Finding Transposons in DNA, A Python Approach

Louis Chatta
University of California, Santa Cruz: BME 160

Abstract

Jumping genes, or Transposons, can be found all throughout many biological domains. They are DNA segments that are constantly moving around a genome. These types of DNA segments have a wide range of use in the genome, from creating new mutations, to developing anti-biotic resistance in bacteria, to even DNA fingerprinting in genetic testing. Finding transposons manually can be daunting. It is difficult to search a sequence of DNA that may be thousands of lines long to find all IR(inverted repeats) or DR(direct repeats). This paper outlines a new method of finding all IRs and DRs in a genetic sequence from a FASTA file

1 Introduction

Transposons exist in two different forms Replicative(Retrotransposons, Class I), which copy DNA segments, and Non-replicative (Class II), which cuts the DNA segments. Both classes 'paste' into their target DNA. The Class I first transcribe the DNA into RNA and then use reverse transcriptase to make a DNA copy of the RNA to insert in a new location. The Class II use the Transpose enzyme, which binds with the IRs(Inverted Repeats), which then cleaves the phsophodiester bond at each end of the segment, then the nicked hairpins attack the new target DNA, which is then incorporated through the use of DNA Polymerase and DNA Ligase. This process will leave short direct repeats before and after the inverted repeats.[2]

There are many different subclasses of the Class I and Class II transposons. In Class I, there exists TEs(Transposable Elements with Long Term Repeats), SINES(Short Interspersed Transposable Elements) and LINES(Long Interspersed Transposable Elements). The LINES and SINES lack long term repeats, and are encoded with RNA polymerase I and II respectively. The TEs with long term repeats, encode reverse transcriptase, similiar to retroviruses like HIV. Combined, Class I transposons represent 30 % of the human genome, while Class II comprising of only 2 % of the human genome.[2] Another type of TE, is the autonomous TE. Autonomous TEs can move by themselves, while non-autonomous TEs need another TE to move. That is because dependent TEs lack transposase or reverse transciptase.

Overall many transposons are useless or harmful, but we can use them as a tool of fingerprinting DNA. They provide a good method of detailing inheritance. Finding transposons in the same sequence at the same positions gives direct evidence that two organisms have a common ancestor. An example of this is discovery that Whales and Hippos share a common ancestor. See figure [1].

Finding all the transposons in a sequnce can be tremendously useful, but quite difficult. For example, the Drosphila genome has 175,000,000 base pairs. That would take 1.8 years to just read the sequence figure [2]. Taking a programmatic approach to this problem, would dramatically reduce the computation time. This paper will discuss a python program that can take in any amount of sequences from a FASTA file and process each sequence and output all the possible Transposons by finding all IRs and DRs in a given sequence.

2 Methods

2.1 Program Overview

This program, findRepeats.py, reads in FASTA file with *N* amount of sequences, and outputs to the STDOUT all sequences with DRs and IRs , which can the be investigated further to be true Transposons. The program runs fast on short sequences, but begins to slow down on larger ones.

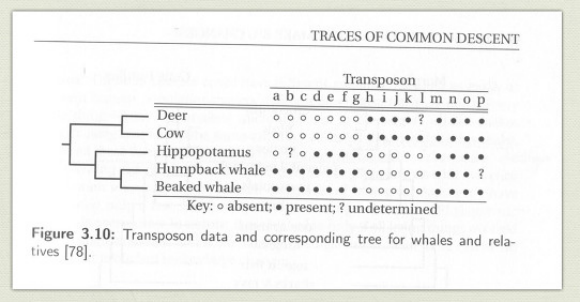

Figure 3.10: Transposon data and corresponding tree for whales and relatives [78].

