

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND
COMPUTING

MASTER's THESIS No. 1126

De novo transcriptome assembly

Robert Vaser

Zagreb, June 2015.

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

Zagreb, 6 March 2015

MASTER THESIS ASSIGNMENT No. 1126

Student: **Robert Vaser (0036456092)**
Study: Computing
Profile: Computer Science

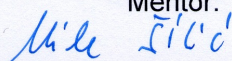
Title: **De novo transcriptome assembly**

Description:

The problem of genome assembly has been around for more than 30 years, but its sister problem - that of transcriptome assembly - is still novel and unsolved. Due to relative novelty of the RNA sequencing technology and the computational difficulties in assembling a complete transcriptome, there is a lack of robust and mature RNA-Seq assembly algorithms. The goal of this work will be to assemble an overlap-layout-consensus implementation of an RNA-Seq assembly. As the name suggests, the implementation consists of the following: 1) Overlap phase, in which the overlap graph is built; 2) Layout phase, in which the graph is simplified and regionally linearized; 3) Consensus phase, in which the ambiguities from the layout phase are resolved. Every phase is to have an output in a standard community format. Synthetic datasets of varying complexity are to be constructed for extensive implementation testing. The project 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: 13 March 2015
Submission date: 30 June 2015

Mentor:



Associate Professor Mile Šikić, PhD

Committee Secretary:



Assistant Professor Tomislav Hrkać, PhD

Committee Chair:



Full Professor Siniša Srbljić, PhD

Zagreb, 6. ožujka 2015.

DIPLOMSKI ZADATAK br. 1126

Pristupnik: **Robert Vaser (0036456092)**
Studij: Računarstvo
Profil: Računarska znanost

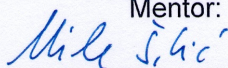
Zadatak: **De novo sastavljanje transkriptoma**

Opis zadatka:

Problemom sastavljanja genome bave se mnogi znanstvenici zadnjih 30 godina, no srodan problem sastavljanja transkriptoma još je uvijek slabo izučavan. Zbog toga što je tehnologija sekvenciranja RNA relativno nova i zbog računalnih problema u sastavljanju kompletnoga transkriptoma nedostaju robusni i dobro istestirani algoritmi za sastavljanje očitavanja dobivenih RNA sekvenciranjem. Cilj ovoga rada je sastavljanje transkriptoma koristeći preklapanje-razmještaj-konsenzus paradigmu. Kao što samo ime sugerira, implementacija se sastoji od: 1) Faze preklapanja u kojoj se izrađuje graf preklapanja; 2) Faze razmještaja u kojoj se graf pojednostavljuje i regionalno linearizira; 3) Faze konsenzusa u kojoj se nepreciznosti nastale u fazi razmještavanja rješavaju. Svaka faza treba imati izlaz koji odgovara standardnim formatima. Potrebno je proizvesti nekoliko testnih skupova podataka različite kompleksnosti i provesti iscrpno testiranje. 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 jednom od OSI odobrenih licenci.

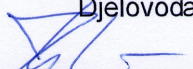
Zadatak uručen pristupniku: 13. ožujka 2015.
Rok za predaju rada: 30. lipnja 2015.

Mentor:



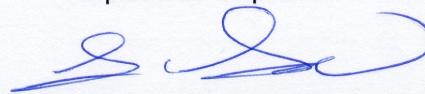
Izv. prof. dr. sc. Mile Šikić

Djelovođa:



Doc. dr. sc. Tomislav Hrkać

Predsjednik odbora za
diplomski rad profila:



Prof. dr. sc. Siniša Srblić

I would like to thank my beloved girlfriend Marina Štefić and my whole family for all the support they gave me.

Further on, many thanks to my mentor Mile Šikić for help and knowledge he gave me over the past years.

Special thanks to Marko Čulinović and Mario Kostelac for helping me with part of the implementation.

TABLE OF CONTENTS

1. Introduction	1
2. Preliminaries	3
2.1. RNA sequencing	3
2.2. Transcriptome assembly	5
2.2.1. Overlap-layout-consensus	6
2.2.2. De Bruijn graphs	6
3. Methods	8
3.1. Enhanced suffix array	8
3.1.1. Algorithms	10
3.2. Error correction	14
3.3. Overlap phase	14
3.4. Layout phase	18
3.4.1. Graph simplification	18
3.4.2. Contig extraction	26
3.5. Consensus phase	26
4. Implementation	28
4.1. Overview	28
4.2. External dependencies	29
4.2.1. Afgreader	29
4.2.2. EDLIB	29
4.2.3. CPPPOA	30
4.3. Code structure	30
5. Results	33
5.1. Testing	33
5.2. Discussion	38

6. Conclusion	39
References	40

1. Introduction

In the year 1990. started the famous Human Genome Project (HGP) with the task to find the sequence of nucleotide base pairs the human DNA consist of. The project was a success, lasted over a decade and costed over 3 billion dollars. It was found out that the human genome has over 3 billion nucleotide pairs and that more than 99% of pairs are equal between any two individuals [1]. Beside the huge discovery this project was the key event in rapid development of sequencing methods.

As the years passed by, novel sequencing technologies started to appear which enabled genomic analysis of many different species at much lower cost [2]. Newly obtained individual genes, chromosomes or entire genomes accelerated the development of various fields related to biological sciences (medicine, forensics, agriculture etc. [3]). Side by side with DNA sequencing emerged RNA sequencing (often called RNA-seq) which enabled transcriptome profiling. A transcriptome is the set of all RNA molecules in a cell and it is regularly changing. With the help of sequencing it is possible to measure the abundance of RNA molecules in different cell states which has many applications from drug discovery [4] to detection of gene fusions in cancer [5].

The problem with sequencing technologies is that they can only read a limited amount of bases at once, called reads, which varies depending on platform used [6]. Those fragments are afterwards assembled together into (whole) genomes/transcripts by programs called assemblers. There are two types of assemblers: mapping and de-novo. The mapping assemblers map all of the sequenced fragments to a reference sequence and thus building a new similar sequence. On the other hand the de-novo approach glues the fragments together to create full length, often novel, sequences without the aid of reference data. Such reconstruction has many algorithmic challenges and can take quite amount of time to execute.

This thesis will try to bring up and describe all the necessities and problems in implementing a transcriptome de-novo assembler.

Chapter 2 will give an introduction to biological background of this thesis.

Chapter 3 will describe the methods used to create a overlap-layout-consensus tran-

scriptome assembler.

Chapter 4 will give an overview on this thesis' implementation which includes design, code structure and external dependencies.

Chapter 5 will show results of the implementation in comparison to three other RNA-seq assemblers. In addition it will offer a discussion and thoughts on future improvements.

Chapter 6 will give a conclusion of this thesis.

2. Preliminaries

This chapter contains required knowledge, mostly biological, to understand the topic of this thesis. First, a brief introduction to RNA sequencing will be given alongside with its challenges. Afterwards, two different types of de novo assembler will be described with their advantages and disadvantages.

2.1. RNA sequencing

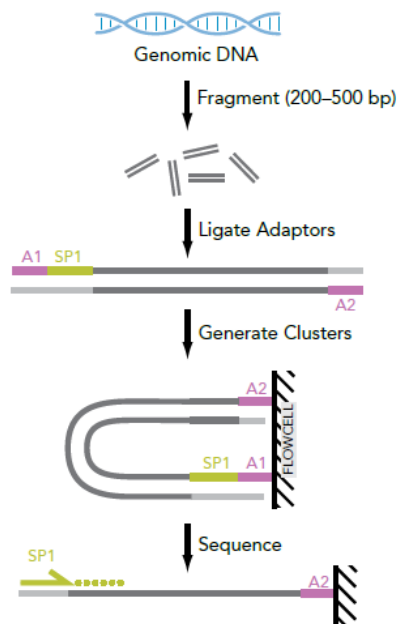
A set of all RNA molecules is called the transcriptome. It includes messenger RNA (mRNA) and non-coding molecules such as transfer RNA (tRNA) and ribosomal RNA (rRNA). To obtain a transcriptome snapshot in a cell at a certain moment the RNA-seq method is required. RNA-seq is also called whole-transcriptome shotgun sequencing and is using high-throughput sequencing methods for RNA characterization [7]. As the RNA sequences are single-stranded and less stable than the DNA, many sequencing platforms convert RNA to cDNA (complementary DNA) prior to sequencing. The cDNA is double-stranded and fragmented into smaller pieces which afterwards serve as a template for sequencing. There are two main types of strategies: single-end and paired-end. The single-end approach partially reads the fragments from one side, while the paired-end approach does it from both ends [7]. Both approaches can be seen in figure 2.1.

The main benefit of paired-end sequencing is that it can help in initial transcriptome assembly and in finding different isoforms [7]. Isoforms are different proteins produced by the same gene¹. This process is called alternative splicing and occurs during gene expression, i.e. when the DNA is translated to mRNA and mRNA to proteins. The whole process can be seen in figure 2.2.

As the sequencing technologies can read a limited amount of bases with less than 100% accuracy (the length and accuracy vary depending on the platform used [6]), the whole process is redone several times so that each base is read more than once. The

¹Isoforms can also refer to different mRNAs sharing exons (coding parts of the DNA).

