DEPENDENCYVIS: HELPING DEVELOPERS VISUALIZE SOFTWARE DEPENDENCY INFORMATION

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Nathan Lui

June 2021

COMMITTEE MEMBERSHIP

TITLE:	DependencyVis: Helping Developers Visualize Software Dependency Information
AUTHOR:	Nathan Lui
DATE SUBMITTED:	June 2021
COMMITTEE CHAIR:	Bruno da Silva, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Ayaan Kazerounmi, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Phoenix Fang, Ph.D. Professor of Computer Science

ABSTRACT

DependencyVis: Helping Developers Visualize Software Dependency Information

Nathan Lui

The use of dependencies have been increasing in popularity over the past decade, especially as package managers such as JavaScript's npm has made getting these packages a simple command to run. However, while incidents such as the left-pad incident [1] has increased awareness of how vulnerable relying on these packages are, there is still some work to be done when it comes to getting developers to take the extra research step to determine if a package is up to standards. Finding metrics of different packages and comparing them is always a difficult and time consuming task, especially since potential vulnerabilities are not the only metric to consider. For example, considering how popular and how actively maintained the package is also just as important [8].

Therefore, we propose a visualization tool called `DependencyVis` that is specific to JavaScript projects and npm packages as a solution by analyzing a project's dependencies in order to help developers by looking up the many basic metrics that can address a dependency's popularity, activeness, and vulnerabilities such as the number of GitHub stars, forks, and issues as well as security advisory information from `npm audit`. This thesis then proposes many use cases for `DependencyVis` to help users compare dependencies by displaying the dependencies in a graph with metrics represented by aspects such as node color or node size.

ACKNOWLEDGMENTS

I would like to acknowledge and thank my thesis advisor, Dr. Bruno da Silva, for an entire year of his guidance, advising, and assistance for this thesis work. I also would like to extend my gratitude to my committee members, Dr. Ayaan Kazerounmi and Dr. Phoenix Fang, for spending the time and energy in reviewing my thesis and as teachers in my graduate classes. Over the course of many graduate classes, I have had many classmates to share our accomplishments and concerns with. I thank these classmates for sticking by my side in the emotional roller coaster of learning to push ourselves in our field of interest together.

I also would like to thank my mother and father for their constant support and understanding. They have both worked hard to give me a life where I am free to try out a wide variety of subjects and ultimately be free to pursue my passions. I also appreciate my sister for pulling me out of my chair from time to time for her food adventures. Also, many thanks to Cindy Do for providing the love and companionship I needed to get through this stage of my life, especially given the circumstances of this year.

Also thanks to Andrew Guenther, for providing the LaTeX template for this thesis.

TABLE OF CONTENTS

Page

LIST OF FIGURES viii

CHAPTER

1 Introduction 1

1.1 Objective 3

1.2 Overview 4

2 Background 5

2.1 Dependencies 5

2.1.1 Categorizing Dependency Information 6

2.1.2 Package Managers 7

2.1.3 Dependency Tree or Graph? 9

2.2 Software Visualization 10

3 Implementation 14

3.1 Metrics 16

3.1.1 Popularity 17

3.1.2 Activeness 19

3.1.3 Vulnerabilities 22

3.1.4 License 23

3.2 Simulating Adding Dependencies 24

3.3 Accessing GitHub API 26

3.4 Accessing npm API 28

3.5 Visualizing Data 29

3.6 Storing Data in a Database 32

3.7 Deploying Online 34
4 Use Cases 36
4.1 General Uses 36
4.2 Finding Specific Metrics 38
4.3 Comparing Dependencies 41
5 Related Work 48
5.1 Determining a Good Dependency 48
5.1.1 Addressing Vulnerabilities 48
5.1.2 Addressing Maintenance 49
5.2 Other Software Visualizations 50
6 Conclusion 53
6.1 Implications 54
6.2 Future Work 55
6.2.1 More Metrics 55
6.2.2 Potential Case Studies 57
BIBLIOGRAPHY 60

LIST OF FIGURES

Figure		Page
2.1	Definitions of words in relation to "Project"	6
2.2	Radial Visualization of Dependency Usage [18]	12
2.3	Node-edge Visualization [9]	12
3.1	General Library Architecture	15
3.2	Labeled Title Screen of `DependencyVis`.	17
3.3	Labeled `DependencyVis` After Title Screen.	18
4.1	Flowchart for using `DependencyVis`	37
4.2	Use Case 1: `DependencyVis` Output	38
4.3	Use Case 2: Adding a Library	39
4.4	Use Case 2: Full Dependency Graph	39
4.5	Use Cases 3, 4, 5: `Axios.js` Dependency Graph with Audit and Stars	42
4.6	Use Case 6: Hovering over `serve-static`	43
4.7	Use Case 8: Selecting the License Option for `d3.js`	43
4.8	Use Case 9: `d3.js` Dependency Graph	46
4.9	Use Case 9: `vis.js` Dependency Graph	46
4.10	Use Case 10: Comparing `vis.js` and `d3.js` by Forks and License	47

Chapter 1

INTRODUCTION

Decades ago, reusing software systematically was simply a dream that would have taken more effort than it was worth it at the time to achieve [27, 14]. Today, software reuse via external packages has become so common to a point that developers barely take the time to consider what code they are reusing for their projects [8]. These packages of code made for developers to reuse in their projects are called *libraries*. When these libraries are used in a project, they are considered *dependencies* to that project. In this thesis, *libraries* and *dependencies* will be used interchangeably.

Dependencies have many benefits. They save a lot of time, essentially offloading the work of one chunk of code to another developer or group of developers to both develop and maintain this dependency. This prevents a lot of developers from repeating the development of a common functionality simply relying on dependencies to do the work instead. This gives the opportunity of using more robust code as multiple developers have likely checked and tested the code in a dependency then if a developer was to write the code itself. The developers of dependencies can also help maintain the dependencies, updating them in case of bugs, implementing more features and enhancements —essentially providing robust code [27].

However, while it has become easy to systematically reuse software by importing external libraries as dependencies, it has also become riskier in different ways, since this is essentially an activity of incorporating external pieces software to the software under development. This operation may add maintainability and security burden as well if dependencies are not carefully analyzed. For instance, it is unlikely that

the maintainers are people we know, so we must trust people we do not have a relation with to be on top of updating and fixing a library. Therefore, it is possible to depend on a library that may not ever be updated anymore (or not enhanced with the frequency you would expect) or have bugs that might not be fixed anytime soon. If any of these bugs are found to be exploitable by users, then depending on that library would be introducing vulnerabilities to the program. Incidents such as the Equifax data breach and the leftpad package removal are good examples of problems with dependencies [16]. Even if a library has no vulnerability problems, it could also be possible that the library itself has a dependency that does. Dependencies of dependencies are called *indirect dependencies* or *transitive dependencies* and can be another serious point of consideration when looking at libraries.

This introduces the problem that developers are not taking the time to research dependencies to see if they can find vulnerabilities associated with them and if they are being constantly updated [32, 24]. Pashchenko et al. [32] performed a qualitative study on developers to find that many of them focus on the functionality support of a library rather than the security and occasionally check licenses when in an enterprise to avoid legal issues. And who is to blame them? There are so many aspects to consider for a dependency that developers would rather just get started instead of worrying about every detail of their dependencies. Cox [8] provides an overview of many of the metrics that developers should be considering. Therefore, in this work we have approached this problem by offering a tool for accessing information concerning dependencies in a project which includes the aggregation of useful dependency attributes in visual and textual format.

1.1 Objective

The ability to visualize software is a very helpful tool for understanding information about a software project or a library quickly. They can help in identifying patterns, finding problems, and exploring potential trends [12]. As software projects become more and more complex with object-oriented programming, functional programming, testing, dependencies, and more, it is becoming increasingly difficult to keep track of all the different aspects of software projects today. This overwhelming amount of information needed for developers to develop and evolve software makes software visualizations a valuable tool and technique for organizing all information quickly and supporting developers in their tasks.

When it comes to reuse software by incorporating dependencies a lot of information need to be considered. Therefore, the objective of this thesis is an interactive software visualization tool called `DependencyVis`[1] to gather and display this information, so that developers would only need to go to one place to analyze a majority of the metrics recommended to decide if a dependency is desirable to them. `DependencyVis` is meant to assist developers in making an informed decision about their dependencies —not to make a decision for them.

Starting with a public repository on Github, `DependencyVis` will gather information that could be useful in deciding if a dependency is suitable or not. This information would relate to categories such as popularity, activeness of maintainers, vulnerabilities, and licenses. All mentioned in the article by Cox [8]. `DependencyVis` will then display this on a graph of nodes and edges where the nodes represent libraries and the edges represent dependencies between those libraries. The information would be represented by the size and color of the nodes for easy comparison between the different

[1] `http://dependencyvis.herokuapp.com/`

libraries. On top of the visualization, `DependencyVis` also shows dependency data as well such as licenses, vulnerabilities info, and activeness and popularity metrics as possible inverse indicators of maintainability risks. We then provide many use cases in order to demonstrate how such a representation would help developers make an informed decision about their dependencies.

1.2 Overview

The rest of this thesis is organized in the following way. Chapter 2 provides the background necessary for the thesis and understanding `DependencyVis`. Chapter 3 presents the approach used to develop the tool `DependencyVis`. Chapter 4 proposes use cases for the tool. Chapter 5 presents related work. Finally, Chapter 6 discusses the implications behind the use cases and possible future work for `DependencyVis` and concludes this thesis work.

Chapter 2

BACKGROUND

This thesis showcases `DependencyVis` which is a software visualization tool that displays information about dependencies within the domain of a specific package manager known as npm. This makes it important to understand the potential of software visualizations, the definitions of different kinds of dependencies, and the prevalence of package managers.

2.1 Dependencies

As defined in Chapter 1, *libraries* are packages of code designed for reuse. When a software project depends on the library, the library would be called a *software dependency* which we will simplify by calling it a *dependency*. However, dependencies of a project can have their own dependencies otherwise known as *indirect dependencies* or *transitive dependencies* to the project. This means that a *direct dependency* is the libraries that the project explicitly depends on. A *dependency graph* is a project and all of its dependencies and how they connect. We describe the intricacies of a dependency graph in 2.1.3. A *layer* is all the dependencies that are the same distance from the project. For example, layer 1 would be all the direct dependencies as they are all one edge away from project. Figure 2.1 diagrams these definitions in relation to a project. The arrows point to the library that is being dependent on. For example, Project points to Library 1 and Library 2. This means that Project directly depends on Library 1 and Library 2. It is also important to note that here on out, *package* is

defined to be a npm library and *project* is defined to be a personal repository created by a user of `DependencyVis`.

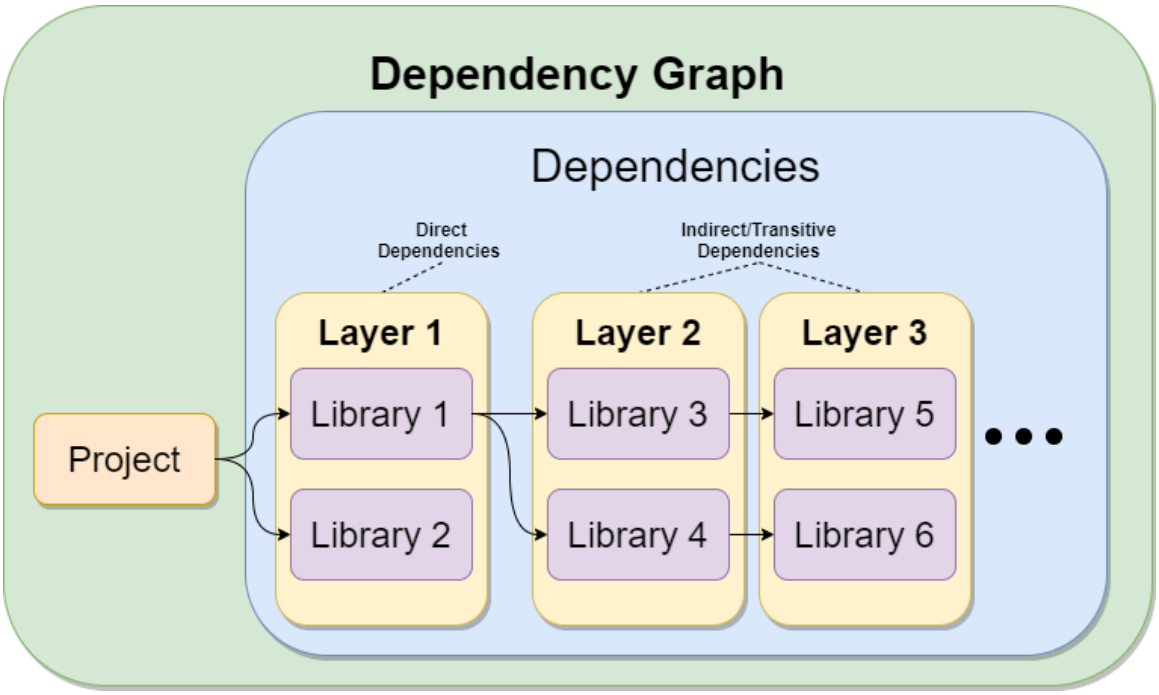

Figure 2.1: Definitions of words in relation to "Project"

2.1.1 Categorizing Dependency Information

Cox [8] clearly defines many categories that should be considered when determining a good dependency. These categories are listed below along with questions that Cox [8] asks to help guide developers in determining if a dependency fulfills that category.

- Design/Documentation - How readable is the documentation/API?
- Code Quality - Is the code well written? Basically, is the code consistent, avoids warnings, and unsafe practices?
- Testing - Does the code have tests? Are they runnable and do they pass?

- Debugging - Does the code have a lot of known bugs? Are most of them fixed? Are they fixed recently?
- Maintenance - How long has the code been actively maintained? How many people are working on the project?
- Usage/Popularity - How many people use or depend on this code?
- Security - Does the code have a history of problems?
- Licensing - Is the code properly licensed? Is the license compatible?
- Dependencies - Does the code have dependencies? Do these dependencies have flaws?

