UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona

De novo discovery of microRNA from small RNA sequencing data

Francisco D. Morón-Duran

15 de setembre de 2015

Projecte Final de Carrera per a

l'Enginyeria Tècnica en Informàtica de Sistemes

Director
Xavier Messeger
Departament en Ciències de la Computació

Co-director extern
Victor Moreno
Institut Català d'Oncologia

Table of Contents

Introduction

Computing Science is a great tool to help understand biological questions arisen since the Molecular Biology revolution that took place in life sciences in the 1950's. The fact that life processes are encoded into genomes containing programs that can be read, modified and suppressed by the cells is somewhat fascinating.

For a biologist with some computational background is not possible to think of DNA and DNA-binding proteins like polymerases or mismatch repair proteins without making an analogy with Turing machines. Hence, the convergence between Information Theory and Biology is nowadays as natural that stereotypes around computational scientists and biologists are rapidly changing as the need that these people have to understand to each other become more evident.

This work pretends to be another example of how the synergies produced between computing and life sciences can enhance biological discoveries and boost our understanding of life. Not only by predictive algorithms parsing a genomic code that we do not fully understand yet, but with computational methods that serve as a magnifying glass or a compass that guides scientists through their biological questions.

About the structure of this document

This document is mainly divided into five sections. The first one is the present introduction and gives the context in which the work is developed —in *A primer on Molecular Biology*—, explains the biological problem —in *Introducing microRNA*—, overviews the current state of the art —in *Brief history of nucleic acid sequencing*— and gives the rationale for this project —in *Discovery of miRNA by computational approaches*—.

The next section —*Project overview*— covers the project's main objective and planning to materialize its goals from the chronological and economic points of view. It also outlines a basic schematics of the structure of the designed software to give a panoramic view of the different steps to follow in order to reach these goals.

During the *Project Implementation* section the main steps outlined in the previous section are detailed as long as strategies to reach the miRNA candidates are explained.

Finally, in *Results*, the main results of the software output are explained in a set of biological samples thanks to the *Biomarkers and Susceptibility Unit* from the *Catalan Institute of Oncology.*

In an extra *Conclusions and final remarks* section you will find exposed the main difficulties arisen during the development of this project, possible improvements and thoughts about the actual decisions taken at the beginning of the project with the perspective given by the obtained results.

A primer on Molecular Biology

As every computing project, before going into the abstraction of problem solving, there is a need to understand the nature of the questions that can arise in the process. Therefore is necessary to collect some information about the field that involves the question to solve. This section aim is to provide some relevant information and a fast panoramic view of the biology background that support the project problem.

Life is coded in deoxyribonucleic acid

Deoxyribonucleic acid (DNA) is a molecule present in all known living organisms and some viruses known by its properties to consistently encode information necessary for the development of biological entities and their functions. This information is what we call the **genome**. Almost every cell of an organism carries one copy of the genome that makes that organism to be as it is.

Discovered in 1869 by Friedrich Miescher (Dahm R, 2008), it was not until 1928 that Frederick Griffith proved the capacity of DNA to carry genetic information (Chambers DL, 1995). Furthermore, its role in the heredity of traits was confirmed in 1952 by Alfred Hershey and Martha Chase in their well known Hershey-Chase experiment (Hershey AD, et al. 1952). The most important reminded fact, however, is the acceptance in 1953 of the double-helix model proposed by James Watson and Francis Crick confirmed by X-ray diffraction images taken by Rosalind Franklin (Watson JD, et al. 1953).

In this model, DNA is conformed by two strands of nucleotide polymers. Each nucleotide comprises a nucleobase —a nitrogen-containing base of either *guanine* (G), *thymine* (T), *cytosine* (C) or *adenine* (A)— jointly with a monosaccharide called deoxyribose and a phosphate group. Nucleotides are bound to their neighbor on the same strand by a covalent bond between the carbon atom located in the 3' position of their monosaccharide group and the neighbor’s phosphate group in the 5' of the other sugar ring (Fig. 1). This is the main reason why it is said that DNA strands have a direction that goes from 5' (leftmost region of the chain) to 3' (rightmost region).

These two polymer strands are bound to each other in opposite directions by hydrogen bonds between the bases —two for the union of T with A and three for the union of C with

G—, forming a double-helix structure with 34Å turns —1 armstrong equals 10^{-8} centimeters—. This capacity of making different number of hydrogen bonds to each pair of bases gives a special property to DNA molecules: **redundancy**. Redundancy allows DNA **replication** when both chains are separated from each other, as the same information is present in both strands, although encoded in a complementary way. Redundancy also allows to repair errors introduced in DNA, when possible, by mismatch repair mechanisms triggered by the cell.

Another key aspect of DNA molecules is their ability to change. DNA can be altered by random mutations introduced either by replication errors, bad mismatch repairs or by external agents like radiation that alters those chemical compounds conforming the double-helix and making possible an important aspect of living organisms: **evolution**.

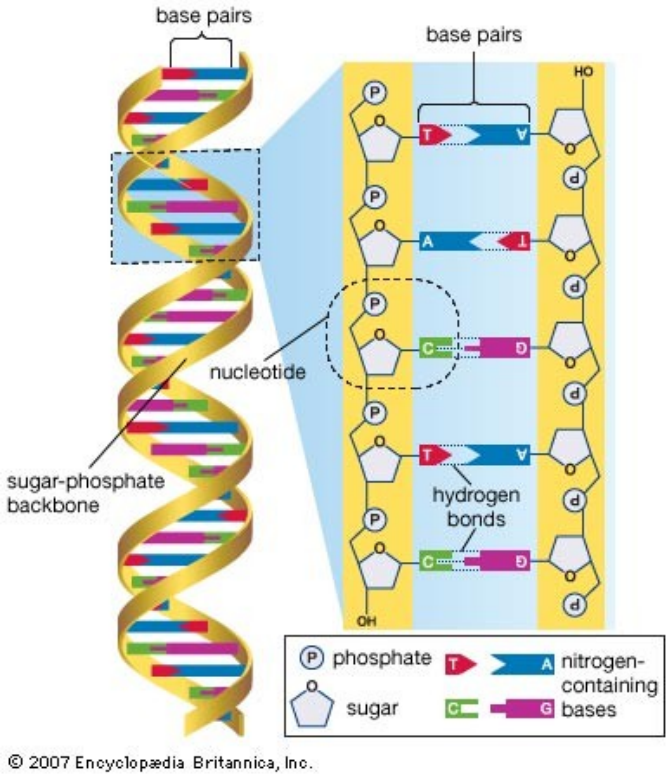

Figure 1 *Scheme of the double-helix model for DNA*

Ribonucleic acid as the working copy of the genome

The genome is a precious possession for a cell. Therefore, it must be protected and saved carefully. It is known that not all genomic locations are compacted the same way. Genes being actively expressed are located in less compact regions to allow the transcription machinery to access their code while non-active genes are kept in more dense regions compacted by some proteins called histones in a structure known as chromatin. In this condensed chromatin, gene code is prevented from unnecessary hazards.

Furthermore, genes encoded in the genome are generic. They must be useful in every cell in all tissues of an organism, but genes are known to have different roles depending on the cell type they are being expressed on. Thus, the functionality of a gene is not obtained directly from its code. Genes are **expressed** into ribonucleic acid molecules (RNA) called messenger RNA (mRNA).

RNA is a more labile version of DNA, and thus error-prone. This is due to the hydroxyl group contained in the 2' position of the ribose that can act as a nucleophile against the rest of the molecule (Fig. 2). Some viruses use RNA instead of DNA as the vehicle for their genome and this confers an advantage to them making their code less stable and more difficult to detect to their hosts immune systems. Typically, RNA molecules are single-stranded and among their nucleobases is found *uracil* (U) instead of T.

For a gene being expressed, an RNA polymerase must access its code in the genome, open the DNA double-helix and start **transcribing** it into RNA. The gene's content is then copied into multiple mRNA molecules and the more mRNA molecules of a gene are produced, the more expressed is said to be that gene.

Then, mRNA molecules can be processed by alternative splicing giving as a result different versions of a gene. Some of these versions are known to be tissue-specific and have different defined functions in the cell. Non-spliced mRNA molecules are known as *pre-*mRNA, while spliced ones are named *mature* mRNA.

However, for most of the genes to be functional it is necessary to **translate** them first into proteins. Proteins are amino acid chains —also called polypeptides—. They are the final functional product of a gene and the actor that plays the most important part: the catalytic

process that lets its intended biological function to take place. Furthermore, protein synthesis from a mature mRNA can have distinct outcomes.

Ribosomes —the molecular complexes required for mRNA translation into protein—, read mRNA transcripts in triplets. Every three consecutive bases correspond to a unique amino acid in the protein. So, for a single mRNA, three possible *reading frames* exist depending on the translation starting point. Each starting point where a ribosome starts its protein synthesis is called an *open reading frame* (ORF), and there can be multiple of them depending on the concrete position they are placed along the mRNA.

But it is important to remember that proteins are not the only catalytic players in the cell. Some RNA molecules also have a defined function by themselves. This is the case of ribosomal RNA (rRNA) which conforms the ribosomes along with some ribosomal proteins and makes the translation of mRNA into amino acid chains. Other RNA motives can also have a function without being translated into protein and, nowadays, a big research area has been found in the so called non-coding RNA (ncRNA), a subgroup of which are microRNA (miRNA) molecules that will be the main focus of this text.

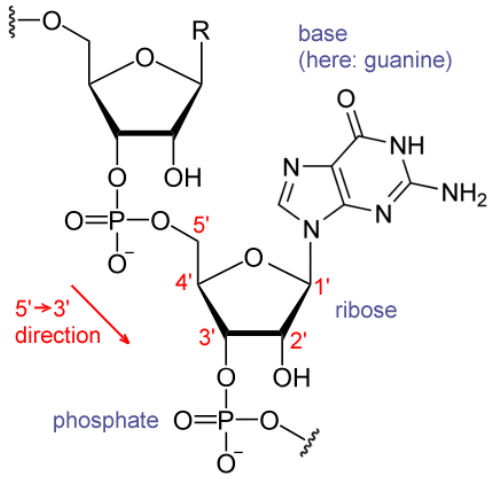

Figure 2 *Structure of a RNA strand*

Messenger RNA and protein levels

The typical mRNA structure for a mature mRNA —the one already processed by alternative splicing— is represented in Fig. 3. This is the RNA molecule that the ribosome will read and will use as a mold to assemble amino acids in the order dictated by the protein-coding gene to synthesize its corresponding polypeptide.

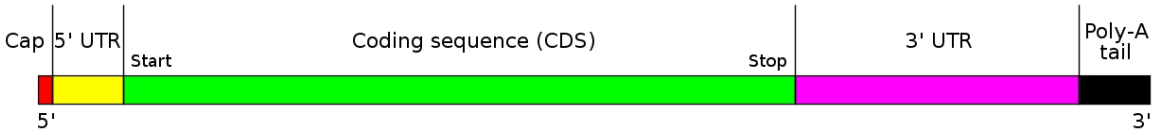

Figure 3 *Structure of a typical human protein coding mRNA including the untranslated regions (UTRs)*

As we can see, not all mRNA sequence is translated into protein. Only the *coding sequence* (CDS) fragment will be converted to amino acids. Flanking the CDS are 5' and 3' *untranslated regions* (UTR) that despite not being translated can participate in the gene translation regulation as we will see later on.

At first, it was thought that protein levels should be somewhat proportional to the expression level of a gene in mRNA. However, proteomic studies soon revealed that this is not really the case. The amount of a protein in a cell depends on a variety of aspects such as the stability of the protein or its synthesis and degradation rates. Recently, translational regulation has emerged as an important key factor governing protein levels. From the pool of available mRNA molecules in a cell not all of them are translated with the same efficiency. Most importantly, translation machinery can be directed to those mRNA that the cell requires in a certain moment, in a process known as *translational control*. Not surprisingly, mRNA expression is currently seen more like a buffer of potential proteins for the cell than like traditional gene expression.