Single-Read Sequencing



Paired-End Sequencing

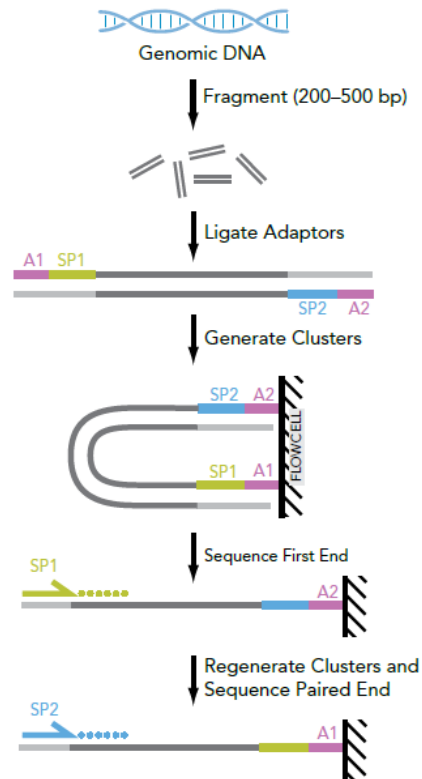


Figure 2.1: Sequencing approaches: single-end vs paired-end [8]

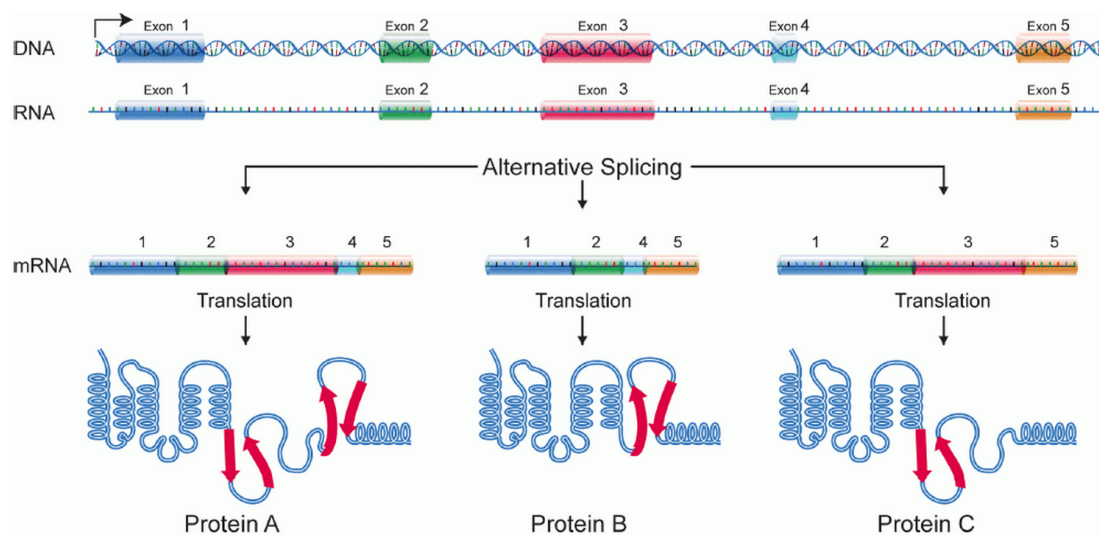


Figure 2.2: Alternative splicing - process where the same gene produces several different proteins [9].

average number of reads per base in the sequence is called coverage [7]. The higher the coverage the higher the chance of avoiding erroneous bases and reconstructing the exact starting sequence.

After the sequencing is finished, the big amount of raw data (i.e. reads) needs to be glued together to obtain the whole transcriptome. This is a task for programs called assemblers which will be discussed in the next section.

2.2. Transcriptome assembly

As mentioned before there exist two types of assemblers: mapping and de novo (there is also a hybrid one as a combination of those two). When there is a reference sequence the mapping approach is used. The reads are mapped to the reference and a new similar sequence is created. Not many species have a reference genome so the de novo approach is required.

De novo expression comes from Latin and it means "anew", "from the beginning" [10]. The de novo assemblers try to reconstruct whole starting sequences without help of any reference data. Many good DNA de novo assemblers were implemented and one might ask why not use them for transcriptome assembly. Here are some reasons why:

1. The DNA sequencing depth (coverage) is usually uniformly distributed across the genome but sequencing depth of the transcriptome can vary a lot due to gene expression. Assemblers often use coverage to distinguish repetitive regions, correct reads and get an optimal set of parameters for the assembly. That would result in marking the abundant transcripts repetitive, the less abundant erroneous and the parameters would favour only a subset of transcripts [11].
2. RNA sequencing can be strand specific and RNA assemblers should use that information to avoid chimeric² assemblies, where reads from different strands are tied together. [11].
3. More transcripts can share the same exons and it is hard to distinguish different isoforms unambiguously [11].
4. The expected result of RNA assembly is a set of sequences (transcripts) while DNA assembly reconstructs only one sequence (genome).

²Chimera (from Greek mythology) is a single organism composed of more different species and is a term often used in genetics [12]

5. RNA sequencing produces a large amount of short reads which sets high requirements for computational speed and memory efficiency [13].

All the given reasons hint that obtaining the whole transcriptome is a difficult task and needs additional methods which are not present in DNA assembly.

Nevertheless, DNA and RNA sequencing share the assembly basics and that is grouping of overlapping reads into contigs and contigs into scaffolds. Scaffolds are contig groups where contig orientation, order and in-between distance is known [12]. There are many different approaches but here only two will be mentioned, both based on directed graphs³. First method is based on the De Bruijn graphs (DBG) and the second is called overlap-layout-consensus method (OLC). The latter one is the core of this thesis and will be described first.

2.2.1. Overlap-layout-consensus

The OLC method was developed for the first generation of sequencing technology which produced longer reads [12] and is based on overlap graphs. Each sequenced read becomes a node in the graph and edges represent the overlaps between nodes (i.e. reads). The method consists of three phases:

1. *Overlap phase* - where the overlap graph is built.
2. *Layout phase* - where the graph is simplified and regionally linearized.
3. *Consensus phase* - where ambiguities from the layout phase are resolved.

Drawback of the OLC method is the overlap phase which needs a lot of computational resources. On the other hand, the method's natural modularity enables optimization of each phase separately for every new assembly project [14]. Detailed explanation of each phase will be given in chapter 3 alongside the methods used in this thesis' implementation.

2.2.2. De Bruijn graphs

Due to the enormous amount of short reads produced by the novel sequencing technologies a fast method based on the De Bruijn graph was developed. The core of this method is as follows:

³Graphs are sets of nodes connected by edges. Edges can have many different attributes and when direction is one of them, graphs get the prefix directed [12].

1. Short reads are split to small overlapping sequences of length k (called k -mers) and are used as edges in the creation of a directional graph.
2. Each edge connects two vertices, first representing the edge prefix of length $k - 1$ and the other the edge suffix of length $k - 1$.
3. To reconstruct the starting sequence, an Eulerian path must be found in the graph, which is a path that visits every edge only once.

The main advantages of this method are that it never looks for pairwise overlaps between reads and the low memory cost of the De Bruijn graphs [15]. Despite that the Eulerian path can be found in linear time, there can exist many such paths (due to repeats[15]) which makes it hard to reconstruct the sequenced genome or transcriptome. Nevertheless, many present-day assemblers, such as SOAPdenovo-Trans [16], Oases [17] and Trinity [18], are using the De Bruijn graphs for transcriptome assembly. Chapter 5 will give a comparison of this thesis implementation and the mentioned assemblers.

3. Methods

This chapter will bring up descriptions of all the methods used in implementing this thesis' assembler. First, the enhanced suffix array and algorithms based on it will be explained. It is the core component of error correction and the overlap phase which follow right after. Afterwards, the layout phase will be described with algorithms for graph simplification and contig extraction. At last, a brief overview of the consensus phase will be given.

3.1. Enhanced suffix array

In string processing the suffix tree plays a great role. It can solve a multitude of problems such as super-maximal repeats, longest common substring, all suffix-prefix matching and many more [19]. Due to their high memory complexity (20 bytes per input character [19]) it is not always possible to use them, especially in bioinformatics where big amounts of data are processed. As a space efficient alternative, suffix arrays were introduced [20] which require $4n$ bytes in their basic form. Suffix arrays can be constructed in linear time and memory whether with the help of suffix trees or directly. At first glance it is not clear how every algorithm based on the suffix tree can be executed with suffix arrays. According to Abouelhoda et al. [19] it is possible to replace the suffix tree and all of its algorithms with the enhanced suffix array which consists of the basic suffix array and additional tables.

The enhanced suffix array is the core of this thesis and it is used for pattern search in optimal time. Two additional tables need to be added to the basic suffix array. First is the longest common prefix (LCP) table and the second is the child table. The LCP table stores lengths of the longest common prefixes of two succeeding suffixes in the suffix array as shown in figure 3.1. It is the same length as the suffix array and needs $4n$ bytes of memory. Before defining the child table, some notations and definitions are needed.

			childtab			
i	suftab	lcptab	1.	2.	3.	$S_{\text{suftab}[i]}$
0	2	0		②	6	aaacatat\$
1	3	2				aacatat\$
2	0	1	1	③	4	acaaacatat\$
3	4	3				acatat\$
4	6	1	3	5		atat\$
5	8	2				at\$
6	1	0	2	⑦	8	caaacatat\$
7	5	2				catat\$
8	7	0	7	⑨	10	tatat\$
9	9	1				t\$
10	10	0	9			\$

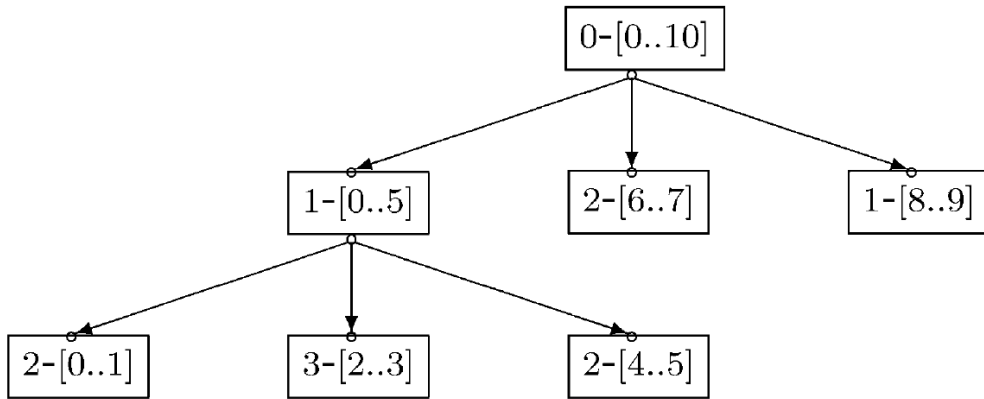


Figure 3.1: The enhanced suffix array (top) of the string $S = \text{acaaacatat}$ and its l -interval tree (bottom). Table values 1., 2. and 3. represent up, down and next l -index fields respectively. Circled values are redundant and can be left out while the arrows indicate field merging of the child table [19].

