Driver coding challenge

Kyle Swanson

October 7, 2016

1 Introduction

In this challenge, we are given a number of segments of DNA, all of which come from the same original sequence of DNA. Our goal is to reconstruct the original DNA sequence by combining segments which overlap by more than half their length.

My solution consists of two main parts. First, I identify the location of the overlaps between the segments of DNA. Then, I use the information about overlaps to reconstruct the original DNA sequence.

2 Finding overlaps

2.1 Overview

In order to reconstruct the DNA sequence, we need to know where our segments overlap. I find overlaps primarily by using a hash table (i.e. a Python dictionary), in which I store subsequences (i.e. substrings of the DNA segments) along with all of the segments which contain that subsequence and the location of that subsequence. Since we only care about overlaps of length greater than half the length of the segment, we only consider subsequences that are at least this size.

2.2 Subsequences at the beginning of the segment

The process occurs as follows. For each segment, we first consider all subsequences of length at least 50% of the length of the segment which start at the beginning of the segment.

For example, if we have the DNA segment

$$CTAA \dots ATTG = a_0a_1a_2a_3 \dots a_{n-4}a_{n-3}a_{n-2}a_{n-1}$$

where a_i is the ith letter of the sequence, then we consider the subsequences

$$a_0a_1 \dots a_{n/2}a_{n/2+1}$$

$$a_0 a_1 \dots a_{n/2+1} a_{n/2+2}$$

$$\vdots$$

$$a_0 a_1 \dots a_{n-2} a_{n-1}$$

We hash each subsequence into our dictionary of subsequences and add a tuple `(segment_id, index)` where `segment_id` is the ID of the segment and `index` is 0 since these subsequences all start at the 0th letter.

2.3 Subsequences at the end of the segment

We do a similar process for all subsequences of length at least 50% of the length of the segment which end at the end of the segment. So we consider the subsequences

$$a_1 a_2 \dots a_{n-2} a_{n-1}$$

$$a_2 a_3 \dots a_{n-2} a_{n-1}$$

$$\vdots$$

$$a_{n/2} a_{n/2+1} \dots a_{n-2} a_{n-1}$$

(We skip $a_0 a_1 \dots a_{n-2} a_{n-1}$ since this was covered already when we considered the subsequences starting at the beginning.)

Again we hash each subsequence into our dictionary and add `(segment_id, index)` tuples. This time `index` will not be 0 but will instead be the index of the first letter in our subsequence.

2.4 Generating the overlap dictionary

When we hash each of these subsequences, we look at all of the other `(segment_id, index)` tuples currently in the dictionary for this subsequence because each of these segments overlaps with this segment at this subsequence.

If the subsequence is the end of this segment, then any other segment which begins with this subsequence (i.e. a tuple with `index` $= 0$) could potentially follow this segment in the original sequence, so we add it to a dictionary (`overlap_dict`) which maps a `segment_id` to its successor segment IDs and the indices at which the overlap occurs.

Similarly, if the subsequence is at the beginning of this segment, then this segment is a successor of any segment which ends with this subsequence. We know that tuples with `index` $\neq 0$ (or with `index` $= 0$ where the subsequence is

the entire segment) end with this subsequence, so we add this segment to the dictionary of successors for each of those segments.

2.5 Example

Let's look at an example. Say we are given the following DNA segments:

$$\{'Frag\_56' :' ATTAGACCTG',$$

$$'Frag\_57' :' CCTGCCGGAA',$$

$$'Frag\_58' :' AGACCTGCCG',$$

$$'Frag\_59' :' GCCGGAATAC'\}$$

We can see that Frag_58 and Frag_57 share the subsequence 'CTGCCG'. In the hash table of subsequences, we would see the two tuples ('Frag_58', 3) and ('Frag_57', 0). Since Frag_57 has the subsequence at index 0, it begins with the subsequence, and since Frag_58 has the subsequence at index 3, it ends with the subsequence. Therefore, Frag_58 follows Frag_57, so we add Frag_58 as a successor to Frag_58. So in our successor dictionary, we get

