Use A Self Developed Java Graph Class and R To Identify and Study Communities in Amherst College Facebook Snapshot (Sep. 2005) Data

Overview

This is the cap stone project for "Java programming: Object-Oriented Design of Data Structures". In this cap stone project, a Java graph class is developed to study a number of graph problems. I chose to implement an algorithm to identify communities in graph data.

Two graph data sets are used for this project. One is a testing data set I created. The other is the Facebook data of Amherst College on Sep. 2005.

Data

"Facebook Data Scrape (2005)" is obtained from the site: https://github.com/caesar0301/awesome-public-datasets#social-networks.

I downloaded the data as a zip file.

The documentation of the zip file said: "This .zip file contains the Facebook networks (from a date in Sept. 2005) for 100 colleges and universities. These files only include intra-school links. (Note that these are the full sets of links inside each school, ignoring isolated nodes; we have not restricted this data to the largest connected components.)"

The Facebook data for each school is in a file of LABMAT format (.mat). There 100 .mat files for 100 schools. **I took Amherst41.mat for analysis.**

Amherst41.mat contains two elements:

- **An element named "A" is a sparse matrix representing the Facebook connection (graph) of people.**
- **An element named "local_info" has seven attributes describing each vertex (person) on the graph. The attributes are: a student/faculty status flag, gender, major, second major/monor, dorm/house, year, high school. Missing data is coded 0.**

Amherst data has 2235 vertices and 90954 edges. The graph is non-directional. It is the 4th smallest dataset out of the 100 university data.

The .mat file is first read into R (a free framework for statistical analysis), and then exported as a matrix text file so that our Java file loader can read. R code is below:

```
#####################################

# Analyze Facebook data using Java/R

#####################################

# author: Lu
```

```
# date: April 2016

# Version2: save graph connection matrix as a csv file

library(R.matlab)

# path to .mat data

path <-
c("C:\\Users\\LU\\java_workspace\\SocialNetworks\\data\\facebook100\\facebook100
")

# get list of school filenames

schools <-
read.csv("C:\\Users\\LU\\java_workspace\\SocialNetworks\\data\\facebook100\\faceb
ook100\\school_list.txt", header = TRUE, stringsAsFactors = FALSE)

# outfile captures info on each school:

# node counts

# edge counts

outfile = paste(path, "school_info.csv", sep = "\\")

cat(paste("school", "node", "edge", "\n", sep = ","),

   file = outfile,

   fill = FALSE,

   append = TRUE)

for (i in c(1:nrow(schools))) {

   # read input file, schools$SCHOOL_NAME[i] one of it has value "Amherst41.mat"

  data <- readMat(paste(path, schools$SCHOOL_NAME[i], sep = "\\"))

   # sparse vertex connection matrix

  A <- as.matrix(data$A)

   graph_input = paste(path, schools$SCHOOL_NAME[i], sep = "\\")

  graph_output = sub(".mat", ".csv", graph_input)

 # convert connection matrix A to Excel format so that it will be converted to the Java

 # loader format.  I can directly convert this to Java loader format in R.  Didn't do

 # so because the Java code was already written.

  write.csv(A, file = graph_output, row.names = FALSE, col.names = FALSE)
```

