Six Degrees of Separation Finder

COREY SCHANINGER
Donald Bren School of Information and Computer Science *cschanin@uci.edu*
University Of California Irvine
Irvine, CA 92697-3425

LAKSHMI THYAGARAJAN *reachlakshmit@gmail.com*
Donald Bren School of Information and Computer Science
University Of California Irvine
Irvine, CA 92697-3425

1. Introduction

1.1 What does Six Degrees of Separation mean?

The Six Degrees of Separation describes the theory that every human being on the planet can be connected to another in a maximum of six links (social connections). This idea has a long history in many forms, the most well known of which is called the "Small World Experiment." The Small World Experiment, developed by psychologist Stanley Milgram and based off Michael Guerevich's earlier studies, studied the structure and mathematics of social networks.[1] Since then, this concept has been explored and expanded into many different research fields including:

- email
- world news
- computer networks
- communications
- charitable social networks and more!

Other popular spin-offs of this theory are Kevin Bacon numbers and Erdos numbers (famous mathematician- link if you publish a mathematical paper with him). Our project attempts to find "Kevin Bacon" numbers, or more generally, connections between actors in movies and television.

1.2 What does the Six Degrees Finder do?

Given any two actors' names, the Six Degrees Finder searches for a path of six degrees (actor – actor connections) or less between them. For example, the path from Katharine Hepburn to Quentin Tarantino might be:

> Katharine Hepburn --------> American Creed (1946) --------> Ingrid Bergman (I) --------> A Matter of Time (1976) --------> Isabella Rossellini --------> AFI's 10 Top 10: America's 10 Greatest Films in 10 Classic Genres (2008) --------> Quentin Tarantino

This application searches through data from the Internet Movie Database (IMDB) containing over 1.5 million actors and over 1.6 million movies and television shows. We discuss the exact nature of the IMDB data in the next section.

2. Methods

2.1 Parsing IMDB data

Our first step after getting the IMDB data was to parse the files in actor- movie form into an integer representation to save space. The files contained a list of all movies each actor was in, the date, episode number (if a television show), character name, and some extra information about the release type. We created lookup tables for actor – actor id's and movie – movie id's and used these ids to represent the actor – movies list. In this process, it became necessary to do a bit of extra processing on the movie titles and actors names for searching.

2.1.1 Formatting Actor Names

Actors' names were in the form "last name [suffix], first name [middle names] [(roman numeral)]" so we decided to reverse the names to match the form in which users would most likely be inputting their queries, starting with first names. We were careful of suffixes and duplicate names- differentiated by trailing roman numerals during this process. Later, when processing user queries, we were able to manipulate this reversed format with much greater ease for stemming to make suggestions.

2.1.2 Formatting Movie Titles

Since the movie titles contained so much extra information, we needed to remove the character names altogether, since the map reduce would be using these individual titles as keys, each movie title with character name would be treated as a different movie, when in fact many were one movie with different character names. During this process, we also decided to remove self-appearances (any occurrence of the words "self" or "selves" in the character name would indicate this was a self-appearance) since for our purposes this information is less interesting. We originally dropped everything after the movie's year, including the episode titles, but realized soon after that many television shows run so long with many guest appearances that it was best to leave individual episodes in, and treat each as its own title.

2.2 Creating Movie-Actors Posting List

Next, we fed the newly created actor id - <movie ids> list into Amazon EC2's Hadoop Cluster to create a posting list mapping each actor appearing in a movie to that movie, in effect "reversing" the lists to be of the form movie id – <actor ids>. This task was performed using the Map Reduce framework to map each movie in an actor's list to one (movie, actor) pair, and then reduce all such pairs into one (movie, actors) list. This enabled us to create the tree structure (discussed later in this paper) to know all movies a given actor has ever acted in and all actors in a given movie.

