EECS E6893 Big Data Analytics - Homework Assignment 2

Name: Qi Wang

UNI: qw2261

Question 1. Friendship Recommendation Algorithm

In [1]:
```
from pyspark import SparkConf, SparkContext
import pyspark
import sys
from collections import defaultdict
```

In [2]:
```
# Configure Spark
sc = pyspark.SparkContext.getOrCreate()
# The directory for the file
filename = "gs://homework0_qi/HW2/q1.txt"
```

In [3]:

```python
def getData(sc, filename):
    """
    Load data from raw text file into RDD and transform.
    Hint: transfromation you will use: map(<lambda function>).
    Args:
        sc (SparkContext): spark context.
        filename (string): hw2.txt cloud storage URI.
    Returns:
        RDD: RDD list of tuple of (<User>, [friend1, friend2, ... ]),
        each user and a list of user's friends
    """
    # read text file into RDD
    data = sc.textFile(filename)

    # TODO: implement your logic here
    data = data.map(lambda line: line.split("\t"))
    data = data.map(lambda line: (int(line[0]), [int(each) for each in line[1].split(',')] if len(line[1]

    return data
```

In [4]:

```python
# Get data in proper format
data = getData(sc, filename)
```

In [5]:

```python
# Show the data structure of data collected
print(data.take(1))
```

```
[(0, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26,
27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 5
2, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 7
7, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94])]
```

In [6]:
```python
def mapFriends(line):
    """
    List out every pair of mutual friends, also record direct friends.
    Hint:
    For each <User>, record direct friends into a list:
    [(<User>, (friend1, 0)),(<User>, (friend2, 0)), ...],
    where 0 means <User> and friend are already direct friend,
    so you don't need to recommand each other.

    For friends in the list, each of them has a friend <User> in common,
    so for each of them, record mutual friend in both direction:
    (friend1, (friend2, 1)), (friend2, (friend1, 1)),
    where 1 means friend1 and friend2 has a mutual friend <User> in this "line"

    There are possibly multiple output in each input line,
    we applied flatMap to flatten them when using this function.
    Args:
        line (tuple): tuple in data RDD
    Yields:
        RDD: rdd like a list of (A, (B, 0)) or (A, (C, 1))
    """
    user = line[0]
    friends = line[1]
    for i in range(len(friends)):
        # Direct friend
        # TODO: implement your logic here
        yield((user, (friends[i], 0)))

        for j in range(i+1, len(friends)):
            # Mutual friend in both direction
            # TODO: implement your logic here
            yield((friends[i], (friends[j], 1)))
            yield((friends[j], (friends[i], 1)))
```

In [7]:
```python
# Get set of all mutual friends
mapData = data.flatMap(mapFriends).groupByKey()
```

```
In [8]: def findMutual(line):
    """
    Find top 10 mutual friend for each person.
    Hint: For each <User>, input is a list of tuples of friend relations,
    whether direct friend (count = 0) or has friend in common (count = 1)

    Use friendDict to store the number of mutual friend that the current <User>
    has in common with each other <User> in tuple.
    Input:(User1, [(User2, 1), (User3, 1), (User2, 1), (User3, 0), (User2, 1)])
    friendDict stores: {User2:3, User3:1}
    directFriend stores: User3

    If a user has many mutual frineds and is not a direct frined, we recommend
    them to be friends.

    Args:
        line (tuple): a tuple of (<User1>, [(<User2>, 0), (<User3>, 1)....])
    Returns:
        RDD of tuple (line[0], returnList),
        returnList is a list of recommended friends
    """
    # friendDict, Key: user, value: count of mutual friends
    friendDict = defaultdict(int)
    # set of direct friends
    directFriend = set()
    # initialize return list
    returnList = []

    # TODO: Iterate through input to aggregate counts
    # save to friendDict and directFriend

    for each_friend in line[1]:
        friendDict[each_friend[0]] += each_friend[1]
        if each_friend[1] == 0:
            directFriend.add(each_friend[0])

    result = sorted(friendDict.iteritems(), key = lambda x : x[1], reverse = True)

    # TODO: Formulate output
    result = sorted(friendDict.iteritems(), key = lambda x : x[1], reverse = True)
    count = 0
```

```
    for each in result:
        if each[0] not in directFriend and count < 10:
            returnList.append(each[0])
            count += 1

    return (line[0], returnList)
```

