

DataW: Finding Connections

Data Choice

The first stop is finding what data to collect. Beyond standard personal information (name and id), the key data to collect is work history and social links. The obvious targets for data collection would be professional or general social networks: LinkedIn, Facebook, and Google+.

LinkedIn

The ideal place to get all of this information would be [LinkedIn](#). LinkedIn is focused around professional connections and as such is ideal since it provides work history and connections between people. Unfortunately, even though a non-logged in visitor to the site can view substantial information, LinkedIn is aggressively against scraping their site. They have even gone so far as to [attempt to sue researchers trying to crawl information](#). LinkedIn does provide an API, but they do not make available the data that would be important to this project.

Facebook

Facebook is the next target to consider. A logged in user can see both the work history and friendship links of a friend. Even a non-logged in visitor can view the work history of users that have more public privacy settings (about 55% of pages crawled). Unfortunately, no friendship information is given to anonymous users. Therefore, Facebook can give us the work history information from a user, but not clear connection information. Facebook also provides an expressive [API](#), but it is very limited with respect to privacy settings. The target user must have installed the API's app and must be a direct friend of the requester.

Google+

Google+ is a general social network like Facebook. It has a fairly expressive API, but has stricter privacy settings than Facebook. In addition, Google+ has less active users than Facebook.

Because of the large number of users, available public information, and low risk of litigation; I chose to work with Facebook.

Page Crawling

Because Facebook does not allow an anonymous user to see friendship links, a large obstruction to crawling users is finding the links to crawl. In an ideal situation, we could start crawling at a few seed users and gather all their friends. Then on the next iteration, we could get all those new users' friends and so on. This would allow us to exponentially grow our graph. All the functional described in this section is provided by the [scrapPeople.py](#) file.

Seed Users

Since the ideal crawling strategy is not possible, I opted to perform the a portion of the ideal first two steps manually. I logged into my Facebook account and saved the html for my "Friends" page. I then choose 85 friends and saved the html for each one's "Friends" page. I used these pages as my seed pages. When the crawler starts up, it uses the seeds to start growing the pages. In addition to just seed users, I also chose an active Facebook group for a computer science department. I collected about 8 months of posts from that group to also use as a seed.

Page Cache

Since there are hundred of thousand of pages to fetch and we do not want to be more of a burden to Facebook then we already are, we cache every page fetch (http GET). Therefore when we restart the crawler, we only have to read from the local cache. We just refer to this as the [cache](#). The final size of the cache is 8.6 GB.

Parsing Links

Once we have a page (either fetched directly from Facebook, or from the seeds), we parse the page for any links that look like a Facebook link (takes the form of "facebook.com/"). We then deduplicate the links against all links we have seen and add new links to the next round of exploration. Because there is often a need to restart the crawler, we also keep a cache of all the links that we have parsed from a specific page. This is referred to as the [link cache](#). The final size of the link cache is 156 MB.

Parsing Pages

After all the pages are fetched, we then parse them for information. All the parsing of the cache is handled by the file [parseCache.py](#). The key information that we parse out of the page is:

- Facebook Id
- Whether or not this is a "page". "Pages" tend to be more for events, places, or celebrities.
- Profile Image
- Work History
- Education History
- Residential History

After we parse all the above information from the html of a page, then we convert it to a json array of dictionaries for easy parsing later.

Database

After parsing the html files into JSON, I then use the JSON to construct SQL insert statements to load the data into a PostgreSQL database. All the code for constructing the queries can be

found in [parseJson.py](#). All the static SQL for this section can be found in the [sql directory](#). A single script that recreates the entire database has been [uploaded](#).

Design

The schema for the database can be found in the [create.sql](#) file. Some particular things to note about the design:

1. Surrogate keys are used in every table for efficiency. Also, it is never a good idea to trust the client/datasource to provide sufficiently unique keys.
2. The *Education*, *Employment*, and *Lived* tables all have two sets of keys: one using the facebook ids and one using the surrogate keys. This is so that we can make all of our inserts at one time without having to find out what key each entity was assigned. These columns will later be optimized out.
3. The *Entities* table contains several columns that have redundant data. All the "is*" (except *isPage*) columns are populated by checking to see if the entity appears as a target in the associated table (eg. *isSchool* is populated by seeing if the entity appears in the *schoolId* column of *Education*). This breaks the spirit of 2nd Normal Form (although not the letter), but makes some later queries much faster.

Optimization

After the database is populated, I perform several optimization steps to make later queries more efficient. The optimizations can be seen in the [optimize.sql](#) file.

The general optimization steps taken are:

1. Populate the Entity surrogate keys using the facebook ids.
2. Drop facebook ids in all tables except *Entities*.
3. Add indexes.
4. Populate the "is*" fields in the *Entities* table.

