University of Colorado
Anschutz Medical Campus

CPBS 7712: Methods/Tools Biomed Informatics Course

Module 3 Assignment

Final Report

DNA Reads Assembly Algorithm

By:

Nourah Salem

April 12th , 2022

Table of Contents

Motivation

The genetic material (the DNA) that creates all the phenotypic characteristics in all possible creatures not only humans. can be seen as a language made of 4 letters (A,T,C,G) called nucleotides. DNA sequence information is important to scientists investigating the functions of genes to help them know about for example: proteins function or disease causes. DNA sequencing is the process where the scientists fragment the DNA into smaller pieces and examine them gene by gene to find the associations between modifications within the gene and potential phenotypic changes.

Genome assembly refers to the process of putting small nucleotide sequence (reads) into the correct order. Assembly is required, as the sequence read lengths are much shorter than most genomes or even most genes. therefore, analyzing them requires assembly. And the process of assembly requires reconstructing the larger size of sequence is called contig into the right order as the information of the relative ordering among the sequenced reads could get lost otherwise. Therefore, end-to-end sequence overlap is an essential approach to preserve the order of the reads that were sequenced.

Reads as simply into contig is a very vital process for the correct retrieval of sequences and analyzing them and this is the problem of our of course this semester and the following report sections will describe the approach for analyzing and solving the problem.

Computational Problem Formulation

The computational view of the problem can be simplified as follows:

Givens/ **inputs**: 1. A list of strings (each string consists of the letters: A,G,T,C)

2. A query; which is specific longer string made up of certain smaller ones that are found in the list (givens: 1) (also consists of the same 4 letter: ATCG)

Examples of input 1:

TTCAGGCT

CTATGCATTAGAAATG

AAATGTGGC

GGGTGG

- The green boxes refer to overlapping areas between the sequences
- The yellow highlight refers to the span of query

Example of input 2:

CTATGCATTAGAAATGTGGC

Problem:

We are required to build the longest sequence possible from the input list of strings (given 1). Each candidate long string must contain the query (given 2), putting into consideration that we do not have a reference (longest sequence carrying the query) that we can compare our results to.

Herein, the **output** should look like:

TTCAGGCTATGCATTAGAAATGTGGC

Specific Approach

The problem can be applied on the medical field when we have a list of DNA small fragments (reeds) and we need to assemble them to make the longest possible sequence called (contig). The following image simplifies the process:

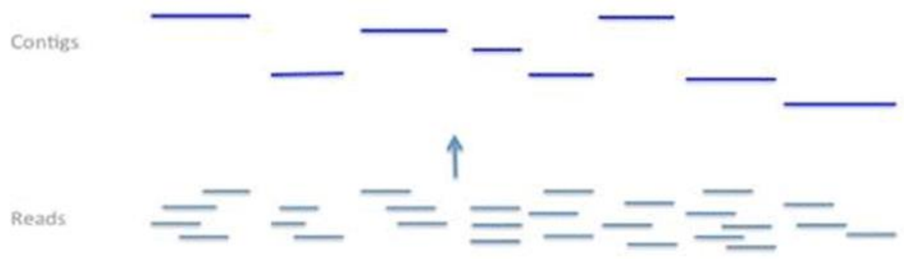
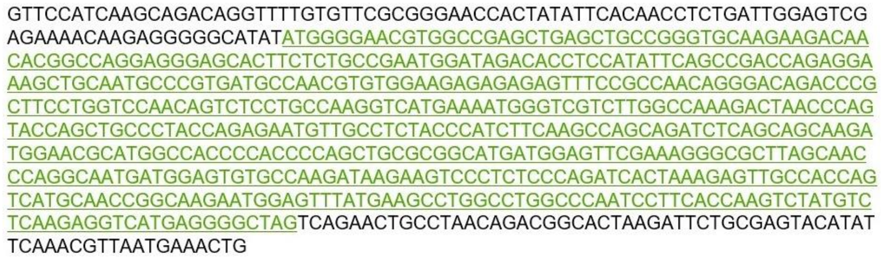


Figure 1 Having a group of reads producing through DNA sequencing, we need to gather them back in one longest contig [1].

The final constructed contig must carry a specific query (could be a specific gene that we are interested in finding), the final output would be similar to the following image:



```
GTTCCATCAAGCAGACAGGTTTTGTGTTCGCGGGAACCACTATATTCACAACCTCTGATTGGAGTCG
AGAAAACAAGAGGGGGCATATATGGGGAACGTGGCCGAGCTGAGCTGCCGGGTGCAAGAAGACAA
CACGGCCAGGAGGGAGCACTTCTCTGCCGAATGGATAGACACCTCCATATTCAGCCGACCAGAGGA
AAGCTGCAATGCCCGTGATGCCAACGTGTGGAAGAGAGAGAGTTTCCGCCAACAGGGACAGACCCG
CTTCCTGGTCCAACAGTCTCCTGCCAAGGTCATGAAAATGGGTCGTCTTGGCCAAAGACTAACCCAG
TACCAGCTGCCCTACCAGAGAATGTTGCCTCTACCCATCTTCAAGCCAGCAGATCTCAGCAGCAAGA
TGGAACGCATGGCCACCCCACCCCAGCTGCGCGGCATGATGGAGTTCGAAAGGGCGCTTAGCAAC
CCAGGCAATGATGGAGTGTGCCAAGATAAGAAGTCCCTCTCCCAGATCACTAAAGAGTTGCCACCAG
TCATGCAACCGGCAAGAATGGAGTTTATGAAGCCTGGCCTGGCCCAATCCTTCACCAAGTCTATGTC
TCAAGAGGTCATGAGGGGCTAGTCAGAACTGCCTAACAGACGGCACTAAGATTCTGCGAGTACATAT
TCAAACGTTAATGAAACTG
```

