REM: Visualizing the Ripple Effect on Dependencies Using Metrics of Health

Zhe Chen
University of Victoria
Victoria, Canada
zkchen@uvic.ca

Daniel M. German
University of Victoria
Victoria, Canada
dmg@uvic.ca

***Abstract*—In recent years, free and open source software (FOSS) components have become common dependencies in the development of software, both open source and proprietary. As the complexity of software increases, so does the number of components they depend upon; in addition, components are also depending on other components. Thus, their dependency graphs are growing in size and complexity. One of the current challenges in software development is that it is not trivial to know the full dependency graph of an application. Developers are usually aware of the direct dependencies their application requires, but might not be fully aware of the dependencies that those dependencies require (the transitive dependencies). Unfortunately, transitive dependencies can break any software application; therefore, project developers need tools, methods and visualizations to inspect the health of these transitive dependencies and their potential impact.In this work, we propose the Ripple Effect of Metrics (REM) dependency graphs, a visualization of dependency graphs that leverages metrics of the health of dependencies. The two main features of REM dependency graph are: first, to display, and potentially summarize, the full dependency graph of an application based on the health of each of its dependencies; and second, to evaluate the ripple effect of potentially risky dependencies on the rest of the dependency graph. The REM helps application developers inspect the health of all of its dependencies, and also the impact that some of these dependencies might have. By showcasing two examples of popular NPM JavaScript application, we demonstrate that the combination of the ripple effect on the dependency graph using health metrics activity can be beneficial to developers. The advantages of REM graphs are: 1) the metric of health annotation is useful for evaluating the health of dependencies, and 2) the ripple effect of a vulnerability provides an easy method to identify potential risk in a dependency chain and 3) the summarizing mechanisms of the REM help reduce the size and complexity of the large dependency graphs, while focusing in specific aspects of the health of the dependency graph.**

***Index Terms*—Dependency graph, Software components, Metrics, Health**

I. Introduction

Contemporary software development relies on the reuse of components, many of them open source. Each of these components—a dependency—might have an independent software development process, with its own developer's team, and release cycles. Dependencies can be described as direct or transitive dependencies based on their visibility to developers. With current technologies such as automated bots from GitHub, developers are only informed of the direct dependencies they explicitly list. However, each of these dependencies might rely on other dependencies for functioning (the transitive dependencies). Security vulnerabilities are among the most pressing problems in open-source software package libraries [1]. Ducan et al. studied the ecosystem of (Node Package Manager) NPM, the world's largest JavaScript software distributor, and identified the risk of depending on transitive dependencies that have become obsolete or inactive [2]. Application project developers should be more careful when seeking any library update opportunity. The well-known November 2018 event-stream incident[1] has also demonstrated how can a package transitively break software that transitively relies on it. Kikas et al. observed that the number of transitive dependencies has grown 60% within a year (from April 2015 to April 2016), but package dependency management practices have received little attention despite being a crucial part of any software development [3].

In this work, we propose a dependency graph visualization that uses metrics of health (e.g. those that measure aspects of the development process, such as quality, development activity, etc.) and propagates these metrics up the dependency graph. We call this visualization the Ripple Effect of Metrics (REM) dependency graph. The REM graph uses a hierarchical layout to distinguish the dependency relationship within the dependency graph of a software application.

Our visualization is designed to help developers identify potentially vulnerable dependencies based upon their metrics of health, and how these potential vulnerabilities propagate through the dependency graphs. The contributions of this work are: 1) the definition of the REM graphs, 2) an illustration of the usefulness of REM graphs in the analysis of two samples of large open-source applications from developers' point of view of both, runtime and development dependencies, which demonstrates that REM graphs help to (i) understand the health of each dependency chain on the graph using metrics of health of each individual dependency, (ii) identify vulnerable dependencies that require further attention by recognizing a second health metric and displays every valid path to the software application through edges.

[1] https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident

978-1-7281-9914-6/20/$31.00 ©2020 IEEE
DOI 10.1109/VISSOFT51673.2020.00011

II. Background and Related Work

In contemporary software development process, developers usually separate runtime from development dependencies. The runtime dependencies are packages used when the application is running and working, while development dependencies are packages used during build time [4]. German et al. studied the dependency graph of Debian GNU/Linux ecosystem using building and running dependency information as defined in a Stage section in Debian package management tool [5]. Similarly, in the NPM ecosystem, the running and building dependencies are separated into two different types: *dependencies* and *devDependencies*, respectively. Assume A is an NPM package that provides some functionalities in helping build another software B within the NPM ecosystem. A is usually referred to as a **dependency** of B, and B is referred to as a **dependant package** of A. We use the term **dependency** and **library** interchangeably.

Many researchers have investigated the modeling and visualization of dependency graphs [3]–[10], [15], [16]. For example, Falke et al. suggested a hierarchical graph layout for visualizing the functional dependency graph of software applications; using this hierarchical layout, users can view the top-level nodes as a first overview and step-wise unfold the nodes on demand to dig into details [9]. Dias et al. implemented Hunter, a visual tool that includes node-link diagrams as dependency graphs for understanding JavaScript source code [15]. German used package management files to visualize the dependency graph in the hierarchical layout of three popular applications in Fink, one of the software distribution for macOS, and attempted to identify the use of applications in terms of different types of success [6]. Our visual tool extends German's work by adapting the technique of creating dependency graph and adding the visual annotations that highlight specific dependencies' information to create a meaningful dependency graph.

In recent years, researchers have studied problems associated with dependency relationships using historical data [2], [3]. Decan et al. have adopted the use of external sources [1], [2]. Researchers conducted an empirical study on library migration and showed that most of the studied projects keep outdated dependencies [11]. Yau et al. investigated the ripple effect from the location of a modification to other parts of the system that are affected by such modification [12]. While researchers have been broadly studying the dependency management, two areas have not gained much attention from researchers: package deprecation as a vulnerability, and the use of NPM package scores as information linked to the health of a dependency. The former area can have a significant impact on the ecosystem [13], and package scores provide valuable information regarding the way a package is perceived by its development process and community. Robbes et al. used the ripple effects of deprecation to study the propagation of changes in the ecosystem as a whole [13].

