$H_{ollywood}A_{ctor}M_{ovie}$ and Bacon

Skye Mah-Madjar, Krisztian Drimba, Joshua Iaboni, Xiayu Lyu.

Monday, March 31st, 2025

Introduction

Many people have at some point learned about the concept of "the Bacon Number". While it may sound related to food, it actually references Kevin Bacon and the idea of "degrees of separation" between actors (Wikipedia contributors, 2025). These degrees of separation are based on the movies that have a commonality between two actors. So, two costars would have a distance of 1, and so on.

On many websites, you can enter the names of two actors, and if a path exists, it will return something that looks a lot like a graph, showing the names of the actors and lines that connect to movies (and possibly other actors) between them in their path (Oracle of Bacon, n.d.).

Recognizing that the Bacon Number problem is represented as a graph helped us visualize how similar questions could be addressed using graph theory. Our initial thought was, could we somehow use this concept to understand which actors are the most influential? Understanding what it meant to be "influential" was difficult, but we eventually decided that "the most influential actors" would be ones with the lowest average bacon number.

This led to our main goal: **enabling users to compute Bacon numbers, visualize actor paths, and rank actors by their average Bacon number.** This became the backbone of our project and inspired us to implement one last major idea.

Partly inspired by the book recommendation system from Exercise 3, we wanted to create a system to recommend movies based on intersecting cast members, which would relate to our bacon number idea. We even wanted to take this a step further and add a filtering system based on release date and ratings for the recommendations, which we also eventually implemented for the functionality of calculating the bacon paths.

Datasets

We used one dataset from the internet in our project, as well as one we computed based on the results from that dataset. The initial dataset was a CSV file of about 200,000 lines, where each line was an actor name, a movie, and information about the movie. Lines would repeat actor names, so there are approximately 10,000 actors and 41,000 movies in the dataset. This dataset is stored in the project zip under the Datasets folder as 'full_dataset.csv'.

Based on this dataset, we calculated and created a CSV file which maps actor names to their average bacon number, as calculated by our compute_average_bacon_numbers function in calculations.py. Since calculating each actor's Bacon number is time-intensive (around 20 minutes), we precomputed these values and saved them as a CSV file to avoid redundant calculations for every program run.

In our final rankings, which is touched on more later in the report, we excluded actors with abnormal Bacon numbers (e.g., scores of 1.0) as outliers, which did not reflect their true influence.

Computational Overview

Graphs play a central role in both aspects of our project. For the actor graph, each vertex represents an actor, and the edges are formed based on whether actors costar in movies together.

Actor Graph

The actor graph is created by utilizing our actor dataset. It creates all the vertices and edges and also allows for the actor vertices to contain information about the movies, as given in our dataset. In graph_create.py, the function initialize_graphs uses the function load_csv_file and the function create_actor_graph to create the actor graph.

The bacon number calculations and average bacon number calculations all rely on our Breadth First Search (BFS) algorithm that we implemented, which calculates the shortest paths between the vertices of our actor graph. BFS is a level-by-level traversal that utilizes a queue to keep track of vertices it needs to explore. It begins with a chosen vertice, then explores its neighbours, its neighbours' neighbours, and so on. It stores what it needs to explore in a queue so that it can efficiently go through the list of vertices that need to be explored. Thus, it can be efficiently used to calculate single shortest paths, between two specific vertices, or even find one vertex's shortest path to all other vertices, since it already explores a full graph level-by-level.

We have two BFS algorithms, the functions shortest_path_bfs and shortest_path_bfs_filtered, where they compute the shortest path, and the filtered version only uses movies that fit the filtering when calculating the path. Then, shortest_distance_bfs works similarly but returns a dictionary of actor names and their bacon numbers relative to the inputted actor.

The filtering has two options, release date and rating, where users can input the threshold as an upper and lower limit. For example, only asking for movies between 2002 and 2012, or movies with rating between 8.5 and 9.0. Then, when calculating paths or average bacon numbers, it will NOT use movies as edges if they do not fall within this threshold. This is our filtering system.

Thus, we utilized BFS to compute the paths between two actors and each actor's average bacon number, which we could then display as our results for the first part of our program. The functions that calculate these results are all in the calculations file under the Bacon Number header.

When the user asks to compute a specific path, it also visualizes a graph in the user's browser. This is done in graph_display in the function visualize_actor_path. ChatGPT helped explain how to use NetworkX and Plotly and create code for graph_display. This function first gets the effective path that will be used (or the fallback actors if no real path exists) and then creates a corresponding NetworkX graph in build_actor_graph. Each actor in the path is added as a node and if a valid connection exists, which is determined by get_common_movies, edges are created between consecutive actors with the common movies stored as strings in the edges. Next, the helper function compute_layout_and_scaling computes the positions of the nodes using a NetworkX layout as well as a scaling factor and canvas buffers to adjust the size and spacing of the nodes, edge widths, and fonts. Helper functions then generate Plotly traces for the nodes, edges and edge labels as return type Scatter. In Plotly, a trace is a visual component of a graph which holds data points and styling information. Once all the traces are made, they are combined into a Plotly Figure using the build_figure helper function. A figure is the complete interactive visualization which can be interacted with online using Plotly tools (like zoom and pan) or saved to an image file.