Figure 2 After assembling all the reads into one long sequence. this sequence holds a specific query inside (in green).

Assumptions

Given the problem, we need to add constrains to be able to solve computationally and efficiently:

1. We are not given the ground truth.
   We are required to build the longest possible contig, but we don't know the exact length of it or what are the exact constructing reeds. In biology this is called de Novo assembly. Therefore, we might make more than one candidate contigs that carry our query. The result depends on how efficiently we built our model the unit tests.
2. The reeds are overlapping
   In order to make one by one assembly of reads, there must be an overlapping area between the beginning of the first read and the end of the following read. This allows us to loop over the reeds and find matching and ladder building once so that we can reach the maximum length of the ladder. If the reads are not overlapping, then it will be hard to find the appropriate continuous sequence, we will need more input information.
3. Genes in the DNA have a directionality nature. One gene can be moving from 5' to 3' prime and another gene could be moving on the opposite direction. In our case we don't have specific directions for the reeds, so it's possible that the read could be viewed/match for either of the 2 directions. This means that during searching for the reeds ladder, we can test the alignment of each read and the reverse of it before jumping to the next one in the list.

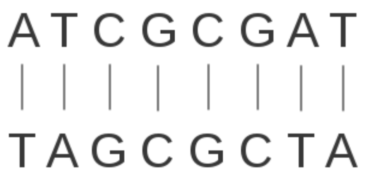


Figure 3 Example should the 2 possible that a read can take to be tested for its alignment with the other reads [2].

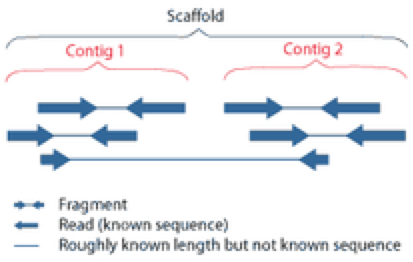


Figure 4 In nature, reads/genes are read from specific direction (because the DNA consists of 2 sequences, and each carries a specific direction) [3].

4. As we are aligning certain number of letters from start of one read to the end of the following read, for example let's say we are aligning 3 letters from each read, it is probable to find more than one candidate reads that can align. It could also happen that one of them matches with more than 3 letters, therefore, we believe that all of these candidates should be tested, but we will select those that make the longest matches first.

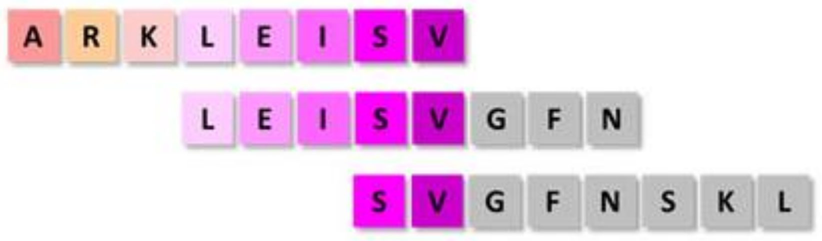


Figure 5 There are 2 candidate sequences that can match the colored read, one making a 2 letter match and the other is making a 5 letter match, we will consider both candidates (in separate trials), but we will pick the ones with the highest matches first[4].

Specific Implementation of the Approach

Link to GitHub

As we described in module 1 presentation, instead of constructing the longest contig, and then search them for our query (those that have the query inside will be the candidates for the solution), we will construct a copy of the query first (from the given list of reads) and then, extend the flanking regions of the query with overlapping reeds to reach the longest possible contig. This assures that every time we end up with a contig, the query is always inside. we do not have to search them back.

Accordingly, the diagram from module 1 presentation simplifies the two main process to reach our goal.

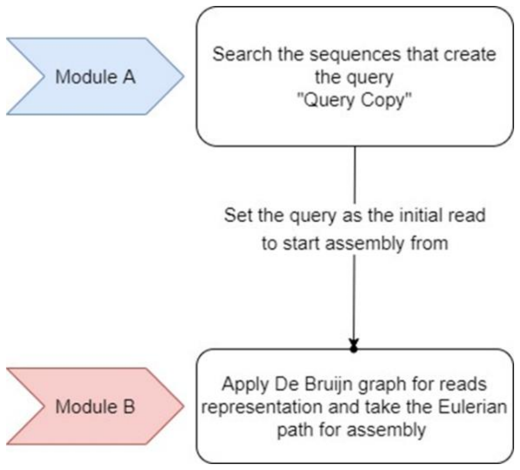

Module A
Search the sequences that create the query "Query Copy"
Set the query as the initial read to start assembly from
Module B
Apply De Bruijn graph for reads representation and take the Eulerian path for assembly

Figure 6 The 2 main modules for building the longest contig with the query.