A brief comment on cell structure

In general terms, all eukaryotic cells can be divided in two main compartments: the nucleus and the cytosol. The cytosol is isolated from the environment through a lipid bilayer that allows the maintenance of the conditions that make possible the correct functioning of the cell, what is known as cell *homeostasis*. The nucleus is, at the same

time, inside and isolated from the cytosol by an additional lipid bilayer.

While the genetic code is located and transcribed in the nucleus, it is on the cytosol that messenger RNA is translated into protein. This implies that both, rRNA and mRNA must be exported from the nucleus. Furthermore, microRNA —which will be described in the next section— also must be exported to the cytosol by proteins known as *exportins* to achieve its function.

Introducing microRNA

MicroRNA are small non-coding RNA molecules comprising sizes among 19 to 24 nucleotides capable of modulating gene activity through the blockage of the translation process of a gene's mRNA to its protein product (Esteller M, 2011). A miRNA binds to its target mRNA by complementarity of sequences hampering the progression of the polypeptide elongation by the ribosome and/or promoting cleavage and degradation of targeted transcripts.

This regulatory capabilities make miRNA interesting molecules as targets for molecular therapies which require directed silencing or activation of genes and possible good clinical disease biomarkers.

Molecular structure

A mature miRNA molecule is a single stranded RNA of 19 to 24 *base pairs* (bp) length with a defined *seed* region across nucleotides 2 to 7 or 2 to 8. This seed region has a key role in the specificity of the molecule for its target mRNA as it binds itself to the 3' UTR of the messenger by sequence complementarity. The rest of the sequence of a miRNA may also interact with the messenger RNA by complementarity in an additive way. The stronger the complementarity of a miRNA for its target, the more probability of the target to be cleaved by nucleases recruited by these miRNA-mRNA interactions.

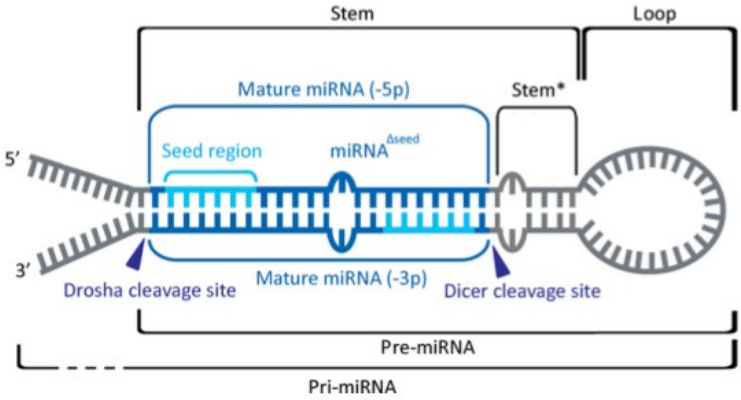

Figure 4 *Different forms of microRNA along its biogenesis process*

Before becoming a single stranded molecule, miRNAs suffer a series of modifications from an original RNA *hairpin* or loop structure (*pri*-miRNA) synthesized by RNA polymerase II —one of the polymerases in charge of transcribing DNA genes into their

RNA form— (Fig. 4).

These loops can be formed either by specifically encoded miRNA genes in the so called **canonical pathway** or originated from intronic regions of a gene —those inside a gene but not coding for protein and spliced out of the mature mRNA during mRNA maturation by the spliceosome— following an alternative or **non-canonical pathway** (Fig. 5).

The existence of non-canonical microRNA makes difficult miRNA prediction by computational algorithms parsing the genome, as they can be found not as a single unique DNA features but obfuscated inside other known ones. This aspect is one of the fundamental motivations that make miRNA discovery from small RNA sequencing an attractive approach to look for them.

Biogenesis

First, in the nucleus of the cell, RNA polymerase II synthesizes an RNA molecule based on the genome sequence. This can be from either a miRNA gene (*pri*-miRNA) or a protein-coding gene in which introns are present short hairpins (loops) that can be processed by Drosha. Drosha is a nuclease protein in charge of excising the loop from the RNA structure formed by the polymerase resulting in a smaller loop called *pre*-miRNA.

It is important to note that from each *pre*-miRNA structure a total of 2 mature miRNA can be produced, one from each extreme of the loop, giving place to the so-called 5p-miRNA or 3p-miRNA.

The *pre*-miRNA can be exported from the nucleus to the cytoplasm by a transporter protein called Exportin 5 (XPO5). In the cytoplasm, a protein complex is recruited including Dicer and an AGO1-4 proteins. AGO recognizes the double stranded part of the *pre*-miRNA while Dicer does the same with the loop. With its nuclease activity, Dicer breaks the loop leaving the stranded part of the molecule with AGO which then decides which mature miRNA sequence keeps and which one liberates to be degraded. Once a single stranded RNA corresponding to a mature miRNA is united to AGO, the recruitment of the RISC complex —that will interact with target mRNA— is produced.

Remarkably, not always both mature miRNA that can be originated from a single pre-miRNA loop have a functional role. In some cases, one of the two molecules is rapidly

degraded and only one becomes a functional miRNA. In other cases, both molecules can be functional and AGO protein decides which one keeps with different probabilities based on the sequence of the structure.

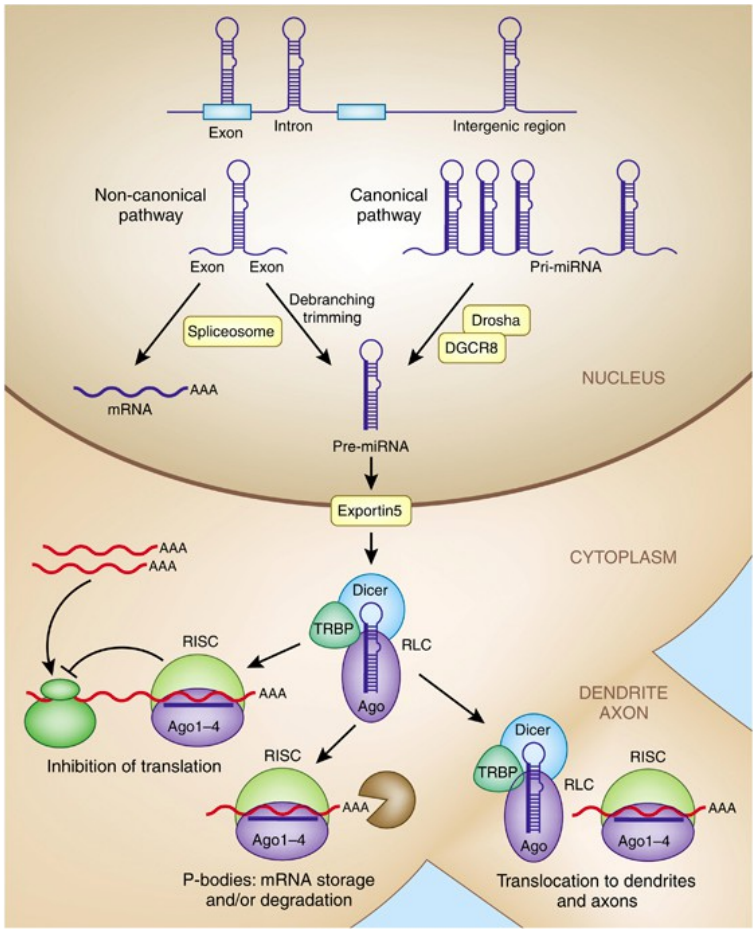

Figure 5 *Canonical and non-canonical biogenesis pathways*

Mechanisms of action

Up to nine different mechanisms of action for microRNA have been described (Fig. 6), but all of them have in common the negative regulation of their target mRNA translation. Therefore, the presence of the miRNA impairs the expression of its target genes.

The classical mechanism is to difficult the progression of the ribosome along the mRNA that is bound to the miRNA by complementarity of sequences, stalling the protein synthesis until the ribosome unbinds the messenger molecule. In cases where sequence complementarity between the miRNA and its target is high, the protein complex recruited by miRNA (*RISC*) can force the cleavage of the messenger RNA, eliminating in fact the

expression of that gene.

It has been described as well the impairment of the pre-initiation complex formation, required for the translation of mRNA from the ribosome before the ribosome itself is recruited. Other more exotic investigated mechanisms include the recruitment of proteases that may degrade the protein synthesized by the ribosome as soon as it comes out of the ribosomal complex, or directly block the binding of the 80S ribosome to the nucleotide chain.

It is important to remark that the miRNA by itself has no function without the proteins and proteinic complexes that it recruits to affect its targets. These protein complexes carrying the microRNA expose the seed region of the sequence to allow for complementarity hybridization to the target and to direct themselves to the regulating mRNA set dictated by different miRNA signatures programmed by the cell in a context-specific manner (dependent on tissue, developmental stage…).

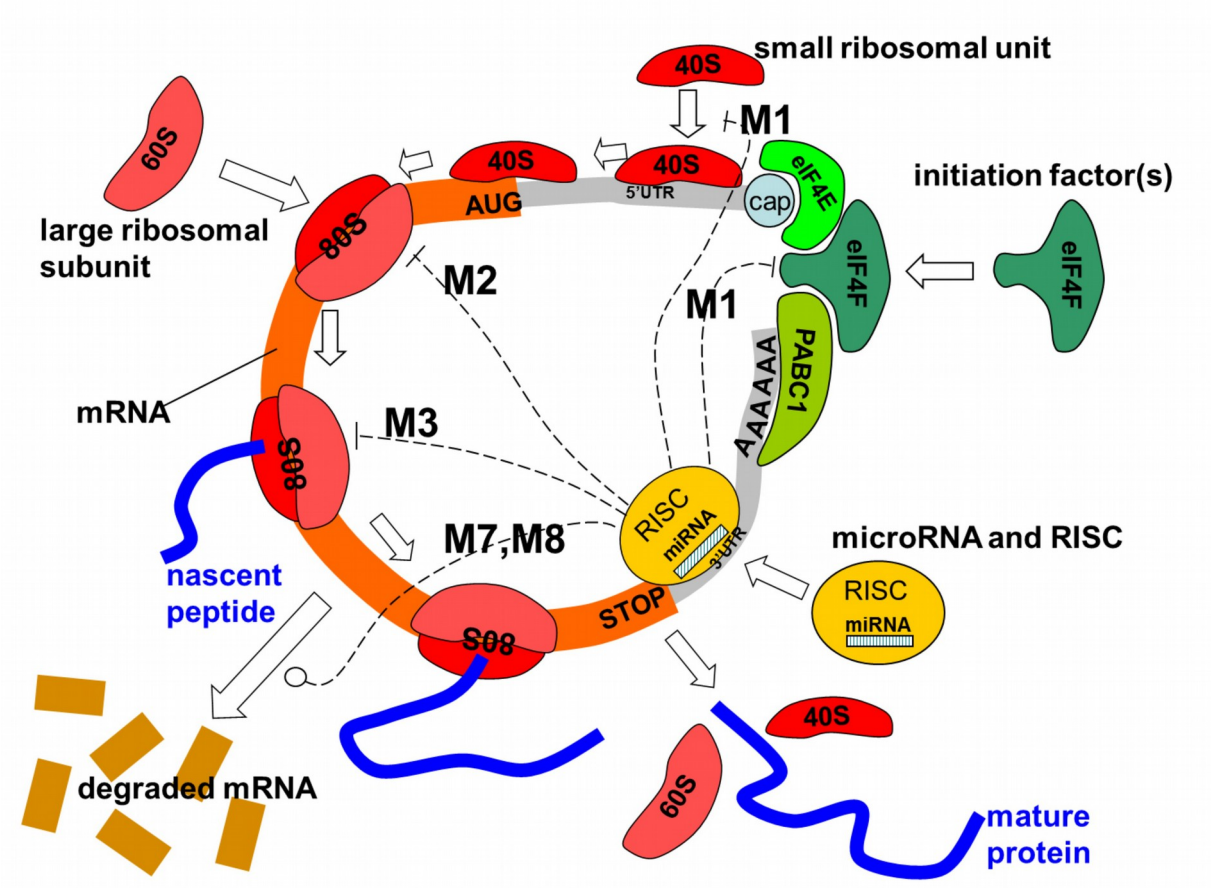

Figure 6 *Schematic of the different described mechanisms of action for microRNA*

Brief history of nucleic acid sequencing

Sanger sequencing

In 1977, Frederick Sanger developed a method for DNA sequencing which nowadays is still considered as the gold standard in the industry (Sanger F, et al. 1977). It is not scalable but convenient for small-scale projects. Its generated reads are relatively long and is used as a confident technique in order to validate findings done by other approaches due to its simplicity and reliability.

