



**Finding most used software application  
by using a time-dependency graph**

**Alexandru Dumitru**

**Supervisor(s): Georgios Gousios, Diomidis Spinellis  
EEMCS, Delft University of Technology, The Netherlands**

**A Dissertation Submitted to EEMCS faculty Delft University of Technology,  
In Partial Fulfilment of the Requirements  
For the Bachelor of Computer Science and Engineering  
22-6-2022**

## Abstract

Using open-source packages when developing software applications is the general practice among a vast amount of software developers. However, importing open-source code which may depend on other existing technologies may lead to the appearance of a transitive dependency chain. As a result, failure of packages with a high amount of transitive dependants may have an impact on the performance of all dependant applications. This work focuses on designing a graph data structure which maps the dependency relationship between packages as edges, with nodes representing a single version of a certain package. Moreover, the data structure may perform queries based on time intervals, being able to resolve versions in the same manner as a package distributor would. By constructing such data structure, an analysis of the most critical packages for an ecosystem can be conducted. This paper looks mainly into the Debian ecosystem, and searches for applications which are most critical. Based on this paper criticality evaluation, the package `dpkg` was found to be both most critical and most used in whole Debian's package main repository.

## I INTRODUCTION

A vast majority of open-source code can be imported effortlessly into software through different package managers such as `npm`<sup>1</sup>, `pip`<sup>2</sup> or `Maven`<sup>3</sup>. As a result, software engineers are greatly encouraged to use already existing tools in their code. However, depending on existing packages may not be as risk-free as one may consider it to be. Recent studies have revealed that there is a high increase in the number of *transitive dependencies* between packages in programming languages such as Java or Javascript, which may lead to programmers adding vulnerabilities or conflicting software unintentionally [14].

Consequently, the rise in the number of transitive dependencies leads to the appearance of so-called "fragile" packages, which may result in high vulnerability issues [10]. In the past few years, multiple renowned cases of dependency-network failures have emerged, such as:

- The Equifax Data Breach incident, in which the data of over 100.000 credit cards was leaked due to an outdated dependency. Their system of handling credit credentials was based on the Apache Struts framework. After a key vulnerability patch was released by Apache Struts, the developers highly encouraged the users to update their dependency as swiftly as possible. However, the developers from Equifax failed to change the dependency in time, thus leading to the data breach event [12].

<sup>1</sup><https://www.npmjs.com/>

<sup>2</sup><https://pip.pypa.io/en/stable/>

<sup>3</sup><https://maven.apache.org/>

- The case of the Log4j library. In December 2021, the Log4Shell vulnerability was discovered, which is estimated to have affected around 4% of the Maven Central, either through direct or transitive dependencies. The vulnerability allowed the attackers to execute code remotely by injecting a string directly into the Log4j library [13].

In addition, these issues may affect and disrupt software at build or run time. Research has shown that, in C++ and Java ecosystems, the most common reason why application fail at run time is dependency related, making up for around 52.68%(C++) or 64.71% (Java) of total cases [20].

As it can be seen, identifying critical packages and software dependencies is imperative for assuring that future developed software works as intended when using imported code. Figure 1 generalises the main problem of transitive dependencies. Suppose at timestamp  $T_1$  the package  $A_{1.1}$  is released. Then, at timestamp  $T_2$  the package  $B_{1.1}$  is deployed, depending on  $A_{1.x}$ , which will always select the highest dependency of  $A$ . As there is only one version of  $A$  at  $T_2$ ,  $B_{1.1}$  will depend on  $A_{1.1}$ . However, at  $T_3$ , as a new version of  $A$  is released,  $B$  will change its dependency from  $A_{1.1}$  to  $A_{1.2}$ . Subsequently, a new package  $C_{1.1}$  is deployed at  $T_4$ , which depends on  $B$ . As it was established before that  $B$  depends on  $A$ ,  $C$  depending on  $B$  means that  $C$  will also depend on  $A$ . Presumably, at  $T_5$  the new version  $A_{1.3}$  is deployed. The release will force both  $B$  and  $C$  to update their dependency. As progressively more versions are deployed, tracking each constraint becomes a convoluted task.

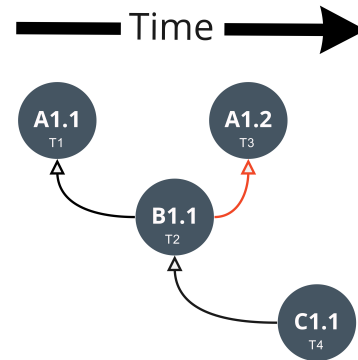


Figure 1: Graph displaying the transitive dependency problem. The edges represent the dependency relationship between packages.

