CSC 301 Project 3: Arbitrage Spring 2023

Due: Friday, March 17th

In this project, you will implement Bellman-Ford to detect arbitrage opportunities given a set of exchange rates between currencies. You will be required to implement your solutions in their respective locations in the provided `project3.cpp` and `project3.hpp` files.

Contents

1 The Arbitrage Problem

1.1 Arbitrage

Arbitrage opportunities occur in finance whenever there is a small mismatch in the prices of goods. These mismatches occur because the "price" of a good is determined by the value at which buyers are willing to buy and sellers are willing to sell, the buyers and sellers don't have perfect communication, and the actual price of the good changes every time any buyer/seller pair reach an agreement. The global market is so massive that these sorts of mismatched prices are occurring almost constantly.

Whenever there is a mismatch in prices, it can create an opportunity for what is essentially "free money" to the person who makes a particular trade or set of trades that leverages the mismatches prices. Consider the following example of currency exchange rates.

Say that:

- 1 USD buys 0.82 Euro,
- 1 Euro buys 129.7 Yen,
- 1 Yen buys 12 Turkish Lira,
- 1 Turkish Lira buys 0.0008 USD

Then if we take 1 USD, convert it to Euros, convert that to Yen, convert that to Lira, and convert that back to USD, we will get:

$$0.82 \cdot 129.7 \cdot 12 \cdot 0.0008 \approx 1.02 \text{ USD},$$

which results in a 2% profit!

Now, of course, this is a contrived example with a rather massive payoff. But while real-world arbitrage opportunities are significantly smaller (fractions of pennies for each arbitrage trade), millions can occur each day, and the money can add up in the long run.

One thing to note about these arbitrage opportunities: when you make the cycle of trades to take advantage of the arbitrage opportunity, you realign the prices and the opportunity disappears. This is actually a critical role of arbitrage in the marketplace: algo-trading arbitrage helps keep prices aligned for the rest of us, so when we go to the currency exchange, we know what our money is worth!

Of course, the practical side of this is that only the first person to make the trades will get the money. So in real life, it's a race!

1.2 Formal Statement

Now that we have a sense of what arbitrage is, let's give it a more formal definition. Let c_i be our various currencies, and let one unit of currency c_i buy $R(c_i, c_j)$ units of currency c_j (e.g., 1 USD buys 0.82 Euros would mean $R(\text{USD,Euro}) = 0.82$).

Our task is to find a cycle of currencies c_1, c_2, c_3, ..., c_k such that

$$R(c_1, c_2) \cdot R(c_2, c_3) \cdot \cdots \cdot R(c_k, c_1) > 1.$$

1.3 Negative Cost Cycles

So we now have a question: given a set of currencies and their exchange rates, how can we detect arbitrage opportunities?

One approach is to encode arbitrage it as a graph problem. Let's define a graph where:

- Every currency in the exchange becomes a vertex in the graph.
- Directed edges will reflect the exchange rates between pairs of currencies.

Now, we are looking for a cycle of currencies where the product of the exchange rates is greater than 1. One interesting thing we saw in class was that Bellman-Ford had the ability to detect a special type of cycle in a directed, weighted graph: a negative cost cycle where the sum of the edge weights is negative.

So, how can we turn a problem about a product into a problem about a sum?

Logarithms.

The original problem is to find a cycle of currencies c_1, c_2, c_3, ..., c_k such that

$$R(c_1, c_2) \cdot R(c_2, c_3) \cdot \cdots \cdot R(c_k, c_1) > 1.$$

Note that this is equivalent to the condition that

$$\frac{1}{R(c_1, c_2)} \cdot \frac{1}{R(c_2, c_3)} \cdot \cdots \cdot \frac{1}{R(c_k, c_1)} < 1.$$

Now we can take the log of both sides to get

$$\log \frac{1}{R(c_1, c_2)} + \log \frac{1}{R(c_2, c_3)} + \cdots + \log \frac{1}{R(c_k, c_1)} < 0.$$

So we are looking for a set of currencies such that the sum of these log values is negative. We'll use these log values as the edge weights in our graph so that an arbitrage opportunity is a negative cost cycle!

In our graph, let the edge weight between vertices u and v be given by

$$\texttt{length(u,v)} = \log \frac{1}{R(u, v)} = -\log R(u, v).$$

If we create such a graph, we can run Bellman-Ford to detect the negative cost cycle. That cycle will be our arbitrage trade!

2 Problem Statement

You will have two primary tasks in this project:

1. Alter the function `createAdjacencyMatrix` so that it correctly creates the adjacency matrix given the exchange rates.

2. Implement Bellman-Ford, to detect and report a negative cost cycle in the currency graph, in the function `detectArbitrage`.