In this thesis, we propose `DependencyVis` as a software visualization approach of software dependencies that will help developers in analyzing the suitability of dependencies in their projects. Due to the wide variety of these categories, we decided to focus on the latter six and narrow them down further by putting them in a context of being a dependency. This redefines the categories into the following.

- Popularity - How well known is the dependency?
- Activeness - How active are the maintainers of the dependency
- Vulnerabilities - How many vulnerabilities are known about the dependency?
- License - What license does the dependency has?

2.1.2 Package Managers

What caused dependencies to be so easy to implement that even the smallest of libraries are considered to be worth depending on? One of the main factors is the rise

of *package managers*. Package managers are systems used to install, uninstall, and manage software packages in software projects. This simplifies package installations and updates to running a single command. Then, all a developer has to do is add a single line of code to their files in order to import the installed package to their projects and development environments.

In the `JavaScript` landscape, one of the most popular package managers is called npm. Npm is a package manager specifically for the `Node.js` runtime environment. `Node.js` is an environment that allows `JavaScript` to run outside of the standard HTML environment. Npm allows `Node.js` projects to run simple commands such as the following to install and uninstall packages.

```
npm install {package_name}
npm uninstall {package_name}
```

Where {`package_name`} is the name of the package that the developer wants to add to their project.

In order to keep track of all the dependencies for a project, npm creates and manages a `package.json` file. This `package.json` file stores all versions, script information to run the project, and dependencies. This makes the `package.json` a good source of information about both the project and its dependencies. Versions in npm are stored in the format of `x.x.x` where the first `x` represents the major number, the second `x` represents the minor number, and the last `x` represents the patch number. For example, 3.5.2 could be a version number where the major version number is 3, the minor version is 5, and the patch is 2. This distinction allows for quick identification of how likely going from one version to the next is going to break the code. For example, updating from 3.5.2 to 4.0.1 is a significant difference compared to updating from 3.0.3 to 3.5.2. In the `package.json` file, versions are stored with one of two symbols marked

before them: ^ or ~. The first symbol means that the highest minor number will be used whereas the second symbol means the highest patch number will be used.

Because npm will automatically use the highest number as specified by the symbol in the `package.json` file, installing a library with dependencies at higher versions than intended can cause unintended breaking changes. Luckily, npm projects also have the ability to specify the exact version of dependencies used. This information is stored in a `package-lock.json` file and can prevent the problem of accidentally installing a higher version number by locking the dependencies to specific versions. This separation is important because it allows the `package.json` to specify the bare minimum versions of dependencies and all important information to a project while `package-lock.json` will specify the exact version of each dependency installed during development. This means that `package.json` is good for providing an overview of the packages, while `package-lock.json` helps reconstruct the exact dependency graph that was used for the package's development. However, the `package-lock.json` file is not required to install a package, so not all repositories have it. Therefore, it was not used as a source of information for this thesis.

2.1.3 Dependency Tree or Graph?

In graph theory, a *directed tree* is defined to be a *directed acyclic graph* (DAG) with only one edge between nodes. This means that there cannot be any cycles in the graph produced in order to be considered a tree, so the denotation of a dependency tree may not be an accurate term for all the nodes and edges in `DependencyVis`.

Theoretically, it could be possible for libraries to depend on each other. This might be achieved if there are two published packages, and the first package depends on the second package, but the second package updates to depend on the first package. There

was no explicit documentation in npm that we could find addressing this potential issue. When researching this issue, most sources talk about the issue of cyclic dependencies within a project[1]. This means that one file or object within a project has a circular dependency with another file or object in the same project. One source [5] did mention the possibility of circular dependencies in packages; however, the book is talking about a different context from the npm ecosystem that `DependencyVis` covers.

The closest source to talking about npm dependencies is in King's blog post [17]. King simply uses the term *dependency tree* and describes the other problem with dependencies which involves different versions of a dependency are depended on in the same project. But because we cannot confirm or deny the existence of cycles in npm packages, we will use the term *dependency graph* in this thesis.

Even though we address these issues in this thesis, `DependencyVis` neither takes in account different versions nor the possibility of cycles in the npm ecosystem as we do not know any examples of packages in the npm ecosystem with this problem. These can be potential improvements in the future.

2.2 Software Visualization

The term *software visualization* is defined as "the art and science of generating visual representations of various aspects of software and its development process" [12]. The general goal of software visualization is to help to comprehend software systems and to improve the productivity of the software development process [12]. These visualizations can be anything from diagrams to interactive, analysis programs as long as the information being displayed helps in the development of software. Merino et al.

[1] `https://www.sciencedirect.com/topics/computer-science/cyclic-dependency`

[21] performed a mapping study in classifying 65 software visualization design papers in order to provide an in depth classification of how software visualization has been used to help the software development process. One classification Merino et al. [21] used for the software visualization papers is the task that the papers are trying to support with visualization. These tasks are categorized into the following.

- Debugging
- Maintenance
- Programming
- Reverse Engineering
- Software Process Management
- Testing

Figure 2.2 and Figure 2.3 shows two examples of visualizations classified by Merino et al. [21] that address dependencies. Figure 2.2 was classified as Maintenance as it demonstrates a radial visualization to show how dependencies evolve overtime in a project using symbols and colors to represent different statuses of dependencies. This is intended to help maintainers prioritize certain dependencies over others when it comes to updating, removing, or adding dependencies [18]. Figure 2.3 shows a node-edge visualization tool classified as Reverse Engineering as it was intended to help simplify reverse engineering tools by making them easier to understand through a visualization [9].

Based on these categories, we can see that software visualization is very useful and can help in a wide variety of tasks for software developers. This thesis hopes to

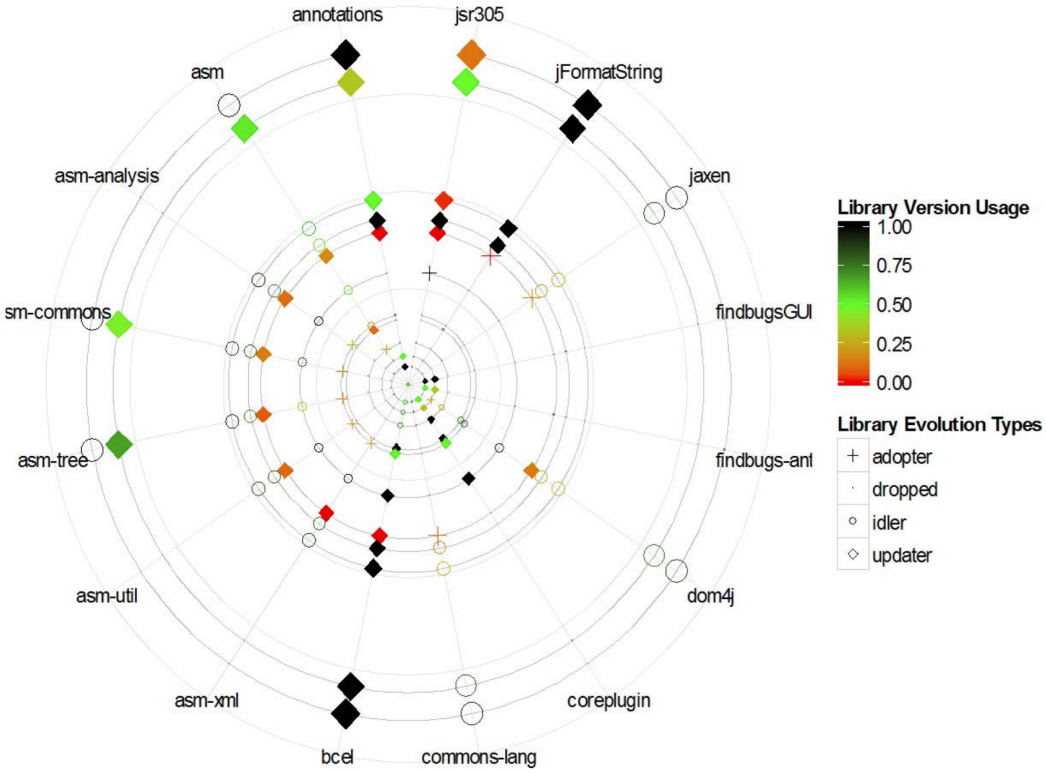

Figure 2.2: Radial Visualization of Dependency Usage [18]

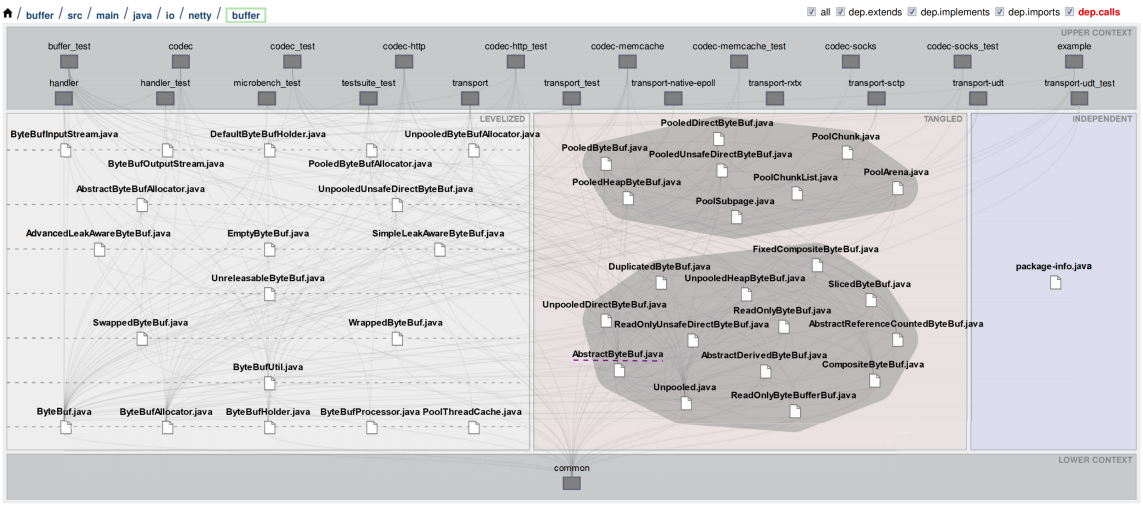

Figure 2.3: Node-edge Visualization [9]

tackle the idea of Maintenance by helping developers manage their dependencies and dependency choices.

The software visualization strategy we use for `DependencyVis` is a graph made up of nodes and edges otherwise known as a *network*. Networks can be used to portray a lot of different connections as each node can represent and object and each edge represents a connection between those objects. For `DependencyVis`, nodes represent libraries and edges show which library depends on the other. An additional advantage of using networks is that it can be used to portray many different metrics at once on top of the nodes and edges. This is achieved by varying many different aspects of the graph such as node size and node color, which are the main parts that `DependencyVis` uses.

Chapter 3

IMPLEMENTATION

In this chapter, we discuss details on how `DependencyVis` is implemented. We begin with the libraries used, decisions made in the types of projects `DependencyVis` supports, all the metrics, databases that `DependencyVis` retrieves and stores information from, and how `DependencyVis` was deployed online.

`DependencyVis` uses a lot of libraries in order to gather, access, and display all the information from NPM open source repositories. Figure 3.1 displays the general dependency architecture of `DependencyVis`. `Express.js` manages all the code in the server side while `React.js` manages all the code in the client side. `Axios.js` is a library that makes it easy to make HTTP requests. This is helpful as it allows the `DependencyVis` client to communicate with the `DependencyVis` server. The arrows in Figure 3.1 point to what libraries the client and server side are using while the dashed lines indicate a network request and response between the two nodes.

Ideally, `DependencyVis` would be able to access all npm libraries and all public GitHub repositories; however, at the moment, `DependencyVis` is only able to directly access public GitHub repositories with a `package.json` file. This covers a majority of npm open-source libraries as well as any public repositories that someone makes using the npm which would provide a `package.json`. The easiest way to verify if a GitHub repository can be accessed by `DependencyVis` is to make sure the repository has a `package.json` somewhere in it. Once that is verified, the user can put the owner name and repository name in input fields (1) and (2) of `DependencyVis` as seen in Figure 3.2.

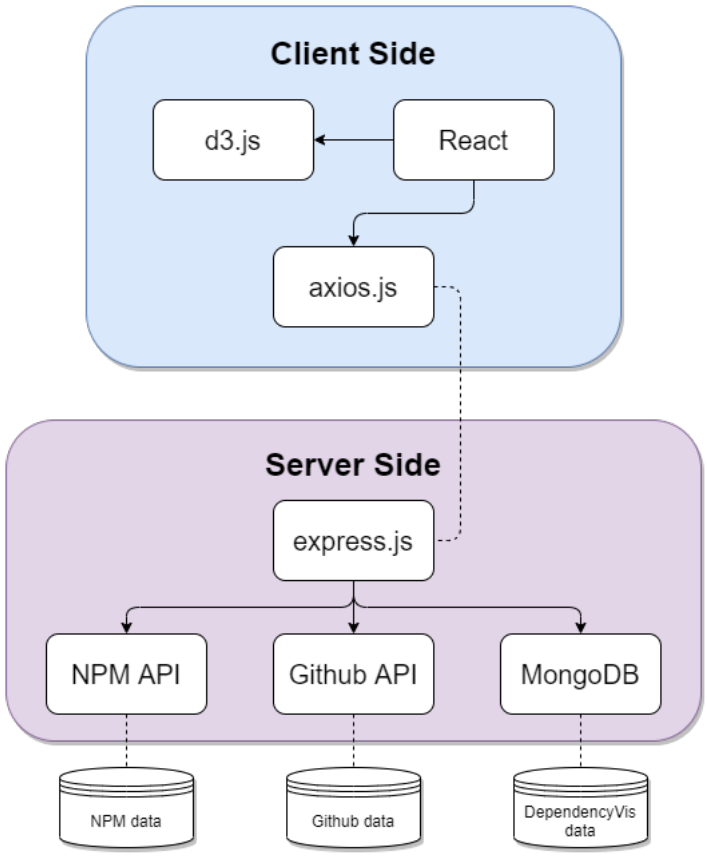

Figure 3.1: General Library Architecture

If a GitHub repository has multiple `package.json` files indicating multiple subprojects relying on npm, then entering the path of the specific folder to input field (3) as seen in Figure 3.2 allows the user to control which `package.json` file to analyze. The path of the specific folder should be entered in relation to the root directory. For example, if the `package.json` was in a folder called `server` which is inside a folder called `project1` and that `project1` folder was in the root directory of the repository, then the path of the specific folder should be entered as "project1/server". Otherwise, `DependencyVis` will find the first `package.json` based on lexicographical order of the path. For example, if a repository has two `package.json` files found with the paths "client/package.json" and "server/package.json", `DependencyVis` will choose "client/package.json" to analyze if no folder is specified on field (3).

