Accurately Sampled Graph Visualizations of IMDb Data

Kevin Woodward
University of California, Santa Cruz
CMPS 161 – Data Visualization – Alex Pang
keawoodw@ucsc.edu

Abstract

Many approaches have been taken to visualizing the ever growing IMDb (films and related data) database through static methods. Few dynamic approaches exist, and no graph based models exist. Inspired by wikiverse.io, we suggest a dynamic and user-friendly tool for representing subsets of the IMDb actor and film data sets. Using this tool, manual sampling and experimentation can allow analysis of the graph output data to determine what does and does not work in considering a graph oriented subset of the data, as well as gaining other insights.

1 Introduction

IMDb.com is a popular website with near complete data on films, TV, actors, and related information. The IMDb database is composed of over fifty files, categorized by format and type, each millions of lines long. While the IMDb website allows quick access to film related data, it lacks in the ability to snapshot related data with varying parameters. The aim of this representation is to focus on the scope of actors and films, and how to optimize the collection of elements in this representation to best suit the user's needs.

With a non-specific end goal, considerations must be taken to accommodate the different demands of users. To demonstrate the success of this, several cases of usage will be dissected to provide evidence of competent design. There have been successful projects of a similar nature such as wikiverse.io[1], a fully featured alternative method to navigate Wikipedia. While this project derives inspiration from wikiverse.io, the scope of the program is simply to be a tool used for exploring relations rather than a replacement to the source.

1.1 Subproblems

The first issue encountered was how to approach the deconstruction of the IMDb database to allow for reasonable access time for queries. The source of the data was from the IMDb alternative interfaces page[2], which is essentially a hosting of the aggregate of the IMDb data in plaintext files. The initial approach was to parse the actors.list, actresses.list, and the movies.list archives through a custom parsing program. The intention is that the output would be converted to MySQL tables for local and fast access. For multiple reasons explained later, this is not the final implementation. Instead an API which returns info from a passed unique IMDb data tag was used.

The JavaScript framework for which to represent the graph was a relatively simple choice. There were only two realistic choices: Sigma.js[3] and Cytoscape.js[4]. The decision for Cytoscape.js was based on the fact that it is more fully featured, documented, and flexible, all benefitting the development as well as the final product.

The algorithm design was straightforward in broad design, and didn't see too many roadblocks in development.

Performing statistical analysis on the output of the graph tool that was built proved to be difficult. The source of difficulty came from the fact that the intention of the analysis was open ended and therefore required a lot of trial

[1] http://wikiverse.io/

[2] http://www.imdb.com/interfaces

[3] http://sigmajs.org/

[4] http://js.cytoscape.org/

and error in determining significance. Initially an automation of the graph tool was going to be written, allowing for background collection of data on certain parameters. The diversity of the exploration of options didn't allow this to be a realistic goal, so most graph data collection was done manually.

1.2 Related Work

The work in this paper is related to several fields, including but not limited to graph generation, graph drawing, and big data. Following are some references to related papers as well as projects.

The first resource is a paper on the topic of sampling from large graphs, including acceptable sampling methods, sampling sizes, and algorithms. It is co-written by two authors from Carnegie Mellon University. [5]

Another relevant paper discusses fast graph generation at multiple levels of implementation, from hardware to algorithms. It is co-written by four authors from the National University of Singapore.[6]

A project of a similar nature is wikiverse.io, a graph representation of a subset of Wikipedia. This project is a great example of using graph style visualization as an alternative means to navigate and interpret a data set.[1]

A project dealing with the same data set as this is the Oracle of Bacon or also known as Six Degrees of Bacon. This is a simple project which finds the shortest path from one actor to another (traditionally Kevin Bacon as the source). This project relates in the fact that it must have tackled a similar graph based issue using the same data. However, it deals more with graph searches than generation.[7]

2.1 Technical Detail - Solving The Problem Of Data

As mentioned in the introduction, the original goal was to parse the relevant plaintext files into MySQL tables to include locally. The parsing was successfully completed to a degree, but several issues existed:

1) The files were simply too large to include on the client side. Due to this program being completely client side (no Node.js[8]) it felt unnecessarily extraneous to add this solely for data collection and manipulation.
2) There was a significant error rate, approximately ~8% during queries, in the formatting of actors and films across the relevant files. Being plain text files, these massive lists have their own established format which is simple in theory. However, the list is contributed to by users and only validated by one or two super-users (depending on the particular file) before being admitted as an acceptable change. While this may be a contributor to the error rate in queries, it is also likely that the actual parsing of this data was not accurate as well.
3) There is no standardized and integrated method for performing queries on local files with pure JavaScript. The assumption prior to development was that there must be a library to perform this, but most of these are aimed toward Node.js.

After the conclusion that this was not a viable method to accomplish data access, alternative methods were searched for. IMDb has two APIs, most likely for internal usage as they are not quite unknown and are completely undocumented[9][10], and for this reason alone they

[5]Leskovec, Jure, and Christos Faloutsos. "Sampling from large graphs." Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2006. http://www.stat.cmu.edu/~fienberg/Stat36-835/Leskovec-sampling-kdd06.pdf

[6] Nobari, Sadegh, et al. "Fast random graph generation." Proceedings of the 14th international conference on extending database technology. ACM, 2011.http://www.openproceedings.org/2011/conf/edbt/NobariLKB11.pdf

[7] http://oracleofbacon.org/

[8] https://nodejs.org/en/

[9] http://sg.media-imdb.com/suggests/a/aa.json

were not chosen. There are a number of third party IMDb APIs available for free use, but only one meeting the essentials required. RESTful JSON API for IMDb[11] is a simple yet functional API with two request types, both HTTP GETs, both returning padded JSON. The first type of request requires sending a simple IMDb data tag, which return a response with information about the associated data. To understand why this is useful, we must understand the structure of the tags and the returned data. An IMDb data tag is a unique identifier for each actor, film, and even site users and other tangential data. Here we are only focusing on actors and film/TV. A tag is easily accessible, as it is the last parameter in the URL of any IMDb actor or film page. For example, the URL for Quentin Tarantino is http://www.imdb.com/name/nm0000233/, and the tag is nm0000233. A film tag is identical in structure, with a difference that the "nm" in the tag is "tt" for easy differentiability. The data that the API returns when given an actor tag is a large object, with a couple very useful items. The first is the actual name of the actor, allowing the graph to be more readable. The second is the entire filmography of said actor organized by recency, complete with tags for each film. This sets up a very nice feedback loop which will be explained in following sections. When passed a film tag, the API seems to not be as fully featured. Each film return null for a title, but it does provide a cast. The caveat is that the cast does not contain tags, which brings us to the second form of request type.

The second form of request is invoked by adding a search string in the data parameter. A response gives multiple search result types, such as character names, film names, and actor names. The useful data is in the actor names, which allows an actor in the response from a film tag request to be reverse searched for their tag and filmography. Without this feature, it would be very difficult to recurse on any data.

It is useful to note that jQuery was used for simplifying HTTP requests.[12]

With the API explained, the structure of data collection should be clear and an algorithm structure should be apparent.

2.2 Technical Detail - Graph Generation Algorithm Design

With an understanding of the API, the algorithm for graph generation is fairly simple. The user inputs as many root actors as desired. For each root, films are chosen from the filmography to add as edges, and each edge will choose another actor to add as a destination node. This repeats until a max size condition is fulfilled. The resulting graph is undirected, as a connecting film will always allow the reverse of the source and destination to be possible.

There is a lot of ambiguity in the given description that is remedied by listing the options given on the graph generation page. Films are chosen from actors either randomly, or by age, either new or old. Actors are chosen from films either randomly, or by cast billing, either high or low. The number of films searched per actor is varying depending on what the user enters, and consequently there is one node resulting from an edge. The size condition is also varying depending on the user, and it is an amalgamation of total node count and search depth.