Definition 3.1. [19] An interval $[i..j]$, $0 \leq i < j \leq n$, is a lcp-interval of lcp-value l if

1. $lcptable[i] < l$,
2. $lcptable[k] \geq l, \forall k, i + 1 \leq k \leq j$,
3. $lcptable[k] = l$ for at least one $k, i + 1 \leq k \leq j$ (note: every such k is called a l -index),
4. $lcptable[j + 1] < l$.

Definition 3.2. [19] An m -interval $[l..r]$ is said to be embedded in an l -interval $[i..j]$ if it is a subinterval of $[i..j]$ (i.e., $i \leq l < r \leq j$) and $m > l$. The lcp-interval $[i..j]$ is then called the interval enclosing $[l..r]$. If $[i..j]$ encloses $[l..r]$ and there is no interval embedded in $[i..j]$ that also encloses $[l..r]$, then $[l..r]$ is called a child interval of $[i..j]$.

The parent-child relationships between intervals virtually creates a lcp-interval tree of the suffix array [19]. An example of it can be seen in figure 3.1. The lcp-interval tree is equivalent to the suffix tree without the leaves (although the lcp-interval tree has implicit singleton intervals $[l..l]$ [19]). This analogy allows top-down traversals of the tree and pattern search is one of them.

Back to the definition of the child table. It is designed to store information which helps to determine all child intervals of a given l -interval $[i..j]$ in constant time. There are three fields in this table: the up, the down and the next l -index field each holding $4n$ bytes in the worst case. The formal definitions can be find in [19]. Essentially, for a l -interval $[i..j]$ with l -indices $i_1 < i_2 < \dots < i_k$, the value $childtab[i].down$ or the value $childtab[j + 1].up$ stores the first l -index (i_1). The other l -indices can be obtained from $childtab[i_1].next_l\text{-index}$, \dots , $childtab[i_k - 1].next_l\text{-index}$. Some values in the child table are redundant, as seen in figure 3.1, which enables merging of the three fields to a single one and thus reducing the memory to just $4n$ bytes.

To summarize, the space complexity of an enhanced suffix array created from a string of n bytes is $13n$ bytes (each table needs $4n$ bytes plus n bytes for the original string). What is left to mention is the construction algorithm of each table. They are shown in table 3.1 with their space and time complexities. Details about each method can be found in cited papers.

3.1.1. Algorithms

In this section pseudocodes for algorithms based on the enhanced suffix array will be shown. First is the lcp-tree traversal, i.e. for a given l -interval $[i..j]$ how to find

Table 3.1: Enhanced suffix array construction

Table	Time	Space	Method
Suffix array	$O(n)$	$O(n)$	SA-IS algorithm - based on induced sorting [21]
LCP table	$O(n)$	$O(n)$	Suffix comparisons [22]
Child table	$O(n)$	$O(n)$	Traversal of the LCP table via stack [19]

a subinterval $[l..r]$ which common prefixes have the character c at position l . This algorithm is marked as algorithm 1.

Algorithm 1 Traversal of the lcp-interval tree [19]

```

1: function SUBINTERVAL( $i, j, c$ )
2:    $i_1 = i < childtab[i]$  and  $childtab[i] \leq j$  ?  $childtab[i]$  :  $childtab[j]$ 
3:    $l = lcptab[i_1]$ 
4:   if  $S[suftab[i] + l] == c$  then
5:     return ( $i, i_1 - 1$ )
6:   while  $childtab[i_1] \neq -1$  do
7:      $i_2 = childtab[i_1]$ 
8:     if  $S[suftab[i_1] + l] == c$  then
9:       return ( $i_1, i_2 - 1$ )
10:     $i_1 = i_2$ 
11:   if  $S[suftab[i_1] + l] == c$  then
12:     return ( $i_1, j$ )
13:   return ( $-1, -1$ )

```

The second algorithm is pattern search. For a given pattern P of length m find the m -interval $[i..j]$ where all suffixes share the same prefix which equals P . The time complexity of this algorithm is $O(m)$ and is shown bellow as algorithm 2. To find the number of times pattern P occurred in the string S , a simple formula can be used: $num_occurrences = j - i + 1$. If all positions of found occurrences are needed, iterating through the m -interval will do the job but will increase the time complexity to $O(m + z)$ where z is the number of occurrences [19].

The third and last algorithm is all prefix-suffix matches, i.e. for a given read find

Algorithm 2 Pattern search [19]

```
1: function INTERVAL_LCP_LENGTH( $i, j$ )
2:   if  $i < \text{childtab}[i]$  and  $\text{childtab}[i] \leq j$  then
3:     return  $\text{lcptab}[\text{childtab}[i]]$ 
4:   else
5:     return  $\text{lcptab}[\text{childtab}[j]]$ 
6:
7: function INTERVAL( $P, m$ )
8:    $c = 0$ 
9:    $\text{patternFound} = \text{true}$ 
10:  ( $i, j$ ) = SUBINTERVAL(0,  $n$ ,  $\text{pattern}[c]$ )
11:  while ( $i, j$ )  $\neq (-1, -1)$  and  $c < m$  do
12:    if  $i \neq j$  then
13:       $l = \text{INTERVAL\_LCP\_LENGTH}(i, j)$ 
14:       $\text{min} = l < m ? l : m$ 
15:       $\text{patternFound} = P[c..\text{min} - 1] ==$ 
16:         $S[\text{sufstab}[i] + c..\text{sufstab}[i] + \text{min} - 1]$ 
17:       $c = \text{min}$ 
18:      ( $i, j$ ) = SUBINTERVAL( $i, j$ ,  $\text{pattern}[c]$ )
19:    else ▷ singleton interval
20:       $\text{patternFound} = P[c..m - 1] ==$ 
21:         $S[\text{sufstab}[i] + c..\text{sufstab}[i] + m - 1]$ 
22:       $c = m$ 
23:    if not  $\text{patternFound}$  then
24:      break
25:  if  $\text{patternFound}$  then
26:    return ( $i, j$ )
27:  return  $(-1, -1)$ 
```

prefix-suffix overlaps with all other reads in the data set. When having a large amount of input strings (reads), it is common to concatenate them together with a delimiter character which is not present in the input alphabet. For example, having reads R_1 , R_2 and R_3 the resulting string would be $S = R_1\#R_2\#R_3\#$ upon which the enhanced suffix array is built. With the help of delimiters it is easy to distinguish suffixes of all reads but an additional dictionary is needed which maps absolute positions in S to read identifiers. The all prefix-suffix matches algorithm could be implemented directly with pattern search, for each prefix of the query string the delimiter would be added at its end and the algorithm 2 would be applied. Such direct approach has quadratic time complexity $O(m^2)$. To lower the complexity to $O(m)$ some modifications are needed as shown in algorithm 3.

Algorithm 3 All prefix-suffix matches

```

1: function PREFIX_SUFFIX_MATCHES( $P, m$ )
2:    $c = 0$ 
3:    $(i, j) = \text{SUBINTERVAL}(0, n, \text{pattern}[c])$ 
4:   while  $(i, j) \neq (-1, -1)$  and  $c < m$  do
5:     if  $i \neq j$  then
6:        $l = \text{INTERVAL\_LCP\_LENGTH}(i, j)$ 
7:        $\text{delimiter} = \text{find '}' in } S[\text{sufstab}[i] + c + 1..\text{sufstab}[i] + l]$ 
8:       if  $\text{delimiter} == -1$  then
9:          $\text{min} = l < m ? l : m$ 
10:        if  $P[c..\text{min} - 1] \neq S[\text{sufstab}[i] + c..\text{sufstab}[i] + \text{min} - 1]$  then
11:          break
12:         $c = \text{min}$ 
13:        if  $c == m$  then
14:          for  $o = i \rightarrow j$  do
15:            if  $S[\text{sufstab}[o] + m] == '}'$  then
16:              report  $(\text{read\_dictionary}[\text{sufstab}[o]], m) \triangleright (\text{id}, \text{length})$ 
17:            else  $\triangleright$  try to finish prefix
18:               $(b, d) = \text{SUBINTERVAL}(i, j, '}' )$ 
19:              if  $(b, d) \neq (-1, -1)$  then
20:                for  $o = b \rightarrow d$  do
21:                  report  $(\text{read\_dictionary}[\text{sufstab}[o]], \text{min})$ 
22:               $(i, j) = \text{SUBINTERVAL}(i, j, \text{pattern}[c])$ 

```

All prefix-suffix matches continued

```

23:         else                                     ▷ # exists in lcp of interval  $[i..j]$ 
24:              $len = delimiter - suftab[i]$ 
25:             if  $P[c..len - 1] == S[suftab[i] + c..suftab[i] + len - 1]$  then
26:                 for  $o = i \rightarrow j$  do
27:                     report ( $read\_dictionary[suftab[o]], len$ )
28:             break
29:         else                                     ▷ singleton interval
30:              $delimiter = \text{find '}' in } S[suftab[i] + c + 1..suftab[i] + c + m]$ 
31:              $len = delimiter - suftab[i]$ 
32:             if  $P[c..len - 1] == S[suftab[i] + c..suftab[i] + len - 1]$  then
33:                 report ( $read\_dictionary[suftab[i]], len$ )
34:             break

```

3.2. Error correction

As stated before, sequencing technologies produce reads with a certain error rate and sometimes it is good to perform error correction before the whole assembly process. To keep in mind is the transcriptome sequencing depth, which can vary a lot due to gene expression. Reads from the less abundant transcripts could be wrongly marked as erroneous.

