Social Network Analysis – Homework 2
George Choumos - DS3616021

This is the report for the second assignment in Social Network Analysis. It consists of 2 parts, one for each of the problems.

Problem 1

The provided VM was used for this and the steps were followed:

- Opened up a terminal
- Used the su command to substitute to the **hduser**
- Started the hadoop filesystem and the mapreduce by triggering the corresponding scripts with the given commands.
- Verified that the required processes were running by executing the jps command.
- As **hduser**, I executed the given command and the output was generated in the */user/hduser/output/shortestpaths* folder.
- Following the instructions, the output file was retrieved and its name was switched to the task id that actually saved the vertices.
- This file for me is the ***task_201705061122_0001_m_000001.txt*** which can be found in the *exe_2_problem_1* folder.

Problem 2 + BONUS

General Comments for Problem 2

Regarding this problem and before diving into the details, here is an overview of the deliverables:

- All the files for the 2nd problem are in the *exe_2_problem_2* folder.
- In that folder there are
    - the ***GiraphAppRunner.java***
    - the ***SimpleBoggleComputation.java***
    - 3 boggle <u>input files</u>:
        - The one that was already provided – **boggle.txt**
        - The one that appears in the assignment's pdf – **boggle_example_on_assignment.txt**
        - The one that I created for the **<u>bonus part</u>** of the exercise – **boggle_bonus.txt**

    - 3 boggle <u>output files</u> which are the respective outputs of the input files
        - **output_boggle.txt**
        - **output_boggle_example_on_assignment.txt**
        - **output_boggle_bonus.txt**

Have in mind that the first output (output_boggle.txt) was generated by the given 3*3 graph using the tiny dictionary that was hardcoded and only included 5 words. The other 2 output files are using the enhanced dictionary instead. I didn't include it here but it is the one that appears in the assignment as a link.

Note that the enhanced dictionary is not included in the deliverables as I suppose you probably already have it so it is redundant.

Comments on the implementation of the 2 java classes

In SimpleBoggleComputation file which is the one that changed the most, there are sufficient comments that describe the thoughts behind the way that things are implemented, so maybe a read through it will be enough! However here are some more details.

GiraphAppRunner

- Imported the SimpleBoggleComputation
- Imported BoggleInputFormat
- The input and the output path were changed accordingly on each exection. The submitted file includes the values of my last execution.
- The computation class argument was changed to the SimpleBoggleComputation
- The setVertexInputFormatClass was changed to the BoggleInputFormat

,

SimpleBoggleComputation

Here is the logic of the additions in this file. Note that the code is commented enough for a read-through it to make sense.

- Some imports that were needed have been made, like the Scanner (used to read the dictionary from a file) and the Iterator.
- In order to read a dictionary from a file, I created the *getDictionaryFromFile* function which gets the filename as input and it returns it in the form of a TreeSet of Strings. This function just reads a word from every line in the file and it converts it to Upper Case.
- If we are in the Superstep **0** then we also have to initialize some things.
    - A new TextArrayListWritable object is created.
    - We extract the letter of the vertex by getting its id converting it to String and keeping only the first character.
    - Then we append the TextArrayListWritable object with the actual vertex id as well.
    - Lastly we send the message that we constructed to all the edges. Since it is the first step, every vertex has to send the message to every neighbor as no further restrictions apply yet.

- If we are not in the first superstep then it is time for the more interesting computational actions to take place.
- We get the letter of the vertex by getting the id and extracting its first character.
- Then we iterate through all the received messages and for each one of them:
    - We create the new word by appending the letter of the current vertex to the first item of the message that is the current constructed word.
    - Since the current vertex has been just visited, we don't want it to be re-visited for the chain of the current word. For this reason we get the id and append it to the received message.
    - If the currently constructed word corresponds to any word in the dictionary (the full word and not just a prefix) then we add this word to the value(s) of the current vertex. If it already exists, then it is ok to re-add it as it means that the same word can be formed by following different paths that end to this node. This is a detail that seems more like a matter of implementation preference.
    - ***We don't want to send the message now through all the edges but only through those that are meaningful***. We achieve this by checking against 2 things:
        - Make sure that the target vertex has not already been visited
        - Make sure that there exist words with the one that was created now as a prefix. Otherwise it would be pointless. Note that this kind of optimization saves us potentially from a ton of meaningless message transmissions. Note that we don't want this to hold true because of the current word itself. The word itself has already been checked. So we check for words with at least one more character.
    - If any such words exist then, before we send the message to the appropriate edges, we modify the current message by replacing the old word with the new one that was created by the character append.
    - Then, we send the message to the vertices that are eligible to receive it. Remember, no pointless communication takes place!
    - In order to achieve the previous we didn't – of course – use the *sendMessageToAllEdges* function that was used at step 0. Instead, we used the ***SendMessageToMultipleEdges*** function passing to it the appropriate list and the message.

And that's about it! I am pretty sure that it would perform good which quite bigger boggle board sizes as well – haven't tried it though!