ g do
5:         if optima(g, v, ∞) = s then
6:             d ← d + v
7:         end if
8:     end for
9:     return g − d
10: end function
```

We first find the number of removed vertices in the optimal solution. After that for each vertex in the graph we check if removing it changes the optimal solution. If the optimal solution remains unchanged, this means that the vertex is removed in some optimal solution, and does not belong in the intersection of all optimal solutions. We keep track of such vertices, and remove them all together from the graph in the end.

5.2.1. Memoization

Since some of these vertices might be contained in some odd cycles, it is possible that we already computed the value of the $optima$ function with that vertex in the set of removed vertices. For this reason it is beneficial to consider memoization. We opted to save the results of computations in a hash map. Since sets of removed vertices might be

very large it may not be ideal to recompute the hash every time, given that for each layer of recursion in the $optima$ function, we merely add another node to the removed set. To avoid wastefully computing the hash every time, we use a positionally independent hash method called Zobrist hashing [12]. At the start of the $fragmentIntesection$ function we generate a field of randomly generated numbers. The field must be large enough so that we can index it with each vertex in the current graph. To calculate the hash of a set of removed vertices we accumulate the corresponding values in the field for each vertex using the xor binary operation. Since xor is commutative, when we add new vertices to the set of removed vertices, we can update the hash value simply by xor-ing it with the corresponding values in the field. Updated algorithms that use memoization are shown below.

Algorithm 9 Fragment intersection that uses memoization

Input: fragment conflict graph

Output: fragment conflict graph reduced to fragments found in all optimal bipartite configurations

1: **function** FRAGMENTINTESECTION(g)
2: $d \leftarrow \varnothing$
3: $z \leftarrow generateZobrist(len(g))$
4: $mem \leftarrow \varnothing$
5: $s \leftarrow optima(g, \varnothing, mem, z, 0, \infty)$
6: **for all** $v \in g$ **do**
7: **if** $optima(g, v, mem, z, z(v), \infty) = s$ **then**
8: $d \leftarrow d + v$
9: **end if**
10: **end for**
11: **return** $g - d$
12: **end function**

On line 3 we generate a field of random numbers using the hypothetical $generateZobrist$ function. When initially calculating the optimum, we pass the hash value 0 to the $optima$ function, as there are no removed vertices. We also pass the field itself, as the $optima$ function needs it when removing vertices to calculate the next hash. While iterating over the vertices of the graph, we pass the corresponding value in the field as the hash value, as it is the only removed vertex in that moment. We also create a map called mem that will hold computed values. Since the Zobrist hashing method does not produce a perfect hash, the mapping of hash to value is not 1 to 1, as there might

be more sets of removed vertices with the same hash. For this reason the map holds pairs of removed vertices and the corresponding values of the $optima$ function.

Algorithm 10 Finding the optimal solution to MFR using memoization

Input: fragment conflict graph, removed vertices, memoization map, field of randomly generated vertices, current hash, bound

Output: minimum number of fragments to remove to make the graph bipartite

```
 1: function OPTIMA(g, r, m, z, h, b)
 2:     if len(r) ≥ b then
 3:         return ∞
 4:     else
 5:         if h ∈ m then
 6:             for all (removedVertices, value) ∈ m(h) do
 7:                 if removedVertices = r then
 8:                     return value
 9:                 end if
10:             end for
11:         end if
12:         c ← oddCycle(g − r)
13:         if c ≠ ∅ then
14:             n ← b
15:             for all v ∈ c do
16:                 n ← min(n, optima(g, r + v, m, z, h ⊕ z(v), b))
17:             end for
18:             m(h) ← m(h) + (r, n)
19:             return n
20:         else
21:             m(h) ← m(h) + (r, len(r))
22:             return len(r)
23:         end if
24:     end if
25: end function
```

In the modified $optima$ function we check if the hash exists in the map, as seen on line 5, and if it does we must perform an additional check to see if any of the pairs corresponds to our current set of removed vertices (lines 6 trough 10). If it does we return the second value of the pair, which is the previously computed value of

the $optima$ function for that set of removed vertices. When adding a new vertex to removed vertices, we apply exclusive or to the existing hash h and the value in the field of the vertex we are adding $z(v)$, as seen in line 16. We also make sure the add the newly computed values to the map, as seen on line 18 and 21.

