Analyzing the Criticality of Apache Maven Packages Through a Temporal Dependency Graph

Denis Corlade
Supervisor(s): Georgios Gousios, Diomidis Spinellis
EEMCS, Delft University of Technology, The Netherlands
22-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Developers rely on different software to improve their efficiency as to reuse parts of code and be able to maintain it with ease, which is why open source software libraries have gained much popularity over the past years. This paper analyzes what are the most used packages from *Apache Maven*, which is a build automation tool used primarily for Java projects. In order to do an accurate analysis, it is necessary to collect all the packages and their dependencies from the Maven Central Repository. But, as of 5th of May 2022, analyzing the whole repository proves to be an extreme challenge, as the repository has 9M of indexed artifacts, which contain metadata such as the version and a list of dependencies. Besides, there are many more other repositories that also contain such artifacts. Thus, this paper examines only a small subset of artifacts and provides a data structure that is able to take into consideration the time dependency between libraries, which can be also scaled to suit a much bigger input. The paper concludes with the most used software on given time-frames, but without specifying any time frames, the popular *junit* package comes out first.

I Introduction

Maven[1] Central Repository is one of the biggest repositories of Java Virtual Machine (JVM) artifacts, which consists of information about the name of the package, the date of upload, the exact version that the package uses and a list of dependencies on other artifacts. Maven provides tools to allow software engineers to reuse code and "to allow a developer to comprehend the complete state of a development effort in the shortest period of time", as the developers of Maven have stated. As Mohagheghi et al. also stated [15], there exists significant evidence on apparent productivity gains from using such automation tools. It is for these reasons that such tools have been adopted by many people, and thus, they are also keen on knowing which are the most used libraries of a specific language, such as Java. In this work, an analysis on some packages from Maven Central repository will be performed by taking into account the timestamps at which the packages were released. Therefore, it will allow for an accurate dataset which will be further analyzed to find any intriguing correlations.

Besides simply analyzing packages to find the most used ones, it is of extreme importance to do analysis of the software, as simply pulling software without any examination done might have unfortunate effects. For instance, a few cases of such a misfortune are:

- **Java library Apache Log4j**. *Log4j*[2] was a widely used software which provided logging utility. In 2021, it was discovered that Log4j posed a vulnerability that could allow a threatening user to take remote control of the customer. It is mentioned in this paper [9] that 17,000 packages were affected by this malware, which amounted to 4% back of the Maven Central Repository at the time of the analysis.
- ***Equifax***[3] **incident**. The incident can be briefly described as a major cybersecurity breach that should never have happened. The cause of it was a package named *Apache Struts*, a package which Equifax's software depended on, which suffered from a vulnerability. A patch for the mentioned vulnerability was released months before the incident happened, but because Equifax did not update the vulnerable software, it resulted into a data breach that affected roughly 148 million customers [19].

Mostly, it is because of transitive dependencies that such misfortunes happen, as they often lead to adding fragile packages unintentionally [10].

Moreover, in 2014, an empirical study [20] was conducted on 26 million builds, including some from Java. It is stated that developers build 6.98 times a day and out of those, 29.8% fail. The most common cause of these fails was found to be in the dependencies between packages. Besides, this work found out that developers with either very low failure rates or very high failure rates tend to rarely build, and thus, it shows that inexperienced developers could use a helping hand on knowing the packages they work with.

Previous research has been conducted on this topic, but it is incomplete, as it suffers from one major issue, which is not allowing the user to query at a specific moment in time, and thus not allowing developers to check for any backwards compatibility between dependencies. It is analyzing packages by linking them together, which shows the data is imprecise due to missing the time component. Therefore, the data it provides cannot be fully relied upon.

To give a brief overview of the problem at hand, which is something that is not taken care of in the research explained above, an example like the following would suffice:

- Library A releases a version at time 1, named A-1.1
- Library B, which depends on library A's latest version at the current time, is releasing a version at time 2. Therefore, library B depends on the node A-1.1
- Library A releases a version at time 3, named A-1.2
- Library C, which depends on library B, releases a version at time 4. Therefore, it also depends on library A. Library C should depend on the latest version of library A, which is A-1.2, even though library B depends on version A-1.1

Thus, querying on T2 will show package B-1.1 depend on A-1.1, while querying on T4 will show package C-1.1 depending on package B-1.1 and transitively depend on A-1.2 instead of A-1.1.

[1] https://maven.apache.org/

[2] https://logging.apache.org/log4j/2.x/

[3] https://www.equifax.co.uk/

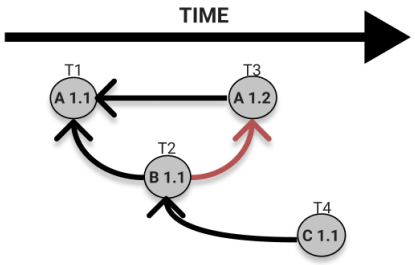

Figure 1: Graph representation of the above problem

The final goal is to measure the number of packages to present an overview of the most used software in Java, taking the above example into account. If that is accomplished, a number of aspects can be concluded, such as if criticality has any correlation to the number of downloads of a package. Thus, finding answers for the research question "**What are the most widely used Java packages at a given time?**" will provide a quantitative and qualitative analysis which considers the time component. To achieve the goal and answer the research question, three sub-questions have been constructed to divide the problem into smaller pieces which need solutions:
RQ1: *What would a graph data structure for package dependencies that contain a time component look like?*
RQ2: *Does the introduction of time increase precision?*
RQ3: *What are the most widely used Java packages?*
Therefore, this work is set out to resolve these questions as they will provide evidence that the data is well analyzed and precise, thus giving users a genuine and reliable overview of the most used packages.