The scope of this research is to build a time dependency graph in which the dependency relationship between software packages is being displayed. By doing so, critical applications can be easily identified and analysed. In order to reach this goal, 3 main research questions have been proposed, which are going to be addressed in this paper. The main questions are:

- **RQ1: What would a graph data structure for package dependency look like?**
- **RQ2: Does the introduction of time increase the precision of the data structure?**

- **RQ3: What are the most critical software applications?**

The first 2 questions deal with the actual implementation of the data structure, while the third one will use the data structure to analyze the criticality of software applications. Question 1 will answer and provide the implementation details of the data structure, whereas Question 2 will analyze the impact of time with regard to the precision of the transitive dependencies.

In the following sections, the Background of the problem will be firstly presented and established. Subsequently, the main Methodology will be presented, including the work behind gathering data and implementing the temporal graph data structure. Afterwards, in the Results section, the outcome of the data structure results will be included and reflected upon. Finally, there will be a section regarding the Responsible research conducted during this study, followed by a Discussion section and concluding with the general Conclusion of the paper.

## II BACKGROUND

### A Terminology

Before going further, certain terms must be defined in order ensure the clarity of the paper. Therefore, this section will clearly explain the most relevant terms for this work.

A **package** represents a piece of distributable software, which may contain either executable code or source code. Packages that contain source code are called **libraries**. The code inside libraries needs to be compiled first in order to be executed. Their main objective is to allow developers to share modular code, meant for code reusability. Contrastingly, packages which contain executable code are defined as **software applications**, and can be executed without the process of compilation.

When a software imports and uses a package, a **direct dependency** between those 2 emerges. However, if the parent (the package that is depended upon) has other dependencies on its own, then the relationship between the child (the dependent package) and those dependencies is called **transitive dependency**.

A network of packages and dependencies may be called a package dependency network(**PDN**). When the element of time is inserted into this system, it may be called temporal package dependency network(**TPDN**). The advantage of using TPDN's instead of PDN's is that, given a timestamp, the structure may represent the latest dependency graph structure available for that time interval.

### B Related Work

In this section, related work relevant to the subject will be presented and discussed. Similar efforts have already been achieved in this field, thus analysing them may present another perspective of the problem.

The website Libraries.io [4] collects package data containing dependencies, dependents and version from multiple

package distributors such as npm, Maven and pip. Nevertheless, due to how the dependencies between different packages are stored and distributed, it is difficult to track how different dependencies change through time (recall the problem described in Fig 1). Therefore, while the data provided by Libraries.io may be useful when inspecting the most used packages, it does not provide the relevant information to build a TDPN, which is the scope this work.

Implementing graphs which have a temporal element is a relative new field. This subject is addressed in [21]. This work presents different approaches of how one could implement a temporal graph. The approach that will be considered and presented in this work is similar to the snapshot graph model described in the paper previously mentioned. The snapshot graph model considers a temporal graph to be a subset of static graphs that have different edges drawn, depending on the time variable. Given time  $t$ , the snapshot will select the  $t_{th}$  static graph. Nonetheless, this model faces drawbacks as well. The main disadvantage described is the vast amount of space needed to store each version of the static graph that is to be used. As mentioned previously, a similar approach based on time intervals will be described in this paper, which can be found under section III.

A similar implementation of a TDPN structure can be seen in [8]. The paper looks on how to create a snapshot of the Maven Repository represented by a dependency graph. The snapshot of the repository is taken on the 6<sup>th</sup> of September, 2018. The JVM repository around that date was estimated to include around 2.8 million artifacts. Currently, the size of the JVM is around 9 million artifacts, and can be tracked here<sup>4</sup>. The main limitation of this work however, is that querying dependencies may return conflicting ones. Therefore, the paper claims that "some metrics like libraries usage and dependencies may not reflect the reality". The structure of a TDPN should always resolve to the right dependencies given a time interval. As a result, the paper described does not solve the main issue presented here. Nevertheless, the paper provides a custom Neo4j Docker image with the whole data set mapped, containing example queries, which may be used as guidance for this work.

Finally, the Ultimate Debian Database (UDD) should be mentioned [15]. The project aims to deal with the 'data hell', by providing a SQL warehouse database which aggregates debian packaging data. By its nature, it cannot be used to solve the problem describe in this paper, however data from their service may prove to be useful for our case.

## III METHODOLOGY

As previously described, the base approach to solve the problem previously considered in this paper is to create a TPDN based on a graph data structure, in which the nodes would represent the actual package, and the edges would depict the dependencies between them. This section will delve into the main steps taken during the research to implement a TPDN. The process of developing the data structure can be split upon

<sup>4</sup><https://search.maven.org/stats>

two main phases: gathering relevant data to populate and test the structure and developing the actual time dependency network.

### A Research clarification

Before presenting the phases of the methodology, a succinct description of each Research question will be given, to clarify their role in this work.

