

Student: Yu Cheng (Jade)

ICS 675

Assignment #3

Nov 23, 2009

Assignment 3

Question: In this assignment, we will be assembling 22 Solexa genomic sequences using the OLC (Overlap Layout Consensus) approach. Very close relatives to the organism from which these fragments have been obtained have been abundantly sequenced in the past.

- a.** Using the fags.fasta file, calculate the overlap and use a graph to display the results of the calculation. Use either blast program or the program water (from the EMBOSS suite of tools) to carry this task.

Answer: After comparing blast and water two programs, I decided to use water for this assignment. Water program takes a pair of sequences and returns a best local alignment. Parsing the water output files, I can obtain the information needed to construct the graph. For each relevant pair of sequences, the overlapping length is the edge weight, and the upstream and downstream sequences are the source and destination nodes respectively for that edge.

In order to carry out the task of executing water programmatically, I used a python script (analyze.py) to parse the fags.fasta file, generate all pair-wise combinations of the sequences, execute water as a command line program on each pair of sequences, extract key information from the water output files, filter out irrelevant pairs based on the overlapping lengths and aligning positions. Finally, the script produces a tab delimited standard output containing all selected alignments. Each entry of this output represents an edge in the graph.

In order to carry out the task of plotting the graph programmatically, I used yapgvb, a python graphviz binding library, and wrote a python script (graph.py) to parse the standard input, compute the number of vertices, and plot an edge weighted direct graph in svg format using the layout option 'dot'.

After carrying out the aforementioned procedures and playing with different filtering overlapping percentages, I obtained the following graph, **Figure 1**. More graphs with different cut-off thresholds will be provided in later sections for the comparison purpose.

- b.** What is the minimum overlap % you will use and why?

Answer:

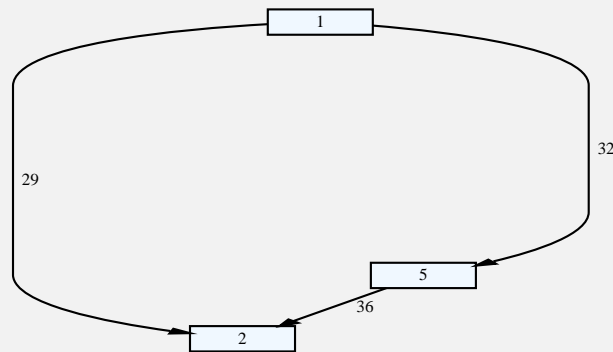
For the graph above, the filtering conditions are, a minimum overlapping cut-off of 20bp, best local alignment starts with the first base pair of one of the two sequences. Therefore, the minimum overlap percentage for the graph above is a proximately $20/50 = 40\%$. If we look at the absolute value, this is a very high cut-off threshold, but as my analysis progress, the input sequences are found to be highly redundant, I was able to obtain the complete sequence by even higher cut-off threshold. Any overlap percentages lower than 40% would provide a pretty huge graph consisting a lot of redundant edges and vertices.

I've copied two graphs with lower overlap cut-off threshold (30%) below, **Figure 2**. As we can see, there are a large number of redundant edges. Of course, we can further simplify all of these output graphs by taking away the redundant edges and vertices. We'll talk about it in the following question.

- c. Layout phase: using the graph fined the putative assembly(ies) of the fragments. You just need to provide the path(s) in the graph. Note that you can reduce the complexity of the graph by removing redundant nodes in the graph.

Answer:

Like we said, the input nucleotide sequences are highly redundant. Many pair-wise combinations provide perfect local alignments with either longer or shorter overlap. This resulted in redundant edges and vertices. Let use sequence 1, 2 and 5 as an example. Extracting these three vertices and edges connecting them from Figure 1, we have



We can represent this triangle relationship with the following local alignments. The two highlighted subsequences together contain the complete sequence information of this section.

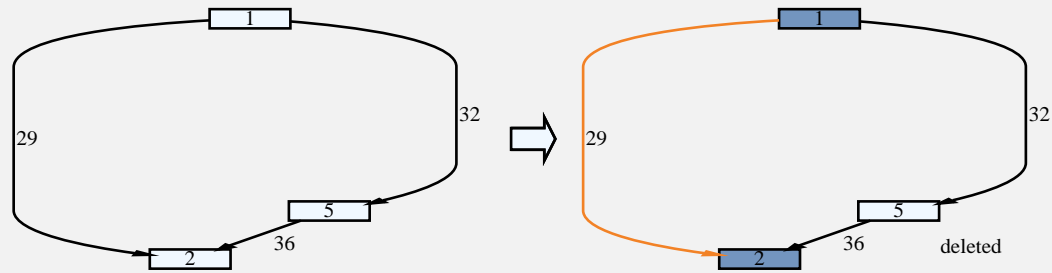
```
Seq 1: ATGGAGGTGTTTTACATGGTAAACGTACTTATCAACCAAATATACGTA
Seq 2:      AAAACGTACTTATCAACCAAATAAACGTAACACAGTAAAGTTCAT
           |      1 2 overlap 29      |

Seq 2:      AAAACGTACTTATCAACCAAATAAACGTAACACAGTAAAGTTCAT
Seq 5:      GGTAAACGTACTTATCAACCAAATAAACGTAACACAG
           |      5 2 overlap 36      |

Seq 1: ATGGAGGTGTTTTACATGGTAAACGTACTTATCAACCAAATATACGTA
Seq 5:      GGTAAACGTACTTATCAACCAAATAAACGTAACACAG
           |      1 5 overlap 32      |
```

