Project 2: Graph Components in Hadoop

1. Introduction

This is the final group programming project in CS5300. It will use AWS Elastic MapReduce to perform a much larger-scale distributed computation than anything we have done up to now.

For this project you will compute some statistics on the connected components of a relatively large undirected graph. The "story" is that you are given a file containing a satellite image of a forest, and you are asked to compute the expected number of trees that will burn if a match is dropped in the forest at a location chosen uniformly at random.

You compute this by constructing an undirected graph

$$G = (V, E),$$

in which each vertex v in V is the location of a tree in the satellite image, and there is an undirected edge $\{u, v\}$ in E if vertices u and v are sufficiently close to one another that flames can jump between them. Clearly if a tree (vertex) u catches fire, and another tree (vertex) v can be reached from u along a *path* of edges in G, then eventually v will catch fire too. The set of all vertices reachable by following paths from vertex u in an undirected graph G is called the *connected component* of G containing u. Thus, the answer to the question "How many trees can be expected to burn?" amounts to computing statistics on the sizes of the connected components of G. This is discussed in greater detail below.

The graph will be presented in a slightly counterintuitive file format, which facilitates customizing the input by netid, so each group will compute the answer on a slightly different graph (only we know the right answers).

The input format also enables processing an image that is smaller than the entire file, by reading a prefix of the file, rather than reading the entire file and filtering out locations you're not interested in. This will be useful for debugging, and possibly for partial credit if you can't get your solution to run fast enough to run the entire file at reasonable cost.

To compute the connected components of the graph, you will use an algorithm like the one discussed in Lecture, and described in our Connected Component notes.

AWS Elastic MapReduce will be used to manage the several MapReduce passes needed to identify the graph edges and compute the connected components.

For this Project you should form **groups of 3 persons**. The default will be to use the same groups as for the previous Project, but you are free to change groups if you wish. If you prefer a 4 person group that is okay, but like Project 1b, the optional part (Section 6) becomes **mandatory** for 4-person groups.

2. Preliminaries

This project should be done using AWS Elastic MapReduce. Amazon's "getting started guide" for Elastic MapReduce is here. This GSG is based on streaming job flows and/or Hive. Given that we will be using custom

Jar files, that's unfortunate. However, you should at least read it carefully. Then the section Where do I Go from Here? is full of useful stuff, including a tutorial on creating a job flow using a custom JAR and the generic Learn More About Hadoop link. And, since it's slightly non-obvious, the tarball for the (extensive) "Cloudburst" example is available for download here.

In addition to the above, tutorials for Hadoop MapReduce are available here (Yahoo!) and here (Apache).

Your solution will consist of four (or possibly more) Elastic MapReduce "Job Flow Steps," controlled using the CLI (Command Line Interface). All input and output will use AWS S3 files. Each MapReduce step will use a different "custom Jar file" with Java code.

After you get your first Job Flow Step to run, adding the remaining steps should not be too difficult. But there is a fairly high entry barrier to this project, so you should get started soon.

Note: By default, AWS Elastic MapReduce uses EC2 Small Instances, which are 1.7 GB 32-bit machines. **You should not change this default!** The point is to solve the problem using MapReduce and horizontal scaling. Computing the answer by vertical scaling -- e.g. using a single EC2 High-Memory Quadruple Extra-Large Instance with 68GB of memory -- might work, but that would be cheating.

3. Input Data

The input data format for this project is a text file, consisting of a sequence of lines. Each line is exactly 12 bytes long (including the terminating newline) and contains a single floating point number between 0.0 and 1.0. The file starts out

> 0.511715114
> 0.187992254
> 0.009772277
> 0.857811962
> ...

The numbers in the file have been chosen uniformly at random. So you can read the file and select a random subset of the points using a function like the following:

```
float wMin = ...
float wLimit = ...
// assume 0.0 <= wMin <= wLimit <= 1.0
boolean getNextFilteredInput() {
        float w = ... (read and parse next input line) ...
        return ( ((w >= wMin) && (w < wLimit)) ? true : false );
}
```

This will return `true` if and only if the next input value falls between `wMin` and `wLimit`, which should happen with probability `(wMin-wLimit)`.

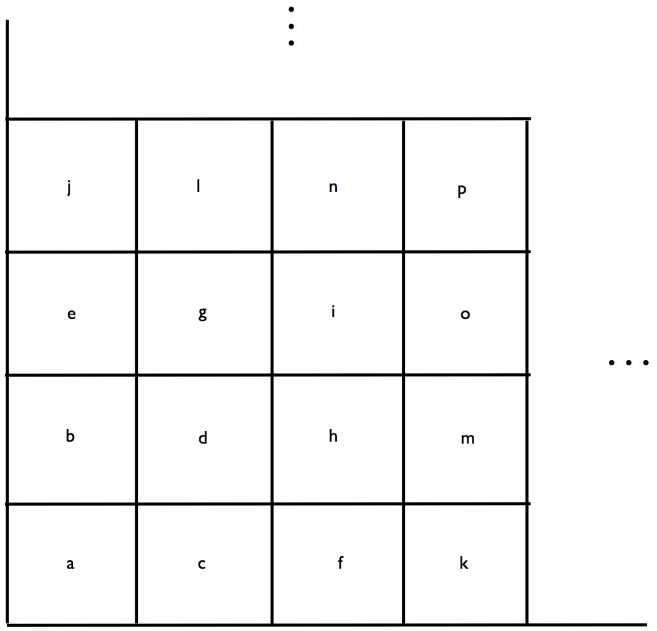
> **We want every group to compute the answers on a different random set of points!**

So you are required to construct `wMin` and `wLimit` values using the netid of one of the members of your group. Specifically, take the digits of the netid and *write them in reverse* preceded by a decimal point. This gives you a number that is more-or-less uniformly distributed between 0.0 and 1.0. (If you don't reverse the digits the distribution is pretty badly skewed towards smaller numbers.) Use your netid in place of mine in the following computation:

```
// compute filter parameters for netid ajd28
float fromNetID = 0.82;
float desiredDensity = 0.59;
float wMin = 0.4 * fromNetID;
float wLimit = wMin + desiredDensity;
```

The `desiredDensity` constant is an experimentally-determined magic number that you should not change, except for the extra credit part described in Section 6.