Structure. Section II contains important terms that are commonly used throughout this paper under subsection A, while it also contains related work that provide as background to this work in subsection B. Afterwards, section III describes the methodology and presents a brief overview of the research sub-questions. Results are presented in section IV. Furthermore, responsible research and reproducibility are discussed in section V. A discussion of the results is showcased in section VI, while also going over the threats to validity and limitations of the paper. Lastly, section VII summarizes the work done in this paper and proposes further possible improvements.

II Background

This section contains brief explanations in regards to some terminology terms while also covering more in depth the related work already done on this subject.

A Terminology

The main subject of this paper is to analyse the packages of Maven along with their dependencies. Packages (artifacts), as explained briefly in the above section, provide reusable code for developers to use in their applications and can be added to the projects by using dependency management tools, such as Maven. In Maven terminology, these artifacts are a product that is generated after a Maven project build. Packages can have multiple versions and are always available in repositories.

- **Dependency**. Packages can depend on other packages and thus form a dependency. A dependency is defined as link to a particular package that needs to be built on in order to run the project, thus, something that you rely on. When package A depends on another package B, it means that A has a dependency on B. On the contrary, it can also be defined that B has a reverse dependency, meaning that B has some package depending on it. Moreover, dependencies are defined in 2 ways: *direct dependencies* and *transitive dependencies*. Direct dependencies are dependencies that are clearly specified in the package's metadata, and which they need to build themselves. The latter, transitive dependencies, are dependencies of another dependency that the initial package relies on.

- **PDN, TDPN**. Packages and dependencies, when put together, create a package dependency network (PDN), with the packages being the nodes and the dependencies being the edges that connect the nodes. When adding the time component in the equation, these become temporal package dependency networks (TDPN). TDPNs allow for an extension with the possibility of filtering on a specific time frame.

- **Exclusions**. Besides the information stated above, which applies to mostly all packages despite the language, Maven also contains *exclusions* in the packages' metadata. Exclusions are generally dependencies that come from transitive dependencies but are not wanted in the project. Hence, by using exclusions, Maven allows protecting projects from vulnerable software as mentioned in the previous section.

- ***POM*** (Project Object Model) files are XML files that contain the metadata of an artifact, meaning, information that is necessary to build the project. Each package has its own POM file. What defines a package to be unique is the 3-tuple 'GroupID:ArtifactID:Version' that can be found in the POM file. GroupID combined with the ArtifactID represent the name of the library. ***Effective POM*** is a file generated from the *simple POM* (POM) file + the *super POM* file, which is the parent POM file. The effective POM combines all the configuration needed and specifies exactly all the dependencies of the artifact.

- **PageRank** is an algorithm used by Google Search to rank web pages by finding out how important a page is in addition to how relevant it is. It does that by calculating the links between nodes and sorting them by the highest scorer. Similarly, as Gleich et al. [8] also pointed out, this algorithm can be applied to any graph or network domain, including calculating the most linked packages in this research.

B Related Work

This section will contain relevant information on previous work that relates to this paper. It will briefly analyze the time-dependent graph structure, while also targeting some other work on the Maven ecosystem. Thus, going over related work and other conclusions in this field of study help to provide some answers to the work this paper covers and allow for a broader understanding of the problem at hand.

Time dependent graphs. Even though there have been multiple papers in regards to simple graph data structures, not much work has been put in acknowledging the possibilities of a time dependent graph data structure. As stated in this paper [22], static graphs have been extensively studied, but there is still far from having a concrete set of structures and algorithm frameworks for time-dependent graphs. Therefore, it is up to us to discover whether, for example, the design of this structure could support scalability. Furthermore, the paper goes on to explaining different models of possible TDPNs. One of those models is a snapshot-based system, which goals perfectly suit the work needed for the success of this project. To briefly summarize this model, it manages to divide the TDPN into multiple static graphs (snapshots) over time, that when put together, give the impression of a dynamic graph. However, the system also has its limitations. When faced to generate snapshots for a relatively large input, it consumes large storage space and takes plenty of time for querying. Therefore, the authors' work can provide important wisdom in creating the network's evolution and observing how the packages influence themselves transitively. In conclusion, as this work puts emphasis on creating a scalable data structure, it remains to be seen in section III how this problem is tackled.

Similar work. A study that analyzed the project from Maven Central repository in September 2018 [6], found that in order do complete this task, a huge dataset has to be analyzed, as the numbers of JVM artifacts was around 2.8 million back then, now increasing to 9M [4] artifacts. Besides this problem, the paper also stated that "dependency relationships among artifacts are not modeled explicitly and cannot be queried". Despite these limitations, the work provided a custom Neo4j[5] Docker image with the entire dataset that was analyzed by them. Although the dataset in question focuses only on artifacts from Maven Central repository and also excludes low-level metadata such as dependency exclusions, it can be adequate to this paper as it can be used as a comparison between snapshots, in order to measure the accuracy and precision of the final results.

Moreover, there exists another work that has analyzed packages of popular programming languages, which is *Libraries.io* [1]. This work collects and analyzes packages from multiple dependency managers, including Maven. While it takes care of collecting the packages and their dependencies, it does not resolve the problem mentioned in Figure 1. With that in mind, it is hard to track the history of dependencies of a package or query at a specific time. Thus, although the work can indeed be helpful to the community, it is incomplete. This paper adds what is missing to a particular part of that work and will compare the accuracy of the final data to the dataset provided from *Libraries.io*.