To identify library update opportunities, Kula et al. defined a library-centric dependants diffusion plot that used a radial layout and heat-map to show the change in dependencies on the package's release history. Their visualization shows the library version usage and dependency change by assigning different colours and shapes to the library nodes [7]. While their work showed a history of package release on direct dependencies, the visualization did not continue to show any connection among these dependencies. In contrast, our visualization contains both direct and transitive dependencies and shows the relationship between each package node.

Todorov et al. extended on Kula's work by developing an orbital layout visualization that adapted the Wisdom of the Crowd to show the libraries' update opportunities [8]. In contrast, our work visualizes a filtered dependency graph with both direct and transitive dependency to expose in-depth dependency chains. Whereas, their visualization places only the first-level dependencies around the application. Their visualization contains node annotation representing the library freshness status. However, our work includes two annotation techniques on graph nodes: vulnerable dependencies (library deprecation, as an example in this work) and other metrics of health (NPM score metrics, as examples in this work). In addition to node annotations, our visualization also annotates the edges to represent the ripple effect of vulnerable dependencies.

III. Visualization Model

Our visualization, the Ripple Effect of Metrics (REM), is a dependency graph with a hierarchical layout where nodes and edges are annotated in a way that helps in identifying dependencies that are problematic, and the ripple effects that these dependencies have over the entire dependency graph. Our work has two variants of the visualization: a full dependency graph and a filtered dependency graph. The latter variant focuses more on the upper level of the dependency chain in the graph by hiding and modifying lower level nodes that are unlikely to be vulnerabilities (according to some metrics of health).

In the following subsections, we first describe the design of the REM. Then, we describe three main features that are highlighted by the REM dependency graph: metric of health, ripple effect metric, and graph filtering technique.

A. Graph Design

REM is a dependency graph; that is, a directed acyclic graph (DAG) where the root of the graph represents the application of which the visualization corresponds, and other nodes are its dependencies. There exists an edge from node A to node B if A explicitly requires the dependency B.

We have tested and compared many other graph layouts such as circular and spectral layout, and we find the hierarchical layout to be the best in visualizing the dependency relationship. Therefore, the REM dependency graph is rendered using a hierarchical layout such that the root of the graph is at the top, and upper-level nodes represent dependencies that require those to which they are connected in lower-level nodes. For instance, Fig. 1 is an example of the REM dependency graph of a hypothetical application A. In this example there

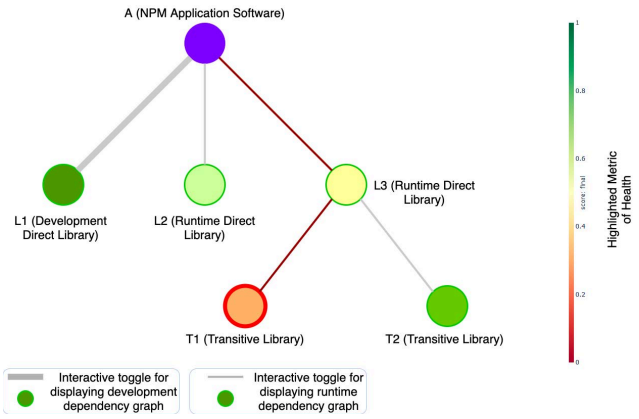


Fig. 1: A REM dependency graph example

are three direct libraries that application A explicitly requires; two of them are runtime libraries (node L2 and L3), and one is a development library (node L1). The REM dependency graph uses thicker edges to represent development dependencies in order to distinguish them from runtime dependencies. Nodes T1 and T2 on the third level of the graph are dependencies of node L3 and hence, both are transitive dependencies of application A.

In the REM, nodes are annotated to highlight two types of information: (i) a metric of health, which is any numeric score associated with a package, and (ii) a vulnerability metric, which is a binary indicator associated with problematic packages. This vulnerability might be provided by a third party (e.g. GitHub listing packages that are known to have security vulnerabilities), or it might be computed from another metric by providing a value that divides the range of the metric into vulnerable or not (we will discuss this method below).

The REM graph allows users to interactively choose the highlighted metric. As shown in Fig. 1, at the right of the REM, there is a colour-scale legend that explains the mapping between the colours of the nodes and the value of the metric of health in each node. We also implemented an interactive function for users to only show runtime dependencies, development dependencies or both at the same time. Another design element is the tool-tip, namely the nodecard, assigned to each node to display information on the node that user selects, as shown as an example in Fig. 2. The nodecard uses the colour of the highlighted metric of health according to the colour scale; for example, it displays a red background according to the defined colour-scale for the highlighted metric of health *maintenance* score (as having italic font in the nodecard). The metadata information includes data such as the name of the dependency, metrics of health (all four examples in this work), ripple effect(RE) metric (the library deprecation state in this work), etc.

B. Metric of Health

A metric of health evaluates some aspects related to the health of a dependency–such as its quality. In other words, a

Fig. 2: A nodecard as tool-tip that displays information of the example dependency node *path-is-absolute*

metric of health is a rating that a third party provides regarding the health of the dependency.

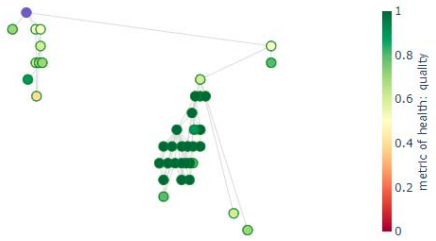
Our implementation is focused on the analysis of NPM applications and their dependencies. For every NPM package (a dependency), NPM provides a set of metrics to evaluate and rank different aspects of its quality, popularity and development process. Specifically, we use the following metrics: popularity, quality, maintenance and final. The *popularity* score is an indicator on how many times the package has been downloaded, the *quality* score includes considerations such as the presence of a README file, stability, tests, up-to-date dependencies, custom website, and code complexity, the *maintenance* score associates with the attention from developers where higher scored package usually is more frequently maintained in terms of release, commit, etc., and the *final* (or optimal) score combines all three score metrics[2].