2.2.2 Other Methods Attempted

Originally, we tried a few other approaches to this problem. We attempted to keep the original text for everything, but soon realized that using integer ids was a much more efficient use of space, since we could store the id lookup tables in memory and this reduced the size of the lists drastically. Our original intent was to create only one list of "actor-<actors>" which would represent a list of every other actor a given actor has ever worked with (been in a movie with). The idea was that we could skip the intermediary step of going from actor to movie to actor and just traverse from one actor to another, speeding up query time. Unfortunately, the size of this list was exponentially larger, due to the number of repetitions of an actor's name in every other actor's list that is in theirs.

We had originally created this structure with text, then moved to using ids, but tried to keep (actor, movie) pairs in the list so we had a structure which looked like "actorid-<(actorid, movieid)s>" where each actor's list contained every actor they'd ever worked with and which film they were in together for each of those actors. Since this was far too large, we next tried to reduce these pairs into sublists, where for each actor in a given actor's list there was a list of movies the pair had worked in together (actor, movies), as opposed to multiple (actor, movie) pairs. Thus, the structure looked like "actorid-<(actorid, movieids)s>", but this was not much better for memory. Finally, we decided to throw out the movieids altogether, and just keep actorid-<actorids> lists and look up the movies in the intersections of the final path at the end. However, even this file was well over 10GB and it proved to be faster to keep the two smaller actorid-<movieids> and movieid-<actorids> lists in memory than to treat the actorid-<actorids> list as a random access file and have to keep performing hundreds of disc reads for each query.

2.3 Parse Files into Main Memory

After deciding to keep the two lists in memory, we needed to parse them into HashMaps from the Hadoop output files and parse the id lookup tables into BidiMaps in memory, which allowed us to do bidirectional lookups for ids. The bidirectional lookup tables for actorids and movieids could then be used to check translate the user's queries into id form to perform the search over and then return the text results to the user.

2.4 Parse User Query Input

The use of bidirectional lookup tables assumed that the user's queries were entered in exactly the same form as in our database. To correct this issue, we first looked for the user's initial input in the actorid table and if we couldn't find it, then we did a search over all the actors names in the table, ignoring case and allowing stemming to accommodate for nick names and duplicate names which were differentiated by roman numerals. We then returned this list of possible suggestions to the user and allowed them to try again. While not ideal, this solution improved the original issue by quite a bit, but certainly leaves room for improvement by accommodating for spelling errors or ignoring case initially, etc. Once this last step was performed, we had the framework in place to perform multi-threaded depth first search on the

actor ids parsed from the user's query and then return the actor and movie names in the final path found in the search.

2.5 Actor-Movie Connections

So as explained in the earlier sections, we go through Movies to find the links between Actor1 and Actor2. So a link like Actor1 -> Movie 1 -> Actor 5 -> Movie 18 -> Actor2 means that Actor1 co-appeared in Movie 1 with Actor 5, Actor5 co-appeared with Actor 2 in Movie 18. In order to establish this link we need the following:

Actor -> Movie list (list of movies in which this actor acted)

Movie -> Actors list (list of actors who appeared in this movie)

Creating the series of links from Actor1 to Actor2 involves starting with Actor1 and searching through every movie that this actor has acted in. Then for every movie check if Actor2 is in that movie, otherwise look at the movies from every actor in this movie. Thus this is a recursive search that examines a maximum of six levels or 5 actors from Actor1 to Actor2 and terminates either when Actor2 is found in some movie along the way or 6 levels is exhausted and Actor2 is not found. This kind of search warrants a tree like data structure.

