**A Pipeline for the *de novo* Identification of Long Non-Coding RNAs**

**Preface:**

This document is simply an expanded methods section to describe the pipeline. It details the specific usages of each program used along the pipeline and describes how one could follow along at home, using their own RNAseq data.

Often times the descriptions will be indiscriminant. This will mostly occur by usages of vague terms like “accession”, “longAccesion”, or “species”. If one is following the pipeline, these will be up to the user. For example, if one of my RNAseq accessions is 588990 for the species “SpecX”, in step 5, I will have a file called 588990.no\_annotation.headers.fasta that will move forward. It will then be combined and compared to another accession. This combined file would be more appropriately named master\_longest\_SpecX.fasta.

When this pipeline is fully constructed these will be defined by the user and the script will handle proper naming.

The Pipeline



Remove known ncRNA

Validate through coding potential

3,223

3,210

Compare sequences across data sets

Assemble RNAseq reads *de novo*

Internally cluster fasta files

Filter lowly expressed contigs

Filter known proteins



RNAseq by Expectation Maximization



QC-derived reads

157,779

110,900

44,555

10,292

62,745

48,769

46,699

13,755

3,529

1.03x108

1.40x107

*De novo* Pipeline

0.

4.

1.

3.

5.

2.

6.

7.

Step by step analysis of the pipeline:

**0. Pre-pipeline processing:**

In order to execute the steps of the pipeline, many people are going to want to first process reads into whatever you consider sufficiently high quality. Here, reads were assorted into bins corresponding to a directional, trial specific barcode using the FASTX tool fastx\_barcode\_splitter. Reads were then processed via FastQC, and trimmomatic is used to remove adapters.

Appropriately selected reads then existed in library-specific FASTQ files. The files were named along the line of species-accession\_library\_direction.fq. All libraries were then concatenated into one file representing the species’ accession and the direction (named accession\_ALL\_direction.fq). Due to the nature of QC, some reads were no longer paired-end reads, so reads were not paired in file. In order to overcome this, reads were paired using pairfq and the following code:

i=accession

PAIRFQ=/path/to/pairfq\_lite.pl

$PAIRFQ makepairs \

-f $i.ALL.left.fq \

-r $i.ALL.right.fq \

–fp \ $i.paired\_forward \

-rp $i.paired\_reverse \

-fs $i.unpaired\_forward \

-rs $i.unpaired\_reverse \

-s > $i.pairfq.log

**1. *de novo Assembly* via Trinity**

Trinity was chosen for assembly due to its abundance of documentation and downstream analysis tools, particularly Trinotate (to which we will come back later). For the execution of Trinity, it is recommended that the workstation or cluster used have 1 GB of RAM for every one million reads. For the data analyzed here, we worked with two data sets: one of 14 million reads and one of 103 million reads (across forward and reverse reads).

Note that Trinity was installed in the system by which this code was executed. If Trinity is not root-installed, either an alias or variable should point to the execution script (examples below):

In the .bashrc file,

alias Trinity=/path/to/Trinity

In an execution script or in the terminal

Trinity=/path/to/Trinity

All contigs were analyzed via the following set up. Note that the --max\_memory and –CPU flags are almost definitely different for your system. Also note that these were not even optimal under our conditions.

LEFT\_PAIRED=point/to/appropriate/file

LEFT\_UNPARIED= point/to/appropriate/file

RIGHT\_PAIRED= point/to/appropriate/file

RIGHT\_UNPAIRED= point/to/appropriate/file

Trinity

--seqType fq

--max\_memory 25G

--SS\_lib\_type FR

--CPU 1

--left $LEFT\_PAIRED,$LEFT\_UNPAIRED

--right \ $RIGHT\_PAIRED,$RIGHT\_UNPAIRED

--output $i.paired\_trinity\_out\_dir

> $i\_trinity.log

**2. Internal clustering via CD-HIT**

Contigs reported by Trinity are clustered into loosely-defined gene clusters. These clusters still represent an incredible amount of redundant information in the form of what Trinity putatively calls isoforms. For the sake of conservatism, and a desire to not overestimate the number of lncRNA in *Vitis vinifera*, we opted to further cluster these contigs via use of CD-HIT.