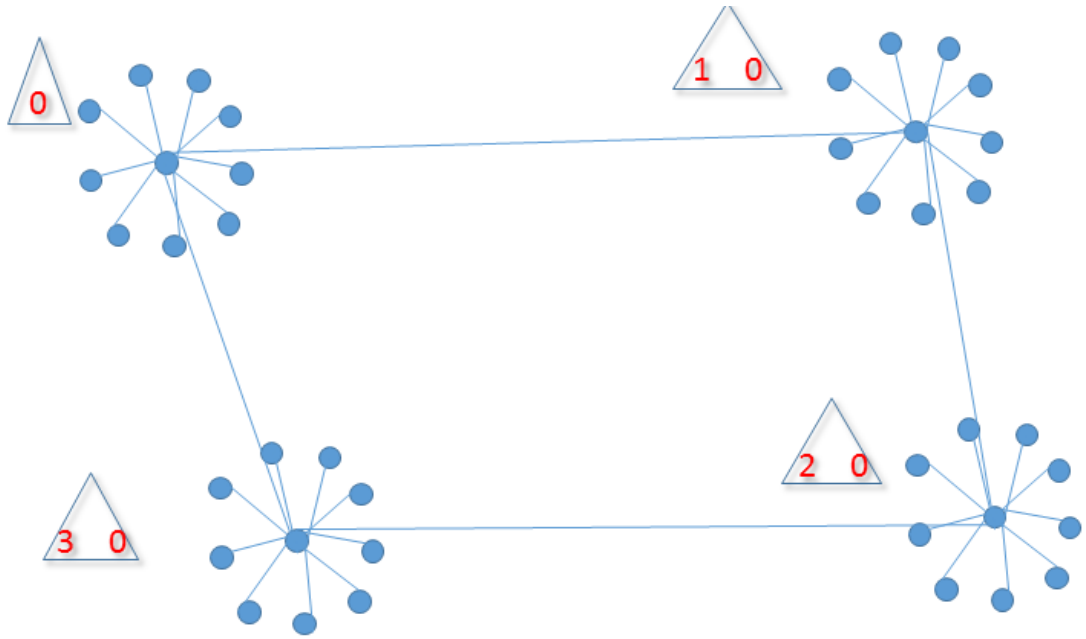
```
  edge_info <- as.data.frame(table(A))
 # matrix A should only have values 0 (no connection) and 1 (connection)
# when it has more than just 0 and 1, something is wrong with this data
 if (nrow(edge_info) > 2) {
  print(schools$SCHOOL_NAME[i])
  print("data corrupted")
  break
 }
  cat(paste(schools$SCHOOL_NAME[i], nrow(A), edge_info[2,"Freq"], "\n", sep = ","),
      file = outfile,
      fill = FALSE,
      append = TRUE)
}
```

Then, I used the CapGraph class developed in week one to compute SCC to make sure all vertices are connected. And, they are.

As a footnote, it is hard to obtain real network data. I am not a heavy Twitter or Facebook user. So, there is not much social network data on me. I do connect to a lot of people on LinkedIn. But, it is a professional network and the connections have a different meaning. I am very grateful to the people who made Facebook Data Scrape (2005) data available to public.

I did convert all the .mat files from all universities to Java readable format. This activity took about five days. One of the university data (not Amherst) seemed to be corrupted.

The testing data set is 4 clusters (snowflakes) connected only via the center vertices.



Questions – Answers – How I did it (Algorithm and Data Structure)

1. **(Easy question) What is the general nature of Amherst Facebook data?**
   These answers are obtained by using R (script on GitHub):
   a) It is a single SCC: all vertices are connected. It is also non-directional: linkage is a pair, e.g. vertex 1 to vertex 21, and vertex 21 to vertex 1.
   b) 2235 vertices and 90952 edges.
   c) Student/faculty flag has 1653 counts of 1s, 553 counts of 2s, 3 counts of 6, 4 counts of 15 and 5 counts of 8. 1 = student. 2 = (maybe) faculty. I did not find the code book, therefore not sure the code meaning. I am assuming the dominant group is the student group.
   d) Gender: 203 counts of 0 (missing data), 1015 of 1s and 1017 of 2s. I assume 1 is male, 2 is female.
   e) Major: there are 29 majors. Some majors are more popular, such as 94, 99, 100, 106, 113, 114. But, the highest group is the undeclared (0 has 582 counts).
   f) Minor: there are 30 minor categories. 99, 113 are popular minors. Overwhelming majority doesn't have a minor (0 has 1370 counts).
   g) Dorm/house: there are 34 different dorm/houses. Big majority are not in any housing: 0 has 945 counts.
   h) Students came from a large number of different high schools. No high school dominates.
   i) Year: This data was collected in 2005. The year may be graduation year. Majority of the year counts are:

2008	380
2009	377
2007	363

2006	358
2005	309
2004	147
0	214

2. **(Hard question) What are the communities among Amherst students on Facebook? And, what are the characteristics of each community, such as gender, major, high school, etc.?**

   Algorithm for identifying communities

   I am using the "outside-in" method to identify communities. The core idea of this algorithm is to think that the linkages from one community to another are limited. For example, only a few highways linking one city to another, whereas there are many small roads connecting everyone within a community.

