Programming Languages
Programming Assignment

Graph Statistics

You are to implement a concurrent algorithm for computing various statistics on potentially large graphs with colored nodes.

Graphs are a relational data-structure where we define a set of nodes and edges, representing some objects and a relationship between them. These graphs can be used to model any number of problems, from the power grid to social networks.

Computing statistics on large graphs becomes more computationally expensive as the number of nodes and edges increase. For applications that run on graphs with the scale of the Internet, it becomes intractable to process the entire graph on a single machine. One solution to this problem of increasing graph sizes is to distribute the computation across multiple processing nodes, dividing the massive graph problem into manageable chunks and combining results from local computations.

For this assignment, you will be given an undirected graph G that contains varying amounts of colored nodes. This graph will be provided to you already split into P partitions, each of which will be handled by its own actor A_p. Each partition consists of a set of unique nodes from G and all edges connected to those nodes. With these partitions the assignment is broken into two parts:

a) You will count the total number of nodes of each color in G, as well as get the sum of the degree of all nodes of each color.

b) You will find, for each partition, the most influential node (the node with the highest degree) that is either in the partition, or directly connected to the partition.

An example input graph is shown in Figure 1. This graph of 12 nodes is divided equally between 3 actors A_0, A_1, A_2. Each actor has its own local partition containing four internal nodes as well as all the edges connected to those internal nodes. Each actor may have edges that contain nodes that are not within their internal node list. These external nodes, which are grey in Figure 1, are necessary to compute the degree in part a) and the most influential node in part b), but the actor is not directly responsible for computing the color statistics of these external nodes.

A. Color Count and Degree

The first part of the assignment will have you compute two statistics on the input graph G. You will first compute the total number of nodes there are of each color in the graph. Then you will find the degree of each color in the graph, where the degree of a color is equal to the sum of the degrees of all nodes of that color. The degree of a node is defined as the total number of edges it belongs to.

For example, you will be provided some graph G that has been split into P partitions $\{g_0, g_1, ..., g_P\}$. Each of these partitions will be handled by an actor, A_p, which will be responsible for computing the count and degree of each color in its local partition g_p. This actor A_p will also be responsible for returning these local count / degree statistics to a requesting actor. The results from all actors can be aggregated into the full results for graph G.

To summarize the tasks of part a), each actor is responsible for three tasks in this program:

- Count the number of nodes of each color in the local partition g_p.
- For each color c in the local partition, sum the degree of the nodes of that color (where the degree of a node is equal to the number of edges it belongs to).
- Return the node count and degree of each color in the local partition.

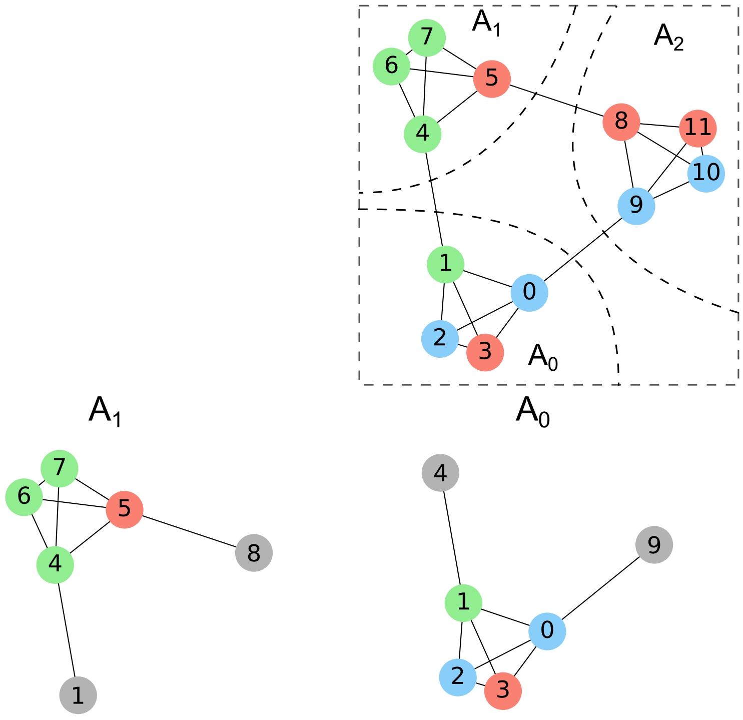

Figure 1: **Graph divided amongst actors.**

```
        blue, 4, 14
        green, 4, 14
        red, 4, 14
```

Figure 2: **Part A output file**. The color output file lists the count and total degree of the nodes for each color in the graph. For each color we will output one line, comma separated, that shows the color, the number of nodes of the color and the degree of the color. The expected output for Figure 1's graph is shown here.

The expected output of part a) will be one file, the name defined as an input parameter, containing the combined results of each actor. These results will show the color counts and color degrees for each color in the graph. The format of this file is shown in Figure 2. These results can appear in any order.

B. Most Influential Node

The second part of your assignment is to find the most influential nodes connected to each partition g_p. The most influential node for an actor A_p is the node with the highest degree that is either in partition g_p or directly connected to partition g_p.

Each actor A_p will be responsible for doing the following:

- Get the degree of each internal node of A_p.
- Find the list of external nodes connected to A_p. (Any node in the edge list that is not in the node list is an external node of A_p)
- For each external node n, message all other actors requesting the degree of node n. (An actor should return 0 if node n does not exist in their local partition).
- Find the largest degree node(s) from among the internal and external nodes of actor A_p. If there are multiple nodes of the same max degree, return all such nodes.