For the most part, default parameters for CD-HIT were used:

cd-hit-est

-i Trinity.fasta

-n 5

-o clust\_Trinity.fasta

-c 0.90

-m 8000

-T 6

**3. Filtering Lowly Expressed Transcripts Via RSEM**

Due to the current state of lncRNA prediction, it is imperative to remove lowly expressed reads because anything that could be considered “noise” could be identified as an lncRNA. Trinity provides support for both alignment-based and alignment-independent transcript quantification tools. For this analysis we opted to use the alignment-based tool RSEM. Trinity provides a script for the execution of RSEM (and every method it supports), one only need to point to the file location of the execution script.

For this execution, all tools used in the analysis must be stored in the environmental variable PATH. One easy way to do this is to edit the .bashrc to define new environmental variables for each of the tools. For example, our local .bashrc has the variables defined as such:

export TRINITY\_DIR=/path/to/Trinity

export BOWTIE\_DIR=/path/to/bowtie

export RSEM\_DIR=/path/to/RSEM

Then the path can be redefined to include the appropriate tools:

export PATH=$PATH:$TRINITY\_DIR:$BOWTIE\_DIR:$RSEM\_DIR

RSEM can then be called from the Trinity script as follows:

LEFT\_PAIRED=point/to/appropriate/file

LEFT\_UNPARIED= point/to/appropriate/file

RIGHT\_PAIRED= point/to/appropriate/file

RIGHT\_UNPAIRED= point/to/appropriate/file

$TRINITY\_DIR/util/align\_and\_estimate\_abundance.pl

--seqType fq

--transcripts clust\_Trinity.fasta

--left $LEFT\_READS\_PAIRED,$LEFT\_READS\_UNPAIRED

--right $RIGHT\_READS\_PAIRED,$RIGHT\_READS\_UNPAIRED

--SS\_lib\_type FR

--est\_method RSEM

--aln\_method bowtie

--trinity\_mode

--prep\_reference

This generates two particular files of interest: RSEM.isoforms.results and RSEM.genes.results. After analyzing both files, for simplicity of the pipeline, we decided to exclusively analyze the isoforms file.

The particular metric by which to filter seems to be differentially advocated by different researchers. As a simple approach, we opted to use the metric FPKM as a filtering parameter. Filtering by FPKM seems to be a fairly common practice but the value by which we consider something expressed is not standardized. Trinity provides a method for filtering these results, but for generalizations, we need only one line of script. Here, we choose 1.50 as our filter in the name of conservatism.

awk ‘{ $7 >= 1.50 }’ RSEM.isoforms.results |

cut –f1 |

cut –f1 –d” “ > filt\_clust\_Trinity.headers

Following this step, the data structure of will have changed from a fasta file to a list of sequences. This list can easily be converted back to a fasta file via samtools. A general script for this task could look something like this, assuming root installation of samtools.

input\_file=filt\_clust\_Trinity.headers

ref\_fasta=clust\_Trinity.fasta

output\_file=$input\_file.fasta

counter=0

lc=`wc -l $input\_file | cut -f1 -d" "`

limit=$(( $lc + 1 ))

until [[ $counter -eq $limit ]]; do

read line

samtools faidx $ref\_fasta $line

counter=$(( $counter + 1 ))

done < $input\_file > $output\_file

**4. Removing Known Protein Coding Genes with Trinotate**

Trinotate is a very lovely software that compiles the results of various gene-identification tools into one user-friendly output file that can easily be parsed by awk, sqlite, or even Excel. Since the formulation of this pipeline, Trinotate has been updated and has dropped support for one of the databases by which user-assembled sequences can be compared. Because of this, we do use an older version of the software. If you are working with a model organism for which many genes have been identified, the new version should work just fine. But for non-model organisms, using the new version could lead to false discovery of lncRNA.

In order for Trinotate to be executed, several steps have to happen. All of this is explained very thoroughly on the Trinotate website (linked below), so this will only focus on what we did to get to the compilation step.