Clearly, the sequence information of sequence #5 is completely covered by sequence #1 and sequence #2. Therefore, deleting sequence #5 would not affect retrieving the final sequence. In other word, if an edge has to present because of the transitivity property of the partial ordering, we don't have to draw this edge. Similarly, the overlapping length information of sequence #1 with sequence #2 is covered by the edges weights of sequence #1 with sequence #5 and sequence #5 with sequence #2. Therefore deleting edge 1, 2 would not affect retrieving the final sequence. In other words, if a node presents as the middle node in a triangle relation of the partial order, we don't have the draw this node.

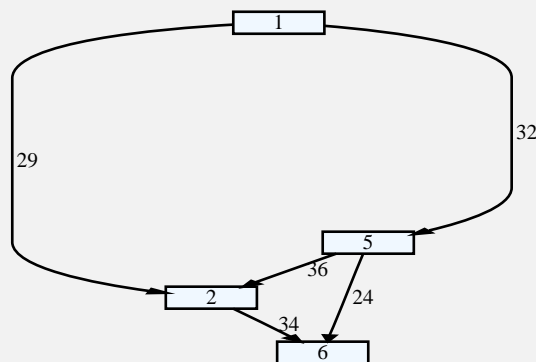
Now, we've detected what kind of edges or vertices are redundant, we can decide to delete either the redundant edges or redundant vertices. To simplify our graph to the greatest extend, deleting vertices is better. So, the conclusion is that we can delete the vertices in the partial ordering graph that present as a middle node of a triangle relation.



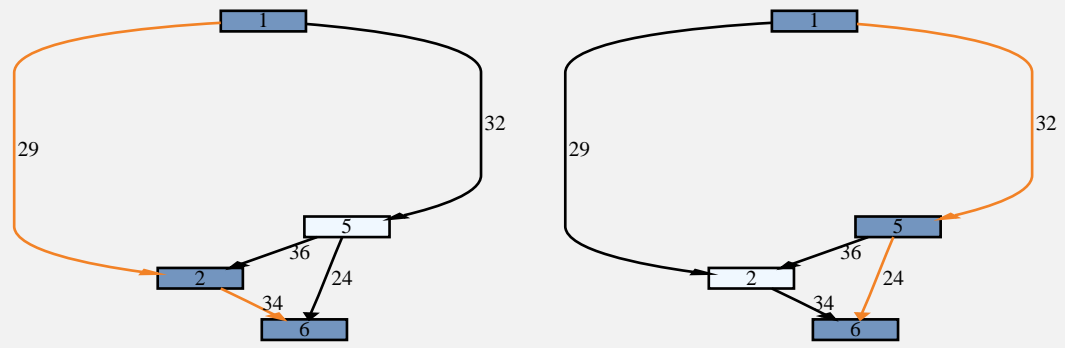
Delete the middle vertex in the triangle relation of a partial ordering graph

By doing so we basically retrieved a path from the starting node of the partial ordering graph to the end node the graph, which contains the complete sequence information. One way of deleting the vertices is shown in **Figure 3**. The dark blue colored vertices are the ones stay; the light blue colored nodes are deleted. The orange colored edges connect the remaining vertices.

Note that there is more than one way to delete the vertices. The previously deleted vertices would affect the triangle relations for the later vertices. Therefore, starting from different node, we would get different output paths. For example, for the following nodes, we can either delete vertex 5 like we did in the example path above, or we can delete vertex 2 if we evaluate from the bottom up.



A section of the original partial ordering graph



Multiple ways of deleting the vertices generating multiple possible paths

- d. Consensus: Proved an assembly view of the path suggested in the question above, If you have obtained more than one path, suggest while of these paths is more likely and provide the assembly view of it. The assembly view is when you provide the complete sequence supported by the layout of the fragments.

Answer:

I'll use the example path provide in **Figure 3**. As we discussed in the previous question, this example path is constructed by deleting vertices that present as the middle nodes in triangle relations of the partial ordering graph generated by the scripts. As we can see, only 9 vertices are needed to retrieve the complete sequence. They are 1, 2, 6, 11, 14, 9, 17, 10, and 21 in this particular order. The corresponding alignments are as below. The highlighted sections are the subsequence that each selected sequence contributes in retrieving the complete sequence.

Selected input sequences for the example path:

```
Seq 1:  ATGGAGGTGTTTTACATGGTAAACGTACTTATCAACCAATATACGTA
Seq 2:              AAAACGTACTTATCAACCAATAAACGTAAACACAGTAAAGTTCAT
Seq 6:              TCAACCAATAAACGTAAACACAGTAAAGTTCATGGTTTCAG
Seq 11:              CACAGTAAAGTTCATGGTTTCAGAAAACGCATGAGCACTAAAAACGGACG

Seq 11: CACAGTAAAGTTCATGGTTTCAGAAAACGCATGAGCACTAAAAACGGACG
Seq 14:              CGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGCGTC
Seq 9:              AGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGTCGTAAGGCCG
Seq 17:              AACGGACGTAAAGTATTAGCGCGTCGTCGTCGTAAGGCCGAAAAGTTTT
Seq 17: AACGGACGTAAAGTATTAGCGCGTCGTCGTCGTAAGGCCGAAAAGTTTT
Seq 10:              CGTAAAGGCCGTAAAGTTTTATCAGCATAAGATCACTGACCTATCAGTG
Seq 21:              ATAAGATCACTGACCTATCAGTGCTCTTTTTTGCTATAAATCATAAA
```