**RQ1:** As seen in Section II, there are only scarce information regarding the implementation of TPDN's, especially when dealing with the dependency hell [11]. Therefore, the paper will provide details regarding how the data structure is implemented, and how the element of time is being considered in its ecosystem.

**RQ2:** As this research questions argues that the introduction of time will increase the precision of the structure, there needs to be a certain procedure to check for the precision of the results. Firstly, by precision it is meant that the data structure reflects the ground truths assuredly. Thus, a way to measure the precision of the data structure would be to compare the results with other existing verified sources.

**RQ3:** The final scope of this paper is to find the most critical software application. In order to measure the criticality of one package, the *PageRank*<sup>5</sup> algorithm will be used, in addition to other main factors, such as popularity (measured by download count) and known vulnerabilities/bugs.

### B Data Collection

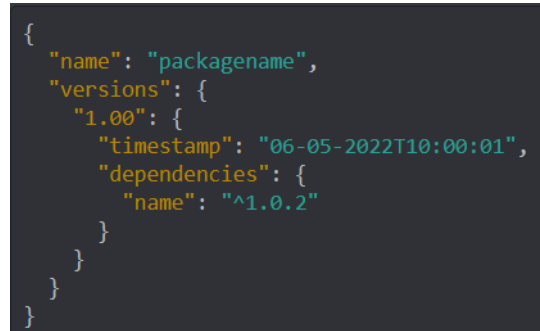
Meta-data regarding packages was collected by my peers from 4 main package distributors: npm, Maven, Pypi and Debian package manager.<sup>6</sup>The meta-data contains information regarding the dependencies, release dates and release versions of each package. The structure of the aggregated data can be seen in Fig 2. However, the scope of this research is to identify critical applications, which means that a way to filter out libraries from the gathered data must be determined. In order to check whether or not the package contained a library or an actual application, the approach chosen was to identify the packages that install binary files, by checking the `/bin` directory.

For the Debian package manager, the command `apt-file -l search /usr/bin` was used to find all binary packages which contain files under the `/usr/bin` directory. Furthermore, in order to remove remaining libraries that may have auxiliary binaries, a filter based on the tag of the package was executed. The filtering checks if the package is tagged as a shared library, and removes the packages that satisfy this condition. As a result, the remaining packages are classified as a software application.

For the other package distributors, the Debian data collected was used as a superset to find applications in specific languages. The main Debian repository hosts the data regarding packages per language. As example, for Python, the list

of all Python related packages can be found here<sup>7</sup>. Therefore, in order to find applications related to another programming language, it is sufficient to perform a difference set operation between the set of applications found antecedently in Debian, and the set of packages which are grouped under the programming language inside Debian's main repository.

However, in order to find past applications, older versions of Debian must be checked. For the scope of this research, the stable, oldstable and oldoldstable were used to gather data for each programming language mentioned above.



```
{
  "name": "packagename",
  "versions": {
    "1.00": {
      "timestamp": "06-05-2022T10:00:01",
      "dependencies": {
        "name": "^1.0.2"
      }
    }
  }
}
```

Figure 2: Structure of the aggregated data

### C Graph Structure

In order to model the dependency relationship's between packages, a directed graph-like data structure was chosen, where each node represents a version of the package and each edge denotes that there exists a dependency between 2 packages. The main reason behind choosing to represent a node as a package version instead of an actual package is that multiple versions of the same package may be needed at a certain timestamp. When querying the structure, a time interval will be given, which will be used to select the edges that corresponds to the latest correct version of the parent node. An example can be seen in Figure 3, where 3 versions of the package *A* exists. Package *B* depends on the latest version of *A*<sub>1.x</sub>, thus there are 3 edges from node *B* to node *A*. At query time, depending on what time interval is given, the edge analogous to the latest version respecting the interval will be selected as the corresponding dependency.

The data structure was implemented in Golang<sup>8</sup>. This programming language was firstly recommended by the supervisor to be suitable for the task. Some of its main benefits can also be found in [9]. For the implementation of the nodes, the package *gonum*<sup>9</sup> was used, which provided an interface for creating simple directed graphs with unique ids. For the purpose of reducing the amount of information stored inside the data structure, a map was used to associate the id of the node with the actual object that meta-data. The object structure can be seen in Figure 4. As it can be observed, the object holds 5

<sup>5</sup><https://neo4j.com/docs/graph-data-science/2.1/algorithms/page-rank/>

