INFO290 Midterm Project Report
Project Title: Party Guest Recommender

Group
Sarah Friedman (sarah_friedman@mba.berkeley.edu / 2 units),
Genevieve Wang (genevieve_wang@mba.berkeley.edu / 2 units), and
Jeff Williamson (jeff_williamson@mba.berkeley.edu / 2 units)

Abstract
For this project, we attempt to develop a solution that identifies the optimal group of people to invite to a party, where the objective of the party is to encourage the development of new connections between guests. We also attempt to identify the handful of people who must commit to attend the party first, in order to maximize yield (positive response rate) from the remaining invitees, while minimizing the time/effort spent by the host.

Project Background (Revised)
We are interested in two main research questions: 1) how to design a party guest list that best enables new connections to be made at a party and 2) how to gather positive responses (RSVPs) from guests as quickly as possible, with as little effort on the part of the party creator as possible. This project focuses on larger parties, where the host's motivation is to create a great environment for guests to meet and mingle – for example, nonprofit fundraisers, tailgates, or networking events. In this project, we focus on a party of 50 people, and we assume that all 50 guests are friends with the host of the party. We use Facebook friendships as a proxy for real-life connections.

The first question is interesting to us, because one of the main motivations for people to attend large parties is to meet new people. However, if left to the host's subjective selection of guests, the party may not end up with the optimal group of attendees. For instance, the host may select 50 people who are mostly connected to each other already (a social network with very high density). In this situation, there are few opportunities to make new connections. Or, the host may not know of commonalities that his/her friends may have with one another, which we hypothesize increase people's likelihood of forming connections with one another. We believe that we can optimize the selection process with the power of social network data analysis.

Regarding the second research question, our primary research with people has shown that, when invited to a party, people commonly wait for others they know to sign up for the party before committing to attend themselves.[1] This results in a chicken-and-egg problem for the first few people who sign up. However, once a party reaches a tipping point of a certain size/number of attendees, more people will readily sign up. It is in the party creator's interest to get RSVPs back as quickly as possible, as the number of attendees is a key piece of information required for planning the party. As a result, party

[1] In almost all event planning websites (e.g. Facebook Events, Evite, Eventbrite), the default is to allow guests to see who else has already signed up to attend the event.

creators often need to actively market their parties to early adopters - via email, Facebook message, etc. – to get a committed "base group" for the party. This process, however, can be haphazard, unsystematic, and inefficient. Our objective is to help the party creator identify the handful of people he/she must market to and get to RSVP first, such that other invitees will respond as quickly as possible.

For this project we are interested in developing a systematic way for any party creator to:

(1) PART I: Identify the 50 best people to come to any party, if the objective of the host is to have as many attendees make as many new connections as possible, and
(2) PART II: Identify the key people in the party creator's social network who need to commit to the party first, such that the rest of the invitees sign up as quickly as possible.

Please note that we have extended our thinking about this project since the submission of our original project proposal. Originally, we had scoped the project primarily around PART II – identifying the few key people who need to commit first to a party, while optimizing for the quality of the party. As we explored our technical execution options, we realized that our best approach involved defining and developing a tool for creating the optimal party guest list first. To this end, we have focused on "making new connections," as the primary party objective to solve for, and we define the optimal guest list as one that is most conducive to guests making new connections.

If successful, this project may be integrated as a feature of Rakoko[2], a web application that makes it easier to plan social events with friends.

PART I

Original Solution Approach

In this section, we will briefly review the high-level solution approach that we attempted to execute for PART I of our project, creating an optimal guest list of 50 people. We have been able to execute closely to this vision, with a few exceptions. In the "Problems Encountered" section of this paper, we will review in detail the technical issues that we encountered, and the correctional measures we took to address those issues. (We will refer to the party creator as Host).

