**CS 251 Project 4**

**Part 1-REQUIRED**

In this part, you will implement various graph-related structures and algorithms. The development is test-driven, so there is a lot of flexibility in how you can choose to implement things. You MUST implement the classes and algorithms yourself--i.e. do not try to use Java libraries to do the work for you. If there is a structure like ArrayList that you want to use to help in your implementation and you are not sure if you should use it, ask!

In general, you must implement classes and methods that will allow you to correctly handle the following commands, which will be provided in test input files. Note that all test cases will contain only a single graph (either directed or undirected.) Also note that vertices are always named with numbers starting with 0. (e.g. a graph with 6 vertices will have vertices 0, 1, 2, 3, 4, and 5). Also, be sure to allow for weights with decimal values (i.e. double values) because Part 2 will require that. To make your work easier for part 1, you may assume that the weight of the edge is at most to ***three*** decimal places. This only holds for part 1. Also, check the expected output files for the exact way information is to be printed out. Your output needs to match it exactly to pass the cases.

* **ugraph *n*** : construct an undirected graph with n vertices and no edges
* **dgraph *n*** : construct a directed graph with n vertices and no edges
* **add *u v w*** : add an edge to a graph that goes from vertex *u* to vertex *v* and has weight *w*; if *u* or *v* are not vertices in the graph, print “vertex does not exist”
* **v** : print the number of vertices in the graph
* **e** : print the number of edges in the graph
* **adj *v*** :print a list of vertices that are adjacent to vertex *v* (for grading purposes, these should be in increasing order--e.g. 2 4 6 if vertex *v* is adjacent to vertices 2, 4, and 6); if there are no adjacent vertices, print “none.”

NOTE: for a directed graph, “adjacency” is defined by edges going out of a vertex.

* **weight *u v*** :print the weight of the edge from vertex u to vertex v; print “edge does not exist” if such an edge does not exist
* **matrix** : print the adjacency matrix representation of the graph; if no edge exists between two vertices, print an X in that spot. (Be careful about the alignment when printing out the matrix. If a number is double, you may assume that it has three decima but only for part 1.)
* **tclosure** : print the adjacency matrix representation of the transitive closure of the graph

NOTE: This will only be tested on directed graphs where edge weights are all one so you don’t need to worry about the alignment of the matrix you print out.

* **spath *u v*** : print the shortest path from vertex *u* to vertex *v*, and print the total weight; if such a path does not exist, print “path does not exist”

NOTE: You can assume that only one shortest path will exist for each pair of vertices in a given graph.

NOTE: in this section, no negative weight edges will be used in the test cases, but you may want to look at part 2 before deciding on a shortest path algorithm.

* **tsort** : print **all** possible topological sortings of the graph; if there are none, print “No topological ordering exists”; if there are multiple orderings, put them in non-decreasing lexicographic order (e.g. 1 4 5 before 4 5 1; this should be easy to do if you are careful about which vertex is chosen first when you have an option.)

NOTE: This is only applicable to directed graphs and does not have to be handled for an undirected graph.

* **mst** : print the edges of **all** minimum spanning trees of the graph

NOTE: This will only be tested on undirected graphs. You should print the edges in non-decreasing lexicographic order (e.g. (1, 3) before (1, 4) before (2, 3)). There maybe multiple MSTs in a graph. See test cases for examples. You should print the different MST in non-decreasing lexicographic order.

Hint: You may modify the MinHeap from Project 3 and write your own Union Find data structure if it helps with your implementation.

NOTE: Some of the MST tests will have a unique MST.

* **sconn**: print whether the graph is strongly connected or not

NOTE: this is only tested on directed graphs.

* **components**: print the connected components of a graph; print vertices in increasing order and order the components by the first vertex in each (e.g. {1, 5, 6} before {2, 3, 4})

NOTE: this is only tested on undirected graphs

* **simple**: print whether the graph is simple or not

**Other Information**

* The only code we are providing for this project is **Main.java**, which reads in the input file and parses the commands. Once you implement each command, you will need to update Main.java to call the appropriate functions. The commands and arguments from the files are already parsed, so it is just up to you to implement them and call them accordingly.
* It is strongly suggested that you implement this project piece by piece and test each bit of functionality separately. We have provided test files for each function, and you can follow this structure to create new test cases if you desire. You are likely to be more successful if you use this more systematic approach than trying to implement everything at once.
* To run one of the tests, use the **test.sh** script by running the following command in the terminal: ./test.sh <input> <output> where <input> is the input file name and <output> is the filename with the expected output. If the test doesn’t pass, you may have to manually inspect the output.
* If you prefer testing with junit, you may use the test files provided and implement them as junit tests.
* **Submission**: You should submit your **Main.java** as well as any other **.java** files you created. Do not submit any other files.

**Part 2-OPTIONAL (and EXTRA CREDIT)**

This part is optional and will be run as a competition to see who can discover the best arbitrage opportunity in a given graph. Extra credit points will be awarded according to how much money you make, following these specifications.

Given a set currency exchanges as discussed in class, you must first construct an appropriate graph to represent the exchanges. Then, given a specific starting currency and an amount of money, you must try to determine the best sequence of exchanges that will make you the most profit in the original currency.

There are two added stipulations: (1) Each exchange will charge a 0.1% fee. For example, if you are exchanging 100 USD to EUR at the rate listed below, you would actually get 100\*0.999\*0.751 EUR. (2) You are limited to 20 exchanges or less (and remember you have to start and end at the same place.)

Your output should print the exchanges you use in the following format:

USD EUR GBP USD EUR GBP USD

This means that you take the starting money in USD, convert it to EUR, then to GBP, and so on, until you end up with USD again.

Note that when this is tested, it will be with a different set of currencies and a different starting currency, so you should try to come up with an algorithm that finds the “best” exchange cycle for any input.

Note also that this is a slightly different problem than the one in class because (1) you are given a specific currency that you must start and end with and (2) there is the 0.1% fee for every exchange and (3) you are limited to 20 exchanges or less.

Example Input to build graph where *i* is the row and *j* is the column and “*i* buys *j*”.

|  |  |  |  |
| --- | --- | --- | --- |
|  | USD | EUR | GBP |
| USD | 1 | 0.751 | 0.667 |
| EUR | 1.349 | 1 | 0.888 |
| GBP | 1.521 | 1.126 | 1 |

Example: Given starting currency as USD

Possible arbitrage path: USD EUR GBP USD EUR GBP USD

In the input files, the first line is the starting currency and the following lines show the exchange rates. It is recommended that you have a mapping of some sort from currencies to integers, so that you can re-use the graph classes from Part 1.

There is no output file for this because the path you output will depend on your algorithm, but the format should be the list of currencies in the exchange. Your program, called **Arbitrage.java** should read in the file, create the graph, perform some kind of algorithm to discover the “best” exchange path according to the above guidelines and print the results.

NOTE: Do not hardcode the currency names! We will provide multiple input files so that you can be sure your program will run for different inputs.

NOTE: Since there is no “right” answer for this, we are not providing tests. It is up to you to test your answer to see if it does, indeed, produce a “profit.”