Original Sanger sequencing consists in four separated reactions in which a DNA template, DNA primer and a polymerase are mixed together with a pool of dNTP —deoxynucleosidetriphosphates or DNA bases: d**A**TP, d**G**TP, d**T**TP and d**C**TP— and a convenient amount of ddNTP (dideoxynucleotides) for each different reaction. ddNTPs lack 3'-OH group, which inhibits the capability of extend the DNA sequence by the polymerase when it is incorporated to the polymerization reaction. ddNTP amount used is sufficient to still allow for the eventual synthesis of the long original transcript jointly with all different possible shorter lengths sequences in a probabilistic way.

Later, each of the four different reactions are purified in an agarose gel that separates molecules based on their weight —therefore their chain length— obtaining sequentially each position of the different nucleotides along the sequence.

An improvement to Sanger method is the dye-terminator sequencing, consisting in the same four reactions taking place jointly in the same reaction process but with ddNTPs labelled with fluorescent dyes with a color that identify each different terminal nucleotide.

Shotgun sequencing

Although Sanger by itself is not a high throughput technique, shotgun sequencing lets to sequence large genomes by splitting the genomes into smaller fragments and perform a separately sequencing process for each one of them. The obtained fragments —contigs— can be later assembled into a genome by complex computational methods that need high amount of data —overlapping contigs— to succeed in spite of repetitive sequences and sequence errors.

While shotgun sequencing had been used since 1979 to sequence small genomes, it was

popularized in the nineties with the Human Genome Project and similar projects.

A tuned version of this technique is *hierarchical shotgun sequencing*, consisting on the selection of the minimum number of fragments that cover the entire genome to achieve more throughput with less infrastructure, though requiring more complex algorithms.

Next generation sequencing

From approximately 2005, what we know today as Next Generation Sequencing (NGS) has been popularized by its cheap and scalable sequencing capacity. The main characteristic of NGS is its key step of solid-phase amplification. Traditionally, library fragments were amplified by Polymerase Chain Reaction (PCR) (Saiki RK, et al. 1988). Despite PCR has been a great tool in molecular biology, a known problem with that technique is the formation of chimerical sequences by hybridization of unspecific fragments along the iterative processes of hybridization-denaturalization of DNA strands. Obviously, that is a nightmare in the sequencing field.

Solid-phase amplification allows to fix DNA fragments on a surface, therefore they cannot have physical contact with other fragments during the amplification process and each elongation of a DNA strand can occur independently from all the others, with different tempos. This situation is ideal, since it lets us to parallelize the process and make it a high-throughput technique.

The principal drawback of NGS technology is the length of the obtained sequences. With Sanger, the gold standard, you can get large fragments of base pairs at the expense of its cost in terms of time and money. In the NGS world, where we get reads of 250-500bp with 454 (Roche) based on emulsion PCR and pyrosequencing or 30-150bp with Illumina (Solexa) based on bridge PCR and sequencing by synthesis, computational algorithms are necessary to get information of the samples analyzed. And here is where short read aligners have an important role.

Given a reference genome, short read aligners try to map NGS reads to different regions allowing deviations from the reference. Short length of sequences, probability of errors in the read and therefore errors in the mappings are overcome by working with a lot of data in order to minimize error in a probabilistic way. So we must be sure our region of interest is

sequenced enough times to get a consensus between different reads in order to call a base, what is called sample coverage.

There are a lot of algorithms used to align sequences to references, from slow but reliable ones like BLAST[1] based on dynamic programming to fast but less accurate like Burrows-Wheeler. Nowadays, the two most popular aligners are BWA[2] and Bowtie2[3], fast aligners both of them.

1 http://blast.ncbi.nlm.nih.gov/Blast.cgi
2 http://bio-bwa.sourceforge.net/
3 http://bowtie-bio.sourceforge.net/bowtie2/index.shtml

Discovery of miRNA from computational approaches

Highly conserved primary sequences jointly with characteristic secondary structure of miRNA and their precursors are used altogether to find novel microRNA genes. Multiple strategies are available to discover novel miRNA, either from homology found by alignment of known miRNA in other species into our target genomes, computationally parsing the genome looking for recognizable patterns of miRNA-like regions or digging into isolated sequences found in RNA sequencing libraries generated in the wet laboratory.

Discovery by forward genetics: *de novo* and by homology

One approach is to align already known miRNA precursor sequences in other species to the desired organism genome in order to find homologies via local alignment techniques like BLAST. Once a valid alignment has been found and sequence similarity has been assessed, secondary structure of its corresponding RNA must be checked to validate if its conformation maintains the characteristics that make precursor miRNA loops possible, as these loops are the ones that are recognized by proteins that participate in miRNA biogenesis.

These kind of analysis by homology can be related with evolutionary studies, as the phylogeny of miRNA can be used to trace back sequence modifications between different evolution-related organisms.

De novo discovery by forward genetics requires to produce *knock out* animal models or cell culture settings, that is, an animal or a cell line that lacks some region of the genome, identify the phenotypic consequences of the deletion and ensure that they are reverted upon the addition of the previously deleted gene. This experiments are based on traditional gene functionality studies, take time and not always show results.

It is difficult to find low expressed or slightly functional miRNA/gene in rare cell types —those that can not be cultured easily— or at specific developmental stages. Furthermore, some genomic regions may be difficult to clone either by their sequence composition or by post-transcriptional modifications that can not be added in a transfection vector, making more difficult to validate phenotypic changes.

Computational prediction by machine learning

With the existing set of known miRNA, algorithms based on machine learning can be trained to detect specific features that make a genomic sequence a good candidate to eventually become a miRNA. Fold-back, conservation information of sequence and secondary structure can be evaluated to classify RNA structures into miRNA-like sequences or not. Genome sequences or reads proceeding from high-throughput sequencing experiments can be used for this kind of detection.

A posterior comparison with already annotated sequences must be done to keep under control false positives and take care of possible duplication entries into annotation databases. Small RNA libraries for the organism, if available, can be used to asses if the candidates are being found at the RNA-level in available samples as a proof of their capability of being an unidentified miRNA.

Identification from small RNA sequencing based on a reference genome

Another approach to the identification of novel miRNA is based on the generation of libraries of small RNA sequencing. These libraries consists on the isolation of small fragments of RNA found in a sample prior to their sequencing by Next Generation Sequencing methods (NGS). Once the sequencing reads are obtained, they are aligned against the reference genome for the organism they proceed and regions covered by those reads are identified into that genome.

MicroRNA-like regions are easily detected when both, 5p and 3p miRNA are found, as they are relatively close to each other in the genome just separated to allow the formation of the more or less complex RNA loop, giving as a result two closely located peaks of coverage of approximately 22 nucleotides each one.

Exploration of small RNA libraries with Next Generation Sequencing techniques can be desirable to identify low expressed miRNA and difficult to validate candidate regions. NGS techniques let us view if these sequences exist or not in our library at a glance. This point is the one that this project will put the focus in.

Project Overview

As we have previously seen, sequence of miRNA molecules is mostly conserved in its central region —including the seed— with the exception of some polymorphism, while 5' and 3' ends of the nucleic acid chain are more variable, which results in the existence of a set of so called isomiRs for each miRNA (Neilsen CT, et al. 2012). miRBase, The authoritative miRNA database contains all the known sequences found in the literature or predicted by computational methods (Griffiths-Jones S, et al. 2006).

On another note, small RNA-seq is a high-throughput technique that allows the sequencing of short RNA molecules present in a prepared sample involving their amplification to minimize the risk of obtaining erroneous reads by simultaneously sequencing the same molecule multiple times.

Up to date, existent alignment tools take subsequences of sequences to align as the *core region*, and usually this part is selected from the most reliable part of a read: its beginning, where base qualities are the best. Mapping this *core region* to genomic coordinates —the so called *seed* step— is straightforward, using either hashing techniques or Burrows-Wheeler transform algorithms (Burrows M, et al. 1994). Then, an *extension step* takes charge of matching the remainder of the read to the selected genomic location in order to validate or to discard the possible match. This *extension step* is computationally expensive, specially in large genomes in which repetitive sequences are frequent and lead to multiple wrong genomic location, like the human genome.

With small RNA-seq, often reads are not alligned directly to the entire genome, but to a curated database of already known miRNA, like mirBase. In the case of miRNA, the extremes of the sequence might not be conserved, possibly interfering with initial steps of traditional alignment software, and the fact that miRNA are represented by very small sequences can difficult the alignment.

Recently, a multi-seed strategy has been published for mapping reads to a reference genome using the *seed-and-vote* paradigm (Liao Y, et al. 2013). In this setting, genomic positions are retrieved by simultaneous multiple local alignment of different massive substrings of a read —*subreads*— and without mismatches to the reference. These subreads are considered to vote

for genomic locations, and that genomic location that has more votes is the accepted one, with the final alignment directly computed counting subread mappings eliminating in great measure the overload of the extension step.

In this project, we will use the *seed-and-vote* strategy to find relevant miRNA groups without the need of a reference database.

Objective

The main aim of this project is to obtain miRNA or novel miRNA-like sequences from small RNA sequencing data without the need of a reference database. That goal is going to be achieved aligning reads obtained in the sequencing process taking into account the particular characteristics of the miRNA sequences and their possible isomiRs. This involves designing algorithms capable of grouping sequences by similarity in their central region while allowing more flexibility in both of their extremes. To this end, efficiency is a very sensitive aspect of the process, since the number of input sequences can be as high as millions of reads and multiple steps of alignment and similar sequences grouping are required.

Planning

This project, as a final project in the Diploma in Computer Systems, consists of 22.5 academic credits estimated in 20 work load hours per credit. Therefore, 450 theoretical total dedication hours are available.

Chronological plan

The most time consuming part of the project is intended to be coding and documentation —writing this memory—. A biology overview is required to open the field and update knowledge about the nature of the problem to solve.

In summary, the list below is an estimation of the intended dedicated hours for each of the parts required for the correct development of this project.

- Biology overview……. 25 h
- Software design…… 75 h
- Software development…… 200 h

- Software testing…… 50 h
- Documentation……100 h

TOTAL **450 h**

Economic budget

With 450 h in mind, an economic budget can be done to estimate this project's cost in case it should be externalized by a company.

The biology overview comes for free, as educational cost is normally part of individual formation and not assumed by the customer. For the software design, development and testing, an advanced bioinformatician or computational biologist is needed with an estimated cost of 60 €/h. The documentation of the project can be elaborated by a documentarist or administrative worker with knowledge in computing with an estimated cost of 30 €/h.

- Biology overview (0 €/h x 25h)……. 0 €
- Software design (60 €/h x 75h)……4,500 €
- Software development (60 €/h x 200h)……. 12,000 €
- Software testing (60 €/h x 50h)……3,000 €
- Documentation (30 €/h x 100h)……3,000 €

TOTAL **22,500 €**

All material support, including software, hardware and physical installations are provided by the *Biomarkers and Susceptibility Unit* from the *Catalan Institute of Oncology.*

Proposed pipeline outline

For the sake of simplicity, preprocessing steps are not shown in the diagram below, as they require the use of existent software within a Bash script proportioned in Annex A.