Assembled Sequence:

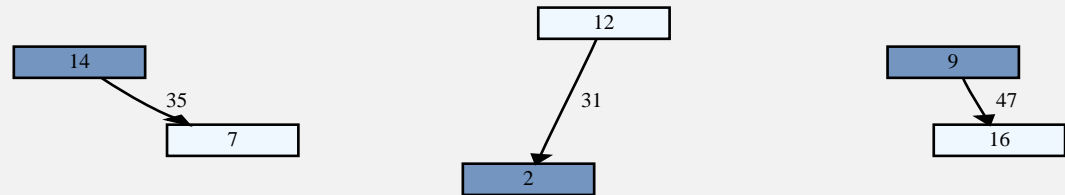
```
ATGGAGGTGTTTTACATGGTAAACGTACTTATCAACCAATATACGTAAACACAGTAAAGTTCATGGTTTCAGAAAACGCATGAGCACTAAAAACGGACGTAAAGTA
TTAGCGCGTCGTCGTAAGGCCGAAAAGTTTTATCAGCATAAGATCACTGACCTATCAGTGCTCTTTTTTGCTATAAATCATAAA
```

- e. What is the easiest way to validate your assembly?

Answer:

There are many ways to validate the assembly. The easiest way, I think, is to pick a random input sequence, which is not used in the path, and use the overlapping information with its neighbor, who is a selected vertex in the path, and see if this sequence is a perfect match on that region of the complete sequence.

I randomly selected three vertices/sequences that are not used in the example path. The following neighbor overlapping information is extracted from the graph.



Three examples to validate the assembled sequence

Example #1: we know the alignment of sequence #14 with the complete sequence:

Assembled Sequence:	ATGGAGGTGTTTTTACATGGTAAACGTACTTATCAACCAAATATACGTAAACACAGTAAAGTTCAT...
Assembled Sequence (continue):	GGTTTCAGAAAACGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTGTAAGGCC...
Seq 14:	CGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGCGTC
Assembled Sequence (continue):	GAAAAGTTTTATCAGCATAAGATCACTGACCTATCAGTGGCTTTTTTTTGCTATAAATCATAAA

We also know the local best alignment of sequence #14 with sequence #7:

Seq 14:	CGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGCGTC
Seq 7:	GAGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTGTAAGGCC

If our assembled sequence is correct, sequence #7 should align on the assembled sequence 35 base pairs downstream of sequence #14:

Assembled Sequence:	ATGGAGGTGTTTTTACATGGTAAACGTACTTATCAACCAAATATACGTAAACACAGTAAAGTTCAT...
Assembled Sequence (continue):	GGTTTCAGAAAACGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTGTAAGGCC...
Seq 14:	CGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGCGTC
Seq 7:	GAGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTGTAAGGCC
Assembled Sequence (continue):	GAAAAGTTTTATCAGCATAAGATCACTGACCTATCAGTGGCTTTTTTTTGCTATAAATCATAAA

It is indeed a perfect alignment. Therefore we validated the assembled sequence.

Example #2: we know the alignment of sequence #2 with the complete sequence:

Assembled Sequence:	ATGGAGGTGTTTTTACATGGTAAACGTACTTATCAACCAAATATACGTAAACACAGTAAAGTTCAT...
Seq 2	AAAACGTACTTATCAACCAAATAAACGTAAACACAGTAAAGTTCAT
Assembled Sequence (continue):	GGTTTCAGAAAACGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTGTAAGGCC...
Assembled Sequence (continue):	GAAAAGTTTTATCAGCATAAGATCACTGACCTATCAGTGGCTTTTTTTTGCTATAAATCATAAA

We also know the local best alignment of sequence #12 with sequence #2:

Seq 2:	AAAACGTACTTATCAACCAAATAAACGTAAACACAGTAAAGTTCAT
Seq 12:	GGTGTGTTTTACATGGTAAACGTACTTATCAACCAAATAAACGTAAA

If our assembled sequence is correct, sequence #12 should align on the assembled sequence 31 base pairs upstream of sequence #2:

Assembled Sequence:	ATGGAAGGTGTTTTTACATGGTAAACGTACTTATCAACCAAATATACGTAAACACAGTAAAGTTCAT...
Seq 2:	AAAACGTACTTATCAACCAAATAAACGTAAACACAGTAAAGTTCAT
Seq 12:	GGTGTGTTTTACATGGTAAACGTACTTATCAACCAAATAAACGTAAA
Assembled Sequence (continue):	GGTTTCAGAAAACGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTGTAAGGCC...
Assembled Sequence (continue):	GAAAAGTTTTATCAGCATAAGATCACTGACCTATCAGTGGCTTTTTTTTGCTATAAATCATAAA

It is indeed a perfect alignment. Therefore we validated the assembled sequence.

