Project Report

Team Members:

Kousthubh Belur Sheshashyee

Rachna Reddy Melacheruvu

GitHub Link to the project:

https://github.ccs.neu.edu/kousthub93/cs6240_project_RK

Project Overview:

This project involves the below two tasks:

1. To explore the degree of separation between two people in the twitter dataset.
2. To explore cycles of different lengths in the dataset.

Given a person in the twitter dataset, the result displays the different people involved in their 1st connection, 2nd connection, 3rd connection and so on based on the input provided. This also provides a way of understanding if a person can be reached through one of his/her connections on the twitter.

Input Data:

Task 1 - Twitter Dataset:

The dataset is the friendship/followership network among the users of the twitter social website. The friends/followers are represented using edges and edges are directed. For example - 1,2. This means, user id "1" is following user with id "2".

Task 2 - Delaunay Triangulation Dataset:

Delaunay triangulation - for a given set P of discrete points in a plane is a triangulation DT(P) such that no point in P is inside the circumcircle of any triangle in DT(P).

Task1: Single source shortest path (MapReduce):

Overview:

The main aim of this task is to explore the degree of separation between two people using the twitter dataset. In the above section, we have already noticed that, each row in the twitter dataset indicates an edge between two nodes. So, given a source node, the program gives the nodes that are at 1-hop from source node, 2-hop from source node and so on. There might be a node that is from both 1-hop and 2-hop from source node, in that case, single source shortest path comes into picture. So, in this way, shortest distance to all the nodes from the given source node is found.

Pseudo-Code:

To generate Adjacency List:

// input for this map is the twitter dataset

```
Mapper {
 map(key, value[follower,followee]){ // map just emits the two nodes in each line of the input file
    String s = covert value to string
    String[] followers = s.split(“,”)                          // split string on delimited as “,”
    write a key-value pair as (followers[0],followers[1]) to context
  } }

Reducer {
 reduce(follower, Iterable<Text> followees){
    String adj = ""
    HashSet<String> nodes = new HashSet<>();
    for (node : values) {                                       // remove duplicate nodes
       nodes.add(node)
    }
    for (n : nodes) {
       adj += n + ",";
    }
    adjacencyList = adj.substring(0,adj.length()-1);           // remove the trailing comma
    emit(vertex, adjacencyList)                               //‘vertex neighbour1,neighbour2….’
  } }
```

Print connections using single source shortest path:

Input to the below mapper is the output from the above step, that is adjacency list generated above.

```
Mapper {
  map(key, value[follower, adjList]) {
     String[] s = covert value to string and split by “,”
    valueKey = s[0]                                          // first element is the vertex
    valueVal = s[1]                                          // second element is its adjacency list
    String sourceV = confs.get("source");                    // get the source from the context passed to the mapper
    String[] valuesIn;
    distance = Long.MAX_VALUE;                               // initialize distance to infinity
    String path=sourceV;                                     // path contains, the path from the source to the current vertex
    String adjacency="";
    outKey.set(valueKey);
    if(!valueVal.contains("#")) {                            // this condition indicates first iteration, from the subsequent
                                                             // iterations, the adjacency list will be in a different form
      if(sourceV.equals(valueKey)) {                         // if the vertex is the source vertex
        distance=0;                                          // initialize the distance as 0
        outValue.set(valueVal+"#"+sourceV+"#"+"0");
      }
      else {                                                 // if not source, initialize the distance= inf and path = null
        outValue.set(valueVal+"#"+"null"+"#"+String.valueOf(Long.MAX_VALUE));
      }
      adjacency = valueVal;
    }
    else {                                                   // this indicates, second iteration. Here the value is of the form,
                                                         // adjacenyList#path_to_the_source#minimum_distance_from_source
      valuesIn = valueVal.split("#");
      adjacency = valuesIn[0];
      distance = Long.parseLong(valuesIn[2]);
      path = valuesIn[1];
      outValue.set(valueVal);
    }
    emit(outKey,outValue);                               // emit the node and its adjacency list along with the path and distance

    for(vertices in adjacency_list) {                        // iterate through each of the nodes in the adjacency list
      String newPath = sourceV;                              // path is initialized to contain the source vertex
      long dist = distance;

        if(dist!=Long.MAX_VALUE) {                           // this is to check to see if it’s an active node or not

            outKey1.set(vertices);
            dist+=1;                                         // increment distance by 1, since edge weight = 1
            if(!path.equals(valueKey)) {
              newPath = valueKey + "<-" + path;
            }
            outValue1.set(dist+";"+newPath);                 // emit distance;path for each of the nodes in the adj list
            emit(outKey1,outValue1);
        }     }    } }
```

```
Reducer {
  reduce(user, [d1,d2….]) {                   // input – vertex as key  adjacency list and computed distances for all m's inlinks
    dMin=Long.MAX_VALUE;                      // initialize to infinity
    M = null
    for all d in [d1,d2…] do {
      if(d.contains("#"))  {                  // the vertex object was found. Recover graph structure.
         M = d                                // recover the graph structure
      }
      else if (val.toString().contains(";")) {  // distance value for an inlink was found
        if(d<dMin) {                          // to keep track of the minimum distance from the source vertex
          dMin=d;
          newPath=pathP;                      // update the path with the shortest path found till now
        }   }     }
    if(dMin < M.distance) {                   // the distance changes from the infinity to newly updated distance
                                              // hence, that vertex becomes the active vertex in the next iteration
      emit("text", key,newPath ,outputPath);  // write to a separate file the connection in every iteration/hop
    }
    Emit(user,M)                              // emit updated adjacency list with updated path and distance
  } }
```