5.2.2. Odd cycles in undirected graphs

The $optima$ function relies on being able to find an odd cycle in the fragment conflict graph. Here we will describe the algorithm used to find them. The algorithm is based on depth-first search, and uses additional information such as current depth, and the previous node.

Algorithm 11 Finding an odd cycle in an undirected graph

Input: fragment conflict graph, map that maps visited vertices to a pair that holds the depth at which they were visited and their parent vertex, parent vertex, current vertex, current depth

Output: odd cycle if it exists, $\varnothing$ otherwise

1: **function** ODDCYCLEIMPL(g, v, p, c, d)
2: $v(c) \leftarrow (d, p)$
3: $d \leftarrow d + 1$
4: **for all** $neighbour \in g(c)$ **do**
5: **if** $neighbour \in v$ **then**
6: **if** $(d - depth(v(neighbour))) \mod 2 = 1$ **then**
7: $n \leftarrow c$
8: $oddCycle \leftarrow \{n\}$
9: **while** $n \neq neighbour$ **do**
10: $n \leftarrow parent(v(n))$
11: $oddCycle \leftarrow oddCycle + n$
12: **end while**
13: **return** $oddCycle$
14: **end if**

```
15:         else
16:             oddCycle ← oddCycleImpl(g, v, c, neighbour, d)
17:             if oddCycle ≠ ∅ then
18:                 return oddCycle
19:             end if
20:         end if
21:     end for
22:     return ∅
23: end function
```

Firstly we add the information about the current vertex c to the map of visited vertices m. This involves the current depth d at which we first visited the vertex, as well as it's current parent p. The helper functions $depth$ and $parent$ retrieve information from this pair. Afterwards we increment the depth by 1. We then go over each neighbour of the current vertex. If we already visited the current neighbour, we check if the difference between the depths is odd, as seen on line 6. If it is, then we found an odd cycle. Given that we keep the record of the previous node in the map of visited vertices v, we can easily backtrack from the current vertex c to the already visited vertex $neighbour$, and reconstruct the cycle. This is shown on lines 7 through 12. If we have not visited the $neighbour$ vertex, we recurse: the current vertex becomes $neighbour$, while the parent vertex becomes c. If this recursion returns an odd cycle, we return it. Once we successfully go over all neighbours we return $\varnothing$ since we have not detected an odd cycle.

To support the $oddCycleImpl$ function we provide a simpler delegating function that is used by the $optima$ function.

Algorithm 12 Odd cycle delegate function

Input: fragment conflict graph

Output: odd cycle if it exists, $\varnothing$ otherwise

```
1:  function ODDCYCLE(g)
2:      visited ← ∅
3:      for all vertex ∈ g do
4:          if vertex ∉ visited then
5:              returnVal ← oddCycleImpl(g, visited, ∅, vertex, 0)
6:              if returnVal ≠ ∅ then
7:                  return returnVal
8:              end if
9:          end if
10:     end for
11:     return ∅
12: end function
```

The benefit of having one extra layer of indirection is that we can make sure that we search the whole graph for odd cycles. For example, if the graph consists of three separate components, we would visit all of them (or return early if we found an odd cycle). This logic is clearly demonstrated in the main for loop. We check for every vertex if it has been visited, and if not, we call the $oddCycleImpl$ function using that vertex as the starting point. In other words we are visiting a new component in the graph.

5.3. Fragment separation

Having successfully obtained the intersection of all optimal bipartite configurations, the next step is to separate this smaller graph into two haplotype groups. The approach is similar as with finding an odd cycle. We use depth-first search as the base, and for each vertex we visit, we assign it to a haplotype group that is the opposite of the parent vertex. As shown below, it is enough to have a boolean flag indicating to which group the fragment belongs.