Fetching Images

Although not required, having a user's Facebook image adds a lot to the visualization. I first fetch all the images using a similar caching scheme used while fetching pages. The [image cache](#) reached a final size of 173 MB and 22834 images. To simplify the web serving of the final application and centralize the required data, I chose to also store the images in the database. Storing images in the database is not a decision that should be taken lightly, as they are typically stored on disk. However, in this case I believe the benefits outweigh the performance costs. To make the image representation in the database useful, I encoded the image data as a base64 string. This makes the images readily accessible to anyone using the database, and allows for easily displaying the images in the web app using [data urls](#) for img tags. Image encoding is handled by the [encodeImages.py](#) file.

Edges

Since we do not have direct friendship links from Facebook, we must create edges between nodes and give weights to these edges. Here, we make an important and necessary assumption: people who have attended the same institution/workplace/city (henceforth known as place) should have some connections. For each pair of entities, we count the number of times that they have occurred at the same place and assign that as the score for that edge. Zero score edges are not included. Therefore a pair of people who have never attended the same place will have no edge between them; however a pair of people who have attended the same school, worked at the same workplace, and lived in the same city will have a score of three. Add edge discovery is done in [edges.sql](#).

Cost

From the edge score, we can normalize to calculate an edge cost between 0 and 1. We use a similar linear normalization:

$$cost = \min(1, \max(0, 1 - (score / 5)))$$

The max score observed in this data is 3, however it is possible for a higher score if, for example, a pair of people attended the same school for undergraduate and graduate programs.

Inner Edges

We make a differentiation between two types of edges: *inner* and *outer*. Inner edges are only between people (defined by the *isPerson* column in the *Entity* table), while outer edges are between people and places. We need to make this distinction, because when searching for a path, only the last node in the path should be a place. All other connections should only be between people.

Graph

The entities (from the *Entities* table) and edges (from the *Edges* table) for a social graph which is the crux of this project. The final graph has 61,724 edges, 22,959 of which are people; and 4,291,242 edges, 4,141,584 of which are inner edges.

Representation

Keeping a dense representation of the graph would consume far too much memory, so we keep a sparse representation of the graph. For the graph itself, we only keep the minimum data to represent the edges, edge type (is inner), and weights. All other information about specific entities (facebook id, name, profile picture, etc) is maintained separately and only fetched when needed. Graph representation is handled by the [loadGraph.py](#) file.

Cache

To speed up initialization and make it easier for other people to run this project, the graph is cached in a pickle. When the pickle does not exist, the graph will be fetched from the database, parsed, and written to the [pickle directory](#). A [copy of the pickles](#) required for basic functionality are uploaded with the other data files.

Path Finding

Path finding is historically a difficult problem. Having fixed terminal nodes (start and end nodes) helps a lot, but there are several caveats that makes this problem difficult:

1. Inner edges must make up the entire path except for the last leg.
2. We do not have a useful admissible heuristic which is required for A*. In fact, we have no useful heuristics, the only cost function we can use is the actual cost between nodes.
3. We need to find multiple paths instead of just one.
4. We need results quickly. There is not enough time to do a full search for the best path.

To satisfy all the requirements, I created an A*-inspired algorithm that finds multiple paths that through the graph. The actual code can be found in the [path.py](#) file. The general idea is to use a similar tactic to A* where we add all nodes adjacent to the current node that we have not yet seen into a priority queue sorted by the cost to get to that node. We then take out the top node, and add all the nodes adjacent to that node. Some points of interest in this algorithm:

1. The nodes we have seen are kept along with the path (and cost) to get to that node.
When we see a node again, we first check to see if the new path to that node is cheaper than the listed cost. If the new path is cheaper, then we replace the old path with the new one. In this dynamic programming fashion, we can always keep track of the cheapest path to any node we have seen.
2. After we have found the required number of paths, we will still continue for a few iterations to find slightly longer paths.
3. On a machine using an Intel i7-4790K (4.0 GHz), a search for 10 nodes typically takes less than a second.

Visualization

The final visualization takes the form of an interactive web application. The backend to this web app is all handled by the [server.py](#) file. The backend provides an API to get the entities in the graph and search for a path between two entities. It also serves the static resources (html, js, and css) for the web app.

Cytoscape

For visualizing paths (graphs), I chose to use [Cytoscape.js](#). Cytoscape is meant especially for graphs, so it is a bit more streamlined with graphs than a more general visualization framework like d3. I had no specific troubles with Cytoscape. The only hurdle I had to jump over was laying out my nodes. Cytoscape has several ways that it can layout the nodes in a graph. However, none of them suited my use case. So, I instead wrote my own algorithm to position the nodes in

my graph. Once the values were calculated, it was no problem to have cytoscape place them in my desired locations.

Getting the Data

All data has been uploaded into a [single directory](#) on Google Drive.

- [cache.tar.gz](#) - 5 GB - A copy of every http get made crawling Facebook profiles.
- [imageCache.tar.gz](#) - 132 MB - A copy of every image fetched from Facebook.
- [linkCache.tar.gz](#) - 36 MB - A cache of the links parsed from each web page.
- [seeds.tar.gz](#) - 26 MB - The seeds used to get the initial set of links to fetch.
- [pickles.tar.gz](#) - 153 MB - Pickles that can be used to replace the database for basic functionality.
- [facebook.sql.gz](#) - 159 MB - A single script that can recreate the entire database.