University of Wisconsin-Madison	Spring 2016
BMI/CS 776: Advanced Bioinformatics	Homework #3
Prof. Anthony Gitter	Due: Thu, Apr 14, 2016, 11:59 PM

Assignment Goals

- Learn an alternative strategy for predicting source-target connections in a network
- Gain familiarity with the RNA-seq and mass spectrometry quantification tasks

Part 1: Pathway Prediction

In class, we studied the ResponseNet algorithm for predicting cellular pathways that connect source proteins (e.g. genetic screen hits) to target proteins (e.g. transcriptional regulators) using network flow. Many other graph algorithms have been used to solve related biological problems. Here will implement the *k*-shortest paths approach to this problem, which remains a popular technique for predicting pathways[1].

Given a weighted, undirected network, a list of source nodes, and a list of target nodes, your program should identify all source-target paths in the network and output the *k* paths with the lowest cost. The weight on an edge represents the cost of using that edge to move from one node to the other. We will only consider simple paths, which are paths that do not contain any particular node more than one time. A path's cost is the sum of all edge costs along the path.

We outline below one possible strategy for solving this problem, but you are welcome to implement an alternative algorithm or use different data structures. One solution to this problem uses a depth-first search to enumerate all simple source-target paths and then sorts the paths to identify the *k* paths with the lowest cost. There are two main components required to implement a depth-first search: a data structure for retrieving the neighbors of a node (nodes connected by an undirected edge) quickly and a recursive function for the search.

Two common data structures for representing a graph are the adjacency matrix and adjacency list. Either is suitable for this problem because the graph instances will be small. Consider the small graph to the right to see how to form an adjacency matrix and adjacency list.

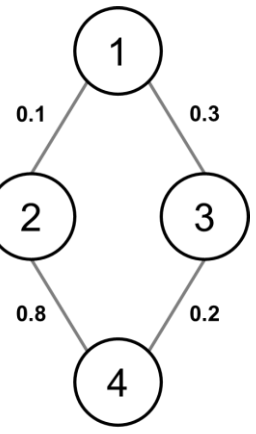

Figure 1: Example Graph

[1] http://www.nature.com/articles/npjsba20162

An adjacency matrix is a matrix with one row and column for every node. The entries in the matrix are 0 if there is no edge between the row node and column node and the weight of the edge if the edge does exist. The matrix is symmetric for an undirected graph. For the example graph in Figure 1:

	1	2	3	4
1	0.0	0.1	0.3	0.0
2	0.1	0.0	0.0	0.8
3	0.3	0.0	0.0	0.2
4	0.0	0.8	0.2	0.0

An adjacency list has an entry that tracks the neighbors of each node. Because we have a weighted graph, the adjacency list must also track the edge costs. This data structure can be implemented using dictionaries, arrays, or other basic data structures, but here we assume dictionaries are used. In that case, the *primary* dictionary has a key for every node in the graph. The associated value for that key is another dictionary with the neighbors of that node. The *neighbor* dictionary has the neighboring node as the key and the edge cost as the value. An example for the graph above that uses **`key:value`** syntax:

```
{
 1: {2: 0.1, 3: 0.3},
 2: {1: 0.1, 4: 0.8},
 3: {1: 0.3, 4: 0.2},
 4: {2: 0.8, 3: 0.2}
}
```

After creating the graph data structure, we need to implement a recursive function to perform the depth-first search. Without detailing the variables that need to be passed during each recursive call, the pseudocode below outlines the major functionality of this function. Note that we need to track which nodes have been visited already on the path in order to avoid cycles. We terminate the search when we reach any target node.

```
dfs(node, visited, …)
      for neighbors of node
            if neighbor not in visited
                  if neighbor in targets
                        add path to discovered_paths
                  else
                        dfs(neighbor, {visited + node}, …)
```

Your function may need to return **`discovered_paths`** depending on whether you store this is a local or global variable. Part of your task is to determine how to use the recursive calls to construct the order of nodes along the path and the cumulative path cost.