Algorithm 13 Separating the fragments

Input: bipartite fragment conflict graph, set of visited vertices/fragments, pair with two haplotype groups, current fragment, flag indicating that the fragment belongs to the alternate haplotype group

1: **function** SEPARATEFRAGMENTSIMPL($graph, visited, pair, current, alternate$)
2: $visited \leftarrow visited + current$
3: **if** $alternate$ **then**
4: $pair(1) \leftarrow pair(1) + current$
5: **else**
6: $pair(0) \leftarrow pair(0) + current$
7: **end if**
8: **for all** $neighbour \in graph(current)$ **do**
9: **if** $neighbour \notin visited$ **then**
10: $separateFragmentsImpl(graph, visited, pair, neighbour, \neg alternate)$
11: **end if**
12: **end for**
13: **end function**

Using the $alternate$ flag we pick into which group the fragment will be separated. We then make sure to negate the boolean flag when visiting a new neighbour. Similarly to the $oddCycleImpl$ function in algorithm 11, this function will not make sure that all the components of the graph are visited, so we provide a delegate function.

Algorithm 14 Fragment separation delegate function

Input: bipartite fragment conflict graph

Output: pair containing two sets of fragments

```
1:  function SEPARATEFRAGMENTS(graph)
2:      visited ← ∅
3:      pair ← (∅, ∅)
4:      for all vertex ∈ graph do
5:          if vertex ∉ visited then
6:              innerPair ← (∅, ∅)
7:              separateFragmentsImpl(graph, visited, innerPair,
                                      vertex, false)
8:              if len(innerPair(0)) < len(innerPair(1)) then
9:                  innerPair ← (innerPair(1), innerPair(0))
10:             end if
11:             if len(pair(0)) < len(pair(1)) then
12:                 pair(0) ← pair(0) + innerPair(0)
13:                 pair(1) ← pair(1) + innerPair(1)
14:             else
15:                 pair(1) ← pair(1) + innerPair(0)
16:                 pair(0) ← pair(0) + innerPair(1)
17:             end if
18:         end if
19:     end for
20:     return pair
21: end function
```

