Math 560 Project 3: Arbitrage Spring 2024

Due: Wednesday, April 10th

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.py` file.

Contents

1 Problem Statement

You will have two primary tasks in this project:

1. Alter the function `rates2mat` in `project3.py` 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` from `project3.py`.

1.1 `rates2mat`

You have been provided with a placeholder code that currently just copies each element from the rates matrix directly into the returned adjacency matrix. With only a very minor modification to this single line of code, you should be able to return the adjacency matrix with the correctly weighted edges. The function `math.log` may be useful here.

1.2 `detectArbitrage`

The function `detectArbitrage` will comprise the bulk of your work for this project. It will output a single list of vertex **ranks** corresponding to the negative cost cycle. This list needs to start and end at the same rank. This function will take 3 inputs:

- `adjList`: the adjacency list representing the currencies graph.
- `adjMat`: the adjacency matrix representing the exchange rates, as generated by the `rates2mat` function.
- `tol`: this is a value that is set at `1e-15` as default, and **should not be altered.**

1.2.1 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 python's `float` type, 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, we originally had the code:

```
# Check each neighbor of u.
# Update predictions and previous vertex.
for neigh of u:
    # Only update if the new value is better!
    if neigh.dist > u.dist + length(u, neigh):
        neigh.dist = u.dist + length(u, neigh)
        neigh.prev = u
```

But now we will have the code:

```
# Check each neighbor of u.
# Update predictions and previous vertex.
for neigh of u:
    # Only update if the new value is better!
    if neigh.dist > u.dist + length(u, neigh) + tol:
        neigh.dist = u.dist + length(u, neigh)
        neigh.prev = u
```

Your implementation of the Bellman-Ford algorithm will have to include this change.

1.2.2 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 `vertex.dist` values.
3. Choose a single vertex that had a change and follow its path backwards (using the `vertex.prev` 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 list 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.*

2 Provided Code

To aid you in these tasks, you have been provided with several fully functioning classes and functions for creating and testing the exchange rates.

2.1 Vertex Class (`p3vertex.py`)

The first of the fully functioning classes is the Vertex class. This class has 4 class attributes:

- rank: the rank (label) of the given vertex
- neigh: the list of the neighboring vertices
- dist: the distance from the start vertex
- prev: the previous vertex in the path

Along with these are three implemented member functions:

- `__init__`: this is the constructor function for the Vertex class. It requires an input rank for the vertex, and sets all of the attributes to have reasonable starting values. You will create a new Vertex with a call: `v = Vertex(rank)`.
- `__repr__`: this function is called whenever a Vertex is printed, i.e. when the call `print(v)` is made. It simply prints the rank of the vertex.
- `isEqual`: this takes in a second Vertex as an input, and compares the rank of the two vertices, returning True if they are equal rank (i.e., if they had the same label). This function can be called using: `v.isEqual(u)`.

2.2 Currencies Class (`p3currencies.py`)

You have also been provided with a fully functioning Currencies class with 5 class attributes:

- rates: a 2D array representing the exchange rates
- currs: a list of the currency names as strings
- adjList: the adjacency list of Vertex objects
- adjMat: the adjacency matrix (stored as a 2D list)

- negCyc: what will ultimately contain the negative cost cycle, stored as a list of ranks (not a list of vertices)

The Currencies class has 6 member functions:

- `__init__`: this is the constructor for the Currencies class. It has one optional input: the exchangeNum which selects which set of exchange rates to use (options: 0,1,2,3 - default: 0). This initialization function correctly creates the adjacency list. The negCyc attribute is initialized as an empty list. A new Currencies object can be created with the call `c = Currencies(exchangeNum)`.

  NOTE: the adjacency matrix is created using the `rates2mat` function. You will have to correctly implement this function (described above).

- `__repr__`: this function is called when a Currencies object is printed. It will simply print all of the exchange rates.

- `printList`: this function can be used to aid with debugging. It prints the adjacency list in a more readable format.

- `printMat`: this function can be used to aid with debugging. It prints the adjacency matrix in a more readable format.

- `printArb`: this function is used to print the currencies listed in the negative cycle stored in negCyc.

- `arbitrage`: this function calls the function `detectArbitrage` on the Currencies to obtain the potential negative cost cycle. You will be responsible for implementing the `detectArbitrage` function (described above). It will then check to ensure that the reported arbitrage (if one was reported) was successful: that it was a cycle where arbitrage occurred. If the arbitrage was successful, it will report the monetary gain per unit input.

2.3 `getRates` and `testRates`

You have also been provided with two functions that create the Currencies graph (adjacency list and matrix) and test your code. The function `getRates` can be found in `p3currencies.py`, while `testRates` is in `p3tests.py`.

- `getRates`: takes in 0, 1, 2, or 3 and outputs the exchange rates for each scenario:

  0. The small arbitrage example from class.
  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.

- `testRates`: this function will test your code on the entire set of exchange rates.

3 Submission

You **must** submit your `project3.py` code online on Gradescope. If you worked with a partner, only submit one version of your completed project and indicate clearly the names and NetIDs of both partners.

4 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. Have one partner upload the code on their Sakai site. Make sure that both partner's names are clearly indicated at the top of the code.
- 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.

5 Style Points

Part of your grade for this project will be 'style points'. The idea here is that the code you turn in must be well commented and readable. A reasonable user aught 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. This means that a grader should be able to read over your code and tell that your algorithms are implemented correctly.

The guidelines for these 'style points':

- 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. In python, you can use the symbol \ to indicate a line break/continuation.