Example #3: we know the alignment of sequence #9 with the complete sequence:

```
Assembled Sequence:          ATGGAGGTGTTTTACATGGTAAACGTACTTATCAACCAATATACGTAACACAGTAAAGTTCAT...

Assembled Sequence (continue): GGTTCAGAAAACGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTAAAGGCC...
Seq 9:                        AGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTAAAGGCC

Assembled Sequence (continue): GAAAAGTTTTATCAGCATAAGATCACTGACCTATCAGTGGCTTTTTTTTGCTATAAATCATAAA
Seq 9 (continue):              G
```

We also know the local best alignment of sequence #9 with sequence #16:

```
Seq 9:      AGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTAAAGGCCG
Seq 16:      GCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTAAAGGC
```

If our assembled sequence is correct, sequence #12 should align on the assembled sequence 31 base pairs upstream of sequence #2:

```
Assembled Sequence:          ATGGAGGTGTTTTACATGGTAAACGTACTTATCAACCAATATACGTAACACAGTAAAGTTCAT...

Assembled Sequence (continue): GGTTCAGAAAACGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTAAAGGCC...
Seq 9:                        AGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTAAAGGCC
Seq 16:                        GCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGCTAAAGGC

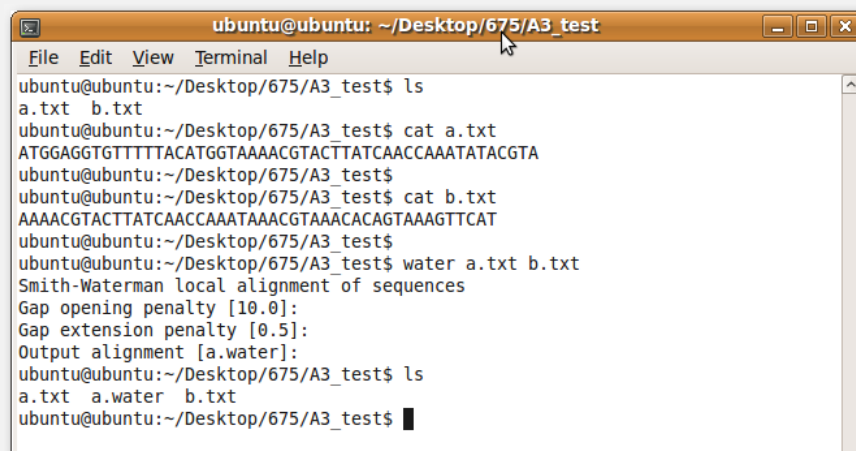
Assembled Sequence (continue): GAAAAGTTTTATCAGCATAAGATCACTGACCTATCAGTGGCTTTTTTTTGCTATAAATCATAAA
Seq 9 (continue):              G
```

It is indeed a perfect alignment. Therefore we validated the assembled sequence.

Tools

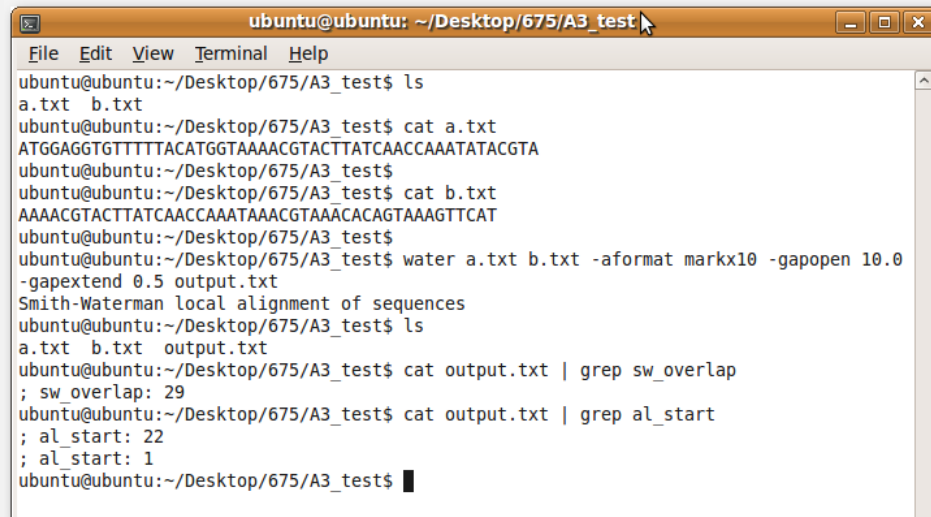
Water: Water program was used to carry out the task of searching for the best local alignment. The following simple examples demonstrate its use.

Discussion: I downloaded the EMBOSS package from the EMBOSS home site, and installed this tool suite on an Ubuntu machine. I was able to run it as a command line program. It takes to input files and output a text file. The following example demonstrate the basic usage of this program.



```
ubuntu@ubuntu: ~/Desktop/675/A3_test
File Edit View Terminal Help
ubuntu@ubuntu:~/Desktop/675/A3_test$ ls
a.txt b.txt
ubuntu@ubuntu:~/Desktop/675/A3_test$ cat a.txt
ATGGAGGTGTTTTACATGGTAAACGTACTTATCAACCAATATACGTA
ubuntu@ubuntu:~/Desktop/675/A3_test$ cat b.txt
AAAACGTACTTATCAACCAATAACGTAAACACAGTAAAGTTCAT
ubuntu@ubuntu:~/Desktop/675/A3_test$ water a.txt b.txt
Smith-Waterman local alignment of sequences
Gap opening penalty [10.0]:
Gap extension penalty [0.5]:
Output alignment [a.water]:
ubuntu@ubuntu:~/Desktop/675/A3_test$ ls
a.txt a.water b.txt
ubuntu@ubuntu:~/Desktop/675/A3_test$
```