This report includes the details for module A implementation:

1. Data Preparation:

- Before proceeding onto the query construction, we made simple preprocessing of the reads and explanatory data analysis. The pipeline on the right side of the page summarizes the main functions that we used for this portion of analysis.
- We split the reads FASTA file IDs from the reads into a 2 column pandas data frame. This will help us to identify the column, not only by its index but also, we have its ID on the same row.
- In addition, we counted the length of each read and concatenated the column resultant to the data frame.
- After that we looked at the distribution of the different lengths of all the reads. The result section includes a plot of it.
- After looking at the sample EDA of the data, we made a small sample of both: the reads and the query, to be able to further work on our scripts with them.
- For the reads sample: We only got the first 100 reads from our prepared data frame.
- For the sample query construction: we selected random reads to modify their starting and ending letters to allow for their overlap, the selected reads' indices are: [2, 25, 49, 76,99]. We flipped the read number 76 to test this point later when searching the query copy.
- We lastly saved our prepared data on CSV file.

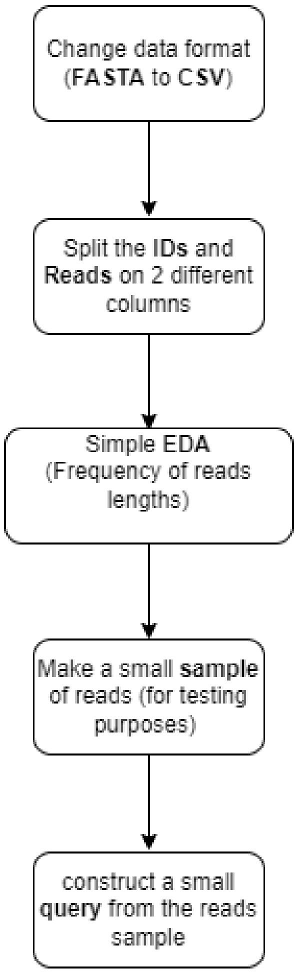

Change data format (**FASTA** to **CSV**)
Split the **IDs** and **Reads** on 2 different columns
Simple **EDA** (Frequency of reads lengths)
Make a small **sample** of reads (for testing purposes)
construct a small **query** from the reads sample

Figure 7 Pipeline for date preprocessing and EDA.

2. Query Copy Construction:

- Given a query and the list of reads that this query could be made of some of it, we worked on reconstructing the query in what we called the “query copy” (figure on the right).
- For this, the first step should be to match between the start of the query and the end of every read we have. If there is a match between the letters of the start and the end of both of sequences, then the read could be the first step of the ladder to make the query.
- For this we thought about using the longest common substring (LCS) algorithm for finding the alignment between the query and the read the sort of code for it is provided in the next section.
- The LCS algorithm builds a matrix of size (query X read). And search for match letter by letter, if there is not match, it adds 0 to the common cell, if there is match, it adds +1. The following table shows an example of our implementation:

		G	G	T	C	A	A	A	T	T
	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	1	1	1	0	0
A	0	0	0	0	0	1	2	2	0	0
A	0	0	0	0	0	1	2	3	0	0
T	0	0	0	1	0	0	0	0	4	1
T	0	0	0	1	0	0	0	0	1	5
T	0	0	0	1	0	0	0	0	1	2
C	0	0	0	0	2	0	0	0	0	0
C	0	0	0	0	1	0	0	0	0	0

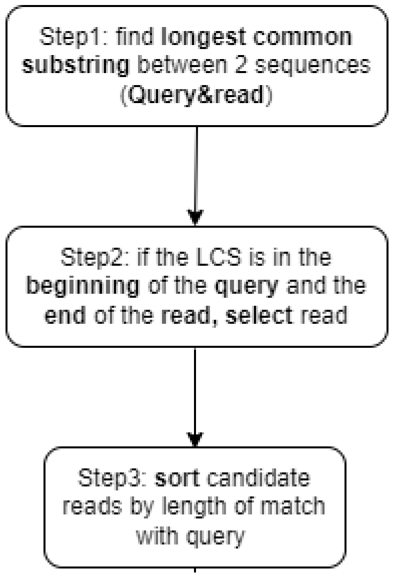

Step1: find longest common substring between 2 sequences (Query&read)
Step2: if the LCS is in the beginning of the query and the end of the read, select read
Step3: sort candidate reads by length of match with query

Figure 8 Pipeline for query copy building.

