Final Project

DS730

In the final project you will be working with all of the technologies you have learned in this course. Problem 1 must be solved using one of the high performance tools learned in this class[[1]](#footnote-1): Hadoop MapReduce, Pig, Hive or Spark. You do not need to choose 1 tool for all problems nor do you need to use all tools. You can mix and match them for problem 1 in any way that you want. Problem 2 must be solved using Java or Python threads. **Note that a solution written without using a tool learned in this course will not earn any points[[2]](#footnote-2).** If you are unsure if the way you solved them meets the requirements of this project, ask first before submitting your solution.

If there is a tie for any of the questions, you should print out all of the correct answers.

1. (63 pts) – See Hadoop Exercises.
2. (42 pts) The solution to this problem must be done using **Java threads or Python threads**. Your Runner class should be stored in a file called **FinalRunner.java** or **FinalRunner.py**. You will likely need several other classes. Your main method must be in the FinalRunner file.

Due to budget cutbacks, the postal services at UW-Oshkosh can only afford 1 mail deliverer. Even worse, that deliverer is a student who works part-time. Postal services wants to minimize the amount of time that student has to work in order to save money. Because of this, they are interested in the fastest way to visit all buildings and return back to the Campus Services Building, BldgOne in the example below. There are obvious routes that are terrible (e.g. going from one side of campus to the other and then back) but the optimal route is not obvious. Your goal is to read in a file that gives the time in seconds to get from a building to every other building and determine the best possible route such that you start at the building listed on the first line, visit all other buildings and end at the building listed on the first line. The building names in the example below are arbitrary and can be called anything. The input file you will read in is called **input2.txt** and will be formatted in the following manner:

BldgOne : t(BldgOne) t(BldgTwo) t(BldgThree) t(BldgFour) t(BldgFive)  
BldgTwo : t(BldgOne) t(BldgTwo) t(BldgThree) t(BldgFour) t(BldgFive)  
BldgThree : t(BldgOne) t(BldgTwo) t(BldgThree) t(BldgFour) t(BldgFive)  
BldgFour : t(BldgOne) t(BldgTwo) t(BldgThree) t(BldgFour) t(BldgFive)  
BldgFive : t(BldgOne) t(BldgTwo) t(BldgThree) t(BldgFour) t(BldgFive)  
  
Take the first line for example. t(BldgTwo) will be an integer value denoting the number of seconds it takes to get from BldgOne to BldgTwo. On the first line, t(BldgOne) will be 0. In other words, it takes 0 time to get from BldgOne to BldgOne. The input will always be formatted in this manner. If another building is constructed, it will be added to the end and the file will be updated accordingly. For example, if BldgSix were constructed, the time to BldgSix will be added at the end of every list and BldgSix will be added to the end of the file. The time from BldgOne to BldgThree may not be the same as the time from BldgThree to BldgOne. There may be one way streets; it may be uphill, etc. A sample input file is shown below:  
  
abc : 0 5 7 3  
def : 4 0 3 6  
ghi : 6 4 0 4

jkl : 4 5 6 0  
  
Each row in the sample input will start with the building ID[[3]](#footnote-3), followed by a space, followed by a colon, followed by another space, then followed by a list of integer values to get to all of the other buildings where each value is separated by a space.   
  
Your goal is this, for the best route possible, print out the total time taken to start with the building on the first line, visit all buildings and then return to the building on the first line. **You must also output the order in which you visited the buildings using the name of the buildings as defined in the input file. You should also output the total route cost.** In the above example, there are only 6 possible routes. Each of them are shown below with their route cost and total cost at the end:

abc → def → ghi → jkl → abc : 5+3+4+4 = 16

abc → def → jkl → ghi → abc : 5+6+6+6 = 23

abc → jkl → def → ghi → abc : 3+5+3+6 = 17

abc → jkl → ghi → def → abc : 3+6+4+4 = 17

abc → ghi → def → jkl → abc : 7+4+6+4 = 21

abc → ghi → jkl → def → abc : 7+4+5+4 = 20

Your output file, which should be called **output2.txt** would contain the following answer:

abc def ghi jkl abc 16

A few constraints on your solution are given below:

* 1. In order to get from BldgA to BldgC, the direct path will always be the best path. In other words, if going from BldgA to BldgC, then going directly from BldgA to BldgC will always be faster than to go from BldgA to any other arbitrary building (say BldgB) to BldgC[[4]](#footnote-4).
  2. Your code must split up the computation in a reasonable fashion. Think about the solution space and how the potential solutions can be split up and checked in parallel.
  3. Your code must run faster than my brute force non-parallelized way of solving it. My brute force solution that uses no multithreading and is not optimized checks all potential solutions with 12 buildings. It runs in roughly 60 seconds.
  4. Because this problem blows up in complexity, your code only needs to work for up to 13 buildings. As a guideline[[5]](#footnote-5), my threaded code with 13 buildings takes ~3.5 minutes to run on a t2.2xlarge EC2 instance (8 CPUs) and it ran in ~12 minutes when using a t2.large EC2 instance (2 CPUs).
  5. Do not prompt the user or expect the user to enter in any arguments. The input file is **input2.txt** and the output file is **output2.txt.** Along the same lines, the number of buildings is an arbitrary size and should be determined by reading in the input file. Do not ask the user for this information. The **input2.txt** file should be in the same folder as your Java or Python files.

1. The only exception is that threading cannot be used for problem 1. [↑](#footnote-ref-1)
2. You can double check your work using Excel or whatever other tools you want. But your solution has to use the tools we learned in this course. [↑](#footnote-ref-2)
3. Building ID’s will not contain any whitespace but could be any other sequence of letters, numbers or special characters. [↑](#footnote-ref-3)
4. For those interested, this is called the triangle inequality. The input will not break this rule. You do not have to do anything to test for this as the input will adhere to this rule. [↑](#footnote-ref-4)
5. My threaded code is not optimized and checks every single possibility. You should see this as an upper bound on your runtime. It is possible to write optimal threaded code that solves 13 buildings in seconds. [↑](#footnote-ref-5)