Vladimir Kozyrev

vkozyrev@ucsc.edu

3-16-2011

CMPS 161 Final Project Write-up

Facebook Visualization

For my final project, my goal was to visualize groups of friends on the Facebook social network. This presented a series of challenges, some of which included gathering the data, storing the data, algorithmically making sense of the data, displaying the results and learning new technologies to make all this possible in a web based environment. The following paper will discuss all these topics section by section.

Section 1: General project hypothesis:

Social networks, such as Facebook, can very naturally be represented as a graph. For my project I decided to make each person a node, while edges represent friendships. The value of the edge signifies friends in common. I hypothesize that close friends share a lot of friends in common, and if we delete the low values edges, we will end up having cliques of close friends.

Section 2: Gathering the data:

The first step to visualizing data is to gather it. Facebook did not lend itself to do this very easily. A social network's most natural representation is a graph. Each node represents a person, and each edge represents a friendship. For my project, I decided to weigh the edges based on friends shared in common. For storing the data

I used a sqlite database. In the database a person consisted of his facebook id, name, picture url, friends (other people), and and “updated” field which also consisted of people. The updated field merely signified that the relationship between that person was up to date. This way if there is one new person in the graph, we don’t have to run the update on every person. This is very important, because Facebook does not make it easy to gather date on other people’s friendships very simple.

Facebook has a method for retrieving an Facebook app user’s entire friends list. This makes it easy for me to get a JSON object representation of all of my friends, and their Facebook user ids. The main problem is that this method is not supported for other people. MY original idea was to retriever my friend list, and all of my friends’ friend lists. I would then be able to run a simple cross reference on the computer to rebuild their friends list. Essentially the goal was to see which of my friends were friends with my other friends. Since Facebook didn’t allow me to retrieve other people’s lists, I had to find a workaround.

The solution I stumbled upon was an old Facebook API method called areFriends. This method takes two user ids, and returns true if they are friends. So I wrote an algorithm to cross-reference all of my friends against all my other friends one at a time. The problem is that this is far slower than my previous method, especially since each call is a HTTP request. To calculate n number of friends, the algorithm would run $\prod_{i=0}^{n} n - i$ times. I had 8 threads pounding the Facebook server for about 3 hours, making about 300,000 HTTP request, in order to complete my dataset.

The reason that the number we multiply by decreases as we go down the set is because the friendships are symmetrical. Meaning we for person 3, we don't have to check against person 1 and 2. But we do still have to check person 3-n. This helps greatly with the overall runtime, although it still takes a few hours to complete all the HTTP requests.

Section 3: Storing the data:

As stated before I used sqlite database to store the data, but I did not actually have to write sql code. I used the Django web framework to both interact with the Facebook API and to handle my database. Django makes it very easy to define database models as classes, and it automatically generates the sql necessary to create the tables. On top of that, to retrieve items I don't have to write any code either. Overall it is a very fast and efficient system to create rapid-prototype projects. Python also allowed me to easily thread my program in order to speed up the data collection process.

The backend used a total of 9 threads and two synchronized queues to complete the data-gathering task. Python includes a synchronized, meaning only one thread can access it at a time, queue class which makes having concurrent threads access the same queue very simple. This is necessary to avoid data corruption and race conditions. The first thing that happens was the server creates a large numbe rof "to-check" objects. These simply consist of two people, and they mean that these two people need to have their friendship checked. It adds all of these to the "todo" queue. After this 8 http-worker threads and one database thread is created. The http-workers take an item out of the “todo” queue and check if those two people are friends. It then takes the result and creates a “result” object, which consists of two people and a Boolean value representing whether they are friends or not. It then adds that object to the “database” queue. The database worker takes items from the “database” queue and updates the database as fast as it can. We only want one thread writing to the database at a time to avoid data corruption, and that is why the http-workers don’t write to the database themselves, and also why we only have one database thread.

Section 4: Technologies used

As stated before I used python and Django for the back end code. Python is a very powerful language that also is very convenient for rapid prototyping. Django allowed me to quickly define database models and skip writing sql code. Django also allows me to send data to the client without having to reinvent the wheel. Python is definitely slower than any compiled language, but since we are dealing with HTTP requests, the time for the data to travel back and forth will always take more time than the python code executing (youtube runs a python backend). For the client side, I stuck with HTTP with Javascript for the computation. I also used the jQuery library to make the site appear dynamic. If the dataset was 2-3 times larger than it currently is, Javascript might have been too slow, but as it stands it works fine.