To add to the list of previous work on the Maven Central repository, Kula et al. created a graph [11] to showcase the evolution of software systems and their library dependencies over time. Although the dataset analyzed is quite small, composed of only 6,437 artifacts, the work provides a model that allows to visualize some packages' popularity from the super repository. However, the paper does not consider transitive dependencies and thus the data provided cannot be fully trusted. Nonetheless, it provides useful insight on querying and representing artifacts over time.

Ecosystem. Panichella et al. in reference [4, p. 38] analyze the project dependencies in the Apache ecosystem and conclude that "the dependency phenomenon has an exponential growth and should therefore carefully be considered by developers contributing to the ecosystem". In another paper [5], the authors analyzed project inter-dependencies in the Apache ecosystem and deduced that whenever a new release of a project comes out, in 69% of the cases, developers do not update it, as they prefer updating only when major upgrades come out. In similar fashion, German et al. in reference [12] show that 81.5% of the projects they studied contained outdated dependencies. Moreover, Pashchenko et al. [16] introduced the term of halted dependencies, and when analyzing a sample of data, they found out that 14% of the dependencies are halted and 1% of them contain known security vulnerabilities. The authors also added that these halted dependencies could introduce even more bugs when adding transitive dependencies, and thus expose the root library instance to bugs. On another note, Soto-Valero et al. discovered in reference [21] that more than 90% of the most used libraries are not the latest releases, when they analyzed a majority of the Maven Central repository. Thus, it is of even more importance to analyze the packages before pulling them into your project. There have been multiple works that have analyzed of ecosystem being exposed to vulnerabilities and this paper aims to provide a solution and raise awareness to developers. By always having an analysis of upgrades of dependencies, you minimize the risk of getting into a position similar to the one mentioned in section I regarding the Equifax incident and risking a major exploit happening because of not triggering an update to the newest release version.

Although there have been many works on analyzing the Maven Central repository and its packages, not many have allowed querying at a specific time on the dataset they produced. Besides, many of the works did not go in depth on taking all the metadata into consideration, such as transitive dependencies or exclusions. The writings take into consideration only the transitive dependencies of packages at their latest release version and disregard the possibility of drastic changes over time.

III Methodology

In this section, particular attention will be paid to describe the data gathering process, structuring the temporal graph and

[4] https://search.maven.org/stats

[5] https://neo4j.com/

thus creating the TDPN. Moreover, answers to the research questions mentioned in section I will be provided in order to reach the goal of the paper.

A Overview

This work studies the ecosystem of Java as it is one of the most used languages in 2022 [2]. Additionally, this language also hosts a central repository, as mentioned previously in this paper, the Maven Central Repository, which contains a very large number of the most commonly used packages available to date. Maven Central updates the packages by updating a weekly index, constantly adding to the dataset of the already 9 million packages. This work will gather packages from this weekly index. Packages provide source code that can be of use to some developers, which can request these by using the dependency manager, *Maven*.

To obtain the required information needed for each package, we access the *effective POM* file of each package. Each package's information is put into a single JSON file, which is given input to a graph network. The network is then optimized to allow algorithms such as PageRank to be performed on it. Querying is also implemented on the network, such data all dependencies of a package can be seen at any point in time, and thus see an evolution of the ecosystem.

Research Questions. Research questions provide a path through the research and serve as an evaluation method. Thus, finding answers to them is crucial for achieving the goal of the paper, answers that can be found throughout this section and also section IV. Furthermore, an explanation of why these 3 sub questions were chosen is given:

RQ1 (Structure) Not much is known about time dependent graphs or TDPNs, and especially in this field of work, there were not many attempts on using a time structure. Therefore, it is up to us to discover whether, for example, the design of this structure could support scalability. Thus, answering this question provides an understanding of the current state of time dependent data structures.

RQ2 (Precision). As the time introduction provides more cases to test, we have to be sure that the preciseness of the data matches the ground truth. Therefore, a comparison of resolutions of the ground truth with a resolution of a simple non-time dependent graph structure will be constructed. Finally, a comparison of those results with a time dependent graph structure can be performed. Our goal is to have a precision that matches the ground truth, and if introduction of time makes backwards resolution precise.

RQ3 (Criticality). Once the data from the structure has been verified for precision, a list of the most used packages can be drawn out. Consequently, it is time to analyse it to check for any correlations. It is important to check if there is any resemblance between the number of download counts and the importance of a library. That is possible by using the *PageRank*[6] algorithm. Further algorithms will be tested on the dataset to find any similarities between packages and dependencies.

[6]https://neo4j.com/docs/graph-data-science/current/algorithms/page-rank/

B Data gathering

The dataset is composed of circa 40K different package versions. The source for these packages was a Zenodo[7] dataset containing 2.4M unique artifacts, which were all the artifacts that the Maven Central Repository contained as of September 2018. The CSV[8] containing the names of these artifacts was loaded in and for each of these packages, a script using mvn dependency:get[9] command was performed to get each artifact's POM file. The list of names was shuffled such that the artifacts downloaded would not be from the same sphere, although at the cost of a more sparse graph. The metadata is parsed first by parsing the effective POMs of the POM files generated, except for their release date, which was scraped from the Maven Central Repository HTML webpage. Some timestamps are missing, mostly from ancient package versions. As the dataset gathered is already on the low number, a decision to assign a constant value (0001-01-01T00:00:00Z) to them was taken, and thus, they do not impact the other data. Exclusions are also taken into account, as it would be inaccurate to count transitive dependencies that are not wanted by the authors of the library.

