Minegrapht Modding

Sam

April 2024

1 Introduction

This project's goal was to scrape data on Minecraft mods that could then be used to make a graph with the mods as nodes and the edges as some measure of similarity.

There are many other neat things that could theoretically be done with this data, perhaps a recommendation system of new mods you might like, identifying communities of mods or mod authors, or any number of other projects. But for the scope of this project we're mostly focusing on collecting and organizing the data and making a few graphs just to play around with it.

2 Background

For those unfamiliar, Minecraft is an extremely popular sandbox game released around 2011 (or 2009, depends who you ask). Since its release it's had a strong modding scene where modders write code to change the behavior of the game, tweaking existing bits and adding their own new features. Mods are quite popular among players too, with many players downloading modpacks as a curated selection of mods to play.

Until relatively recently almost all mods were made using Forge Mod Loader to load the mods into the game and interact with the base game code. More recently, Fabric has gained a lot of steam as a lighter modloader with a separate Fabric api mod. Quilt is also a decently popular loader forked from Fabric with a very dedicated community, but comparatively small playerbase. Finally, NeoForge is a very recent hard fork of Forge with almost everyone switching (minecraft modding drama is weird). In general, since multi-loader mods take a bit more effort, most mods get made for one loader or another, which creates a noticeable divide.

Similarly, curseforge has historically been the go-to hosting site for mods and modpacks with Modrinth being a newer open source site that is better in almost every way except not being as popular. Curseforge also has a much wider audience, with it hosting mods for many other popular games, while Modrinth came directly out of the Minecraft modding scene and only hosts Minecraft mods.

Over the years there have also been groups/platforms that publish their own modpacks, usually with dedicated launchers, the most popular one still around is Feed The Beast (ftb).

3 Methodology

3.1 Data Sources & APIs

The first decision to make was where to get the data. For this project I decided to scrape only curseforge since it has a significantly larger userbase and history compared to Modrinth. Unfortunately, the curseforge api isn't the greatest, specifically it has no way to get the dependents of a given mod, meaning no way to see what modpacks a given mod is in or what other mods depend on it. This is where a site called modpackindex.com and its API comes in. With this modpackindex (mpi) api we can fetch all the modpacks hosted on curseforge and ftb that any given mod is in, as well as all

the mods in any given modpack.

Every curseforge mod has its own project id, throughout the project we use this as a 'true name' of sorts for identifying any given mod. We can use a project id with the curseforge api to get some basic metadata on the mod. Specifically we get the mod's name, slug (human readable id), source url, categories, devs/authors, dependencies, summary, logo url, and download count. Some of this data is iffy, the categories aren't the most helpful and the authors listed only includes certain roles like owner or maintainer. Also, since the curseforge api only returns data on the two most recent files, we can't reliably get comprehensive data on what modloaders and versions a given mod supports.

As for the mpi api, there's no way to fetch data using the curseforge project id, so we need to get the mpi id by taking the first result of searching the mod's name. We can then start fetching the modpacks that a given mod is in. Unfortunately, these requests return quite a lot of the modpack metadata as well, so they're paginated with a cap of 100 modpacks per request. I originally planned to fetch all modpacks each mod was in, but quickly realized that was not going to happen when I timed out fetching the 536th page of modpacks with the JEI mod. As a compromise we only save up to the first 20 pages of modpacks, so up to 2000 modpacks per mod. Only about 660 mods ended up hitting this limit.

For modpacks we use their mpi id as their 'true name', since some of the packs aren't on curseforge and the mpi api gives us all the information we need. The only information we save is a list of mods in the pack pulled from mpi. Note that this is a single non-paginated api request.

Our final data source is GitHub, through the GitHub api and Octokit library. We can get the source url from the curseforge metadata and parse out the info we need with a simple regex. For each mod repo we want to fetch all of the issues, pull requests, and comments. Specifically, we'll be saving a list of users that interacted with each repo and a count of their interaction types, with the types being 'opened an issue', 'opened a pr', 'commented on an issue', and 'commented on a pr'. The Octokit library made this part pretty easy with its built in throttling.