A REM graph shows information regarding one metric of health at a time, which is selectable by the user. Nodes are annotated with the corresponding metric of health. As shown in Fig. 1, the node's filling-colour highlights the value of the selected metric for each of the dependency nodes. The range of values of the metric is normalized from 0 to 1; hence, the gradient colour-scale varies from red to green, representing values from 0 to 1, respectively. In our example in Fig. 1, node L3 and node T1 have lower scores than L1 and L2 based on the colour of the node and legend on the left-hand side, indicating that developers should be further looking into node L3 and T1.

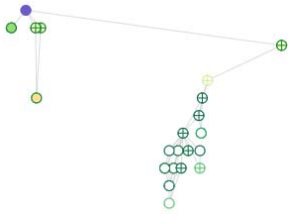
C. Ripple Effect (RE) Metric

The ripple effect is a concept of how an event affects another through propagation. In REM, the 'event' is the existence of a dependency that is considered risky, and the propagation (the ripple effect–RE) is any other dependency (including the root of the graph) that uses the risky dependency. Whether a dependency is risky is determined by a metric of health. Some metrics, such as library deprecation, can be by definition binary. In this case, if the dependency is deprecated, it is considered to be risky. For metrics that have a range of values (such as popularity, quality, number of downloads, etc.), the user of the REM can define a value of the metric as

[2]https://docs.npmjs.com/searching-for-and-choosing-packages-to-download



(a) Original Graph



(b) After Filtering

Fig. 3: Before and After Graph Filtering Comparison - on the runtime dependency graph for *Photonstorm/Phaser(master) using* ***quality*** *score metric*.

threshold: when the metric is above (or below, depending on the interpretation of the metric), the dependency is considered to be risky. Consider the ripple effect of dependencies with maintenance score lower than 0.5 as an example. Nodes with maintenance score higher or equal than 0.5 are considered to be the normal (non-vulnerable), and nodes with score lower than 0.5 are considered risky (vulnerable).

We utilized the outline-colour of the node to represent the RE metric. In REM, the ripple effect is represented as the propagation from a risky dependency node up to the application node. It forms a subgraph where the root is the root of the original graph, and the leaf nodes are all the risky dependencies (and all the nodes in any path between the root and any risky dependency).

Visual-element wise, in the REM dependency graph, every edge has a light grey colour, except for those edges affected by risky dependency nodes through the ripple effect are annotated with dark red colour. For instance, consider Fig. 1 as an example that uses deprecation state of a dependency as the RE metric; node T1 is considered vulnerable, since it has thicker red outline-colour, which represents a deprecated dependency. Node T1, A, and L3 are part of the ripple effect, because node T1 is the vulnerable dependency, node A is the application, and node L3 depends on vulnerable node T1 and is also the direct dependency of application node A. This ripple effect is visually represented by the node T1 annotated with red outline-colour and every path from node T1 to node L3 and node L3 to application A is annotated with dark-red colour.

D. Graph Filtering

It is not uncommon for large applications to have dependency graphs with dozens (if not hundreds) of nodes. Not only the graphs are large in term of size, but also in complexity. We have created the filtered REM to address this issue. It is a variant of the REM in which dependencies are collapsed (i.e. removed from the visualization) and grayed-out (i.e. had filling-colour removed from the visualization).

The filtering is composed of two steps. First step: a node B, which is a direct dependency of A, is collapsed if the following two conditions are both satisfied: a) there is no ripple effect between B and A; b) and the metric of health of node A is less than the metric of health of all the descendent dependency nodes that are reachable from node B including node B itself. This collapsed graph will be the input of the second step. The second step: a node B will be grayed-out if its metric of health is better than the metric of health of all of its ancestors that are explicit dependencies of the application (i.e. direct dependencies of the root of the dependency graph). In other words: B (and the edges that connect it) is hidden if B and all of its descendants have better health than the nodes that directly require it, and B is grayed-out if B has better health than the direct dependency nodes that directly or transitively require it.

More specifically, the algorithm is:

- The root of the the filtered REM is the root of the REM
- All nodes directly connected to the root in the original REM are part of the filtered REM, along with the edges that connect the root to them
- The graph is iteratively built until no more changes are observed:
- For any node B not yet part of the filtered REM, if there exist an edge from another node A that is already part of the filtered REM, and the minimum metric among the nodes that B has a path to including B is no better than the metric of A, then B is added to the filtered REM. Also, any edge in the REM connecting B to any other node already in the filtered REM is also added to the REM
- Any node or edge that is part of the Ripple Effect are also added to the filtered REM
- Nodes that have at least one child removed in the filtered REM are annotated as such (i.e. they are collapsed). We use the $\bigoplus$ symbol to represent collapsed nodes.
- Finally, run a second pass that goes through each node in the filtered graph and compare its metric of health against every ancestors in direct dependency nodes. The node will be modified to have white filling-colour and metric of health as outline-colour if its metric is no worse than those ancestors that are direct dependency nodes.

Fig. 3 exemplifies the graph filtering. Fig. 3a is the full runtime REM graph of *Photonstorm/Phaser*[3], a popular GitHub 2D game framework for HTML 5, that has 4 direct dependencies and 34 transitive dependencies. The example uses

[3]https://github.com/photonstorm/phaser

quality as a metric of health. Fig. 3b is the filtered version of Fig. 3a. As a result, the graph size has been filtered down to 4 direct dependencies and 15 transitive dependencies. Nodes that were removed (by collapsing) are having better metrics of health than their parents (as depicted in having greener filling-colour). After the collapsing, A group of green nodes located at right side of the graph updated their filling-colours to white and outline-colours changed to the colours according to their metric of health. Filtered REM highlights the dependency that has a worse numerical metric (yellow filling-colour) by collapsing healthier nodes (greener filling-colour) and graying-out non problematic nodes (greener filling-colour than the direct dependency). In the visualization, we annotate a cross symbol (+) on the circle shape (O) to nodes with at least one of their dependencies collapsed.