Nevertheless, this thesis' error correction is an optional phase and is based on k -mer frequencies [23]. Each k -mer of a read is checked if it occurs more than c times in the whole data set and if so it is called a solid k -mer. All bases that are not covered by a solid k -mer need correction. A read is corrected if there exists a set of operations so that all erroneous bases are corrected, otherwise it is left intact. This method is shown in algorithm 4.

3.3. Overlap phase

In a nutshell, the overlap phase is finding overlaps between reads. Each read is compared to every other read and its reverse complement¹, because the strand from which the read was sequenced is unknown. In this thesis the enhanced suffix array is used to find exact prefix-suffix and suffix-prefix overlaps (algorithm 3). Before going into details,

¹Reverse complement of a sequence S is a sequence \overline{S} which contains base pairs of nucleotides from sequence S in reversed order.

Algorithm 4 Error correction [23]

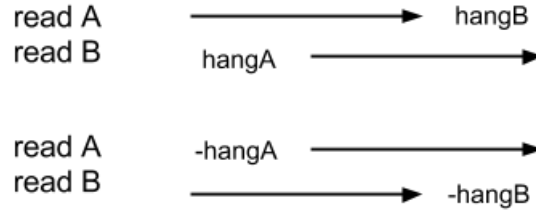
```
1: function CORRECT_READS(reads, k, c)
2:   esa = CREATE_ENHANCED_SUFFIX_ARRAY(reads)
3:   for read  $\in$  reads do
4:     correct = true
5:     for i = 0; i < read.length - k + 1 do
6:       if esa.NUM_OCCURRENCES(read[i..i + k - 1], k) > c then
7:         i += k
8:         continue
9:       l = max(0, i - k + 1) ▷ left most covering k-mer
10:      if a substitution at position i exists such as
11:        esa.NUM_OCCURRENCES(read[l..l + k - 1], k) > c then
12:          store substitution
13:          i = l + k
14:          continue
15:      r = min(i, read.length - k) ▷ right most covering k-mer
16:      if a substitution at position i exists such as
17:        esa.NUM_OCCURRENCES(read[r..r + k - 1], k) > c then
18:          store substitution
19:          i = r + k
20:          continue
21:      correct = false ▷ unable to correct base, drop read
22:      break
23:    if correct then
24:      apply stored substitutions
```

there are two conventions used for overlaps:

1. Id of the first read in an overlap is **always smaller** than the id of the second one.
2. First read in an overlap is **never** a reverse complement.

The first convention is needed for an easy way to pick the optimal overlap for a pair of reads. Algorithm 3 can report suboptimal overlaps due to repeats in reads, both prefix-suffix and suffix-prefix overlaps and both normal and reverse complement overlaps can exist simultaneously. The second convention is used to avoid duplication of data, i.e. overlapping two normal reads (A and B) is the same as overlapping their reverse complements (\overline{A} and \overline{B}). The same stands when only one read is a reverse complement (A and \overline{B} versus \overline{A} and B). Types and descriptions of possible overlaps with such conventions can be seen in figure 3.2.

Normal overlaps



Innie overlaps

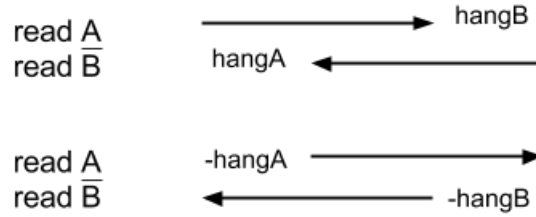


Figure 3.2: Different overlap types. Normal type overlaps are those where both reads are normal while the innie type indicates that read B is a reverse complement. A hang of a read equals the number of its bases before or after the overlap. If the prefix of A matches the suffix of B the hangs will be negative [24].

To find all mentioned overlap types using the algorithm 3, two enhanced suffix arrays need to be constructed. The first one consists of normal reads and the other of reverse complements. All normal overlaps can be retrieved by matching each read against the first array and applying the first convention. To retrieve all innie overlaps, first

each normal read needs to be matched with the second array and only matches with ids greater than the query id are picked. Afterwards, each reverse complement needs to be matched with the first array and only matches with ids smaller than the query id will be picked. Picking matches in such manner fulfils mentioned conventions. Furthermore, it is common to declare a minimum overlap length to decrease the number of overlaps found. For clarification the whole pseudocode is shown in algorithm 5.

Algorithm 5 Overlap phase

```

1: function OVERLAP_READS(reads, minOverlapLen)
2:   reads_rk = CREATE_REVERSE_COMPLEMENTS(reads)
3:   esa = CREATE_ENHANCED_SUFFIX_ARRAY(reads)
4:   esa_rk = CREATE_ENHANCED_SUFFIX_ARRAY(reads_rk)
5:   overlaps = [ ]
6:   ▷ operation ← creates tuples by adding read.id to the match pairs (id, length)
7:   for read ∈ reads do
8:     ▷ normal prefix-suffix and suffix-prefix overlaps
9:     overlaps ← all matches with length > minOverlapLen from
10:      esa.PREFIX_SUFFIX_MATCHES(read, read.length)
11:     ▷ innie suffix-prefix overlaps
12:     overlaps ← all matches with length > minOverlapLen and
13:      matches with id < read.id from
14:      esa_rk.PREFIX_SUFFIX_MATCHES(read, read.length)
15:   for read ∈ reads_rk do
16:     ▷ innie prefix-suffix overlaps
17:     overlaps ← all matches with length > minOverlapLen and
18:      matches with id > read.id from
19:      esa.PREFIX_SUFFIX_MATCHES(read, read.length)
20:   ▷ pick only the best overlap for a pair of reads
21:   sort overlaps in descending order
22:   prev = nil
23:   for overlap ∈ overlaps do
24:     if overlap.reads ≠ prev.reads then
25:       report overlap
26:     prev = overlap

```

3.4. Layout phase

The huge amount of generated overlaps conceptually creates an overlap graph which tends to be big and intricate. Task of the layout phase is to extract contigs from such graphs. To ease up the task, methods for graph simplification need to be applied first. That includes removal of redundant overlaps (containment and transitive [25]) and advanced methods which are based on the string graphs². Even after simplification some unresolved parts of the graph often remain. Assemblers tend to extract contigs from contiguous parts of the graph and fill the gaps afterwards, but in this thesis a heuristic algorithm was implemented to extract the longest path from a given graph. In this section each of the simplification methods will be explained as well as the heuristic extraction algorithm.

3.4.1. Graph simplification

To simplify an overlap graph the redundant data needs to be removed. This includes removal of containment and transitive overlaps which descriptions are shown bellow:

1. Removal of containment overlaps is straightforward, if a read is contained by another read it is removed from the graph with all of its overlaps. The pseudo-code for this method is shown in algorithm 6.
2. A given overlap (o_1) is transitive considering overlaps o_2 and o_3 if the following stands ($o_x.part(y)$ returns which end the read y is using in overlap o_x , possible values: begin or end):

$$\begin{aligned} o_1.part(A) &= o_2.part(A) \\ o_1.part(B) &= o_3.part(B) \\ o_2.part(C) &\neq o_3.part(C) \end{aligned} \tag{3.1}$$

Alongside this conditions the overlaps must refer to the same data. To avoid comparing the actual data (which can be a costly task) a position-based heuristic [26] is used which includes two formulas ($o_x.hang(y)$ returns the hanging length of read y in overlap o_x):

$$\begin{aligned} o_2.hang(A) + o_3.hang(C) &\in o_1.hang(A) \pm (\epsilon \cdot o_1.length + \alpha) \\ o_2.hang(C) + o_3.hang(B) &\in o_1.hang(B) \pm (\epsilon \cdot o_1.length + \alpha) \end{aligned} \tag{3.2}$$

²String graph is a bidirectional graph build from reads and overlaps between them. Reads are equivalent to vertices and overlaps to edges. Each edge direction has a label which equals to the hanging part of the read it points to [25].

An example with a detailed look into different overlap parts is shown in figure 3.3. After the given overlap is declared as transitive, it can be removed from the graph without any loss of information. Although, the removal should be done at the end of the algorithm because there might be a transitive overlap considering other transitive overlaps as shown in figure 3.4. Pseudocode for this method is shown in algorithm 7.

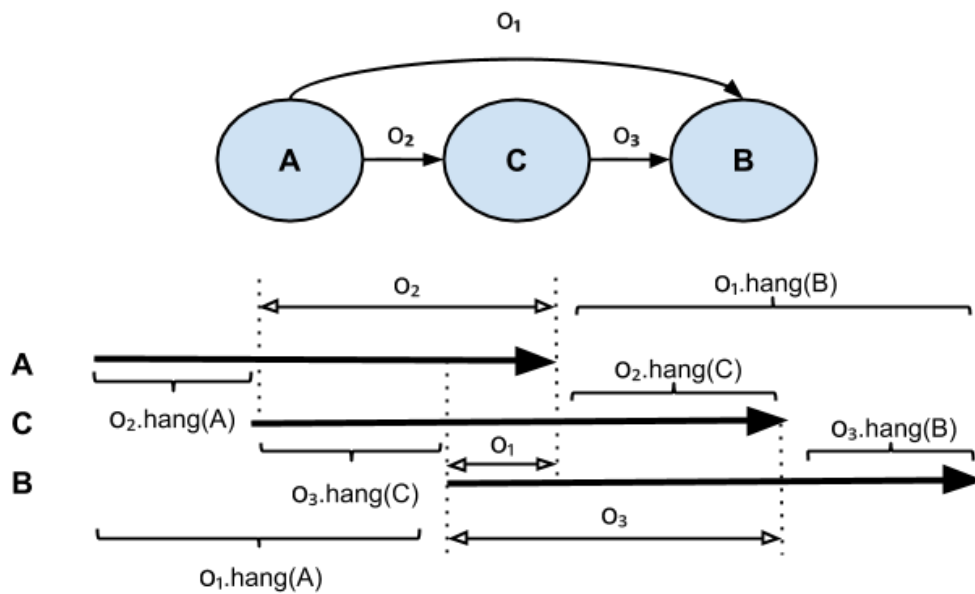


Figure 3.3: Detailed example of an overlap graph with a transitive overlap (o_1)

When the overlap graph is refined a string graph can be build which is needed for further simplification. Due to sequencing errors, discontinuities like tips and bubbles[27] often appear. Tips are "dangling" vertices, meaning that they have edges in only one direction and should be purged. On the other hand, bubbles are structures consisting of two or more paths which share the starting (called *root*) and the ending vertex (called *junction*) and contain similar sequences. To determine the similarity of two sequences the edit distance³ can be used. Examples of a tip and a bubble can be seen in figure 3.5.