Then, the user could also specify the type of output file and output file name in the command line as well as other parameters, such as the gap opening penalty and gap extension penalty.



```
ubuntu@ubuntu: ~/Desktop/675/A3_test
File Edit View Terminal Help
ubuntu@ubuntu:~/Desktop/675/A3_test$ ls
a.txt  b.txt
ubuntu@ubuntu:~/Desktop/675/A3_test$ cat a.txt
ATGGAGGTGTTTTACATGGTAAACGTACTTATCAACCAAATACGTA
ubuntu@ubuntu:~/Desktop/675/A3_test$ cat b.txt
AAACGTACTTATCAACCAAATAACGTAAACACAGTAAAGTTCAT
ubuntu@ubuntu:~/Desktop/675/A3_test$
ubuntu@ubuntu:~/Desktop/675/A3_test$ water a.txt b.txt -aformat markx10 -gapopen 10.0
-gapextend 0.5 output.txt
Smith-Waterman local alignment of sequences
ubuntu@ubuntu:~/Desktop/675/A3_test$ ls
a.txt  b.txt  output.txt
ubuntu@ubuntu:~/Desktop/675/A3_test$ cat output.txt | grep sw_overlap
; sw_overlap: 29
ubuntu@ubuntu:~/Desktop/675/A3_test$ cat output.txt | grep al_start
; al_start: 22
; al_start: 1
ubuntu@ubuntu:~/Desktop/675/A3_test$
```

In this example, I used the same input files which contain a sequence each. I specified the output file type to be “markx10”, the output file name to be output.txt, the gap opening penalty to be 10.0, and the gap extend penalty to be 0.5.

This example also shows the way to retrieve the useful information from the output file. In my script, they are done differently, but these are the entries in the output files that were used to analyze the sequence combination and filter out the irrelevant pairs. If the overlap is too short (with high penalties), or neither of the starting positions of the alignment is 1, this pair would be filtered out.

In this case, the overlapping length of the two input sequences is 29, which is reasonably high. Sequence contained in a.txt should be the upstream sequence and the other one is the downstream sequence. In other words, the edge representing this particular pair goes from a.txt’s sequence to b.txt’s sequence and the edge weight is 29.

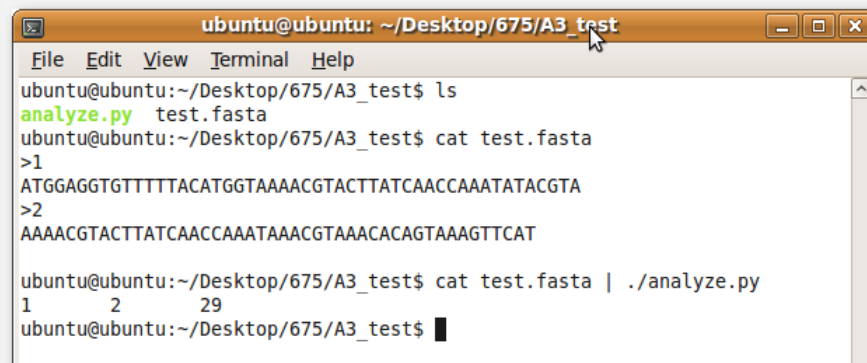
Graphviz: Graphviz program was used to carry out the task of plotting the edge weighted directed graphs using the processed water program outputs.

Discussion: I installed yapgvb, which provides python bindings to Graphviz program with an intuitive python interface. Then I used python script to plot the graphs. Examples will be provided in the Script-Graphs section.

Scripts

analyze.py: This script parses the fags.fasta file, generates all pair-wise combinations of the sequences, executes water on each pair of sequences, extracts key information from the water output files, and filters out irrelevant pairs. Finally, the script produces a tab delimited standard output containing all selected alignments

Discussion: In the following example, I used a test file in the fasta format. The file contained two sequences and there sequence names are 1 and 2. The output indicating that these two sequences are related with a overlap of 29 base pairs and sequence 1 should be the upstream sequence, sequence 2 should be the downstream sequence.