- This can also be visualized like this:
  Q: AAATTTCCCTTTGGGTAGTGTT
  R: GGTCAAATT
  (Where Q refers to the query and R to the current read)
- We retrieve the highest match value in the matrix and trace back the matching sequence. in addition to that, we put on if condition to check if this match is within the start of the query and at the end of the read so that we make sure that it's in the right position not an internal match between the two sequences.
- We collect all the reads that satisfy the previous conditions in one list and sort them by the length of match with the query.
- At this step, we select the read with the highest match.

Pseudocode

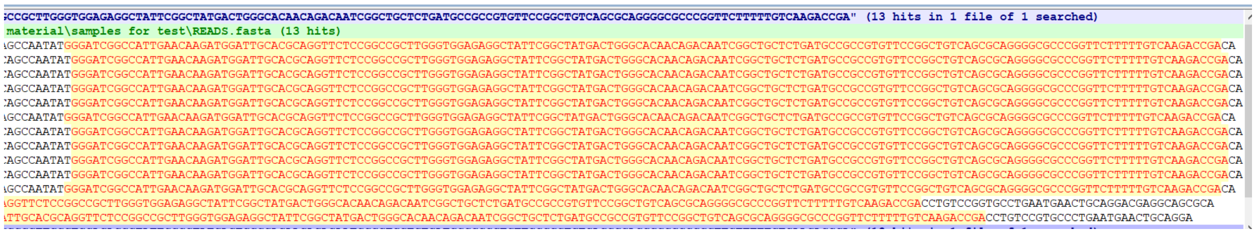
Simple pseudocode for LCS:

```
Input: String X,Y

Output: Longest Common Substring

M = length of string S

N = length of string T

L = 2D array(0…M, 0…N):

result = 0

Mtch_seq = {}
```

```
For i = S(1) … m do
  For j = T(1) … n do
    If S(i) = T(j) then
        L[i, j] = +1
    Else:
         L[I,j] = 0
    If L[I,j] > result then
        result = L[I,j]
        Mtch_seq = {}
```

Working on the whole dataset

After examining the implementation of the longest common substring algorithm on our whole data set which consists of 124520 reads, we noticed a condition that breaks the assumption of overlapping the start of 1 read with the end of another as we can see in the following example:



```
CCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGA" (13 hits in 1 file of 1 searched)
 material\samples for test\READS.fasta (13 hits)
AGCCAATATGGGATCGGCCATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACA
CAGCCAATATGGGATCGGCCATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACA
CAGCCAATATGGGATCGGCCATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACA
CAGCCAATATGGGATCGGCCATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACA
CAGCCAATATGGGATCGGCCATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACA
AGCCAATATGGGATCGGCCATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACA
CAGCCAATATGGGATCGGCCATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACA
CAGCCAATATGGGATCGGCCATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACA
CAGCCAATATGGGATCGGCCATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACA
CAGCCAATATGGGATCGGCCATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACA
AGCCAATATGGGATCGGCCATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACA
AGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACCTGTCCGGTGCCTGAATGAACTGCAGGACGAGGCAGCGCA
ATTGCACGCAGGTTCTCCGGCCGCTTGGGTGGAGAGGCTATTCGGCTATGACTGGGCACAACAGACAATCGGCTGCTCTGATGCCGCCGTGTTCCGGCTGTCAGCGCAGGGGCGCCCGGTTCTTTTTGTCAAGACCGACCTGTCCGTGCCCTGAATGAACTGCAGGA
```

Figure 9 Alignment of portion of the query to see if the building reads can overlap with each other from their ends or not.

The highlighted characters in yellow represents the start of a query to a certain length. We noticed that his portion of the query is inside reads. It can only be found within those reads, and not extending to one of their ends of the reads. This means that we cannot overlap another read above these reeds to make longer contig because this will include external characters (could be in this example the “CA” on the right) that are not in the query, and if we want to search later for the query, it will not be possible to find them.

Since we noticed this on our dataset, we decided to not limit our overlapping between reads and query to the ends of them only, we allowed four all possible overlaps between the query and all reads we have. for that we made an initial trial of 10,000 reeds and looked at the range of characters (match length) that match to the query. from that, we had an initial assumption about the legit alignments versus those that can happen at random the following graph shows a sample of 10,000 reads aligned to the query.

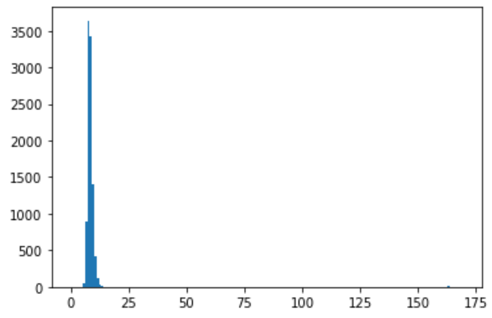


Figure 10 Alignment of 10000 reads selected at random with the query.

This gave us an initial view that the longest alignment of the 10000 reads, where the longest alignment in this case was of length 175 bases with the query and this spike in the graph tells us that the majority of the alignments were too short that they might be seen as random matches. Their lengths were from 7 character match to 13 characters.

