University of Wisconsin-Madison	Spring 2019
BMI/CS 776: Advanced Bioinformatics	Homework #4
Prof. Colin Dewey	Due: Thu, May 2, 2019, 11:59 PM

Assignment Goals

- Match observed and theoretical spectra from mass spectrometry
- Compare and contrast algorithms for finding paths in networks
- Practice with suffix tree and trie data structures for DNA matching

Instructions

- To submit your assignment, log in to the biostat server **`mi1.biostat.wisc.edu`** or **`mi2.biostat.wisc.edu`** using your biostat username and password.
- Copy all relevant files to the directory **`/u/medinfo/handin/bmi776/hw4/<USERNAME>`** where **`<USERNAME>`** is your biostat username. Submit all of your Python source code and test that it runs on **`mi1.biostat.wisc.edu`** or **`mi2.biostat.wisc.edu`**. Do not test your code on **`adhara.biostat.wisc.edu`**.
- Compile all of your written answers in a single file and submit as **`solution.pdf`**.
- Write the number of late days you used at the top of **`solution.pdf`.**
- For the written portions, show your work for partial credit.

Part 1: Peptide-spectra matching

1A: *Generating simplified theoretical spectra*

Write a program **`spectra_generator.py`** that takes as input an amino acid mass table[*] and a set of peptide sequences and outputs a theoretical spectra library. Given a peptide sequence, the program should determine the mass-to-charge ratio (m/z) of all distinct type-b and -y ions and assign appropriate relative magnitude to each ion following the rules below.

(a) Assume all ions carry a single positive charge.
(b) Round a calculated m/z value to its nearest integer.
(c) Values equal to the m/z of the ions have a magnitude of 50.
(d) Values equal to ± 1 of the m/z of the ions have a magnitude of 25[†].
(e) If an m/z value can be assigned two magnitudes, pick the larger one.

You can use a simplified formula that ignores water molecules to compute the m/z for an ion:

[*] From http://rosalind.info/glossary/monoisotopic-mass-table, which also has other useful references for bioinformatics tasks in mass spectrometry.

[†] This and other implementation details come from https://link.springer.com/content/pdf/10.1016%2F1044-0305%2894%2980016-2.pdf

$$m/z = \frac{charge + \sum_{a \in ion} mass(a)}{charge} = 1 + \sum_{a \in ion} mass(a)$$

The $charge$ is always 1 per the assumption above, and $mass(a)$ is the monoisotopic mass for amino acid a in the ion from **mass_table.txt**. For example, the peptide **AYDN** can be fragmented into **A**, **AY**, **AYD**, **YDN**, **DN** and **N**. Its theoretical spectrum should be printed in the following format

```
AYDN
71	25
72	50
73	25
114	25
115	50
116	25
...
```

The first row is the peptide sequence. For all following rows, the first column has the m/z values and the second column has their corresponding magnitudes. The m/z values are sorted in increasing order. The two columns are separated by a tab. Insert an empty line to separate two spectra.

Your program should be runnable from the command line as follows:

```
spectra_generator.py --mass_table=<table> --peptides=<peptides> --out=<out>
```

You can test your program using the input file **peptides.txt**.

1B: *Computing cross-correlation scores*

Given an experimental spectrum, your task is to identify the best match in the theoretical spectra library. Write a program **xcorr.py** that takes as input an experimental spectrum and the theoretical spectra library generated by **spectra_generator.py** run on **mass_table.txt** and **peptides.txt** and outputs a ranked list of cross-correlation scores.

The experimental spectra **spectrum1.txt**, **spectrum2.txt** and **spectrum3.txt** have been preprocessed so that each of them only includes the relative intensities for the 200 most abundant ions. The first column has the m/z values and the second column has their corresponding relative intensities. For the experimental and theoretical spectra, assume an intensity of 0 for any m/z value that is not listed. Compute the cross-correlation score between the experimental spectrum and each theoretical spectrum over the range [1, **max**], where **max** is the maximum m/z value over all experimental and theoretical spectra. The offset τ is varied over [-10, 10].

$$xcorr = R_0 - \left(\sum_{\tau=-10}^{10} R_\tau \right) \Big/ 21$$

$$R_\tau = \sum_{i=1}^{max} theoretical[i] \cdot experimental[i+\tau]$$

Normalize the xcorr scores to 1 by dividing them by the maximum score, rank them in decreasing order, and round them to three decimal places. In your output file, the first column has the normalized scores, and the second column has their corresponding peptide sequences. The two columns are separated by a tab.

You can run your program from the command line as follows:

```
xcorr.py --query=<spectrum> --library=<library> --out=<out>
```

You can use the example output files **`xcorr_out1.txt`**, **`xcorr_out2.txt`**, and **`xcorr_out3.txt`** to check the output of your **`xcorr.py`** implementation for the respective experimental spectra.