The checkbox (4) in Figure 3.2 determines whether or not `DependencyVis` will use its own database information for the visualization. The `DependencyVis` database is updated by previous runs of `DependencyVis` which will update the database under two conditions: unchecking the checkbox or if the database does not have the repository it is searching for. When either of those conditions occur, `DependencyVis` will gather all its data from the GitHub and npm API sources and then update its own database accordingly. More detail on how `DependencyVis` stores its data can be found in 3.6.

There are a few main advantages to using the database instead of always generating the data on the spot. First, the database is significantly faster than making all the API calls and waiting for responses. This can help curb the impatience users will likely have when using this program. Second, GitHub has a limit to the amount of calls it allows per account, so using the database to offset the amount of calls required to populate the visualization helps prevent `DependencyVis` from hitting this limit. Finally, the database allows the ability to simulate the program in a specific point in time. This can potentially be used to conduct studies on the validity of these kinds of programs or studies to see how people react to a certain set of data through visualization.

After getting through the title screen, there are a number of UI elements that appear. Figure 3.3 shows `DependencyVis` after entering the `Axios.js` library into it.

3.1 Metrics

There are a number of metrics that `DependencyVis` can retrieve and display about a dependency. These metrics can be organized into the different categories mentioned in 2.1.1. As a reminder, these categories are: Popularity, Activeness, Vulnerabilities, and License.

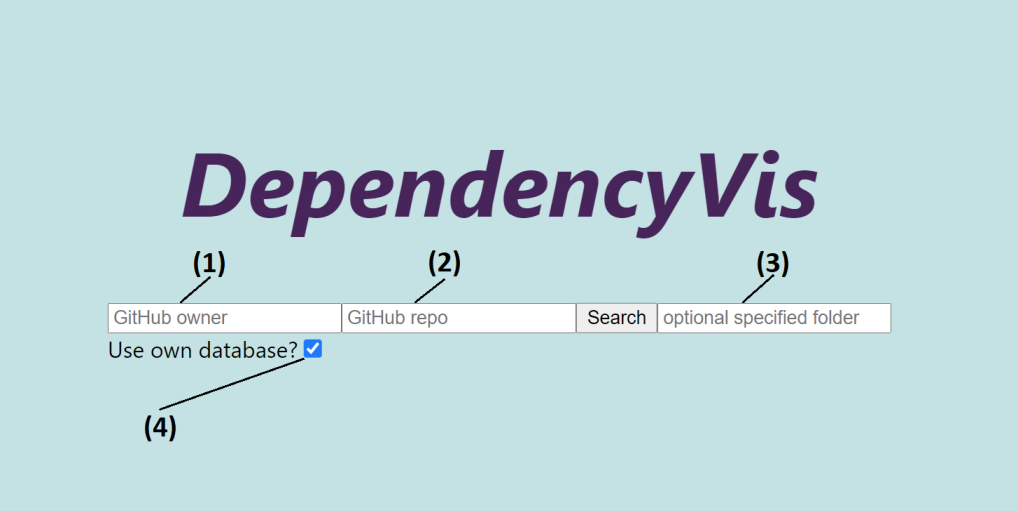

Figure 3.2: Labeled Title Screen of `DependencyVis`.
(1) Text field for entering owner name. **(2)** Text field for entering repository name. **(3)** Text field for entering the name of the subfolder. **(4)** Checkbox for whether or not to use `DependencyVis` database.

3.1.1 Popularity

When determining if a library is good or not, it is important to consider how "popular" it is. Popularity is the measure of how much attention the library has gotten, meaning popularity can be a way of measuring how many people rely on the library. So if the library fails in some way, then the failure will more likely be addressed by others and be addressed faster. A higher popularity can also mean that the developers are less likely to abandon the library and would instead be more motivated to maintain the library. Even if the developers do abandon the project, the more popular a library is, the more likely other people can revive it [2]. Not only that, more people using the library would lead to more questions and answers on how to use the library, ultimately providing better documentation and examples.

`DependencyVis` provides two simple GitHub metrics that represent how popular a library is:

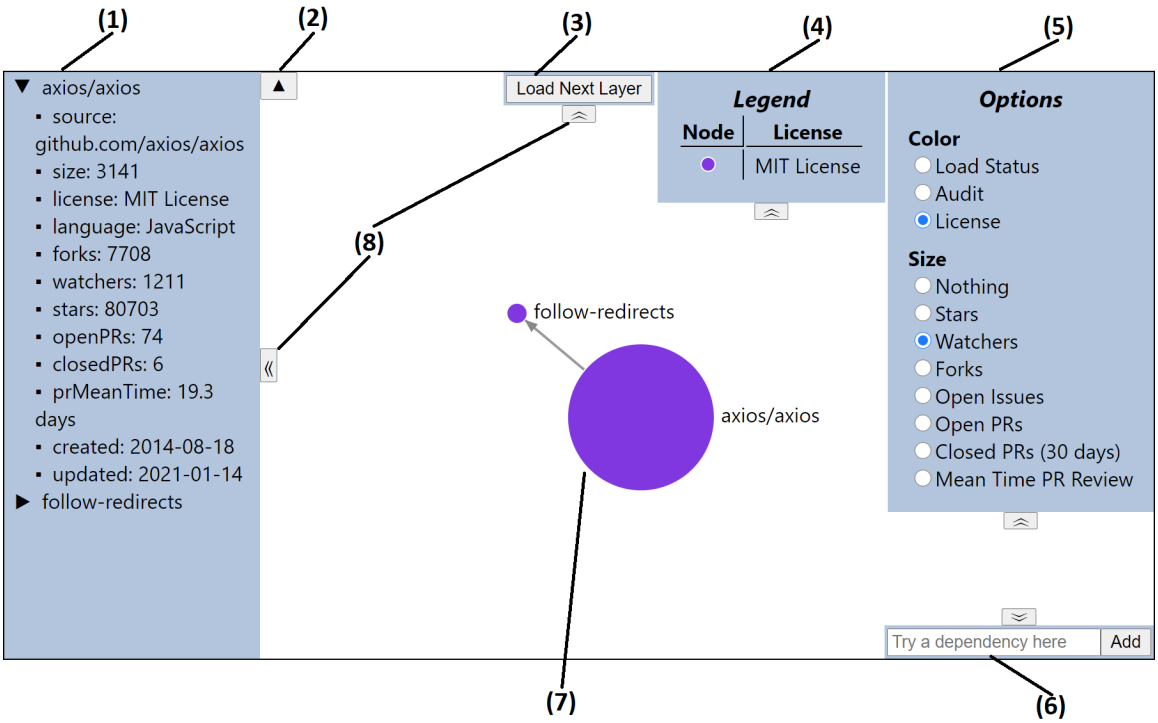

Figure 3.3: Labeled `DependencyVis` After Title Screen.
(1) Sidebar containing dropdown menus full of metric information for the corresponding node. **(2)** Button to hide all dropdown menus. **(3)** Button to load next layer of dependencies. **(4)** Legend for what the colors represent. **(5)** Options Panel for selecting different visualization options. **(6)** Text field for adding a new direct dependency by name. **(7)** Node-Edge graph. **(8)** Buttons for hiding the panels they are next to.

- Stars - the number of people who have saved the GitHub repository to their list of Starred repos.

- Watchers - the number of people who receive notifications about updates to the GitHub repository.

In order to collect these metrics, `DependencyVis` takes the owner name and repository name that the user inputs and makes request to the "repos" access point in the GitHub API as explained in Section 3.3 for the default repository information. This default information is provided in a `.json` that includes a "stargazers_count" property and a "subscribers_count" property that correspond to "stars" and "watchers"

respectively. It is important to note that the "watchers_count" property returned does not correspond to "watchers" in the GitHub repository and instead provides the same number as the "stargazers_count" property. The GitHub documentation specifies that this is the case for backwards compatibility purposes as "watchers" and "stars" used to be the same:

> "In August 2012, we changed the way watching works on GitHub. Many API client applications may be using the original "watcher" endpoints for accessing this data. You can now start using the "star" endpoints instead"

3.1.2 Activeness

Popularity is a good starting attribute to consider when deciding on a good dependency, but sometimes it helps to get a little more in depth on how maintained a library is. A library might be more likely to have active developers when it is popular, but how "active" are they really? In order to answer this question quickly, `DependencyVis` provides six metrics from GitHub that help determine how active the developers of a project are:

- Forks - the number of forks
- Open Issues - the current number of open issues
- Open PRs - the current number of open pull requests
- Closed PRs (30 days) - the number of pull requests closed within the last 30 days.
- Mean Time PR Review - the average amount of time it took for closing pull requests within the last 30 days.

The number of forks indicates how many people have copied the repository. These people usually fall under two groups: people who are trying to help maintain and update the repository and people who are saving a copy of it for personal use. This makes the number of forks arguably both a popularity metric and an activeness metric as the number of people who are saving a copy shows how popular the repository is whereas the number of people helping to maintain the repository by submitting pull requests from their forked repository are active developers. We chose to put this metric as an activeness metric because of the possibility that people who fork a repository can become active developers or are already active developers for the library.

If a repository uses GitHub to keep track of bugs, issues, and other tasks necessary to update and maintain the repository, then the repository will likely have open issues, open PRs, and closed PRs. A repository can be considered more active when they have greater numbers in all these metrics. However, having more open issues might simply amount to either more popularity or developers not actually completing the issues which would actually indicate less activeness. Open PRs will have a similar problem where having a lot may indicate a lot of people trying to help, but no actual maintainers are taking the time to approve them. That makes it important to consider the number of closed PRs as well. Due to the shear volume of closed PRs and that GitHub does not directly provide the number, `DependencyVis` provides the amount of closed PRs that closed within the last 30 days with a cap of 100 PRs. If a repository has a relatively high number of close PRs and a low number of open PRs, then it can imply the possibility of the developers updating the library very often within the last 30 days, indicating a high recent activeness. But because this might not directly correlate to active developers, `DependencyVis` provides one last activeness metric: Mean Time PR Review.

The Mean Time PR Review is essentially how many days it takes for a PR to be reviewed. In order to calculate it, `DependencyVis`'s server takes all closed PRs that are counted in the closed PR metric and requests the open and close time. Then the server subtracts the close time with the open time and sums up all the differences before dividing that sum with the number of closed PRs to get the average time in milliseconds. The client then converts this number to days in order to make it more readable. This metric probably provides the most accurate information on how active a repository is. The lower the Mean Time PR Review, the faster it took for developers of the repository to approve a PR, thus indicating more activeness.

To be more accurate about the calculations, "last 30 days" can be misleading as it is calculated by taking the current time and subtracting one from the month number. This can indicate inconsistencies from month to month due to a month having anywhere from 28-31 days. However, saying "last 30 days" has less confusion compared to "last month" which implies either starting calculations in the beginning of the current month or the way `DependencyVis` calculates it.

There are few reasons for calculating the number of closed PRs and the Mean Time PR Review with the last 30 days. First, limiting the time frame to a month can give a more solid comparison of recent activity as opposed to the entire project's lifespan. Second, GitHub has a limit of 100 PRs they can send per request. This means that there will be a lot of requests required to calculate all the PRs of a project, so the calculation is capped at the most recent 100 PRs and will not request for anymore. Thus a month was considered a good balance as it is not that likely to exceed 100 PRs within a month.

3.1.3 Vulnerabilities

As Open-Source Software becomes more and more prevalent, vulnerabilities found in dependencies or indirect dependencies are becoming more and more likely. Hejderup found in his thesis study [15] that one third of the modules in npm have vulnerable dependencies based on advisory information and half of them can be fixed through updates.

Vulnerabilities should probably be one of the biggest factors when considering a dependency. Many databases keep track of vulnerabilities found in libraries. Some of the most prominent are the Common Vulnerabilities and Exposures (CVE) list and the National Vulnerabilities Database (NVD). However, going through all the advisory data of these databases is tedious as a developer cannot simply enter a library name into either of the search engines for CVE or NVD. Doing so will result in an entire list of vulnerabilities unrelated to the library in question or nothing at all. Because of this difficulty, many companies have developed centralized ways of revealing vulnerabilities in the dependency graph. GitHub has their `Dependabot` – an AI that analyzes all repositories for vulnerabilities in out of date dependencies and then creates pull requests for the repository to update them. On the other hand, npm has introduced the tool `npm audit` that informs npm users how many vulnerabilities and how severe their `Node.js` projects are, even whether or not an update would fix them.

In order to demonstrate the possibilities of using a tool like `DependencyVis` for visualizing how many vulnerabilities and how severe they are within a repository's dependency graph, `DependencyVis` provides an Audit option that uses the `npm audit` for data.

Getting the information of `npm audit` programmatically proved to be a difficult task. There was no clear documentation on how to access the information and even if `DependencyVis` used the command line tool, `DependencyVis` would not be able to get any information other than for its own repository due to the inability to specify a specific repository for the command. Luckily, there was a hidden access point in the npm API that was found by Gorav Singal [29]. Using his method that imported `npm-registry-fetch`, `DependencyVis` is able to get `npm audit` information about one dependency at a time. This deviates from the behavior of running the command `npm audit` as the command will search the entire dependency graph whereas `DependencyVis` will only display the vulnerabilities of each dependency individually. So, as long as all outer layers are loaded, then all the vulnerabilities will be displayed.

`DependencyVis` is also given the info behind the vulnerabilities from the npm API. This means full description of each vulnerability is accessible to `DependencyVis`. Currently, `DependencyVis` does not display the vulnerability specific information anywhere. However, when the user hovers their mouse over a dependency with vulnerabilities, a tooltip will display listing out the vulnerability ID and how severe it is. This vulnerability ID corresponds to npm's own advisory database's IDs. For example, if 134 was listed in the tooltip in `DependencyVis`, then going to `https://www.npmjs.com/advisories/134` would result in all the information `DependencyVis` receives from the npm API.