1. Access Host's Facebook Data
   Access information about Host's demographic characteristics and social graph using Facebook's Graph API (through the pyfacebook Python wrapper). Upon accessing our SocialComputing Facebook app, the user is asked for permission to access his/her information. Please note that the current version of the program is designed to only work with team member accounts[3], but it will be easy to update the program to work with any Facebook account once core functionality is complete.
2. Evaluate Guest List Quality

[2] Rakoko is a start-up project that Genevieve and Sarah are working on outside of class; much of the early-stage customer research we conducted for Rakoko led to the motivation for this project; no technical work to solve this problem was performed before the beginning of this course.
[3] For most of the project, we have focused on using data from Genevieve's Facebook account, thus assuming that Genevieve is the party Host.

For a given combination of 50 friends of the Host, we need to determine how many new connections will be made between guests at the party.

a. **Potential new connections:** There are two inputs that determine whether there exists a potential new connection between two people. First, we look at the social graph and determine if there is an existing connection (there should not be an existing connection). Secondly, we evaluate user data about religion, teams and brand likes, hometown, past employers, etc., to see if the two people have something in common (e.g., they both list Boston as their hometown). If they have at least one item in common, we consider this a "potential new connection."
b. **Potential new connection score:** We quantify the strength/quality of each "potential new connection" by summing up the number of things the two people have in common and taking the ln(1+ number of things in common). This yields the *Individual Connection Score*. The sum of all the individual connection scores yields the *Group Connection Score* for the set of 50 people. This sum is log-transformed so that each additional commonality between two people will improve the connection score an ever-decreasing amount. This ensures that the value (Group Connection Score improvement) of adding a new pair of people with only one commonality will be greater than the value of a new connection pair finding one additional commonality.

3. Iterate to Find the Group with the Highest Group Connection Score
   Evaluate all possible combinations of 50 friends and identify the group of 50 friends with the highest Group Connection Score (the maximum).

4. Test for Further Party Quality Optimization
   Optimize for other descriptive statistics, e.g., targeting 50% male and female.
   a. We use weights on each variable to account for the importance of the new metric introduced (e.g., "group connection score" is more important than a perfect 50/50% mix of males and females, so we weight the former factor more).
   b. We will test weights with surveys. We will ask real people on the guest list to rate the party guest list generated against a randomly generated guest list.

Execution/Progress Update

In this section, we will review in detail the progress we have made in executing PART I.

Build Application/Connect to Facebook - Completed

Our **SocialComputing application is able to successfully connect to Facebook, request the appropriate user permissions, acquire an authorization token, and initiate an authenticated session**. Because the official Facebook API does not include a Python interface, we rely on pyfacebook for the bulk of the interactions with the website. However, the pyfacebook wrapper does not implement all of the methods that the Facebook API makes available, so where appropriate we rely on direct URL calls to obtain the necessary information (e.g. we cannot request extended user permissions through pyfacebook).

Obtain Host's Friends Data - Completed

Once connected, we access the Host's friends list to identify friends in the same current geographic area (currently hard-coded for San Francisco, Oakland, and Berkeley).[4] From this list, we filter for only those friends who have granted access to all of the following pieces of information: hometown, current/former employers, and current/former educational institutions. We also need to know which of the Host's friends are already connected to each other, so that we can identify possibilities for new connections.

Unfortunately, Facebook does not make available the list of friends of the Host's friends through the API, even though that information is readily available through the web interface. We spent a great deal of time trying to scrape this information from the web page itself, but because of the dynamic loading nature of Facebook, and possibly the intentional obfuscation of its design, we were unsuccessful with this approach[5]. We later realized that we could obtain most of what we needed through the "mutual friends" attribute, which Facebook does make available through the API. This property indicates if two of the Host's friends are connected. The limitation of this approach is that all of the invitees must already be friends with the Host, so the Host him/herself will not be able to make any new connections at the party.

Determine Group Connection Score for Any Party Guest List - Completed

We have finalized the algorithms and statistical techniques that we plan to use to define an optimal party guest list. Our objective was to create an elegant, systematic definition that we could leverage our code to evaluate and replicate. We identified that using Facebook data to look for commonalities in people with no prior Facebook connection would be a good way to identify people who would enjoy connecting at a party. We then identified a way to calculate the number of commonalities based on Facebook data. We use the natural log to give extra weight to strong potential relationships, as described above.