Most of the programs called for by Trinotate require the input of protein sequences, so we use Trinity’s downstream tool TransDecoder for this task. TransDecoder is easily called by:

export PATH=$PATH:$TRINITY\_DIR

export TRANSDECODER\_EXEC=path/to/TransDecoder-2.0.1/TransDecoder.LongOrfs

$TRANSDECODER\_EXEC -t filt\_clust\_Trinity.headers.fasta

The most important program for our use of Trinotate is BLAST+. As such, we aligned our contigs to two different versions of the Blast database, Uniprot-Sprot and Uniprot-Uniref90 as available for download on the Trinotate website. Blast searches (of which there will be four) were executed in the following manner:

query=filt\_clust\_Trinity.fasta

db=/path/to/database

out=accession\_database

makeblastdb

-in $db

-dbtype prot

blastx

-query $query

-db $db

-num\_threads 16

-evalue 1e-20

-max\_target\_seqs 1

-outfmt 6

> $out.outfmt6\_x

Note that two of the BLAST searches will be blastp, for which we wrote the output to $out.outfmt6\_p.

BLAST is the only program that was used in a manner other than as directly as specified by the Trinotate documentation. Again, this was in the effort of conservatism; we did not wish falsely identify transcripts as protein coding.

Query transcripts in a BLAST search will often find more than one hit, as such we use a simple sort function to select the best hit for each transcript. Here, we defined this as a function in our .bashrc file as follows:

function sortBlast {

sort -k1,1 -k12,12gr -k11,11g -k3,3gr $1 | sort -u -k1,1

–-merge > SORTED\_$1

}

This function sorts by transcript ID, then sorts by highest bit-score, lowest e-value, highest percent identity, and merging across a unique sort.

Calling this function is as easy as

sortBlast accession\_sprot.outfmt6\_x

Credit for this function should be given to /u/5heikki on BioStars.

As per the instructions of Trinotate, the results were loaded into an SQLite database, and Trinotate calls the appropriate queries to generate an output Report. Following this result, we filtered the Report via bash/SQLite script (in the GitHub directory) that separates hits into 3 bins. Those that hit Viridiplantae, those that hit something other than Viridiplantae, and those that hit nothing. Those that hit nothing, were carried forward for analysis. *NOTE: If you do not work with green plants, you might want to change every incidence of “viridiplantae” to something else.*

This would be easily accomplished by the following

sed –i ‘s/[vV]iridiplantae/somethingNew/g’ script

Following the output of sql\_filter.sh, files will have changed naming scheme so that they can be compared across trials. For example, the script calls for the definition of acc, for accession. This allows the files from each trial to differ from one another.

accession.no\_annotation.headers will need to be converted back into a fast file. This can be done using the same loop above for samtools faidx. In this case, we would simply redefine the input\_file to accession.no\_annotation.headers. We can also redefine the ref\_fasta to filt\_clust\_Trinity.headers.fasta to save time building the fasta file.

Starting at this fasta (although it could be done much earlier for simplicity) we changed the scheme by which sequences are named to include the accession so that different accessions can be compared. Trinity, by default, names transcripts in the following manner: TR3756|c0\_g1\_i3. We simple replaced every incidence of “TR” with “accession.TR”. For examples, the new system could name the same transcript 588990.TR3756|c0\_g1\_i3. This is done with sed:

sed –i ‘s/TR/accession.TR/g’ accession.no\_annotation.headers.fasta

**5. Comparing Results Data Sets**

Many pipelines that attempt to classify transcripts as non-coding hinge on the combination of data sets, but this combination is often done before the assembly of transcripts. While, hypothetically, this should only lead to to better transcript assembly, it also tends to lead to a tremendous amount of noise. As such, we try a different approach. Rather than combining all data and then assembling, we assemble and then compare data sets. This is done with the seemingly simplest approach: BLAST.

So, until this step, it is assumed that in two different directories, this pipeline has been proceeding. One for each of two different RNAseq data sets. These two resultant files can be compared using a simple blastn.

