University of Wisconsin-Madison	Spring 2017
BMI/CS 776: Advanced Bioinformatics	Homework #4
Prof. Anthony Gitter	Due: Thu, May 4, 2017, 11:59 PM

Assignment Goals

- Gain experience with the suffix tree and interpolated Markov model algorithms
- Compare and contrast algorithms for finding paths in networks

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: Maximal Unique Match (MUM) Finder Implementation

Write a program, **`find_MUM.py`**, that takes as input a list of pairs of DNA sequences and finds longest maximal unique matching subsequence(s) in each pair.

In this assignment, you must implement a suffix tree to build a MUM finder. Build one generalized suffix tree for both sequences, as shown in lecture. You do not need to implement the highly-optimized suffix tree construction algorithm described in the MUMmer paper.

If there are several maximal unique matches, output all MUMs that have the longest (*maximum*) length. For example, if you find MUMs **`ACCTG`**, **`GATC`**, and **`TTACC`**, then output the length 5 MUMs **`ACCTG`** and **`TTACC`**.

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

```
python find_MUM.py --seq_file=<sequences> --output=<ouput>
```

where

- **<sequences>** is a text file containing DNA sequences one per line. Every two lines consist of a pair. Every pair is separated by a blank line.
- **<output>** is the name of the text file into which the program will output the predicted the longest MUM or MUMS for each input pair. Write the MUM(s) for each input pair on a single line, using a comma to separate them if there is more than one longest MUM. If there are no MUMs, output a blank line.

Input files and sample scripts can be downloaded from
https://www.biostat.wisc.edu/bmi776/hw/hw4_files.zip

For this assignment, we recommend using object-oriented programming. A basic example is provided as **class_template.py** for you to see how a class is represented in Python[1].

To test your program, you may use the examples given in the slides:

Input:

```
ccacg
cct

acat
acaa
```

[1] https://www.tutorialspoint.com/python/python_classes_objects.htm provides more details on Python classes including the **__init__** and **self** syntax

Output:

```
cc
aca
```

Part 2: Interpolated Markov Models

We will use the interpolated Markov model approach from GLIMMER to estimate the probability $P_{IMM,3}(\mathtt{A}|\mathtt{TTA})$. For the sub-parts below, suppose we have the following counts in our training data. Show your work for partial credit.

TTAA	15
TTAC	20
TTAG	10
TTAT	5
Total	50

TAA	80
TAC	70
TAG	40
TAT	10
Total	200

AA	450
AC	260
AG	150
AT	40
Total	900

2A: χ^2 *test*

In order to calculate the λ values, we must first perform the χ^2 statistical test to determine whether the distributions of the current character depend on the order of the history. First, compute the χ^2 test statistic, rounded to the tenths place, comparing the 3rd order and 2nd order counts in the training data. Then use the *p*-value table for a χ^2 test with 3 degrees of freedom in the provided **`chisquare_df3_pvalues.txt`** to lookup the *p*-value for this test statistic and round to the thousandths place. Finally, compute $d = 1 - p$ to obtain the GLIMMER confidence score.

Repeat the χ^2, *p*-value, and *d* calculations for the 2nd order and 1st order comparison.

Recall that the χ^2 test statistic for an *n* by *m* contingency table is defined as

$$\chi^2 = \sum_{i=1}^{n} \sum_{j=1}^{m} \frac{\left(O_{i,j} - E_{i,j}\right)^2}{E_{i,j}}$$

where $O_{i,j}$ is the observed count in the contingency table and $E_{i,j}$ is the expected count

$$E_{i,j} = \frac{R_i C_j}{N}$$

R_i is the sum of the entries in row i, C_j is the sum of the entries in column j, and N is the sum of all entries in the contingency table. In this test there are $n = 4$ rows for the nucleotides and $m = 2$ columns for the n^{th} and $(n-1)^{\text{th}}$ order histories so there are 3 degrees of freedom.

2B: *Calculating λ*
Use the values of *d* calculated above, the training data counts, and the λ definition from GLIMMER to calculate λ_3(**TTA**), λ_2(**TA**), and λ_1(**A**).

2C: *Interpolated Markov model probability*
Use the λ values and the probabilities estimated from the training data counts to compute $P_{IMM,3}$(**A**|**TTA**).

Part 3: Source-target paths in networks

You will use the Python **networkx** package 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[2]. The **networkx** flow algorithms

[2] 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.

require directed graphs, so we represent an undirected edges 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.

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

3A: *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.readthedocs.io/en/stable/reference/index.html will be useful for learning how it represents the graph data structure and implements the path finding algorithms. In particular, review:

- https://networkx.readthedocs.io/en/stable/reference/classes.digraph.html
- https://networkx.readthedocs.io/en/stable/reference/algorithms.flow.html#capacity-scaling-minimum-cost-flow
- https://networkx.readthedocs.io/en/stable/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`. When it is run with `--k=7` you should obtain `example_paths_shortest_file.txt`.

3B: *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.

3C: *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? What types of solutions, edge usage, and/or paths do we obtain with one method but not the other?

Hint: Examining how the edge 2-5 and the edge 5-11 are used in the flow-based and shortest path-based solutions in 3B will reveal some interesting behavior. This should not constitute your entire answer but can help you start to think about differences between the methods.

3D: *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 that is relevant for this special case?