IMPERIAL COLLEGE LONDON, DEPARTMENT OF MATHEMATICS

Scientific Computing (M3SC)

Francesca Sogaro

February 1, 2017

1 ARBITRAGE - INTRODUCTION

The Bellman-Ford algorithm is routinely applied in financial engineering to search for currency arbitrage opportunities. Arbitrage opportunities are investment strategies that promise a positive return without assuming any underlying risk. These opportunities appear rather rarely, but are of course highly lucrative as they provide a guaranteed return on investment (although rather small). Arbitrage opportunities do not prevail very long; once exploited, they commonly vanish. For example, assume we have the following (unrealistic) currency exchange rates between the US dollar and the British pound:

$$\begin{aligned} GBP &= w_1 US \\ US &= w_2 GBP \end{aligned} \tag{1.1}$$

Where $w_1 = 1.25$ and $w_2 = 0.81$. After changing one dollar to 1.25 pounds and then changing the pounds back to dollars, we end up with 1.01 dollars, i.e., we made one percent return on our dollar without any associated risk. Doing this over again and with far larger amounts, we are in a position to achieve a positive return on capital with no risk. In an arbitrage case, we are looking for a product $w_1 * w_2 * ... > 1$. This tutorial will use the BellmanFord routine to identify lucrative loops among currency exchange rates.

The following tasks will guide you through developing the code:

- Import the currency exchange data such that the code is provided with the adjacency matrix

- Run the BellmanFord routine and use it to identify the presence of cycles
- Devise a method to find the currencies that are part of a lucrative loop
- What profit can you make?

2 TASKS

On blackboard there will be different .txt files containing various examples of adjacency matrices that you can test your routine on. Download these files into your working directory but until you actually have completed your code and are sure it works, play only with the data in the 'Data1.txt' file.

2.1 TASK1:IMPORTING THE DATA

Complete the small subroutine called 'ImportData' such that the exchange rates can be loaded from the text files into python. This should take no longer than 5 minutes
The function should take the name of your file (e.g. 'Data1.txt'), and return a $list$ called 'data'. Recall that the file to import should be stored in the same folder as the code, otherwise you should provide your function with the path to the file. Also the input to your function ('filename') should be a string. You might want to use the module 'csv', which is already imported in the function, as it easily deals with comma separated values. There are other ways to do this, so feel free to explore.

```
1 def ImportData(filename):
2     import csv
3     data=[]
4     ##########################################
5     #   complete here PART 1 (about 4-8 lines)
6     ##########################################
7     return data
```

2.2 TASK 2: PREPARING THE ADJACENCY MATRIX

Once you have loaded the data, there is still one step missing before calling 'Bellman-Ford'. This is a small part that you have to complete in the main section of the file 'Arbitrage.py'. You should recall that the algorithm is capable of handling $negative$ cycles. Hence think of a way to turn the current exchange matrix into an equivalent matrix that would classify a profitable loop ($w_1 * w_2 * w_3 ... > 1$) into a negative one ($e_1 + e_2 + e_3 ... < 0$). Come and see us if you are still struggling with this after 5-10 minutes.

```
1
2  if __name__ == '__main__':
```

```
    # load data
    filename= 'Data1.txt'
    data=ImportData(filename)
    # prepare for the BellmanFord routine

    ##########################################
    #   complete here PART 4 (about 1-2 lines)
    wei='something'#modify apporopiately
    ##########################################
    #call bellmanford routine on the data
    paths = BellmanFord(wei)
```

2.3 Task 3: Identifying negative cycles

In this task we want you to think on how the BellmanFord algorithm actually can detect negative cycles. The algorithm is run $V-1$ times. What happens if you run it one more time? Are the distances updated? Are the parents updated? Can we use this result to identify vertices that are part of a cycle?

In this section you should:

- Define the **boolean** 'val' which assumes a value of either 'true' or 'false' such that the following if statement is executed only if there are negative cycles in the wei matrix.
- Define the **list** 'g' such that it contains all the vertices that are part of a negative cycle, you can see that the list 'g' will be then used in the section of the code which identifies these cycles.

```
    ######################################################
    #    complete here   PART 3 (about 2-4 lines)
    # step 4: are there negative cycles?
    #         which vertices are interested?

    val= True # change appropriately
    g=[1,2,3] # change appropriately
    ######################################################
# step 5: identify those loops
    if val:
        print 'graph contains a negative-weight cycle'
        for i in g:# g is a list, contains the vertices of
                                #       the negative cycle
            print 'this section should identify the loops'
```

2.4 Task 4: Finding the actual loops and calculating the arbitrage associated to them

Once you found the vertices that are part of a cycle, can you devise a way to check what loop the vertices are part of? What if they are part of multiple loops?
You are asked to complete the 4th missing part such that you can successfully identify the arbitrage opportunities in the current data. This section is a bit more challenging.

```
# step 5: identify those loops
    if val:
        print 'graph contains a negative-weight cycle'
        for i in g:# g is a list, contains the vertices of
                                #       the negative cycle
            print 'this section should identify the loops'
        ########################################################
        #   complete here PART 4
        ########################################################

    return  npaths # routine can also return other lists
```

You might find the routine 'Weights_vals' useful for your code. What paths would you use this routine for?

2.5 Task 4: display the output

Lastly, print out the cycles and the profit you can make from them. Which one is the most lucrative? Make your code print out the opportunities in the main part.

2.6 Optional Tasks

Food for thought:

- Can you think of a way to recover cycles by simply storing the evolution of the BellmanFord algorithm?
- The identification of arbitrage opportunities in this exercise was done on currencies exchange rates. Generally you can always convert any currency to any other. There are cases where, for example, you con only get to a point P in your graph by passing through the point L (like in the example in class, not all the vertices are connected to one an other). Would your routine work in this case? What modification should you implement?