³Edit distance is the minimum number of operations needed to transform one string to another [28].

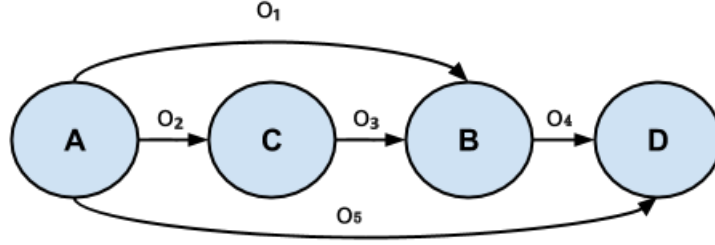


Figure 3.4: Example of a transitive overlap (o_5) considering another transitive overlap (o_1)

Algorithm 6 Removal of containment overlaps

```

1: function FILTER_CONTAINMENT_OVERLAPS(overlaps)
2:   marked = {}
3:   for overlap  $\in$  overlaps do
4:     if overlap.hangA  $\leq$  0 and overlap.hangB  $\geq$  0 then
5:        $\triangleright$  read A is contained in read B
6:       marked  $\leftarrow$  overlap.readA
7:       continue
8:     if overlap.hangA  $\geq$  0 and overlap.hangB  $\leq$  0 then
9:        $\triangleright$  read B is contained in read A
10:      marked  $\leftarrow$  overlap.readB
11:      continue
12:   for overlap  $\in$  overlaps do
13:     if overlap has at least one read  $\in$  marked then
14:       delete overlap

```

Algorithm 7 Removal of transitive overlaps

```
1: function OVERLAP::IS_TRANSITIVE( $o_2, o_3$ )  $\triangleright o_1 = \text{this}$ 
2:    $A = o_1.read_A$ 
3:    $B = o_1.read_B$ 
4:    $C = o_2.read_A == A ? o_2.read_B : o_2.read_A$ 
5:   if  $o_2.part(C) == o_3.part(C)$  then
6:     return false
7:   if  $o_1.part(A) \neq o_2.part(A)$  then
8:     return false
9:   if  $o_1.part(B) \neq o_3.part(B)$  then
10:    return false
11:   if  $o_2.hang(A) + o_3.hang(C) \notin o_1.hang(A) \pm \epsilon \cdot o_1.length + \alpha$  then
12:     return false
13:   if  $o_2.hang(C) + o_3.hang(B) \notin o_1.hang(B) \pm \epsilon \cdot o_1.length + \alpha$  then
14:     return false
15:   return true
16:
17: function FILTER_TRANSITIVE_OVERLAPS( $overlaps$ )
18:   for  $o_1 \in overlaps$  do
19:     for  $(o_2, o_3) \in (\text{overlaps of } o_1.read_A \cup \text{overlaps of } o_1.read_B)$  do
20:       if  $o_1.IS\_TRANSITIVE(o_2, o_3)$  then
21:         mark  $o_1$ 
22:   for  $overlap \in overlaps$  do
23:     if  $overlap$  is marked then
24:       delete  $overlap$ 
```

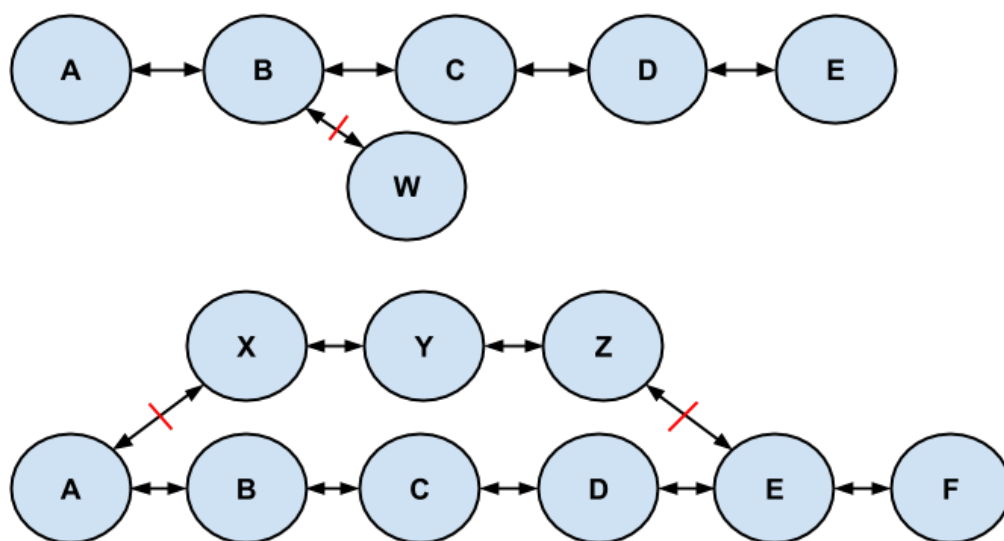


Figure 3.5: Example of string graphs containing a tip (top) and a bubble (bottom)

To remove discontinuities from the string graphs, methods called trimming and bubble popping (a modified version of [27], similar to the ones found in the implementation of SGA [23]) were implemented which descriptions are as follows:

1. Trimming is a method where disconnected and dangling vertices are removed. Disconnected vertices do not have any edges and in transcriptome assembly one short read often does not form a transcript. Dangling vertices (i.e. tips) are removed if any of the vertices they are holding onto has a similar edge pointing to a vertex which is not a tip. The pseudocode is shown in algorithm 8.
2. The bubble popping method is a bit complex. It first locates bubbles in the string graph with breadth first search (BFS). For each bubble found it determines the path with the highest coverage. At the end it removes every other path in the bubble which sequence is at least 90% equal to the selected path's sequence. Pseudocode for bubble removal is shown algorithm 9.

Each execution of one of these methods can enable further simplification possibilities and therefore multiple executions are advised. Removal of almost all tips and bubbles can be achieved by calling both methods in an alternating fashion by keeping in mind that bubble popping is far more slower and should be called less often than trimming.

Algorithm 8 Trimming

```
1: function VERTEX::IS_TIP_CANDIDATE()  
2:   ▷  $edges_B$  contains edges where the beginning of vertex is in overlap  
3:   ▷  $edges_E$  contains edges where the end of vertex is in overlap  
4:   if  $this.edges_B.size() == 0$  or  $this.edges_E.size() == 0$  then  
5:     return true  
6:   return false  
7:  
8: function STRING_GRAPH::TRIM()  
9:   for  $vertex \in this.vertices$  do  
10:    if  $vertex.edges_B.size() == 0$  and  $vertex.edges_E.size() == 0$  then  
11:      ▷ disconnected vertex  
12:      mark  $vertex$   
13:      continue  
14:    if  $vertex.IS\_TIP\_CANDIDATE()$  then  
15:       $edges = vertex.edges_B$  or  $vertex.edges_E$   
16:      for  $edge \in edges$  do  
17:        ▷ check if the opposing vertex has a similar edge  
18:         $overtex = edge.opposite(vertex)$   
19:         $oedges = edge.overlap.part(overtex) == B ?$   
20:           $overtex.edges_B : overtex.edges_E$   
21:         $isTip = false$   
22:        for  $oedge \in oedges$  do  
23:          ▷  $vertex$  is a tip only if the vertex on the similar edge is not  
24:          if not  $oedge.opposite(overtex).IS\_TIP\_CANDIDATE()$  then  
25:             $isTip = true$   
26:            break  
27:        if  $isTip$  then  
28:          mark  $vertex$   
29:          break  
30:    for  $vertex \in this.vertices$  do  
31:      if  $vertex$  is markex then  
32:        delete  $vertex$  and all correlated edges
```

Algorithm 9 Bubble popping

```
1: function VERTEX::IS_BUBBLE_ROOT_CANDIDATE(direction)
2:   if direction == B and edgesB.size() > 1 then
3:     return true
4:   if direction == E and edgesE.size() > 1 then
5:     return true
6:   return false
7:
8: function STRING_GRAPH::FIND_BUBBLE(root, direction)
9:   ▷ Breath First Search until a juncture vertex is found
10:  openedQue = [root]
11:  closedQue = [ ]
12:  bubble = [ ]
13:  while not que.empty() do
14:    expandQue = [ ]
15:    while not que.empty() do
16:      vertex = openedQue.pop_front()
17:      if not vertex.EXPAND(expandQue, direction) then
18:        closedQue ← vertex
19:      openedQue.swap(expandQue)
20:      if a duplicate exists in (openedQue ∪ closedQue) then
21:        for each duplicate do                                     ▷ juncture vertex
22:          path = backtrack duplicate to root
23:          bubble ← path
24:        break
25:    ▷ orientation of the juncture vertex can vary in different paths
26:    bubbleB = pick paths from bubble where the juncture uses its beginning
27:    bubbleE = pick paths from bubble where the juncture uses its end
28:    if bubbleB.size() > 1 then
29:      return bubbleB
30:    if bubbleE.size() > 1 then
31:      return bubbleE
32:  return nullptr
```