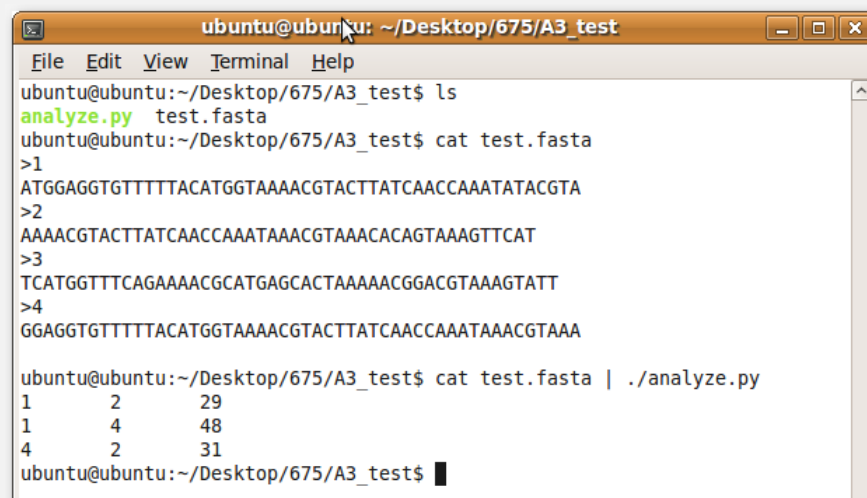
```
ubuntu@ubuntu: ~/Desktop/675/A3_test
File Edit View Terminal Help
ubuntu@ubuntu:~/Desktop/675/A3_test$ ls
analyze.py test.fasta
ubuntu@ubuntu:~/Desktop/675/A3_test$ cat test.fasta
>1
ATGGAGGTGTTTTACATGGTAAACGTACTTATCAACCAATATACGTA
>2
AAACGTACTTATCAACCAATAAACGTAAACACAGTAAAGTTCAT

ubuntu@ubuntu:~/Desktop/675/A3_test$ cat test.fasta | ./analyze.py
1      2      29
ubuntu@ubuntu:~/Desktop/675/A3_test$
```

We can double check this output by aligning the two sequences in the fasta format file:

```
Seq 1: ATGGAGGTGTTTTACATGGTAAACGTACTTATCAACCAATATACGTA
Seq 2:          AAACGTACTTATCAACCAATAAACGTAAACACAGTAAAGTTCAT
```

Indeed, we obtained a perfect alignment with an overlap length 29, starting from sequence #1 to sequence #2. Now let's look at a slightly more complicated input.



```
ubuntu@ubuntu: ~/Desktop/675/A3_test
File Edit View Terminal Help
ubuntu@ubuntu:~/Desktop/675/A3_test$ ls
analyze.py test.fasta
ubuntu@ubuntu:~/Desktop/675/A3_test$ cat test.fasta
>1
ATGGAGGTGTTTTACATGGTAAACGTACTTATCAACCAATATACGTA
>2
AAACGTACTTATCAACCAATAAACGTAAACACAGTAAAGTTCAT
>3
TCATGGTTTCAGAAAACGCATGAGCACTAAAAACGGACGTAAAGTATT
>4
GGAGGTGTTTTACATGGTAAACGTACTTATCAACCAATAAACGTAAA

ubuntu@ubuntu:~/Desktop/675/A3_test$ cat test.fasta | ./analyze.py
1      2      29
1      4      48
4      2      31
ubuntu@ubuntu:~/Desktop/675/A3_test$
```


In this second example, we had 4 sequences to be aligned. The output indicates that sequences #1, #2, #4 are related but sequence #3 is not. We double check this output by aligning these four sequences together.

```
Seq 1: ATGGAGGTGTTTTTACATGGTAAACGTACTTATCAACCAATATACGTA
Seq 4:  GGAGGTGTTTTTACATGGTAAACGTACTTATCAACCAATAAACGTAAA
Seq 2:                                     AAAACGTACTTATCAACCAATAAACGTAAACACAGTAAAGTTCAT

Seq 3: TCATGGTTTCAGAAAACGCATGAGCACTAAAAACGGACGTAAAGTATT (not related)
```

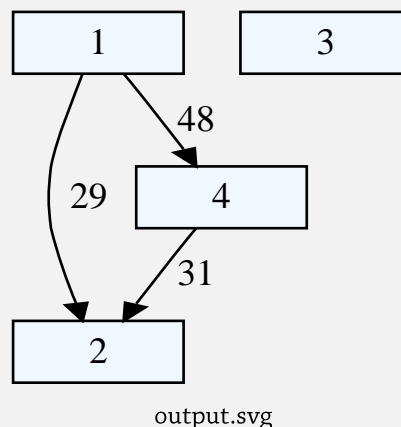
Indeed, we obtained three perfect alignments, between sequences #1 and #2 with an overlap length 29, between sequences #1 and #4 with an overlap length 48, and between sequences #2 and #4 with an overlap length 31. Sequence #3, on the other hand, is not really related with them.

graph.py: This script parses the standard input, computes the number of vertices, and plots an edge weighted direct graph in svg format using the layout option 'dot'.

Discussion: In the following example, I used the output from the last example of the previous section. As we discussed above, sequence #1 #2 are related, #1 #4 are related, #2 #4 are related, and #3 should not be related with any other sequences. These information should be presented in the graph.

```
ubuntu@ubuntu: ~/Desktop/675/A3_test
File Edit View Terminal Help
ubuntu@ubuntu:~/Desktop/675/A3_test$ ls
analyze.py graph.py test.fasta
ubuntu@ubuntu:~/Desktop/675/A3_test$ cat test.fasta | ./analyze.py
1      2      29
1      4      48
4      2      31
ubuntu@ubuntu:~/Desktop/675/A3_test$ cat test.fasta | ./analyze.py | ./graph.py
ubuntu@ubuntu:~/Desktop/675/A3_test$ ls
analyze.py graph.py output.svg test.fasta
ubuntu@ubuntu:~/Desktop/675/A3_test$
```

The output.svg graph is copied as below. It indeed demonstrates the sequence aligning information.