Develop and Build Method for Optimizing Party Guest List – In Progress

Using a greedy algorithm approach, we have built a program that iterates through different combinations of 50 people by selectively adding and subtracting potential invitees one at a time to see how that affects the overall Group Connection Score. For each iteration, the program removes the person with the lowest Individual Connection Score from the group of 50. From the remaining pool of potential invitees, the program selects the person with the highest Individual Connection Score (specific to the group being tested) and adds that person to the party guest list. With this new set of 50 people, the program iterates again to see if dropping the person with the lowest score and adding a new person will increase the score even further. Using this approach, our program is not guaranteed to find a global maximum, but it will find a local maximum if it continues to iterate until no additional person increases the Group Connection Score. Or, for practicality's sake, it could continue to iterate until the score increases by less than a specified threshold (e.g. 5%).

Please note that this searching algorithm has a strong potential of being the program bottleneck, so our team is anticipating having to factor the code, so that these parts are written in a fast compiled language like C and called from the main Python program. Even with this approach, the algorithm as described may take too long to be feasible, so our team is prepared to revisit this key part of our program to explore other options.

[4] We made a conscious decision to restrict to the Host's friends in the same geography, given that this is a realistic constraint in planning a party guest list. This decision also results in fewer computations.

[5] We thank Nate Murray for his time advising our team about ways to try solving this problem.

Develop and Execute Plan for Testing Quality of Solution – In Progress

We want to know how well our tool works in a real context. We know that people have trouble answering questions like "on a scale of 1 to 10, how good is [X]," but are much better at comparing. Therefore, we want to create a testing system that allows people to compare the party our system creates versus a random party. This way, we as a team can be sure that we are creating a preferable guest list to a random draw.

We will test two versions of our tool, likely testing two different weight levels for the "group connection score" vs. a gender split score. When testing the two versions of our tool, there will be some people who show up in both guest lists generated. We propose asking those people which of the two parties they prefer, in addition to having them compare each to a random sample. This means we should have a sample of 50 who rated each guest list against a random sample and 20-30 people who rated the two generated guest lists against one another.

We will begin by running the tool on Genevieve's friends list and surveying Genevieve's friends selected for the guest list(s). We would then execute the same test with Sarah's friends list and with a few test users (who would assist in distributing the survey to their friends). In the end, we aim to have surveyed both versions of the tool with 2-5 hosts and their potential guest lists to have directional results as to which version is the best. An additional potential test will be to run the algorithm on many individuals' Facebook accounts to see if they, as party hosts, like the suggested party guest list that the tool creates. This would enable us to test more people, but we believe the results will less conclusive[6].

Detail on Problems Encountered and Responses

As we have attempted to execute on our original solution approach, we have come across a number of challenges. In this section, we review the key challenges and our team's responses to those challenges.

- Problem #1: massive data processing required to analyze all possible combinations of guest lists of 50 people.
    - Response: We cannot include "friends of friends" in the guest list; this makes the pool of potential guests far too large.
    - Response: People will only get together if they are in same geography. For this test we will limit party guests to only friends currently located in the Host's geographic area. Please note that we have currently hard-coded this modification; in the long run we would want to build a more robust, flexible solution.

- Problem #2: We tried to manually scrape connections data by going to each friend's "friend page", and then going to the source code. Unfortunately, the source code page does not list all of the friend's friends. The source code only has the code for the frame, and not all of the content inside the friends-of-friend list. We also tried other technical approaches including: URL open, open with cookies set, and the curl. We also consulted with both Natarajan Chakrapani and Nate Murray. Nate gave us some helpful recommendations, but warned that Facebook does not actually want users getting a full graph. (To that point, we could not make any of his recommendations work.)
    - Response: We were able to obtain a list of mutual friends without scraping. Since we have already decided to constrain potential guests to only the Host's friends (and not

[6] It may also be challenging to obtain Facebook permissions from a large number of people.

friends of friends), this gets us enough information to know all potential connections between anyone that can be chosen for the group of 50.