Figure 1: This is a diagram of the relationship between whales and hippos. Based on Transposons. [1]

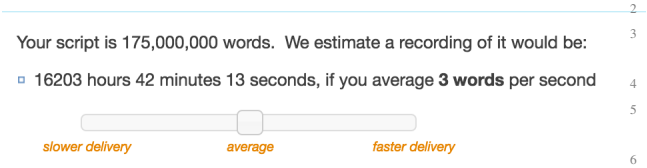

Figure 2: This is an estimated of how long it would take a human to read the Drosphilla genome. [3]

2.2 Program Architecture

2.2.1 Overview

The program is split into the following components

```

# previously written library, that reads a
    sequence from a FASTA file
from sequenceAnalysis import NucParams,
    FastAreader, ProteinParam

# handy utility class
class Util():

# handles all command line arguments
class CommandLine():

# handles all of the IRs and DRs
class RepeatFinder():

# handles all the flow
main():
```

Listing 1: overview

2.2.2 SequenceAnalysis.py

This is a program written from a previous lab. Its functionality is to read in FASTA files and generate readable sequences. Here is how it used:

```
from sequenceAnalysis import NucParams,
    FastAreader, ProteinParam

def main():
    for head, seq in orfReader.readFasta():
        nucParams = NucParams('')
        nucParams.addSequence(seq)
        nucParams.buildNucleotide()
        #nucParams.codons now has the sequence
```

Listing 2: sequenceAnalysis.py

2.2.3 Util Class Overview

This class is used a utility for the whole program. It contains two functions, one that returns the inverted repeat of a seqeunce and one that returns the complement strand. The code is below:

```
class Util():

    # Function that returns the inverted repeat
     of a sequence
    def getInvertedRepeat(self,string):
        return self.getComplementStrand(string
    [::-1])

    # Function that returns the complement of a
     given strand
    def getComplementStrand(self,bases):
        reverseBase = []
        for base in bases:
            if base == 'A':
                reverseBase.append('T')
            elif base == 'T':
                reverseBase.append('A')
            elif base == 'C':
                reverseBase.append('G')
            elif base == 'G':
                reverseBase.append('C')
            else:
                reverseBase.append(base.upper()
    )
        return reverseBase

    '''
    A nice function that returns the reverse
    strand
    '''
    def getReverseStrand(bases):
    reverseBase = []
    for base in bases:
        if base == 'A':
            reverseBase.append('T')
        if base == 'T':
            reverseBase.append('A')
        if base == 'C':
            reverseBase.append('G')
        if base == 'G':
            reverseBase.append('C')
    return reverseBase[::-1]
```

Listing 3: Util Class

2.2.4 RepeatFinder Class Overview

This class is used for all the heavy lifting of finding all of the IR and DR in the sequence. It takes in the sequence that was generated from the sequenceAnalysis.py and then finds all the possible sub-sequences and from

those sub-sequences it finds all of IR and DR. Here is the overview of the class:

```
class RepeatFinder():

    def __init__(self, seq, header,revSeq=False
    ):
        # FASTA Sequence
        self.seq = seq

        # Reverse Sequnce
        self.revSeq = revSeq

        # Header of FASTA
        self.header = header

        # All the sub-sequences
        self.subStrings = []

        # An instance of utility class
        self.util = Util()


    # Function that creates all of the possible
     contiguous sub-sequences
    def getAllSubstrings(self,string):

    # Function that finds all the Direct Repeat
     sequences and prints to the STDOUT
    def findAllRepeats(self):

    # Function that finds all the Inverted
    Repeat sequences and prints to the STDOUT
    def findAllInversionRepeats(self):

    # Function that builds a sequence from the
    original sequence based on the start and
    end positions
    def buildSequence(self,startRepeat,
    endRepeat, startPos, endPos):

    # prints all of the transposons
    def printTransposons(self):
```

Listing 4: RepeatFinder Class

2.2.5 RepeatFinder.getAllSubstrings(self,string)

This function is the one that is the most computationally complex. It loops through every element in a string and computes all possible combinations from that position to the end of the string, which runs at $O(n^2)$ In an attempt to optimize this algorithm, I added a line, (6), to skip all sequences smaller than 1. After running a few test cases, it appears that this algorithm is the bottleneck of the program, under larger files. In the future, I hope to optimize this algorithm further. The code is below:

```
    def getAllSubstrings(self,string):
      length = len(string)
      alist = []

      # loop through every element in the
    string
      for i in range(length):

        # loop through every possible combination
     of the current index to the end of the
    string
        for j in range(i,length):
          if len(string[i:j]) >7  or len(string
    [i:j]) <1 : continue
          newString = string[i:j + 1]
          if newString not in alist:
            alist.append((newString, i, j+1) )
      self.subStrings = alist
      return alist
```