Bubble popping continued

```
1: function STRING_GRAPH::POP_BUBBLE(bubble)
2:   sequences = [ ]
3:   for path  $\in$  bubble do
4:     sequences  $\leftarrow$  path.extract_sequence()
5:   bestPath = select path  $\in$  bubble with the highest coverage
6:   for  $i = 0 \rightarrow \text{sequences.size}()$  do
7:     if  $i == \text{bestPath}$  then
8:       continue
9:     distance = EDIT_DISTANCE(sequences[i], sequences[bestPath])
10:    if  $\text{distance} \div \text{sequences}[\text{bestPath}].\text{size}() < \text{maxDifference}$  then
11:       $\triangleright$  path bubble[i] can only be removed if all of its vertices (except root,
12:        junction) do not have any external edges
13:      if  $\forall \text{vertex} \in \text{bubble}[i] \setminus \{\text{root}, \text{junction}\}$ 
14:        vertex.edgesB.vertices  $\in$  bubble[i] and
15:        vertex.edgesE.vertices  $\in$  bubble[i] then
16:        delete bubble[i] from graph
17:
18: function STRING_GRAPH::POP_BUBBLES( )
19:   bubbles = [ ]
20:   for vertex  $\in$  this.vertices do
21:     for direction  $\in$  {B, E} do
22:       if not vertex.IS_BUBBLE_ROOT_CANDIDATE(direction) then
23:         continue
24:       bubbles  $\leftarrow$  this.FIND_BUBBLE(vertex, direction)
25:   for bubble  $\in$  bubbles do
26:     this.POP_BUBBLE(bubble)
```

3.4.2. Contig extraction

Last step of the layout phase is contig extraction, i.e. joining reads from contiguous parts of the string graph. It is not rare that the graph has unresolved parts even after applying simplification methods. Those parts become gaps between contigs and need to be filled with methods which will not be discussed in this thesis. In transcriptome assembly the string graph tends to split into smaller components which extraction needs to be done first (a straightforward task). To avoid troublesome gaps between contigs, a heuristic method was implemented which tries to extract the longest path⁴ from a graph component. Its step-by-step description is as follows:

1. Find all possible starting vertices and pick the top N . The ranking criterion is the length of the contiguous path from a starting vertex to the first branching vertex.
2. For each of the best starting points try to extract a path as long as possible. Expand the graph gradually with a new vertex until an end point is reached or a cycle is formed. If a branching vertex occurs, recursively determine which branch to take (again the criterion is the path length). Terminate the recursion if M subsequent branching vertices are reached or again if a cycle is formed. To efficiently detect cycles, the expand function and the recursion share a set of visited vertices. All vertices visited during a recursive call have to be removed from the set before the recursive call ends because the recursion is used only for decision making.
3. Pick the longest of all found paths.

This heuristic algorithm is not as complex as other layout phase methods so no pseudocode is provided.

3.5. Consensus phase

In the consensus phase ambiguities in found contigs need to be resolved. All reads that make a contig are aligned together, with a multiple sequence alignment method, and at each base a majority vote takes place [29]. In this thesis' implementation only exact overlaps are produced and there is no need for a consensus phase at this moment. Transcripts can and are provided directly from graph components at the end of the

⁴Graphs can be cyclic which makes the longest path problem NP-hard and no polynomial-time solution is known.

layout phase. Nevertheless, consensus is a part of this assembler and is meant for future use when inexact overlaps might be added. It uses the partial order alignment (POA)[30] algorithm which is based on partial order graphs. The algorithm was not implemented directly but is used from an external library and will not be discussed into any details.

4. Implementation

This chapter includes a brief overview of this thesis' implementation, its external dependencies and code structure.

4.1. Overview

The name of the implemented OLC transcriptome assembler is **Ra** which is short for **R**NA **a**ssembler. It is entirely written in the C++ programming language and is documented according to Doxygen conventions (both html and latex documentations will be created if Doxygen is run). The whole project is freely available at <https://github.com/rvaser/ra> with usage instructions written in *readme* files.

Ra was designed to be a library of assembly tools. The core functions and classes can be found in the *ra/src* directory while the external dependencies are found in *ra/vendor*. Code statistics of the implementation's core directory can be seen in figure 4.1. After the *Makefile* is run from the root directory, the static library *libra.a* is created which holds the core functionality needed by all Ra modules. There are five modules in total and each of them consists of a *main.cpp* file and a *Makefile*. Their names and short descriptions are shown below:

1. *ra_correct* - Module is optional and is used to correct reads before the assembly process (it should be used before *ra_overlap*).
2. *ra_overlap* - Module is used to find all (prefix-suffix and suffix-prefix) overlaps between single-end reads. It also removes containment and transitive overlaps to avoid storing a large amount of data to a file.
3. *ra_layout* - Module is used to create a string graph from a set of overlaps (which are not containment nor transitive). It then simplifies the graph with trimming and bubble popping methods. At the end it tries to extract the longest path from

```

22 text files.
22 unique files.
0 files ignored.

http://cloc.sourceforge.net v 1.53  T=0.5 s (44.0 files/s, 10592.0 lines/s)
-----
Language             files            blank           comment           code
-----
C++                   10              1023             157              2262
C/C++ Header          12              231             1181              442
-----
SUM:                  22             1254            1338             2704
-----

```

Figure 4.1: Code statistics of the implementations' core directory obtained with cloc

every graph component. As the current overlapper is exact, it also extracts whole transcripts so that the consensus phase can be avoided.

4. *ra_consensus* - Module is used to build consensus sequences (i.e. transcripts) with the help of the POA algorithm.
5. *to_afg* - Module is used to convert read sets from FASTA[31] or FASTQ[32] format to the afg (AMOS[33]) format. It is necessary to convert reads because all other modules are using the afg format.

As mentioned before, Ra has external dependencies (all publicly available on GitHub) which will be briefly described in the next section.

4.2. External dependencies

4.2.1. Afgreader

Afgreader is a C++ implementation of the afg format reader. It was written by Mario Kostelac and is publicly available at <https://github.com/mariokostelac/afgreader>. The class is utilized for reading files in afg format and was wrapped in the *IO.cpp* file. It supports many AMOS message types but it is mainly used for input of reads and overlaps.

4.2.2. EDLIB

EDLIB is a C/C++ library for sequence alignment using the edit distance. It was written by Martin Šošić and is publicly available at <https://github.com/Martinsos/edlib>. In the Ra project it was wrapped in the *EditDistance.cpp* file and is mainly

used for edit distance retrieval of two sequences. This distance is used in the bubble popping procedure to determine if different paths in a bubble are similar enough before removal.

4.2.3. CPPPOA

CPPPOA is a C++ implementation of the Partial Order Alignment algorithm. It was written by Marko Čulinović and is publicly available at <https://github.com/mculinovic/cpppoa>. It was wrapped in the *PartialOrderAlignment.cpp* file and is used in the consensus phase. Its main task is to retrieve the consensus sequence (i.e. transcript) from a set of reads.

4.3. Code structure

To use Ra as a library the header file *ra.hpp* needs to be included as well. It includes main header files from the implementations' core and its dependency tree can be seen in figure 4.2. The Ra core is organized in multiple classes and files which descriptions are shown below:

- *CommonHeaders.hpp* - includes often used standard libraries like *stdlib.h*, *stdio.h*, *string*, *vector* etc.
- *EnhancesSuffixArray.cpp* / *EnhancesSuffixArray.hpp* - class *EnhancesSuffixArray* which is used for pattern search.
- *IO.cpp* / *IO.hpp* - contains functions for input and output of *Read*, *Overlap* and *Contig* objects. It also has functions for input and output of byte buffers needed for caching *EnhancedSuffixArray* and *ReadIndex* objects.
- *Overlap.cpp* / *Overlap.hpp* - class *Overlap* which represents an overlap between two *Read* objects. It also contains methods for filtering of containment and transitive *Overlap* objects.
- *Preprocess.cpp* / *Preprocess.hpp* - contains functions for error correction and duplicate filtering of *Read* objects.
- *PartialOrderAlignment.cpp* / *PartialOrderAlignment.hpp* - contains a function for generating the consensus sequence from a *Contig* object.
- *Read.cpp* / *Read.hpp* - class *Read* which represents read sequences.

- *ReadIndex.cpp / ReadIndex.hpp* - class *ReadIndex* which is a wrapper for *EnhancedSuffixArray*. It was implemented to keep the memory complexity at $13n$ (n being the length of all reads) when the input data is too big and to concatenate read sequences to a single string. It also contains prefix suffix and pattern matching methods for *Read* objects.
- *StringGraph.cpp / StringGraph.hpp* - class *StringGraph* and its helper classes: *Edge*, *Vertex*, *StringGraphNode*, *StringGraphWalk*, *StringGraphComponent* and *Contig*. Contains methods for string graph simplification and contig extraction.
- *Utils.cpp / Utils.hpp* - class *Timer* and its methods for measuring and printing of the elapsed time.