2.1 `createAdjacencyMatrix`

Your task is to write a function that will take in a matrix of the exchange rates along with a `vector` of currency labels, and will return the corresponding adjacency matrix for solving the arbitrage problem. The current placeholder simply copies the exchange rates. The function `log10` may be useful here.

2.1.1 Representing Matrices

You will notice that all matrices for this project are represented as 1D `vector`s. This is actually a fairly common representation of a matrix. The idea is that if you need to store an $n \times m$ matrix, you can simply map the elements to an array of size $n * m$. There are two common forms for this mapping:

- Row-major form: where each row of the matrix is kept consecutive in memory. Here is an example of how a 3x4 matrix would map to a 1d array using row-major form:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \longrightarrow [a\ b\ c\ d \quad e\ f\ g\ h \quad i\ j\ k\ l]\,.$$

- Column-major form: where each column is kept consecutive in memory. Here is an example of how a 3x4 matrix would map to a 1d array using column-major form:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix} \longrightarrow [a\ e\ i \quad b\ f\ j \quad c\ g\ k \quad d\ h\ l]\,.$$

We will be using **Row-Major Form** for this class.

Note that there is a relationship between the row/column indices in the matrix and the index in the array.

- Idea 1: to get to the row r (indexing from 0), you have to pass r rows. Each of those rows has m columns in an $n \times m$ matrix (recall that we write $row \times column$ for matrix dimensions). So you have to pass $r * m$ elements to get to row r.

- Idea 2: to get to column c in a given row, you need to pass c elements within that row.

Putting these together, to get to (row r, column c) you have to pass $r * m + c$ elements.

So $r * m + c$ is the index of the element in row r and column c.

Note: since our matrices in this project are square, $m = n$. In this case, it is common convention to write the index as $r * n + c$ since it is an $n \times n$ matrix.

2.2 `detectArbitrage`

The function `detectArbitrage` will comprise the bulk of your work for this project. It will output a single `vector` of vertex **labels** (`int`s corresponding to their index in the adjacency matrix) corresponding to the negative cost cycle. This vector needs to start and end at the same label. This function will take 3 inputs:

- `adjMatrix`: the adjacency matrix, as generated by the `createAdjacencyMatrix` function.
- `currencies`: the `vector` of currency labels.
- `tol`: this is a value that is set at `1e-15` as default, and **should not be altered.**

2.2.1 Representing the Graph

Note that the `detectArbitrage` function is only given an adjacency matrix and a list of vertex labels (`currencies` which is stored as strings), where the label at index k is the currency at row/column k in the matrix. There are no `Vertex` objects and there is not an adjacency list.

So how will you store info about the `vertex.distance`, `vertex.previous`, `length(u,v)` values?

- You have been given a `vector` called `distances` that will store the distance info. A vertex's index in this vector corresponds with its index in the adjacency matrix and its index in the currency list. The distance values have all been initialized to infinity for you.
- You have been given a `vector` called `previous` that will store the info about the previous values. Consider the vertices at indices i and j in the adjacency matrix. If we wanted vertex i to have its previous value set to point at vertex j, we would set the ith entry of the `previous` array equal to j. These values have all been initialized to `-1` for you.
- The edge lengths are in the adjacency matrix!

2.2.2 Tolerance and Machine Epsilon

The value `tol` will be used to deal with one very important problem. Consider what happens when there is an exchange rate where 1 Yen is equal to 12 Lira. Then clearly 1 Lira is equal to 1/12 Yen. But how is this represented on the computer? This is an infinitely repeating decimal, and so we must truncate its value. But this means that, on the computer,

$$R(\texttt{Yen,Lira}) * R(\texttt{Lira,Yen}) \neq 1 \quad \Rightarrow \quad -\log[R(\texttt{Yen,Lira})] - \log[R(\texttt{Lira,Yen})] \neq 0,$$

because there will be an error in the smallest digit. We are promised one important fact about the size of this error: it is smaller than the value known as machine epsilon (the smallest representable number using the given number of bits). For a `double`, machine epsilon is about `2.2e-16`. This means that we **cannot trust** any updates of a size on the order of `1e-16`. Therefore, we will ignore any updates that are smaller than our tolerance value, `tol=1e-15`. This will change the Bellman-Ford implementation very slightly. When we make an offer during an update step, if the update is smaller than `tol` we ignore it. So, if we originally had the code:

```
for each neighbor of u:
    if neighbor.dist > u.dist + length(u, neigh):
        neighbor.dist = u.dist + length(u, neigh)
        neighbor.prev = u
```

We will now have the code:

```
for each neighbor of u:
    if neighbor.dist > u.dist + length(u, neigh) + tol:
        neighbor.dist = u.dist + length(u, neigh)
        neighbor.prev = u
```

Your implementation of the Bellman-Ford algorithm must include this change!