   These steps are taken to identify the "highways", i.e. most frequently used edges:

   a. Identify the shortest path(s) from a vertex to another by using Breath First Search, and do this for all the vertices;

      BFS algorithm and data structures:
      Given a start vertex (integer) and an end vertex (integer), find the shortest path(s). It is possible that multiple paths are all of the shortest length, and all these paths must be identified.

      Each shortest path is represented as a list of integers.
      The data structure for multiple shortest paths is Set<ArrayList<Integer>>.
      Processing queue (first in first out) is to capture what vertex BFS is to process next.
      ArrayDeque<Integer> is used to represent this queue.
      Visited set captures all the vertices that are already processed and no need to process again.
      In order to facilitate quick look up, hash is used. Data structure: HashSet<Integer>.
      As the algorithm searches on, we need to remember which vertex is the parent (right in front) of which child. This is a parent centric view: if a vertex (v1) has parent A and parent B, it is represented as A -> v1; B -> v1. The parental data structure is HashMap<Integer, HashSet<Integer>>.

      Java CapGraph (cap stone graph class) has a method BFS:

```
public Set<ArrayList<Integer>> BFS (Integer start, Integer end) {

        // reject error conditions: start is equal to end; start or end not a valid vertex

        // init data structures listed above

        Put start vertex on processing queue
        While (processing queue is not empty) {
                Retrieve from front of processing queue a vertex as current vertex
                If (current vertex == end vertex) {
```

```
                        Get shortest paths from parental structure
                        Return shortest paths
                }
                // need to process current vertex
                Else if current vertex is not already visited {
                        Put current vertex on the visited list
                        For each vertex in the neighbor list of the current vertex {
                                If this vertex is not visited before {
                                        Add it to the processing queue
                                        Add current vertex as the parent of this vertex
                                }
                        }
                }
        }
}
```

Algorithm for getting the shortest paths from parental structure:

```
private Set<ArrayList<Integer>> processPaths(Integer s, Integer g,
HashMap<Integer, HashSet<Integer>> p)
```

It turns out to reconstruct all the shortest paths from the parental structure is more involved than I originally thought. I am constructing the paths from end (goal) vertex to start vertex. I first put the end vertex on the shortest path. Then, I recursively all a build path routine given it the start vertex, the current vertex, and the parental structure.

Build path routine:

```
private Set<ArrayList<Integer>> buildPaths(Integer s, HashMap<Integer,
HashSet<Integer>> parent_structure, Set<ArrayList<Integer>>
unfinished_paths)
for each unfinished paths {
        get all the parental vertices from the last vertex of the path
        if parental vertices exist {
                if start vertex is among the parental vertices {
                        since set has no duplicate,
                        we can add this to the path, and
                        put the path on next level set, and
                        set the DONE flag
                }
                Else {
                        Duplicate the current path to match the parent set
                        Add each parent vertex to each path
                        Put the new paths on next level set
                }
        }
}

If DONE {
```

```
            Check each path on the next level list,

            remove any path whose top vertex is not start
            return the remaining next level list as the final list
      }
      Else {
            Remove all records in parent structure that has been used at this
            level.  This is necessary to ensure a correct reconstruction of
            the path.
            Recursive call buildPaths(start, parental structure, next level
            list)
      }
```

b. Each time an edge is used in a shortest path, its utilization score increments by 1;
   Edge utilization algorithm and data structure
   In order to efficiently count how many times a particular edge is used in a shortest path, I decide to store the edge as a string of the format "from vertex,to vertex". The "from vertex" is always going to be smaller than the "to vertex". For example, vertex 9 to vertex 1 is always going to be "1,9". A hash table (HashMap<String, Integer>) is used to map the edge to its utilization count.
   Each path (ArrayList<Integer>) is walked edge by edge, with the count of the edge incremented.
   The utilization counts of all edges are saved in a file.