2.6 Tree Structure

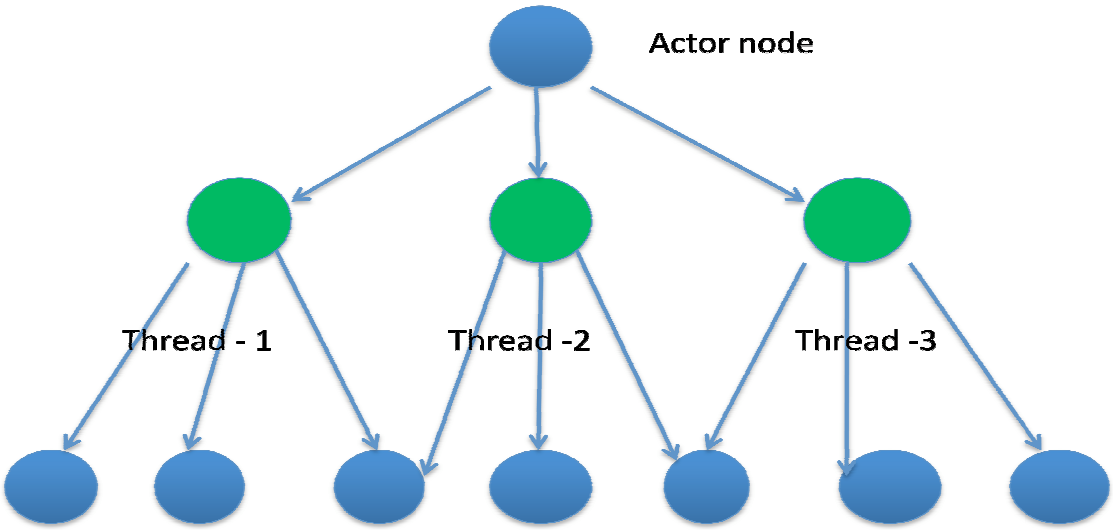
The tree that we use consists of two types of nodes: Actor Nodes and Movie Nodes. Actor Nodes contain Actor Id, Node Type = ACTOR and Parent = Pointer to its parent movie node. Movie Nodes contain Movie Id, Node Type = MOVIE and Parent = Pointer to its parent actor node. We built the tree on the fly since pre-building is not possible. Pre-building involves coming up with every possible pair of actors and storing the links for each pair. Considering that there are a total of almost 3 million nodes and that pre-building results in duplication it becomes impossible to store all the links in main memory and hence pre-building is not an option.

2.7 Tree Traversal

For traversing the tree we considered two main algorithms namely breadth first search and depth first search. While breadth first search offered the advantage of always computing the shortest path between two actors there were severe disadvantages in terms of memory. It became impossible to operate fully on main memory as it is necessary to keep track of all the children of the current level being explored. The next approach we considered was depth first search. DFS gave us considerable savings in memory as it requires only a maximum of 12 lists (6 for actors and 6 for movies) to be kept in memory at any point of time. The downsides however were that the search could be stuck locally in some portion of the tree where a connection did not exist, and, that a dfs search does not look for the shortest path. However the good news was that a dfs search lended itself well to multi-threading. By leveraging multi-threading we could make sure that the search did not get stuck locally in some unuseful

part of the tree because of which we obtained a considerable speed-up. Mulithreading also implied that neither could we guarantee to return the same path during multiple runs between the same pair of actors, not could we guarantee the time taken to compute the paths during multiple runs between the same pair of actors. However in-spite of these seeming disadvantages, we found that in most runs, multi-threaded dfs could fetch a valid link within a matter of 1 to 10s of seconds.

So the approach we decided to adopt was DFS but in a multi-threaded fashion to get time savings.



2.8 Multi-threaded DFS Implementation

To optimize the time taken for the search we adopted a two way search strategy. This is because we observed that searching from Actor2->Actor1 was sometimes faster than searching from Actor1 -> Actor. We spawn one thread each for every movie that Actor1 has acted in and one each for every movie Actor2 has acted in. Actor1 movie threads look for Actor2 while Actor2 movie threads look for Actor1. All threads share the 'foundActor' object. Once Actor2 or Actor1 (depending on which way a thread is searching) is found by one thread all other threads get to know this and they exit gracefully. The route from Actor2 node to Actor1 node found by following the Actor2 node's parent recursively till you reach Actor1. Every thread keeps a list of seen actors and movies are locally in order to avoid repeated searches. Seen actors and movies are not shared between threads as the level at which an actor/movie is discovered by a thread can vary.