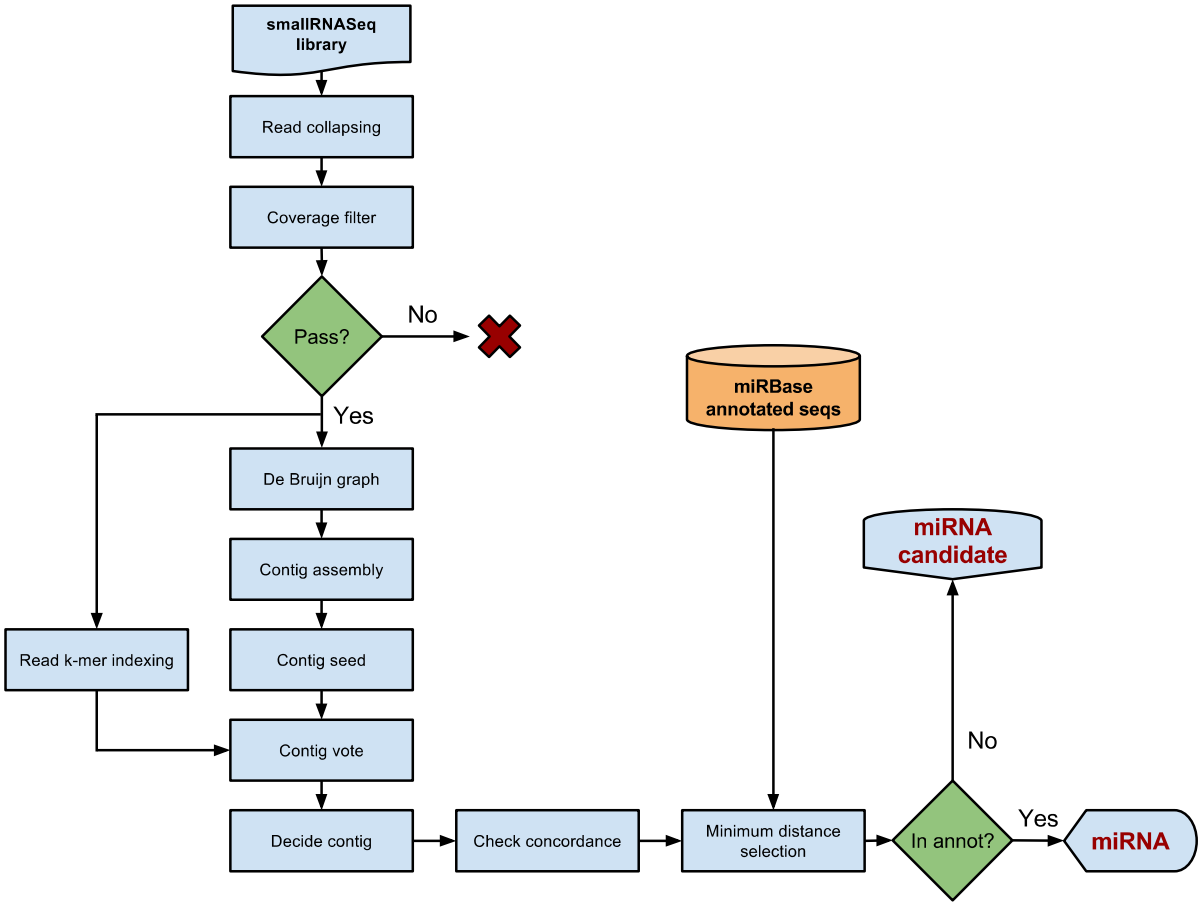

Figure 7 *Pipeline to identify miRNA from small RNA-seq datasets*

Project Implementation

The first approach considered for this project was to use pairwise alignments among input sequences to find miRNA-like sequence candidates. Due to the cost in computational time for this approach, it was rapidly discarded in favor of the *seed-and-vote* strategy, which allows for directly finding plausible candidates in constant time, discarding more rapidly those comparisons between sequences that are not related to each other.

One step further in decreasing time cost is the formation of contigs from a de Bruijn graph built with the input reads against which sequences should be compared. This let sequences to be compared to a small set of unambiguous nucleotide strings, instead of compare all the sequences with one another. De Bruijn graphs condensate information of the assembly of the reads in a memory-efficient structure, and allows for rapid consensus groups formation taking advantage of the most stable and unambiguous parts of the input data.

On the following paragraphs, the main steps taken for the assembly of miRNA-like candidate groups are described, at the same time that relevant formalities of these procedures are outlined.

Preprocessing of input FASTQ files

The first step is to perform a quality control check to input reads, discarding those with low qualities or flagged as bad reads by the sequencer. All sequencing platforms provide such tools to ensure the quality of their given output. This is the case of CASAVA (Illumina)[4], although independent software also exist like FASTX-Toolkit[5] or FastQC[6].

With remaining quality reads, adaptor trimming must be performed to remove from the sequences the linking adaptors introduced during the library preparation required for the sequencing amplification process. Sometimes, also barcodes are included in these adaptors to identify sequences providing from multiple different samples sequenced at once, what we call multiplexed sequencing —this gives cheaper sequencer runs at the expense of lower per-sample coverage—.

4 https://support.illumina.com/sequencing/sequencing_software/casava.html
5 http://hannonlab.cshl.edu/fastx_toolkit/
6 http://www.bioinformatics.babraham.ac.uk/projects/fastqc/

A third optional preprocessing step is to filter reads based on read length. In this case, as we are seeking reads comprised between 16 and 30 nucleotide bases, we discard all reads above 30 and below 16 bases length. This should help to avoid noise from partially degraded RNA eventually captured during library preparation.

Read collapsing and representativity filtering

In order to save computing time, the first step of our alignment strategy consists on collapsing all identical reads. For clarity, in this project, a read is defined as a piece of input and a sequence as a unique character string that represents a read. Therefore, multiple reads can be represented by a unique sequence, and one sequence can be present more than once in the input while being considered one unique sequence.

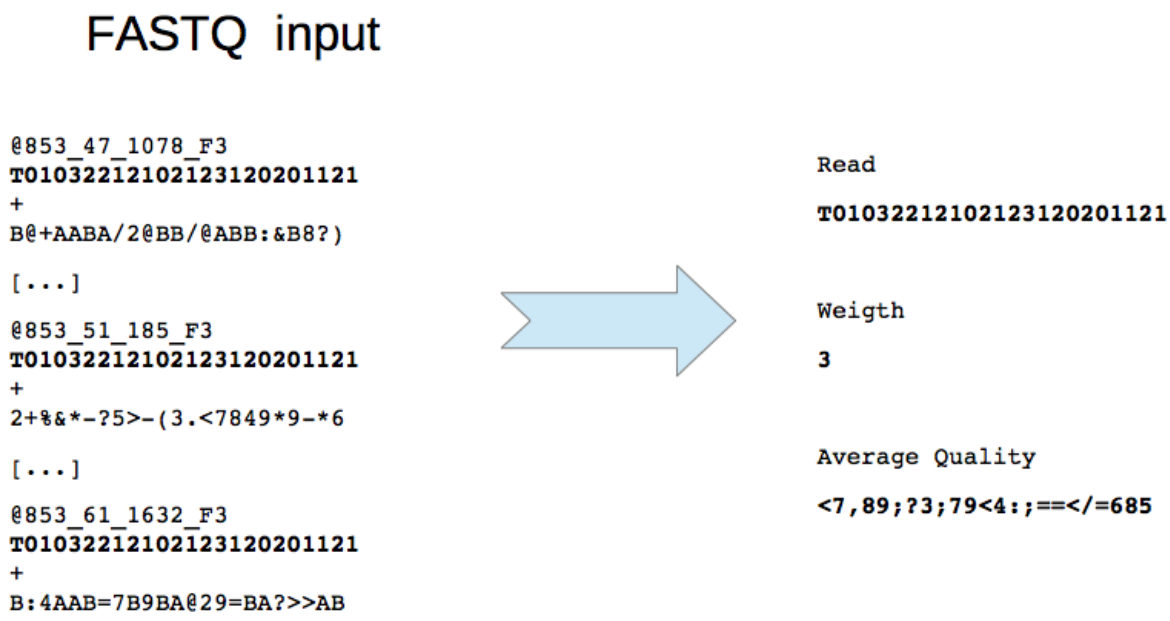

Figure 8 *Graphical representation of read collapsing motivation*

Expression profiles of microRNA are usually of interest. For this reason, keeping record of the number of times a sequence appears in the input is necessary. Some reads, and sometimes a lot of them, may appear a really low number of times (i.e. one). Taking into account that this data comes from an amplification process, where the sequencer amplifies the sample library before really perform any sequencing, this is a strange case pointing to some possible error either from the sequencer or the PCR step (Kebschull JM, et al. 2015). In consequence, sequences appearing less than a threshold can be filtered out before going any further. By

default, in the software developed in this project this representativity value is set to 5 times. Any sequence with less or equal than 5 reads is not going to be processed.

Alignment strategy

De Bruijn graph construction and contig assembly

With collapsed reads into sequences and after filtering by representativity threshold, sequences are then used to perform a de Bruijn graph with their *k*-mers. A default *k* value of 11 is taken. This value is small enough to capture variability in sequences and big enough to limit the extent of spurious constructed contigs, due to the fact that our sequences are between 16 and 30 nucleotides length.

Once a de Bruijn graph is built, those nodes that come from less than a threshold sequences are removed from the graph. The default threshold value is 2. This limits further erroneous paths giving as a result chimeric contigs.

The graph is then visited from each of its nodes until all the unambiguous contigs are found. Unambiguous contigs are defined by the longest paths without branches in the graph. These contigs will be the ones around which all the other reads will be grouped looking for miRNA-like sequences.

Seed step: Indexing contigs and sequence k-mers

Contigs obtained after navigating de Bruijn graph are then indexed into a dictionary by its contained *k*-mers in the *seed step*. For this step, a smaller *k* value is set in order to allow the finding of more variability for sequences inside a contig. By default *k* is 6 in the software developed for this project.

The contig index will later allow to look for contig candidates for given sequences in constant time, avoiding pairwise alignments. The contig is stored jointly with the sequence offset index where the corresponding *k*-mer is found. If a *k*-mer is found more than once, entries with the pair (contig, offset) are stored as many times it is found.

Similarly, input sequences are indexed into another dictionary for the sole purpose of computing *k*-mer occurrence frequencies, that will be needed while computing sequence distances later on, as we will see. *k*-values for the sequence distance computation can be

different from other processes described (k=4 by default in our software).

Voting step: Sequences voting contig candidates

Once the contig index has been set up, it is time to allow the sequences to vote for their preferred contig. This is achieved by having each *k*-mer (or subsequence of length *k*) from a sequence, looking into the index for contigs containing the *k*-mer and generating a vote for each of the positions in the contig where the *k*-mer beginning is found.

A vote is a tuple of integers (p_c, p_s), where p_c is the position of the *k*-mer in the contig and p_s its corresponding position in the sequence. Votes do have order, and that order only makes sense if they belong to the same {contig, sequence} relation. Considering two votes *v* and *w*, *v* is higher than *w* if and only if $p_{c,v} > p_{c,w}$ and $p_{s,v} > p_{s,w}$. It is important to remember that contigs are unambiguous paths in a de Bruijn graph, therefore votes should have subsequent increasing coordinates.

By the time all subsequences from a sequence have generated their votes, a further deciding step is done by assigning it to the contig it has more votes for.

Breaking ties: distance between sequences

Sequences that do not have a unique contig with a global maximum number of votes, therefore, can not be assigned to multiple contigs. In this case, it is necessary to design a way to break the tie between the multiple contigs with maximum votes. We could perform pair-wise alignments to this end, but taking advantage of sequence and contig k-mers we can compute a sort of distance between nucleotide strings in a way that more similar ones remain nearer than those which are not.

In the case of sequences that vote the same maximum amount of votes for different contigs, the contig nearer to the sequence will be assigned for them.

A formal definition of distance

Given an integer *k*, we define a sequence *s* as a point *p(s)* with one coordinate $p(s)_i$ for each possible sequence of length *k*. As the alphabet we are working with consists of four letters —A, T, C and G—, a point *p(s)* has 4^k coordinates. Each coordinate consists of the number of times that such *k*-sequence is present in *s*.