New packages are released every week as an incremental to the index[10] that the Maven Central Repository holds. The incremental can be downloaded and indexed, upon which the same procedures can be performed to collect the data and add it to the final JSON file that the graph needs to be fed, such that the work is not limited to the time of its creation.

C Graph Structure

When deciding to model data with a network, it is essential of thinking what nodes and edges represent. When it comes to this field of work, an effortless approach is to consider packages as nodes, and all the dependencies as the edges that link them. Each package version is treated as an individual node such that there is no deceiving information about the analysis of the graph. Thanks to this work [10] that studied different network constructions, if it were to construct a network with an aggregated approach of having nodes per name of package, it would give a false image of the data. Another similar idea was considered, but instead to add attributes on the edges, such as timestamps and versions. These 2 latter ideas were discarded as one main requirement of the graph is to be as fast as possible to be able to handle big amounts of data, and thus a decision was taken on having simple edges without attributes and simple nodes with a unique id. Thus, even though the number of nodes would be bigger, it would still be more efficient than storing other unnecessary information in the graph, and it also proved to be easier to maintain. Furthermore, to prioritize the efficiency, Go (Golang) was chosen as the language for implementation, as it is simple to use and provides quick compilation, while also allowing for easy upkeep with its tools for automatic code mainte-

[7]https://zenodo.org/record/1489120

[8]https://en.wikipedia.org/wiki/Comma-separated_values

[9]https://maven.apache.org/plugins/maven-dependency-plugin/get-mojo.html

[10]https://repo.maven.apache.org/maven2/.index/

nance [7]. Thus, the project uses one of Go's popular packages, *gonum*[11]. Gonum contains a generalized graph (network) package for Go language, which sped up the process for creating the graph-like data structure.

By using the above approach, when answering what are the most used packages on a specific time range, all project versions are considered and the result is accurate. Although this likely will create a more dissociate graph, it is the best way of assuring the integrity of the result. In regard to querying, the graph was created static such that it would be more efficient when generating the graph (compared to a dynamic one), even for much larger input. The structure uses the snapshot-based approach mentioned in B such that it allows querying by converting the TDPN into multiple static graphs.

However, the approach chosen in regard to modelling the graph structure makes it hard to compare the results to some other existing work, for example, Libraries.io [1], as they use the aggregated nodes approach. The other existing work performed by Benelallam et al.[6] uses a similar approach and can be used as a ground truth to compare whether the time component made had an impact in improving the accuracy.

To add more to the graph construction, a decision was taken to only create nodes for artifacts that are specified in the input file, and not when new dependencies come in. For example, if package A depends on package B, but package B is not found in the JSON file outside of it being mentioned as a dependency, then package B does not have a node created and thus the edge from A to B does not exist. This approach was chosen as it heavily adds to the efficiency of the graph creation, although it misses data when smaller datasets are used.

D *Resolving dependencies constraints*

The challenge that arose with this is that many packages state their dependencies under a constraint. This is required because of the huge number of dependencies that exist nowadays, and thus the appearance of the popular term "Dependency hell" [3]. Therefore, a convention exists in the software development ecosystem surrounding constraints and version names. That is, *SemVer* [17], which consists of a set of rules and requirements that specify how versions should be named and how constraints are specified. Consequently, versions name should follow the structure of *MAJOR.MINOR.PATCH*, with optional extensions labels for *pre-release* and *build metadata*. Furthermore, a constraint denotes what versions of packages the artifact in question depends on. The general syntax rule in place for those is specifying a version that is preceded by one of the following characters: $[<, >, =, <=, >=, !=]$. An example of how a constraint would be treated can be found in Figure 2. The implementation made use of a parsing library[12] from Go that simplified the process. It is also assumed that unstable released are taken into account when querying only when the constraint specifically mentions an unstable constraint. This decision was taken after different developers argued [14] that unstable releases should not be included if the author did not specify them in the constraint. However, not all languages use the general convention when it comes to constraints, as Maven uses ranges, such as $(, 2.1.3]$. As a consequence, the ranges that Maven uses for constraints had to be translated to the characters that SemVer makes use of. Therefore, a function that translated every version syntax range [13] from Maven into the usual SemVer convention was built. Furthermore, one of the cases of these constraints was leaving the version as it is, without any added parentheses, for instance, 1.3.2. Maven treats this as a soft constraint, meaning that it would like to have this version, but if not, it will search for the first version that came after 1.3.2. Thus, for the integrity of the results, this case was treated as >= 1.3.2, as theoretically, any version after 1.3.2 could be considered a dependency.

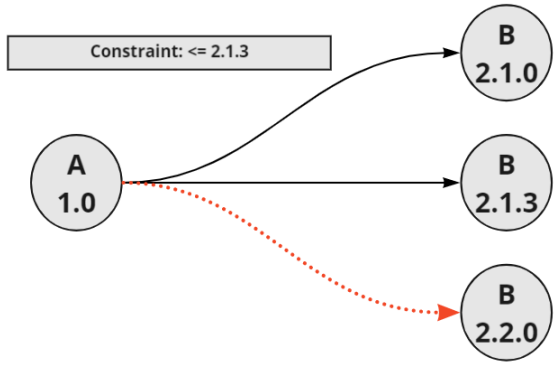

Figure 2: Creating edges based on constraints

Furthermore, when it comes to data from specific time ranges, a query can be performed on the graph, allowing the user to select the edges (dependencies) that match the latest available correct version while respecting the time interval. The following example in Figure 3 would suffice to gain a better understanding:

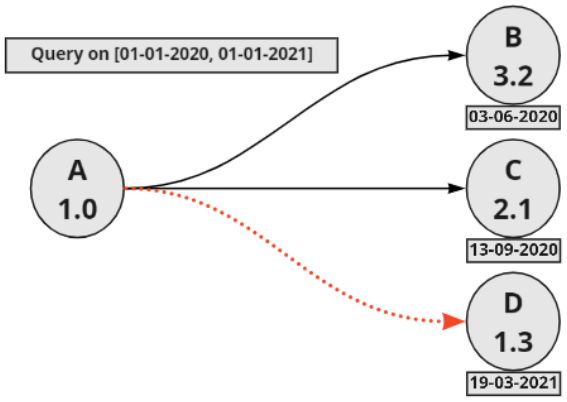

Figure 3: Graph query on a time interval

[11]https://www.gonum.org/

[12]https://github.com/Masterminds/semver

IV RESULTS

A Describing the data structure (RQ1)

In this subsection, a description of the graph data structure will be given and how it can incorporate the time aspect.

Initially, a dataset of 100K POM files was collected but due to technical limitations, only 40K were used. More exactly, the graph contains 41671 nodes (artifacts) and 1587503 edges (dependencies). Figure 4 shows a visualization of the graph using a popular tool, Graphia[13]. A large connected subgraph can be observed in the middle, while the outer lines either contain small subgraphs with a few links or nodes that do not depend on anything, nor do they have a reverse dependency. Having this many nodes unrelated to the main connected component is a side effect of having a smaller dataset in use, as although there exist more packages that have dependencies, if the dependencies packages are not nodes already, there will not be any edges connecting them.

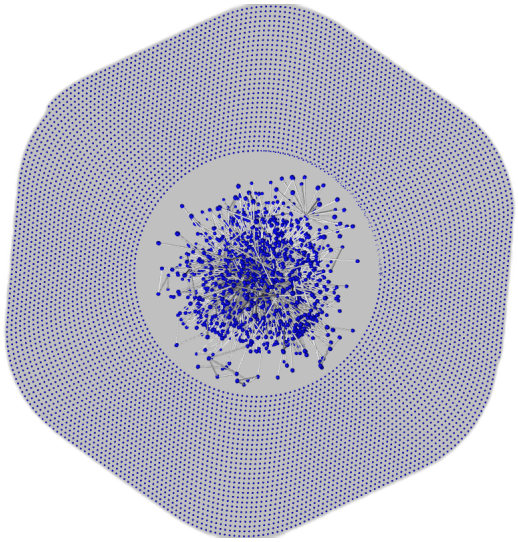

Figure 4: Visualization of the dataset in a graph

The graph is composed of as little information as possible so that it supports larger input data. More information on how the graph was thought out and built can be found in section III. However, what is of interest is if adding the time component in the graph retains the correctness of the data. Therefore, it is useful to check whether dependencies of a random artifact match the dependencies shown in the repository manager. To get all dependencies of a package from the graph, one can make use of a query "Find all possible dependencies of a package", which performs a Breadth-First Search starting from the node that is given as input. For instance, take the widely popular testing package *junit:junit:4.13.2* as an example in table I

As explained in section III, if a version is provided as it is without any ranges to specify its constraint, it is treated as "bigger than". Thus, the dependencies selected satisfy the requirement (>= 1.3). A quick look on the repository[14] can show exactly that the above dependencies are correct.

[13]https://graphia.app/

TABLE I: Dependencies (direct and transitive (t)) of *junit:junit:4.13.2*

GroupID:ArtifactID (constraint)	**Versions**
org.hamcrest:hamcrest-core (1.3)	2.1, 2.2
org.hamcrest:hamcrest-library (1.3)	2.1, 2.2
(t) org.hamcrest:hamcrest (2.1)	2.1, 2.2

B Precision when including time (RQ2)

Table I does not take time into consideration, and in case one is to query for a specific interval, it would not provide the wanted results. Thus, the TDPN created allows for querying on the structure given a time range and will provide only the dependencies that were released in that time frame. Table II shows the results.

TABLE II: Dependencies (direct and transitive (t)) of *junit:junit:4.13.2* after 2019

GroupID:ArtifactID (constraint)	**Versions**
org.hamcrest:hamcrest-core (1.3)	2.2
org.hamcrest:hamcrest-library (1.3)	2.2
(t) org.hamcrest:hamcrest (2.1)	2.2

Compared to the previous table, it only shows the 2.2 versions, as the 2.1 versions appeared in 2018 and therefore are not considered in the above query. Consequently, this also proves as a definite answer that time indeed improves the precision of the data structure. A TPDN is always going to improve on the accuracy of data over a PDN at the expense of some efficiency due to querying on snapshots. Moreover, querying can also be used to find the latest dependencies of a package on a given time interval by using the function "Find the latest dependencies of a package", which again performs a Breadth-First Search but which is followed afterwards by a filtering to select only the latest artifact. Therefore, using this method, one can easily replicate the results that are shown when listing the dependencies of a package through Maven's dependency manager.