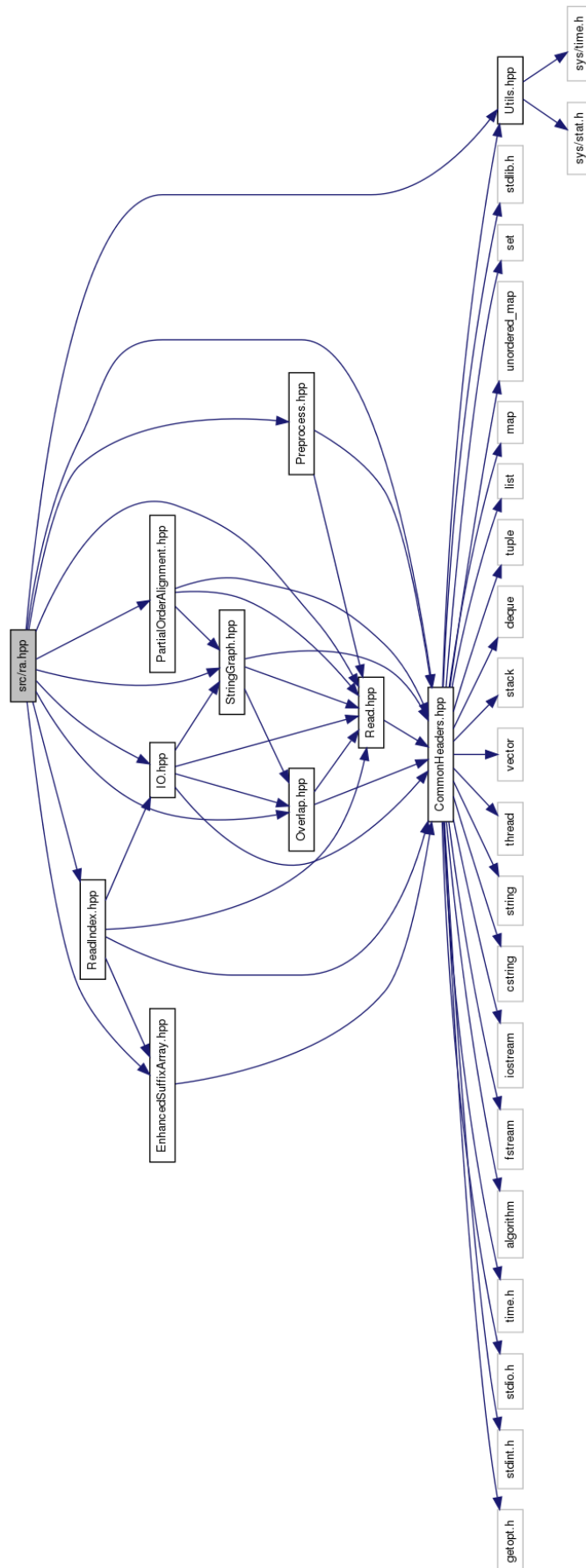


Figure 4.2: Dependencies tree of Ra's core header file

5. Results

To see how the Ra assembler performs, two benchmark tests were executed where the transcriptome is assembled and aligned to the corresponding genome. In this section first the performed tests will be explained in detail and afterwards a discussion about results and possible future improvements will be given.

5.1. Testing

Both benchmark tests were found within the SOAPdenovo-Trans paper [16] and were applied on the Ra assembler. The tests include execution time, memory usage, number of transcripts found and how they align to corresponding genomes. They include the rice and the mouse transcriptome data. The paper shows two versions of each test, one using a smaller version of the data sets and the other one using full sets. In this thesis only the tests with the smaller versions were executed and the description of each transcriptome set is as follows:

- *rice transcriptome set* - generated from the *Oryza sativa 9311* on an Illumina Genome Analyzer. It contains 9.8 million paired-end reads with length of 75bp (short for base pairs). This set is available for download at <http://www.ncbi.nlm.nih.gov/sra/SRX017631>. It is part of the full set which other parts can be obtained at the same web adress with different names: *SRX017631*, *SRX017632* and *SRX017630*.
- *mouse transcriptome set* - generated from the *Mus musculus* on an Illumina Genome Analyze II. It contains 52.6 million paired-end reads with length of 76bp. The set is available for download at <http://www.ncbi.nlm.nih.gov/sra/SRX062280>. First a script from the supplementary files [16] needs to be executed for quality filtering. Afterwards, the data needs to be manually down-sampled by extracting one of every three reads [16] and thus the smaller version is obtained which contains 12 million read pairs.

As mentioned before, Ra supports only single-end reads so the paired-end data was split into two files each containing one half of each read pair. To split and convert the files to FASTQ format the SRA toolkit [34] was used. After the splitting, the two created files were concatenated together and utilized for Ra's assembly process as they would contain single-end reads.

Testing of the Ra assembler was performed in parallel with 10 threads, as the other assemblers in the paper [16], on a machine with following specifications:

- Software: Ubuntu 14.04.2 LTS
- Architecture: x86_64
- Processor: Intel(R) Xeon(R) CPU E5645 @ 2.4GHz
- Cores: 12
- RAM: 192GB

Ra is an OLC based assembler and it was unclear how to "fairly" set up its parameters so they match to the De Bruijn graph assemblers (they are using k -mers of length 25 [16]). The error correction, as it is based on k -mer frequencies, was set to use the k -mer length 25 while the minimal overlap length was set to 35 (almost the half of the reads length) so that Ra could compete with the DBG assemblers.

Results showing the comparison between Ra, SOPAdenovo-Trans, Oases and Trinity can be seen below scattered into several tables. Values for the other three assemblers were found in the SOAPdenovo-Trans paper [16]. In table 5.1 peak memory usage and the execution times are shown. Table 5.2 is showing how many transcripts were produced by each of the assemblers and how many genes or isoforms were recovered. Only the basic Ra results are shown because error correction did not impact the final transcripts, most likely due to the high minimal overlap length. The found transcripts are first filtered so that they contain only those longer than 300bp (simple bash script). After the filtering is done they are aligned to the corresponding genome with BLAT [35] using minimal sequence identity of 95%. Transcripts marked as *correct* are those which have one consistent alignment with $\geq 95\%$ of their length included. If they have two or more such alignments whether on different chromosomes or with different orientations they are marked as *chimeric*. At the end, the *correct* alignments are checked if they cover any annotation entirely or at least 95%. Scripts for alignment and annotation checking were as well found in the supplementary files [16]. A detailed look into execution times and descriptions of each of Ra's modules can be seen in tables 5.3 and 5.4. As mentioned before, the consensus phase was not executed due to the exact overlap phase.

Table 5.1: Computational requirements

Method	Rice		Mouse	
	Peak memory (GB)	Time (h)	Peak memory (GB)	Time (h)
Ra	23	2.0	30	10.2
Ra + correction	25	2.5	30	10.7
SOAPdenovo-Trans	10.7	0.2	10.5	0.3
Trinity	11	4.3	11	4.5
Oases	9.9	0.4	9.1	0.5

Table 5.2: Classification of assembled transcripts

	Rice				Mouse			
	Ra	SOAPdenovo-Trans	Trinity	Oases	Ra	SOAPdenovo-Trans	Trinity	Oases
>100 bp	176363	61425	107403	64490	147640	48224	96551	42993
>300 bp	18981	25800	37548	36097	15831	16286	29900	27598
Correct	8457	23682	31764	30001	5767	15959	28239	26005
Correct %	44.5	91.8	84.6	83.1	36.4	98.0	94.4	94.2
Chimeric	3782	526	2021	2185	6642	170	1101	757
Chimeric %	19.9	2.0	5.4	6.1	42.0	1.0	3.7	2.7
Coverage = 100%								
Genes	41	386	472	355	211	2897	3071	2984
Isoforms	46	405	524	382	234	3505	3939	3922
Coverage \geq 95%								
Genes	100	1904	1780	1469	470	6000	5090	5563
Isoforms	113	2300	2229	1849	607	9043	7619	8975

Table 5.3: Detailed look of Ra’s execution times for the rice data set

Method	Overview	Time (min)
ra_correct		31.50
↔ Input	19 600 898 reads	1.61
↔ Error correction	4.18% reads corrected	24.07
↔ ESA construction		22.06
↔ Output	19 600 898 reads	5.82
ra_overlap		59.34
↔ Input	19 600 898 reads	1.67
↔ Read duplicate filtering	removed 32.61% reads	16.21
↔ ESA construction		15.63
↔ Read overlapping	found 222 974 107 overlaps	24.39
↔ ESA construction (x 2)		19.46
↔ Containment filtering	removed 10.98% overlaps	0.33
↔ Transitive filtering	removed 72.84% overlaps	3.40
↔ Output	19 600 898 reads and 18 689 694 overlaps	13.34
ra_layout		57.58
↔ Input	19 600 898 reads and 18 689 694 overlaps	3.03
↔ String graph construction		3.02
↔ String graph simplification		47.41
↔ Trimming (x 26)	removed 4 450 199 tips and 9 238 144 disconnected vertices	4.88
↔ Bubble popping (x 7)	removed 35 848 bubbles	42.53
↔ Contig extraction	extracted 236 700 contigs	3.76
↔ Output	236 700 contigs	0.36

Table 5.4: Detailed look of Ra’s execution times for the mouse data set

Method	Overview	Time (min)
ra_correct		30.31
↔ Input	24 093 968 reads	2.00
↔ Error correction	2.63% reads corrected	20.68
↔ ESA construction		18.81
↔ Output	24 093 968 reads	7.63
ra_overlap		73.91
↔ Input	24 093 968 reads	2.07
↔ Read duplicate filtering	removed 37.18% reads	18.95
↔ ESA construction		18.28
↔ Read overlapping	found 445 272 532 overlaps	29.34
↔ ESA construction (x 2)		22.47
↔ Containment filtering	removed 20.70% overlaps	0.88
↔ Transitive filtering	removed 70.66% overlaps	6.66
↔ Output	24 093 968 reads and 38 471 546 overlaps	16.01
ra_layout		540.39
↔ Input	24 093 968 reads and 38 471 546 overlaps	3.46
↔ String graph construction		3.21
↔ String graph simplification		469.02
↔ Trimming (x 39)	removed 4 708 702 tips and 11 089 563 disconnected vertices	8.23
↔ Bubble popping (x 9)	removed 45 750 bubbles	460.79
↔ Contig extraction	extracted 197 181 contigs	64.41
↔ Output	197 181 contigs	0.29

5.2. Discussion

The testing has shown that Ra's performance is currently not satisfying. The high memory consumption, seen in table 5.1, is expected due to the $13n$ memory complexity of enhanced suffix arrays and not much can be done about it. The high memory requirements were the reason the de Bruijn graphs were introduced to bioinformatics in the first place. In addition, construction of the suffix arrays takes a good portion of the execution time considering other parts of the correction phase or the overlap phase as shown in tables 5.3 and 5.4. For that matter they are cached for further runs on the same data. The tables also shows that the most time consuming step of the layout phase is the bubble popping procedure. It needs drastic improvements which might include parallelization and a smarter bubble search which would ignore graph areas that have not changed between two subsequent searches. Altogether, Ra's execution times are higher when compared to the other assemblers which is expected due to the OLC paradigm.