3.2 Scraping

Now that we're familiar with the APIs, we can start talking about the actual scraping. All of the scraping code was written in Nodejs with Typescript. The scraping code is meant to double as a wrapper for the saved data, so someone using this code can call `mod.getCFMeta()` to get the curseforge meta data regardless of if it's saved in a file or needs to be fetched. Once any new data is fetched it can be easily saved by calling `mod.saveToDisk()`. This system both makes it easier to work with the scraped data and helps the scraper run non-continously since it can stop and start without losing any information.

The main scraper uses two queues, a mod queue and a modpack queue. While there are mods in the mod queue it fetches their curseforge metadata and the modpacks they're in, adding any new modpacks it finds to the modpack queue. When the mod queue is empty, it starts scraping from the modpack queue to replenish it. It does this until both queues are empty or we hit rate limits. Since the mpi api has a limit of 3600 requests an hour, we usually scraped a couple hundred mods an hour. This meant the scraper required a good bit of babysitting to start it back up every hour.

After the main scraper runs, the GitHub scraper and the logo scraper can run and get their respective data for any new mods found. The logo scraper is really only needed if you want the images locally for whatever reason, and since they take up a good 1.5GB I don't push them to the repo. There's also no rea-

son you can't go back and forth, if we want more mods we can just keep running the main scraper for a bit and then go back and rerun the other scrapers after.

3.3 Graphs !

Now that we have some data we can start playing with it. The graph data was put together still in Typescript and then sent over to python to be toyed with.

I started by transforming the data into a few different views, specifically a `USER_VIEW` that maps github users to the repos they interact with and a `MP_VIEW` that maps modpacks to the mods they contain (that we scraped). Note that the `MP_VIEW` is different from the scraped modpack data since we only directly scrape modpacks to fuel our mod queue but this view contains all modpacks referenced in our mod scraping. These views are quick to construct but give us constant time lookup of mods by user and modpack, which helps us more efficiently construct our graphs.

The first graph I made was a simple dependency graph, with mods as nodes and directed edges from each mod to any mod it depends on. I then loaded it into python with networkx and used pygraphviz with the sfdp layout mode to render the graph. Since we scraped the images before, we can use those for the nodes of the rendered graph to make it easier to identify what each node actually is. It took a while to find a graphviz setup that made a decent graph render, some of the issues I ran into included nodes overlapping, poor scaling that maxed out the image writer resolution making giant and unreadable images, and edges covering nodes.

Next I made the GitHub data graph, with mods as nodes and edges for each shared contributor weighted by number of interactions. This ended up being much denser. The first render of it took a while and came out as mostly a large black blob of edges. To get around this I flattened the data, so instead of being a multi graph with a unique edge for each shared user between any repos it just uses a single weighted edge between each repo with shared contributors. This ran faster but still gave a blob of edges. To reduce the number of edges I added a threshold that kept only the top 5% of edges on each node, this gave a much more readable render but may have sacrified some precision in the shaping. There's also a render of the subgraph induced on the neighborhood of my mod 'HexGloop' using the same threshold method.

Finally we have the modpack graph. This was by far the hardest to generate the data for simply because of how dense it was. My initial solution was to iterate over every pair of mods and take the intersection of modpacks that they're each in, but this was highly inefficient as most mod pairs will have relatively few packs in common and calculating the intersection is relatively expensive. It took a couple minutes to get 2% finished, so I killed the program and tried another way. The second attempt iterated over all modpacks using the `MP_VIEW` and incremented a counter for each pair of mods it found in the modpack. At the end these counters would tell us how many packs each pair of mods shared. This ran much faster! but was consistently freezing up around 92%. Switching from storing the counters in raw js objects to using the Map class and cleaning up some of the other code got it through the counting section but it started hitting memory limits. So finally after giving it a good 8 gigs and a couple minutes we got 280MB of raw edge data! Rendering the graph was about the same process as before except this time I discovered I can put the nodes over the edges so they don't get covered and used a higher threshold for which edges to keep with an additonal threshold for which edges were visible, which should improve the shaping of it while keeping it readable.