Section 5: Data analysis

Once the client starts the program he is given the choice to log in with Facebook. Once he does he can update his friends network on our servers (disabled because I have my data-set, and don't want other people to destroy my server by having it make millions of http requests). Or they can use the sample data, which is my friends network. This is the only option that is activated right now.

Once the client receives the friend network information from the server in the form of a JSON object, I proceed to create a graph from it. The person class contains all the basic information about the person, an array of friend objects, and a Boolean value used for BFS. The friend objects represent edges between two people, and have a value based on friends shared in common.

There is also a graph class. This contains the people in the graph (nodes and edges), and an array of graphs (this are BFS results with different tolerances). To construct the graph we run through each person, and see how many friends in common they have with every other person. This is also a pretty slow operation, although with a dataset of about 800 people, it doesn't take too long for the computer to calculate this.

After this I run BFS on the graph 100 times, each time increasing the tolerance for the edges. As the tolerance increases, lower values edges are skipped. Pretty soon the graph starts having multiple cliques, so BFS is used here to find groups of connected people. These groups are added to an array called groups, and that array is added to the graphs array. This results in a 3D array as follows, graphs[tolerance][group][person]. Once this was complete, it was time to display the data.

Section 6: Displaying the data

My original plan was to display the data using webGL in the browser. As I looked more into the data I had, I realized this would probably not be the best solution. It would have resulting in a more cluttered and glitzy presentation that didn't add much to the actual analysis of the data. Instead I went for a 3-tired approach of a slider, group display area, and people in the group display area. Since we have pre-calculated all our data, switching between the different graphs takes no computation time, this makes the slider seem very smooth and fluid (it actually displays the groups as you are sliding it).

First, to create the slider I used the jQuery slider library. This allows me to easily create a HTML slider element that hooks directly into my Javascript with callback functions. As the person moves the slider, which is valued 0-99, the relevant groups from the graphs array are displayed (which is also numbered 0-99).

The groups in the current graphs array index are instantly displayed in the groups section. This is done using a callback function. Whenever the person changes the value of the slider, I use a jQuery function to clear the contents of the groupBox element, and replace it with new groups. Inside the group identifier boxes is a label and a number representing the people in that group. These are all clickable elements, and when a user clicks on one, he is shows a list of people in that group in the friendBox area. These include the profile pictures and links to their Facebook profile.

I think this is a much better UI for a number of reasons. Firstly, it is far more browser compatible as opposed to using webGL. Secondly, it is more efficient and faster. Lastly, it is more informative. A person instantly sees the size of the groups, and once the click on a group the instantly see every one in that group. Moreover they also see their profile picture, which has a very good effect of the readability of the data.

Section 7: Results and analysis of the data

Originally I was expecting 5-6 large groups of people, but the algorithm acted in a way I did not fully prepare for. Instead it usually found small groups of very close friends, as in people who all know the same people.

One major snag, and the reason for the slider, was the fact that if one has a group of 15 people who all know each other, once the slider goes beyond 15 (the highest value of their edges), they will disappear from the visualization. This means that the user has to slide that slider back and forth to see all the interesting results. I could not find an algorithm that would pick the most relevant groups out of every BFS result.

But the end result was interesting nonetheless. While to an outside observer looking at the results of my friends might not make much since, to a person who owns the list, it is very interesting. These are some notable results that I found.

On the far left (most edges are kept) end of the slider, I instantly saw a group of people from my summer internship. It would make sense that they wouldn't know any one from Santa Cruz or my other home in San Jose. The other group was basically every other friend I had.

Around the middle it also got very interesting. The UCSC swim team diverged from the rest of the school very quickly. And as one dragged the slider, one could see how it transitioned from swimmers and non-swimmers who hung out with us, to the closer-knit swim team group.

Once the slider was pushed even further a group emerged of the 4 members of the Santa Cruz TME entertainment company. Once the slider was pushed further, one of them dropped out and left the 3 founders in the group.

But the most notable result for me was when I pushed it as far right as I could and there was only one group of 3 people. When I clicked on that group, there were 3 swimmers, all of whom lived in the same dorm-room together freshman year.