# Homework 4 Writeup

In this homework, we'll focus again on graph algorithms and applications. There will be two main problems: in the first, we'll focus on a financial opportunity known as "arbitrage", and how we can discover it by representing our problem as a graph. The second part of the homework will be more theoretical, covering a graph algorithm for minimum spanning trees. As usual, read the homework notes carefully. If there are any topics that you aren't as comfortable with, it is highly encouraged to rewatch the lecture recordings! Note: in both questions, we will give you the runtime of the most efficient algorithm that you should try to write. If you can't come up with the fastest possible algorithm, you can still get partial credit for implementing something less efficient!

Before starting each question, we will briefly explain the Graph class that you will be working with:

The abstract class WeightedGraph.java is a slightly more heavy-weight graph class than the one you used in the last homework. We've used a 2d array to store edge weights, in addition to using the same adjacency list from the last homework to store edges. The main functions that you'll use from it are:

-addEdge(int u, int v, E val). Adds the edge (u,v) to the graph, with weight "val"

-neighbors(int v). Returns a list of vertices adjacent to v

-weight(int u, int v). Returns the weight of the edge (u,v), if it exists (and null if otherwise)

Notice that the addEdge() function takes in a weight value with a parametrized type. As you'll see below, the two algorithms that you will be implementing will treat weights differently. In the arbitrage question, weights will be doubles. In the minimum spanning tree question, edge weights are classified as either light, medium or heavy. To represent this, we've created a "Weight" enum type that you will use in the MST algorithm (don't worry to much about this until you get to the second part of the homework).

## Part 1: Arbitrage

You will mainly be working with two files:

1) Arbitrage.java
2) WeightedDigraph.java

1) Arbitrage.java contains a single method that you will have to implement, arbitrageOpportunity()

2) WeightedDigraph.java is an directed-graph implementation of the WeightedGraph class. You shouldn't have to modify any part of this file, but you will be using it to create a graph as part of the arbitrage question.

Arbitrage is the process of using discrepancies in currency exchange values to earn a profit. Whenever the prices of currencies change relative to each other, traders jump at the opportunity to look for these small discrepancies and make a profit. For example, consider three different currencies: the US Dollar (USD), Canadian Dollar (CAD), and Chinese Yuan (CNY). Each currency has a certain value relative to the other currencies. For example, suppose we have the following (entirely fake) exchange rates:

```
    1 USD = 2 CAD    <->  1 CAD = 0.5 USD
    1 USD = 2.5 CNY  <->  1 CNY = 0.4 USD
    1 CAD = 1.5 CNY  <->  1 CNY = 0.667 CAD
```

Is there any way to trade currencies and make a profit? In this case, the answer is yes: Starting with 1 USD, we can take the following "path":

1 USD -> 1*(2 CAD/USD) = 2 CAD
2 CAD -> 2*(1.5 CNY/CAD) = 3 CNY
3 CNY -> 3*(0.4 CNY/USD) = 1.2 USD

(1 USD) * 2 * 1.5 * 0.4 = (1.2 USD)

We can see here that by following that path of trades, multiplying each exchange rate, we end up with more money than we started with! However, here's an updated exchange rate example:

```
    1 USD = 2.0 CAD  <->  1 CAD = 0.5 USD
    1 USD = 3.0 CNY  <->  1 CNY = 0.333 USD
    1 CAD = 1.5 CNY  <->  1 CNY = 0.667 CAD
```

In this example, there are no opportunities for arbitrage. By starting with 1 amount of any currency, any sequence of transactions will eventually lead you back to 1 amount of that same currency.

There is a relationship here between following a path from 1 USD back to USD through different currencies, and detecting negative cycles in a graph. You have already seen a method of detecting negative-weight cycles in a graph using the Floyd-Warshall algorithm.

Even so, they aren't quite the same. How could we tell that there was an arbitrage opportunity from the exchange rates above? Starting at USD (although you can start from anywhere in the cycle), we "multiplied" the exchange rates: (2 CAD/USD) * (1.5 CNY/CAD) * (0.4 USD/CNY) = 1.2. Since this value was larger than one, we know that starting at 1 USD, we can end up with 1.2 USD by following this cycle. In a normal cycle, we consider it negative if the edge weights in the cycle "sum" to a number less than zero. How might you choose to create a graph from these exchange rates, so that it contains a negative cycle only when an arbitrage opportunity is possible?

HINT: Think carefully about what edge weights you choose to use for your graph. Are there any mathematical functions that can help with this idea of converting from multiplication to addition?