We next made our run on the whole data set allowing each read to align with the query (if there is characters to match) and the graphs at the results section show the ranges of the character match lengths of all the reads we have on our data set. By this approach we should guarantee that we have implemented the longest contig building algorithm that spans the whole query. However, there are two points that we need to further investigate:

1. what is the minimum number of reads that makes my query and at the same time are our long enough to make the longest query-spanning contig?
2. How can we validate that all the reads we have now in my candidate list are actually spanning the query and not missing one character at least?

For the first point, as we have 124520 reads aligned to the query of course, we can say that the query is inside all of them. However, we cannot consider the whole data set to be a query building contig, because as we saw, the majority of them can make random (relatively short) alignment with the query, in addition, this would be very far from being accurate to determine the query-building reads.

In order to solve this point, we decided to make a sequence of trials to look at which lengths of alignment between the reads and the query are sufficient to give out a number of reads that can for build the whole query and at the same time be the longest contig spanning this query. We first excluded the reads that are shorter than 13 bases aligned with the query, then we tested the rest if they can spend the whole query or there will be missing character not covered by the current candidate reads list. we then increased the number to 45 90, 130 and 140. at each increment we tested the remaining bundle of candidate treats can spend the whole query or not so that we can make a threshold for the number of reeds that can spend the whole query as well as the lengths of the alignment. the table for the threshold results is available in the results section.

Next is the validation step, assuming that we have 10 reads that we believe they are the reads that can make the longest query-spanning contig. how can we make sure that the whole query is covered by these 10 reeds only? We thought about using the logical operation (OR) on a boolean representation of the characters of the query and the reads the following example simplifies the approach we've taken:

Let's assume that the following list of characters represent our query:

[G G G A T C G G C C A T T G

We have represented each character here by a Boolean (initiated as false) at the beginning of this Boolean list. Each false here if there's two each character in the query that is not covered by any match of any of the reeds yet.

[False, False, False, False, False, False, False, False, False, False, False, False, False, False, False]

Here we made other 10 lists of the same length of the query to represent each read we have in this example. However, we turned some of the false values into true according to the span of the reeds that they made with the query:

1. [True, True, True, False, False, False, False, False, False, False, False, False, False, False, False]

2. [Flase, Flase, Flase, True, True, True, True, True, True, True, True, True, True, False, False, False]

3. [Flase, Flase, Flase, False, False, False, False, False, False, False, False, False, True, True, True]

Of the three lists we have, each represent the characters of the reads on the query scale. We can see that the first list shows only three true elements. this means that this read was only able to cover three characters of the query (3 characters match). The second list shows 10 true characters, referring to 10 character alignments in the middle of the read with the middle of the query and so on. The OR logical operator compares two values at a time and output the updated result in the query list. One element from the query is compared to the corresponding one from the read, if the element value in the query is false and the element from the read is also false, the query list is updated as false in the same current position. If either of the elements in any of the lists (either the query or the read) is true, the current position of the query is updated to true. We are only sure that the reads we have had already covered the query when we get the final Boolean list of the query filled with true only.

Therefore, after every threshold of the number of character match and number of reads we have selected, we made the wrong on the query coverage Boolean list. We were successfully able to cover the whole query characters with the candidate reads! The results section shows the True Boolean list after a threshold of 130 base match.

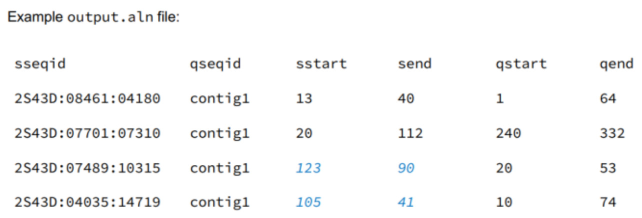
Unit tests

- We have made a couple of files that test the two main modules of our script getting more about how the input is inserted to the data set we cared that the inputs should be:
    - The two sequences must be of type string
    - Matches are only between 4 characters ATCG
    - The four characters are upper case only
    - None of the input is an empty string of length 0
    - For the validation step, we checked that all the lists were in the Boolean (true, false) type.

Limitations

For the query copy script:

- It's not an actual limitation of the scripts we made to achieve our goal however, we modified our view of the problem from building a query copy and then elongating this copy sequence to make the longest possible contig, we decided to limit the length of the longest contig -marginally- to the length of the query. The longest contig is not strictly limited to the 648 bases of the query, it's longer because we used the reads that made the longest possible match in the query, however, after building the query, we did not extend the ends of the query copy with additional reads that can align with those query-copy ends, and at the same moment, not found in the query itself. this is because we checked back at the file output format that we are required to submit, and each read we saw in the example showed alignment with the query only.


Example output.aln file:

```
sseqid                qseqid     sstart     send      qstart     qend
2S43D:08461:04180     contig1    13         40        1          64
2S43D:07701:07310     contig1    20         112       240        332
2S43D:07489:10315     contig1    123        90        20         53
2S43D:04035:14719     contig1    105        41        10         74
```

Figure 11 output file 2 format, considering only the reads that align with the query to be the candidates for making the longest contig as well.