[14]https://mvnrepository.com/artifact/junit/junit/4.13.2

Let:

- A be the set of transitive dependencies resolved by the package manager, Maven
- B be the set of transitive dependencies resolved using the implemented TDPN
- E be the number of dependencies that have a correct name but incorrect version

We calculate the accuracy of the algorithm by the following formula [18]:

$$Acc = \begin{cases} 1 - \frac{|A| - |(B \cap A)| + 0.5 * E}{|A|}, & \text{if } |A| \neq 0 \\ 1, & \text{otherwise} \end{cases} \quad (1)$$

Table III contains the accuracy values for the top 5 most used packages as defined by the Maven repository, where the data was manually downloaded as to not miss any nodes. Although *slf4j-api* and *scala-library* do not have any dependency, it is still beneficial to make sure that the graph does not attribute any as well. However, there is on dependency that was missing from the list received when calling for all the dependencies of the artifact *guava*, as one of the versions could not be resolved by the Maven translator, as the name was not specified according to the SemVer conventions. Performing the following formula on other parts of the dataset where dependencies nodes are missing might result in low accuracy values.

TABLE III: Accuracy value of a package's dependencies

GroupID:ArtifactID:Version	**Value**
junit:junit:4.13.2	1
org.slf4j:slf4j-api:1.7.36	1
org.scala-lang:scala-library:2.13.8	1
com.google.guava:guava:31.1-jre	0.833
org.mockito:mockito-core:4.6.1	1

C Analysis of packages (RQ3)

In this subsection, a few remarks are made regarding the most used packages from the subset collected, while also applying some metrics to measure criticality.

To obtain the most used packages from the dataset, the PageRank algorithm is applied. To get a list of unique results, such that an artifact's versions do not appear multiple times, a filter is applied on the graph that only keeps the latest package versions. Figure 5 lists the 10 most used packages using the PageRank algorithm. As the algorithm is non-deterministic, the results can slightly vary. Thus, this list was taken by averaging the results of the algorithm after 5 iterations.

Besides the PageRank algorithm, another metric to measure the use of the package is *Betweenness Centrality*[15]. This algorithm detects the amount of influence a node has over the graph by measuring how often a node appears on shortest paths. Figure 6 shows the 5 most used packages by applying the above described algorithm.

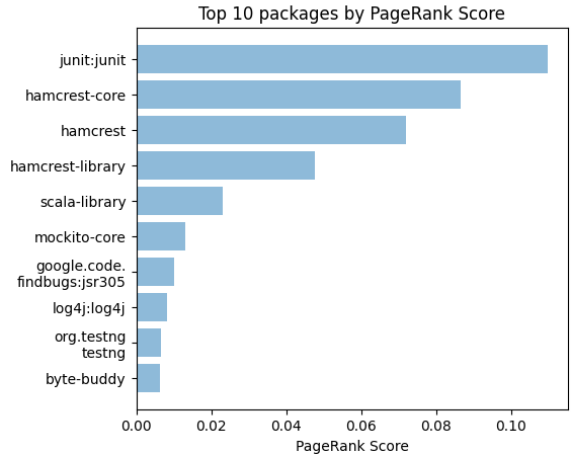

Figure 5: Top 10 most used packages ranked by PageRank

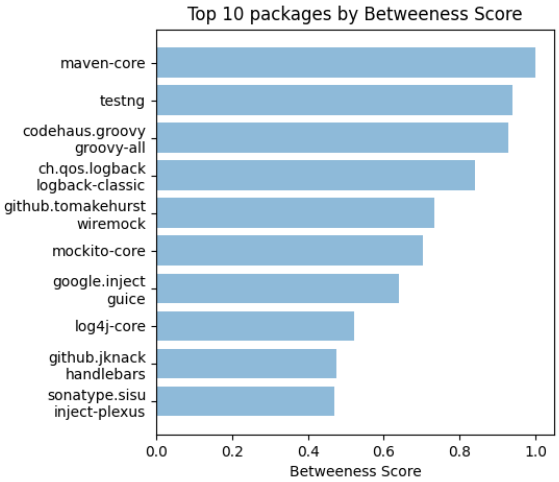

Figure 6: Top 10 most used packages ranked by Betweenness

Evidently, the results are quite different. Although some of the packages can be found in both tables, such as "*mockito-core*" or "*otg.testng:testng*", the 1st ranked package from Figure 6 cannot be found anywhere in the first 100 packages from Figure 5. This is because betweenness centrality is not suitable for ranking the large scale networks, while it also focuses more on measuring how many shortest paths include a certain node. PageRank, however, insists more on scoring higher nodes that are linked by other high score nodes. Note that there is also a difference in the distribution of the results. In the PageRank calculation, the ranking drops much quicker

[15]https://neo4j.com/docs/graph-data-science/current/algorithms/betweenness-centrality/

as the first few nodes have much more importance, while the scores from the Betweeness Centrality algorithm do not have such a sudden drop, indicating there are more packages that are used in these shortest paths calculated.

D Analysis of packages depending on time ranges (RQ)

Now that the previous research questions have been answered, the initial research question this paper revolved around can also be resolved, that is, **What are the most used packages on given time frames**? Figure 7 can answer exactly this. The figure plots PageRank's ranking performed on time ranges starting from 2005 up to 2022. For instance, it computed the algorithm for the period 2005-2010, 2005-2013, 2005-2016 and so on. The reason why 2005 was chosen as a starting year is because before that, not many packages are recorded. Many packages that are believed to be from before that period did not have a release date and thus, were not considered in the computation. Moreover, if a package's position is last in the graph, so ranked 20th, it means that the package did not appear at all in the top 20 most used packages on the specific time frame.

By analyzing Figure 7, a few interesting trends show up:

1. ***junit*** is once again top of the list, except for the 2005-2019 period. That is because after 2015, not a single other stable version was released until 2020. Thus, although the graph still calculated the version released in 2014, it was still not popular enough to appear in the top 20 packages used over that period.