Your must implement the static function "List<Integer> arbitrageOpportunity(String filename)", that takes in a file with exchange-rate information and any "cyclical sequence" of currency trades that results in arbitrage. For example, an arbitrage opportunity is available by trading from USD to CAD, CAD to CNY, and CNY back to USD. Note that in the homework, different "country currencies" will be represented as integers (i.e. USD = 1, CAD = 2, and CNY = 3). In this case, outputting any of the following lists would be correct:

1 -> 2 -> 3 -> 1
2 -> 3 -> 1 -> 2
3 -> 1 -> 2 -> 3

In the second example, no arbitrage is possible. Your function should an Empty List in this case, indicating that the best we can do is to not lose any money, by executing no trades.

We can represent this exchange rate information as a matrix, where element (i,j) gives the exchange rate from currency i to currency j. Using the example above, we could represent these exchange rates with the matrix

$$\begin{array}{cccc} & \text{USD} & \text{CAD} & \text{CNY} \\ \text{USD} & 1.000 & 2.000 & 3.000 \\ \text{CAD} & 0.500 & 1.000 & 1.500 \\ \text{CNY} & 0.333 & 0.667 & 1.000 \end{array}$$

Here is another example where no arbitrage is possible:

$$\begin{array}{cccc} & \text{USD} & \text{CAD} & \text{CNY} \\ \text{USD} & 0.500 & 0.500 & 0.500 \\ \text{CAD} & 0.500 & 0.500 & 0.500 \\ \text{CNY} & 0.500 & 0.500 & 0.500 \end{array}$$

In this (less sensible) example, trading any one currency (ex. CAD) for any other currency (ex. CNY) will net you half of your previous currency. Clearly in this case, you can only lose money from trading any one currency for another. Again, in this case, you should output an empty list.

Finally, looking at the function itself. We've done the work for you of parsing a file into the 2d array explained above.

Implement the rest of the function, arbitrageOpportunity(). First you should read in the data from the 2d array to create a graph. Next, you should use cycle detection on the graph to output whether or not an arbitrage opportunity is possible.

**Part 2: Minimum Spanning Trees**

The next question is more theoretical, and comes in two parts:

Given an undirected graph G(V, E), where edges have weights. The actually weights of the edges are not known, however; instead, each edge is classified as either Light, Medium or Heavy.

All Light edges have a smaller weight than any Medium or Heavy edge.

All Medium edges have a smaller weight than any Heavy edge

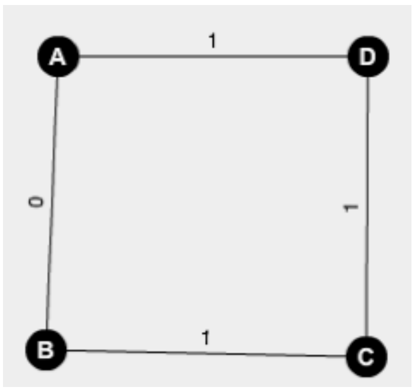
In general, nothing is known about the relationship between two edges in the same weight class

Your goal is to output:

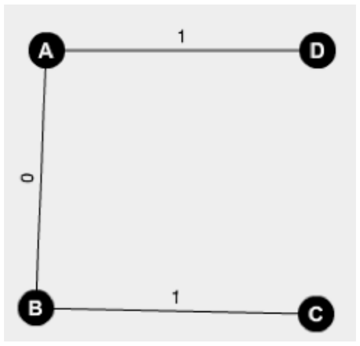
2a) All edges that must exist in every minimum spanning tree of G

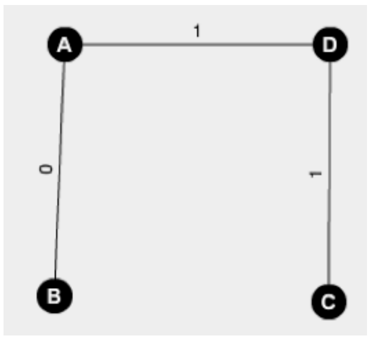
2b) All edges that must "not" exist in any minimum spanning tree of G

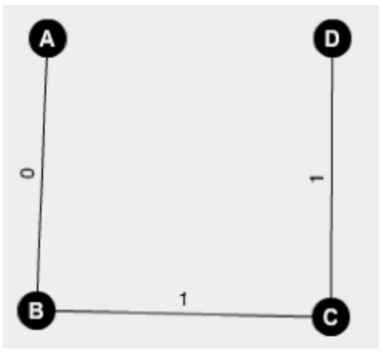
For example, take this square graph below (let 0 represent Light, 1 represent Medium and 2 represent Heavy edges).



