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 reason 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.