Link to the GitHub repository: https://github.ccs.neu.edu/kousthub93/cs6240_project_RK/tree/master/SingleSource/MR-Demo

Algorithm and Program Analysis:

To achieve our goal to display the user IDs of people involved in the 1st connection, 2nd connection and so on, single source shortest path algorithm seems to be the best fit. Single source shortest path using Dijkstra's algorithm is an elegant option, but its not a good fit for parallel execution since it involves removal of multiple nodes from the priority queues at once.

Shortest path using BFS seems to be a good fit since we can find all the vertices reachable from a source node in exactly one hop and update the distance in the first iteration and in the second iteration, we can find all the vertices reachable from source in exactly two hops and so on. This majorly relies on the property that if there exists a path of length d[u] to some vertex u, then for each vertex v in u's adjacency list there exists a path of length d[u]+weight(u,v). We are considering uniform weights, so, weight of each of the edge is taken as 1.

Experiments:

Cluster size of 11 and 6 is used to run our program on M4 large machine on AWS. As said above, we have chosen to run our program on twitter dataset which consists of 11316811 nodes and 85331846 edges trying to run 5 iterations to display all the nodes reachable within 5-hop distances. It ran successfully for the entire dataset. The logs and output files related to this are given in the link below.

Link to Log files:

https://github.ccs.neu.edu/kousthub93/cs6240_project_RK/tree/master/SingleSource/MR-Demo/logs/large%20cluster/FullInput

https://github.ccs.neu.edu/kousthub93/cs6240_project_RK/tree/master/SingleSource/MR-Demo/logs/small%20cluster/fullInput

Link to Output files:

https://s3.amazonaws.com/mapreduce-course/index.html?prefix=outputConnections/

https://s3.amazonaws.com/kous-mr/index.html?prefix=outputConnections/

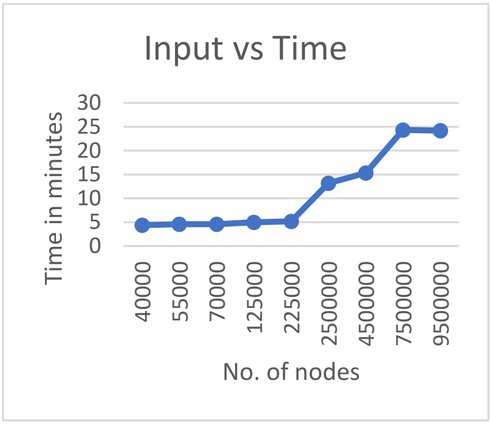
Speedup:

Cluster Size	Number of nodes	Time taken
Small (5 workers)	11316811(full input size)	32min 48sec
Large (11 workers)	11316811(full input size)	23min 52ec

Speedup = 1.37. Theoretically running time on 11 workers should be half the running time on 5 workers. But we have not obtained the perfect speed up. The reason for this can be seen when the adjacency list obtained is examined carefully. The adjacency list is not uniform which results in data skewness. For few nodes, the adjacency list is very huge. Therefore, workers which are assigned any of these nodes having very huge adjacency list tend to take longer time.