<sup>6</sup><https://www.debian.org/distrib/packages>

<sup>7</sup><https://packages.debian.org/bullseye/python/>

<sup>8</sup><https://go.dev/>

<sup>9</sup><https://pkg.go.dev/gonum.org/v1/gonum/graph>

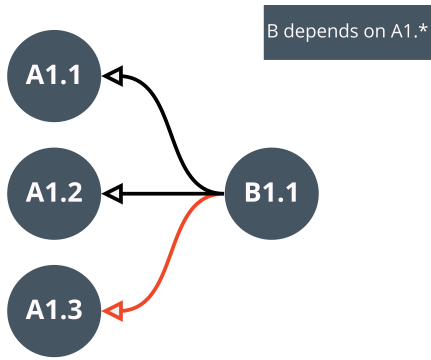


Figure 3: Graph displaying multiple versions of package  $A$  and how the dependency between  $B$  and  $A$  is represented

fields. The id field is the node id generated by the gonum library, which creates a unique number identifier to each new node. The name and version field equate to the name and version of the package which is represented by the node. The timestamp field holds information regarding the release date when the package was deployed. Finally, the isApplication field holds a boolean value, which is used to mark applications apart from other existing packages.

```

type NodeInfo struct {
    Timestamp    string
    Name         string
    Version      string
    id           int64
    IsApplication bool
}
  
```

Figure 4: The information of a node

Now that the node structure has been presented, the edge construction will be discussed next. Edges are created based on each package dependency. The dependencies can be found in the aggregated data seen in Figure 2. One challenge that comes with parsing the dependency is reading the semantic versioning of each package. Different package managers may have different semantic versioning syntax. As mentioned in Section B, the data gathered for this research comes from 4 main package distributors, namely npm, Maven, Pypi and Debian package manager. Therefore, in order to parse the dependencies of each package, the semantic version rules of each package manager are needed to be checked. Python packages adhere to the semantic versioning described by semver[17]. The version number of a package is formatted in the structure of MAJOR.MINOR.PATCH, and the constraints are written with comparison signs:  $<$ ,  $>$ ,  $=$  (Ex: " $<= 1.0.0$ ").

Debian, npm and Maven have different conventions, but share similar formatting rules. The rules for Maven are described in [16], whereas npm's versioning rules can be found in [1]. The main difference of Maven is that

it may use intervals to express dependency constraints. For example, the constraint " $<= 1.0.0$ " would be represented as " $[,1.0]$ " in Maven. Regarding npm, the main distinction is that the dependency constraints may use the  $\sim$  (tilde) and  $\wedge$  (caret) characters. Concerning Debian, the rules for version naming can be found in [2]. Its main dissimilarity from the rest consists in the name structure. The format which Debian packages adhere to is  $[\text{epoch:}] \text{upstream\_version} [-\text{debian\_revision}]$ , as opposed to the MAJOR.MINOR.PATCH, which is used by the other package distributors.

In order to parse the semantic versioning of the dependency constraints, the library semver<sup>10</sup> by Masterminds was used. The library provides an interface to parse semantic versions and constraints which adhere to the rules described in [17]. In addition, it is also able to recognise and correctly interpret npm's specific characters,  $\sim$  and  $\wedge$ . For Maven specific constraints, a translator was implemented which transforms the Maven-specific syntax into syntax which can be parsed by the semver library. Regarding Debian packages and constraints, the same approach could not be repeated. As Debian packages follow different semantic rules, a separate interpreter was implemented, which could parse Debian package names, in addition to the constraint semantic. The implementation was based on the one found in [19].

As the structure and functionality of TPDN was presented, an answer to **RQ1** can now be formulated:

A **TPDN** is defined as a directed graph  $G = (V, E)$ , where  $V$  represents the set of all package versions and  $E$  represents the set of all time-dependent edges. An edge  $e = (a_{v_1}, b_{v_2})$  represents that there exists a direct dependency between version  $v_1$  of package  $a$  and version  $v_2$  of package  $b$ . Given a time interval  $[t_i, t_j]$ ,  $e(a_{v_1}, b_{v_2}) \in E_{[t_i, t_j]}$  if and only if:

1.  $v_1$  and  $v_2$  are the latest released versions of  $a$  and  $b$ , with regards to  $[t_i, t_j]$
2.  $e$  is the only edge which maps  $a$  to  $b$

As a result, given a time interval  $[t_i, t_j]$ , a TPDN will return a static graph which reflects the dependency network throughout the period given.

## IV RESULTS

In this section, the main results of the research conducted will be presented and discussed. Consequently, a validation of the results will be shown, in order to ensure the integrity of the work done.

### A Graph Construction

A total of 86376 packages were collected from the Debian repositories. From those, 15807 packages were marked as applications, by using the criteria described before in the Data Collection section. Due to the fact that, as of now, packages which were published before 2015 are archived, it is impossible to perform the same check to find applications, as the

<sup>10</sup><https://pkg.go.dev/github.com/Masterminds/semver/v3>



file-paths of the packages are unavailable. Therefore, all data presented here will have the earliest timestamp set in 2015.

When building the Debian graph, around 250k nodes were created, each of them representing a certain version of a certain package. The edge count linking the nodes was reported to be around 5.5million. Figure 5 shows a picture of the whole Debian environment, generated by Graphia<sup>11</sup>. The image was generated by a .dot file, created from the graph constructed.