IV. Implementation

The visualization of REM is implemented using a third-party Python graphing library, Plotly[4]. The following sections describe our process for creating the dependency graph model and obtaining data required in the REM dependency graph.

A. Building the Dependency Graph

1) Data Collection: Before constructing a dependency graph, it is necessary to obtain the dependency information of an application. Researchers [3], [4] have been obtaining the NPM package data via its public API endpoint[5]. However, after the NPM official announced in 2016 in the official blog[6] that the API registry endpoint that queries all NPM metadata has been deprecated. Therefore, we retrieved metadata from an alternative domain, namely the replicate registry[7], which is similar to the public NPM registry and is hosted on CouchDB[8] by the NPM official team. We executed the *GET /db/_all_docs view* with *include_docs=true* built-in CouchDB feature enabled and successfully extracted a recent list of NPM `package.json` metadata.

2) Data Pre-processing: From the collected NPM metadata, we extracted the information from *dependencies* and *devDependencies* field to obtain the dependency relationships and store in a SQLite3[9] database for both *runtime* and *development* dependencies, respectively.

3) Graph Rendering: We implemented a Python script that creates the dependency graph from Networkx[10] DiGraph Model. The layout of the graph is done using the **hierarchical layout** from Graphviz dot model[11]. The user interface to navigate, inspect, and filter for the graph was done using the Plotly graphing library[12].

[4]https://plotly.com/python/

[5]registry.npmjs.org/-/all

[6]https://blog.npmjs.org/post/157615772423/deprecating-the-all-registry-endpoint

[7]http://replicate.npmjs.com/

[8]https://couchdb.apache.org/

[9]https://www.sqlite.org/

[10]https://networkx.github.io/

[11]https://www.graphviz.org/pdf/dotguide.pdf

[12]https://plotly.com/python/

B. Metrics of Health

The metric of health is used to annotate the nodes in a REM dependency graph. Such a metric can be any numerical value (in the actual visualization we normalize the range of a metric to values between 0 and 1). In this work, we used four package *score metrics* from the NPM search engine (as described above, these were final, popularity, quality, and maintenance). These metrics are computed by NPM and provided to its users as a method to evaluate and rank each package. We collected the score metrics using the NPM public registry API[13]. The user can interactively choose the metric of health he or she is interested to inspect.

C. Ripple Effect (RE) Metric

In this work, we implemented a version of the REM dependency graph that uses the deprecation state of a dependency as the RE metric and shows the ripple effect of deprecated dependencies (the condition) as an example. We extracted the *deprecated* field from NPM metadata, which is a field attribute that will only be presented in a deprecated package. To compute the ripple effect of the deprecated dependencies, we calculated every path from each deprecated dependency to the application node and marked those paths as ripples through the visualization of graph edges.

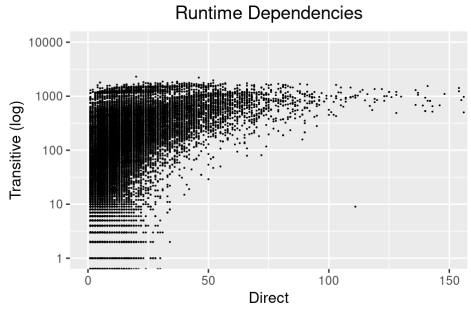
V. Application Examples

In this section, we will illustrate the use of the REM dependency graph using the four NPM's metrics of health from a developer's point of view. We applied the REM on two selected examples from popular GitHub JavaScript software that use NPM as their dependency package manager.

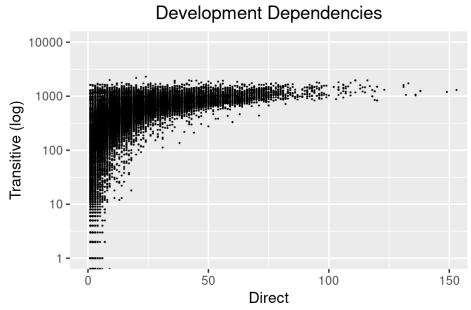
We first wanted to see what the typical dependency graphs of an NPM application looks like. For this reason we decided to use GitHub applications that use NPM packages that are not themselves hosted in NPM. This way we expect to retrieve actual applications and not libraries. We downloaded the top 107,467 (in terms of number of stars) NPM applications that satisfied this conditions. Using the *package.json* we created their dependency graphs. We noticed that many of these applications did not have any dependencies. So we further filtered them to at least have one runtime dependency and one development dependency. We were left with 54,736 applications. Fig. 4 shows scatter plots for runtime and development dependencies for these GitHub NPM application. As these figures shows, the number of direct runtime dependencies tends to be small (median 7) but the number of total transitive dependencies is very large (median 80). The numbers for development dependencies are even more skewed with a median of 9 direct development dependencies and 408 transitive ones.

From these GitHub applications we then chose two software applications: *Adobe Brackets* and *Wekan* as examples to show the use of REM graph. These are the two top popular GitHub JavaScript repositories that are real-world NPM applications (there are some NPM repositories that are more popular, but

[13]https://github.com/npm/registry/blob/master/docs/REGISTRY-API.md



(a) direct and transitive runtime dependencies distribution



(b) direct and transitive development dependencies distribution

Fig. 4: Two scatter plots that show distributions of the relationship between direct and transitive dependencies across over 100,000 popular GitHub NPM/JavaScript applications

they are tutorials). For each application we built their REM dependency graphs by parsing the file `package.json` from which we extracted both the runtime and the development dependencies information of each application.

A. Example - Adobe Brackets

Adobe Brackets[14] is a popular open-source code editor for the web written in JavaScript with over 30,000 stars on GitHub. It has 12 direct runtime dependencies and 30 development dependencies. Fig. 5 shows a full REM graph of the *Adobe Brackets* with final score metric of health and library deprecation ripple effect. As it can be seen, this small number of dependencies has exploded into a graph that contains a total of 432 runtime dependencies and 592 development dependencies. This size of the graph for this application highlights the scalability problem that developers encounter. To alleviate this problem, the user has several mechanisms. First, the ability to zoom-in and out of regions of interest. Second, the ability to choose either run-time or development dependencies, and finally, the filtering (collapsing and graying-out) on nodes.