3.1.4 License

One final metric included in `DependencyVis` is the license of each library. The license is probably one of the most overlooked metric as it is a complex system in itself that is difficult to consider as it does not fall under a numeric or ordinal scale, making it difficult to compare licenses visually. However, licenses are important as they

are required for an open-source library to be considered open-source and should be approved by the Open Source Initiative (OSI) [19]. Also, due to the prevalence of depending on many different libraries for software projects, it is important to consider whether or not the licenses of the dependencies are compatible, since licenses indicate how source code or binaries can be modified, reused, and redistributed [31].

Projects that have components of different licenses are called *multi-licensed* projects. Already a good portion of multi-licensed projects tend to have compatibility issues between licenses [22]. So if a user knows what licenses are compatible with each other, `DependencyVis` can help them find incompatibilities or stray licenses in their project's dependency graph.

`DependencyVis` helps users visualize their project's and dependencies' licenses by associating a color to each license. This color is determined during runtime where the first license that `DependencyVis` encounters gets assigned the first predetermined color. Currently, there is a maximum of 13 predetermined colors limiting `DependencyVis` to work with a maximum of 13 different licenses at a time. This maximum can easily be increased by adding more colors to the predetermined list, but seems sufficient at the moment. Ideally, we would autogenerate these colors as so far it is known for a project to have up to 256 licenses [22]. The only criteria for picking predetermined colors was to pick colors that decently differed from each other and was not used in the other metrics.

3.2 Simulating Adding Dependencies

So far, we have covered how `DependencyVis` makes finding information within a single project or library very easy; however, if `DependencyVis` was only able to show one library and its dependencies at a time, then comparing two libraries' numerical met-

rics that are not on the same dependency graph would be difficult. This is because `DependencyVis` represents numerical metrics as the node size. In order to accommodate different numerical scales, `DependencyVis` will adjust the numbers so that the highest number and lowest number will scale a predetermined max node size and minimum node size. Though scaling the sizes is not perfect, it is to ensure that giant nodes do not clutter the screen. Therefore, looking up one library in `DependencyVis` and seeing a small node size for the number of forks in the central node then looking up another library in a separate instance of `DependencyVis` and seeing a large node size for the number of forks in the central node does not mean that the first library has fewer nodes than the second.

So, to make the comparisons easier, `DependencyVis` supports adding a library as a dependency to the central node. This will force the node sizes to use the same scale allowing for a visual comparison between the library and the project as well as all the dependencies both the library and the project have. If the user wants to decide between two similar libraries, adding both to their project on `DependencyVis` can give a visual comparison of all the metrics supported by `DependencyVis` for both of the libraries without having to actually install them to their project. The ability to add dependencies also lets the user know what their dependency graph will look like if they add a dependency.

To implement this feature, `DependencyVis` has an input field on the bottom right of the visualization where the user can input the name of the npm package they wish to add as a dependency to the central node. Then `DependencyVis` will use this name to search for the package the same way as mentioned in 3.4. Once the search is performed, `DependencyVis` will connect a side node to the central node. If that node is grey when the Loaded option is selected, then that means that the node has dependencies that are not loaded in yet. Otherwise, the node will be light blue if the

package has no dependencies. To resolve any grey nodes, the user should press the "Load Next Layer" button for `DependencyVis` to load all the grey nodes currently in the graph. If more grey nodes appear, the user can simply press the same button again until no more appear to complete the dependency graph. Then the user can compare metrics by selecting the metrics on the Options Panel as before. Examples of comparing dependencies with and without this feature are provided in 4.3.

3.3 Accessing GitHub API

A majority of the metrics from `DependencyVis` come from the GitHub API. So accessing this API is very important. When the user enters an owner and repository name, the `DependencyVis` client will send that information to the `DependencyVis` server through the `/lookup` access point. The `DependencyVis` server will then generate the corresponding GitHub URLs and use its own GitHub authentication token to request information from the GitHub API before compiling it all into a single packaged response for the `DependencyVis` client. For each repository, the URLs generated to request information are

- `https://api.github.com/repos/{owner}/{repo}`
- `https://api.github.com/repos/{owner}/{repo}/pulls?state=open&per_page=100`
- `https://api.github.com/repos/{owner}/{repo}/pulls?state=closed&per_page=100`
- `https://api.github.com/repos/{owner}/{repo}/git/trees/{default_branch}?recursive=1`
- `https://api.github.com/repos/{owner}/{repo}/contents/{path}`

where the variables are defined as follows:

- {`owner`} - the owner as entered by the user

- {`repo`} - the repository as entered by the user
- {`default_branch`} - the name of the default branch set for the GitHub repository. This is usually either "master" or "main".
- {`path`} - the path to the `package.json` file.

The first URL is used for a majority of the metric information as it provides an overview of the repository. This request alone provides the number of stars, watchers, forks, open issues, and license. The second URL simply provides the number of open PRs with a maximum of 100. The third URL provides the number of closed PRs and enough information about each PR to calculate the Mean Time PR Review metric. The fourth URL gets the file tree of the repository. This is how `DependencyVis` finds the path to the `package.json` to be used for the fifth URL. The fifth URL returns the contents of the `package.json` which allows `DependencyVis` to know what the repository depends on which is necessary to draw the visualization and make further requests for dependency information which uses the npm API as explained in 3.4.

As an example, if the user entered "visjs" for the owner and "vis-network" for the repository, the resulting URLs generated would be

- `https://api.github.com/repos/visjs/vis-network`
- `https://api.github.com/repos/visjs/vis-network/pulls?state=open&per_page=100`
- `https://api.github.com/repos/visjs/vis-network/pulls?state=closed&per_page=100`
- `https://api.github.com/repos/visjs/vis-network/git/trees/master?recursive=1`
- `https://api.github.com/repos/visjs/vis-network/contents/package.json`

This means that for every repository, `DependencyVis` needs to make five GitHub API requests in order to generate all the information the visualization supports. Having all

these requests slows down DependencyVis and justifies the need for its own database to cache this information. Ideally, this would just be a single request, but this did not seem possible with just the Github API. In the future, it would be ideal to explore other databases to see what database can provide all the information needed in the fewest requests possible while still being reliable.

It is also important to note that all this URL building and requests to the GitHub API for DependencyVis is done through a light-weight API wrapper called github-api as it simplifies a lot of this into a few commands.

3.4 Accessing npm API

The GitHub API does almost everything DependencyVis needs on its own. However, when searching for information about the dependencies, DependencyVis only has the name and the version number of the dependency to go off of. The name does not necessarily correspond to the repository name and even if it did, DependencyVis will still be missing the name of the owner. In an older version of DependencyVis, the name of the dependency was used in a GitHub search query [1] to find the repository, but this also did not guarantee that DependencyVis would get the correct GitHub repository. Therefore, the DependencyVis client must ask the DependencyVis server to request the package.json file of the dependency from the npm API instead of using the GitHub API.

Once DependencyVis receives the package.json, DependencyVis will scan the file for any GitHub URL and extract the owner and the repository name from the URL. This allows DependencyVis to request the rest of the information on the repository from the GitHub API using the first three URLs as explained in 3.3. The last two

[1] https://docs.github.com/en/rest/reference/search

URLs are for extracting the `package.json` file which `DependencyVis` already has from the npm API.

The npm API does not really have much documentation on it as it is intended for just the `npm` command to access it. However, there are already some people that have spent the time to figure out the corresponding URLs and built libraries that will access the npm API. The library that `DependencyVis` uses is called `npm-api`. This library follows a similar style as `github-api`.

3.5 Visualizing Data

Being a visualization tool, `DependencyVis` needed to have the ability to display an interactive graph that could change overtime, so `DependencyVis` uses a visualization library called `d3.js`. This library makes it easy to display data in any graph, from bar graphs to pie graphs to the node and edge graph seen in `DependencyVis`. All that is needed to get the basic graph up is to define what a node looks like, define what an edge is, then pass `d3.js` the data to display and an HTML5 canvas to display it on.

There are two types of nodes in `DependencyVis`: the central node and side nodes. The main difference between the two is that the central node represents the repository the user entered whereas the side nodes represents all dependencies to the repository. The central node is also centered on by `d3.js` and will be the only node that has both the owner and repository name to identify it; all the side nodes will be associated with the a dependency and therefore have just the name of the dependency to identify it. Finally, because `DependencyVis` stores its nodes in a node array, the central node is guaranteed to be the first node in that array or more specifically the node at index 0.

Here is an example of the central node if the user entered “expressjs” as the owner and “express” as the repository name.

```
{
    all: {...},
    clicked: true,
    color: "blue",
    details: {...},
    id: "expressjs/express",
    isCentral: true,
    loaded: {...},
    radius: 10,
    x: 508,
    y: 351
}
```

A side node example to `express.js` would be:

```
{
    all: {...},
    audit: [...],
    clicked: true,
    color: "lightblue",
    details: {...},
    id: "qs",
    loaded: {...},
    radius: 8,
    source: "api.github.com/repos/ljharb/qs",
    x: 397,
    y: 412
}
```

For nodes, `d3.js` only cares about the following properties

- `color` - color of the node
- `id` - the text next to the node displayed as the name
- `radius` - the size of the node

- `x` - the x position of the node where the origin is on the top left of the canvas
- `y` - the y position of the node where the origin is on the top left of the canvas

The `x` and `y` properties are automatically generated by `d3.js`. The `color` and `radius` are modified by `DependencyVis` based on the display options that the user chooses. `React.js` will detect this modification and reload `d3.js`. Ideally this update would just be handled by `d3.js`, but as explained later on in this section, `d3.js` is difficult to work with when inside `React.js`.

The `id` and `index` properties are set by `DependencyVis` in the beginning and never changed afterwards. The `id` is used both as the text displayed next to the node and as an identifier for `d3.js` to differentiate between nodes. This means `d3.js` does not support multiple nodes with the same `id` and will print a warning that the behavior is undefined. If `DependencyVis` was improved to have different nodes representing different versions of a dependency, then using both a `version` property and `id` property to differentiate nodes would be better.

The `all`, `audit`, `details`, and `loaded` properties all store data about the node itself. All information that would be stored in the database will be in the `all` property. If the information was successfully retrieved from the database, then the `all` property will have an object with a `_id` property. This `_id` property corresponds to the document id that is generated in the database. The `audit` property stores an array if there are vulnerabilities found through `npm audit`. Each index in this array will correspond to a vulnerability, so the length of the array is the number of vulnerabilities found. The `details` property corresponds to the details listed in the Sidebar of `DependencyVis`. So any information put in the `details` property is listed in the Sidebar in the default format of "property-name": "value". The `loaded` property stores information about

the load status, the color, and the tooltip information for the node when the "Load Status" option is selected.

A typical edge otherwise known as a link between the previous two nodes is simply:

```
{
    source: "expressjs/express",
    target: "qs"
}
```

For edges, `d3.js` is set up to draw an arrow from the source node to the target node.

The `DependencyVis` client is run using `React.js` as a base as seen in Figure 3.1. This makes generating a dynamic, component-based user interface much easier; however, `React.js` does not work very cleanly with `d3.js`. All of the tutorials and documentation of `d3.js` run on pure HTML, so the code must be converted to run with `React.js`. The basics of this conversion are mentioned in an article by Stenius [30], but in order to implement complicated features such as tooltips into `DependencyVis`, a lot of trial and error was required.

3.6 Storing Data in a Database

`DependencyVis` works just fine gathering all the data from the GitHub API and the npm API when the user requests it; however, it is a slow process as each repository will either take five GitHub API requests and one npm API request or at best three GitHub API requests and two npm API requests. Because `DependencyVis` does not have a loading screen yet, the user will have to stare at the title screen for a few seconds before `DependencyVis` will be able to display its visualization.

By storing the information in a separate database, `DependencyVis` can remove a majority of this lag time by reducing the number of requests to just one as long as

DependencyVis has encountered the dependency before. This will also help circumvent the limited number of requests that the GitHub API allows. And as mentioned in the beginning of this chapter, the database can allow studies that use DependencyVis to simulate and control the data by modifying the database.

For example, if a researcher wanted to study people's reactions to certain dependencies and wanted to make sure that each participant saw the exact same data, then the researcher just needs to uncheck the checkbox in the title screen, load up all the dependencies in question in order to update the database, then ensure that all participants use the database. As long as the database was not changed during the study, then all participants will see the same data as the researcher saw when the researcher first loaded up all the dependencies. Ideally in the future, DependencyVis will have a client for researchers and a client for participants in order to support such a study.

Currently, DependencyVis uses MongoDB for its personal cache database. MongoDB is a document-based database that stores and returns .json files. Because objects in JavaScript are inherently represented by the JSON (as JSON stands for JavaScript Object Notation), using MongoDB in DependencyVis is simple as it does not require any object conversions to read the data. In order to access the database, DependencyVis has been given a username and password to access and modify the database using the mongodb library. This library provides the MongoClient object that has functions for connecting to the database when given the proper credentials. DependencyVis can then insert entire JavaScript objects to update or add more documents to the database.

3.7 Deploying Online

A tool like DependencyVis would need to be convenient for a software developer to use. Because DependencyVis focuses on public GitHub repositories, it would be ideal to make DependencyVis a GitHub add-on. However, due to how limited GitHub add-ons are, we had to go with the next best method which is to simply get DependencyVis online. This makes DependencyVis at the very least accessible online.

In order to get DependencyVis online, we chose a free cloud server provider called heroku. Heroku seamlessly attaches to git projects by creating separate remote for deployment. This means that to update heroku, all it took was choosing the heroku remote when pushing to the default branch. One caveat to using heroku is that the repository must have a package.json in the root directory. This required a reformat of the DependencyVis directories from splitting the server and client into their own folders to having the server be pulled out into the root directory, which was not ideal for owner. Also, because React.js and express.js deploy differently, there are quite a few extra steps that was needed to make express.js output the React.js when the heroku site was accessed. Following Ceddia's directions in a blog post [4], we added modifications to both the server's and the client's package.json for DependencyVis.