2. ***slf4j-api*** was the runner-up of most used artifacts up until 2019. This decline is clearly due to the incident involving *log4j*, which is explained briefly in section I.

3. ***mockito-core*** and ***easymock*** are packages that provide mocking for testing. A clear trend can be seen here that *easymock* was the preferred mocking package up until 2016, when *mockito-core* made its way around and became more and more popular. After the rise of *mockito-core*, *easymock* has slowly dropped outside the top 20 most used software.

4. ***hamcrest-core*** is probably the least expected software to appear on the list, but with the rise of *junit*, *hamcrest* also picked up places in the rankings, especially in the latest years. Given the smaller dataset and how PageRank looks ranks high nodes that are linked to other important nodes, this is no surprise.

For these reasons alone, it is clear as to what the time dimension adds to the perspective, and although much of it relates to the popularity of the simple "download count" metric, it is more complex than that.

V Responsible Research and Reproducibility

A Ethics

The field that is covered by this paper generates minimal ethical risk, except for a few points. Thus, all the sources from which the data was acquired provide licensing agreements which have been well respected. Some sources specified how the data should be used and that has been done accordingly, as well as citing the work used as required. AS the research did not contain any human input or other sources that needed consent, there are no other involvements that cause ethical risk.

B Reproducibility

In regard to reproducibility, everything can be redone with ease by following section III and IV. To obtain the same results, it is a requirement that the same dataset of unique artifacts needs to be used. The dataset of used packages can be found here: https://zenodo.org/record/6653542. It contains a JSON file that contains all packages that were put in the graph and analyzed.

However, the first research sub-question can be interpreted differently if one desires. The graph structure can be done following an aggregated nodes approach, while the querying can be performed with using a different model from the snapshot-based one. If the same approaches and models are used, then the results gathered should be the same if you allow for a very small margin of error. When answering the third research sub-question, different algorithms can be applied and thus different results can be obtained. If one is to reproduce the work done here, it should use the same libraries, such that algorithms like PageRank do not differ slightly. The source code of the libraries used are publicly available online and can be found in the footnotes that mention them.

All code used to obtain the data and process it for the graph can be seen in the following repository: https://github.com/DenisCorlade19/maven-package-metadata. For everything related to the graph construction and algorithm, the code can be found here: https://github.com/DenisCorlade19/SoftwareThatMatters

VI Discussion

In the following section, a more in depth explanation of the results is given, highlighting their practical implications. Moreover, a comparison between the results and other related work's results is showcased, and further possible improvements are mentioned. Finally, the section will outline some of the limitations that emerged.

A Results

Figure 5 provides an interesting finding when it comes to vulnerability, and that is the 8th package which comes as most important, log4j. This package is the widely known artifact

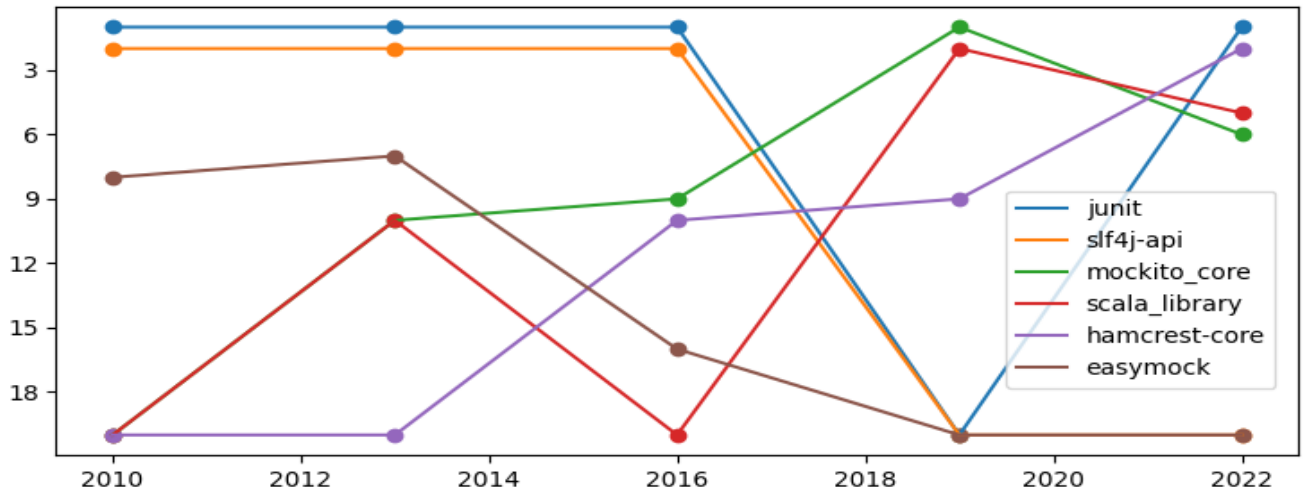

Figure 7: Pagerank ranking for some popular packages throughout different time periods

that was mentioned already in section I. It is the most critical, which might imply that it is very vulnerable. It shows that it plays a big role in the network constructed, and the fact that many packages depend on it should raise awareness to developers that the package should be put in an exclusion tag.

Another important point to take out from the results is the decline in usage of the package *slf4j-api* from around 2019 and afterwards, as seen in Figure 7. This is one clear improvement that the time component provides to the PDNs, as with the use of such a tool, one can observe that something is irregular for some reason and research the cause of it afterwards. Moreover, an interesting conclusion can also be drawn from the change of usage from *easymock* package to the *mockito-core* one. It highlights the change in the trend of what developers use at a specific time.