Fig. 6 is an example of selecting only runtime dependencies with filtering. It shows the graph using two different health metrics. Fig. 6a uses the *final* score metric, while Fig 6b uses the *maintenance* score metric. As we can see the *final* score metrics on most nodes look relatively healthy (as having mostly green nodes). However, the *maintenance* score shows a large number of nodes with low maintenance (as having yellow to red node filling-colours), suggesting that most of its dependencies might not be well-maintained (i.e. lack of the attention from development team).

We then take a close look at the ripple effect of library deprecation by zooming in the graph from Fig. 6a, as shown in Fig. 7. We noticed that *Adobe Brackets* has 4 dependencies (out of 12 dependencies) affected by the ripple effect of library deprecation (as connected by dark-red edges), and two of them have been deprecated: *opn* and *request*. Therefore, these two dependencies show update opportunities and need to be further examined carefully by developers. Specifically, this graph highlights that the ripple effect originates in three transitive dependencies: *path-is-absolute*, *os-tmpdir*, and *os-homedir*, and they affect three direct dependencies (*npm*, *temp* and *decompress-zip*). This suggests that there is a potential risk of using *npm*, *temp* and *decompress-zip*, hence the developers should take a careful look at the transitive dependencies that have been deprecated.

B. Example - Wekan

In the second example, we selected *Wekan*[15], a popular open-source kanban board software that has over 15,000 stars on GitHub. Fig. 8 is the full REM graph of *Wekan* with *final* score metric and library deprecation ripple effect. As opposed to the previous example, we will focus on the development dependencies in this example. Fig. 9 shows development dependencies on two filtered REM graphs of *Wekan*. Fig. 9a shows a REM graph with the NPM *popularity* score and Fig. 9b shows a REM graph with the *quality* score metric as their metrics of health. Both graphs use the deprecation state of a dependency as RE metric. Fig. 10 is a closer look of Fig. 9a, and we added names to the dependencies on the graph that we will be discussing later on.

Fig. 9a shows the *popularity* score of development dependencies of *Wekan* on a REM graph. From Fig. 10, we noticed three direct dependencies are having relatively bad metrics on the graph: *eslint-import-resolver-meteor*, *eslint-plugin-meteor*, and *eslint-config-meteor*. It suggests that these three dependencies are less popular than other dependencies (i.e. less downloads, etc.). To further examine them, we look at all three dependencies using the *quality* score metric, as shown in Fig. 9b. And we see that the overall quality of the REM graph of *Wekan* is good since there are no red nodes on the graph. Because *Wekan* is built with Meteor framework, and the three dependencies we examined (*eslint-import-resolver-meteor*, *eslint-plugin-meteor*, *eslint-config-meteor*) are designed to support this framework which cannot be replaced, therefore, they do not need to be further inspected.

Another aspect of the graph we looked at is the ripple effect. From the REM graph in Fig. 10, We noticed that two direct

[14]https://github.com/adobe/brackets

[15]https://github.com/wekan/wekan

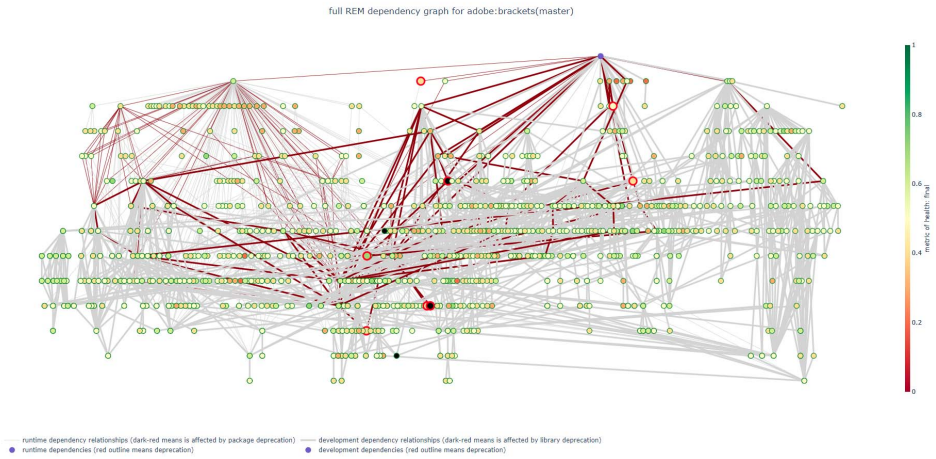
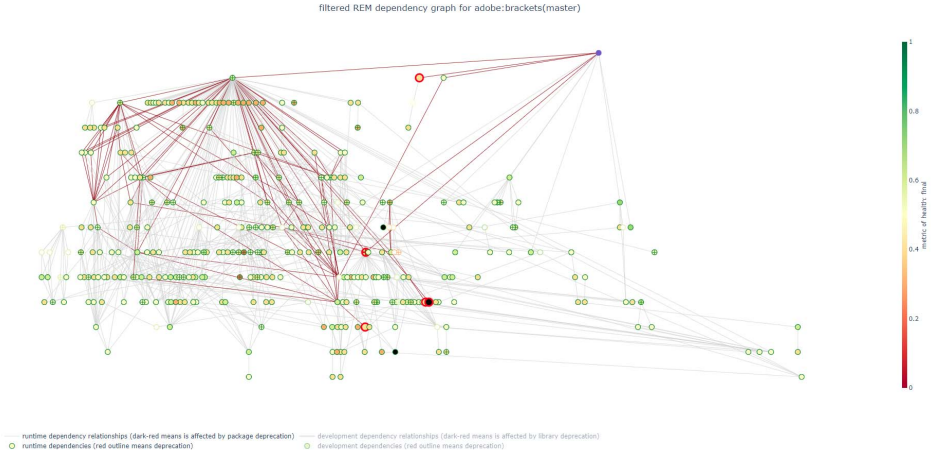
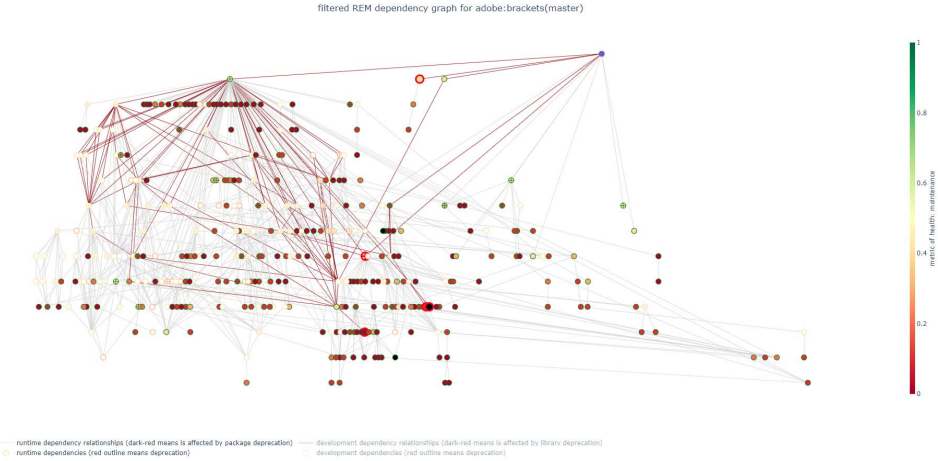