Listing 5: getAllSubstrings(self)

2.2.6 RepeatFinder.findAllRepeats()

This function is one that loops through all of the possible sub-sequences and tries to find all the Direct Repeats in a sequence. Here is some psuedo-code to help you understand. Loop through all of the sub-sequences, if it has not been seen before, add it to a list of seen sequences. If we encounter a sub-sequence that has been seen before, than it is a repeat! . Then we print the sequence from the first repeat to the second repeat. Here is the actual code:

```
    def findAllRepeats(self):
        seenRepeats = []
        for subString, startPos, endPos in self
    .subStrings:
            foundRepeat = ([item for item in
    seenRepeats if item[0] == subString])
            if not foundRepeat:
              seenRepeats.append((subString,
    startPos,endPos))
            else:
                print('    ', self.
    buildSequence((foundRepeat[0])[0],
    subString,(foundRepeat[0])[1],endPos))
```

Listing 6: findAllRepeats(self)

2.2.7 RepeatFinder.findAllInvertedRepeats()

This Function follows the same structure as the previous one, with only one addition. Instead of checking if we encounter the same sequence again, we check if we encounter the Inverted Repeat of the original sequence. Here is the code:

```
    def findAllInversionRepeats(self):
        seenRepeats = []
        for subString, startPos, endPos in self
    .subStrings:
            invertedRepeat = self.util.
    getInvertedRepeat(subString)
            if subString == invertedRepeat:
    continue
            foundRepeat = ([item for item in
    seenRepeats if item[0] == invertedRepeat])

            if foundRepeat:
```

```
                print('    ', self.
    buildSequence((foundRepeat[0])[0],
    subString,(foundRepeat[0])[1],endPos))
            else:
                seenRepeats.append((subString,
    startPos,endPos))
```

Listing 7: findAllInvertedRepeats(self)

2.2.8 RepeatFinder.buildSequence()

This function takes in the positions of the start and end of the new sequence, and the start and end repeats of the sequence. It then splices the original sequence according to the parameters, and returns it to the caller. Here is the code:

```
    def buildSequence(self,startRepeat,
    endRepeat, startPos, endPos):

        startStrting = ''.join(startRepeat)
        endString = ''.join(endRepeat)

        innerSequence = ''.join((self.seq[
    startPos+len(startRepeat):endPos-len(
    endRepeat)]))

        return '{} {} {}'.format(startStrting,
    innerSequence,endString)
```

Listing 8: buildSequnce(self)

3 Discussion

After implementing all the functions and the classes defined in the methods, I went ahead and ran the program on multiple FASTA files. One test file, and one encoding the AJ009736 locus on the Drosophila Melanogaster genome [4]. The program returned all the IRs and DRs found in all the sequences.

On the test file, the program ran instantly and outputted every IR and DR for all the sequences. A sample run can be seen at figure[3]. The output is in the following format:

- [sequence separated by spaces]
- 'All Direct Repeats'
- [DR] [sequence] [DR]
- 'All Inverted Repeats'
- [IR] [sequence] [IR]

All of the output looked good, and was in good format. However, there were a few things that could have been done better. For example when running the Drosiphila Melanogaster, AJ009736 FASTA file, the speed was much slower. The output completed in 5 mins, as opposed to instantly before. I

```
A A T T A C A T C A T A A
Direct Repeats
     AT TAC AT
     CA T CA
     CAT  CAT
     AT TACATC AT
     TA CATCA TA
     AA TTACATCAT AA
Inverted Repeats
     AAT  ATT
     AA  TT
     TTA CATCA TAA
```

Figure 3: This is the sample output of the sequnce: AATTACATCATAA

suppose this is due to the slow nature of the algorithm, $RepeatFinder.getAllSubstrings(self, string)$, which runs at $O(n^2)$. In addition to the slow nature of the program, I would maybe think about also outputting the positions of the sequences as well as printing them. I didn't do so in this version, because I wanted to keep the formatting looking nice.

