Analyzing the effect of introducing time as a component in Python dependency graphs

Andrei Purcaru
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

The use of open-source packages is a common practice among developers. It decreases the development time and improves maintainability. But adding a dependency to a project comes with inherit risks such as introducing vulnerabilities. A few solutions that help visualize all of the dependencies of a project exist already. However, none provide the capability of selecting a moment in time for analyzing the generated structure. This research paper formalizes a time-based dependency graph that can be generalized to any ecosystem and then showcases its usefulness by analyzing the Python ecosystem throughout time. The results indicate that the Python ecosystem does have a subset of packages that are the most used - such as $numpy$ and $requests$ - but overall it is well balanced, meaning that it is able to withstand the removal of one of its most used packages. The data structure also provides satisfactory results, having a 89.6% accuracy when compared to the Python resolver. The findings of this study can be used to improve existing dependency networks.

1 Introduction

Each programming language has libraries - content created by other users and publicly shared for the common good. Most libraries are built on top of other libraries. There is no need for a developer to reinvent the wheel each time they want to build something. This means that most libraries depend on a subset of other libraries. Due to this relationship, the entire concept can be imagined as a graph, where the nodes are libraries, and the (directed) edges represent the concept of dependency.

Libraries are hosted on code repositories like PyPI [1] or npm [2]. From these repositories, using package managers like pip[1] and npm, developers can quickly install any needed dependency by using a simple yet powerful Command Line Interface (CLI). These tools recursively resolve the library requirements specified in each of the dependencies and install them on the host system.

Reliance on various dependencies (directly or transitively), while reducing development time and increasing efficiency [3], has the unwanted side effect of introducing vulnerabilities which can then be abused by malicious third parties to disrupt the normal workflow of various applications. One well known example of this happening is the Equifax incident, when 100.000 private credit card records were leaked due to an outdated dependency [4]. Another side effect is that the entire functionality of a system is entrusted to third parties that might turn malicious. An example of this happening is the faker.js corrupt update [5]. The maintainer of the library wanted to showcase the reliance of projects (especially that of closed source commercial projects) on their library, so they pushed an update that introduced an infinite loop, crashing thousands of projects worldwide.

[1]https://pip.pypa.io/en/stable/

As software engineering evolves, the depth of the transitive dependency chain also increases. Even now, it is challenging, if not impossible, to manually analyze such relations. As such, automated tools, such as Libraries.io [6], were developed. The purpose of such tools is to clearly inform the developer of all the dependencies their project requires and signal any possible vulnerability. But Libraries.io only shows the most recent compatible versions of dependencies.

What happens if historic data about vulnerabilities is needed? Researchers might be interested in analyzing how the vulnerability of an ecosystem manifests throughout time. But, currently, the only way to generate a dependency graph for a given time frame is to find data specifically for that period, then analyze it. This problem is even more apparent if vulnerabilities from different time frames need to be analyzed (for example for a comparative analysis of possible vulnerabilities throughout time). The need for a dependency graph data structure that takes as input a regular dataset (with timestamps) and then allows time filtering based on the release dates of each package becomes apparent.

This leads us into the the goals of this research: to construct a time-based dependency graph and to use this data structure to analyze the most used Python packages at a given time. This idea can be formalized into the main research question: **What are the most widely used Python packages at a given time?** To answer this, three sub-questions need to be answered first:

- **RQ1:** How would a data structure for package dependencies that contains a time component be designed?
- **RQ2:** Does the introduction of time make backwards resolution precise?
- **RQ3:** What are the most widely used Python packages?

Answering these sub questions will provide the foundation needed in order to answer the main research question.

Since this paper focuses on the dependency graph of Python packages, some context needs to be provided first. This language was chosen because it represents one of the most popular programming languages [2], being the go-to language for both data science and hobbyist programming. It has a central repository, PyPI, and provides a tool that can fetch dependencies - pip. To install dependencies, developers can either do it one by one or by providing a *requirements.txt* file.

The research paper will be presented in the following structure. Background about the researched field and information about existing studies can be found in Section 2. The methodology of the project is described in Section 3. Section 4 showcases the results of the research. The ethical aspects of this research and its reproducibility are discussed in Section 5. In Section 6, the results are investigated. Section 7 summarizes the most relevant aspects of the paper and a few ideas for future improvements are discussed.