Scalability:

We tried running for different input size data to generate a graph to get an idea of how the input size affects running time.



Number of Nodes	Running time
40000	4min 36sec
55000	4min 54sec
70000	4min 55sec
125000	5min 0sec
225000	5min 16sec
1000000	7min 36sec
2500000	13min 14sec
4500000	15min 34sec
7500000	24min 34sec
9500000	24min 18sec

It can be seen from the above graph that up to 225000 nodes, increase in the running time is not very much evident. But, once the input size increases above 2500000, the increase in the running time is also very high. This is because the amount of intermediate output increases exponentially after a certain point and this results in worker machines taking long time to process the intermediate output files.

Result Sample:

D C<- B <- S. This indicates that D can be reached in two hops from source vertex S through B and C.

Task2: Interesting Structures in a graph (MapReduce):

Overview:

The main aim of this task is to find the number of cycles of input (k length) in the given dataset.

Pseudo-Code:

Input: Length of the cycles to be explored, input file path, output file path

The pseudo-code for generating the adjacency list is same as task1.

The pseudo-code for finding cycles is as below: Adjacency list is fed to the below mapper in the first iteration.

```
Mapper {
 map(key, value[user, adjList])  {
     String[] s = Convert value to string and split by delimiter "\t"
     String valueKey = s[0]
```

```
    String valueVal = s[1]
    AdjObj adj = new AdjObj(valueVal);          // AdjObj class has adjacency list, active and fromUsers path as properties
    if(iterationCount==1)
      adj.activeStatus=true;

    if(adj.activeStatus)  {
      adj.activeStatus=false;
      String[] adjNodes = adj.adjList.split(",");
      for(String s: adjNodes)   {                          // for each node in adjacency list, emit node as key and the
                                                           // node or nodes it can be reached from as value
        if(adj.fromUsers.isEmpty()) {
          outVal="path-"+valueKey;
          emit(s,outVal);
        }
        else {
          for(String node: adj.fromUsers) {
            int cycl=0;
            if(node.contains(s)) {                                   // to calculate length of cycle if it exists
              count nodes in the node string and add 2(as we need to consider the starting and
              closing edge )
           }
           if(cycl>0) {
             if (cycl is same as iteration and required cycle length){
             context.getCounter(CYCLES.K_CYCLE_COUNT).increment(1);  // increment global counter
                }  }
            else
            {
               outVal="path-"+node+"->"+valueKey;                    //add the obtained node to existing path
              emit (outKey, outVal);
            }
         }  } } }

         String activeOrNot=adj.activeStatus? "active":"not";          // mark visited nodes as active or not
         if(!adj.fromUsers.isEmpty())                                  // to create the updated adjacency list
         {
              for(String s :adj.fromUsers)
                   users+=":"+s;
               adjVal = adj.adjList+"/"+activeOrNot+users;
          }
          else
               adjVal= adj.adjList+"/"+activeOrNot;

          emit(valueKey, adjVal);   // emit input node as key and adjacency list, active or not and nodes path which can be
                                       // used to reach input node
  }
}

Reducer {
    reduce(user, values[d1,d2….]) {
    for(Text v: values) {
      if(v.toString().contains("path-")) {                   // if it's a path that can reach the key node add to a hashset
```

```
          String p= v.toString().replace("path-","");
           userNodes.add(p);
           status=true;
        }
         else
            adj = new AdjObj(v.toString());                        // else it's an AdjObj object with adjacency list, active status
                                                                   // and nodes from which key node may be reached
        }
      if(adj!=null) {
         adj.activeStatus=status;
         for (String s : userNodes) {                              // code to remove duplicate path for the same key node
            for (int i = 0; i < adj.fromUsers.size(); i++) {
               if (s.contains(adj.fromUsers.get(i)))
                  adj.fromUsers.remove(i);
            }
            adj.fromUsers.add(s);
         }

         String activeOrNot = adj.activeStatus ? "active" : "not";
         if (!adj.fromUsers.isEmpty()) {
            for (String s : adj.fromUsers) {
               users += ":" + s;
            }
            adjVal = adj.adjList + "/" + activeOrNot + users;
         } else {
            adjVal = adj.adjList + "/" + activeOrNot;
         }
         emit (user, adjVal);                                       // emit input node as key and adjacency list, active or not
                                                                    // and nodes path which can be used to reach input node
      } }}
```