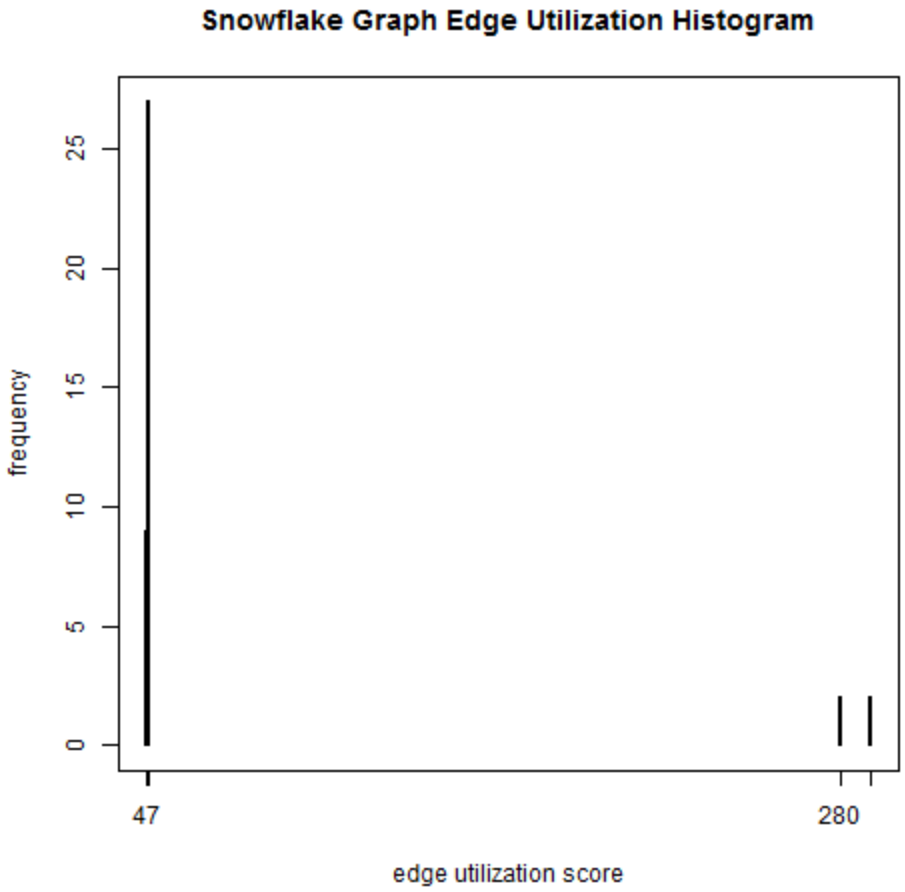
c. After all the shortest paths have been identified, select a set of edges based on their utilization scores (max in descending order);
   Edge utilization is sorted based on the count. The result (count: edges) is saved to a file.

d. Manually select the edges with scores greater than N to remove; (I have to play with N to see how many edges to remove and what communities I would end up with.)

e. Use SCC (strongly connected community) algorithm to identify members of communities;
f. Export the vertices of each community into a comma separated file as input to R (a statistical analysis program) to get the attribute signature of each community.

Trying the algorithm on the testing graph:

After running the community identifying algorithm, edges 1 to 10, 10 to 20, 20 to 30, and 0 to 30 have the highest utilization counts. The histogram of the edge utilization counts for the snowflake data set shows that a total of 4 edges have utilization count >= 280 (they are used most frequent), and rest of the edges have utilization count <= 47.



By removing these edges with high counts, 4 communities are created (code below).