In practical terms, what we want is to count the number of times a *k*-mer —that is a

subsequence of length *k* from *s*— occurs in the sequence *s*. So, for all the *k*-sequences that are not *k*-mers of *s*, their coordinate value will be zero.

Given the previous definition we can define a distance like

$$d^2(s_{1,}s_2)=\sum_{i=1}^{4^k}(p(s_1)_i-p(s_2)_i)^2 \quad (1)$$

The formula in (1) is called the *Squared Euclidian distance* between $p(s_1)$ and $p(s_2)$. It should be noted that for comparison purposes, this distance is sufficient without taking the square root, but it is not a *true statistical distance*, that means it not satisfies the triangle inequality (Wu TJ, et al. 1997). Triangle inequality states that *the sum of the lengths of any two sides must be greater than or equal to the length of the remaining side* (Mohammed AK, et al. 2001). For a true statistical distance, when needed, the square root of the value should be taken.

Given the nature of our problem, where sequences may be short and variable in length, this way of counting distances may be misleading. Therefore, we modify slightly the definition of coordinate $p(s)_i$ to be a frequency of the *k*-mer defined by a *k*-sequence in *s* as a way to remove from the model any sequence length related bias.

$$p(s)_i=\frac{\text{occurrences of } k\text{-mer } i \text{ in } s}{\text{total } k\text{-mers in } s} \quad (2)$$

It is known that *k*-mer occurrence is variable between genomes from different species, and that generally *k*-mers do not have the same chance to appear in a biological sequence. For this reason, another definition of distance that takes into account the standard deviation of *k*-mer occurrence is more suitable in our case

$$D^2(s_{1,}s_2)=\sum_{i=1}^{4^k}\left(\frac{p(s_1)_i-p(s_2)_i}{\sigma_i}\right)^2 \quad (3)$$

with a diagonal covariance matrix

$$\sigma_i=\frac{1}{N-1}\sum_{n=1}^{N}\left(p(s_n)_i-\mu_i\right)^2 \quad (4)$$

taking N as the number of total sequences in our dataset, and

$$\mu_i = \frac{1}{N} \sum_{n=1}^{N} p(s_n)_i \quad (5)$$

The formula in (3) is known as the *Mahalanobis distance* (Mahalanobis PC, 1936) that with a diagonal covariance matrix is known as *normalized Euclidean distance* (Wu TJ, 1997). This is the distance definition that the software developed during this project adopts.

In our setting, sequence distance is only computed for the overlap between sequences when the offset between them is known.

As a final remark in contig assignments, for the extreme case where distances between two or more contigs to a sequence are identical, a final decision is taken by lexicographical order to assign the sequence to a contig in order to ensure a deterministic behaviour of the algorithm to reach reproducible outputs.

Errors detection

Once each sequence is assigned to a contig, remaining contigs with zero associated sequences are removed from memory. Also, contigs with fugitive votes can be inspected to gain knowledge about which sequences received those votes to see if those contigs could be merged. Integrity of contigs is then assessed to remove bad sequences: for each sequence in a contig, the generated votes should be in increasing order.

As microRNA have sequences relatively variant —recall *isomiRs*—, some variability inside a contig must be allowed. To limit the extent of variability in a contig, an outlier detection method is implemented using the definition of sequence distance announced above.

For each sequence in a contig, a point with 4^k coordinates —each representing a *k*-mer and filled with the frequency of *k*-mer occurrence in the sequence— is defined (k = 4 by default). Then, the distance from the point representing the contig sequence to each of the sequence points is computed. Calculating interquartile range (IQR) for the distances, outlier candidates are found above Q3 + 1.5(IQR). These candidates are then isolated from the main set, the centroid of the remaining points is found and the standard deviation of distances from the remaining points to the centroid is computed. If distance from each of the candidate points to the centroid is greater than the standard deviation, the candidate is definitely removed from the contig.

Reaching a consensus sequence

Within curated contigs, sequences voting for them can start before the contig sequence or end after it. So we need a representative consensus sequence to represent that group probably bigger than the contig sequence itself.

To this end, an offset for each sequence based on their votes for the contig is computed to align paired nucleotides. then, for each overlapping base across the multiple sequences inside the group, the frequency of occurrence for each found base is calculated, and the most representative base is keep as the consensus one for that position.

Identifying already annotated miRNA

As a final and optional step, group consensus sequences are aligned to a database of annotated miRNA downloaded from miRBase. This alignment can be used to identify contigs to already described miRNA, unveil different contigs being the same miRNA (because of isomiRs or unavailability of unambiguous contigs in the de Bruijn graph) and to detect possible sequences present in te dataset that could be miRNA not yet present in the curated database.

For this step, BLAST+ software from the U.S. NCBI is used as described in the script on Annex C.

Results

Using the designed software, five small RNA-seq libraries of normal colonic mucosa from patients with colorectal cancer from the COLONOMICS project of the *Biomarkers and Susceptibility Unit* at *Catalan Institute of Oncology* were processed.

As summarized on table 1, library depths were variable in the range of 1 to 3.5 million reads approximately. Before sequence collapse, sequence data was reduced to an approximate 10% of original input. Accounting sequences with more than 5 occurrences, the numbers were further decreased to around a 5% of unique sequences, showing that the vast majority of sequences had very few reads.

These low-count sequences could be present as contaminants or degradation products coming from existent total RNA in the samples that were captured while filtering for sequence length, though some errors introduced by the PCR amplification can also be responsible for these reads.

Table 1 *Descriptive of samples processing output*

Sample	Total reads	Unique sequences	Sequences with > 5 occurrences	*N*odes in de Bruijn graph	Initial contigs	Final groups
A2004_N	3,344,128	331,800	15,622	13,529	3,594	1,787
A2027_N	2,804,984	300,946	15,307	13,163	3,211	1,690
A2050_N	1,254,252	130,091	6,924	6,197	1,270	746
A2073_N	1,379,620	133,766	7,797	7,030	1,678	852
A2096_N	1,431,128	143,936	8,106	7,406	1,557	856

The number of reads per final contig show an exponential distribution, as seen in figure 9. Few contigs get most of reads, while a long tail of contigs only get a few reads. This measure gives value for contig abundance and this is consistent with existent gene expression analysis, where log-transformation of expression values is generally accepted as a normalization step before working with generated data in gene expression experiments.

Exponential distribution is also seen in the number of sequences in each contig (Fig. 10). We should recall that a sequence can be representative of one or more reads, so its number is not necessarily related with contig abundance, but it gives a perspective on diversity of molecules.

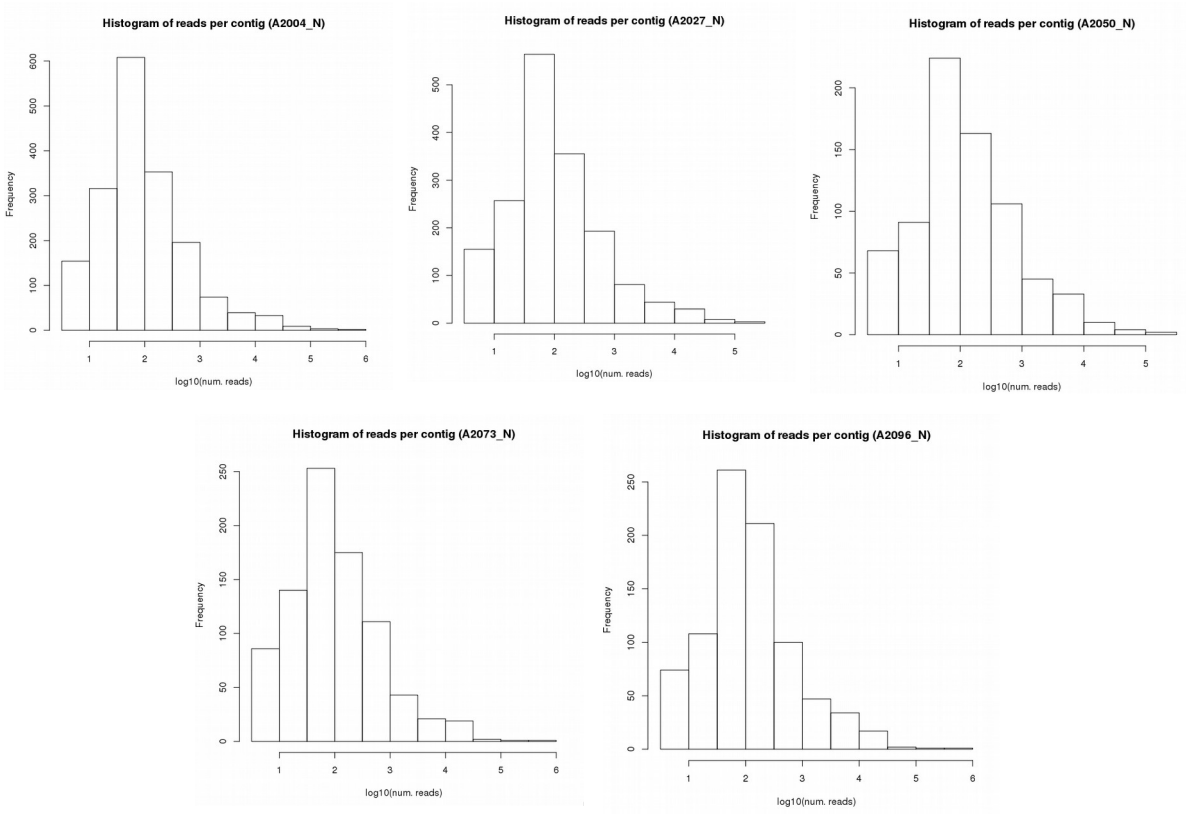

Figure 9 *Histograms of reads per contig for each analyzed sample.*

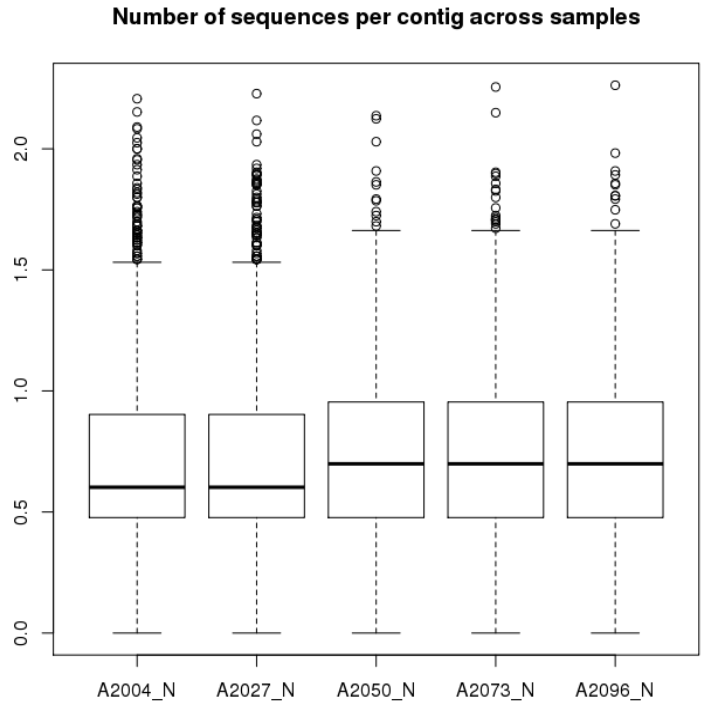

Figure 10 *Box plot of sequences per contig*

Table 2 describes the number of matches before aligning consensus sequences from final contigs against miRBase with BLAST algorithm with an E-value threshold of 0.01. E-value is a measurement of the expectation that the alignment could be produced by chance, so the lower E-value, the lower are the chances for being a spurious alignment.