The other problem with deploying DependencyVis is that all the credentials for accessing the GitHub API and the MongoDB database was stored in a .env file that did not get pushed to GitHub as it would make such credentials publicly available, which would not be desirable for security reasons. Normally, this file would simply be read in by the dotenv library, but as the file did not exist on the heroku server, the dotenv library would not be able to get the credentials necessary for DependencyVis to function. Luckily, heroku supports this in their settings. By going into the heroku server's settings on their web interface and adding all variables that was in the .env

file to the "Config Vars" setting, `DependencyVis` was able to be deployed with all the credentials necessary to run.

Chapter 4

USE CASES

This chapter describes use cases to demonstrate how `DependencyVis` can be used for many different scenarios. Figure 4.1 shows a decision flowchart to provide a basis of how `DependencyVis` can be used to compare dependencies. We begin with general use cases that go over the usage of `DependencyVis`, then we go over use cases for finding metrics, and end with use cases that compare these metrics. Please note that all of these use cases should have been written in the beginning of February 2021.

4.1 General Uses

1. **Visualizing Dependencies:** Suppose a user was considering the library `Axios.js` to manage their promises for accessing other APIs and wanted to know what additional dependencies `Axios.js` comes with. By entering into `DependencyVis` the owner name (`axios`) and the library repository name (`axios`), the user will immediately see that `Axios.js` has one dependency: `follow-redirects`. The `DependencyVis` output is shown in Figure 4.2.

2. **Adding a Dependency:** Suppose a user happened to be working on a project called `axios` and wanted to add a npm library called `d3.js` to their project to see how it would affect their dependency graph. By entering into `DependencyVis` the owner and repository name of their project and then entering into the bottom right input field "d3", the user will see `DependencyVis` add a new `d3` node to axios as seen in Figure 4.3. However, the node is grey which means that none of the dependencies for that node are loaded. So, the user can press the

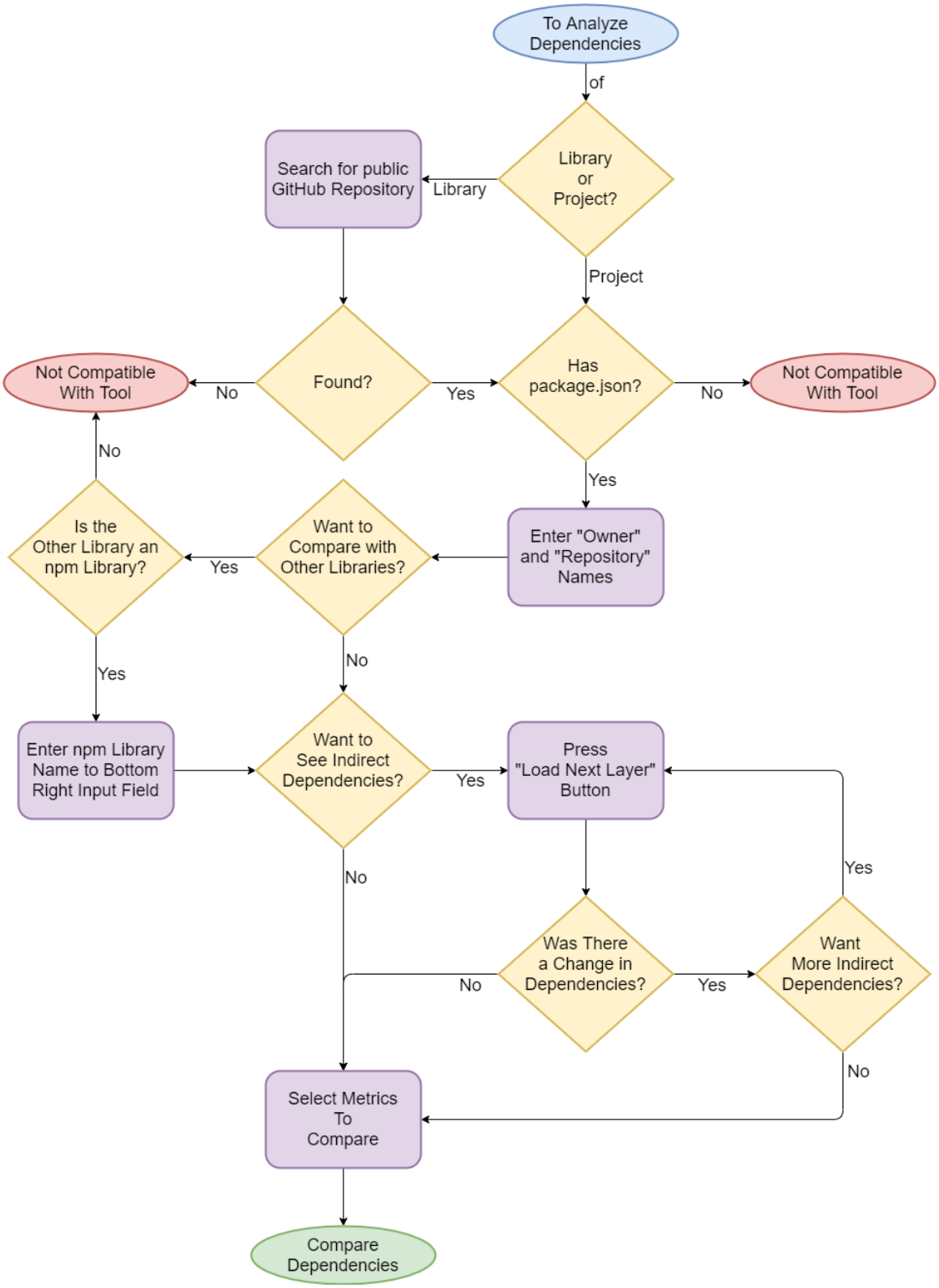

Figure 4.1: Flowchart for using `DependencyVis`

"Load Next Layer" button until all nodes are lightblue which means all their dependencies are loaded, as seen in Figure 4.4. The user can now see the entire dependency graph for after `d3.js` is added.

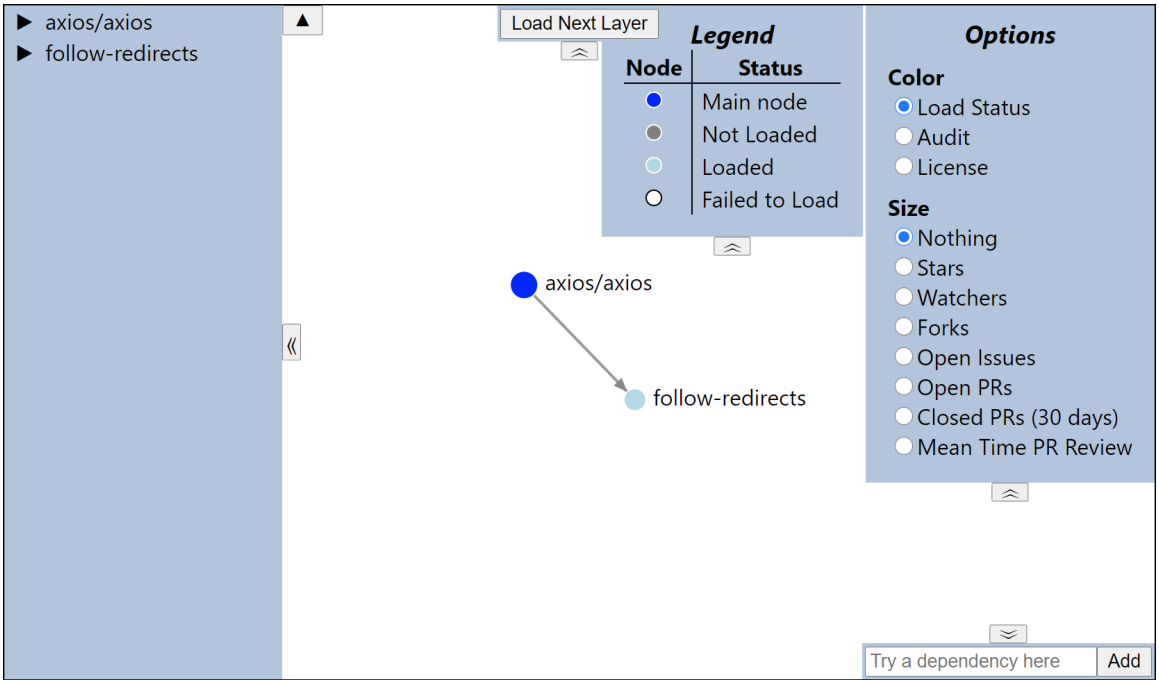

Figure 4.2: Use Case 1: `DependencyVis` Output

4.2 Finding Specific Metrics

3. **Popularity:** Suppose that a user wanted to see how "popular" `Axios.js` is. By clicking on "axios/axios" on the Sidebar on the left side of the application, the user can see that `Axios.js` has over 1200 watchers and over 80,000 stars. By clicking the Stars option or the Watchers option, the user can see that `Axios.js` has far more watchers and stars than its dependency `follow-redirects` as a reference point. Figure 4.5 shows how `Axios.js` appears on `DependencyVis` when the Stars option is selected as well as the metrics shown on the Sidebar and Figure 3.3 shows node sizes for when the Watchers option is selected.

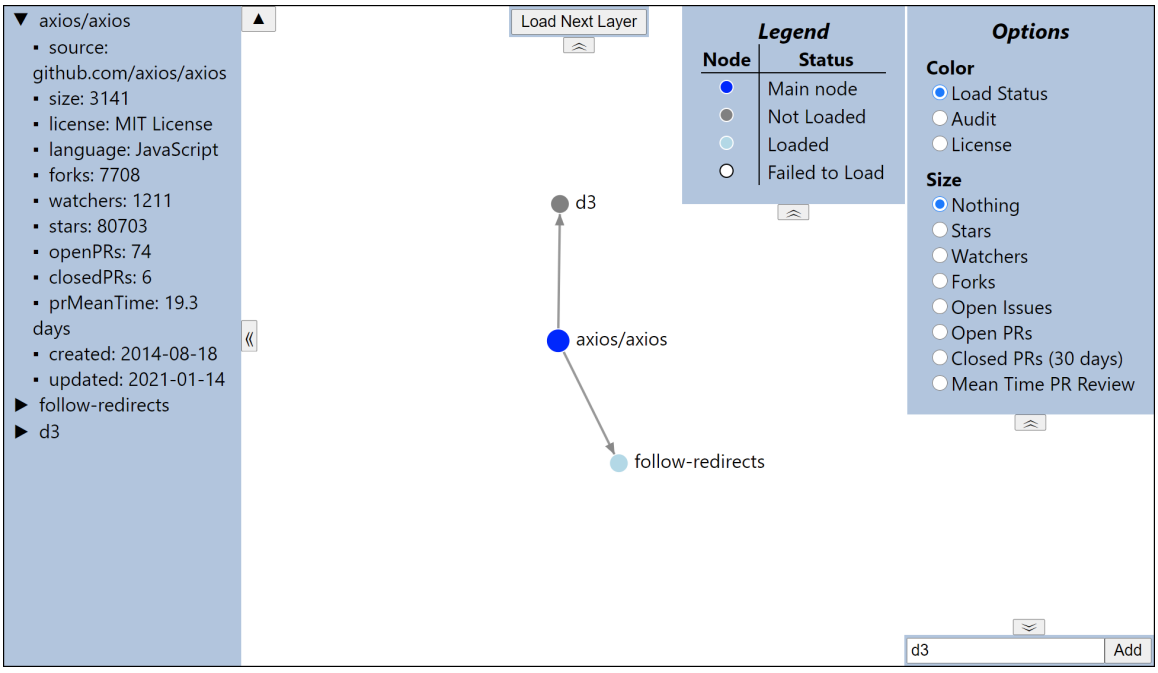

Figure 4.3: Use Case 2: Adding a Library

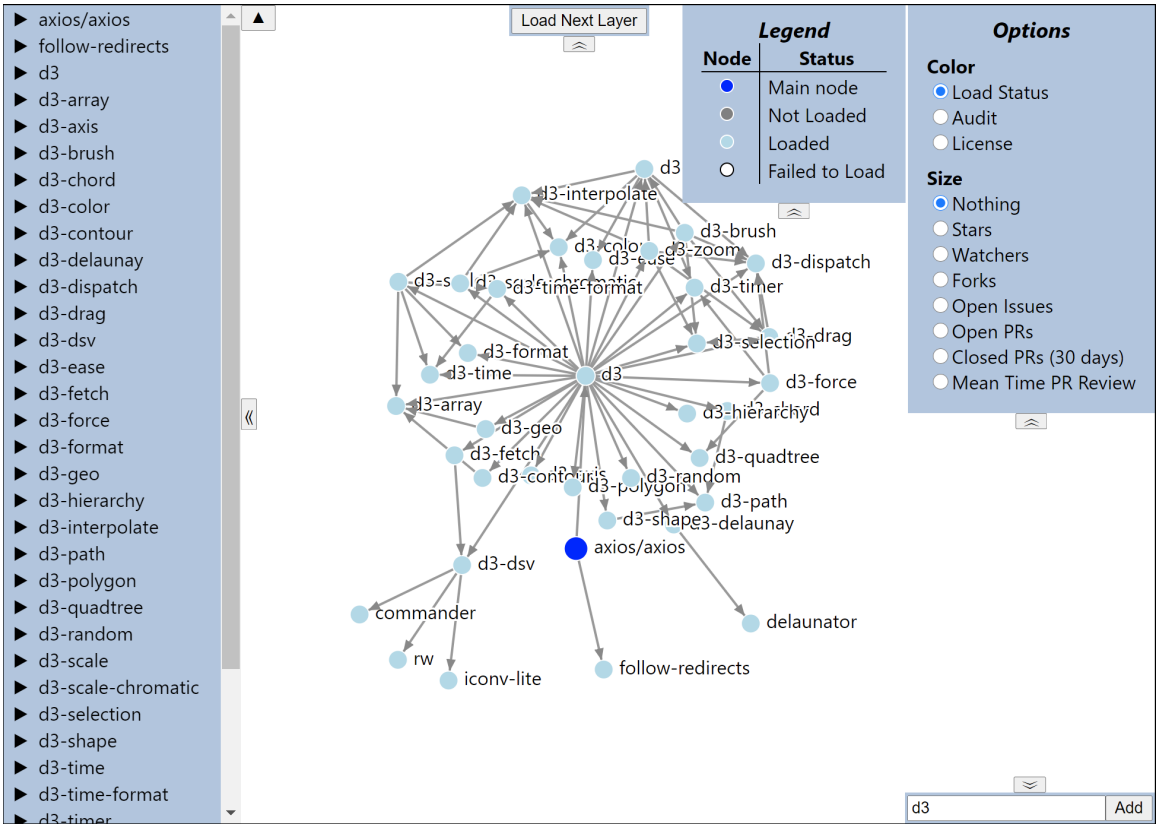

Figure 4.4: Use Case 2: Full Dependency Graph

4. **Activeness:** Suppose that a user wanted to see how "active" the maintainers of `Axios.js` are. By clicking on "axios/axios" on the sidebar on the left side of the application, the user can see as shown by Figure 4.5 that `Axios.js` has over 7000 forks, over 70 open PRs, and 6 closed PRs in the past month which took an average of 19.3 days to close from when they were first created. The user can also see that the project was last updated on 2021-01-14 (Jan 14, 2021). So `Axios.js` has still been active in the past month.