There are 3 possible minimum spanning trees for this graph:







Notice that the single light edge in the graph is in all three of these trees. The other 3 edges, however, show up in some but not all of the possible MSTs for this graph. Finally, out of the 4 edges, all of them appear in at least one of the MSTs. Thus, we would output

1) (a,b) as the only edge that must exist in every MST of G
2) no edges that must not exist in any MST of G

You should take some time to think about the two parts of this problem. Some tips and ideas to get you started can be found at the beginning of each of the two sections below. But first, we will quickly review the files that you will be using:

1) MST.java
2) WeightedUndirectedGraph.java
3) Tuple.java

1) MST.java contains two functions that you will implement, mustExist() and mustNotExist(). These are the two functions that you will be implementing. They should output the edges that must exist in every MST of a graph, and must not exist in any MST of a graph, respectively.

2) WeightedUndirectedGraph.java is an undirected-graph implementation of a weighted graph. This will be the input to both of your functions. Remember that you can use the methods from the graph class, explained above (i.e. neighbors() to get the neighbors of a vertex, and weight(), to get the weight of an edge).

3) Tuple.java is a simple class that will be used to represent edges. It consists of two ints, representing the two vertices of the edge. As both of your MST functions need to return a set of edges, we will represent this by returning a set of Tuples. As a quick example, to create a Tuple:

```
Tuple t = new Tuple(2,3);
t.u; //2
t.v; //3
```

Now, stepping away from the code and back to the problem:

**Part 2a: Edges that must exist in every MST (we will call them "necessary edges")**

First, try to come up with an algorithm for identifying the edges that must exist in every MST of a graph. The most efficient answer that will full credit runs in O(n+m) time.

Some general hints and tips are provided below, but you should take some time to think about the problem and how it relates to some of the algorithms that you've seen from the lecture recordings. Draw out some examples and identify the necessary edges in the graph. Why do they show up in every MST, and not just some of them?

Tip 1: What does it mean for an edge to be a necessary edge? What would happen to the graph if it were removed?

Tip 2: It might help to first consider how to find necessary edges in a graph without weights (see the third tip). Afterwards, try to expand your solution to deal with edges that are either light, medium or heavy. Identify a light edge that is a necessary edge. What would happen if it were a different weight? Is there anything that would prevent it from being a necessary edge?

Tip 3: An important hint for this problem is the articulation point algorithm from the lecture recordings. Rewatch the animation of that algorithm if you aren't fully comfortable with it. Afterwards, try to look for commonalities in that algorithm and this problem. The algorithm identifies vertices which, upon removal from the graph, will disconnect it. Unfortunately, the algorithm as it is doesn't identify the edges you are looking for- it only identifies vertices. Is there any relationship between an articulation point and a necessary edge? Recall what the low[] array in the algorithm represents: low[v] represents the earliest vertex reachable by any vertex in the subtree of v in the dfs traversal. Try walking through the algorithm on one of your sample graphs. Can you identify any commonalities between necessary edges and this low[] array?

Finally, fill in the first function in MST.java:
Set<Tuple> mustExist(WeightedUndirectedGraph<Weight> g).
	-Notice that you are outputting a set of tuples, which represent edges
	-(see the explanation at the beginning of the homework)
	-We've provided you with a Weight enum, which you can find at the bottom of MST.java. It is a simple enum with 3 possible values (light, medium, heavy)

**Part 2b: Edges that are not in any MST**

As with the last part of the question, you should first take some time to come up with an algorithm. The most efficient answer that will get full credit runs in O(mlgn) time. Some general hints are once again provided below.

To get you started, recall the Cycle Property from the lecture recordings. Here is a slight variant:

Let e be an edge of uniquely maximum weight in a cycle C of G. Then no MST for G contains e.

You should prove this for yourself. In this case, because there are no other edges in the cycle with the same weight as e (e is uniquely maximum), then we get the stronger claim that no MST for G can contain this edge e.

Tip 1: Recall Kruskal's algorithm from the lecture recordings. It sorts the edges in increasing order of weight, and builds an MST by only adding edges that will not create a cycle (by using union-find). The edges that kruskal's algorithm rejects fall into two categories:

1) They might exist in some other MST for G (your algorithm should not output these edges)

2) They don't exist in any MST for G (your algorithm should output these edges)

Is there a way to identify these edges in the algorithm?

Tip 2: It might be helpful to consider the following: Suppose that there are multiple edges of weight k in the graph. How does the order in which kruskal's algorithm process these edges affect the MST that it outputs?

Finally, fill in the second function in MST.java:

Set<Tuple> mustNotExist(WeightedUndirectedGraph<Weight> g)!