Part 2: Source-target paths in networks

You will use the Python **`networkx`** package (version 1.11 required) to compare and contrast two algorithms for finding source-target paths in a network. One optimizes the min cost flow and is similar to (but not identical to) ResponseNet. The other finds the *k* shortest weighted paths.

In both cases, you are given an undirected network where each line in the input file lists a pair of nodes followed by their weight. The weight is the cost of transmitting flow in the flow problem[‡]. The **`networkx`** flow algorithms require directed graphs, so we represent an undirected edge as a pair of directed edges with the same weight.

In addition to the network, you are provided with a list of source nodes and target nodes. These sources and targets will be connected to an artificial source and an artificial target, as in ResponseNet. The objective is then to find connections from the artificial source to the artificial target.

[‡] Note that we must use integer-valued weights. The **`networkx network_simplex`** implementation appears to not terminate in some cases when floating point weights are used, as noted in its source code.

You are provided a mostly complete implementation **`find_paths.py`** that you will finish and test below. This file contains the flow-based and shortest paths-based source-target path algorithms, and the algorithm is selected based on the input parameters. The program is callable from the command line as follows:

```
python find_paths.py --edges_file=<edges> --sources_file=<sources>
     --targets_file=<targets> --flow=<flow> --output=<output>
```

or

```
python find_paths.py --edges_file=<edges> --sources_file=<sources>
        --targets_file=<targets> --k=<k> --output=<output>
```

where

- **`<edges>`** is a text file listing weighted undirected edges one per line
- **`<sources>`** is a text file listing source nodes one per line
- **`<targets>`** is a text file listing target nodes one per line
- **`<output>`** is a the filename for the output
- **`<flow>`** is a positive number specifying the amount of flow to send from the artificial source to the artificial target
- **`<k>`** is a positive integer specifying the number of shortest paths to find

2A: *Completing the path-finding implementations*
Search for and complete the parts of the functions annotated with five *TODO* comments in **`find_paths.py`**. The networkx documentation at https://networkx.github.io/documentation/networkx-1.11 will be useful for learning how it represents the graph data structure and implements the path finding algorithms. In particular, review:

- https://networkx.github.io/documentation/networkx-1.11/tutorial/tutorial.html#directed-graphs
- https://networkx.github.io/documentation/networkx-1.11/reference/algorithms.flow.html#capacity-scaling-minimum-cost-flow
- https://networkx.github.io/documentation/networkx-1.11/reference/algorithms.simple_paths.html

You can use the provided **`print_graph`** and the **`networkx draw`** function to inspect the directed graph object that you load. The example input files **`example_graph.txt`**, **`example_sources.txt`**, and **`example_targets.txt`** can be used to test your code. When **`find_paths.py`** is run with **`--flow=3`**

you should obtain **`example_paths_flow_file.txt`** or the equally good[§] **`example_paths_alt_flow_file.txt`**. When it is run with **`--k=7`** you should obtain **`example_paths_shortest_file.txt`**.

2B: *Test your implementation on a new network*
Test **`find_paths.py`** on the input files **`test_graph.txt`**, **`test_sources.txt`**, and **`test_targets.txt`**. Run **`find_paths.py`** with **`--flow=3`** and store the results in a file named **`test_paths_flow_file.txt`** that you should include in your handin directory.

Then run **`find_paths.py`** with **`--k=8`** and store the results in a file named **`test_paths_shortest_file.txt`** that you should include in your handin directory.

2C: *Compare min cost flow and shortest paths*
Based on your empirical testing of the two algorithms, their descriptions in the **`networkx`** documentation, and any experiments you conduct on your own, compare and contrast min cost flow (specifically the version we have implemented with unit capacity on all edges) and *k* shortest paths. What are the unique advantages of each method?

Hint: Examining how the edge 2-5 and the edge 5-11 are used in the flow-based and shortest path-based solutions in 2B will reveal some interesting behavior. This should not constitute your entire answer but can help you start to think about differences between the methods.

2D: *Special cases of the algorithms*
So far we have used infinite capacity on the edges incident to the artificial source and artificial target and capacity of 1.0 on all real edges in the network. Describe how to change the capacities such that the min cost flow solution will return essentially the same solution as *k* shortest paths for some value of *k*. What value of *k* is relevant for this special case?

Part 3: Suffix trees and tries for DNA matching
Consider the reference DNA sequence `CACTACGTACG`.

- **a)** Draw the suffix tree for this sequence.
- **b)** Show the path taken through the suffix tree to match the query sequence `ACG` and give the matching start position(s) for this query string.
- **c)** Draw the threaded suffix trie for all *k*-mers of the reference sequence, with $k = 3$ (be sure to include the threading pointers).

[§] The flow algorithm is not completely deterministic and can break ties among equally good solutions that have the same cost in different ways on different machines.

d) Show the path taken through the suffix trie to find all matching *k*-mers of the query sequence CGTACGT and give the matching position of *k*-mers in the query and reference.