Fig. 5: a full REM graph of *Adobe Brackets* with metric of health (*final* score) and RE metric (*library deprecation*)



(a) a filtered REM graph on runtime dependencies with ripple effect of *library deprecation* and NPM *final* score metric



(b) a filtered REM graph on runtime dependencies with ripple effect of *library deprecation* and NPM *maintenance* score metric

Fig. 6: Two filtered REM graphs of *Adobe Brackets* (master branch) runtime dependencies

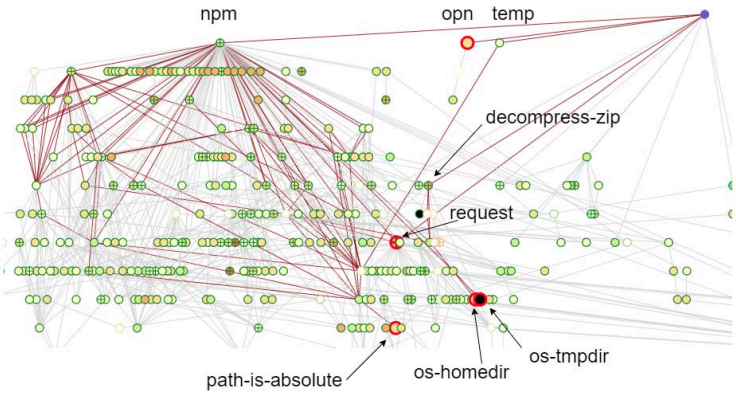


Fig. 7: a zoomed version of Fig. 6a with added names for focused dependencies

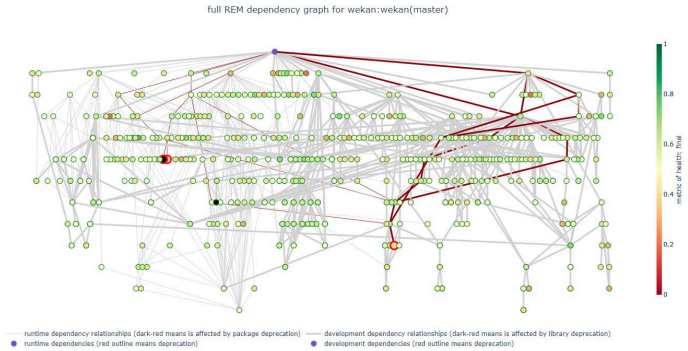


Fig. 8: A full REM graph of Wekan with metric of health (final score) and RE metric (library deprecation)

dependencies (*prettier-eslint* and *eslint*) are affected by the ripple effect of a deprecated dependency (*path-is-absolute*). Because this deprecated dependency is hidden in the deepest of the dependency chain, developers might not be aware of its existence. It suggests that the dependency *path-is-absolute* needs to be reviewed by developers, and developers should take actions (such as look for alternative suggested from deprecation message that owner of the deprecated package would often offer) if any risk has found in this dependency.