2 Background

This section will focus on explaining some of the terminology that will be used throughout this paper. Then, some related

[2]https://web.archive.org/web/20220606182015/https://www.tiobe.com/tiobe-index/

work is analyzed.

2.1 Terminology

- **Transitive Dependency**. An indirect dependency resulting when direct dependencies also have their own dependencies.
- **Semantic Version**. A standard that specifies strict rules about the format of released versions of a piece of software. The standard format of semantic versioning is $MAJOR.MINOR.PATCH$.[3].
- **Dependency Graph**. A graph where each node represents a package or package version and the edges represent the dependency/dependent relationship.

2.2 Related Work

Time-Based Graphs

The idea of introducing time as a component of regular graphs has been discussed in past research. Various versions, ranging from simply adding the time component to the edge [7] to creating an entire framework based on the time component [8] were explored. The overarching idea is the same: time acts as a filter. The graph contains all possible edges and when a time constraint is introduced, only the connections that conform with it remain viable. This is a core idea of the graph that will be explained later in this paper.

Dependency Graphs

Analyzing the dependency graphs of programming languages is an emerging field. Kikas et al. [9] conducted a broad research, focusing on the structure and evolution of package dependency graphs for JavaScript, Ruby and Rust. Their work is based on gathering meta-data about packages from various sources (central repositories for each language, GitHub etc.), structure it in a dependency graph and then analyse it. The proposed graph structure heavily influenced the design of the graph that will be discussed in the next sections. Their study concludes that the number of packages are highly vulnerable to the removal of most used packages is increasing, while the removal of any other package has a decreasing vulnerability .

Wang. et al. [10] conducted research focused on the dependency graph of the Ubuntu operating system. While the research does not focus on the structure of the graph itself, it provides a few dependency graph analysis techniques that can be applied to the graph that will be constructed for this research.

Python Dependency Ecosystem

The python dependency ecosystem has been explored in the past. Ma et al. [11] analyzed the GitHub Python ecosystem in order to find its dominant packages. Their research focused on calculating centrality measures for all the Python packages that had more than 20 stars on GitHub. Their findings indicate that only a small number of projects have large values for their centrality measures. A thing to note is that this research only encompassed approximately 20,000. Bagmar et al. [12] also conducted research in this field. The focus of their research was to find vulnerabilities that could be exploited. A side effect of this process is that they also produced a ranking of the top 10 Python packages based on the amount transitive dependents each of them has. Both of these studies provide a way of ranking the most used packages in the Python ecosystem. The findings of this paper will be compared to their findings in Section 6.

Synthesis of related work

While there is previous research done on both time-based graphs/networks [7, 8] and regular dependency graphs [9, 10], the field of time-based dependency graphs remains mostly unexplored. This opens up the opportunity of combining previously found concepts into something new, a time-based dependency graph data structure. The research done on the Python dependency ecosystem [11, 12] will provide a way of validating the results obtained from this research.

3 Methodology

This section will explore the methodology used throughout the research project. As previously mentioned, the goals of this research are to construct a time-based dependency graph and to use this data structure to analyze the most used Python packages at a given time. To achieve this, a series of steps need to be completed first. The subsections that follow will each describe one step in this process.

3.1 Data acquisition

In order to generate a time-based dependency graph, metadata about packages and their dependencies had to be collected. This process took place during April and May 2022. To this end, a few different data sources were explored:

- **PyPI API** [13] . The first attempt was to gather data directly from PyPI using the APIs they provide. After some research into this method, it was discovered that their *robots.txt* file (a file that is used to let users know on which endpoints scraping bots are allowed) disallows access to the endpoints that would be needed for data acquisition.
- **Libraries.io** [6]. This website provides metadata about libraries and their dependencies (but only for the latest releases). As such, it could be used as a source of partial data that would allow further progression. Unfortunately, calls to the public API are heavily rate limited. This would result in a large number of hours being wasted waiting around for an incomplete data set. The last two data sources from this subsection are both faster and include more data.
- **Zenodo** [14]. A previous project provides a dump of PyPI metadata. This data dump is quite old and was itself gathered from Libraries.io so it would not provide any advantage compared to the other sources.
- **Project Metadata Table - Google BigQuery** [15]. This dataset represents the official data dump as provided by the team behind PyPI. It contains a large amount of data but some entries are duplicates caused by authors updating the metadata of their libraries. These updates create duplicates because the data set disallows changes to its

[3] https://semver.org/

entries, instead only allowing appends to it. This is the best overall data set, but processing it requires a substantial amount of time due to its size. One of the greatest drawbacks of this source is that it can be access only with a Google account making its automated retrieval quite hard.