- Problem #3: The Group Connection Score reflects double-counting, since, for each person, we count their "individual connection score" with every single other person in the group.
    - *No change required*: While we are counting twice, because of the nature of Facebook, all relationships are symmetrical, so all scores will be twice as high as they should be. Thus, so long as we are comparing within this set of scores, this should not be a problem. (We will need to keep this in mind if we choose to change something else in the scoring system that changes the symmetric nature of the data.)

- Problem #4: How do we pick the first group of 50 friends to kick off the iterative optimization process?
    - *Response*: We start with friends that have the fewest mutual friends with the host, as a percentage of their total friends. This is because friends with the fewest mutual friends with the host have the most potential to make new connections at the part.

- Problem #5: Even with restricting the potential pool to the Host's geography, running through every single combination of 50 friends is not feasible. If we cannot evaluate the Group Connection Score for every group of 50, how can we find the optimal group?
    - *Response*: Professor Irwin King directed us to study the greedy algorithm approach as a potential solution to this problem. We accept that using the greedy algorithm approach will only help us find a local maximum, not the global optimal solution (which can only be achieved by exhaustively running through every single combination).

      Having studied this approach, we have devised a solution. From the starting guest list of 50, and we will remove the person with the lowest Individual Connection Score. We will then calculate the Individual Connection Score for everyone in the total friend pool to identify the best person to add to the guest list. We then re-sort the guest list by Individual Connection Score and exclude the new person with the lowest score. We will iterate this process to move closer and closet to the local maximum.

- Problem #6: Following the response to Problem #5, at what point should we stop trying to iterate to improve the Group Connection Score?
    - Response: We are considering two methods. The first is following the greedy algorithm approach: we will loop through all possibilities an unlimited number of times until we have reached a local maximum (or perhaps until we've reached a minimal level of incremental improvement). The other method is using a score threshold. After we have conducted testing, we may be able to identify a Group Connection Score, above which we have a "great party". With this information, we can iterate the optimization process until the guest list of 50 meets this threshold score.

- Problem #6: It is not feasible to evaluate the Individual Connection Score using all factors available via Facebook user data.
    - Response: We will start with the three factors that we believe are the most likely to drive new connections (and are readily available across most users). These three factors are: hometown location ID, work employer ID, education school ID. We realize that an optimization exercise for our algorithm in the future could be testing adding other

commonality factors (and perhaps removing ones that do not, in reality, support the creation of new connections).

- Problem #7: Due to some people's Facebook privacy settings, we cannot gather all pieces of data for each person.
    - No change: We have accepted this to be a limitation of our method. We have ensured that we do not count two "blanks" as a match, and we realize that we are missing some potential new connections due to data imperfections.

Next Steps

Our next steps for finalizing PART I of our project are focused on potentially improving the performance of the search algorithm and testing the quality of our solution, as described above.

PART II

The majority of our time in the remainder of the semester will be focused on executing PART II of our project. PART II follows PART I. After we have identified our optimal party guest list of 50 people, we then determine how the Host can get the greatest number of people to sign up for the party as quickly as possible, while expending the least amount of effort. Please note that we are holding the "availability" and "convenience" variables constant – we assume that all invitees are available on the date and time of the event and that the location of the event does not pose a deterrent.

Solution Approach and Planned Next Steps

1. We assume that the effort expended by the Host increases linearly, with each additional invitee whom he/she must initially convince to sign up for (RSVP "yes" to) the party. We also assume that sending out reminders to invitees about the party takes effort.

