Twitter

In this project, you are asked to implement a simple graph algorithm that needs two Map-Reduce jobs. You will use real data from Twitter from 2010. The dataset represents the *follower graph* that contains links between tweeting users in the form: user_id, follower_id (where user_id is the id of a user and follower_id is the id of the follower). For example:

```
12,13
12,14
12,15
16,17
```

Here, users 13, 14 and 15 are followers of user 12, while user 17 is a follower of user 16. The complete dataset is available on Expanse (file `/expanse/lustre/projects/uot187/fegaras/large-twitter.csv`) and contains 736,930 users and 36,743,448 links. A subset of this file (which contains the last 10,000 lines of the complete dataset) is available in `small-twitter.csv` inside `project1`.

First, for each twitter user, you count the number of users she follows. Then, you group the users by their number of the users they follow and for each group you count how many users belong to this group. That is, the result will have lines such as:

```
10 30
```

which says that there are 30 users who follow 10 users. To help you, I am giving you the pseudo code. The first Map-Reduce is:

```
map ( key, line ):
  read 2 integers from the line into the variables id and follower_id (delimiter is comma ",")
  emit( follower_id, id )


reduce ( follower_id, ids ):
  count = 0
  for n in ids
     count++
```

```
  emit( follower_id, count )
```

That is, the ids in the reducer is the sequence of all users followed by the user with follower_id.
The second Map-Reduce is:

```
map ( key, line ):
  read 2 integers from the line into the variables follower_id and count (delimiter is tab "\t")
  emit( count, 1 )


reduce ( count, values ):
  sum = 0
  for v in values
     sum += v
  emit( count, sum )
```

Matrix Multiplication

For this project, you are asked to implement matrix multiplication in Map-Reduce. Here is the pseudo code to multiply two matrices M and N. The first Map-Reduce job finds all possible products $M_{ik}*N_{kj}$, for all k, by joining M and N:

```
// mapper for matrix M
map(key,line) =
  split line into 3 values: i,k,m
  emit(k,new Triple(0,i,m))


// mapper for matrix N
map(key,line) =
  split line into 3 values: k,j,n
  emit(k,new Triple(1,j,n))
```

```
reduce(k,values) =
  M_values = all triples (0,i,m) from values
  N_values = all triples (1,j,n) from values
  for (0,i,m) in M_values
    for (1,j,n) in N_values
      emit(new Pair(i,j),m*n)
```

The second Map-Reduce adds all products for each pair (i,j):

```
map(pair,value) =
  emit(pair,value)


reduce(pair,values) =
  sum = 0.0
  for v in values
    sum += v
  emit(pair,sum)
```

Graph Processing

A directed graph is represented in the input text file using one line per graph vertex. For example, the line

```
1,2,3,4,5,6,7
```

represents the vertex with ID 1, which is linked to the vertices with IDs 2, 3, 4, 5, 6, and 7. Your task is to write a Map-Reduce program that partitions a graph into K clusters using multi-source BFS (breadth-first search). It selects K random graph vertices, called centroids, and then, at the first iteration, for each centroid, it assigns the centroid id to its unassigned neighbors. Then, at the second iteration. it assigns the centroid id to the unassigned neighbors of the neighbors, etc, in a breadth-first search fashion. After few repetitions, each vertex will be assigned to the centroid that needs the smallest number of hops to reach the vertex (the closest centroid). First you need a class to represent a vertex:

```
class Vertex {
  long id;         // the vertex ID
  Vector adjacent;  // the vertex neighbors
  long centroid;    // the id of the centroid in which this vertex belongs to
  short depth;      // the BFS depth
  ...
}
```

Vertex has a constructor Vertex(id, adjacent, centroid, depth).