Similarly to $oddCycle$ we visit each vertex that is not already in the set of visited vertices. In addition to that, we try to keep the two fragment sets balanced in size by always adding the larger set in the loop to the smaller set of existing fragments.

6. Integrating changes into Raven

Our end goal is to incorporate these algorithms into Raven. We do this by separating the fragments before the actual assembly phase, and simply carry out the assembly on the two groups of fragments separately. High level overview is available below.

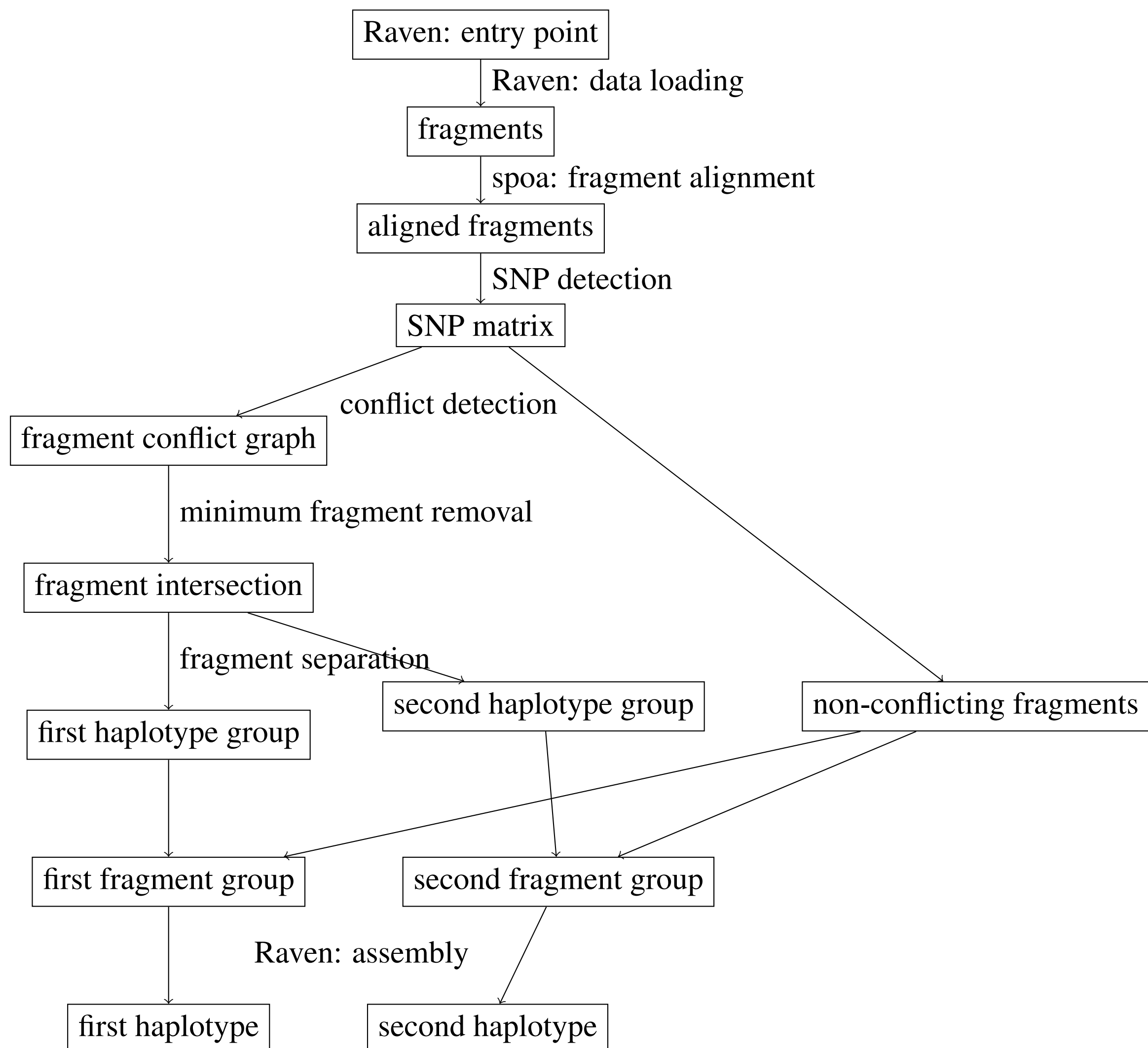

Figure 6.1: Incorporating diploid assembly into Raven

The first step involves loading the data. This is already handled by Raven. Once we acquire the fragments, we align them using spoa. Using this alignment we can construct an SNP matrix, from which we detect conflicts and build the fragment conflict graph. The fragments that do not conflict are kept in a separate group that will be added in a later step. From the fragment conflict graph we can find the intersection of all optimal solutions, which we then separate into two haplotye groups. We then add the non-conflicting fragments to both of these groups, and then carry out the assembly normally on these groups to obtain the final results: the two haplotypes.

The code is available at `https://github.com/yatsukha/raven` repository on the "diploid" branch. This repository is a fork of `https://github.com/lbcb-sci/raven` which is the original repository for Raven, created and maintained by Robert Vaser.

7. Conclusion

Genome assembly has advanced significantly in the last few decades, both the sequencing technologies used to obtain data as the assembly starting point, and the algorithms used to carry out and perfect the assemblies themselves. We leverage this growth and expand the capabilities of Raven by adapting its existing assembly pipeline to work on diploid organisms.

In order for Raven to be comparable to state of the art diploid assemblers such as Falcon [7] and Trio-Canu [8], it is necessary to continue the work. The assembly should be tested on real world data from diploid organisms, and all performance optimizations, as well as alternative approaches should be explored, and thoroughly tested.