Movie Graph

The movie graph is not initialized in the same way as the actor graph. Instead, we initially create a movie dictionary as seen in graph_create in the function load_csv_file. In the calculations file, we have the function get_similarity_score_dict, where we use the concept of similarity score inspired by Exercise 3 to find the scores between movies in this dictionary, as that runs much more efficiently than comparing vertices in a graph. We changed to this system due to that exact reason.

Then, using those scores, the function get_recommendations returns either a list or a dictionary of recommended movies containing as many recommendations as the user asks for. It returns a list in descending order of similarity when the user does not filter and a dictionary in descending order of similarity when the user does, which also displays either the rating or release date of the recommended movie, depending on the option of filtering the user inputs. Like the bacon paths, the user can enter the threshold of filtering.

Then, in graph_create, the function create_recommended_movie_graph uses either the list or dictionary of recommendations to create and visualize a graph of recommendations, which is visualized in graph_display in the function visualize_movie_graph. ChatGPT helped explain how to use NetworkX and Plotly and create code for graph_display. The movie graph visualization follows a similar process as the actor graph visualization, but instead displays the movie recommendations based on similarity score. The movie graph is first converted into a NetworkX graph using the movie_graph_to_networkx helper function. Each movie node includes attributes like its label and similarity score. Edges are created between movie nodes with the similarity score edge labels converted to percentages. Then, the layout positions and border margins are computed. The node traces for movies are returned as a Plotly type Scatter, with the node sizes scaled proportionally to the similarity scores, which allows for easier viewing of movies that are more similar to the inputted movie. Edge traces and edge labels traces are also generated as type Scatter to represent connections between nodes and display the similarity percentages on the edges. The traces are then combined into a Plotly Figure, which is the complete visualization and can be interacted with online or saved as a specified image, just like the actor graph visualization.

Instructions for Datasets and Running the Program

The two necessary datasets used when normally running the program have been uploaded as a part of the zip file. When you run main.py, a menu will pop up with a few options. This contains the entire functionality of our program other than the fact that a few of the options may display graphs through your browser. To use an option, just type in the number of the option you want to use.

The first option will display the ranking of actors in increasing order of average bacon number, which satisfies our question of "the most influential actors".

```
Your choice: 1
Number of actors: 10
1 : Samuel L. Jackson with average bacon number: 1.9046623458080703
2 : Frank Welker with average bacon number: 1.9176249215973238
3 : Danny Trejo with average bacon number: 1.9184612168095339
4 : Bruce Willis with average bacon number: 1.9218063976583735
5 : Richard Riehle with average bacon number: 1.929542128371315
6 : Steve Buscemi with average bacon number: 1.937800543591888
7 : Stanley Tucci with average bacon number: 1.941772945849885
8 : John Goodman with average bacon number: 1.9422956303575163
9 : Nicolas Cage with average bacon number: 1.942609241062095
10 : Liam Neeson with average bacon number: 1.943654610077357 3
```

Figure 1: Option 1 in the Menu

The second option allows you to get insight into a specific actor and how they contribute to the rankings.

```
Your choice: 2
Actor Name: Jennifer Coolidge
The Average Bacon Number for Jennifer Coolidge is: 2.0248797825632447
The actor is number 466 out of 9610 in the overall rankings.
```

Figure 2: Option 2 in the Menu

The third option will allow you to find the bacon path between two actors and get both text outputs in Python and a visualized graph of the bacon path it calculates.

The fourth option will allow you to find movie recommendations for a specific movie and get both text output in Python and a visualized graph of the movie recommendations it calculates.

Changes to the Project Plan and Final Submission

Many of the changes to the plan were based on areas we were worried about due to inefficiency and limitations. For example, to overcome the fact that calculating the average bacon number of each actor, which would be 10,000 actors compared to 10,000 actors each time, would be super inefficient in terms of time, we decided to calculate it

Figure 3: Visualized Graph for some Input using Option 3 in the Menu

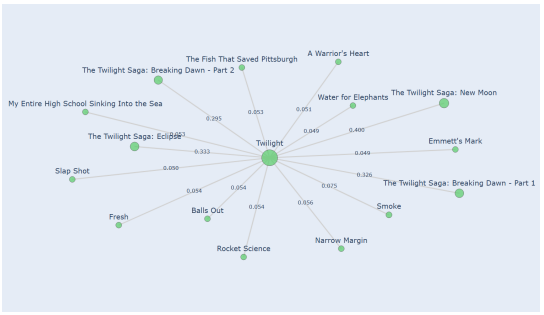

Figure 4: Visualized Graph for some Input using Option 4 in the Menu

once and store it, something recommended to us by the TA.