In [9]:
```
# For each person, get top 10 mutual friends
getFriends = mapData.map(findMutual)
```

In [10]:
```
# Only save the ones we want
wanted = [924, 8941, 8942, 9019, 49824, 13420, 44410, 8974, 5850, 9993]
result = getFriends.filter(lambda x: x[0] in wanted).collect()
```

In [11]:
```
# Visualize the result for check
for each in sc.parallelize(result).sortBy(lambda pair: pair[0]).collect():
    print(each)
```

```
(924, [43748, 2409, 6995, 11860, 439, 15416, 45881])
(5850, [5819, 5805, 5811, 5815, 5828, 5831, 5836, 30209, 13315, 13322])
(8941, [8943, 8944, 8940])
(8942, [8939, 8940, 8943, 8944])
(8974, [12241, 8960, 8774, 6973, 8969, 8982, 8980, 8984, 8979, 8978])
(9019, [9022, 317, 9023])
(9993, [9991, 13134, 34485, 34642, 13877, 34299, 37941, 13478])
(13420, [10469, 7651, 4736, 14264, 2101, 21869, 30403, 10500, 47366, 10532])
(44410, [4231, 44462, 28779, 22553, 14907, 10328, 10370, 17032, 28332, 6318])
(49824, [49846, 41581, 49786, 49788, 49789, 49814, 49819, 49834, 43382, 10760])
```

In [12]:
```
sc.stop()
```

Question 2. Graph Analysis

In [13]:
```
# Install graphframes
!pip install "git+https://github.com/munro/graphframes.git@release-0.5.0#egg=graphframes&subdirectory=py
```

```
DEPRECATION: Python 2.7 will reach the end of its life on January 1st, 2020. Please upgrade your Pyth
on as Python 2.7 won't be maintained after that date. A future version of pip will drop support for P
ython 2.7. More details about Python 2 support in pip, can be found at https://pip.pypa.io/en/latest/
development/release-process/#python-2-support (https://pip.pypa.io/en/latest/development/release-proc
ess/#python-2-support)
Requirement already satisfied: graphframes from git+https://github.com/munro/graphframes.git@release-
0.5.0#egg=graphframes&subdirectory=python in /opt/conda/anaconda/lib/python2.7/site-packages (0.5.0)
```

In [14]:
```
from graphframes import *
from pyspark import SQLContext
import os
```

In [15]:
```
# Configure Spark
if not os.path.isdir("checkpoints"):
    os.mkdir("checkpoints")
conf = SparkConf().setMaster("local").setAppName('connected components')
sc = SparkContext(conf = conf)
```

In [16]:
```
# Configure sqlcontext and directory
sqlContext = SQLContext(sc)
SparkContext.setCheckpointDir(sc, "checkpoints")
```

In [17]:
```
# Get data in proper format
data = getData(sc, filename)
```

In [18]:

```python
def getVertices(data, sqlContext):
    """
    Get the vertices of the friends network

    Args:
        data: RDD: RDD list of tuple of (<User>, [friend1, friend2, ... ]), each user and a list of user
        sqlContext: SQLContext

    Returns:
        vertice: ID DataFrame of all users
    """

    return sqlContext.createDataFrame(data.map(lambda line: (line[0], )), schema = ["id"])
```

In [19]:

```python
vertices = getVertices(data, sqlContext)
```

In [20]:
```
vertices.show()
```

```
+---+
| id|
+---+
|  0|
|  1|
|  2|
|  3|
|  4|
|  5|
|  6|
|  7|
|  8|
|  9|
| 10|
| 11|
| 12|
| 13|
| 14|
| 15|
| 16|
| 17|
| 18|
| 19|
+---+
only showing top 20 rows
```

In [ ]:

In [21]:
```
def friendRelationBuid(line):
    '''
    Get the egde of the friends network

    Args:
        line (tuple): a tuple of (<User1>, [(<User2>, 0), (<User3>, 1)....])

    Yields:
        friend relation (tuple): a tuple of (friend1, friend2)
    '''
    user = line[0]
    friends = line[1]

    for each_friend in friends:
        yield (user, each_friend)
```

In [22]:
```
def getEdges(data, sqlContext):
    edges = data.flatMap(friendRelationBuid)
    return sqlContext.createDataFrame(edges, schema = ["src", "dst"])
```

In [23]:
```
edges = getEdges(data, sqlContext)
```

In [24]:
```
edges.show()
```

```
+---+---+
|src|dst|
+---+---+
|  0|  1|
|  0|  2|
|  0|  3|
|  0|  4|
|  0|  5|
|  0|  6|
|  0|  7|
|  0|  8|
|  0|  9|
|  0| 10|
|  0| 11|
|  0| 12|
|  0| 13|
|  0| 14|
|  0| 15|
|  0| 16|
|  0| 17|
|  0| 18|
|  0| 19|
|  0| 20|
+---+---+
only showing top 20 rows
```

In [25]:
```
# Build graph
graph = GraphFrame(vertices, edges)
```

In [26]:
```
result = graph.connectedComponents()
```

In [27]:
```
result.show()
```

```
+---+---------+
| id|component|
+---+---------+
|  0|        0|
|  1|        0|
|  2|        0|
|  3|        0|
|  4|        0|
|  5|        0|
|  6|        0|
|  7|        0|
|  8|        0|
|  9|        0|
| 10|        0|
| 11|        0|
| 12|        0|
| 13|        0|
| 14|        0|
| 15|        0|
| 16|        0|
| 17|        0|
| 18|        0|
| 19|        0|
+---+---------+
only showing top 20 rows
```

(1). How many clusters/connected components in total for this dataset?

In [28]:
```
cluster_num = result.select("component").distinct().count()
print "The NO. of clusters are", cluster_num
```

```
The NO. of clusters are 917
```

(2). How many users in the top 10 clusters?

In [29]:
```
result.show()
```

```
+---+---------+
| id|component|
+---+---------+
|  0|        0|
|  1|        0|
|  2|        0|
|  3|        0|
|  4|        0|
|  5|        0|
|  6|        0|
|  7|        0|
|  8|        0|
|  9|        0|
| 10|        0|
| 11|        0|
| 12|        0|
| 13|        0|
| 14|        0|
| 15|        0|
| 16|        0|
| 17|        0|
| 18|        0|
| 19|        0|
+---+---------+
only showing top 20 rows
```

In [30]:
```
group_info = result.groupBy("component").count().orderBy('count', ascending = False)
```

In [31]:
```
top_ten = group_info.head(10)
```

In [32]:
```
user_num_ten = 0
for each_row in top_ten:
    user_num_ten += each_row["count"]
    print("Cluster %d has %d Users" % (each_row['component'], each_row['count']))
print("\nThere are %d users in the top 10 clusters" %user_num_ten)
```

```
Cluster 0 has 48860 Users
Cluster 38403 has 66 Users
Cluster 18466 has 31 Users
Cluster 18233 has 25 Users
Cluster 18891 has 19 Users
Cluster 864 has 16 Users
Cluster 49297 has 13 Users
Cluster 19199 has 6 Users
Cluster 7658 has 5 Users
Cluster 22897 has 4 Users

There are 49045 users in the top 10 clusters
```

(3). What are the user ids for the cluster which has 25 users?

In [33]:
```
target_cluster = group_info.where(group_info['count'] == 25).select("component").collect()
```

In [34]:
```
target_cluster = [row['component'] for row in target_cluster]
```

In [35]:
```
target_userid = [row['id'] for row in result.where(result['component'].isin(target_cluster)).select('id'
```

In [36]:
```
print('The user ids for cluster which has 25 users are: (the cluster is %s)'%target_cluster)
print(target_userid)
```

```
The user ids for cluster which has 25 users are: (the cluster is [18233])
[18233, 18234, 18235, 18236, 18237, 18238, 18239, 18240, 18241, 18242, 18243, 18244, 18245, 18246, 18
247, 18248, 18249, 18250, 18251, 18252, 18253, 18254, 18255, 18256, 18257]
```