During the graph making process, GitHub started throwing warnings and errors that my files were too large and suggested using Git Large File Storage (git lfs). It seemed easy

enough to set up, so I did and committed my data and graph renders with it, but turns out that if you go over the quiet 1GB limit you have to either pay $5 a month for more storage or your repo can't be properly cloned until you entirely remove git lfs from your commit history, and it's not cleared from lfs storage until you delete the entire repository. Luckily I hit that limit quickly and could just reset back to a previous commit, but I wasn't able to commit some of the larger graph data files, so they'd need to be rebuilt locally if you want to use them.

4 Results

All of the scraped data, graphs, and renders are available at `https://github.com/SamsTheNerd/minegraphtmods`. The graph renders are in the `demos` folder.

4.1 Data

We scraped a total of about 8000 mods, with about 5000 of them having github repos we could scrape, which gave us around 122,500 total users who interacted with these repositories.

The data had a few strong biases. First of all, since we're only scraping curseforge mods, we're completely missing all the mods that for one reason or another are only hosted on Modrinth. I would guess that those mods are going to lean towards being newer, using fabric/quilt more, and being made either by newer mod devs who don't want to set up and manage a curseforge project or older mod devs who have had bad experiences with curseforge and are purposely avoiding it. Whether my guess is true or not, it's unfortunate to not have those mods represented in this data.

The other bias is less direct. Since our scraper collects new mods by modpack and dependency only, it's likely to find popular mods since they're going to be in many modpacks and have many chances to be found, but it's complete chance for whether or not a smaller mod that's only in a few modpacks is going to be picked up. Also, since the modpack queue moves much slower than the mod queue, it can get stuck in certain modpack niches for periods of time. As far as I could tell, this seemed to mainly prevent it from picking up on older mods on its own, so while it seemed to have already scraped most modern popular mods, I had to manually stick some older mods in the queue for it to start finding them. It's also possible that there were other bubbles it didn't pick up on that I had my own blind spots for as well. Running the scraper more would probably help with all of these just by letting it get more data, but with all that being said, there are still a lot of mods here, many that I recognize and many that I don't, so I'd still consider it a success.

4.2 Graphs

Since we have so much data and no real ground truth for how connected any two mods should be, the best way to see if our data looks right is just to look at the graph renders. This isn't a perfect method for a number of reasons. It requires some background knowledge of the mods and the graphs themselves aren't the most finely tuned. Also, the layout engine, `sfdp`, that I used is meant for quickly laying out large graphs so while it's probably getting the general shape right, it doesn't seem to always put nodes as close to eachother as I'd expect based on their edges.

For the rest of this section I'll be referencing specific graph renders, I suggest opening the repo to be able to follow along. I'll also be referencing specific mods in the graph, I'll give any relevant information about them, but it may be helpful to look them up to see what their icon looks like so you can find it on the graph.

4.2.1 Dependency Graph

First let's look at our dependency graph (`dependency_graph_all.png`). You'll notice a few large mods with many connections, these

are mostly library mods and very popular content mods. By far the largest here is Fabric API in the middle left. As previously mentioned most mods using the Fabric modloader use this, there's not a Forge equivalent in the graph since its api is built into the modloader. Besides Fabric, we can see that library mods like Quilt API, Architectury API, Kotlin, Trinkets, Curios, and documention mods like REI, JEI, Jade, and Patchouli have some of the most connections, as we'd expect.

Some of the other most connected mods here are popular content mods, like Create (fabric version is top middle, forge is a bit lower right), Origins (bottom middle), and Farmer's Delight (middle right). You'll notice a lot of Create addons around it with the identifiable blueprint icon template and the same with various origins addons around it. Interestingly though, some of these aren't as physically close to their neighbors as you'd expect given their limited connections. While it's quite visually noticeable with the addons, it seems to also apply elsewhere in the graph, for example my mod WNBOI is quite far away from HexGloop despite having no other connections. I believe this is just from the layout engine as previously mentioned.