5. **Visualizing Vulnerabilities:** Suppose that a user wanted to see how vulnerable both `Axios.js` and its dependency (`follow-redirect`) are. By clicking on the Audit option under Color in the Options panel, the user will see that both the nodes are green as shown by Figure 4.5 which means that `npm audit` has no vulnerability cases for the most recent version of `Axios.js` and the versions of `follow-redirects` that `Axios.js` relies on.

6. **Vulnerability Details:** Suppose a user wanted to see how vulnerable the dependency graph of `express.js` is and wanted to see the details of any vulnerabilities the user finds. By entering "expressjs" and "express" for the owner and repository name into `DependencyVis`, the user can press "Load Next Layer" button three times to see the full dependency graph. Then, by selecting the Audit option, the user can see that there are some dependencies that have vulnerabilities. By hovering over `serve-static`, the user would see a tooltip appear with a vulnerability ID of 35 and the severity of "low" associated with it as seen in Figure 4.6. Hovering over the Audit option will provide a tooltip that describes a URL based on the ID for more information on the vulnerability: `https://www.npmjs.com/advisories/<id>`. The user can then fill in the URL to get: `https://www.npmjs.com/advisories/35`. Going to this URL

will provide all the information known about the vulnerability with ID 35. The user can then repeat this process for all vulnerabilities in the dependency graph.

7. **License:** Suppose that a user wanted to see what license `Axios.js` has in order to determine if the user can use it in their project. By clicking on the License option the right side of the application, the user can see that `Axios.js` has the color associated with an MIT License as described by the `Legend` as portrayed by Figure 3.3.

8. **All Licenses:** Suppose that a user decides to add `d3.js` to their project and wants to know all the licenses in the dependency graph of `d3.js` to check if they are compatible with the MIT License. The user can enter "d3" into both the owner and repository box, click the "Load Next Layer" button until all nodes are loaded, and then click the License option to see all licenses in the dependency graph of `d3.js`. As seen in Figure 4.7, the user can see four licenses through the program: BSD 3-Clause "New" or "Revised" License, ISC License, MIT License, and Other. The user will just need to check if the BSD 3-Clause "New" or "Revised" License and ISC License are compatible with MIT License. To be more thorough, the user can see that `rw` and a few `d3` nodes have "Other" as their license. The user may click on the `rw` node to open up more info on the sidebar and go to the url indicated in the "source" to quickly get to the github page of the library and manually figure out the license from there.

4.3 Comparing Dependencies

Here we provide two use cases. The first one compares libraries on separate instances of `DependencyVis` and the second one provides a method of comparing libraries in

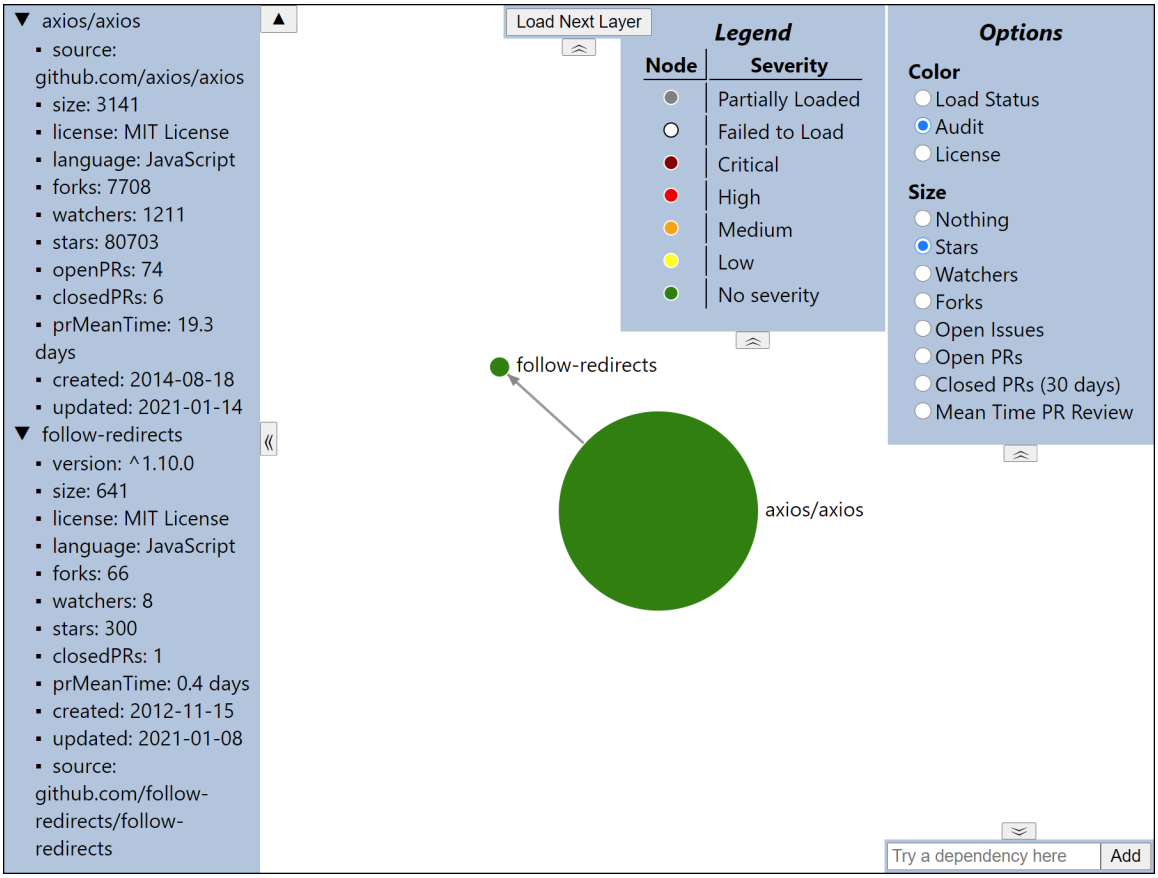

Figure 4.5: Use Cases 3, 4, 5: `Axios.js` Dependency Graph with Audit and Stars

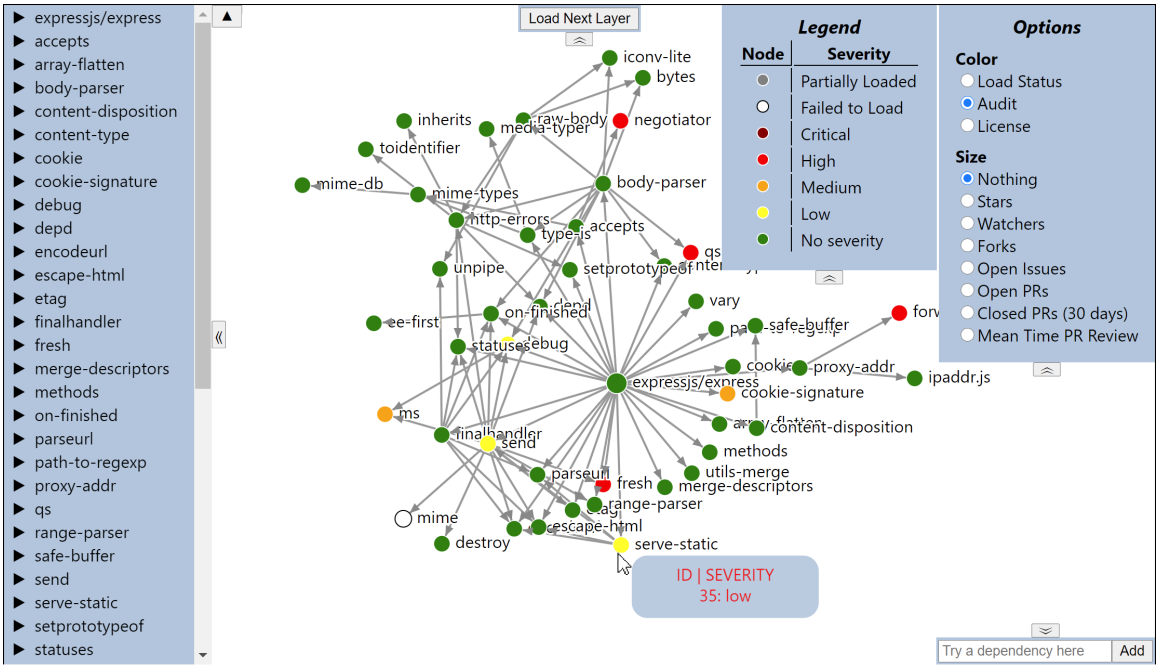

Figure 4.6: Use Case 6: Hovering over `serve-static`

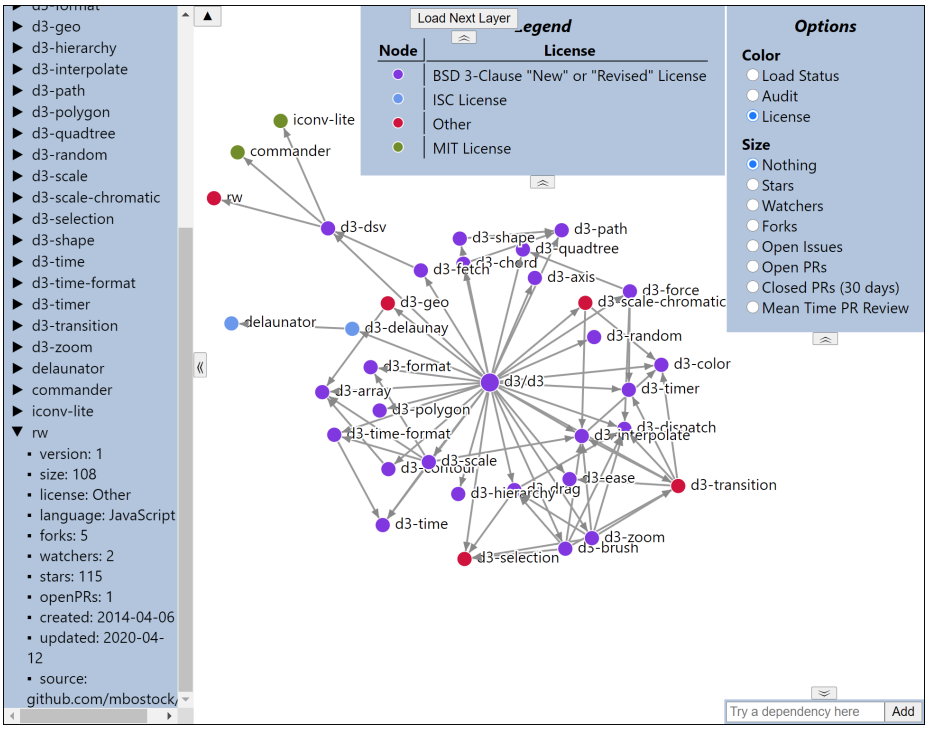

Figure 4.7: Use Case 8: Selecting the License Option for `d3.js`

the same instance. This provides an example for the benefits of being able to add dependencies as mentioned in 3.2.

9. **Comparing Libraries Independently:** Suppose a user wanted to use a visualization library to manage how their graphs are drawn for their `Node.js` project. They found two visualization libraries: `d3.js` and `vis.js`. In order to decide between the two, the user wants to determine how "attractive" the libraries are compared to each other. "Attractiveness" will be split into three categories: "how popular", "how active", and "how secure".

   To begin comparing, the user can run two instances of `DependencyVis`, then enter "d3" into both owner and repo boxes for the first instance and "visjs" into the owner box and "vis-network" into the repo box for the second instance. The user can then look at the sidebar and compare the amount of `forks` (`d3`: 22449, `vis.js`: 169), `watchers` (`d3.js`: 3967, `vis.js`: 23), and `stars` (`d3.js`: 95278, `vis.js`: 1223) to see that `d3.js` appears to be more "`popular`" based on those 3 metrics.

   To compare "how active" each library is, the user can look at the "date last updated" (`d3.js`: 2021-01-14, `vis.js`: 2021-02-04), "date created" (`d3.js`: 2010-09-27, `vis.js`: 2019-07-16), and "pr mean time" (`d3.js`: not listed, `vis.js`: 0.3 days) to see that `vis.js` was updated more recently by a single month, `d3.js` is much older by 9 years, `d3.js` did not have any pull requests closed within the last 30 month.

   To compare "how secure" they are, the user can look at the visualization and select the Audit option to see that they do not have any `npm audit` vulnerabilities. Then the user can see through the graph that `vis.js` does not have any dependencies, but `d3.js` has a lot of dependencies of mostly `d3` sub libraries.

When clicking the load next layer button, the user can see that `d3.js` indirectly depends on `delaunator`, `commander`, `iconv-lite`, and `rw`.

In summary the user will have the basic idea that `d3.js` is more popular and older compared with `vis.js` which has fewer dependencies and more recent pull requests which might indicate more activeness. The user can then decide which aspects of the dependencies is more valuable.