(4). A list of 10 important users and the most important one

In [37]:
```python
page_rank = graph.pageRank(resetProbability = 0.15, tol = 0.01)
```

In [38]:
```python
page_rank
```

Out[38]:
```
GraphFrame(v:[id: bigint, pagerank: double], e:[src: bigint, dst: bigint ... 1 more field])
```

In [39]:
```python
ten_important_user = [row['id'] for row in page_rank.vertices.orderBy('pagerank', ascending = False).hea
```

In [40]:
```python
print("The 10 important users are \n")
print(ten_important_user)
print('\nThe most important one is %d' % ten_important_user[0])
```

```
The 10 important users are

[10164, 15496, 14689, 24966, 7884, 934, 45870, 5148, 20283, 46039]

The most important one is 10164
```

(5). Using different parameter settings for PageRank, is there any difference?

In [41]:
```python
# Using different parameters
page_rank = graph.pageRank(resetProbability = 0.1, maxIter = 30)
ten_important_user = [row['id'] for row in page_rank.vertices.orderBy('pagerank', ascending = False).hea
print("The 10 important users are \n")
print(ten_important_user)
print('\nThe most important one is %d' % ten_important_user[0])
```

```
The 10 important users are

[10164, 15496, 14689, 24966, 7884, 934, 45870, 20283, 46039, 14996]

The most important one is 10164
```

In [42]:
```
# Using different parameters
page_rank = graph.pageRank(resetProbability = 0.5, maxIter = 30)
ten_important_user = [row['id'] for row in page_rank.vertices.orderBy('pagerank', ascending = False).hea
print("The 10 important users are \n")
print(ten_important_user)
print('\nThe most important one is %d' % ten_important_user[0])
```

```
The 10 important users are

[10164, 15496, 14689, 24966, 5148, 38123, 7884, 934, 910, 44815]

The most important one is 10164
```

In [43]:
```
# Using different parameters
page_rank = graph.pageRank(resetProbability = 0.15, tol = 0.1)
ten_important_user = [row['id'] for row in page_rank.vertices.orderBy('pagerank', ascending = False).hea
print("The 10 important users are \n")
print(ten_important_user)
print('\nThe most important one is %d' % ten_important_user[0])
```

```
The 10 important users are

[10164, 15496, 14689, 24966, 5148, 38123, 934, 7884, 910, 44815]

The most important one is 10164
```

** Based on the tests, **

resetProbability = 0.15, tol = 0.01:

[10164, 15496, 14689, 24966, 7884, 934, 45870, 5148, 20283, 46039]

resetProbability = 0.1, maxIter = 30:

[10164, 15496, 14689, 24966, 7884, 934, 45870, 20283, 46039, 14996]

resetProbability = 0.5, maxIter = 30:

[10164, 15496, 14689, 24966, 5148, 38123, 7884, 934, 910, 44815]

resetProbability = 0.15, tol = 0.1:

[10164, 15496, 14689, 24966, 5148, 38123, 934, 7884, 910, 44815]

** we can determine that there are some differences for PageRank when using different parameter setting, which might result from that the tolerance is too large to converge within the given iterations. So we should set small tolerance and larger iterations to make the convergence happen.**

(6) Why this user become the most important one? What are the possible reasons?

The PageRank outputs a probability distribution which represents how likely a person/object will be selected randomly. So, the reason that this user become the most important one is that this user belongs to the largest cluster in the graph. Also, this user has the largest incoming and outcoming edges, from the edges the user will be as possible as much linked to other users, making him/her obtain the highest PageRank weight in the graph, like a important transportation center.