Table 2 *miRBase matches of candidate contigs*

Sample	Final groups	Matches with E-value < 0.01
A2004_N	1,787	739
A2027_N	1,690	654
A2050_N	746	372
A2073_N	852	417
A2096_N	856	398

On figure 11, an example of BLAST output is given for the most represented contig in sample *A2004_N*. Multiple alignments show significance for this contig, but the lowest E-value match is selected.

```
Query= contig_1 votes: 4844101 sequences: 142 counts: 543015

Length=25
                                                                      Score     E
                                                                      Score     E
Sequences producing significant alignments:                          (Bits)  Value

  hsa-miR-192-5p MIMAT0000222 Homo sapiens miR-192-5p                 38.1    1e-06
  hsa-miR-215-5p MIMAT0000272 Homo sapiens miR-215-5p                 34.4    2e-05
  hsa-miR-192-3p MIMAT0004543 Homo sapiens miR-192-3p                 23.3    0.038
  hsa-miR-4256 MIMAT0016877 Homo sapiens miR-4256                     19.6    0.49
  hsa-miR-512-3p MIMAT0002823 Homo sapiens miR-512-3p                 17.7    1.7
  hsa-miR-196b-3p MIMAT0009201 Homo sapiens miR-196b-3p               17.7    1.7
  hsa-miR-4526 MIMAT0019065 Homo sapiens miR-4526                     15.9    6.3
  hsa-miR-424-5p MIMAT0001341 Homo sapiens miR-424-5p                 15.9    6.3
  hsa-miR-154-3p MIMAT0000453 Homo sapiens miR-154-3p                 15.9    6.3


> hsa-miR-192-5p MIMAT0000222 Homo sapiens miR-192-5p
Length=21

 Score = 38.1 bits (20),  Expect = 1e-06
 Identities = 20/20 (100%), Gaps = 0/20 (0%)
 Strand=Plus/Plus

Query  2   CTGACCTATGAATTGACAGC  21
           ||||||||||||||||||||
Sbjct  1   CTGACCTATGAATTGACAGC  20
```

Figure 11 *BLAST example with 100% coincidence between contig and annotated miRNA*

Conclusions and final remarks

We can conclude that the software identifies microRNA. Among contigs not matching the miRBase, we can't ensure they are miRNA before a biological validation in the lab. This sequences can be fruit of partial degradation of longer RNA sequences present in the cell or true small non-coding RNA. Contigs with high vote and read numbers could help to identify relevant abundant sequences that can be later tested in the wet lab.

This software design, in order to find isomiRs, tries to separate similar sequence groups that differ from the contig initial sequence in their overlapping regions. This, in fact, can result in multiple contigs being the same microRNA. A further step trying to merge those contigs into bigger ones could be implemented as an improvement to the current program.

During the design phase of this project, Python language was selected for its efficient use of dictionaries indexes and its object model that reduces RAM usage by using pointers to objects instead of mainly copying data. However, this behavior makes Python a *thread-unsafe* language in which some precautions have to be made. In order to ensure sequential access to its objects, Python implements a mechanism called *Global Interpreter Lock* (GIL). This GIL avoids typical drawbacks in concurrent programs regarding their sequential consistence that can result in non-deterministic behavior.

A parallel version of the software was designed using threads and processes that was soon discarded because of the low performance of Python in multicore computers due to the GIL. Though the GIL achieves its goal, some designs like the one described in this document can't benefit of a true parallel computation as the GIL rapidly becomes a bottleneck when trying to fill dictionaries from different processes or threads. Knowing this in advance, another computing language would have been used, definitely.

References and Bibliography

Burrows, M, Wheeler, DJ (1994). “A block sorting lossless data compression algorithm”, Technical Report 124, Digital Equipment Corporation

Chambers, DL (1995). *DNA: the double helix: perspective and prospective at forty years*. New York, N.Y: New York Academy of Sciences, p. 49.

Dahm, R (2008). “Discovering DNA: Friedrich Miescher and the early years of nucleic acid research”. *Human Genetics* 122 (6): 565–81.

Esteller, M (2011). “Non-coding RNAs in human disease”. *Nature Reviews*. Genetics, 12(12), 861–74.

Farinelli, C (2008). *Complexity measures and similarity metrics: properties and applications to biological signals*. PhD thesis (Alma Mater Studiorum - Università di Bologna, Bologna, Italy) pp. 119–114.

Gentleman, JF, Mullin, RC (1989). The distribution of the frequency of occurrence of nucleotide subsequences, based on their overlap capability. *Biometrics*, 45(1), 35–52.

Griffiths-Jones, S, Grocock, RJ, van Dongen, S, Bateman, A, Enright, AJ (2006). “miRBase: microRNA sequences, targets and gene nomenclature”. *Nucleic Acids Research*, 34(suppl 1), D140–D144.

Hershey, AD, Chase, M (1952). “Independent functions of viral protein and nucleic acid in growth of bacteriophage”. *J Gen Physiol*. 36:39-56.

Kebschull, JM, Zador, AM (2015). “Sources of PCR-induced distortions in high-throughput sequencing data sets”. *Nucleic Acids Research*, gkv717.

Liao, Y, Smyth, GK, Shi, W (2013). “The Subread aligner: Fast, accurate and scalable read mapping by seed-and-vote”. *Nucleic Acids Research*, 41(10).

Mahalanobis, PC (1936). "On the generalised distance in statistics". *Proceedings of the National Institute of Sciences of India*, 2 (1): 49–55.

Mohammed, AK, William, AK (2001). "§1.4 The triangle inequality in ℝn". *An introduction to metric spaces and fixed point theory*. Wiley-IEEE

Neilsen, CT, Goodall, GJ, Bracken, CP (2012). “IsomiRs—the overlooked repertoire in the dynamic microRNAome”. *Trends in Genetics*: *TIG*, 28(11), 544–9.

Saiki RK, Gelfand DH, Stoffel S, Scharf SJ, Higuchi R, Horn GT, Mullis KB et al. (1988)

"Primer-directed enzymatic amplification of DNA with a thermostable DNA polymerase". *Science* 239: 487–491

Sanger F, Nicklen S, Coulson AR (December 1977). "DNA sequencing with chain-terminating inhibitors". *Proc. Natl. Acad. Sci*. U.S.A. 74 (12): 5463–7.

Mäkinen, V, Belazzougui, D, Cunial, F, Tomescu AI (2015). *Genome-Scale Algorithm Design. Biological Sequence Analysis in the Era of High-Throughput Sequencing.* Cambridge University Press.

Watson, JD, Crick, F (1953). "A structure for deoxyribose nucleic acid". *Nature* 171 (4356): 737–738.

Wu, TJ, Burke, JP, & Davison, DB (1997). A measure of DNA sequence dissimilarity based on Mahalanobis distance between frequencies of words. *Biometrics*, 53(4), 1431–1439.

Zerbino, DR (2009). *Genome assembly and comparison using de Bruijn graphs*. PhD thesis (Univ of Cambridge, Cambridge, UK).

Image credits

Figure 1. DNA. *Encyclopedia Britannica.*

http://www.britannica.com/EBchecked/topic/167063/DNA

Figure 2. RNA. *English Wikipedia.*

http://en.wikipedia.org/wiki/RNA

Figure 3. Messenger RNA. *English Wikipedia.*

http://en.wikipedia.org/wiki/Messenger_RNA

Figure 4. Jevsinek Skok, D., Godnic, I., Zorc, M., Horvat, S., Dovc, P., Kovac, M. and Kunej, T. (2013), Genome-wide in silico screening for microRNA genetic variability in livestock species. *Animal Genetics*, 44: 669–677.

http://onlinelibrary.wiley.com/doi/10.1111/age.12072/abstract

Figure 5. O'Carroll D., Schaefer, A. (2012), General Principals of miRNA Biogenesis and Regulation in the Brain. *Neuropsychopharmacology Reviews*, 38: 39–54.

http://www.nature.com/npp/journal/v38/n1/full/npp201287a.html

Figures 6. MicroRNA. *English Wikipedia.*

https://en.wikipedia.org/wiki/MicroRNA

Figures 7, 8, 9 and 10 were created for this project.

Annex A: Bash script for FASTQ preprocessing

```
#!/bin/bash

## Usage: ./$0 file.csfasta file.qual

## SOLiD Preprocess Filter
## Discards bad quality reads
solid_prepfilt2.pl -f $1 -g $2 \
                   -o ${1%.csfasta}_nomis \
                   -v -x no -p no -y no -n yes

## Cutadapt v1.0 downloaded from
## https://cutadapt.readthedocs.org/en/stable/
## Removes adaptor from reads
## '330201030313112312' is the adaptor for our libraries, this value is variable
cutadapt -c -e 0.1 -z -m 16 -M 30 -a 330201030313112312 \
         -o ${1%a}q {1%.csfasta}_nomis_T_F3.csfasta {1%.csfasta}_nomis_T_F3.qual
```

Annex B: Python code for the project

Main source: *main.py*

```
################################################################################
### miRNAligner - microRNA alginment utility for smallRNAseq data |coding: utf8
################################################################################
### Main source
################################################################################
### Author: Francisco D. Morón-Duran <fdmoron@iconcologia.net>
### Description: Parse input, show help and init workflow.
################################################################################


import sys
import os.path
from seqs import Collapse


def help():
  """ Prints usage info. """

  f = sys.stderr
  print >> f, ""
  print >> f, "Usage: " + sys.argv[0] + " kG kT kD nT",
  print >> f, "file.csfastq outfile.txt"
  print >> f, ""
  print >> f, "\tkG:\t\tk-mer size for de Bruijn graph construction."
  print >> f, "\tkT:\tMinimum count for k-mer inclusion in de Bruijn graph."
  print >> f, "\tkD:\t\tk-mer size for contig index used in votation."
  print >> f, "\tnT:\tThreshold for sequence occurence to be considered."
  return


def main():
  """ Main body of the application. """

  global collection             # For debugging purposes

  if not (len(sys.argv) == 6 or len(sys.argv) == 7):
    help()
    return

  kG = int(sys.argv[1])         # k for de Bruijn graph
  kT = int(sys.argv[2])         # Threshold for k-mer coverage in DBG
  kD = int(sys.argv[3])         # k for contig votation
  nT = int(sys.argv[4])         # Threshold for sequence occurence
  fname = sys.argv[5]           # File name to read input from
  outfile = sys.stdout

  if len(sys.argv) == 7:
    if os.path.isfile(sys.argv[6]):
      print >> sys.stderr, "ERROR: filename " + sys.argv[6] + " already exists."
      help()
      return
    outfile = open(sys.argv[6], "w+")  # File to write the output
  if not os.path.isfile(fname):
    print >> sys.stderr, ""
    print >> sys.stderr, "ERROR: filename " + fname + " does not exist."
    help()
    return

  collection = Collapse(fname, kG, kT, kD, nT)
  print >> outfile, collection.contigs
  if outfile != sys.stdout:
    outfile.close()
  return
```

```
if __name__ == "__main__":
  main()
```