Another subtle aspect of the input data file is the order in which the points appear. We wanted to make it possible to read and process a subset of the image file (e.g., for debugging) by reading a prefix of the file, rather than having to scan the entire file and eliminate points whose positions are out of range. To make this possible, the points are ordered so that for any N, the N^2 points nearest the "southwest" corner appear first in the file, as shown in the following figure:



This is a bit hard to describe intuitively; but the following code snippet will properly load an N by N boolean array a[*,*] from the initial N^2 elements of the file:

```
for( int j = 0; j < N; j++ ) {
    for( int i = 0; i < j; i++ ) {
        a[i, j] = getNextFilteredInput();
        a[j, i] = getNextFilteredInput();
    }
    a[j, j] = getNextFilteredInput();
}
```

The S3 bucket

```
edu-cornell-cs-cs5300s12-assign5-data
```

contains a few small test data files

```
test1.txt, test2.txt, ...
```

These can be accessed at the usual URLs for S3 files:

```
http://edu-cornell-cs-cs5300s12-assign5-data.s3.amazonaws.com/test1.txt
```

and so forth. You can open these files in your Web browser to look at them.

In the same bucket there is also the fairly large file

```
production.txt
```

This file contains millions of lines, so don't try to open it in in your browser. You should run your final solution on *production.txt*, using the filter derived from your netids by the procedure described above. (If your program can't handle the entire file in a reasonable time, then run it on the longest prefix of *production.txt* that you can manage).

4. Computing the Connected Components

The basic algorithm you should use for computing connected components was discussed in Lecture, and is described in our Connected Component notes.

5. Output

The output we want from your program is a few statistics about the input graph:

1. the number of vertices;
2. the number of edges;
3. the number of distinct connected components;
4. the (weighted) average size of the connected components -- that is, if you choose a vertex uniformly at random from the graph, what is the expected size of its connected component?
5. the average *burn count* -- that is, if you choose a position uniformly at random and drop a match there, how many trees will burn? (Note that some positions do not contain trees, so dropping a match there causes 0 trees to burn).

There is a "right" value for each of these statistics, and it is a function of the netid used to define your filter values. We can compute the right answers in real time to check your results. Getting the wrong answer will not affect your grade too much; but getting the right answer is *prima facie* evidence that your program is correct.

Computing the required statistics will probably require an additional MapReduce pass after you have computed the connected components. The final pass can be very similar to the "word count" example discussed in the original MapReduce paper.

6. Extra Credit (mandatory for 4-person groups)

You may do the extra computation described in this section for up to 25% extra credit. For 4-person groups, you **must** do it.

First, modify your program so edges can run diagonally as well as horizontally and vertically; in this case the maximum number of edges a vertex can have grows from 4 to 8. This should be a fairly straightforward change to your program. The number of edges inferred from the test data will increase significantly, so your program will run

somewhat more slowly and the results will change. In particular, the connected components will get dramatically larger.

A result from percolation theory (no, we don't expect you to learn any percolation theory for this project) implies there is a *critical density* for this problem. If the density of trees in the satellite image is below the critical density, the connected components are "small" with high probability, even in the limit of an infinite graph. Above the critical density, the connected components are "large", and in the limit of an infinite graph there are infinite components with high probability.

The magic constant `desiredDensity = 0.59` from Section 4 was chosen to be near the critical density for this system *when edges are only horizontal and vertical*. Of course, when diagonal edges are added, there are more possible ways for paths to form, so you should expect the critical density to be smaller.

For full extra credit you should estimate the critical density for the system in which diagonal edges are allowed. Choose a modest file size -- say 1000 by 1000 -- and make a plot of average component size or burn count against density. The critical density should be pretty obvious from such a plot.

7. Submit Your Work

CMS: Create a file, in `zip` archive format, named `solution.zip` This archive should contain

1. a `README` file.
   This may be in `.pdf` or `.txt` or `.doc` format. This file should include anything we need to know to grade your assignment. In particular, it should briefly describe the overall structure of your solution, and specify what functionality is implemented in each MapReduce pass.

   Of course it should contain the output -- the statistics discussed in Section 5 -- resulting from running your program on the `production.txt` file. It should also contain the parameters you used for your run:

   - the netid and computed filter parameters (`wMin` and `wLimit`) used for your filter definition to select vertices;
   - if you found it necessary to run on a prefix of `production.txt`, the number of grid points you processed;
     if you did the extra credit part, the parameters used for that part and your best estimate of the critical density.
2. the Java source code for each MapReduce pass.
   Include understandable high-level comments.

As always, you may include additional files in the archive if you wish. Submit your `solution.zip` file using CMS by the specified deadline.

Presentation: For this Project, we will ask you to sign up for a brief (15 minute) presentation of your solution. This is not a demo -- we won't ask you to run anything at Amazon in real time -- it's more like a short oral exam, in lieu of a Final, where we get to ask you questions about your solution. You should prepare a few PowerPoint (or .pdf or ...) slides for this presentation -- a projector will be available. Script your presentation so everybody in the group gets to say something.

Note: the deadline for submitting your project to CMS is the same for everybody; it is *not* the the time your presentation is scheduled.