(7) PageRank Calculation

In [60]:
```
import numpy as np
```

In [89]:
```
def checkTol(prev, cur):
    for each in np.abs(prev - cur):
        if each > tol:
            return False
    return True
```

In [107]:

```
from copy import deepcopy
PR = {'ID1': 0.2, 'ID2': 0.2, 'ID3': 0.2, 'ID4': 0.2, 'ID5':0.2}
In = {'ID1': ['ID2'], 'ID2': ['ID3', 'ID5'], 'ID3': ['ID1', 'ID2', 'ID4', 'ID5'], 'ID4': ['ID2'], 'ID5'
L = {'ID1': 2, 'ID2': 4, 'ID3': 1, 'ID4': 1, 'ID5': 2}
N = 5
d = 0.85
tol = 0.1

result = []
prev = np.array(PR.values())
count = 0
while count == 0 or not checkTol(prev, np.array(PR.values())):
    prev = np.array(PR.values())
    cur_PR = deepcopy(PR)
    for each_user in ['ID1', 'ID2', 'ID3', 'ID4', 'ID5']:
        PR[each_user] = (1 - d) / N
        for each_In in In[each_user]:
            PR[each_user] = PR[each_user] + d * (cur_PR[each_In] / L[each_In])
        PR[each_user] = round(PR[each_user], 4)
    count += 1
    result.append(deepcopy(PR))
    print("Iteration " + str(count))
    print(PR)
    print
```

```
Iteration 1
{'ID4': 0.0725, 'ID5': 0.1575, 'ID2': 0.285, 'ID3': 0.4125, 'ID1': 0.0725}

Iteration 2
{'ID4': 0.0906, 'ID5': 0.1214, 'ID2': 0.4476, 'ID3': 0.2499, 'ID1': 0.0906}

Iteration 3
{'ID4': 0.1251, 'ID5': 0.1636, 'ID2': 0.294, 'ID3': 0.2922, 'ID1': 0.1251}

Iteration 4
{'ID4': 0.0925, 'ID5': 0.1456, 'ID2': 0.3479, 'ID3': 0.3215, 'ID1': 0.0925}
```