Finally, looking around the edges there are many completely separated nodes. I believe these are mostly forge mods that just don't use any libraries, but some may just have their dependencies set up wrong too. Interestingly, these outer mods seem to have less designed icons, with some of them just being screenshots from ingame, I have no idea why this is but it's interesting. You may also notice some mods have their own little noticeable communities or multiple mods depending on a small personal library, like for example YUNG's Better structure mods in the very bottom middle, or Buildcraft's modules in the very top midright. It's interesting that the layout seems to put these small and strong communities on the outer edges.

4.2.2 GitHub User Graph

Moving on to our GitHub graph, feel free to take a peek at `gh_graph.png`, it's a beautiful black blob. For a better graph we have `gh_graph_bettericons_...png`. As mentioned before, this is approximately 5% of the edges and it's still incredibly dense. We'll see in the middle a lot of popular mods, some of which were mentioned before on the dependency graph. You may also notice that some mods seem to appear twice, this is because some older mods were originally made for forge but later got fabric ports that are listed as different projects, however many of these share the same repo so they end up right next to eachother.

To get an idea of how connected this graph is, there's a mod called Hex Casting, which is a relatively niche but still quite popular mod about a magical hexagon based esolang, its neighborhood in the full github graph is somewhere around 2000 nodes, so approaching half the graph. My mod HexGloop (which if you haven't guessed is an addon to Hex Casting) has a neighborhood of around 450 nodes, which is somewhere around 10% of the full graph. I think that alone is quite neat, that the minecraft modding scene is so interconnected.

Since this graph is so dense, let's zoom in on the subgraph induced on HexGloop's neighborhood, `gh_gloopgraph _thresholded.png`. You'll find HexGloop itself around the middle right, with Hex Casting and the other Hex addons close by. Although for some reason all five of the other Hex addons are grouped very close together, almost touching, but Hex Casting and HexGloop are slightly further. You'll also see my other mods WNBOI and Ducky Peripherals close to it and eachother, which makes sense since I'm quite active in both repos.

Although they're partially obscured, you can also see some of the more popular mods poking through in the center, as well as a

lot of the Violet Moon mods (identifiable by the purple V in the corner) are quite close to eachother. Interestingly, REI, EMI, and Patchouli all seem quite close too, I'm not sure if this is just a coincidence or not since they're all documentation mods but I don't believe have much dev overlap. Strangely though, ComputerCraft:Tweaked (CC:T) and ComputerCraft:Restitched (CC:R) are quite distant despite CC:R being a fork of CC:T, and both are quite far from Ducky Peripherals and Advanced Periphals, which are both ComputerCraft addons.

Overall this GitHub graph seems to be pretty good at clustering mods as I'd expect it to. It's worth noting that edge weights here are simply given by the sum of interactions in both repos from any shared users, so more popular mods are likely going to have larger weights due to having more interactions. Experimenting with different weights per user could improve or change the results as well, for example counting only the PR interactions would likely show stronger connections based on dev teams, while weighing based on issue interactions and decreasing the impact of the most active users in each repo would hopefully show more of player preference (or atleast players who happen to be active on github, which is quite a strong bias).

4.2.3 Modpack Graph

Finally we have the modpack graph (`mp_graph_thresholded.png`). This was by far the densest graph, with I believe around 8 Million edges before thresholding. Immediately you'll notice that the graph has two very dense sections, these appear to be made up of forge and fabric mods with a strong divide between them, which is quite cool to see that show up. Unfortunately that's about the only useful information in this graph. Since the edge weights were calculated based on number of shared modpacks divided by the minimum of the count of modpacks that each mod was in separately, you end up with mods that are only in a few modpacks having very high weights with almost everything around them while mods in a lot of modpacks get pushed to the outer edges of the graph with lower weights.

5 Conclusion

This project was a great start at collecting and analyzing data on mods. The graphs made for this project showed that the data does work to measure similarity at least to some degree, with the github data being the most promising so far.

For future work, tweaking the edge weight calculations and combining the various data attributes collected into a single graph could make a much more useable general purpose graph. It may also be worth investigating using a weighted directed graph to better represent some sort of non symmetric similarity. And if possible, it'd be great to scrape Modrinth data as well, although that would require some reworking of the core data wrappers, including switching away from project ids/mpi ids as true names.