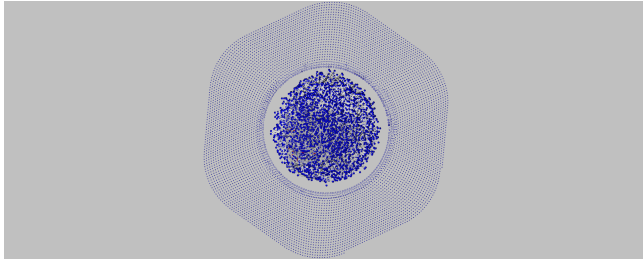


Figure 5: Visual representation of Debian's graph

## B Precision & Validation

In order to check the precision of the described data structure, multiple queries were performed, and compared to the results returned by the apt repositories. One may check the dependencies of a package in a Debian system by calling apt depends 'pkgname'. Therefore, a first check will be to see if the dependency requirements are parsed as expected. The package curl - 7.74.0-1.3+deb11u1 was used as a benchmark. Table I shows the direct dependencies requirement of the package curl, and how the graph structure resolves its prerequisites. As it can be observed, all versions returned by the TPDN actually adhere to the rules implied by the dependency requirement. The results were generated by performing a Breath-First Search, starting at a node provided by input(curl in this case), and returning a list of all nodes found, which corresponds to the list of direct and transitive dependencies of the package.

However, the results show in Table I do not take time into account. As a TPDN allows to query the structure given a time interval, more precise results can be obtained. In order to find the actual versions which the Debian package manager chooses, a query based on the latest dependencies available must be done. This can be achieved by performing a Breath-First Search, in addition to executing a filtering and selecting only the latest version which satisfies the time constraint. The marked versions in Table I shows which versions will be selected based on the given requirement. In addition, it only selects the most recent version which satisfies the time constraint. Thus, this results genuinely reflect the versions which are chosen by the Debian package manager.

The number of transitive dependencies returned by the data structure must also be compared to the results returned by Debian's package manager. Therefore, the accuracy of the data

Table I: Curl dependency requirements and the resolved versions returned by the TPDN.

Dependency Requirement	Resolved Versions
libc6 (>= 2.17)	2.19-18+deb8u6 <b>2.31-13+deb11u3</b> 2.28-10 2.24-11+deb9u3 2.31-13+deb11u2 2.24-11+deb9u4 2.24-11+deb9u1 2.19-18+deb8u1 2.19-18
libcurl4 (= 7.74.0-1.3+deb11u1)	<b>7.74.0-1.3+deb11u1</b>
zlib1g (>= 1:1.1.4)	1:1.2.8.dfsg-5 1:1.2.11.dfsg-1 <b>1:1.2.11.dfsg-2</b> 1:1.2.7.dfsg-13 1:1.2.8.dfsg-2+b1

Table II: The final versions of curl dependencies chosen by TPDN based on the latest timestamp possible.

Package	Accuracy score
curl - 7.74.0-1.3+deb11u1	0.91
dpkg - 1.20.9	0.95
apt - 2.2.4	0.84
perl-5.32.1-4+deb11u2	0.87

structure will be computed next. We define accuracy as [18]:

Let:

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

We calculate the accuracy of the algorithm by this formula:

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

To evaluate the overall accuracy of the data structure, the accuracy of 4 packages which contain applications will be computed. In order to get the list of transitive dependencies from Debian, the command apt-rdepends 'pkgname' will be used, which searches recursively for all dependencies of the package provided as input. In Table II, the results computed can be observed. The average of the results is equal to 0.8925. From the Debian dataset, some new package versions were missing, thus impacting the overall score. For example, the gcc-12-base was a recurring package returned by the Debian resolver when querying for transitive dependencies. However, the latest version in the data set was gcc-10-base, thus lowering the accuracy score. Nevertheless, most versions returned by the TPDN coincided with the ones solved by the Debian package manager, resulting in a relatively high accuracy score.

<sup>11</sup><https://graphia.app/>

As we have seen that effect of introducing time in a query on a TPDN structure, a clear answer for **RQ2** can now be given:

By introducing a **time interval** to a query, a TPDN is able to resolve the version which a package manager will also choose at the specific time. Thus, we can say that adding the element of time does indeed increase the precision of the data illustrated, as it correctly reflects the state of the package network at the given timestamp.

### C PageRank

In order to find the most relevant packages which contain applications, the PageRank algorithm was used to rank them. Before applying the algorithm, each dependency was resolved to only consider the latest version of a package, leading to the graph being pruned. The top 10 results sorted decreasingly by the PageRank Score can be observed in Figure 6. As PageRank is a non-deterministic algorithm, multiple results may appear when running it on the same data sample. Thus, the results seen in the plot represent an average of 5 different PageRank sessions. The same ranking between packages was always maintained for each result returned by PageRank.