10. **Comparing Libraries Together:** Suppose a user wanted to analyze `d3.js` and `vis.js` in the previous use case, but without having to directly compare the numbers `DependencyVis` provides.

    To compare `d3.js` and `vis.js` visually, the user can run `DependencyVis` and enter "visjs" into the owner box and "vis-network" into the repo box. Then the user can add `d3.js` by entering "d3" into the bottom right input field. By clicking on the Forks, Watchers, and Stars options, the user can see that `d3.js` appears larger than `vis.js` for all three metrics making `d3.js` appear more popular. Figure 4.10 shows how `DependencyVis` appears when the user clicks on the License and Forks options.

    In this case, the user cannot compare "how active" each library is visually because `d3.js` does not have any recently closed PRs and thus no Mean Time PR Review to calculate. To indicate this, these two metrics are missing from `d3.js` in the sidebar of `DependencyVis`. Thus the user has to compare them the same way as in Use Case 9. Comparing "how secure" the two libraries are, will also be the same process as in Use Case 9 as long as the user clicks on the "Load Next Layer" button until no more nodes loads.

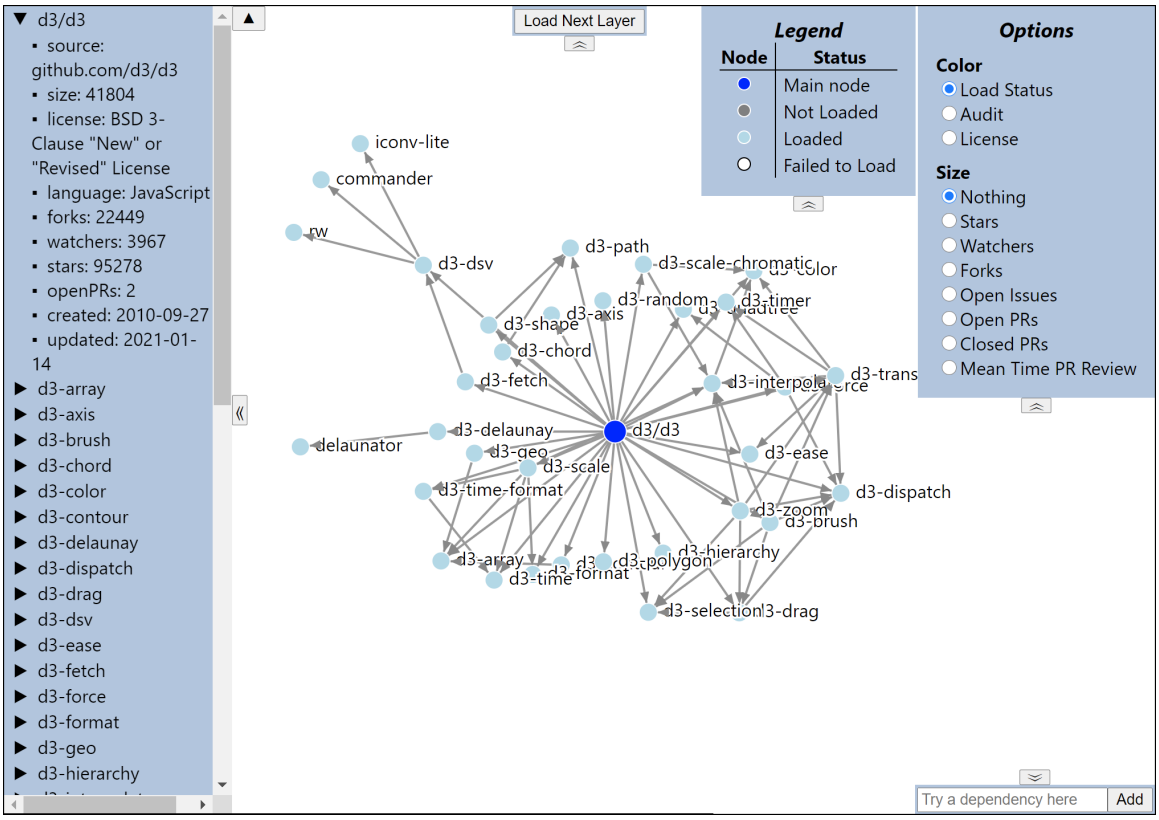

Figure 4.8: Use Case 9: `d3.js` Dependency Graph

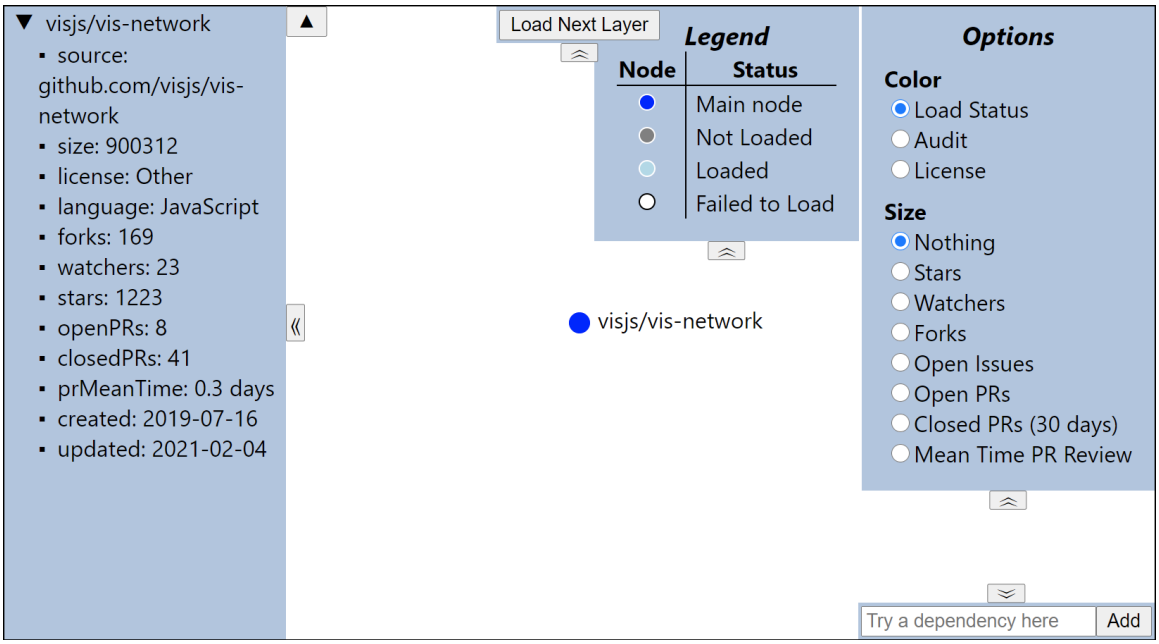

Figure 4.9: Use Case 9: `vis.js` Dependency Graph

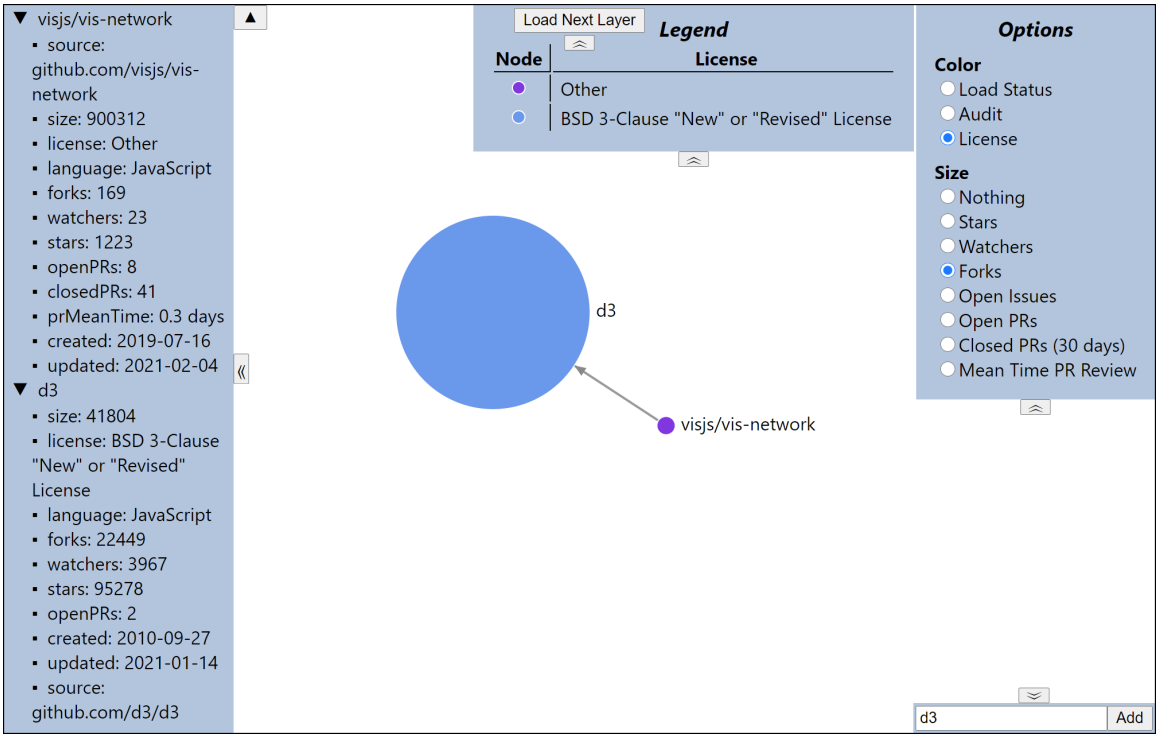

Figure 4.10: Use Case 10: Comparing `vis.js` and `d3.js` by Forks and License

Chapter 5

RELATED WORK

There have not been a lot of works that combine the idea of visualizing software and analyzing dependencies. However, the separate ideas of visualizing data and analyzing dependencies are not new. For analyzing, there have been many papers researching what makes a good dependency and have gone from determining what metrics about each dependency is important to look at to building actual programs that would do the analysis automatically for developers. Some papers investigate answers to a single question such as "why do developers use trivial packages?" [1]. Other papers provide an overview of the situation as well as their suggestions on what to look for in dependencies [8]. In terms of visualization, papers have ranged from building graphs for function calls within a project [3] to building 3D Visualizations in virtual reality environments [20].

5.1 Determining a Good Dependency

5.1.1 Addressing Vulnerabilities

One of the biggest reasons dependencies are a big deal is the vulnerabilities they can introduce. Vulnerabilities are constantly found and usually patched very often so a developer would have to keep making sure that they are updating all their dependencies in a timely manner. Vulnerabilities are even worse when left undiscovered or not patched. Pashchenko et al. [24] found that 81% of the vulnerable dependencies they analyzed can fix their vulnerability by simply updating their dependencies. Vulnera-

bilities are even worse when left undiscovered or not patched. Decan et al. [11] found that many vulnerabilities tend to take a long time to discover and another long time to fix. Decan et al. [11] also found that a large percentage of packages have not even updated or are improperly using their dependencies thus keeping many found and already fixed vulnerabilities active.

One paper went another route and investigated the idea of trivial packages [1]. A trivial package is essentially a package that the developer could easily code themselves. These trivial packages are generally considered bad to use as they introduce more factors for the developer to consider such as maintaining the package and increased vulnerability risks. The paper found that more than 11% of the trivial packages studied had at least 20 dependencies, greatly increasing the vulnerability risks. Given the extra overhead of adding trivial packages on top of the increased possibility of vulnerabilities, it is surprising that the paper found that over 50% of the developers they surveyed considered these packages as beneficial and safe.

5.1.2 Addressing Maintenance

Some papers have taken the approach of studying the concept of truck factors [13][2]. The idea of the *truck factor* essentially came from the question of how many people would need to get hit by a truck in order for a project to no longer be developed and updated. So a project's truck factor helps determine how stable the project is. There are many algorithms concerning Truck Factor. Ferreira et al. [13] compares each algorithm in order to determine the best ones for the most scenarios. Avelino et al. [2] takes it a step farther by using those algorithms to determine when *Truck Factor developers detachment* (TFDD) occurs which essentially is when Truck Factor hits zero. The paper found that 16% of the studied projects were abandoned and

41% of those were revived by new developers. This is a good metric to look at when trying to figure out how well-maintained a project is.

Coelho et al. [6] took a more general approach by developing a machine learning solution in order to tackle the problem of determining how maintained a Github repository is. They used a lot of metrics including number of forks, number of commits, number of issues, number of pull requests, number of new and distinct contributors, and project owner's history in projects. The accuracy of this machine learning solution was evaluated based on surveys with key project developers, developers in general, and the descriptions of the README files in order to determine whether or not a project was actually still maintained, getting new features, or deprecated. As their evaluations show that their solution has a decent accuracy, this machine learning solution has been implemented as a Google Chrome extension called `isMaintained` for the public to use.

5.2 Other Software Visualizations

Plenty of work has been done using many different kinds of visualizations to portray software. Merino et al. [21] performed a mapping study in classifying 65 software visualization design papers in order to provide an in depth classification of how software visualization has been used to help the software development process. Problem domain classifications include debugging, history, performance, reverse engineering, and dependencies.

There was a period of time where visualizing coupling between classes were quite popular through visualizations such as the function call graphs. Function call graphs showed the connections between classes and functions providing a visualization that lets developers watch how their code flow in order to find where problems and bugs

lie in their code. This visualization also helps developers organize code and make sure that classes and are as decoupled as possible. Bohnet & Dollner [3] built a tool that will analyze C/C++ code in order to display the function call graph that helps in understanding how features are implemented into the system and identifying where new features can be implemented in the structure.

Hejderup et al. [16] takes this idea a step further by proposing a function call graph that would traverse dependencies to display how code flows through different dependencies and what functions are used the most in dependencies. This would essentially allow developers to see how their code traverses the libraries and how calling one function in a library might go much deeper in the dependency graph than is expected.

One paper even took dependency visualization to the next level by using AR and VR devices to display the metrics [20]. This paper wanted to address the problem of many 3D visualizations only displayed on a 2D screen. They found that AR and VR technologies do help in easing 3D visualization problems; however, selecting and text readability remain an issue for this domain.