You need to write 3 Map-Reduce tasks. The first Map-Reduce job is to read the graph:

```
map ( key, line ) =
  parse the line to get the vertex id and the adjacent vector
  // take the first 10 vertices of each split to be the centroids
  for the first 10 vertices, centroid = id; for all the others, centroid = -1
  emit( id, new Vertex(id,adjacent,centroid,0) )
```

The second Map-Reduce job is to do BFS:

```
map ( key, vertex ) =
  emit( vertex.id, vertex )   // pass the graph topology
  if (vertex.centroid > 0)
     for n in vertex.adjacent:     // send the centroid to the adjacent vertices
        emit( n, new Vertex(n,[],vertex.centroid,BFS_depth) )


reduce ( id, values ) =
  min_depth = 1000
  m = new Vertex(id,[],-1,0)
  for v in values:
     if (v.adjacent is not empty)
        m.adjacent = v.adjacent
     if (v.centroid > 0 && v.depth < min_depth)
```

```
        min_depth = v.depth
        m.centroid = v.centroid
    m.depth = min_depth
    emit( id, m )
```

The final Map-Reduce job is to calculate the cluster sizes:

```
map ( id, value ) =
  emit(value.centroid,1)


reduce ( centroid, values ) =
  m = 0
  for v in values:
     m = m+v
  emit(centroid,m)
```

Kmeans Clustering

You are asked to implement one step of the Lloyd's algorithm for K-Means clustering using Spark and Scala. The goal is to partition a set of points into k clusters of neighboring points. It starts with an initial set of k centroids. Then, it repeatedly partitions the input according to which of these centroids is closest and then finds a new centroid for each partition. That is, if you have a set of points P and a set of k centroids C, the algorithm repeatedly applies the following steps:

1. Assignment step: partition the set P into k clusters of points P_i, one for each centroid C_i, such that a point p belongs to P_i if it is closest to the centroid $C_i$among all centroids.
2. Update step: Calculate the new centroid C_i from the cluster P_i so that the x,y coordinates of C_i is the mean x,y of all points in P_i.

The datasets used are random points on a plane in the squares (i*2+1,j*2+1)-(i*2+2,j*2+2), with 0≤i≤9 and 0≤j≤9 (so k=100 in k-means). The initial centroids in `centroid.txt` are the points (i*2+1.2,j*2+1.2). So the new centroids should be in the middle of the squares at (i*2+1.5,j*2+1.5).

In this project, you are asked to implement one step of the K-means clustering algorithm using Spark and Scala. A skeleton file `project4/src/main/scala/KMeans.scala` is

provided, as well as scripts to build and run this code on Expanse. **You should modify KMeans.scala only**. Your main program should take two arguments: the text file that contains the points (points-small.txt or points-large.txt) and the centroids.txt file. The resulting centroids will be written to the output. This time, the process of finding new centroids from previous centroids using KMeans must be repeated 5 times.

```
for ( i <- 1 to 5 ) {
  /* broadcast centroids to all workers */

  /* find new centroids using KMeans */
  centroids = points.map { p => val cs = /* the broadcast centroids */
                    ( closest_centroid(p,cs), p )
                }
            .groupByKey()
            .map { /* calculate a new centroid */ }
            .collect()
}
```

where `closest_centroid(p,cs)` returns a point `c` from `cs` that has the minimum `distance(p,c)`, which is the Euclidean distance between `p` and `c`. The centroids must be broadcast to all workers so that the first map can retrieve the centroids as an array `cs`.

Graph.pig / Graph.scala

In this project, you are asked to implement a simple graph algorithm on Apache Pig. A directed graph is represented as a text file where each line represents a graph edge. For example,

```
20,40
```

represents the directed edge from node 20 to node 40. First, for each graph node, you compute the number of node neighbors. Then, you group the nodes by their number of neighbors and for each group you count how many nodes belong to this group. That is, the result will have lines such as:

```
10    30
```

which says that there are 30 nodes that have 10 neighbors. In your Pig script, you can access the path of the input graph as '$G' and the output path as '$O'. That is, you can use LOAD '$G' USING ..., to load the graph and STORE X INTO '$O' ..., to write the relation X to the output directory.