2. We will explore centrality and reachability as our two key social graph metrics for evaluating the level of influence that an individual invitee has in getting additional party guest invitees to sign up for the party. We will call the initial subset of guests whom the Host convinces to sign up for the party, Group A.
    a. Simple degree centrality may make sense as a starting point for measuring influence, since degree centrality indicates which people within the group of 50 have the highest number of direct connections to the other party guest invitees.[7] We hypothesize that direct connections will be a source of influence. For example, the Host convinces a small group of people (Group A) with high degree centrality to sign up for the party before the invitation goes out to the rest of the guest list. When other invitees receive the party invitation, they will see that they already have a friend who has signed up to attend, and they will be more likely to respond "yes". Note: we intend to refine this

[7] Betweenness centrality and closeness centrality, while useful in other situations, is less relevant for our context, given that an individual may have no idea what his/her geodesic (shortest path) distances are to other people on the guest list.

measure to arrive at an overall degree centrality score for Group A; e.g. we wouldn't necessarily focus on the top three guests with the highest individual degree centrality, but rather focus on the group of three that collectively connect to the greatest number of guests.

b. Exploring reachability will allow us to explore what happens if we encourage cascading influential behavior. The Host convinces Group A to sign up for the party. If Group A influences all of their friends on the guest list to attend (let's call these friends Group B), then Group B will influence all of their friends as well, and so on. Again, we need to evaluate based on an overall group reachability score (rather than looking at individual scores). Please note that this approach will take longer and may require additional reminders and updates from the host, given that we may be working with greater distances between people and thus a cascading, non-simultaneous effect.

3. We will build a solution algorithm that identifies the people on the party guest list with the greatest influence. We may explore two kinds of solutions:
    a. The optimal set of people to pre-invite if the Host has a one-shot deal to get invitees to sign up (either because of time or attention – e.g. the party is happening tonight or the Host has only one chance to entice invitees to read and respond the invitation).
    b. The optimal set of people (Group A) for the Host to pre-invite, if there is time and opportunity for the cascading effect to occur among guests.

    Note: We would also like to explore and attempt to incorporate the concept of a tipping point. For example, is there a point at which the party has gathered so much momentum (positive response rate), such that all invitees are apt to respond positively?

4. We will test our hypothesis and solution by surveying the people in our party guest list. For example, we may ask them, "If you knew these people were going to this party, how willing are you to attend?" As in PART I, we intend to test against a random selection of people as a control measure. We may need to conduct a few rounds of surveys to account for the cascading effect of the increasing guest list over time (from Group A to Group B).

References

Borgatti, Stephen. "Graph Theory."

Chen, Jilin; Werner Geyer; Casey Dugan; Michael Muller; Ido Guy. "Make New Friends, but Keep the Old – Recommending People on Social Networking Sites." 2009.

Cook, D. J. and L. B. Holder. *Mining Graph Data*, 1st ed. Wiley-Interscience, 2006.

Cormen, Thomas H.; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein. "Chapter 16: Greedy Algorithms." *Introduction to Algorithms*. The MIT Press, 2001.

Hanneman, Robert A. and Mark Riddle. *Introduction to social network methods*. University of California, Riverside, 2005.