- **PyPI Package Metadata Cache** [16]. This data source was created by hooking into a deprecated PyPI API. This data set gets updates as author update their libraries but because of the usage of a deprecated feature it can't be used reliably. Because the data does not contain as many duplicates as the previously mentioned source and is efficiently compressed, its size is more manageable. This source was used for the beginning phases of the project as an easier to use alternative to the Metadata Table data.

Each of these data sources were explored and in the end the decision was made to further process the Project Metadata Table as it provided the most accurate and complete version of the data due to it coming directly from the PyPI team.

3.2 Data processing

The data set from Project Metadata Table contained a large amount of information. Fortunately, not all of it was necessary for the purpose of this research, so only the **name**, **versions**, **upload time** and **required dependencies** of each package had to be extracted using the Big Query web application.

A common JSON structure was decided upon by the team as an expected input data format for creating the graph - Figure 1. A common format had to be chosen because the goal of the software created as part of this research is to have a way to construct time-based dependency graphs for multiple package repositories. It also makes it easier for future research to have a format that can be used as a base. The shown format was chosen because of its simplicity. Because it only uses dictionaries/maps, its parsing and conversion into a different data structure would prove trivial later down the line.

At this point, the data still contained duplicates and needed to be normalized in the previously mentioned format. To this end, the repository https://github.com/AndreiPurcaru/PyPIAnalyser was created. It contains a series of Jupyter Notebooks and Python scripts that allow the data to be cleaned and processed. This can be replaced by any other software that is able to convert the raw data into the previously mentioned JSON format.

3.3 Graph Design (RQ1)

The team spent the first couple of weeks researching the field of dependency graphs. During this time, a few designs were created and their usefulness was debated:

- **Node**: *name*, *version*. **Edge**: *timestamp-start*, *timestamp-end*. This design closely resembles the work done in [7]. It was decided against this design because in order to calculate the start and end of a timestamp, multiple comparisons would need to be done. These comparison would need to happen between each consecutive version of a package. They could provide insights into the lifetime of a version, but that's not the goal of this paper.


```
{
  "name": "pandas",
  "versions": {
    "1.4.2": {
      "timestamp": "2022-04-02T10:32:27",
      "dependencies": {
        "python-dateutil": ">=2.8.1",
        "pytz": ">=2020.1",
        "numpy": ">=1.21.0",
        "hypothesis": ">=5.5.3",
        "pytest": ">=6.0",
        "pytest-xdist": ">=1.31"
      }
    },
    "1.4.1": {...}
  }
}
```


Figure 1: JSON structure that was used as a middle step between raw data and the graph data structure

- **Node**: *name*, *version*. **Edge**: *timestamp*. This design was not used because we decided that instead of having to also store information on the edges, it would be cleaner to store all of the needed information in the nodes.
- **Node**: *name*, *version*, *timestamp*. **Edge**: *no information stored*. The nodes contain all the information about each specific version of a package. The directed edges represent the dependency/dependent relation, with the direction starting from the dependent and going towards the dependency. This is the final design. An example (using one of the dependencies of pandas) of this design can be seen in Figure 2. As it can be seen, the structure of graph is similar to that proposed by Kikas et al. [9]. The only difference is that instead of storing just the name and the version of each package, information about its release timestamp is also kept. This will allow time-based querying of the graph.