Link to GitHub Repository:

https://github.ccs.neu.edu/kousthub93/cs6240_project_RK/tree/master/Cyclefinder/MR-Demo

Algorithm and Program Analysis:

The input for the above program is the value k (length of cycles to be explored), the input and output file path. The first job generates adjacency list. The second job runs for k iterations. For the first iteration, it takes the adjacency list generated in the first job as input and produces updated graph structure. This acts as input for the next iteration. So, ith iteration uses graph structure produced in (i-1)th iteration. In each iteration, the program runs BFS from all the nodes and appends all the paths that pass through this node to produce a new graph structure each time. When it is kth iteration, number of cycles are found, and the counter is updated in the mapper. Once the job ends, the driver function reads this value and outputs to the logger giving us the number of cycles of length k.

Since a new graph structure is generated in each iteration, the size of the intermediate files keeps piling up which resulted in out of memory problems initially. So, we found a solution to solve this problem. After i-th iteration, graph structure produced in (i-1)th iteration is deleted from HDFS so that the program always runs within the available memory. This also improves the running time of the program. The experiments related to this are as given in the below section.

Experiments:

Cluster sizes of 11 and 6 are used to run our program on M4 large machine on AWS. we chose to run our program on Delaunay Triangulation dataset which consists of 8.4M nodes and 25.2M edges with a maximum degree of 28. We initially tried running our program on different datasets. For this program, important factor was not just the number of edges, it was also the degree of each node. Degree is nothing but the number of edges incident on that node.

Dataset #	**Maximum degree**	**Number of edges**	**Comments about the run**
1	40	63.5M	600sec time out error
2	13	54.1M	No Cycles
3	18	42.7M	Java Heap Space Error
4 (chosen dataset)	28	25.2M	Successful run without errors

For each node, the program keeps appending all the paths through that node in each of the iteration. If the in-degree of each node is very high, the program keeps appending more paths to each of the nodes. This results in 'out of space error'. Even if the space issue is not faced, a task which is assigned a node with a huge adjacency list takes a very long time to process all the adjacent vertices and results in the time out error.

Link to log files: https://github.ccs.neu.edu/kousthub93/cs6240_project_RK/tree/master/Cyclefinder/MR-Demo/logs

Link to output files:

https://github.ccs.neu.edu/kousthub93/cs6240_project_RK/blob/master/Cyclefinder/MR-Demo/logs/LargeCluster/stdout.txt

https://github.ccs.neu.edu/kousthub93/cs6240_project_RK/blob/master/Cyclefinder/MR-Demo/logs/small%20cluster/stdout

Link to files showing cyclic Paths:

https://s3.amazonaws.com/spark-cs6240-course/index.html?prefix=output/

https://s3.amazonaws.com/kous-spark/index.html?prefix=output/

Speedup:

Cluster Size	**Number of edges**	**Time taken**
Small (5 workers)	25.2M (full input size)	59min 49sec
Large (11 workers)	25.2M (full input size)	38min 13sec

Speedup = 1.56. Theoretically running time on 11 workers should be half the running time on 5 workers. But we have not obtained the perfect speed up. The reason for this can be clearly understood from the above experiments section. Load on worker node is not distributed equally because of the data skewness in the adjacency list.

Optimization for Scalability:

Number of BFS iterations depends on the length of cycles that we are trying to explore. Initially the program successfully generated the output for cycles of length 4. But due to time out error, the program failed to explore cycles of length greater than 4. We then added a code snippet to handle duplicate paths. In each iteration, the function checks if the path is already present. If its not, then the path is appended to the adjacency list. Due to this optimization, the program ran successfully to generate cycles of length 7 within 60min.

Result Sample:

No. of cycles of length 7 = 992.