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

```
PredictPaths <graph file> <k> <paths file>
```

Where

- **`<graph file>`** a text file describing the graph and formatted as follows:

`N`	Number of nodes in the graph labelled from 1 to *n*
`s1 s2 s3 …`	List of source node ids separated by whitespaces
`t1 t2 t3 …`	List of target node ids separated by whitespaces

`n1    n2    w1`
`n3    n4    w2`
`n5    n6    w3`
`…`
	List of weighted undirected edges one per line. The components of each line are separated by whitespaces

- **`<k>`** is an integer value represents the number of paths with lowest cost that should be returned. In case the *k*th path and the (*k*+1)th path have the same cost, the program may return either randomly.
- **`<paths file>`** is a text file into which the paths and their corresponding costs should be written. The paths should be sorted in an increasing order of their cost and formatted as follows:

```
1 -> 5 -> 9  [0.5000]
1 -> 4 -> 5 -> 9  [0.9000]
3 -> 6 -> 10  [1.0000]
2 -> 8 -> 9  [1.1000]
1 -> 5 -> 7 -> 9  [1.9000]
…
```

The line `1 -> 4 -> 5 -> 9  [0.9000]` means there is a path starts at source node `1`, passes through the nodes `4` and `5` to reach the target node `9` with a total path cost of `0.9000`.

Example input files and sample scripts can be downloaded from https://www.biostat.wisc.edu/bmi776/hw/hw3_files.zip

To test your program, you may use the graph in Figure 2. The file describing the graph is called **`testgraph.txt`** and the expected output when trying to predict the **`6`** lowest cost paths are stored in the file called **`testpaths.txt`**.

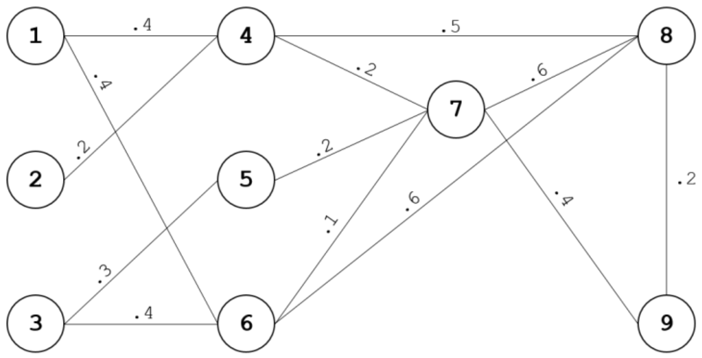

Figure 2: Test Graph

If you are using a language that is not compiled into machine code, such as Java, then you should make a small script called **`PredictPaths`** that accepts the command line arguments and invokes the appropriate source code and interpreter.

Recall that only standard libraries for the chosen language may be used. Third-party libraries must be approved by the instructor or TA on the Piazza forum.

Part 2: RNA-seq Rescue Algorithm

The full RSEM algorithm is too complicated to execute manually, but we can use the RNA-seq rescue method presented in class to approximate one iteration of expectation maximization. The bipartite graph below contains two types of nodes: transcripts and read groups. The transcript nodes contain a transcript id and the transcript length in base pairs (bp). The read nodes contain the read counts for a group of reads that all align to the same transcripts. Transcript-read group edges designate the transcripts to which each read group aligns.