For these types of analysis, we always searched for transcripts against the largest file. This can be easily determined by using grep to count the incidence of “>” per line.

grep –c “>” accession.no\_annotation.headers.fasta

To complete the BLAST, we defined the following:

query=/path/to/shortest

db=/path/to/longest

out=shortAccession\_to\_longAccession

makeblastdb

-in $db

-dbtype nucl

blastn

-query $query

-db $db

-num\_threads 16

-outfmt 6

> $out.outfmt6

As above, we used the sortBlast function to retrieve the best hits per transcript in the query file. To ensure significant homology, and following the operational definition of lncRNAs (>200 nt in length), we only considered the sequences the same if their overlap existed at over 200 nt. To do this, we filtered by awk:

awk ‘$4 >= 200’ SORTED\_shortAcession\_to\_longAccession.outfmt6

> filt\_SORTED\_shortAcession\_to\_longAccession.outfmt6

For each set of similar sequences, we needed only one sequences to proceed in analysis. For this, we put together a table that included information about the length of the sequence. Construction of this table is a bit involved, but conceptually very simple. To begin, we needed fasta files from each data sets of the sequences that found homology. To construct this, we used the following code:

cut –f1 filt\_SORTED\_shortAcession\_to\_longAccession.outfmt6

> shortAccession.headers

cut –f2 filt\_SORTED\_shortAcession\_to\_longAccession.outfmt6

> longAccession.headers

Again, we just used the loop over samtools, above, to generate new fasta files. Then we could calculate the length of the transcripts. To do this, we downloaded the tool bioawk, available at the link below. Length can be calculated using the following on the shorter and longer accession fasta files:

bioawk –c fastx ‘{ print name, length($seq) }’

shortAccession.headers.fasta > shortAccesison.len

These two files could be pasted together to generate a table with columns corresponding to the following: sequence name in the short accession, that sequence’s length, sequence name of the long accession, and that sequences length. This table can then be used to construct a master-longest-sequence file using the script in the repository grab\_longest\_fasta.sh.

**6. Removing Known ncRNA**

In order to remove known non-coding RNA from this master sequence file, we compared the results to the Rfam database by use of cmscan, a tool in the suite Infernal. Downloading the Rfam database generates a subdirectory called CMs. These CMs are the models by which cmscan looks for homology. cmscan can be called by the following:

cmscan –-tblout cmscan.tabout /path/to/CMs/Rfam.cm

master\_longest\_species.fasta > cmscan.out

The “tab-delimited” output for this is actually space delimited, so cut doesn’t work as well as awk. To filter these results, we considered all matches that Infernal dubbed significant to actually be significant. To get these results, we used the following:

awk ‘ $11 == “!” ‘ cmscan.tabout |

awk ‘{ print $1 }’ > filt\_cmscan.headers

To filter these results, we used SQlite. Particularly, we created a database and queried for transcript names that occurred in the master list, but NOT in the cmscan output. This can be completed using the script provided called compare\_list\_negative.sh

The resultant file, species.rfam\_filt.headers, will need to again be converted into a fasta file, which can easily be done by redefining the input\_list and ref\_fasta above and looping over the samtools faidx command.

**7. Validation through Pseudo-Independent Software**

The goal of this pipeline is to identify lncRNA without the use of machine learning algorithms that are based on predicted sequenced (like PLEK or CNCI). As such, we opted to use a machine learning algorithm based on features of sequences that, by definition, define lncRNA as a pseudo independent validation. Here, we opted to use CPC, the Coding Potential Calculator, that classifies sequences on six parameters: three involving a BLAST search and three involving the length and quality of an open reading frame. It’s use of BLAST invalidates the software as completely independent, but its use of different parameters does differentiate it.

CPC was ran using the following code:

fasta=species.rfam\_filt.headers.fasta

output=cpc.out

base\_out\_name=species

export CPC\_HOME=/path/to/CPC Directory

export CPC\_EXECUTABLE=$CPC\_HOME/bin/run\_predict.sh

$CPC\_EXECUTABLE $fasta $output $CPC\_HOME $base\_out\_name

The output of this file will be a four column table with the following titles: transcript\_id, length of transcript, classification, and the classification index. If the classification index is negative, it is classified as noncoding. If the index is positive, it is classified as coding. While this step is not meant to serve as a filtering mechanism, discrepancies were removed from the final list.

All remaining transcripts are now considered putative long non-coding RNAs.