In [108]:
```
result
```

Out[108]:
```
[{'ID1': 0.0725, 'ID2': 0.285, 'ID3': 0.4125, 'ID4': 0.0725, 'ID5': 0.1575},
 {'ID1': 0.0906, 'ID2': 0.4476, 'ID3': 0.2499, 'ID4': 0.0906, 'ID5': 0.1214},
 {'ID1': 0.1251, 'ID2': 0.294, 'ID3': 0.2922, 'ID4': 0.1251, 'ID5': 0.1636},
 {'ID1': 0.0925, 'ID2': 0.3479, 'ID3': 0.3215, 'ID4': 0.0925, 'ID5': 0.1456}]
```

```
Except the code implementation, we can calculate this by hand:
PR = {'ID1': 0.2, 'ID2': 0.2, 'ID3': 0.2, 'ID4': 0.2, 'ID5':0.2}
In = {'ID1': ['ID2'], 'ID2': ['ID3', 'ID5'], 'ID3': ['ID1', 'ID2', 'ID4', 'ID5'], 'ID4': ['ID2'],
'ID5': ['ID1', 'ID2']}
L = {'ID1': 2, 'ID2': 4, 'ID3': 1, 'ID4': 1, 'ID5': 2}

Iteration 1:
Initial PR = {'ID1': 0.2, 'ID2': 0.2, 'ID3': 0.2, 'ID4': 0.2, 'ID5':0.2}
new_PR[ID1] = (1 - 0.85) / 5 + 0.85 * (PR[ID2] / L[ID2]) = 0.0725
new_PR[ID2] = (1 - 0.85) / 5 + 0.85 * (PR[ID3] / L[ID3] + PR[ID5] / L[ID5]) = 0.285
new_PR[ID3] = (1 - 0.85) / 5 + 0.85 * (PR[ID1] / L[ID1] + PR[ID2] / L[ID2] + PR[ID4] / L[ID4] + PR[ID5]
/ L[ID5]) = 0.4125
new_PR[ID4] = (1 - 0.85) / 5 + 0.85 * (PR[ID2] / L[ID2]) = 0.0725
new_PR[ID5] = (1 - 0.85) / 5 + 0.85 * (PR[ID1] / L[ID1] + PR[ID2] / :[ID2]) = 0.1575

After iteration 1, we have {'ID1': 0.0725, 'ID2': 0.285, 'ID3': 0.4125, 'ID4': 0.0725, 'ID5': 0.1575}
Compare to original PR, we find someone in the differences between iteration 1 and original PR is
larger than tolerance and we should continue on next iteration.

Iteration 2:
Initial PR = {'ID1': 0.0725, 'ID2': 0.285, 'ID3': 0.4125, 'ID4': 0.0725, 'ID5': 0.1575}
new_PR[ID1] = (1 - 0.85) / 5 + 0.85 * (PR[ID2] / L[ID2]) = 0.0906
new_PR[ID2] = (1 - 0.85) / 5 + 0.85 * (PR[ID3] / L[ID3] + PR[ID5] / L[ID5]) = 0.4476
new_PR[ID3] = (1 - 0.85) / 5 + 0.85 * (PR[ID1] / L[ID1] + PR[ID2] / L[ID2] + PR[ID4] / L[ID4] + PR[ID5]
/ L[ID5]) = 0.2499
new_PR[ID4] = (1 - 0.85) / 5 + 0.85 * (PR[ID2] / L[ID2]) = 0.0906
new_PR[ID5] = (1 - 0.85) / 5 + 0.85 * (PR[ID1] / L[ID1] + PR[ID2] / :[ID2]) = 0.1214

After iteration 2, we have {'ID1': 0.0906, 'ID2': 0.4476, 'ID3': 0.2499, 'ID4': 0.0906, 'ID5': 0.1214}
Compare to last PR, we find someone in the differences between iteration 2 and last PR is larger than
tolerance and we should continue on next iteration.

Iteration 3:
Initial PR = {'ID1': 0.0906, 'ID2': 0.4476, 'ID3': 0.2499, 'ID4': 0.0906, 'ID5': 0.1214}
new_PR[ID1] = (1 - 0.85) / 5 + 0.85 * (PR[ID2] / L[ID2]) = 0.1251
```

```
new_PR[ID2] = (1 - 0.85) / 5 + 0.85 * (PR[ID3] / L[ID3] + PR[ID5] / L[ID5]) = 0.294
new_PR[ID3] = (1 - 0.85) / 5 + 0.85 * (PR[ID1] / L[ID1] + PR[ID2] / L[ID2] + PR[ID4] / L[ID4] + PR[ID5]
/ L[ID5]) = 0.2922
new_PR[ID4] = (1 - 0.85) / 5 + 0.85 * (PR[ID2] / L[ID2]) = 0.1251
new_PR[ID5] = (1 - 0.85) / 5 + 0.85 * (PR[ID1] / L[ID1] + PR[ID2] / :[ID2]) = 0.1636

After iteration 3, we have {'ID1': 0.1251, 'ID2': 0.294, 'ID3': 0.2922, 'ID4': 0.1251, 'ID5': 0.1636}
Compare to last PR, we find someone in the differences between iteration 3 and last PR is larger than
tolerance and we should continue on next iteration.

Iteration 4:
Initial PR = {'ID1': 0.1251, 'ID2': 0.294, 'ID3': 0.2922, 'ID4': 0.1251, 'ID5': 0.1636}
new_PR[ID1] = (1 - 0.85) / 5 + 0.85 * (PR[ID2] / L[ID2]) = 0.0925
new_PR[ID2] = (1 - 0.85) / 5 + 0.85 * (PR[ID3] / L[ID3] + PR[ID5] / L[ID5]) = 0.3479
new_PR[ID3] = (1 - 0.85) / 5 + 0.85 * (PR[ID1] / L[ID1] + PR[ID2] / L[ID2] + PR[ID4] / L[ID4] + PR[ID5]
/ L[ID5]) = 0.3215
new_PR[ID4] = (1 - 0.85) / 5 + 0.85 * (PR[ID2] / L[ID2]) = 0.0925
new_PR[ID5] = (1 - 0.85) / 5 + 0.85 * (PR[ID1] / L[ID1] + PR[ID2] / :[ID2]) = 0.1456

After iteration 4, we have {'ID1': 0.0925, 'ID2': 0.3479, 'ID3': 0.3215, 'ID4': 0.0925, 'ID5': 0.1456}]
Compare to last PR, we find  the differences between iteration 3 and last PR are all smaller than
tolerance and the convergence happen, and our result is the same as the code running.
```