Similarly, for the movie graph, we ended up finding out the dataset had around 41,000 movies after playing around with the calculations, and we understood that it was infeasible to display all of these. That is why we decided to display the recommendations found for an input movie, as we thought it would be a nice way of displaying our results and would not be too inefficient, depending on the number of recommendations the user asks for.

Discussion Section

The results of our project do accomplish what we strived to achieve. The main two ideas were to create a system to 1. Calculate the most influential actors using the concept of average bacon numbers, and 2. Create a movie recommendation system. We were able to accomplish both of these things.

However, there were some obstacles and limitations that we encountered that changed the way we were going to proceed in our work. As explained in the previous section, there were lots of problems with efficiency, time, and more.

The TA recommended that we use Dijkstra's algorithm to implement the shortest path, which was also our original plan. We initially implemented and used Dijkstras, but then we found that to be inefficient, as it would not even calculate the average bacon numbers after 2 hours. So, we decided to switch it to the Breadth First Search (BFS) algorithm and utilize queues, which ended up being way more efficient.

Thus, we ultimately were able to get by any limitations for computing all the average bacon numbers, but the limitations that we had were primarily based on visualizing our results, as we found that there was nothing meaningful we could visualize from our calculations other than the paths we did end up showing. While it is not necessarily a large graph that displays much information, we thought it was a neat and sleek way of displaying the results outside of a Python console.

Similarly, for the movie graph, as explained above as well, we ended up only being able to display the recommendations for a movie rather than an entire graph, which was a bit disappointing but did accomplish our main goal.

There are quite a few things that our group immediately pointed out for the next steps. The similarity score system, which used a similar algorithm from the book review one in Exercise 3, did not work as efficiently in ours since it was based upon cast members rather than neighbours. Since vertices only share edges when they share at least 1 cast member, this means that recommendations can only be for immediate neighbours of an input movie, which leaves a whole avenue of unrecommended movies since the recommendation is purely based on cast members.

Thus, a next step would be to find a different similarity score measuring system that might produce more interesting results, especially since for our recommendation system, most of the higher scores we see achieve about a 25% similarity, which seems pretty low.

Another next step that we discussed would be using visualization in a more meaningful way. Most of our work was put into creating the algorithms that would compute bacon numbers and find the similarity score. Since these databases were so large, there would be no easy way of displaying these vertices en masse, and we instead opted to only visualize the specific computations the user could perform in a menu. A next step might be to find a way to allow the user to interact or visualize more things, thus incorporating the visual aspect more into our project and problem domain.

Lastly, our filtering system was a fun addition that also produced interesting results. It would be interesting to see more filters added or more places for filtering, such as when calculating average bacon numbers, and to see how that influenced results.

Overall, this project provided a lot of interesting results. The bacon number and movie recommendation domain allowed us to work through very challenging algorithms and find very interesting results, such as how in the top 10 actors in the average bacon number ranking, we collectively only recognize 6 of them, or that Samuel L. Jackson is number 1! While the project answered many of our questions, it also left us wondering what other computations or methods of filtering we could apply to this data to find out more about the movie industry.

References

[1] DataCamp. (n.d.). *Breadth-first search in Python.* Retrieved March 23, 2025, from `https://www.datacamp.com/tutorial/breadth-first-search-in-python`

[2] FavTutor. (n.d.). *Breadth-first search in Python.* Retrieved March 23, 2025, from `https://favtutor.com/blogs/breadth-first-search-python`

[3] GeeksforGeeks. (n.d.). *Breadth-First Search or BFS for a graph.* Retrieved March 23, 2025, from `https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/`

[4] hugequiz. (2021, May 5). *IMDB Films By Actor For 10K Actors.* Kaggle. `https://www.kaggle.com/datasets/darinhawley/imdb-films-by-actor-for-10k-actors`

[5] OpenAI. (2025). ChatGPT [Large language Model]. `https://chat.openai.com`

[6] Oracle of Bacon. (n.d.). Oracle of Bacon. `https://oracleofbacon.org/`

[7] Wikipedia contributors. (2025, January 12). *Six Degrees of Kevin Bacon.* In *Wikipedia, The Free Encylopedia.* Retrieved March 3, 2025, from `https://en.wikipedia.org/wiki/Six\_Degrees\_of\_Kevin\_Bacon`

[8] Wikipedia contributors. (2025, January 12). *Kevin Bacon.* In *Wikipedia, The Free Encylopedia.* Retrieved March 3, 2025, from `https://en.wikipedia.org/wiki/Kevin\_Bacon`