$$\{'Frag\_58' : \{'Frag\_57' : 3\}\}$$

If we continue this process, we will get the successor dictionary:

$$\{'Frag\_59' : \{\},$$

$$'Frag\_58' : \{'Frag\_57' : 3\},$$

$$'Frag\_57' : \{'Frag\_59' : 3\},$$

$$Frag\_56' : \{'Frag\_58' : 3\}\}$$

2.6 Conclusion

Using the above process, we will get a dictionary which maps segment IDs to successor segments and the indices of the overlaps. Next, we will perform a search through the segments in this dictionary to reconstruct the original DNA sequence.

3 Reconstructing the sequence

3.1 Overview

Now that we know where our segments overlap, we want to use that information to reconstruct the original sequence. We can do this by representing the problem as a graph and using depth first search (DFS) to find a path through the graph.

3.2 Graph representation of the problem

It is easy to convert our DNA problem to a graph problem. Each segment of DNA is a vertex in our graph. If the end of segment i overlaps with the beginning of segment j (with an overlap of at least 50%), then we have an edge from segment i to segment j. Since we want to reconstruct the original sequence from all of the segments, we must find a path in the graph which touches every vertex exactly once.[1]

3.3 Reconstructing the sequence

To reconstruct the sequence, we will perform a DFS through the graph. We don't know which segment to start at, so we have to try a DFS starting from every segment until we find a starting segment that allows us to reconstruct the entire sequence.

For each starting segment, start by adding the the tuple `(segment_id, depth)` to a stack, where `depth` $= 0$. Additionally, initialize a set which will contain the segment IDs in our path and initialize an array which will place the segment IDs in order. We perform the following loop either until the stack is empty (we ran out of paths to try) or until we have found a path through all of the segments.

In each iteration of the loop, we pop a `(segment_id, depth)` tuple off the stack. If the depth is less than the depth of the previous node, then we are backtracking, so we need to delete segment IDs for each of the nodes we are backtracking to from our set and array of segment IDs. Then, add the segment ID of the current node to the set and array. Then, add all of the successors of this segment ID (except for those which are already in our set since they are already in our path) to our stack where the depth of these new nodes is one more than the depth of the current node.

Since we know there is a way to reconstruct the sequence, the above loop will eventually break once it has found the correct ordering of all of the segments. Finally, initialize a string for the reconstructed sequence and we loop through each of the segments in the array of segment IDs, adding the segment up to the point of overlap with the next segment.

[1] This is actually the Hamiltonian Cycle problem (also know as the Traveling Salesman problem), which is NP-hard. Therefore, even the best solution will be exponential in time. Luckily, it is very unlikely that our graph will have many edges since it is unlikely that at least half of very long strands of DNA will match by chance, so in reality it won't take too long to find a path through the graph touching all the vertices.

3.4 Conclusion

The above method will produce a string containing the reconstructed DNA sequence. Finally, we save this reconstructed sequence to a file specified by the user.

4 Running and testing the code

4.1 Running the code

To run the code, simply open a terminal, navigate to the directory with `sequence_assembler.py`, and execute the command

```
python sequence_reconstructor.py <file_with_segments> <file_for_reconstructed_sequence>
```

where `<file_with_segments>` is the file containing DNA segments in FASTA format and `<file_for_reconstructed_sequence` is the file where the reconstructed sequence will be written.

4.2 Testing

To test the sequence reconstructor, execute the command

```
python sequence_reconstructor.py <number_of_segments> <segment_length>
```

where `<number_of_segments>` is the number of DNA segments to be used in the test (must be at least 2) and `<segment_length>` is the length of a DNA segment (must be at least 3).

This will generate a random sequence of DNA with overlaps of at least 50% between adjacent segments of DNA and will write these segments to a file in FASTA format. It will then call the sequence reconstructor on this file and will compare the output of the sequence reconstructor to the original sequence to see if the reconstructor accurately reconstructed the sequence from the DNA segments.