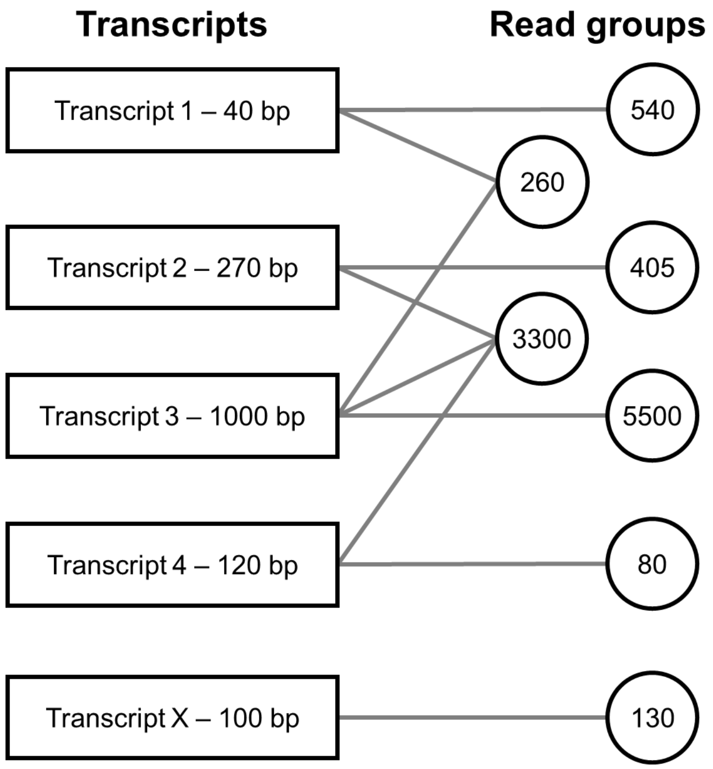

Figure 3: Bipartite Graph

2A: *Estimating relative abundance*

Use the rescue method to calculate the relative abundance for the five transcripts to three decimal places. Show your work for partial credit.

2B: *Estimating absolute abundance*

Transcript X is a RNA spike-in. 1000 copies of transcript X were mixed into the experimental sample when preparing the sample for RNA-seq. That is, its absolute abundance is 1000. Use the relative abundances you calculated above to calculate the absolute abundances for the other four transcripts, rounded to the nearest whole number. Show your work for partial credit.

Hint: Review the "Issues with relative abundance measures" slide from the RNA-seq lecture. Given the relative abundances for all six genes and the absolute abundance of Gene 6, you can derive the absolute abundances of Genes 1 through 5.

Part 3: Mass Spectrometry Theoretical Spectrum

In this problem you will generate a simplified version of a theoretical spectrum to better understand the relationship between peptide fragments and spectra. This problem is inspired by similar problems from the Rosalind bioinformatics programming platform. Consult the Rosalind problems http://rosalind.info/problems/prtm/ and http://rosalind.info/problems/spec/ for a review of mass spectrometry terminology.

For each of the peptide fragments below, you will create a table that shows the masses of all possible ions that correspond to the peaks in the theoretical spectrum. Specifically, for a peptide fragment containing *k* amino acids, provide a two-column table in which the first column lists the *k*-1 b-ions and *k*-1 y-ions. Enumerate *all* b-ions and y-ions even if they are redundant. In the second column of the table, provide the monoisotopic mass of the ion (amino acid string). An ion's monoisotopic mass is the sum of the monoisotopic masses of all amino acids in the string, where the amino acid masses can be obtained from the Rosalind table http://rosalind.info/glossary/monoisotopic-mass-table/. You do not need to draw the theoretical spectrum so you can ignore the charge of the ions.

3A: *First peptide fragment*
Create the table for the peptide fragment **`AYDNV`**.

3B: *Second peptide fragment*
Create the table for the peptide fragment **`CCHNQC`**.

3C: *Third peptide fragment*
Create the table for the peptide fragment **`GHYLCNA`**.

Submission Instructions

a) Login to the biostat server **`mi1.biostat.wisc.edu`** or **`mi2.biostat.wisc.edu`** using your BMI (biostat) username and password.
b) Copy any relevant files to the directory **`/u/medinfo/handin/bmi776/hw3/<USERNAME>`** where **`<USERNAME>`** is your BMI (biostat) username. For the programming part, make sure to submit the source code as well as any required files to run the program on the biostat server. For the rest of the assignment, if not otherwise instructed, compile your answers in a single file and submit as **`<USERNAME>.pdf`**.
c) Make sure to write the number of late days used at the first line of your pdf file.
d) If you are submitting a program, make sure it is runnable from the script and not only from the source code.