Back in 2.2, Figure 2.2 shows a visualization tool fairly similar to `DependencyVis`. Kula et al. [18] intended for this radial visualization to help novice maintainers of a library learn the history of how the library's dependencies have been maintained. For example, a novice maintainer can generally see what dependencies have been consistently updated and which dependencies should be kept at older versions.

`DependencyVis` defers from this radial design in that it provides more specific information about each and every dependency in order to judge many different aspects ranging from popularity to vulnerabilities. This variety also sets `DependencyVis` apart from many of the other works mentioned in this chapter as `DependencyVis` can potentially provide one location to find everything a developer or maintainer could

want to know about a project or library and the entire corresponding dependency graph.

Chapter 6

CONCLUSION

Thanks to package managers, the ease of software reuse has made software developers today give little to no thought about the libraries they depend on. Libraries are generally trusted for their benefits of saving time and having groups of people manage and critique the code in the library resulting in better code than what a developer can write by themselves. However, this inherent trust of libraries can make developers fail to consider the downsides of depending on essentially strangers to manage code the developer is using, especially considering that one of the main points of software reuse is to save time, yet developers have to spend time to research if a dependency should be trusted?

The objective of this thesis is to establish an interactive software visualization tool called `DependencyVis` for gathering and displaying information about a software project and its dependencies in order to save time spent on researching dependencies while alleviating the issue of developers not taking into consideration the problems and vulnerabilities that come with each and every library their project depends on. `DependencyVis` gathers and displays information from the npm ecosystem by following each dependencies' `package.json` file in order to produce the dependency graph of a library.

There are many use cases for `DependencyVis` and some of them are described in Chapter 4. A user can use `DependencyVis` to get the basic information of a library such as number of forks, pull requests, and `npm audit` vulnerabilities on top of the dependency graph. A user can have `DependencyVis` analyze their own project in

order to see all their project's dependency information. This can help if the user wishes to find potential problems in the dependency graph such as license conflicts or possible vulnerability points as well as what dependencies are easy to be removed or replaced due to indirect dependencies. A user can also simulate adding dependencies to their project in `DependencyVis` in order to visually compare dependencies between each other.

6.1 Implications

`DependencyVis` is a good demonstration of how it is possible to gather a lot of information related to dependencies and display them in a graph to help developers quickly decide how good a dependency is without having to lookup all the details themselves in separate screens and through different services and tools. The visualization can also help developers to compare and contrast libraries by comparing sizes of the nodes in different metrics. Our tool or the approach behind it can also be incorporated in online services such as GitHub. GitHub itself has attempted to provide dependency information to repositories hosted in the platform[1]. However, it lacks visualization features and additional information that we gather and provide with `DependencyVis`, such as npm advisory data. Also, software teams and software organizations could use our approach to help their developers design and implement software systems with dependency analysis in mind and reduce risks in dependencies incorporation. Finally, we believe researchers could benefit from `DependencyVis` by using this tool to conduct empirical assessments about dependency analysis in software development. The tool concept can be extended by adding new metrics or even being adapted to analyze private GitHub repositories.

[1] `https://docs.github.com/en/github/visualizing-repository-data-with-graphs/about-the-dependency-graph`

6.2 Future Work

This thesis work is the groundwork for developing a tool to help developers make more informed decisions about dependencies. The following are more improvements to `DependencyVis` and this thesis.

To start, `DependencyVis` has a limited scope with the requirement of every library or project having a public GitHub repository and a `package.json` file in the repository. Ideally, `DependencyVis` will be able to read the `package-lock.json` file if the repository has it and allow users to choose between the two files. `DependencyVis` is also limited to the npm ecosystem. In order to become more flexible, `DependencyVis` should support more package managers such as pip, yarn, and gems. This would also move `DependencyVis` away from analyzing purely the JavaScript language as well, further increasing the scope of what `DependencyVis` can analyze. This will ultimately let all developers be able to analyze their dependencies regardless of their programming language or package manager of choice.

In terms of user interface, more consideration should be given to the colors of the nodes in `DependencyVis`. People with different levels of blindness may not be able to tell the difference between some colors —namely red and green. This might prove a problem if `DependencyVis` is to be as inclusive as possible to all developers.

6.2.1 More Metrics

`DependencyVis` implements a lot of metrics but there are more to be covered. The number of forks, issues, and PRs, are all numbers that may help developers to determine if a dependency is desirable or not. Still, `DependencyVis` can benefit from the addition of many more metrics. For example, Cox [8] mentions how develop-

ers should consider whether or not a dependency is tested. In order to cover this, `DependencyVis` could find and show whether or not there is a Continuous Integration (CI) workflow in the dependency. More metrics related to the CI can be added as well such as the Mean Time to Resolution (average time for the CI builds of a library to become green once it has a build failure) and the Mean Time Between Failure (average time for the CI builds of a library to become red again after the build has been fixed). On top of this, `DependencyVis` could also display how long it takes for the CI workflow to execute. According to Powell Stahnke [26] from CircleCI, the ideal Mean Time to Resolution should be under an hour and the CI workflow should execute somewhere under 5-10 minutes.

Also, `DependencyVis` currently displays `npm audit` information as a vulnerabilities metric. However, there are a lot of advisory databases with this information updated by different organizations and companies such as the Common Vulnerabilities and Exposures (CVE)[2] database managed by the MITRE corporation with the intentions of a community effort, the National Vulnerabilities Database (NVD)[3] managed by the government organization called National Institute of Standards and Technology (NIST), and private databases like Snyk.io[4]. All of these would provide more in-depth information for developers on the vulnerability status of their dependencies. On top of adding more databases, `DependencyVis` should show vulnerabilities of specific versions of dependencies. Currently, `DependencyVis` shows all the vulnerabilities found for all versions of a dependency, so being specific about whether or not the vulnerability affects the current version of the library that the user's project depends on would improve the ability for users to know where security problems might occur.

[2] `https://cve.mitre.org/index.html`

[3] `https://nvd.nist.gov/`

[4] `https://snyk.io/`

As mentioned in 5.1.2, Ferreira et al. [13] and Avelino et al. [2] brought up the idea of truck factor and truck factor developers detachment (TFDD) and how to calculate them. These are useful metrics to include in `DependencyVis` for determining the number of key maintainers in a library. By calculating these metrics for software developers, there would be no need to research a library's maintainers and figure out how much has each maintainer contributed to the library recently, saving a lot of time in figuring out a valuable metric.

Not only are new metrics a good addition to `DependencyVis`, but the limitation of 30 days for the Closed PRs metric and Mean Time PR Review metric should be adjustable. Libraries could be stable or just happen to not be updated recently, so being able to adjust the 30 day limitation would give users more flexibility on how they want to evaluate the Closed PRs metric and Mean Time PR Review metric.

6.2.2 Potential Case Studies

Finally, a case study to show the usefulness of `DependencyVis` would be a great improvement to this work. In order to provide empirical evidence on how and to what extent `DependencyVis` can be useful in the real world, it would be helpful to study how real developers use `DependencyVis`. There are many kinds of case studies that can be performed for `DependencyVis`, but they would generally answer overarching questions such as

- How helpful is the visualization approach of `DependencyVis`?
- How well do `DependencyVis` and its metrics help a developer determine an acceptable dependency?

These overarching questions can be broken down into further into research questions such as:

- Does showing vulnerability levels help developers choose acceptable dependencies?
- Does showing vulnerability levels help developers maintain their dependencies?
- Does showing popularity metrics help developers choose acceptable dependencies?
- Does showing activeness metrics help developers choose acceptable dependencies?
- Do developers prefer a visualization program over searching for the information themselves?
- Are developers from large companies more likely to prefer a visualization program?

Here, *acceptable dependencies* can be defined by either participants of the case study (based on their perceptions of whether or not the metrics help) or in a separate study to define a boundary between acceptable and not acceptable dependencies.

One method to answer the questions above would be to have two groups of software engineers. These groups might be easier to find in a software engineering classroom, but can also come from contacting developers of libraries and projects directly. Both groups will be given the same list of objectives. These objectives would provide the name of some dependencies and ask the participants to find out metrics such as vulnerabilities or number of forks. One group will be given `DependencyVis` whereas the other group will either have to research on their own or be given links to databases to

simulate developers who are aware of where to find metrics. Then all the participants would be asked to compare libraries that fulfill similar purposes based on the metrics they have found and give why they made such decisions. For example, `vis.js` and `d3.js` are both libraries that fulfill the purpose of creating visualizations, so seeing software engineers' preference would help establish what metrics do they value. Of course, they would need to be asked if they already are familiar with one or both of the libraries as that would most likely skew results. In order to determine if `DependencyVis` saved any time, it will be important to keep track of how long it took for the participants to complete each task. Once participants perform all these tasks, a survey at the end would help gauge the perceptions of the participants. The survey may include questions such as "how frustrating was it to find the metrics?". The group that used `DependencyVis` might be asked questions relating to how useful did they find the tool and whether or not they would use the tool. The group that did not use `DependencyVis` might be asked questions relating to whether or not they want the process automated and whether or not they would like a visualization program to help them compare the metrics. To finalize this case study would be to have both groups switch whether or not they use `DependencyVis` to see if perceptions of `DependencyVis` improve or stay the same.

Such a study can be simplified to a simple demonstration of `DependencyVis` and its use cases in the form of either a video or a class in front of software engineers and then survey the participants on how useful do they see `DependencyVis` and whether or not they see themselves using `DependencyVis` in the future.

BIBLIOGRAPHY

[1] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, page 385–395, New York, NY, USA, 2017. Association for Computing Machinery.

[2] G. Avelino, E. Constantinou, M. T. Valente, and A. Serebrenik. On the abandonment and survival of open source projects: An empirical investigation. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2019.

[3] J. Bohnet and J. Döllner. Visual exploration of function call graphs for feature location in complex software systems. In *Proceedings of the 2006 ACM Symposium on Software Visualization*, SoftVis '06, page 95–104, New York, NY, USA, 2006. Association for Computing Machinery.

[4] D. Ceddia. Deploy react and express to heroku, May 2018.

[5] M. J. Chonoles. Chapter 8 - packages and namespaces. In M. J. Chonoles, editor, *OCUP Certification Guide*, page 144. Morgan Kaufmann, Boston, 2018.

[6] J. Coelho, M. Valente, L. Milen, and L. Silva. Is this github project maintained? measuring the level of maintenance activity of open-source projects. *Information and Software Technology*, 02 2020.

[7] J. Cox, E. Bouwers, M. van Eekelen, and J. Visser. Measuring dependency freshness in software systems. In *Proceedings of the 37th International*

Conference on Software Engineering - Volume 2, ICSE '15, page 109–118. IEEE Press, 2015.

[8] R. Cox. Surviving software dependencies: Software reuse is finally here but comes with risks. *Queue*, 17(2):24–47, Apr. 2019.

[9] D. T. Daniel, E. Wuchner, K. Sokolov, M. Stal, and P. Liggesmeyer. Polyptychon: A hierarchically-constrained classified dependencies visualization. In *2014 Second IEEE Working Conference on Software Visualization*, pages 83–86, 2014.

[10] A. Decan, T. Mens, and M. Claes. An empirical comparison of dependency issues in oss packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 2–12, 2017.

[11] A. Decan, T. Mens, and E. Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 181–191, New York, NY, USA, 2018. Association for Computing Machinery.

[12] S. Diehl. *Software Visualization Visualizing the Structure, Behaviour, and Evolution of Software*. Springer, 2007.

[13] M. Ferreira, G. Avelino, M. T. Valente, and K. A. M. Ferreira. A comparative study of algorithms for estimating truck factor. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pages 91–100, 2016.

[14] M. L. Griss. Software reuse architecture, process, and organization for business success. In *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*, pages 86–89, 1997.

[15] J. Hejderup. *In Dependencies We Trust: How vulnerable are dependencies in software modules?* PhD thesis, 05 2015.

[16] J. Hejderup, A. v. Deursen, and G. Gousios. Software ecosystem call graph for dependency management. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, pages 101–104, 2018.

[17] A. King. Understanding the npm dependency model, Aug 2016.

[18] R. G. Kula, C. De Roover, D. German, T. Ishio, and K. Inoue. Visualizing the evolution of systems and their library dependencies. In *2014 Second IEEE Working Conference on Software Visualization*, pages 127–136, 2014.

[19] R. Meloca, G. Pinto, L. Baiser, M. Mattos, I. Polato, I. Wiese, and D. M. German. Understanding the usage, impact, and adoption of non-osi approved licenses. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 270–280, 2018.

[20] L. Merino, A. Bergel, and O. Nierstrasz. Overcoming issues of 3d software visualization through immersive augmented reality. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 54–64, 2018.

[21] L. Merino, M. Ghafari, and O. Nierstrasz. Towards actionable visualisation in software development. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 61–70, 2016.

[22] J. P. Moraes, I. Polato, I. Wiese, F. Saraiva, and G. Pinto. From one to hundreds: Multi-licensing in the javascript ecosystem, 2020.

[23] A. Pano, D. Graziotin, and P. Abrahamsson. Factors and actors leading to the adoption of a javascript framework. *Empirical Software Engineering*, 03 2018.

[24] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, New York, NY, USA, 2018. Association for Computing Machinery.

[25] F. Pfahler, R. Minelli, C. Nagy, and M. Lanza. Visualizing evolving software cities. pages 22–26, 09 2020.

[26] R. Powell and M. Stahnke. The 2020 state of software delivery, 2020.

[27] D. C. Schmidt. Why software reuse has failed and how to make it work for you.

[28] D. Seider, A. Schreiber, T. Marquardt, and M. Brüggemann. Visualizing modules and dependencies of osgi-based applications. pages 96–100, 10 2016.

[29] G. Singal. How to use an npm rest api to get npm audit results - dzone integration, Sep 2019.

[30] M. Stenius. How to get started with d3 and react, Jun 2019.

[31] C. Vendome, D. German, M. Di Penta, G. Bavota, M. Linares-Vásquez, and D. Poshyvanyk. To distribute or not to distribute? why licensing bugs matter. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 268–279, 2018.

[32] D.-L. Vu. A qualitative study of dependency management and its security implications. 07 2020.

[33] E. Wittern, P. Suter, and S. Rajagopalan. A look at the dynamics of the javascript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, page 351–361, New York, NY, USA, 2016. Association for Computing Machinery.