The output for part b) should be a text file containing one line per actor, plus one for the global graph G. Each line will print the partition number and the list of most influential nodes in or connected to the partition. The final line will print *G* and then the list of most influential nodes in the entire combined graph. A sample of this output can be seen in Figure 3.

```
        partition 0: 0,1,4,9
        partition 1: 1,4,5,8
        partition 2: 0,5,8,9
        G: 0,1,4,5,8,9
```

Figure 3: **Part B Sample Output**. The output of part B is a list of nodes per partition and one for the entire graph *G*. This list contains the most influential nodes in each partition of the graph, as well as the most influential nodes of the graph as a whole. The expected output for Figure 1's graph is shown here.

Sample Input

The input to your program will be a text file defining the edges belonging to each partition as well as a list of colors per node. Figure 4 shows what the input file for the graph defined in Figure 1 looks like.

```
    partition 0
    0,1,2,3
    blue,green,blue,red
    0,1 0,2 0,3 1,2 1,3 2,3 0,9 1,4
    partition 1
    4,5,6,7
    green,red,green,green
    4,5 4,6 4,7 5,6 5,7 6,7 4,1 5,8
    partition 2
    8,9,10,11
    red,blue,blue,red
    8,9 8,10, 8,11 9,10 9,11 10,11 8,5 9,0
```

Figure 4: **Input file.** The input file is broken into groups of nodes and edges defining each partition. The file is grouped into sets of 4 lines, each defining all the information needed by a partition actor. Line 1 the partition / actor number. Line 2 shows all the nodes in the actor's local graph partition. Line 3 shows the color assigned to each of those nodes. And line 4 gives a space separated list of edges for the actor's local partition. This pattern repeats for all graph partitions P.

The graph provided to you will be an undirected graph with no self edges. You can expect all partition and node names to be integers and that the input files will contain no errors (like duplicate nodes/edges, undirected/self edges, or mismatching node list / color list size).

Requirements

Your submission will consist of two modules, one for a local concurrent implementation of the solution, and the other for a distributed solution. We recommend you implement the concurrent solution first and then modify that code into the distributed solution.

All output should be written into the same directory as your project.

Your program should be able to work for any number of actors and various numbers of actors will be tested.

Notes for Erlang Programmers

To start your Erlang implementation, first create new directories and files in the following structure. Feel free to create additional Erlang files to help with your implementation.

```
.
|-- concurrent
|   |-- graph_stats.erl
|-- distributed
|   |-- graph_stats.erl
|-- input.txt
|-- README.md
```

Begin by implementing your concurrent program in `concurrent/graph_stats.erl`. Once you have finished the concurrent version, copy your code and paste it into the file located at `distributed/graph_stats.erl`. Proceed to modify your code into the distributed solution.

In both the concurrent and distributed `graph_stats.erl` file, you must include a function called `start` that takes three parameters, the input file name, the Part a) output file name, and the Part b) output file name. Running this function will write the results of your computation to the two output files. Please also export the `start` function so it can be run from another Erlang program. Here is an example of the `start` function declaration.

```
-export([start/3]).

start(Input_file_path, Part_a_output_file_path, Part_b_output_file_path) ->
    % Your code starts here
```

Runing the concurrent implementation

To run your Erlang concurrent program, first go to the concurrent directory, the execute the following:

```
/** Launch the Erlang shell */
erl

/** In the erlang shell, compile all files, then run the start function in graph_stats.erl */
c("graph_stats"), graph_stats:start("../input.txt", "a_output.txt", "b_output.txt").
```

Runing the distributed implementation

To run the distributed solution, multiple nodes need to be created. Each graph partition requires a new node. Additionally, a node is required for the parent. You should name your nodes using the format `child<node_num>@127.0.0.1`. For example, if there are three partitions in the input file, the node names would be `child0@127.0.0.1`, `child1@127.0.0.1`, and `child2@127.0.0.1`.

To run your Erlang distributed program, first go to the distributed directory, then given the number of nodes needed as n, open n new terminal sessions. For each session, execute

```
/** Start a node, replace <node_num> with the a number ranging from 0 to n - 1 */
erl -name child<node_num>@127.0.0.1
```

Then open a new terminal window for the parent node, execute

```
/** Start a parent node, erlang shell should open */
erl -name parent@127.0.0.1

/** In the erlang shell, compile and execute */
c("graph_stats"), graph_stats:start("../input.txt", "a_output.txt", "b_output.txt").
```

Erlang Tips

- For reference, please see the Erlang documentation on concurrency.

- Using a dictionary might be helpful for storing your computed statistics.
- In Erlang, the node name's type is an atom. To convert a string to an atom, use `list_to_atom` function.

Submission guidelines

Grading: This assignment will be graded mostly on the correctness of the output for each given graph. As always, code clarity / readability is important, and will be a factor in your final grade.

Submission Requirements: Please submit a zip file with your solutions in two separate directories, **concurrent** and **distributed**, plus a README file at the top-level directory. README files must be in plain text; markdown is acceptable. Your zip file should be named with your LMS user name(s) and chosen language as the filename, eg, userid1 userid2 erlang.zip. In the README file, place the names of each group member (up to two). Your README file should also have a list of specific features / bugs in your solution.

Do not include unnecessary files. Test your archive after uploading it. Name your source files appropriately: `graph_stats.erl` .