3 Results

The time taken to compute a link depends on the input actors pair. I could vary anywhere from 1 to a few tens of seconds for links that involve atleast one popular actor and anywhere between 60 to 350 seconds for links between unknown actors.
Examples:

1) Between two popular actors:
   Nicolas Cage-------->Amos & Andrew (1993)-------->Jeff Blumenkrantz-------->Anastasia (1997)-------->Meg Ryan

   Time taken < 1 second

2) Between one popular actor (Emma Watson) and one lesser known actor
   Chow Wai Keung-------->"Yellowthread Street" (1990) {Power Play (#1.1)}-------->Bruce Payne (I)-------->Britannic (2000)-------->Martin Savage (I)-------->All or Nothing (2002/I)-------->Edna Dor  -------->44 Inch Chest (2009)-------->Tom Wilkinson (I)-------->A Business Affair (1994)-------->Simon McBurney-------->Harry Potter and the Deathly Hallows: Part II (2011)-------->Emma Watson (II)

   Time taken = 16 seconds

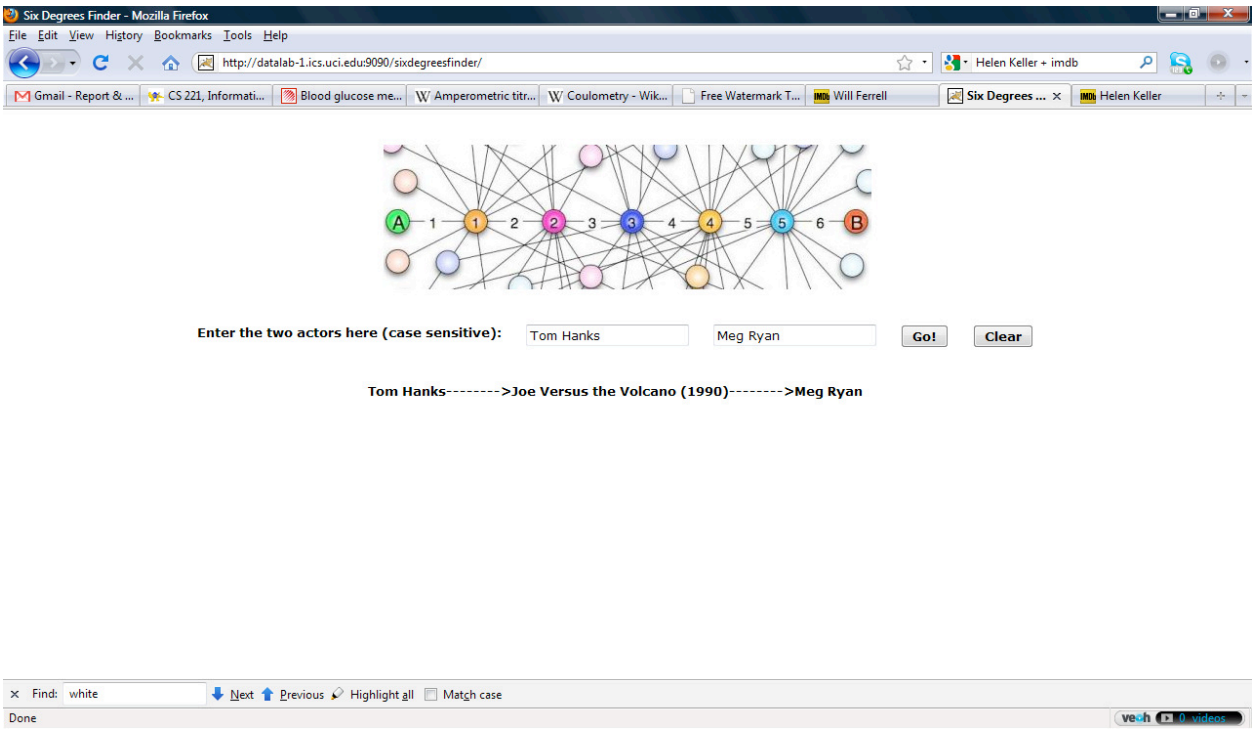
3) Between two unknown actors

Fun-Ti Ai ---> Gui ji shuang xiong (1976) ---> Wai-Man Chan ---> 'A' gai wak juk jap (1987) ---> Mickey (V) ---> Geraftaar (1985) ---> Shyam Kumar (I) ---> Apna Khoon Apna Dushman (1969) ---> Mumtaz (I) ---> Apna Desh (1972) ---> Mukkamala ---> Muthyala Muggu (1975) ---> Murali (I)

Time taken = 183 seconds

3.1 Web User Interface

We created a graphical user interface for the user to submit their queries and view their results, as shown in the figure below.



3.1.1 Features Provided

The Web UI provides a number of interesting features, such as a simple intuitive interface to enter actor names and view the result, actor names validation to ensure that they are how we expect them to be and asynchronous response once computing the links is over at the server side.

Actor names validation involves searching for entered patterns in the first and last names to provide a list of actors that the user could have meant, and, returning the case sensitive equivalents that our application expects.

3.2 Technologies Employed

- Core Java
  - Multi-threading
  - Parsing
- Amazon EC2 Hadoop Framework [2]
  - Map Reduce
- Google Web Toolkit [3]
  - User Interface
  - Asynchronous server calls
- Apache Tomcat [4]
  - Web hosting

3.3 URL

http://datalab-1.ics.uci.edu:9090/sixdegreesfinder/

4. Future Work

4.1 Algorithmic Improvements