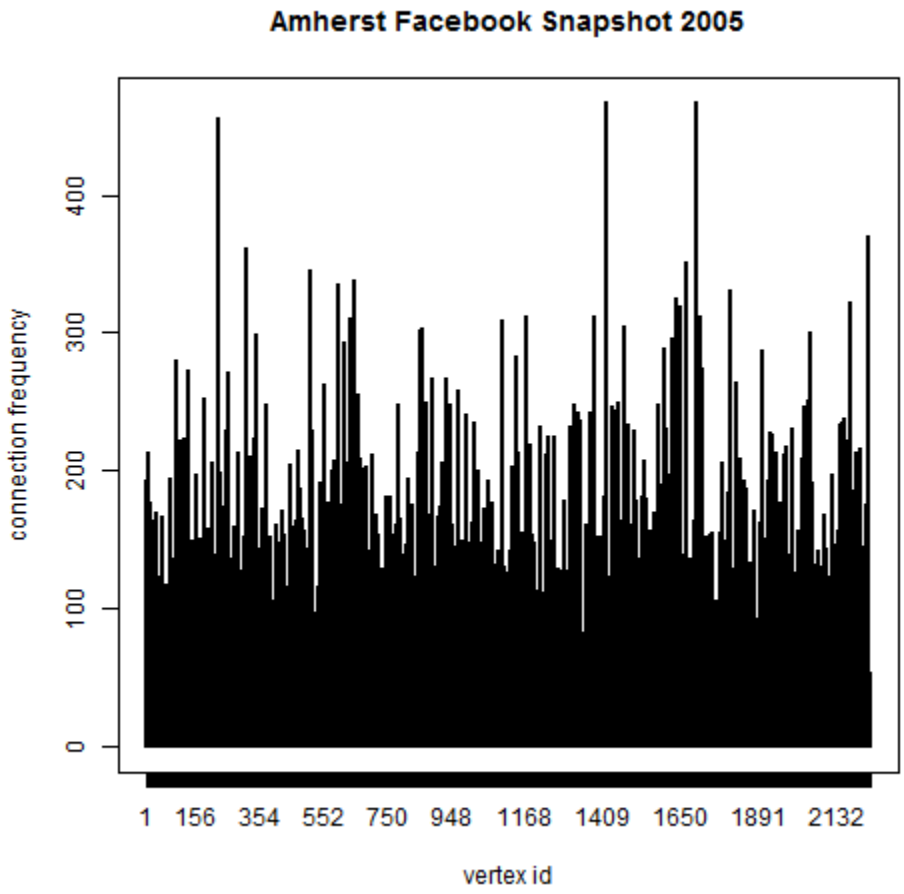
```
        public static void main(String[] args) {

          CapGraph graph = new CapGraph();
          GraphLoader.loadGraph(graph, "data/SnowflakeNetwork.txt");

           List<Graph> scc;
           // remove most used edges to create communities
          graph.createCommunities("data/edge_utilization.txt", 250);


           scc = graph.getSCCs();

           System.out.println("After remove most used edges SCC with graph counts: " +
scc.size());

           for (Graph g : scc) {
                      CapGraph cg = (CapGraph)g;
                      System.out.println(cg.getNumVertices());
           }
```

```
    }

Output (number of communities and each community in from vertex: to vertex format):
After remove most used edges SCC with graph counts: 4
32:30,
33:30,
34:30,
35:30,
36:30,
37:30,
38:30,
39:30,
30:32,33,34,35,36,37,38,39,31,
31:30,

20:21,22,23,24,25,26,27,28,29,
21:20,
22:20,
23:20,
24:20,
25:20,
26:20,
27:20,
28:20,
29:20,

16:10,
17:10,
18:10,
19:10,
10:16,17,18,19,11,12,13,14,15,
11:10,
12:10,
13:10,
14:10,
15:10,

0:1,2,3,4,5,6,7,8,9,
1:0,
2:0,
3:0,
4:0,
5:0,
6:0,
7:0,
8:0,
9:0,
```

Amherst Result:

1. Some people in Amherst College have more connections on Facebook than others. Some other people only have 1 connection. Below is a histogram with X axis as the vertex, and Y