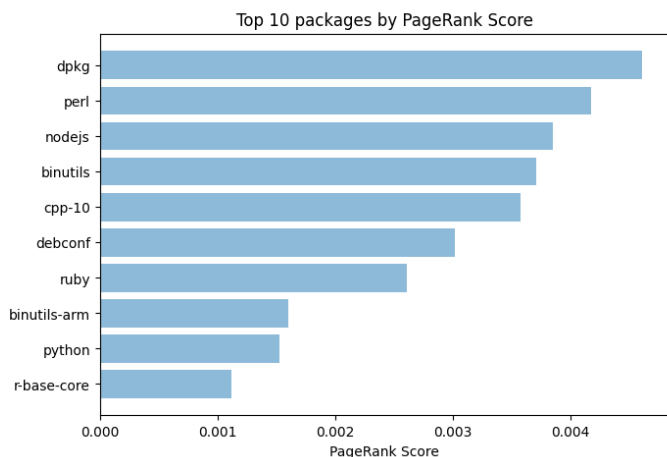


Figure 6: Pagerank Score of the top 10 packages which contain applications

In addition, multiple PageRank sessions were applied to observe how the ranking of the most popular packages would evolve through time. Figure 7 plots the ranking of the most popular packages, from 2016 to 2022. For each year, a PageRank algorithm was applied on the time interval [2015, 20xy], to find the most popular packages in that frame of time.

The main trend that can be observed is how the package `nodejs` becomes more relevant (according to PageRank values), as time goes on. An argument for this tendency may be that Javascript became a prominently used language in the last few years, ranked as the most used programming language for

the 9<sup>th</sup> time in 2021 by Stack Overflow users [7]. As `nodejs` is a package vital for a vast amount of Javascript application, it seems natural that its PageRank score increases, as more Javascript packages are released.

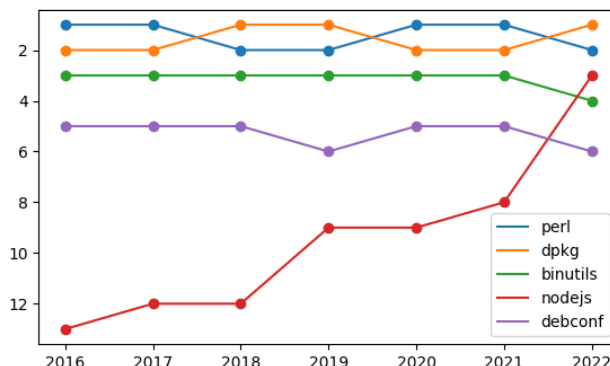


Figure 7: Evolution of the top 5 packages through the years

First and second place are predominantly switched between the package `dpkg` and `perl`. Both represent crucial packages needed by the Debian ecosystem. The package `dpkg` installs the actual package manager for Debian, which is responsible for installing, updating and deleting packages from the system. Therefore, it is expected to see this package ranked the highest by PageRank, due to its role in the Debian architecture. Regarding `perl`, some amounts of Debian’s core functionality are written using this language, in addition to many other libraries and applications [6]. As a result, `perl` represents a core package for Debian’s ecosystem, which explains its high PageRank score.

Finally, both `binutils` and `debconf` can be seen to have an almost constant ranking throughout the years. Both packages contain major programs needed to run and execute other applications installed. The package `binutils` contains a bundle of applications needed to “assemble, link and manipulate binary and object files” [5], whereas `debconf` holds a configuration management system for Debian packages. Consequently, both have a constant high PageRank score throughout the whole time interval chosen.

### D Criticality evaluation

As seen in the previous section, the PageRank algorithm may provide a fine ranking between packages. However, there exists other factors which may increase or decrease the criticality of a package. For the scope of this work, two other factors will be considered to evaluate the criticality of Debian applications, namely their **priority** and their **popularity count**.

Priority can be measured by the tag given to a package. Each Debian package has a priority tag corresponding to how vital the package is for the system. There are 4 possible tags: required, important, standard and optional. The order in which they are listed corresponds to their importance. Packages tagged as required or important may, as a result, have a higher criticality than packages tagged as optional.

Table III: Priority and Install count measurements of the top 5 packages ranked by PageRank

Package Name	PageRank Score	Priority Tag	Install Count
dpkg	0.004574	required	205719
perl	0.004186	standard	204012
nodejs	0.003851	optional	32892
binutils	0.003759	optional	134730
cpp-10	0.003624	optional	72852
debconf	0.003021	required	205720

For the popularity count, the Debian Popularity Contest results will be used [3]. The project aims to track statistics regarding the install, usage and upgrade count of Debian packages. However, Debian users are required to give their consent by installing the package `popularity-contest`, in order to participate in the survey. Thus, the data collected represents only a subset of the actual figures. Nevertheless, it can still serve as a benchmark for comparing the criticality between different applications.