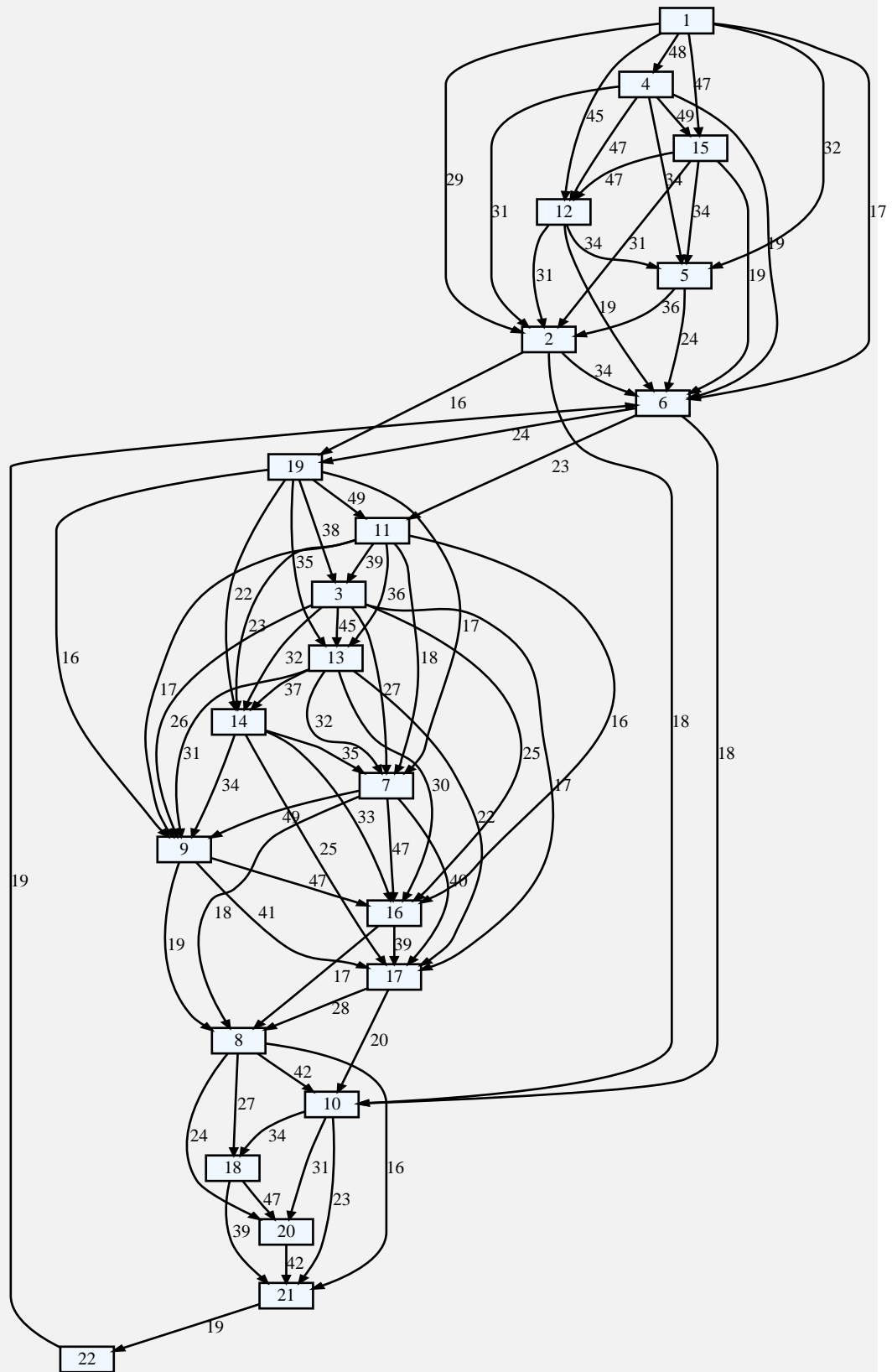


Figure 2. Overlap cut-off 30%

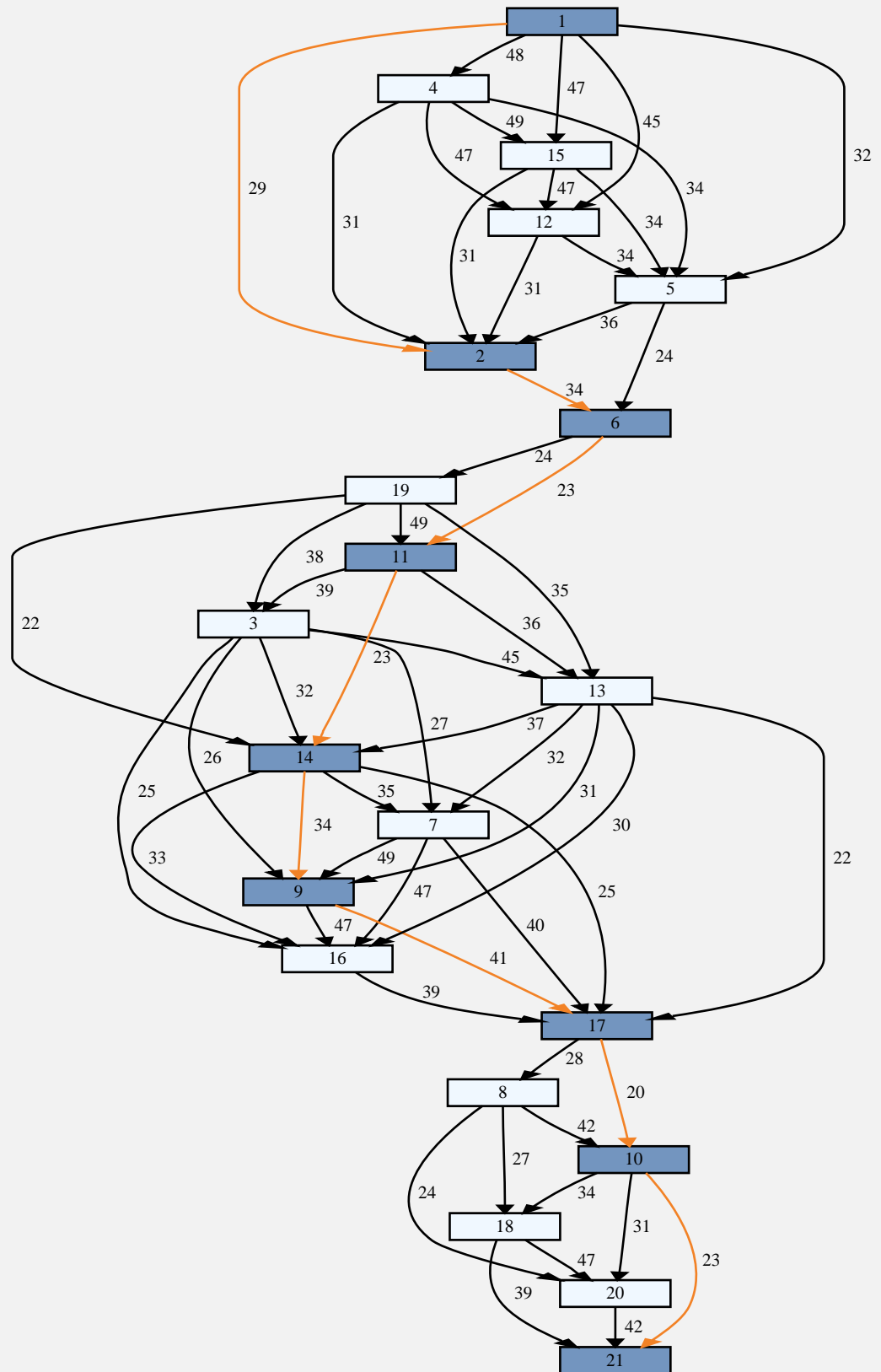


Figure 3. Example path constructed by deleting vertices

```

Seq 1:  ATGGAGGTGTTTTTACATGGTAAACGTACTTATCAACCAAATATACGTA
Seq 2:                AAAACGTACTTATCAACCAAATAAACGTAAACACAGTAAAGTTCAT
Seq 3:                                TCATGGTTTCAGAAAACGCATGAGCACTAAAAACGGACGTAAAGTATT
Seq 4:  GGAGGTGTTTTTACATGGTAAACGTACTTATCAACCAAATAAACGTAAA
Seq 5:                GGTAACCGTACTTATCAACCAAATAAACGTAAACACAG
Seq 6:                TCAACCAAATAAACGTAAACACAGTAAAGTTCATGGTTTCAG
Seq 7:                                GAGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGTAAAGGCC
Seq 8:                                GTCGTCGTCGTAAAGGCCGTAAAGTTTTATCAGCATAAGATCACTGACCT
Seq 9:                                AGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGTAAAGGCCG
Seq 10:                                CGTAAAGGCCGTAAAGTTTTATCAGCATAAGATCACTGACCTATCAGTG
Seq 11:                CACAGTAAAGTTCATGGTTTCAGAAAACGCATGAGCACTAAAAACGGACG
Seq 12:  GGTGTTTTTACATGGTAAACGTACTTATCAACCAAATAAACGTAAA