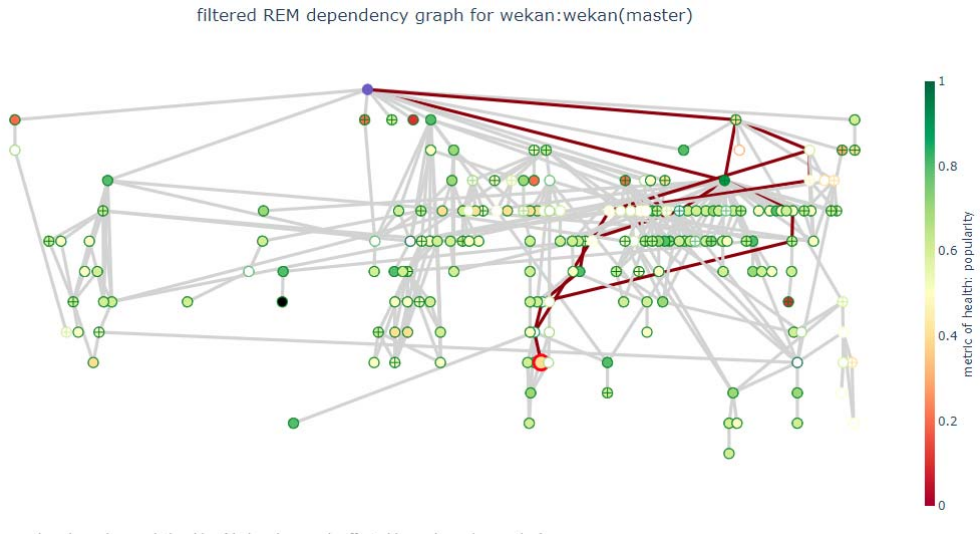
VI. Discussion

A. Data Reliability

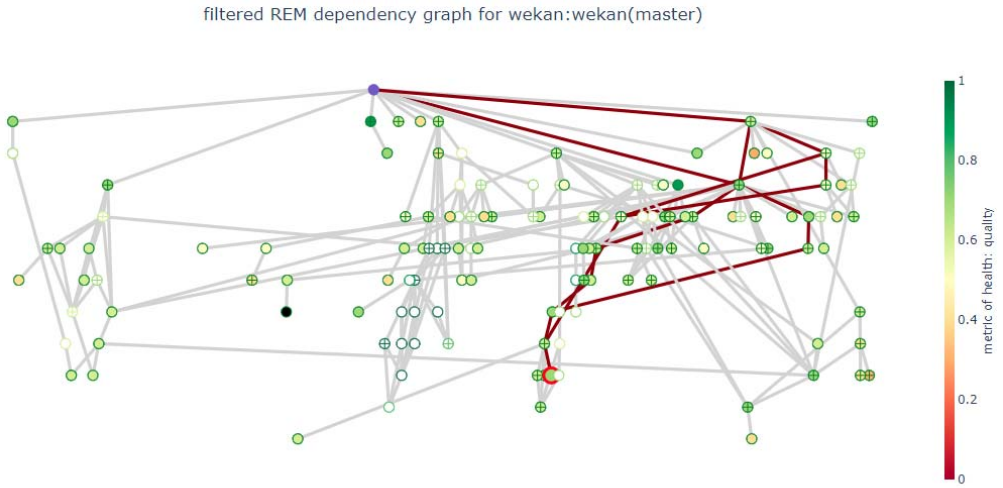
1) NPM Package Data Source: A robust dependency graph is built upon reliable data source. We implemented the REM graph based on the data collected from the data source, namely the replicate registry[16], which is a registry database containing NPM package metadata. However, NPM uses another one as its official data source, namely the public registry[17], which has more recent and accurate package metadata than what we used in our work. But ever since the public registry removed its API endpoint for large data collection, the replicate registry becomes one of the few choices. Another source we have found that contains package data is the libraries.io[18], which had been used as the data source by many researchers in their works [2], [14]. We chose to use the replicate registry as our data source mainly because it is the suggested solution provided by NPM to collect package data[19]. Although we have not systematically tested the data integrity on either data source, we still believe the official team's data source is more reliable and consistent than third-party platforms.

2) Metrics Data Source: We presented four scores as examples of the metric of health in the REM graph. These four metrics, also known as NPM search rank criteria, are what developers see when searching for packages. However, sometimes community members claim that these scores are often broken and have unreliable results[20]. We noticed that one of the NPM official blogs[21] mentioned that the NPM uses scores from a third-party platform, namely npms.io[22]. We looked at services from npms.io and found out that the scoring system in npms.io is similar to NPM search rank

[16]https://replicate.npmjs.com/
[17]http://registry.npmjs.org/
[18]https://libraries.io/
[19]https://blog.npmjs.org/post/157615772423/deprecating-the-all-registry-endpoint
[20]https://npm.community/t/package-search-scores-are-broken/10188/4
[21]https://docs.npmjs.com/searching-for-and-choosing-packages-to-download
[22]https://npms.io/



(a) a filtered REM graph on development dependencies with ripple effect of *library deprecation* and NPM *popularity* score metric



(b) a filtered REM graph on development dependencies with ripple effect of *library deprecation* and NPM *quality* score metric

Fig. 9: Two filtered REM dependency graphs of *Wekan* (master branch)

criteria according to its documentation[23].

B. Limitations and Threats to Validity

1) Defined Dependencies: We built the dependency graph based on the dependency relationship defined in the runtime and development dependency fields defined in the `package.json` file of the respective application. Therefore, our implementation might be biased and does not properly include or describe the information on every dependency for several reasons: 1) dependencies can be required directly through the JavaScript file, 2) private or non-NPM published dependencies will either have inaccurate or no metric of health; therefore, can hide from the REM graph, 3) the list of dependencies required might include some that are not used.

2) Dependency Version Constraints: Another factor that might affect the accuracy of the build process of the dependency graph is the dependency version constraint. Although there are few package management systems such as CRAN that only allows applications to use the latest dependency, most of other package managers for variety of programming languages such as NPM that allows developers to define the range of the release of a dependency to accept. Our implementation of REM graph for NPM applications ignores the version constraint that might exclude certain dependency releases by only accepting the latest release, and therefore,

[23] https://github.com/npms-io/npms-analyzer/blob/master/docs/architecture.md

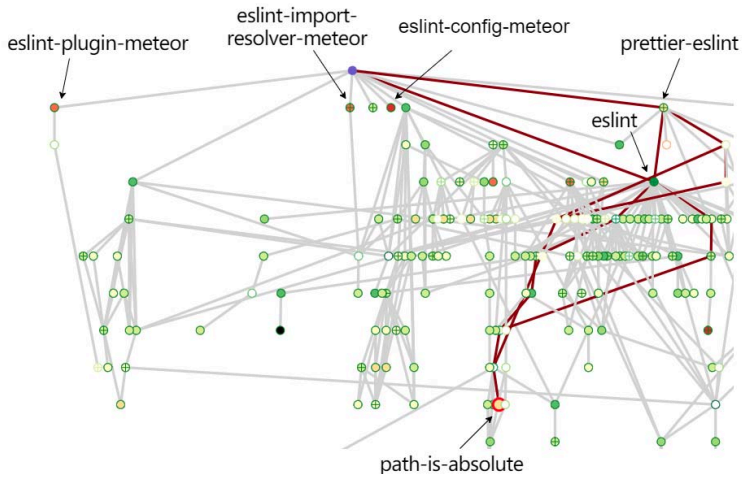


Fig. 10: a zoomed version of Fig. 9a with added names for focused dependencies

may result in inaccurate information on the dependency graph, due to the lack of differentiation between project versions in NPM [3].