- Another point is that: we first put an assumption that we may flip the reads and realign them with the query if the original orientation does not cover the query at the end, however, we were able to generate four contigs that can all cover the whole query without using the flipping approach. Therefore, we did not flip all the reads, realign them with the query and then filter them for the candidate ones.

Results and Discussion

- Our reads list consists of 124520 reads. The longest read size is 346 and the shortest is 30 letters.
- The following figure shows the distribution of the lengths of the reads:

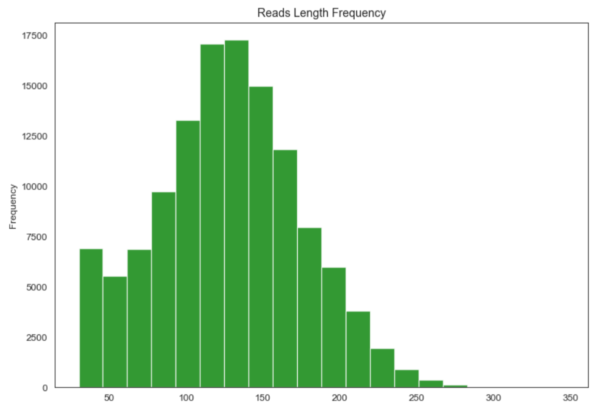


Figure 12 Distribution of all the read lengths.

The graph shows the whole spectrum of lengths of the reads, where we can see that most of the reads are between 100 and 150 letters. This gives us a good insight when we look at the minimum and maximum match size. For example, we now do not need smaller matches like 3 or 5 letters to cut off the massive duplicates of none contributing matches. we will be testing from sizes from 15 letters match or more.

- We aligned the 124520 reads to the query. If one read was able to make more than one alignment with the query, we selected the longest alignment in the matrix. After looking at the longest common substring in all of them, we plotted a histogram of the lengths of the alignments of the whole dataset:

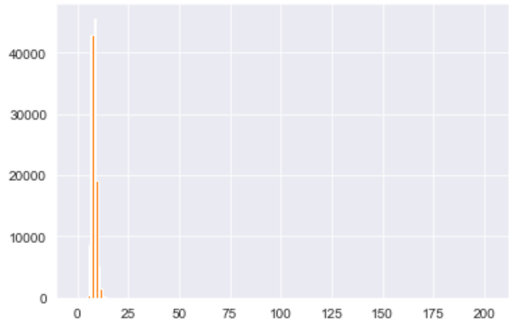
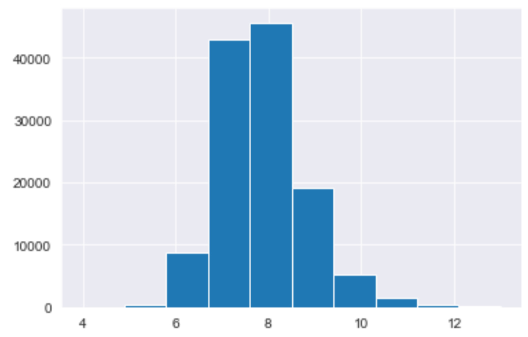
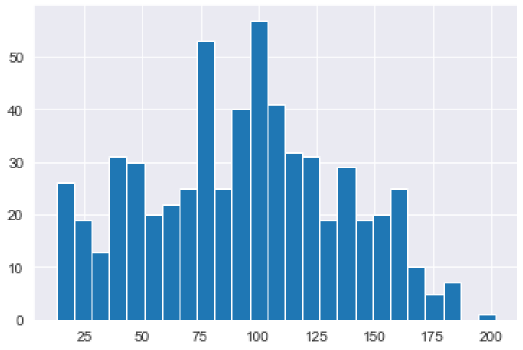


Figure 13 character match lengths between each read and the query, the majority of the reads made alignments with the query with short lengths.

- We can see here that the majority of the reads more (than 40,000 reads) made alignments that range between 4 to 20 bases with the query. The number of these alignments is very high in a way that the rest of the reads alignment lengths is not seen, so we have split the graph into 2 one with alignments of length from zero to 13 base match, and one from 13 character match 12 the longest match between a read and the query:

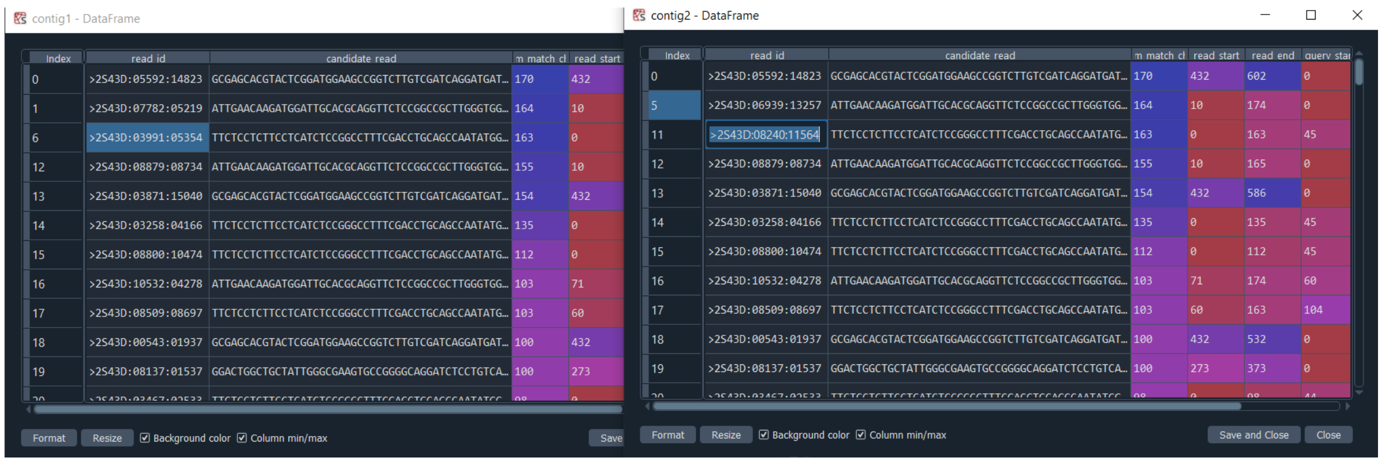




- This way we can view the distribution of the lenses of alignment of reads with the query and we can see that the majority are made at random having very short alignment with the query. However, the distribution around 100 can tell that those reads could be partly the ones that have significant common parts with the query. They are actually the building blocks of the query.
- After running over all the reads, we wanted to decide the threshold of the character match number, At the same moment, we want the number of reads the resultant from this threshold to be covering the query. The following table shows our trials of cutting the number of reads that have lower number of matches and then testing if the rest of the reads in the bundle can cover the query or not.

Length of Alignment Threshold (bases)	Number of reads per contig	Do they cover the whole query range?
> 13	153	True
> 45	127	True
> 90	82	True
> 130	43	True
> 140	24	False

- we noticed that when we cut the number of reads according to the length of 0:140 character match, the remaining reads cannot cover the query. Herein, we set our threshold to 130, as this gives out 43 reads that covers the whole query and makes our longest contig at the same time without missing characters of the query.
- We were also able to make variants of the contig generated, as there are reads that are very similar to each other, only the difference is 1 character. both cover the query with the same span size and within the same area as in the following example:



contig1 - DataFrame

Index	read id	candidate read	m match cl	read start
0	>2S43D:05592:14823	GCGAGCACGTACTCGGATGGAAGCCGGTCTTGTCGATCAGGATGAT...	170	432
1	>2S43D:07782:05219	ATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGG...	164	10
6	>2S43D:03991:05354	TTCTCCTCTTCCTCATCTCCGGCCTTTCGACCTGCAGCCAATATGG...	163	0
12	>2S43D:08879:08734	ATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGG...	155	10
13	>2S43D:03871:15040	GCGAGCACGTACTCGGATGGAAGCCGGTCTTGTCGATCAGGATGAT...	154	432
14	>2S43D:03258:04166	TTCTCCTCTTCCTCATCTCCGGGCCTTTCGACCTGCAGCCAATATG...	135	0
15	>2S43D:08800:10474	TTCTCCTCTTCCTCATCTCCGGGCCTTTCGACCTGCAGCCAATATG...	112	0
16	>2S43D:10532:04278	ATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGG...	103	71
17	>2S43D:08509:08697	TTCTCCTCTTCCTCATCTCCGGGCCTTTCGACCTGCAGCCAATATG...	103	60
18	>2S43D:00543:01937	GCGAGCACGTACTCGGATGGAAGCCGGTCTTGTCGATCAGGATGAT...	100	432
19	>2S43D:08137:01537	GGACTGGCTGCTATTGGGCGAAGTGCCGGGGCAGGATCTCCTGTCA...	100	273