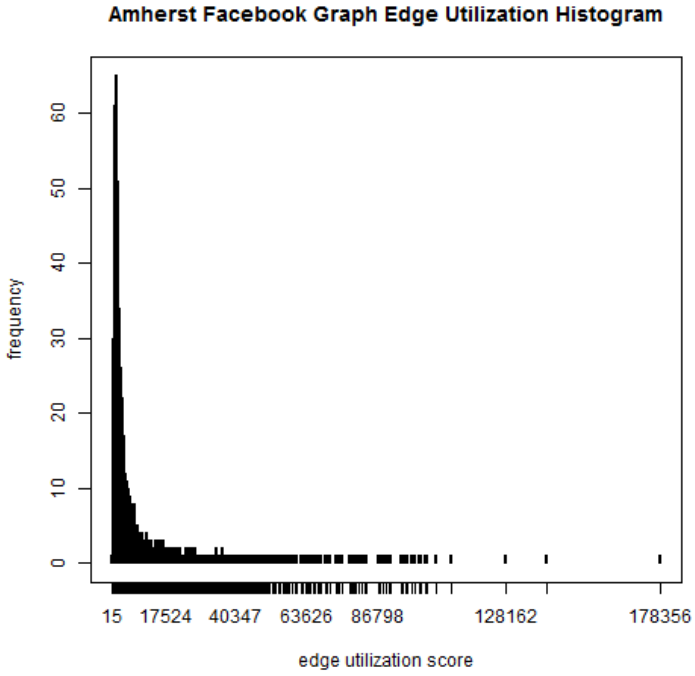
axis is how many connections it has. Because vertices are too crowded along X axis, the connection of 1 isn't graphically shown.

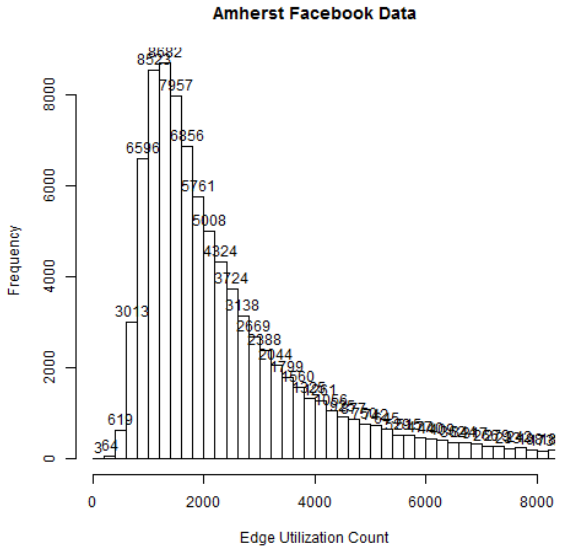


The R code that generated this histogram:

```
# this is the java loader format "from_vertex to_vertex". undirected graph has
the same edge twice with switched from and to
data_path <- c("C:\\Users\\LU\\java_workspace\\SocialNetworks\\data")
amherst_data <- read.table(paste(data_path, "Amherst41.txt", sep = "\\"),
header = FALSE, sep = " " )

# get a count of how many connections a vertex has
vertice_frequency <- table(amherst_data$V1)

# plot the connection frequency
plot(vertice_frequency, type = "h", xlab = "vertex id", ylab = "connection
frequency", main = "Amherst Facebook Snapshot 2005")
dev.copy(png, file = paste(data_path, "Amherst_vertex_hist.png", sep = "\\"))
dev.off()
```

2. Below are the “Amherst Facebook graph edge utilization histogram” and a partial histogram for the peak area.





I used the same routine on snowflake data to isolate communities in Amherst Facebook data. Initially, I set the cut off edge utilization count to 20000. 9 communities were created. 8 of the communities had just one vertex, and one had all the rest vertices. Then, I moved the cut off

value lower. More communities were created, but the pattern was the same, all except one community had a single vertex, and one had all the rest. This routine didn't produce communities with several members as in the case of snowflake test date.

I think the reason is that if communities exist, they are not like the snowflake test data where only one route connects the different communities. Most likely, there are "highways" and "local roads" that connect the communities. If the "highway" is removed, "local roads" are still there. Maybe the identification routine should be run multiple times: identify "highway", remove "highway", re-run identification routine, remove newly found "highways', …

Since the routine runs long (36 hours), I am out of time for the write up of this project. I will carry out this plan and update my result to this report outside the time span of this course.

Algorithm Analysis, Limitations and Risks

Since the community identification algorithm requires computing the shortest paths from vertex any vertex to any other vertex, given the number of vertices is N, the algorithm is of O(N!) time.

Each time, the computational magnitude is determined by the number of edges that connect the two vertices. It is O(E) (E as the number of edges) time.

So, combining these two factors together, the entire algorithm has a growth time of O(N! * E). Because of the challenge in the growth of computational time, this problem is a NP hard problem.