Appendix: SocialComputing Application Code

```
#!C:\Python26\python.exe

# INFO290: Social Computing
# Group Project
# Project Title: PartyAwesome
# Members: Sarah Friedman, Genevieve Wang, Jeff Williamson
#
# Description:
#   Part 1: Map facebook social graph of logged-in user

import facebook #python wrapper for facebook API
import webbrowser #so we can open URLs directly through browser
from django.utils.encoding import smart_str, smart_unicode #so we can handle unicode strings properly
import time

#SocialComputing App facebook IDs
my_api_key = "258927080808294"
my_secret_key = "[redacted]"

#get facebook object and authentication token
f = facebook.Facebook(api_key=my_api_key, secret_key=my_secret_key)
f.auth.createToken()

# Show login window
# Set popup=True if you want login without navigational elements
# TODO: find way of not having to login every time; maybe through cached cookies?
f.login()

# Login to the window, then press enter
# This step is only necessary because we need to wait till after user is logged
# in and we can't tell when that's happened, so we wait for user input.
print 'After logging in, press enter...'
raw_input()

f.auth.getSession()
friends = f.friends.get()


# Get all user attributes from all friends of current user and save to Excel
# file to see what information we have to work with
from pyexcelModule import pyexcel as xl
wb = xl.Workbook()


# Get all non-array info for friends
sht = wb.add_sheet("FriendsInfo")
# Add column headers
sht.writerow(0, 0, items)
i=0
for friend in friendInfo:
    line = []
    friend['uid'] = smart_str(friend['uid']) #make into string b/c Excel 2002/03 can't handle big longs
    if friend['current_location']:
        friend['current_location'] = friend['current_location']['city'] + ', ' +
    friend['current_location']['country']
    if friend['hometown_location']:
        friend['hometown_location'] = friend['hometown_location']['city'] + ', ' +
    friend['hometown_location']['country']
    for item in items:
        line.append(friend[item])
    i += 1
    for x in range(len(line)):
        if type(line[x]) is long or type(line[x]) is int:
            line[x] = smart_str(line[x])
    sht.writerow(i, 0, line)


# Add array info; 1 sheet per item
items = ['uid','name', \
         'affiliations','meeting_sex','meeting_for','family','work','education','sports', \
         'favorite_athletes','favorite_teams','inspirational_people','languages']

i=0
friendInfo = []
while i <= len(friends):
    friendInfo.append(f.users.getInfo(friends[i:(i + 200 - 1)], items))
    i += 200
    time.sleep(2)
```

```
#flatten friendInfo list
import itertools
friendInfo = list(itertools.chain.from_iterable(friendInfo))


def ProcessPropertyList(item, subItem, structure, line):
    if type(subItem) is dict:
        for property in subItem:
            if type(subItem[property]) is dict:
                for key in subItem[property].keys():
                    colname = smart_str(item) + '_' + smart_str(property) + '_' + key
                    if not colname in structure:
                        structure[colname] = []
                        structure[colname].append(smart_str(property))
                        structure[colname].append(smart_str(key))
                        structure[colname].append(len(structure))
                        structure[colname].append('dict')
            elif type(subItem[property]) is list:
                for x in range(len(subItem[property])):
                    # assume this is a dictionary
                    for key1 in subItem[property][x].keys():
                        if type(subItem[property][x][key1]) == list:
                            for y in range(len(subItem[property][x][key1])):
                                # assume this is a dictionary
                                for key2 in subItem[property][x][key1][y].keys():
                                    colname = smart_str(item) + '_' + smart_str(property) + '_' + \
                                              str(x) +'_' + key1 + '_' + str(y) + '_' + key2
                                    if not colname in structure:
                                        structure[colname] = []
                                        structure[colname].append(smart_str(property))
                                        structure[colname].append(smart_str(key1))
                                        structure[colname].append(len(structure))
                                        structure[colname].append('list2')
                                        structure[colname].append(x)
                                        structure[colname].append(smart_str(key2))
                                        structure[colname].append(y)
                        elif type(subItem[property][x][key1]) == dict:
                            for key2 in subItem[property][x][key1].keys():
                                colname = smart_str(item) + '_' + smart_str(property) + '_' + \
                                          str(x) +'_' + key1 + '_' + key2
                                if not colname in structure:
                                    structure[colname] = []
                                    structure[colname].append(smart_str(property))
                                    structure[colname].append(smart_str(key1))
                                    structure[colname].append(len(structure))
                                    structure[colname].append('listDict')
                                    structure[colname].append(x)
                                    structure[colname].append(smart_str(key2))
                        else:
                            colname = smart_str(item) + '_' + smart_str(property) + '_' + str(x) +'_' + key1
                            if not colname in structure:
                                structure[colname] = []
                                structure[colname].append(smart_str(property))
                                structure[colname].append(smart_str(key1))
                                structure[colname].append(len(structure))
                                structure[colname].append('list1')
                                structure[colname].append(x)
            else:
                colname = smart_str(item) + '_' + smart_str(property)
                if not colname in structure:
                    structure[colname] = []
                    structure[colname].append(smart_str(property))
                    structure[colname].append(None)
                    structure[colname].append(len(structure))
                    structure[colname].append('simpleDict')
    else:
        colname = smart_str(item)
        if not colname in structure:
            structure[colname] = []
            structure[colname].append(None)
            structure[colname].append(None)
            structure[colname].append(len(structure))
            structure[colname].append('simpleType')

    for property in sorted(structure.iteritems(), key=lambda x: x[1][2]):
        if property[1][3] == 'simpleDict':
            line.append(subItem.get(property[1][0],None))
        elif property[1][3] == 'dict':
            line.append(subItem.get(property[1][0],{}).get(property[1][1],None))
        elif property[1][3] == 'list1':
                try:
```

```
                    line.append(subItem[property[1][0]][property[1][4]][property[1][1]])
                except:
                    line.append(None)
        elif property[1][3] == 'list2':
            try:

    line.append(subItem[property[1][0]][property[1][4]][property[1][1]][property[1][6]][property[1][5]])
            except:
                line.append(None)
        elif property[1][3] == 'listDict':
            try:
                line.append(subItem[property[1][0]][property[1][4]][property[1][1]][property[1][5]])
            except:
                line.append(None)
        elif property[1][3] == 'simpleType':
            line.append(subItem)
    return line

for item in items:
    if item == 'uid' or item == 'name': continue
    sht = wb.add_sheet(item)
    structure = {}
    i=0
    for friend in friendInfo:
        if friend[item]:
            for subItem in friend[item]:
                line = []
                line.append(friend['uid'])
                line.append(friend['name'])
                ProcessPropertyList(item, subItem, structure, line)
                for x in range(len(line)):
                    if type(line[x]) is long or type(line[x]) is int:
                        line[x] = smart_str(line[x])
                i += 1
                sht.writerow(i, 0, line)

    # Make column headers
    line=[]
    line.append('uid')
    line.append('name')
    for property in sorted(structure.iteritems(), key=lambda x: x[1][2]):
        line.append(property[0])
    sht.writerow(0, 0, line)

wb.save("dbgOutput.xls")

# Process friends list to calculate social connection scores

########################################################
# Step 1. Filter for San Francisco / Berkeley / Oakland
########################################################
# Location IDs:
# Oakland, CA: 108363292521622
# San Francisco, CA: 114952118516947
# Berkeley, CA: 113857331958379
frLoc = f.users.getInfo(friends, ['uid','current_location'])
friendsF = []
for fr in frLoc:
    if fr['current_location'] and 'id' in fr['current_location'] \
        and fr['current_location']['id'] in [108363292521622L,114952118516947L,113857331958379L]:
                friendsF.append(fr['uid'])


#####################################################