3) Graphing Library: In this work, we implemented the visualization using a Python graphing library, Plotly. It offers an easy access to a wide range of tools that helped in constructing REM graph such as built-in colour-scale, zoom feature, and HTML output option. However, Plotly generates a static REM graph and lacks of many interaction features when comparing to an other library, D3.js[24], which makes the current implementation hard to expand any new features without changing to other graphing libraries.

4) Huge Graph Presentation: Because nodes and edges are two dimensional in REM, the visibility of the edge on huge graphs can be difficult to see. Especially when developers try to inspect the dependency relationships between certain packages, they can get confused because of the overlapping edges in static REM. However, this has not been tested and evaluated by the developers.

VII. Future Work

For future work, one significant change we plan to add to our current implementation is to include the consideration of dependency version constraint. We plan to test our REM graph based on different dependency graph construction approaches that Kikas et al. has discussed [3]. And to test REM graph, we plan to work on the evaluation of the usefulness of the REM graph by r it up as a GitHub plugin on the Marketplace[25] for developers to use on their NPM software projects and collecting their feedback. We will adjust and create more features to the visualization based on the feedback.

Since our implementation currently generates separate REM graphs on different metrics, that is, for every application, there will be a full and a filtered version of the REM graph for each application with metrics selected (one metric of health and one RE metric). Therefore, the second future work is divided into two tasks: 1) to develop a mechanism that interactively compares different metrics of health on the same REM dependency graph, 2) to have an interactive way to allow user view ripple effect of different RE metrics.

Lastly, the fast pace of the software development process requires developers to assess their dependencies every short period of time due to fast release cycle. Therefore, we would like to see the possibility of complementing the REM graph with real-time data instead of the collected data that we used in this work which can quickly age over time. To achieve this, we need to investigate a cost-efficient way to keep NPM metadata updated and reflect it on the real-time version of the REM.

VIII. Conclusion

Dependency management has been a critical task for developers in the contemporary software development process. Our analysis has shown that NPM applications on GitHub are using a large number of dependencies, which have further resulted the increasing complexity in software. The challenge for developers has become to be aware of the transitive dependencies, their health, and their potential impact in the dependency graph of a given application. In this work, with the goal to help developers study the dependency health of the application, we proposed a Ripple Effect of Metrics (REM) dependency graph using four metrics (final, popularity, quality, and maintenance) and one Ripple Effect metric (library deprecation). We also demonstrated, using two NPM applications and several NPM metrics of health, how the REM graphs can be used to identify potentially problematic libraries in their dependency graphs.

[24]https://d3js.org/
[25]https://github.com/marketplace

REFERENCES

[1] A. Decan, T. Mens and E. Constantinou, "On the Impact of Security Vulnerabilities in the npm Package Dependency Network", IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), 2018, pp. 181-191.

[2] A. Decan, T. Mens, P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems", Empir Software Eng 24, 381–416 (2019). https://doi.org/10.1007/s10664-017-9589-y

[3] R. Kikas, G. Gousios, M. Dumas, D. Pfahl, "Structure and Evolution of Package Dependency Networks", IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2017.

[4] E. Witten, P. Suter, S. Rajagopalan, "A Look at the Dynamics of the JavaScript Package Ecosystem", IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2016.

[5] D. M. German, J. M. Gonzalez-Barahona and G. Robles, "A Model to Understand the Building and Running Inter-Dependencies of Software", 14th Working Conference on Reverse Engineering (WCRE 2007), 2007, pp. 140-149, doi: 10.1109/WCRE.2007.5.

[6] D. M. German, "Using Software Distributions to Understand the Relationship among Free and Open Source Software Projects", Fourth International Workshop on Mining Software Repositories, 2007.

[7] R. G. Kula, C. D. Roover, D. German, T. Ishio, K. Inoue, "Visualizing the Evolution of Systems and their Library Dependencies", Second IEEE Working Conference on Software Visualization, 2014.

[8] B. Todorov, R. G. Kula, T. Ishio, K. Inoue, "SoL Mantra: Visualizing Update Opportunities Based on Library Coexistence", IEEE Working Conference on Software Visualization, 2017.

[9] R. Falke, R. Klein, R. Koschke and J. Quante. "The Dominance Tree in Visualizing Software Dependencies", 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2005, pp. 1-6, doi: 10.1109/VISSOF.2005.1684311.

[10] R. G. Kula, C. D. Roover, D. M. German, T. Ishio and K. Inoue, "A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem", IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2018, pp. 288-299, doi: 10.1109/SANER.2018.8330217.

[11] R. G. Kula, D. M. German, Ouni, A. Ouni, T. Ishio, K. Inoue. Do developers update their library dependencies?. Empir Software Eng 23, 384–417 (2018). https://doi.org/10.1007/s10664-017-9521-5

[12] S. S. Yau, J. S. Collofello and T. MacGregor, "Ripple effect analysis of software maintenance", The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. pp. 60-65, doi: 10.1109/CMPSAC.1978.810308.

[13] R. Robbes, M. Lungu, D. Rothlisberger, "How Do Developers React to API Deprecation? The Case of a Smalltalk Ecosystem", Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012.

[14] A. Zerouali, E. Constantinou, T. Mens, G Robles, J. González-Barahona, "An Empirical Analysis of Technical Lag in npm Package Dependencies", New Opportunities for Software Reuse. ICSR 2018. Lecture Notes in Computer Science, vol 10826. Springer, 2018

[15] M. Dias, D. Orellana, S. Vidal, L. Merino, A. Bergel, "Evaluating a Visual Approach for Understanding JavaScript Source Code", 28th International Conference on Program Comprehension (ICPC '20). 2020. https://doi.org/10.1145/3387904.3389275

[16] K. E. Isaacs and T. Gamblin, "Preserving Command Line Workflow for a Package Management System Using ASCII DAG Visualization," in IEEE Transactions on Visualization and Computer Graphics, vol. 25, no. 9, pp. 2804-2820, 1 Sept. 2019, doi: 10.1109/TVCG.2018.2859974.