As a result of this design, a formal definition of the state of a time-based dependency graph in a given time frame can be constructed:

$$G_{[t_1,t_2]} = (V, E) \tag{1}$$

Where:

- t_1, t_2 are the timestamps that define the time frame of the state
- V is the set of all package versions, where $v \in V$ if $v_{timestamp} \in [t_1, t_2]$
- E is the set of edges, where an edge $e = (a, b)$ exists only if the package version a depends on package version b
- G is a directed graph

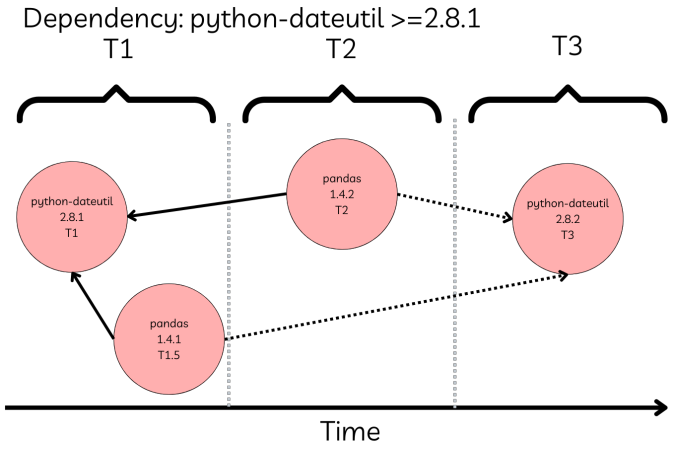

Figure 2: Example for the Final Design of a Time-Based Dependency Graph
(using Pandas and one of its dependencies)
The node from which the arrows start represent the dependent while the node at the other end represents the dependency. Dashed arrows represent a possible relation but at a later timestamp while full arrows represent the relation at the release of the dependent. Note: time is abstracted for brevity

3.4 Graph Creation

The time-based data structure represents the core element of this research. It also represents the part of this project that was conducted as a team. As such it was designed, implemented and tested in collaboration. The logic behind creating the graph can be seen in Algorithm 1. This can be implemented in any programming language, but a low-level language is recommended since the abstractions provided by higher level languages might make the implementation use more memory. The final implementation we constructed is available here: [17]

Algorithm 1 Creation of time-based dependency graph

Require: A JSON List of Packages in the normalized JSON format (Figure 1)
Ensure: A graph that contains each version of each package as a node, following the format mentioned in 3.3

```
listOfPackages ← parseJSON()
graph ← createEmptyGraph()
for package ← listOfPackages do   ▷ Create the nodes
    newNode ← createNode(package)
    graph.nodes.append(newNode)
end for
for package ← listOfPackages do   ▷ Create the edges
    for dependency ← package.dependencies do
        newEdge ← createEdge(package, dependency)
        graph.edges.append(newEdge)
    end for
end for
```

While the process was not complicated, a few caveats that had to be taken into account are:

- **Different package managers use different semantic versioning notations**. While PyPI uses the notation specified in PEP 440 [18], npm and maven (and probably others) use their own particular notation. To solve this, care should be taken in the selection (or creation) of a semantic versioning library that can handle multiple notations.
- **Different data sources use different timestamp format**. This is something to look for when implementing, especially if the goal is to accept normalized data from different package managers. This can also be fixed by introducing a timestamp specification to the normalized JSON format.
- **Some packages do not specify their dependencies**. Even though PyPI requires projects to include all of their dependencies in a specific file, some authors do not abide by this requirement. This problem is not really solvable, and it mostly shows one of the weaknesses of using dependencies.
- **Some packages do not specify a semantic version constraint**. On the same note as the previous caveat, authors sometimes do not specify a strict enough semantic versioning. While using the notation "$numpy$" to specify that the project is using $numpy$ works, specifying the version that was actually used for the development of the package is safer "$numpy >= 1.21.0$". This has little impact on the research, mostly in the fact that libraries that specify a generic dependency will have connections to all versions of that package (as they are all possible dependencies).

4 Results

This section will showcase the results obtained during this research project.

4.1 Graph Structure

When it comes to the generation of the graphs, 3 variations were created in order to balance data-size and creation speed with accuracy. The statistics of the generated graphs can be seen in Table 1. The first two graphs were generated using data from the Metadata Table provided by PyPI. The difference between these two entries is that for the second one, all of the dependencies that were tagged "extra" were removed. These dependencies are usually used for specifying extra requirements for specific processor platforms or for letting developers know what requirements are needed for contributing to said dependency. As it can be seen, this removed around 93 million edges, from 454M to 361M. The number of packages and package versions (nodes) are the same for both graphs, 420,000 and 3,6M respectively. Due to them having the same amount of nodes, the average number of versions per package is also the same: 8,76. The last source that was used is the Metadata Cache from Repology. This dataset is comparatively smaller, containing 169,000 packages and nodes and 350,000 edges. Since it contained exactly one version for each of the packages stored, the average number of versions per package is 1. All of these graphs use the design that was described in Section 3.