# Step 2. Get relevent info for filtered friends list
#####################################################
items = ['uid','name','sex','current_location','hometown_location','friend_count', \
         'mutual_friend_count','work','education']

i=0
friendInfo = []
while i <= len(friends):
    friendInfo.append(f.users.getInfo(friendsF[i:(i + 200 - 1)], items))
    i += 200
#flatten friendInfo list
import itertools
friendInfo = list(itertools.chain.from_iterable(friendInfo))
#identify which friends have all the info we're looking for
x=0
for fr in friendInfo:
    fr['fullInfo'] = True
```

```
    for i in items:
        if i not in fr.keys() or fr[i] is None or fr[i] == [] or (i=='hometown_location' and 'state' not in
    fr[i].keys()):
            fr['fullInfo'] = False
            break

###############################################################################
# Step 3. Choose the 50 friends with the least number of shared connections
# as percentage of total connections
###############################################################################
#calculate connection %
#friendInfo[:]['mutual_connect_pct'] = 1.0  why doesn't this work?
for fr in friendInfo:
    if fr['fullInfo']:
        fr['mutual_connect_pct'] = fr['mutual_friend_count'] / float(fr['friend_count'])
    else:
        fr['mutual_connect_pct'] = 1.0


friendInfo.sort(key=lambda fr: fr['mutual_connect_pct'])

friendGroup = friendInfo[0:50]


###############################################################################
# Step 4. Calculate social connection score
###############################################################################
import math
def calcSocConnectScore(friendData):
    scr = 0
    for fr1 in friendData:
        if 'connectScore' not in fr1.keys():
            fr1['connectScore'] = 0
            for fr2 in fr1['connectMatrix'].keys():
                fr1['connectScore'] += fr1['connectMatrix'][fr2]
            if fr1['connectScore'] > 0: fr1['connectScore'] = log(fr1['connectScore'])
        scr += fr1['connectScore']
    return scr

import urllib
import ast #so we can convert the string representation of mutual friends into an actual dictionary
for fr1 in friendGroup:

    #TODO: Can't figure out how to get an OAuth access token
    #t = "https://graph.facebook.com/oauth/access_token?client_id={YOUR_APP}_ID& \
    #    redirect_uri=https://www.facebook.com/connect/login_success.html \
    #    &client_secret={YOUR_APP_SECRET}" \
    #    .format(YOUR_APP=my_api_key, YOUR_APP_SECRET=my_secret_key)
    #res = urllib.urlopen(t)
    #access_token = res.read()

    s = "https://graph.facebook.com/me/mutualfriends/{uid}? \

    access_token=AAACEdEose0cBACVyG7imQeRVhTmYx1OtZAZAgGwgHPzZCktWmYao7gOj60IT7bsaZBWrqEJJ9VDZAwYwFQFL6gzZADTxS
    gmEIZD" \
        .format(uid=fr1['uid'])
    res = urllib.urlopen(s)
    mutualFriends = ast.literal_eval(res.read())
    fr1['connectMatrix'] = {}
    for fr2 in friendGroup:
        fr2_uid_str = str(fr2['uid'])
        #Don't count connection to self
        if fr1['uid'] == fr2['uid']:
            fr1['connectMatrix'][fr2_uid_str] = 0
        elif fr2['uid'] not in [mutFriend['id'] for mutFriend in mutualFriends['data']]:
            #Not connected, so there is a basic connection potential
            fr1['connectMatrix'][fr2_uid_str] = 1
            #Check if common hometown state
            if fr1['hometown_location']['state'] == fr2['hometown_location']['state']:
                fr1['connectMatrix'][fr2_uid_str] +=1
            #Check if common past/current employer
            if [emp1['employer']['id'] for emp1 in fr1['work']] in [emp2['employer']['id'] for emp2 in
    fr2['work']]:
                fr1['connectMatrix'][fr2_uid_str] +=1
            #Check if common educational institution
            if [ed1['school']['id'] for ed1 in fr1['education']] in [ed2['school']['id'] for ed2 in
    fr2['education']]:
                fr1['connectMatrix'][fr2_uid_str] +=1
        else:
            fr1['connectMatrix'][fr2_uid_str] = 0
```

```
import pickle
#Save friendGroup variable for later
output = open('friendGroup.pkl', 'wb')
# Pickle dictionary using protocol 0.
pickle.dump(friendGroup, output)
output.close()



pkl_file = open('friendGroup.pkl', 'rb')
friendGroup = pickle.load(pkl_file)
pkl_file.close()

from math import log
def calcSocConnectScore(friendData):
    scr = 0
    for fr1 in friendData:
        if 'connectScore' not in fr1.keys():
            fr1['connectScore'] = 0
            for fr2 in fr1['connectMatrix'].keys():
                fr1['connectScore'] += fr1['connectMatrix'][fr2]
            if fr1['connectScore'] > 0: fr1['connectScore'] = log(fr1['connectScore'])
        scr += fr1['connectScore']
    return scr

score = calcSocConnectScore(friendGroup)
```