2.2.3 Tasks for `detectArbitrage`

Your implementation of `detectArbitrage` will have to perform 4 major tasks:

1. Perform the $|V| - 1$ iterations of Bellman-Ford, taking the `tol` value into account.
2. Perform the extra iteration and track changes in the `distance` values.
3. Choose a single vertex that had a change and follow its path backwards (using the `previous` values) until you find a cycle. **Note** that this cycle does not necessarily include the changed vertex you started with.
4. Once you have this path, remove any vertices that are not part of the cycle, and make sure that the vector is in the correct order (hint: you traced the path backwards). This cycle will be your return value.

Note: this process does not guarantee that you will find the best negative cost cycle to choose. However, I am only asking that you find any arbitrage opportunity. *You do not need to find the best arbitrage opportunity.*

3 Provided Code

To aid you in these tasks, you have been provided with several functions for testing your code and printing the currencies and exchange rates. These functions can be found in `arbitrage.cpp` and `arbitrage.hpp`.

3.1 `getRates`

The function `getRates` takes in 0, 1, 2, or 3 (defaulting to 0) and outputs the exchange rates for each scenario:

0. The four currency arbitrage example from earlier in this document.
1. A set of actual exchange rates between 14 currencies as of 11/12/18.
   EUR = Euro, GBP = British Pound, CHF = Swiss Franc, USD = US Dollar,
   AUD = Australian Dollar, CAD = Canadian Dollar, HKD = Hong Kong Dollar,
   INR = Indian Rupee, JPY = Japanese Yen, SAR = Saudi Riyal,
   SGD = Singapore Dollar, ZAR = South African Rand, SEK = Swedish Krona,
   AED = U.A.E. Dirham
2. The same set of exchange rates, but with the US Dollar underpriced with respect to the British Pound.
3. The same set of exchange rates, but with the US Dollar underpriced with respect to the British Pound, the Japanese Yen overpriced with respect to the Indian Rupee, and the Saudi Riyal overpriced with respect to the Hong Kong Dollar.

Note: when your code is tested on these scenarios, you should expect to detect arbitrage in every case except 1, the real world exchange rates. Remember that real world arbitrage keeps the prices properly aligned for the typical user.

3.2 `testExchange`

This function will directly test your code. It takes in an `exchangeNum` to select from the options provided by `getRates`, and obtains the rates and currency labels. It then creates the adjacency matrix using your `createAdjacencyMatrix` function, and looks for a negative cost cycle using your `detectArbitrage` function. It checks for correctness of the cycle (comparing against the expected result input as the boolean `cycleExpected`), and will print the results of the test if the input `verbosity` flag is set. This function is called by the `arbitrage` main function, which runs all of the various tests.

3.3 `printCurrencyMatrix`

This function is provided for your use in debugging if needed. Given an $n \times n$ matrix and a correspond n-length vector of labels, it will cleanly print the labels and matrix in a readable format. Note: there are other inputs that control the actual printing. This is very finicky and the default values were chosen to print all scenarios from this project. You should just rely on those default values by calling the function with something like: `printCurrencyMatrix(rates,currencies)`.

3.4 `main`

The provided main function tests your code on all of the four scenarios and prints the results. You should feel free to edit your local version of the main function as you write your code, though you should ensure your code can pass all four original tests before submitting.

4 Submission

You **must** submit your `project3.cpp` and `project3.hpp` code online on gradescope. If you worked with a partner, only submit one version of your completed project, associate both partners with the submission, and indicate clearly the names of both partners at the top of all submitted files. Attribute help in the header of your `.cpp`.

5 Pair Programming

You are allowed to work in pairs for this project. If you elect to work with a partner:

- You should submit only one version of your final code and report. Have one partner upload the code and report through gradescope and associate both partners with the submission. Make sure that both partner's names are clearly indicated at the top of all files.
- When work is being done on this project, both partners are required to be physically present at the machine in question, with one partner typing ('driving') and the other partner watching ('navigating').
- You should split your time equally between driving and navigating to ensure that both partners spend time coding.
- You are allowed to discuss this project with other students in the class, but the code you submit must be written by you and your partner alone.

6 Readable Code

You are expected to produce high quality code for this project, so the code you turn in must be well commented and readable. A reasonable user ought to be able to read through your provided code and be able to understand what it is you have done, and how your functions work.

The guidelines for style in this project:

- Your program file should have a header stating the name of the program, the author(s), and the date.
- All functions need a comment stating: the name of the function, what the function does, what the inputs are, and what the outputs are.
- Every major block of code should have a comment explaining what the block of code is doing. This need not be every line, and it is left to your discretion to determine how frequently you place these comments. However, you should have enough comments to clearly explain the behavior of your code.
- Please limit yourself to 80 characters in a line of code.