Module *seqs.py*

```
##############################################################################
### miRNAligner - microRNA alignment utility for smallRNAseq data |coding: utf8
##############################################################################
### Module seqs
##############################################################################
### Author: Francisco D. Morón-Duran <fdmoron@iconcologia.net>
### Description: Collection of Sequences with different Reads
##############################################################################


from poll import Poll
from debg import Assembly
from math import sqrt
import sys

class Collapse:
  """ Collection of reads sharing common sequences. """

  def __init__(self, fname, kg = 11, kT = 2, kd = 6, nT = 5):
    """ Initializing Collection dictionaries and attributes. """

    self.numReads = 0     # Total number of input reads
    self.uniReads = 0     # Total number of unique sequences
    self.minLen   = 0     # Length of the shortest sequence
    self.maxLen   = 0     # Length of the longest sequence
    self.avgQual  = []    # Per-base average quality
    self.baseQual = {}    # Per-base accumulative quality
    self.bsequence= {}    # All reads indexed by basespace sequence
    self.sequence = {}    # All reads indexed by colorspace sequence
    self._collect(fname)  # Read input file
    self._computeQuals()  # Update consensus qualities for each sequence
    self._printSummary()  # Print summary of input reads
    seqs = [x for x in self.sequence.values() if x.n > nT] # Filter seqs by n
    print >> sys.stderr, str(len(seqs)) + " considered seqs."
    seqs.sort(key = lambda x: x.n)  # Order seqs by n
    self.contigs = Assembly(seqs, kg, kT)  # Get contigs from De Bruijn G
    self.poll = Poll(seqs, self.contigs, kd)
    return


  def _collect(self, fname):
    """ Read csfastq input. """

    f = open(fname)
    print >> sys.stderr, "Reading input file."
    while True:
      i = f.readline().strip()[1:]              # Read identifier
      if not i:
        break;                                  # EOF
      c = f.readline().strip()                  # Colorspace sequence
      f.readline().strip()                      # Plus sign (FastQ separator)
      q = f.readline().strip()                  # Per-base quality values
      l = len(c) - 1                            # Length of bspace seq
      self.numReads += 1                        # Update counter of input reads
      if not c in self.sequence:
        self.sequence[c] = Sequence(i, c, q, l) # Index colorspace
        self.bsequence[self.sequence[c].b] = self.sequence[c] # Index basespace
        self.uniReads += 1                      # Update unique sequences count
      else:
        self.sequence[c]._addRead(i, c, q)      # Add read to defined sequence
      if self.minLen == 0 or l < self.minLen:
        self.minLen = l                         # New minimum length
      if self.maxLen == 0 or l > self.maxLen:
        self.maxLen = l                         # New maximum length
    f.close()
    return
```

```
  def _computeQuals(self):
    """ Update consensus qualities for Sequences. """

    for i in range(1, self.maxLen + 1):
      self.baseQual[i] = { 'q': 0, 'n': 0 }
    for i in self.sequence.values():
      c = 1
      for j in i.getQual():
        self.baseQual[c]['q'] += j
        self.baseQual[c]['n'] += 1
        c += 1
    for i in range(1, self.maxLen + 1):
      self.avgQual.append(self.baseQual[i]['q'] / self.baseQual[i]['n'])
    return


  def _printSummary(self):
    """ Writes the summary of the reads. """

    print >> sys.stderr, "There are " + str(self.numReads),
    print >> sys.stderr, "reads collapsing into",
    print >> sys.stderr, str(self.uniReads) + " unique reads."
    return



class Sequence:
  """ Group of unique colorspace sequences. """

  def __init__(self, i, c, q, l):
    """ Creates a new sequence from a read. """

    self._q = []                  # Cumulative per-base quality
    for j in q:
      self._q.append(ord(j))
    self._qm       = self._q      # Average per-base quality
    self._qm_valid = True         # Validity flag for the qm average
    self._aq       = 0            # Average sequence quality
    self._aq_valid = False        # Validity flag for the aq average
    self.l         = l            # Lenght of the basespace sequence
    self.c         = c            # Colorspace sequence
    self.b         = self._cs2bs()# Basespace sequence
    self.i         = [i]          # List of read identifiers with this sequence
    self.n         = 1            # Number of reads with this sequence
    self.bruijn    = {}           # De Bruijn graph
    self.contigs   = set()        # Contigs to which seq has voted
    self.poll      = None
    return


  def __lt__(self, seq):

    return self.l < seq.l


  def __le__(self, seq):

    return self.l <= seq.l


  def __gt__(self, seq):

    return self.l > seq.l


  def __ge__(self, seq):

    return self.l >= seq.l


  def _cs2bs(self):
```

```
  """ Gives basespace sequence from a colorspace input. """

  d = { 'T': "TGCA", 'A': "ACGT", 'C': "CATG", 'G': "GTAC" } # Transition tab
  ret = []
  c = self.c[0]                       # current colorspace character to treat
  for i in range(1, self.l + 1):      # Skip the first base (primer)
    c = d[c][int(self.c[i])]          # Base obtention from transition table
    ret.append(c)
  return "".join(ret)


def _addRead(self, i, c, q):
  """ Adds read with identifier id to the Sequence. """

  if self.c != c:
    return False                      # Read does not belong to this sequence
  self._qm_valid = False              # qm is calculated only on demand
  self._aq_valid = False              # aq is calculated only on demand
  self.i.append(i)                    # Count read as a new sequence item
  self.n += 1                         # Update counts for the sequence
  n = 0
  for j in q:                         # Save the quality for each transition
    self._q[n] += ord(j)
    n += 1
  return True


def getQual(self):
  """ Gets average quality for each sequenced transition. """

  if not self._qm_valid:                        # Update mean quality value
    self._qm = []
    for i in self._q:
      self._qm.append(i/self.n)                 # Average qual for this base
    self._qm_valid = True                       # Sets validity flag
    self._aq_valid = False
  if not self._aq_valid:
    self.getAvgQual()
  return self._qm


def getAvgQual(self):
  """ Gets average quality for the sequence. """

  if not self._qm_valid:
    self.getQual()
  if not self._aq_valid:
    self._aq = 0
    for i in self._qm:
      self._aq += i
    self._aq = self._aq / len(self._qm)
    self._aq_valid = True
  return self._aq


def kmers(self, k):
  """ Builds list of k-mers for the reads. """

  r = []
  for i in range(0, self.l - k + 1):
    r.append([ self.b[i:i + k], i ])
  return r


def getKmFreqs(self, k, i = 0, f = None):
  """ Gets frequencies of k-mers in sequence. """

  if f is None:
    f = self.l - 1
  d = {}
  for km, ps in self.kmers(k):
```

```
      if ps < i or ps > f - k + 1:
        continue
      if not km in d:
        d[km] = 0
      d[km] += 1
    tk = f - i - k + 2
    for km in d:
      d[km] /= float(tk)
    return d


  def decideContig(self):
    """ Decide to which contig do we assign the sequence.
        Keeps nearest most voted contig. """

    sl = sorted(self.contigs, key = lambda c: (len(c.v[self]),
                                               -self.distance(c),
                                               c.s))
    for c in sl[:-1]:
      c.d[self] = sl[-1]
      c.n -= self.n*len(c.v[self])
      del c.v[self]
      self.contigs.remove(c)
    return


  def distance(self, other):
    """ Computes Mahalanobis distance between two basespace strings. """

    if self.poll == None:
      return False
    if not isinstance(other, dict):
      si, oi, sf, of = 0, 0, self.l - 1, other.l - 1
      if other in self.contigs:
        o = other.v[self][0][0] - other.v[self][0][1]
        if o > 0:
          oi = o
        if o < 0:
          si = -o
        of = min(of, o + self.l - 1)
        sf = min(sf, other.l -1 - o)
      f1 = self.getKmFreqs(self.poll.kv, si, sf)
      f2 = other.getKmFreqs(self.poll.kv, oi, of)
    else:
      f1 = self.getKmFreqs(self.poll.kv)
      f2 = other
    kms = set(f1.keys()) | set(f2.keys())
    for km in [x for x in f1.keys() if x not in f2]:
      f2[km] = 0.0
    for km in [x for x in f2.keys() if x not in f1]:
      f1[km] = 0.0
    return sqrt(sum([((f1[km] - f2[km])/self.poll._kc[km])**2 for km in kms]))
```