contig2 - DataFrame

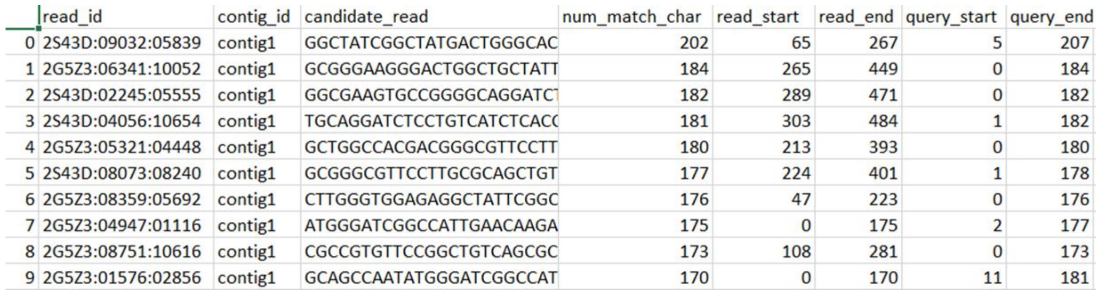
Index	read id	candidate read	m match cl	read start	read end	query star
0	>2S43D:05592:14823	GCGAGCACGTACTCGGATGGAAGCCGGTCTTGTCGATCAGGATGAT...	170	432	602	0
5	>2S43D:06939:13257	ATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGG...	164	10	174	0
11	>2S43D:08240:11564	TTCTCCTCTTCCTCATCTCCGGGCCTTTCGACCTGCAGCCAATATG...	163	0	163	45
12	>2S43D:08879:08734	ATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGG...	155	10	165	0
13	>2S43D:03871:15040	GCGAGCACGTACTCGGATGGAAGCCGGTCTTGTCGATCAGGATGAT...	154	432	586	0
14	>2S43D:03258:04166	TTCTCCTCTTCCTCATCTCCGGGCCTTTCGACCTGCAGCCAATATG...	135	0	135	45
15	>2S43D:08800:10474	TTCTCCTCTTCCTCATCTCCGGGCCTTTCGACCTGCAGCCAATATG...	112	0	112	45
16	>2S43D:10532:04278	ATTGAACAAGATGGATTGCACGCAGGTTCTCCGGCCGCTTGGGTGG...	103	71	174	60
17	>2S43D:08509:08697	TTCTCCTCTTCCTCATCTCCGGGCCTTTCGACCTGCAGCCAATATG...	103	60	163	104
18	>2S43D:00543:01937	GCGAGCACGTACTCGGATGGAAGCCGGTCTTGTCGATCAGGATGAT...	100	432	532	0
19	>2S43D:08137:01537	GGACTGGCTGCTATTGGGCGAAGTGCCGGGGCAGGATCTCCTGTCA...	100	273	373	0

Figure 14 two contig lists where we have a couple of different reads (different IDs), but they span the query within the same area and with the same length.

Read1 (alignment length with query: 163 bases): ------TTCCTCATCTCCGGCCTTTCGACCTGC------

Read2 (alignment length with query: 163 bases): ------TTCCTCATCTCCGGGCCTTTCGACCTGC----

- This allowed us to generate 4 contigs. A couple of them are similar to each other with a whole contig lengths: 7180 bases while the other 2 have the lengths of 7171. The output files have all the information about them (ALLELES.fasta).
- We also generated the ALLELES.aln file that has describing alignment of sequence reads to contig(s) in ALLELES.fasta the with the required columns. We added 2 columns, the reads themselves and the length of the match they can make with the query.



	read_id	contig_id	candidate_read	num_match_char	read_start	read_end	query_start	query_end
0	2S43D:09032:05839	contig1	GGCTATCGGCTATGACTGGGCAC	202	65	267	5	207
1	2G5Z3:06341:10052	contig1	GCGGGAAGGGACTGGCTGCTATT	184	265	449	0	184
2	2S43D:02245:05555	contig1	GGCGAAGTGCCGGGGCAGGATC	182	289	471	0	182
3	2S43D:04056:10654	contig1	TGCAGGATCTCCTGTCATCTCAC	181	303	484	1	182
4	2G5Z3:05321:04448	contig1	GCTGGCCACGACGGGCGTTCCTT	180	213	393	0	180
5	2S43D:08073:08240	contig1	GCGGGCGTTCCTTGCGCAGCTGT	177	224	401	1	178
6	2G5Z3:08359:05692	contig1	CTTGGGTGGAGAGGCTATTCGGC	176	47	223	0	176
7	2G5Z3:04947:01116	contig1	ATGGGATCGGCCATTGAACAAGA	175	0	175	2	177
8	2G5Z3:08751:10616	contig1	CGCCGTGTTCCGGCTGTCAGCGC	173	108	281	0	173
9	2G5Z3:01576:02856	contig1	GCAGCCAATATGGGATCGGCCAT	170	0	170	11	181

Bibliography

[1]“INSDC standards for genome assembly submission,” www.ddbj.nig.ac.jp. https://www.ddbj.nig.ac.jp/ddbj/assembly-e.html (accessed Mar. 11, 2022).

[2]“SOLVED:Primer design: Given below is a single stranded DNA sequence. Design suitable reverse and forward primers that can be used to amplify the region highlighted here:www.numerade.com. https://www.numerade.com/ask/question/primer-design-given-below-is-a-single-stranded-dna-sequence-design-suitable-reverse-and-forward-primers-that-can-be-used-to-amplify-the-region-highlighted-here_-gttccatcaagcagacaggttttgtgttc-76151/ (accessed Mar. 11, 2022).

[3]J. A. Martin and Z. Wang, “Next-generation transcriptome assembly,” Nature Reviews Genetics, vol. 12, no. 10, pp. 671–682, Oct. 2011, doi: 10.1038/nrg3068.

[4]Wikipedia Contributors, “Contig,” Wikipedia, Oct. 02, 2019. https://en.wikipedia.org/wiki/Contig (accessed Oct. 13, 2019).