Seq 13:                                TGGTTTCAGAAAACGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGC
Seq 14:                                CGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGCGTC
Seq 15:  GAGGTGTTTTTACATGGTAAACGTACTTATCAACCAAATAAACGTAAA
Seq 16:                                GCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGTAAAGGC
Seq 17:                                AACGGACGTAAAGTATTAGCGCGTCGTCGTAAAGGCCGAAAAAGTTTT
Seq 18:                                GTTTTATCAGCATAAGATCACTGACCTATCAGTGGTCTTTTTTTTGCTTTT
Seq 19:                ACACAGTAAAGTTCATGGTTTCAGAAAACGCATGAGCACTAAAAATGGAC
Seq 20:                                TTATCAGCATAAGATCACTGACCTATCAGTGGTCTTTTTTTTGCTTTAAA
Seq 21:                                ATAAGATCACTGACCTATCAGTGGTCTTTTTTTTGCTATAAATCATAAA
Asmbld: ATGGAGGTGTTTTTACATGGTAAACGTACTTATCAACCAAATATACGTAACACAGTAAAGTTCATGGTTTCAGAAAACGCATGAGCACTAAAAACGGACGTAAAGTATTAGCGCGTCGTCGTAAAGGCCGAAAAAGTTTTATCAGCATAAGATCACTGACCTATCAGTGGTCTTTTTTTTGCTATAAATCATAAA

```

Figure 4. Overview of all sequences that aligned