B *Limitations and Threats to Validity*

Due to technical limitations, a larger dataset could not be downloaded to be fed as input to the graph. To reproduce such an event of analysing the whole of Maven Central Repository would require enormous amounts of space and time, as the whole dataset would be circa 5TiB or more. Even if one would have the space and time for this, it is known that Maven itself does not permit this for unknown reasons. Therefore, it is possible that some metrics obtained from analysing the most used software are not entirely accurate, as it would require all 9M unique artifacts to be gathered.

When using extremely big datasets that have hundreds of millions of edges, the graph is not able to perform on a server with 64GB RAM. This is a limitation of the language that we use, more exactly, the graph network library that is *gonum*.

Furthermore, some packages cannot be analysed as they do not specify any versions or are halted. In similar fashion, there are some dependencies that do not follow a regular format and specifying the versions they depend on is not achievable, thus leading to their exclusion.

Moreover, comparing the results to the download count measure might not be of extreme interest, as the download statistics do not represent how much an artifact is being used. For instance, many companies are using repository managers, which means that an artifact is being downloaded exactly once but internally used a lot. On top of that, even though some artifacts are counted as dependencies through transitivity, it might not mean that they are used for sure. Therefore, some correlations between the results and the download count might not represent the reality.

Finally, the dataset collected is strictly limited to Maven Central repository, and thus the results should not be generalized over the whole ecosystem of Java, as there exist 32 other repositories as of June 2022.

VII Conclusions and Future Work

The paper has one major contribution which is adding the possibility of querying on different time ranges in a package dependency network, more precisely, a TDPN. Moreover, it provides an insight into the ecosystem of Maven, its dimensions, and how vulnerability can be observed as a potential side-effect of showing software usage over time. Besides, the work presented how an increasing number of transitive dependencies might add on to the risk of becoming vulnerable to a threat. Finally, the paper also goes over the increasing growth of the Maven Central repository and the various trends emerged from what developers use.

As this paper solves the most used software over time, it reflects that the top packages by download counts as of now still have some correlation, as *junit* is also considered the most used by the algorithm implemented. However, it shows interesting trends and changes from one package to another, while also showing periods where certain packages were not used anymore due to not releasing new stable versions, or by simply becoming vulnerable.

There are multiple possibilities to extend this work. First

of all, efficiency of the graph could be improved. Brief analysis showed that perhaps Rust would prove to be more fitting language for such an implementation, as the *GoNum* library has its limitations when reaching bigger input. Moreover, no comparison could be done between the graph implemented in this paper and other previous work. The grounds of information are already laid out in section B, but it would remain to be seen this comparison would actually result in much more accurate results by the graph implemented here. The paper proves the correctness by comparing to the ground truth which is the repository manager, but it does not fully cover the whole scope. Another improvement would be for the graph to be able to take data based on different scopes. Maven has 6 different types of scopes, and their uses are to limit the transitivity of the dependencies. The current implementation does not filter on any scope, and thus shows all the transitive dependencies that there can be.

REFERENCES

[1] Libraries.io - the open source discovery service. https://libraries.io/.

[2] Stack overflow developer survey 2021. https://insights.stackoverflow.com/survey/2021.

[3] What is software dependency hell. https://www.boldare.com/blog/software-dependency-hell-what-is-it-and-how-to-avoid-it/, nov 26 2019. [Online; accessed 2022-06-15].

[4] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the apache community upgrades dependencies: An evolutionary study. *Empirical Softw. Engg.*, 20(5):1275–1317, oct 2015.

[5] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, page 280–289, USA, 2013. IEEE Computer Society.

[6] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. The maven dependency graph: A temporal graph-based representation of maven central. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 344–348, 2019.

[7] John Biggs and Ben Popper. What's so great about go? https://stackoverflow.blog/2020/11/02/go-golang-learn-fast-programming-languages/, Nov 2020.

[8] David F. Gleich. Pagerank beyond the web. *SIAM Review*, 57(3):321–363, 2015.

[9] Raphael Hiesgen, Marcin Nawrocki, Thomas C Schmidt, and Matthias Wählisch. The race to the vulnerable: Measuring the log4j shell incident. *arXiv preprint arXiv:2205.02544*, 2022.

[10] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 102–112, 2017.

[11] Raula Gaikovina Kula, Coen De Roover, Daniel M. German, Takashi Ishio, and Katsuro Inoue. A generalized model for visualizing library popularity, adoption, and diffusion within a software ecosystem. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 288–299, 2018.

[12] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, May 2017.

[13] Andrea Ligios. Use the latest version of a dependency in Maven. https://www.baeldung.com/maven-dependency-latest-version, oct 11 2018.

[14] Masterminds. v3 - Constraints don't appear to work as expected · Issue 150 · Masterminds/semver. https://github.com/Masterminds/semver/issues/150. [Online; accessed 2022-06-08].

[15] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: a review of industrial studies. *Empirical Software Engineering*, 12(5):471–516, Oct 2007.

[16] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, Oct 2018.

[17] Tom Preston-Werner. Semantic versioning 2.0.0. https://semver.org/spec/v2.0.0.html.

[18] Andrei Purcaru. Analyzing the effect of introducing time as a component in python dependency graphs, 2022.

[19] Consumer Reports. Equifax data breach affected 2.4 million more consumers, Mar 2018.

[20] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: A case study (at google). In *International Conference on Software Engineering (ICSE)*, 2014.

[21] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. The emergence of software diversity in maven central. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, page 333–343. IEEE Press, 2019.

[22] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering*, 4:1–15, 12 2019.