Table 1: Summary of Datasets

Source	Metadata Table	Metadata Table	Metadata Cache
Applied Processing	Keep All	Remove Extra	Keep All
Packages	420K	420K	169K
Nodes/Versions	3,6M	3,6M	169K
Edges	454M	361M	350K
Average Versions per Package	8,76	8,76	1,00

Table showcasing a summary of the used datasets. "Keep All" means that all of the dependencies of each package were taken into account. "Remove Extra" means that extra requirements (for example testing requirements) were removed

Due to its reduced size, the Metadata Cache graph is the only one that can be visualized. Graphia [19] was used to generate the visualization that can be seen in Figure 3. The red circles represent clusters of packages. For example, the cluster in the top-right represents code used by the the Alibaba Group[4]. When talking about connected components, the central component represents the biggest (weakly) connected component in the graph, while the components on the margins represent small projects that only rely on their own dependencies or on no dependencies at all. Due to the way the graph is drawn, it can also be seen that the more central a node is, the more dependents it has.

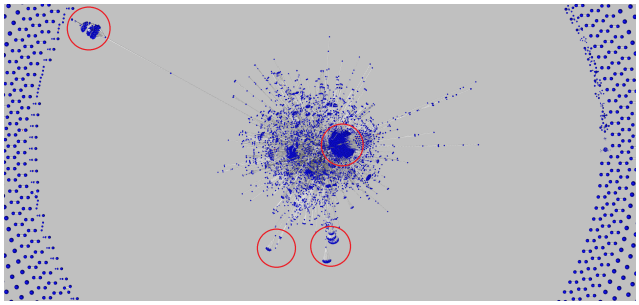

Figure 3: Visualization of Dependency Graph for 169K distinct package versions (zoomed in and cropped for showcase purposes). Circles represent groups of projects that stem from a common library.

4.2 Accuracy of Resolution (RQ2)

To test if the introduction of time keeps the same level of precision as the ground truth (in this case pip), a basic test was conducted. 10 package versions were arbitrarily chosen. Pip was used to install each of them and resolve their dependencies in a fresh environment. Then, pipdeptree [20], dependency tree generation tool, was used to generate all of the dependencies (both direct and transitive) of the package version. The graph was then generated (using the Metadata Table without extra requirements) and queried about the same package version. In order to calculate the accuracy of the resolution, the following formula was used:

Let:

- A be the set of transitive dependencies resolved by pip
- 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:

$$Acc = \begin{cases} 1 - \frac{|A| - |(B \cap A)| + 0.5 * E}{|A|}, & \text{if } |A| \neq 0 \\ 1, & \text{otherwise} \end{cases} \quad (2)$$

This formula doesn't take into account if the implemented resolution algorithm contains more dependencies than the ground truth. This is due to the fact that the implemented algorithm is also able to select platform specific dependencies, while pip only installs the dependencies required for the system it is being ran on. Because of this, we decided to not account for this case. Another caveat is that sometimes the algorithm selects the correct dependency with an incorrect version. This is most likely caused by the fact that the collected data is at this point one month old while pip was ran in the present. To make the accuracy calculation more balanced, the calculation gives 0.5 per each dependency in this case.

The calculated accuracy per package version can be seen in Figure 4. Overall, the accuracy of the algorithm is 0.896. This is a good result, considering the fact that the implementation is naive and that there is a discrepancy between the time when the data was collected and when pip was used.

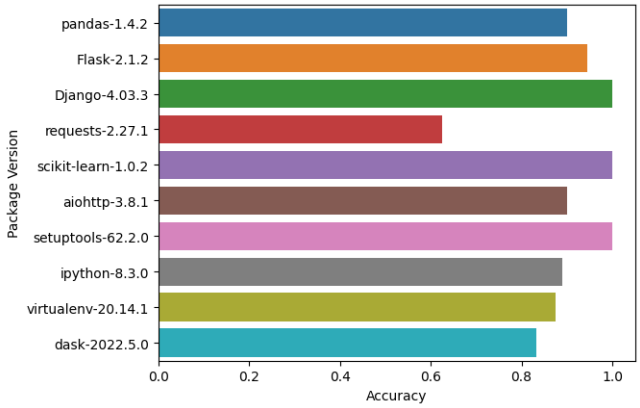

Figure 4: The accuracy per package version as calculated using Equation (2). The averaged accuracy is 0.896.