LITERATURE

[1] Eric E. Shadt, Steve Turner, and Andrew Kasarskis. A window into third-generation sequencing. *Human Molecular Genetics*, 19(R2):227–240, 2010.

[2] Christoph Bleidhorn. Third generation sequencing: technology and its potential impact on evolutionary biodiversity research. *Systematics and Biodiversity*, 16 (1):1 – 8, 2016.

[3] YongKiat Wee, Salma Begum Bhyan, Yining Liu, Jiachun Lu, Xiaoyan Li, and Min Zhao. The bioinformatics tools for the genome assembly and analysis based on third-generation sequencing. *Briefings in Functional Genomics*, 18(1):1–12, 2019.

[4] Robert Vaser. De novo genome assembler for long uncorrected read. URL `https://github.com/lbcb-sci/raven`.

[5] Ryan R. Wick and Kathryn E. Hold. Benchmarking of long-read assemblers for prokaryote whole genome sequencing. *F1000Research*, 2019.

[6] Lars Feuk, Andrew R. Carson, and Stephen W. Scherer. Structural variation in the human genome. *Nature Reviews Genetics*, 7:85 – 97, 2006.

[7] Chen-Shan Chin, Paul Peluso, Fritz J Sedlazeck, Maria Nattestad, Gregory T Concepcion, Alicia Clum, Christopher Dunn, Ronan O'Malley, Rosa Figueroa-Balderas, Abraham Morales-Cruz, Grant R Cramer, Massimo Delledonne, Chongyuan Luo, Joseph R Ecker, Dario Cantu, David R Rank, and Michael C Schatz. Phased diploid genome assembly with single-molecule real-time sequencing. *Nature Methods*, 13:1050–1054, 2016.

[8] Sergey Koren, Arang Rhie, Brian P. Walenz, Alexander T. Dilthey, Derek M. Bickhart, Sarah B. Kingan, Stefan Hiendleder, John L. Williams, Timothy P. L. Smith, and Adam M. Phillippy. De novo assembly of haplotype-resolved genomes with trio binning. *Nature Biotechnology*, 36:1174–1182, 2018.

[9] Robert Vaser. Approaches to haplotype assembly. *UNIZG-FER Laboratory for Bioinformatics and Computational Biology*, 2016.

[10] Robert Vaser, Ivan Sović, Niranjan Nagarajan, and Mile Šikić. Fast and accurate de novo genome assembly from long uncorrected reads. *Genome Research*, 27 (5):737 – 746, 2017.

[11] Ross Lippert, Russel Schwartz, Giuseppe Lancia, and Sorint Istrail. Algorithmic strategies for the single nucleotide polymorphism haplotype assembly problem. *Briefings in Bioinformatics*, 3(1):23 – 31, 2002.

[12] Albert L. Zobrist. A new hashing method with application for game playing. Technical Report 88, 1970.

De Novo diploid assembly using third-generation sequencing data

Abstract

With the advent of third-generation sequencing technologies, the problem of genome assembly is more approachable than ever. We leverage this to explore diploid genome assembly. In this work we discuss algorithms used in various phases of diploid genome assembly, and adapt Raven, an existing *de novo* genome assembler to work with diploid organisms.

Keywords: genome, assembly, diploid, Raven

De novo sastavljanje diploidnih genoma koristeći tehnologiju za očitanje DNA treće generacije

Sažetak

Dolaskom treće-generacije tehnologija očitavanja DNA, problem sastavljanja genoma je postao pristupačniji. Potaknuti time, u ovom radu istražujemo problem sastavljanja genoma diploidnog organizma. Uz opis algoritama koji se koriste u različitim fazama sastavljanja genoma diploidnog organizma, pokazujemo primjenu u stvarnom svijetu nadogradnjom Ravena, postojećeg *de novo* sastavljača genoma, kako bi podržao sastavljanje genoma diploidnih organizama.

Ključne riječi: genom, sastavljanje, diploidni, organizam, Raven