In actual code structure, the algorithm is designed such that each root node spawns a recursive chain of searches and element additions until the size limitation is reached. The main reason taken into account when deciding on this design was that the first prototype of the program did not use the jQuery ajax format to do HTTP requests. This resulted in an incredibly slow generation time due to the linear nature of execution. Using the asynchronous format allows each next request or addition to be a callback of the previous, allowing multiple element additions at a time and dramatically reducing generation time. Each HTTP request follows up with another HTTP request, until the

[10] http://sg.media-imdb.com/suggests/h/hello.json

[11] http://imdb.wemakesites.net/

[12] http://jquery.com/

required amount of data is collected to add a new node and edge. For most cases, this requires four requests. One for the source node, one for the chosen film, one for reverse searching the new chosen actor from the film, and one for gathering data about the new actor. With the exception of the root nodes, the first and last calls essentially serve a dual purpose during graph expansion (the new node data serves as the next source).

A separate yet important feature in this algorithm is the cross checking of each node. Upon a node addition, a handler function will iterate over every other node in the graph. For each node, the filmographies of the two nodes will be compared for intersections, and any results will be added as edges between said nodes. The intersection rate greatly depends on the options selected. If recent movies and high billing actors are selected, then there is a greater chance of shared films over true random. Having this cross checking fills out the graph, connecting components and providing more information on relations between actors.

2.3 Technical Detail - Other Features

A few other functionalities exist in the graph generation tool that do not directly affect the data outcome.

The layout is selectable between three options: concentric by degree outward, circular, and breadthfirst, each providing use. The concentric style is a circular format which puts nodes of high degree in the center, with lower degree nodes expanding outward. This is useful in determining which nodes have the most connections. The circular style is an unbiased circle of nodes, resulting in all edges going through the circle. It is the clearest way to view the nodes. The breadthfirst style organizes the graph in a breadth first search traversal, starting at nodes which have no incoming edge, i.e. the root nodes. This is useful in understanding the result of the generation process.

Actions occur on the click of a node or edge. The action for an edge is static; it always opens the related IMDb page. A node click can do several things, depending on the option selected in the dropdown menu. The default is also to open the related IMDb page. Drag allows manual movement of the nodes. Breadth first search and depth first search will color the component of the graph that the selected node is a part of by performing either a BFS or DFS with the selected node as the root. The Karger Stein min cut will color the component nodes into two categories and the edges determining the minimum cut. After addition, it was realized that the result is almost always one edge, as the structure of the generation results in many leaves. Even though it seemed rather useless, it was kept in since there was no need to remove it.

An option to turn off graph animation is available, but it prevents the actual layout from updating upon new node and edge additions. This is due to the fact that Cytoscape.js is not designed as a dynamic library and has not such feature. However, the animation is computationally expensive and visual output performance will be increased in large graphs with animation off.

The recenter button simple snaps the existing graph to fit nicely in the viewport to prevent manual zooming hassles.

2.4 Technical Detail - Performance

On the first prototype, graph generation took about a minute per ten nodes on average. The bottleneck of this pass was due to waiting on a single HTTP request to return for the next to fire. With the current model, depending on the options, up to sixty nodes per minute can be attained, and the bottleneck is still the response time of the API. In terms of actually rendering the graph, performance doesn't become an issue with animation on until about 500 nodes, and with animation off it has been tested to 1000 with no issue.

The speed of graph generation varies greatly depending on certain options. If a graph is set to generate from one source with only one edge expansion per node, we essentially have the original prototype case which also happens to be the worst case. To maximize node addition speed, the goal is to have as many parallel chains of request as possible. This means a large number of root nodes and many edges per node, regardless of the total graph size.

It is also to important to note that the number of pending requests on the API server greatly affects response time. The general peak for test cases was anywhere from 200 to 300 pending requests. This is easily observable by setting the number of edges expanding off of each node to be high (>10). The first degree of depth will return fairly quickly. There will be a significant stall in node additions, usually a minute or two. Them when the server recovers from the shock it has been given and starts sending responses, a steady flow of nodes will be added rather quickly.