4.3 Finding the most used package versions (RQ3)

In order to answer the third proposed research question, PageRank[5] was employed. The choice of using it was threefold. Firstly, Gleich [21] has indicated that the algorithm can be successfully applied to graphs. The two other reasons were the algorithm's speed and popularity. Its popularity made it so we were able to have an already built implementation of it in

[4]https://en.wikipedia.org/wiki/Alibaba_Group

[5]https://en.wikipedia.org/wiki/PageRank

the graph library that was used for this project, while its speed meant that it could be applied to the largest created graph with relative ease and without incurring large wait times.

The Metadata Table without the extra dependencies was used for this process. This was done in order to capture the scores of packages and their versions across a large spectrum, both in terms of time and versions per package. Figures 5 and 6 represent the obtained results. A good thing to note is that the scores resulting from applying PageRank on the graph always add up to 1. This means that the ranking also provides insights into the distribution of the data.

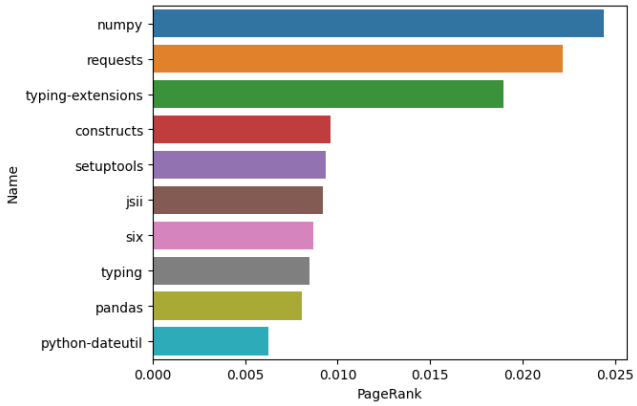

Figure 5: Top 10 packages of all time calculated using PageRank. The scores were calculated per individual package version, then aggregated into one score per package

Figure 5 contains a ranking of the top 10 most used packages from the whole dataset. This data was generated by running PageRank on the entire graph (without filtering for a specific time frame), then aggregating all the results by the package name. Packages numpy and requests come first with a score of 0.024410 and 0.022157 respectively. This makes them the most used libraries in the Python ecosystem.

With this data, a discussion can also be held about the vulnerabilities of the Python ecosystem. In this context, we define vulnerability in the same manner Kikas et al. [9] do: "fraction of the network nodes that is impacted by a removal of a single package or a single package version". As it can be seen, even the scores of the most used packages are quite low (< 0.025), meaning that even though some libraries can be marked as "most used", the ecosystem is balanced. Due to this, the overall vulnerability of the ecosystem is quite low. While removing a version or even the entirety of numpy would impact a large number of packages, it wouldn't bring the whole ecosystem down. Nevertheless, it is good to understand the importance of the most used packages in order to make the community aware that some packages are objectively more critical than others.

Figure 6 contains a ranking of the top 10 distinct most used package versions of all time. The PageRank scale is 10 times smaller than that of Figure 5. This is because this data was not aggregated, so each version represents a portion of the total PageRank of the entire package. This 10 times factor is also reinforced by the Average Versions per Package statistic

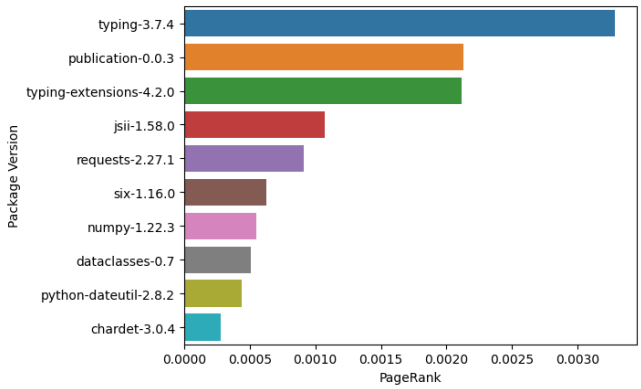

Figure 6: Top 10 package versions (for distinct packages) of all time calculated using PageRank. Note: the PageRank scale is different than that of Figure 5

that was presented in Table 1.