A few of the areas for improvement to be made to the current algorithm were search, input handling and data filtering. We'd like to see improvements made in the search algorithm that would allow us to find the shortest path (or paths) between two actors, without forcing the user to wait triple the amount of time of the current search. While the multi-threading cut down the search time drastically, there may be additional improvements or additions that can be made to speed up search time and return the optimal paths. We would also like to be able to handle user input in a more efficient fashion, such as: allow non-ASCII characters (or even correct with non-ASCII e.g. Renee might be Renée), correct or suggest spelling, and make faster stemming suggestions. The final technical area of improvement we'd like to see would be to remove some of the archive data and remove all self-appearances (some were not explicitly labeled like Movie Awards shows). This would be solely to remove what we consider "uninteresting" results, since so many queries were returning connections containing Screen Actors Guild Awards and other such get-togethers that aren't really movies or television shows.

4.2 Expansions

Other areas of improvements mentioned that would expand upon our current project would be to incorporate additional data from outside of IMDB. We have discussed adding live theater performances and those actors, which might be interesting since there is often a lot of crossover between Broadway and Hollywood. We've also considered the effect of adding some sort of popular gossip magazine, like People, where a lot of the same actors and actresses appear, but in a different manner, and we would consider a "connection" to be a co-appearance in an article. This could have very interesting implications in exploring the social networking of Hollywood. Finally, we thought it would be nice to work on the beautification of our website, such as display information, images, or IMDB links for the inputted actors while the search is performing. This would not only distract from the wait time, but would add some character to the site. Formatting the output as well, to create some sort of timeline image rather than plaintext would be another idea to improve the aesthetics of the website.

5.0 Bibliography

[1] Wikipedia. "Six Degrees of Separation" http://en.wikipedia.org/wiki/Six_degrees_of_separation
[2] Amazon EC2 Hadoop Framework http://aws.amazon.com/elasticmapreduce
[3] Google Web Toolkit http://code.google.com/webtoolkit/overview.html
[4] Apache Tomcat http://tomcat.apache.org