Table III shows the priority tag and install count of the top 5 packages ranked by PageRank. In order to assess their criticality, both priority and installation count should be taken into account, besides the PageRank score. Thus, a criticality formula which takes into account these 3 variables must be formulated.

Let:

- $PR(p)$  be the normalized PageRank score of the package  $p$
- $I(p)$  be the normalized installation count of the package  $p$
- $w_{priority}(p)$  be the weight of the priority tag, which can be a value from the set  $\{1, 0.75, 0.5, 0.25\}$ , depending on  $p$ 's priority.

The criticality of a package could be evaluated as:

$$Cr(p) = \left( \frac{1}{2}PR(p) + \frac{1}{2}I(p) \right) \cdot w_{priority}(p)$$

In order to normalize the PageRank and install count values, the formula

$$x_n = \frac{x - x_{min}}{x_{max} - x_{min}}$$

will be used. For both PageRank and install count data sets,  $x_{min} = 0$ , thus the normalization will only consists of dividing every element by their respective  $x_{max}$ . The maximum value for the PageRank is equal to 0.004574 and is achieved by `dpkg`, whereas the maximum value for installation count is reported to be 205725.

The results of the criticality formula can be seen in Table IV. As it can be observed, the packages which have the highest criticality score are mainly the ones with high weighted priority. Due to the fact that packages tagged as required or important are vital not only for other packages, but also

Table IV: Criticality measurements of the top 5 packages

Package Name	PR normalized	Weighted Priority	I normalized	Criticality Score
dpkg	1	1	0.999	0.9995
perl	0.915	0.5	0.991	0.4765
nodejs	0.841	0.25	0.160	0.125
binutils	0.821	0.25	0.655	0.184
cpp-10	0.792	0.25	0.354	0.143
debconf	0.660	1	0.999	0.830

for the Debian's main functionality, they have a much higher criticality value, even if their PageRank value may be lower. Therefore, it can be seen that the package `debconf` has the second highest criticality score, even if it has a lower PageRank score. Packages such as `nodejs` which have a high PageRank score but a low weighted priority are expected to have a lower criticality score, due to them not being as needed for the system's main functionalities.

By performing this analysis, an answer to **RQ3** can now be given:

By the definition of criticality used in this paper, the most critical applications found in the Debian ecosystem are `dpkg`, `debconf` and `perl`. These packages are most critical for the Debian ecosystem, as multiple core functionalities of the system are based on them. Moreover, a high amount of optional packages depend on them, as reflected by the PageRank score.

## V RESPONSIBLE RESEARCH

In this section, the reproducibility of this work will be discussed, next to the ethical aspects involved during the research.

### A Reproducibility

All data gathered can be obtained in the same manner as described earlier in Data Collection section. The requirements for obtaining the relevant data is a Debian-based Linux system, which can access the Debian repositories through `apt` or `dkpg` commands. Regarding the implementation of the graph, the Graph Structure section should provide all information needed, for one to be able to implement a similar TPDN structure in any other programming language. Finally, for all plots which illustrated resulting data, the library `matplotlib` was used to produce the results shown. All code used in this research is open-source, and can be seen here.

### B Ethical Aspects

The research conducted and described in this paper has overall presented minimal ethical risks. All information gathered from outside sources has been rightfully accredited, through the use of references, footnotes, and adhering to the licences published by the free software used. Regarding the field of studies, as no human interaction was required to perform this research, no human related ethical issues emerged.



One final aspect which could have risen the ethical risk was the data acquisition. In order to avoid legal concerns, only data regarding the main division of Debian packages was gathered. Two more divisions exists, namely contrib and non-free, but the packages inside them do not follow the DFSG (Debian Free Software Guidelines) or they depend on packages which do not adhere to those rules.

## VI DISCUSSION

This section is dedicated mainly towards the main limitations faced during the research conducted. Moreover, a few concise remarks will be made about the results gathered in the previous section.

### A Limitation

One of the main technical limitation was not being able to gather data before the year 2015. As Debian archives their repository, it becomes harder to perform the application check for packages in older repositories, as the filepaths cannot be obtained as easily. In addition, as mentioned in the previous section, only the main free division of Debian packages was used as sample data. Perhaps gathering either non-free or contrib classified packages would change the results obtained. Nevertheless, the results collected still reflect the state of Debian's application dependency infrastructure.

Another technical limitation faced during the research consisted of the size and performance of the TPDN. When trying to fit higher amount of data which came from popular package managers, the graph would occupy a large amount of memory. In addition, querying based on time interval would take a considerable amount of time and computing power to return the expected results. The main reason behind this may be more at a design level. As TPDN creates a node for each version of a package, and multiple edges per single dependency, it becomes clear that memory becomes a fundamental problem, when trying to load a vast amount of interconnected packages with numerous versions. Thus, perhaps this implementation of a TPDN structure may not be suitable for fitting the entire data for popular package managers, like npm, Maven or Pypi.