3.1 Graph Generation Results and Meaning

With the generation tool built and optimized as much as the structure would allow, many graphs were generated to accumulate an understanding of how the relations varied on the options. Unfortunately, development and debugging of the program took much longer than expected which resulted in a suboptimal method and number of samples collected. About twenty graphs were generated per set of root nodes, with the set of nodes being of size five and ten. These sets were chosen from actors who would be considered common knowledge, and as a result are fairly popular and recent. The scale factor was set to a constant 50, and each permutation of how films were selected from actors and how actors were selected from films was used. This resulted in about two to three graphs per permutation, per set of root nodes. While this is not enough data to be considered statistically significant, it did allow general trends to be seen. To no surprise, the random option for each category resulted in much more varied actors (based on pay, lifetime, and popularity) than any of the other permutations. As a results, there were significantly less edges added due to a successful cross check. Ideally, more time would have been spent here even though it is not the main goal of the project.

3.2 Program Results

Overall the final version of the program was a success considering the limitations and roadblocks experienced. The evolution of the design went from essentially a slow simple single path search to a less slow dynamic graph builder with interactive features. The algorithm is flexible enough to the point where it is comfortable to say it is well featured. Of course even more flexibility in options is always welcomed. The method of data collection could have been better, as it is sole bottleneck for the current performance restriction. This is true only because the edge and node styling were set to be very simple. Certain settings such as Bezier curve edges, complex node shapes, and compound nodes are known to cause rendering slowdown that scale very quickly with graph element count. There are some known issues as well:

When generating a second graph on the same instance of the page, pending requests tend to persist even after destroying the initial graph. This results in false nodes being added to the second graph as the pending requests return with some data and are handled. While inconvenient, this can be resolved by simply refreshing the page.

When the animate option is unchecked, the graph layout does not update even when a layout is forced. It is not determined if this is a Cytoscape.js bug or a misuse of the library. A simple fix for resetting the layout on non-animated graphs is to check and uncheck the animate field, allowing the graph to be repositioned and then preventing animation. Note that this is best done at the end of generation, as new nodes added afterward will not adhere to the layout if the graph is set to not animate.

4 Conclusion and Future Work

This paper presents an explanation on the representation of relational data in a graph format, the process to develop a program for use, and the results of said program output and performance. This program is featured to allow a user to explore the relationships of multiple actors and films, as well as provide exploratory information about related data. The open endedness of the intention of the program allows it to be used as simply a toy, as well as a tool for potentially powerful graph analytics.

There are currently plans to expand and refine this project in the future. Ideas include reworking the IMDb data model (perhaps expanding to custom data sets), porting intensive code to Node.js, adding more algorithm flexibility, and generally cleaning up the interface. While these could have been accomplished during the development scope of this project, it was found more important to have a simpler yet solid foundation to show the robustness and applicability of this idea.

While there are many possibilities for expansion and improvement, the introduction of this program solidifies a foundation for approaching the IMDb data from a new perspective and adding to the existing IMDb data model in the future.

5 References

[1] Wikipedia Graph Theory Representation. Related work and inspiration. http://wikiverse.io/
[2] IMDb alternative interfaces/data formats. http://www.imdb.com/interfaces
[3] Considered but unused JavaScript graph library. http://sigmajs.org/
[4] Chosen and implemented JavaScript graph library. http://js.cytoscape.org/
[5] Leskovec, Jure, and Christos Faloutsos. "Sampling from large graphs." Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2006.
http://www.stat.cmu.edu/~fienberg/Stat36-835/Leskovec-sampling-kdd06.pdf
[6] Nobari, Sadegh, et al. "Fast random graph generation." Proceedings of the 14th international conference on extending database technology. ACM, 2011.http://www.openproceedings.org/2011/conf/edbt/NobariLKB11.pdf
[7] Degree of separation finder for actors. Related work and inspiration. http://oracleofbacon.org/
[8] Server-side JavaScript platform, unused. https://nodejs.org/en/
[9] An official but undocumented IMDb API. Unused.
http://sg.media-imdb.com/suggests/a/aa.json
[10] An official but undocumented IMDb API. Unused.
http://sg.media-imdb.com/suggests/h/hello.json
[11] Third party IMDb API, used for all IMDb information. http://imdb.wemakesites.net/
[12] Common JavaScript framework, used in this case for HTTP requests and simple DOM manipulation. http://jquery.com/

Note: These are also found as footnotes.