4 Conclusion

Finding transposons in genomes can be very useful. From genetic fingerprinting to studying new mutations, finding all the transposons in a genome can really help researchers. Doing so by hand is daunting, but when utilizing the power of computation with the power of Python, a solution emerges. findRepeats.py , allows researchers to input any number of sequences and immediately find out all the possible transposons, reducing the number of hours that would have been spent doing so by hand. Moving forward, it would be helpful to optimize the findAllSubstrings code, in order to process larger data sets faster. Another approach could be a multi-threaded approach, to increase speed. Overall, this program is a good start in solving the Transposon Finding problem.

References

[1] Masato N. "The Evidence For Evolution". In: *Nat'l Acad. of Sciences* 98.13 (2001), pp. 7384–7389.

[2] *Transposons: Mobile DNA*. URL: `http://users.rcn.com/jkimball.ma.ultranet/BiologyPages/T/Transposons.html`.

[3] *Words to Time Calculator*. URL: `http://www.edgestudio.com/production/words-to-time-calculator`.

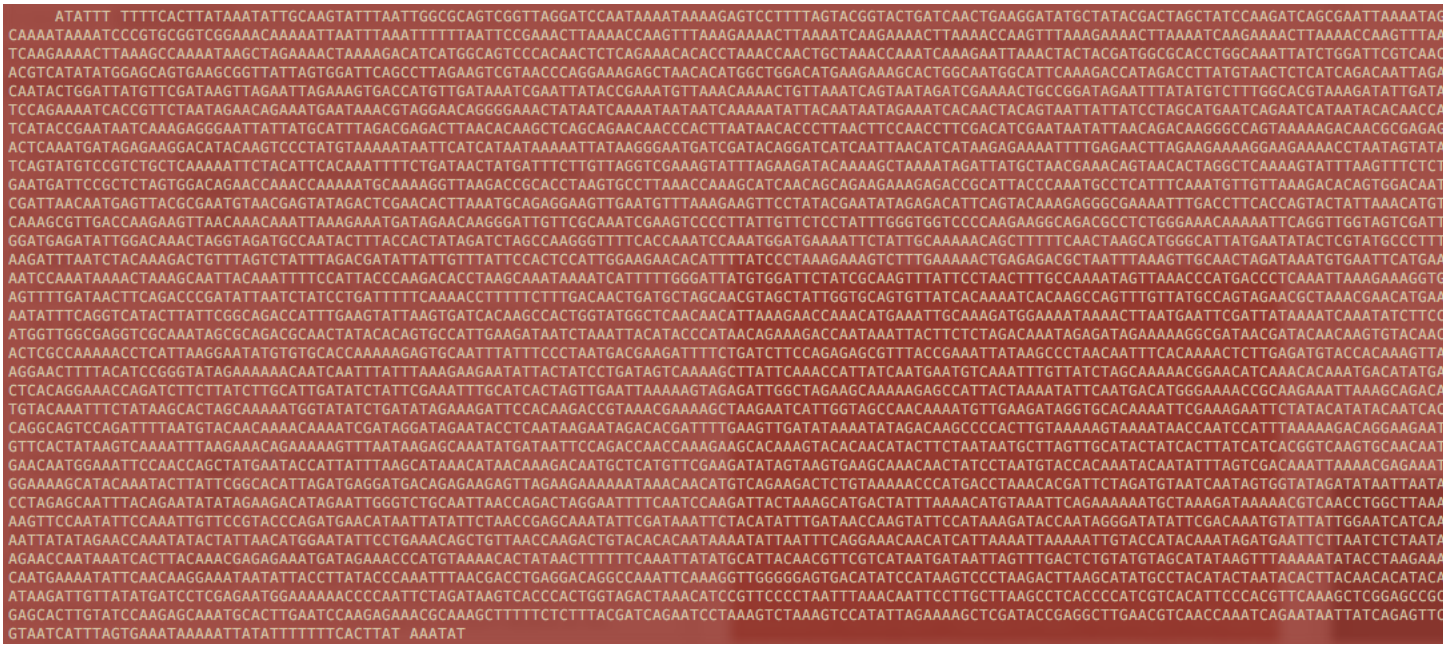

Figure 4: This is one of the outputs of the Drosophila IR Sequence, Transposon