Finally, it should be mentioned that, in Debian, there are relations between packages which are not modeled in the graph structure. A notable example is the *breaks* relation, which is used when 2 packages are not compatible with one another. This relation is not modeled in the graph presented, and there might be cases when a query returns a set of packages in which 2 are seen as incompatible.

### B Results

The results obtained from this work have been mostly discussed in section IV. The main remark which can be made is that the most popular applications in Debian are either system-related (like dpkg or debconf) or are based on a certain popular programming language (perl or nodejs).

Regarding the criticality ranking, one final note could be that the **priority** weight values may be changed to get different rankings. As priority is a discrete value, it may be hard to find arbitrary fair weights. For the scope of this paper, the weights were chosen to emphasize the importance of applications which both have a high PageRank score, and are tagged as required. In addition, time may also have a high impact of the overall criticality of a package, by influencing its PageRank score.

## VII CONCLUSIONS AND FUTURE WORK

This paper has mainly looked into how to map the dependencies of Debian packages which contain applications in a temporal graph data structure. A general explanation of the structure of a TPDN has been presented, in addition to showcasing how querying such a data structure functions. Moreover, the paper presents how the introduction of time influences the results returned by the graph data structure, reflecting the behaviour which package managers have when resolving dependencies. Finally, the paper's goal is to find the most important software application, which is achieved by analysing the Debian package ecosystem. The results show that the most important Debian application is dpkg, followed by perl and debconf, based on a criticality score which takes into account the PageRank score, the priority tag and install number of packages.

There are multiple ways in which this work may be improved. The main bottleneck of the TPDN has proven to be memory, as larger amount of data would not fit into the structure. Thus, perhaps a way to reduce the number of edges or nodes by grouping them would improve the performance of the data structure. In addition, querying based on a time interval could be improved or changed. The operation performs a filtering action on the edges, by first disconnecting the edges which do not respect the time interval. Subsequently, only the edge which leads to the latest version corresponding to the time interval is chosen, the rest being eliminated. In a future iteration of this work, the process of choosing the right edge should be rethought, to resolve the edges by not deleting others. As of now, if an operation based on a time interval is applied on the data structure, in order to use another query, the user would need to restart the application and rebuild the TPDN, which may take a considerable amount of time.

Nevertheless, the work presented in this paper could be considered valuable for the field of package dependencies, as it presents a new implementation of a dependency network, which takes time into account.

## REFERENCES

- [1] About semantic versioning — npm Docs. <https://docs.npmjs.com/about-semantic-versioning/>.
- [2] Control files and their fields - debian policy manual v4.6.1.0. <https://www.debian.org/doc/debian-policy/ch-controlfields.html#version>.

- [3] Debian popularity contest. <https://popcon.debian.org/index.html>.
- [4] Libraries - the open source discovery service. <https://libraries.io/>.
- [5] Package: Binutils (2.35.2-2). <https://packages.debian.org/bullseye/binutils>.
- [6] Perlfaq. <https://wiki.debian.org/PerlFAQ>.
- [7] Stack overflow developer survey 2021. <https://insights.stackoverflow.com/survey/2021>.
- [8] 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.
- [9] John Biggs and Ben Popper. What’s so great about Go? <https://stackoverflow.blog/2020/11/02/go-golang-learn-fast-programming-languages/>, 11 2020.
- [10] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1):381–416, February 2018.
- [11] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 463–474, 2020.
- [12] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. Software ecosystem call graph for dependency management. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, pages 101–104. IEEE, 2018.
- [13] 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.
- [14] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th Working Conference on Mining Software Repositories, MSR ’17*, pages 102–112. IEEE press, May 2017.
- [15] Lucas Nussbaum and Stefano Zacchiroli. The ultimate debian database: Consolidating bazaar metadata for quality assurance and data mining. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 52–61. IEEE, 2010.
- [16] Brett Porter. Dependency Mediation and Conflict Resolution - Maven - Apache Software Foundation. <https://cwiki.apache.org/confluence/display/MAVENOLD/Dependency+Mediation+and+Conflict+Resolution>, 01 2006.
- [17] Tom Preston-Werner. Semantic Versioning 2.0.0. <https://semver.org/>.
- [18] Andrei Purcaru. Analyzing the effect of introducing time as a component in python dependency graphs, 2022.
- [19] Romlok. Romlok/python-debian: Python modules to work with debian-related data formats. <https://github.com/romlok/python-debian>.
- [20] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers’ build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering*, pages 724–734, 2014.
- [21] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. Time-dependent graphs: Definitions, applications, and algorithms. *Data Science and Engineering*, 4(4):352–366, 2019.