Classification of the assembled transcripts can be seen in table 5.2. For both data sets, Ra finds the most "transcripts" although 90% of them are shorter than 300 base pairs. This might be due to the high overlap length or absence of inexact overlaps which is more probable. Inexact overlaps could be added to the prefix-suffix matches algorithm described in chapter 3. Insertions, deletions or substitutions would be allowed at each position up to a predetermined number of mismatches. Another problem with Ra is that a lot of the longer transcripts are declared as *chimeric*. The contig extraction algorithm is heuristic and most probably the cause of this issue. It could be solved by implementing a new algorithm for longest path extraction or by implementing a scaffold which would fill the gaps between contiguous areas of the graph components. At the end of table 5.2 number of retrieved genes and isoforms are shown. Ra finds only a small fraction (5 – 10%) of genes and isoforms in comparison to other RNA assembler. The cause of such poor performance could be tied to all of the mentioned possibilities or some other issues are at stake which are not known at this moment.

In the future, Ra should definitely be upgraded with inexact overlaps. This addition will increase the execution time of the overlap phase but it is an essential step towards a more efficient transcriptome assembler. Bubble popping and contig extraction should be reimplemented as well.

6. Conclusion

Main focus of this thesis was transcriptome assembly, i.e. implementing a de novo assembler in the C++ programming language. As the result Ra emerged, short for RNA assembler, which is based on the overlap-layout-consensus paradigm. It was designed as a library of assembly tools which methods were brought up and explained throughout the thesis. To obtain its performance, tests were conducted on rice and mouse transcriptome data sets. They included memory consumption, execution time and number of genes and isoforms retrieved. Results were compared to three other transcriptome assemblers which are based on the de Bruijn graphs. With the state Ra is at the moment, it could not compete with the other assemblers due to the OLC paradigm boundaries. Lack of inexact overlaps and the heuristic algorithm for longest path retrieval in graphs had their impact on the results as well.

On the bright side, design of the implementation enables easy improvements and can serve as a starting point in building a better and more efficient de novo OLC transcriptome assembler.

REFERENCES

- [1] NHGRI. All about the Human Genome Project (HGP). National Human Genome Research Institute. [Online]. Available: www.genome.gov/10001772
- [2] K. Wetterstrand. Dna sequencing costs: Data from the NHGRI Genome Sequencing Program (GSP). [Online]. Available: www.genome.gov/sequencingcosts
- [3] A. Adnan. (2010) DNA Sequencing: Method, benefits and applications. [Online]. Available: www.biotecharticles.com/Genetics-Article/DNA-Sequencing-Method-Benefits-and-Applications-248.html
- [4] Z. Khatoon, B. Figler, H. Zhang, and F. Cheng, “Introduction to RNA-Seq and its applications to drug discovery and development,” *Drug Development Research*, 2014.
- [5] C. A. Maher, C. Kumar-Sinha, X. Cao, S. Kalyana-Sundaram, B. Han, X. Jing, L. Sam, T. Barrette, N. Palanisamy, and A. M. Chinnaiyan, “Transcriptome sequencing to detect gene fusions in cancer,” *Nature*, 2009.
- [6] Genohub. Choosing the right NGS sequencing instrument for your study. [Online]. Available: <https://genohub.com/ngs-instrument-guide/>
- [7] J. B. W. Wolf, “Principles of transcriptome analysis and gene expression quantification: an rna-seq tutorial,” *Molecular ecology resources*, 2013.
- [8] Illumina. Genomic sequencing: figures 6a and 6b. [Online]. Available: www.illumina.com/documents/products/datasheets/datasheet_genomic_sequence.pdf
- [9] NHGRI. Figure 9: Alternative splicing. [Online]. Available: <http://www.genome.gov/25020001>
- [10] Wikipedia. De novo. [Online]. Available: https://en.wikipedia.org/wiki/De_novo

- [11] J. A. Martin and Z. Wang, "Next-generation transcriptome assembly," *Nature reviews genetics*, 2011.
- [12] M. Šikić and M. Domazet-Lošo, "Bioinformatika," 2013.
- [13] K. Clarke, Y. Yang, R. Marsh, L. Xie, and K. K. Zhang, "Comparative analysis of de novo transcriptome assembly," *Science China Life Sciences*, 2013.
- [14] L. Taylor. PHAST: OLC vs de Bruijn Assemblies. [Online]. Available: gcat.davidson.edu/phast/olcdebruijn.html
- [15] J. H. University and B. Langmead. De Bruijn Graph assembly. [Online]. Available: www.cs.jhu.edu/~langmea/resources/lecture_notes/assembly_dbg.pdf
- [16] Y. Xie, G. Wu, J. Tang, R. Luo, J. Patterson, S. Liu, W. Huang, G. He, S. Gu, S. Li, X. Zhou, T.-W. Lam, Y. Li, X. Xu, G. K.-S. Wong, and J. Wang, "SOAPdenovo-Trans: de novo transcriptome assembly with short RNA-Seq reads," *Bioinformatics*, 2014.
- [17] M. H. Schulz, D. R. Zerbino, M. Vingron, and E. Birney, "Oases: robust de novo RNA-seq assembly across the dynamic range of expression levels," *Bioinformatics*, 2012.
- [18] B. J. Haas, A. Papanicolaou, M. Yassour, M. Grabherr, P. D. Blood, J. Bowden, M. B. Couger, D. Eccles, B. Li, M. Lieber, M. D. MacManes, M. Ott, J. Orvis, N. Pochet, F. Strozzi, N. Weeks, R. Westerman, T. William, C. N. Dewey, R. Henschel, R. D. LeDuc, N. Friedman, and A. Regev, "De novo transcript sequence reconstruction from rna-seq using the Trinity platform for reference generation and analysis," *Nature protocols*, 2013.
- [19] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of discrete algorithms*, 2004.
- [20] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," *First annual ACM-SIAM symposium on discrete algorithms*, 1990.
- [21] G. Nong, S. Zhang, and W. H. Chan, "Two efficient algorithms for linear time suffix array construction," *Computers, IEEE Transactions on*, 2010.
- [22] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park, "Linear-time longest-common-prefix computation in suffix arrays and its applications," *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, 2001.

- [23] J. T. Simpson and R. Durbin, “Efficient de novo assembly of large genomes using compressed data structures,” *Genome Research*, 2011.
- [24] Amos overlap format. [Online]. Available: <http://www.sourceforge.net/p/amos/mailman/message/19965222/>
- [25] J. T. Simpson and R. Durbin, “Efficient construction of an assembly string graph using the FM-index,” *Bioinformatics*, 2010.
- [26] E. W. Myers, “Toward simplifying and accurately formulating fragment assembly,” *Journal of Computational Biology*, 1995.
- [27] D. R. Zerbino and E. Birney, “Velvet: Algorithms for de novo short read assembly using de Bruijn graphs,” *Genome Research*, vol. 18, pp. 821–829, 2008.
- [28] Wikipedia. Edit distance. [Online]. Available: https://en.wikipedia.org/wiki/Edit_distance
- [29] J. H. University and B. Langmead. Overlap Layout Consensus assembly. [Online]. Available: www.cs.jhu.edu/~langmea/resources/lecture_notes/assembly_olc.pdf
- [30] C. Lee, C. Grasso, and M. F. Sharlow, “Multiple sequence alignment using partial order graphs,” *Bioinformatics*, 2002.
- [31] Wikipedia. FASTA format. [Online]. Available: https://en.wikipedia.org/wiki/FASTA_format
- [32] ——. FASTQ format. [Online]. Available: https://en.wikipedia.org/wiki/FASTQ_format
- [33] AMOS. Message Types. [Online]. Available: http://www.amos.sourceforge.net/wiki/index.php/Message_Types
- [34] Sequence Read Archive Submissions Staff. Using the SRA Toolkit to convert .sra files into other formats. [Online]. Available: <http://www.ncbi.nlm.nih.gov/books/NBK158900/>
- [35] W. J. Kent, “BLAT - The BLAST-Like Alignment Tool,” *Genome Research*, 2002.

De novo transcriptome assembly

Abstract

In this thesis, a de novo transcriptome assembler was implemented based on the overlap-layout-consensus paradigm. It was written in the C++ programming language and was named Ra which is short for RNA assembler. Its overlap phase relies on the enhanced suffix arrays and reproduces exact overlaps between input reads. The layout phase uses several methods for graph simplification which includes trimming and bubble popping. Due to the exact overlap phase there is no need for a consensus phase at this moment but there exists one which is based on the partial order alignment algorithm. Conducted tests have shown that Ra needs improvements to compete with other transcriptome assemblers. Source code is available at <https://github.com/rvaser/ra>.

Keywords: RNA, transcriptome assembly, overlap-layout-consensus, suffix array, graphs

De novo sastavljanje transkriptoma

Sažetak

U ovom radu, implementiran je de novo assembler transkriptoma koji je baziran na preklapanje-razmještaj-konsenzus paradigmi. Napisan je u programskom jeziku C++ i imenovan je Ra što je skraćeno od RNA assembler. Faza preklapanja bazirana je na poboljšanom sufiksnom polju i reproducira egzaktna preklapanja između ulaznih očitavanja. Faza razmještaja koristi nekoliko metoda za pojednostavljenje grafova koji su izgrađeni nad očitanjima i njihovim preklapanjima. Kako faza preklapanja pronalazi samo egzaktne parove, trenutno ne postoji potreba za fazom konsenzusa ali ona je svejedno dio implementacije i bazirana je na partial order alignment algoritmu. Provedeni testovi upućuju da su Ra assembleru potrebna dodatna poboljšanja kako bi mogao konkurirati drugim assemblerima transkriptoma. Izvorni kod dostupan je na <https://github.com/rvaser/ra>.

Ključne riječi: RNA, sastavljanje transkriptoma, preklapanje-razmještaj-konsenzus, sufiksno polje, grafovi