Module *debg.py*

```
################################################################################
### miRNAligner - microRNA alignment utility for smallRNAseq data |coding: utf8
################################################################################
### Module dbg
################################################################################
### Author: Francisco D. Morón-Duran <fdmoron@iconcologia.net>
### Description: De Bruijn Graph implementation. Definition of Assembly and
###              Contig classes
################################################################################

from math import sqrt
import sys

def kmers(seq, k):
  """ Yields k-mers belonging to seq. """

  for i in xrange(len(seq) - k + 1):
    yield seq[i:i + k]


def _fw(km):
  """ Yields next possible forward k-mers for km. """

  for x in 'ACGT':
    yield km[1:] + x


def _bw(km):
  """ Yields next possible backward k-mers for km. """

  for x in 'ACGT':
    yield x + km[:-1]



class Dbg:
  """ Class for De Bruijn Graph. """

  def __init__(self, seqs, k, threshold):
    """ Init graph from seqs with k-mers present more than threshold times. """

    print >> sys.stderr, "Building De Bruijn Graph from collected reads."
    self.G = {}
    for seq in seqs:
      for s in seq.b.split('N'):  # Split reads with unknown bases
        for km in kmers(s, k):    # Get k-mers from each sequence
          if not km in self.G:
            self.G[km] = 1        # Initialize new k-mer
          else:
            self.G[km] += 1       # Add k-mer coverage
    lowcov = [x for x in self.G if self.G[x] <= threshold] # List low covered
    for x in lowcov:
      del self.G[x]               # Remove low covered k-mers from graph
    print >> sys.stderr, str(len(self.G)) + " total k-mer nodes in the graph."
    return


  def get_contig_fw(self, km):
    """ Navigate graph forwards from km while reaching non-ambiguous paths. """

    c = [km]                  # First k-mer
    while True:
      if sum(x in self.G for x in _fw(c[-1])) != 1:
        break                 # One possible path only!
      cand = [x for x in _fw(c[-1]) if x in self.G][0]  # Next candidate k-mer
      if cand == km:
        break                 # Break cycles! # or Möbius contigs
      if sum(x in self.G for x in _bw(cand)) != 1:
```

```
        break                 # Candidate should be reached by last k-mer only!
      c.append(cand)          # Append candidate unambiguous k-mer path
    return c


  def get_contig_bw(self, km):
    """ Navigate graph backwards from km while reaching non-ambiguous paths. """

    c = [km]
    while True:
      if sum(x in self.G for x in _bw(c[0])) != 1:
        break
      cand = [x for x in _bw(c[0]) if x in self.G][0]
      if cand == km:
        break
      if sum(x in self.G for x in _fw(cand)) != 1:
        break
      c.insert(0, cand)     # Insert candidate at the begining of path
    return c


  def get_contig(self, km):
    """ Get unambiguous path containing k-mer (if it exists). """
    fw = self.get_contig_fw(km) # Forward path
    bw = self.get_contig_bw(km) # Backward path
    bw = bw[:-1]                # Remove km from backward path (present in both)
    if km in _fw(fw[-1]):
      c = fw                    # bw path is fw as well?
    else:
      c = bw + fw               # Merge fw and bw paths
    # Return contig, k-mer path and k-mer coverage
    return self.contig2string(c), c, [self.G[x] for x in c]


  def contig2string(self, c):
    """ Write sequence string from k-mer path. """

    return c[0] + ''.join(x[-1] for x in c[1:])


  def all_contigs(self):
    """ Get all unambiguous paths contained in the graph. """

    done = set()  # Set of visited k-mers
    r = []        # List of contigs to return
    for x in self.G:
      if x not in done:
        s, c, cov = self.get_contig(x)  # Get seq, k-mers and k-mer coverage
        for y in c:
          done.add(y) # Flag as visited all k-mers in the contig
        r.append(s)
    return r



class Assembly:
  """ Collection of contigs. """

  def __init__(self, seqs, k, threshold):
    """ Initialize from a list of contig sequences. """

    self.d = Dbg(seqs, k, threshold)
    self.n = 0
    self.c = {}
    self.k = k
    print >> sys.stderr, "Extracting unique unambiguous contigs."
    for x in self.d.all_contigs():
      self.c[x] = Contig(x, k)
      self.n += 1
    print >> sys.stderr, str(self.n) + " total contigs were generated."
    return
```

```
  def __getitem__(self, x):
    """ Get contig x from assembly. """

    if x in self.c:
      return self.c[x]
    return False

  def __repr__(self):
    """ . """

    i = 1
    o = ""
    for x in sorted([c for c in self.c],
                    key = lambda p: (-self.c[p].n, -len(self.c[p].v))):
      o += "> contig_" + str(i) + " votes: " + str(self.c[x].n)
      o += " sequences: " + str(len(self.c[x].v)) + " counts: "
      o += str(sum([s.n for s in self.c[x].v])) + "\n"
      o += self.c[x].__repr__(False) + "\n"
      i += 1

    return o


  def updateCovs(self):
    """ Force update of contig coverage values. """

    for c in self.c.values():
      c.updateCov()
    return


  def checkIntegrity(self):
    """ Checks coverage of contigs by seqs is increasing. """

    self.sanitize()
    for c in self.c.values():
      c.checkIntegrity()
    self.sanitize()
    for c in self.c.values():
      c.removeOutliers()
    self.sanitize()
    return


  def sanitize(self):
    """ . """

    d = set([c for c in self.c if len(self.c[c].v) == 0])
    for c in d:
      del self.c[c]
    return



class Contig:
  """ Contig from a unique unambiguous path in the grpah. """

  def __init__(self, x, k):
    """ Initialize from a contig sequence. """

    self.s = x                              # Contig sequence
    self.l = len(x)                         # Contig length
    self.c = []                             # Per-base coverage
    self.n = 0                              # Number of contig votes
    self.v = {}                             # Votes contenidor
    self.o = 0                              # Offset (to print contig)
    self.d = {}                             # Discarded reads
    self.k = k
    self.cons = ""                          # Consensus seq
```

```
    return


  def __repr__(self, verbose = True):
    """ Print graphical representation of a contig. """

    o0 = abs(self.o)
    if verbose:
      for i in xrange(o0):
        print '',
      print self.s
      ss = self.v.keys()
      ss.sort(key = lambda seq: (-seq.distance(self),
                                 len(self.v[seq]),
                                 seq.n),
              reverse = True)
      for s in ss:
        o = self.v[s][0][0] - self.v[s][0][1] + o0
        for i in xrange(o):
          print '',
        print s.b,
        print "(x" + str(s.n) + ",",
        print "v" + str(len(self.v[s])) + ",",
        print "d" + str(s.distance(self)) + ")"
    return self.cons


  def __len__(self):
    """ Return number of sequences in the Contig. """

    return len(self.v)


  def _iqrOutlier(self, l, side = "upper"):
    """ Finds interquantile range and return a threshold for outliers. """

    n = len(l)
    if n < 3:
      return False
    if (n+1)%4 == 0:
      q1 = l[(n+1)/4-1]
      q3 = l[3*(n+1)/4-1]
    else:
      q1 = (l[(n+1)/4-1]+l[(n+1)/4])*.5
      q3 = (l[3*(n+1)/4-1]+l[3*(n+1)/4])*.5
    iqr = q3 - q1
    if side == "upper":
      return q3 + 1.5*iqr
    else:
      return q1 - 1.5*iqr
    return


  def updateCov(self):
    """ Update coverage values per contig base and get consensus sequence. """

    os = [self.v[s][0][0] - self.v[s][0][1] for s in self.v]
    ls = [s.l + self.v[s][0][0] - self.v[s][0][1] for s in self.v]
    self.o = abs(min(0,min(os)))
    length = max(ls) + self.o
    b = [{} for i in range(length)]
    self.c = [0 for i in range(length)]
    for c in self.v.keys():
      o = os.pop(0) + self.o
      o2 = 0
      for i in c.b:
        if not i in b[o + o2]:
          b[o + o2][i] = 0
        b[o + o2][i] += c.n
        o2 += 1
      for o in range(len(b)):
```

```
      total = float(sum(b[o].values()))
      self.c[o] += total
      if len(b[o]) > 0:
        for i in b[o]:
          b[o][i] /= total
  self.cons = ''.join([sorted(b[o].keys(),
    key = lambda n: -b[o][n])[0] for o in range(len(b)) if len(b[o]) > 0])
  return


def checkIntegrity(self):
  """ Removes trivial non-matching sequences. """

  d = set()
  for s in self.v:
    v0 = self.v[s][0]
    for v in self.v[s][1:]:
      if not v0 <= v:
        # Both coords. in v0 must <= than their respectiv. in v
        d.add(s)
        break
      v0 = v
  for s in d:
    self.d[s] = self.v[s]
    self.n -= s.n
    del self.v[s]
  return


def removeOutliers(self):
  """ Remove outliers flagged by interquantile range and confirmed.
      with Mahalanobis distance to centroid. """

  s = sorted(self.v, key = lambda seq: seq.distance(self))
  l = [seq.distance(self) for seq in s]
  d = set()         # Set of sequences to remove from contig
  threshold = self._iqrOutlier(l)
  if not threshold: # Too few sequences
    return
  o = sum([i > threshold for i in l])
  if o > 0:
    cands = s[-o:]  # Candidates to outlier
    group = s[:-o]  # Reliable group
    cgroup = self.centroid(4, set(group)) # Get centroid
    lgroup = [seq.distance(cgroup) for seq in group] # Dists. to centroid
    m = sum(lgroup)/len(lgroup)     # Mean distance
    d2 = [(i-m)**2 for i in lgroup] # Covs.
    v = sqrt(sum(d2)/len(d2))  # Std. error
    for x in cands:
      if x.distance(self) > v: # Remove seqs with distance > std. error
        d.add(x)
  for s in d:                  # Delete flagged sequences from contig
    self.d[s] = self.v[s]
    self.n -= s.n
    del self.v[s]
  return


def updateVote(self, s, v, k):
  """ Updates the votes for the contig. """

  if not s in self.v:
    self.v[s] = []
  self.v[s].append(v)
  self.n += s.n
  return


def kmers(self, k):
  """ Builds list of k-mers for the contig """
```

```
    r = []
    for i in range(0, self.l - k + 1):
      r.append([ self.s[i:i + k], i ])
    return r


  def centroid(self, k, seqs = set()):
    """ . """

    c = {}
    l = len(self.v)
    for seq in self.v:
      if len(seqs) > 0 and seq not in seqs:
        continue
      p = seq.getKmFreqs(k)
      for km in p:
        if not km in c:
          c[km] = 0
        c[km] += p[km]
    for km in c:
      c[km] /= l
    return c


  def getKmFreqs(self, k, i = 0, f = None):
    """ Gets frequencies of k-mers in sequence. """

    if f is None:
      f = self.l - 1
    d = {}
    for km, ps in self.kmers(k):
      if ps < i or ps > f - k + 1:
        continue
      if not km in d:
        d[km] = 0
      d[km] += 1
    tk = f - i - k + 2
    for km in d:
      d[km] /= float(tk)
    return d
```

Module *poll.py*

```
################################################################################
### miRNAligner - microRNA alignment utility for smallRNAseq data |coding: utf8
################################################################################
### Module poll
################################################################################
### Author: Francisco D. Morón-Duran <fdmoron@iconcologia.net>
### Description: Dictionaries of sequences indexed by their contained k-mers.
###              Definition of Vote. Executor of the vote-and-seed process.
################################################################################


import sys


class Poll:
  """ Object to take care of the votation process. """

  def __init__(self, seqs, contigs, k):
    """ Initialize k-mers dictionary from contigs. """

    self.k = k                  # K-mer k value
    self.kv = 4                 # K-mer value for distance computation
    self.voted = False          # Some operations only make sense after votation
    self._dc = {}               # K-mer dictionary to index contigs
    self._ds = {}               # K-mer dictionary to index sequence sets
    self._kf = {}               # K-mer expected frequencies
    self._kc = {}               # K-mer frequencies covariances
    self._c = 0                 # Counter of voted sequences
    self._t = 0                 # Total k-mers value
    print >> sys.stderr, "Indexing contigs into a dictionary of",
    print >> sys.stderr, str(k) + "-mers."
    for contig in sorted(contigs.c.values(), key = lambda x: x.n):
      self._seed(contig)        # Seed dictionary with k-mers from contig
    print >> sys.stderr, "Starting votation of contigs by read sequences."
    print >> sys.stderr, "Indexing sequences into a dictionary of",
    print >> sys.stderr, str(self.kv) + "-mers."
    for seq in seqs:
      self._index(seq)
      self._vote(seq)           # Vote sequences with k-mers from seq
      seq.poll = self
    self.voted = True
    self._initCovariances()
    for seq in seqs:
      seq.decideContig()
    contigs.checkIntegrity()
    contigs.updateCovs()
    print >> sys.stderr, "Votation done."
    return


  def _index(self, seq):
    """ Index sequence by its k-mers. """

    for km, ps in seq.kmers(self.kv):
      if not km in self._ds:
        self._ds[km] = set()
      if not km in self._kf:
        self._kf[km] = 0
      self._ds[km].add(seq)
      self._kf[km] += seq.n
      self._t += seq.n
    return


  def _seed(self, contig):
    """ Store contig indexed by its k-mers. """

    for km, ps in contig.kmers(self.k):    # Get k-mer and its position in seq
```

```
      if not km in self._dc:
        self._dc[km] = set()                  # Initialize empty k-mer entry
      self._dc[km].add(tuple((contig, ps)))  # Store sequence and k-mer position
    return


  def _vote(self, seq):
    """ Store votes generated from seq into contigs dictionary. """

    for km, ps in seq.kmers(self.k):         # Get k-mers and positions in seq
      if km in self._dc:
        for cseq, cps in self._dc[km]:           # Get positions in indexed reads
          cseq.updateVote(seq, Vote(cps, ps), self.k)   # Vote contig
          seq.contigs.add(cseq)
    self._c += 1                              # Sum number of voted sequences
    print >> sys.stderr, "\r" + str(self._c) + "\tprocessed pieces of data.",
    sys.stdout.flush()                        # Force to empty buffer
    return


  def _initCovariances(self):
    """ . """

    if not self.voted:
      return False
    k = self.kv
    c = self._c
    for km in self._ds:
      # Compute expected frequency for km
      l = self._ds[km]
      self._kf[km] = sum([s.getKmFreqs(k)[km] for s in l])/float(c)
      # Compute covariance for km
      f = self._kf[km]
      r = ((c - len(l) + 1)*f**2)/(c - 1)
      self._kc[km] = sum([(s.getKmFreqs(k)[km] - f)**2 for s in l])/(c - 1) + r
    return


  def getKmFreqs(self, kms):
    """ Gets frequency of k-mer km in input reads. """

    if not self.voted or self._t == 0:
      return False
    d = {}
    for km in kms:
      if not km in d and km in self._kf:
        d[km] = self._kf[km]/float(self._t)
    return d



class Vote:
  """ Vote supporting a k-mer match. """

  def __init__(self, p1, p2):
    """ Vote initialization from sequence indexes. """

    self.p1 = p1  # Position of the k-mer in sequence1
    self.p2 = p2  # Position of the k-mer in sequence2
    return


  def __getitem__(self, i):
    """ Retrieve individual vote values. """

    ret = False
    if i == 0:
      ret = self.p1
    elif i == 1:
      ret = self.p2
    return ret
```

```
def __repr__(self):
  """ Written representation of a vote '(p1, p2)'. """

  return str(tuple((self.p1, self.p2)))


def __add__(self, other):
  """ Sum of two votes. """

  return Vote(self.p1 + other.p1, self.p2 + other.p2)


def __sub__(self, other):
  """ Substraction of a vote from another. """

  return Vote(self.p1 - other.p1, self.p2 - other.p2)


def __eq__(self, other):
  """ Equalty of votes. """

  return (self.p1 - other.p1 == 0) and (self.p2 - other.p2 == 0)


def __ne__(self, other):
  """ Difference of votes. """

  return (self.p1 - other.p1 != 0) or (self.p2 - other.p2 != 0)

def __le__(self, other):
  """ Lower equal. """

  return self.p1 <= other.p1 and self.p2 <= other.p2
```

Annex C: Script for BLAST+ alignments

```
#!/bin/bash

## NCBI BLAST+ 2.2.31 Universal binaries for Mac OS X downloaded from
## ftp://ftp.ncbi.nlm.nih.gov/blast/executables/blast+/

## Build miRBase database from annotated FASTA input
./makeblastdb -in miRBase.fa -dbtype nucl -out miRBase

## BLAST against database pipeline output
./blastn -query miRNAligner_output.fa -db miRBase -word_size 6 -outfmt 6 \
       | sort -grk11 \     # Sort by decreasing E-VALUE (lowest on bottom)
       | awk '{c[$1] = $0}; END {for (x in c) { print c[x] }}' \ # Keep lowest E-VAL
       | sort -gk11 \ # Sort by increasing E-VALUE (lowest on top)
      > miRBase_matches.txt # Final ranking of matches
```