This data can provide insights into how different packages handle versioning and releases. Let's consider the typing package and numpy. When comparing packages, numpy is clearly ahead of typing. This changes drastically when comparing package versions, with typing's 3.7.4 release being clearly in the lead, and numpy's version 1.22.3 coming in the 7^{th} place. After checking each of their PyPI [1] pages, the cause of this discrepancy became clear. The typing package has rare releases, meanwhile numpy regularly has monthly releases. Another reason for this difference might be that typing does not use the standard semantic versioning which might create problems in the implementation. When comparing requests with numpy, the same trend emerges. The requests package releases less frequent than numpy.

Another aspect than can be seen in this data is that the top package versions are among the latest ones available in the processed dataset. This means that the Python community prefers to use the latest available release of the dependency they need. This in turn decreases the risk of vulnerabilities in the entire ecosystem.

4.4 Finding the most used package at a given time (RQ)

In order to answer the overarching research question, a process similar to the one in the previous subsection was conducted. For each year between 2015 and 2022, the graph was filtered to only contain package versions released that year (between 1^{st} of January and 31^{st} of December). Then, PageRank was applied to each of the filtered graphs. 7 packages that presented interesting trends throughout the years were selected and their PageRank score was mapped into a plot that can be seen in Figure 7. Note that the aggregated PageRank scores were used, not the per version ones. Quite a few interesting trends emerge from this data.

When looking at the PageRank scores throughout the years, the trend of relying more and more on a set of "core" packages emerges. While the scores of the top rated packages in 2015 was just above the 0.005 threshold, the scores of the top rated packages in 2022 are well into the 0.030 to 0.040

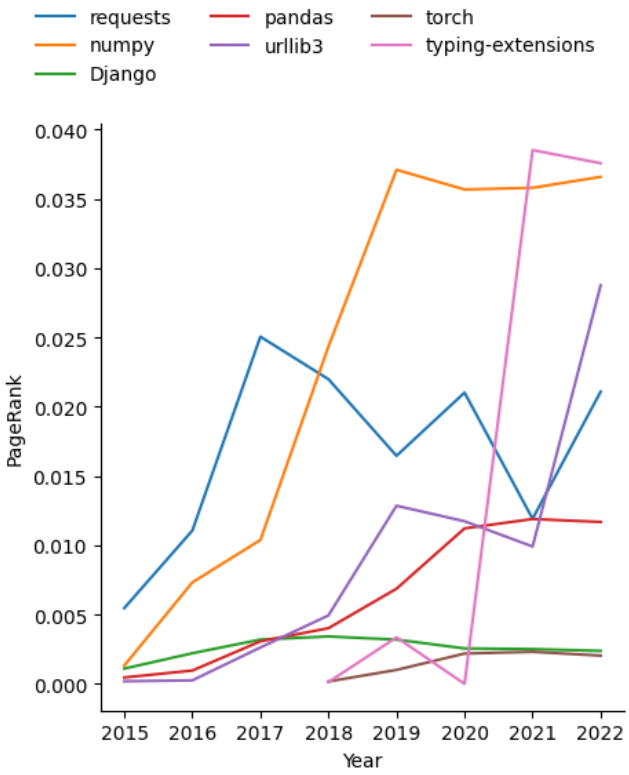

Figure 7: Popularity in terms of usage (calculated as PageRank scores) of 7 selected packages between 2015 and 2022

range. This means that in 2015 the importance of packages was more evenly distributed than in 2022. Even thought this trend might sound alarming, when looking at the whole picture, the highest PageRank score is still low enough to not indicate that the ecosystem is vulnerable to the removal of a singular package or package version. This trend can be seen as the community realizing the importance of not reinventing the wheel each time they create a package.

Some trends can be materialized when taking the packages one by one in isolation:

- torch (PyTorch) appeared in the dataset in 2018. Its score almost doubled from 2018 to 2020. Even though the usage of Machine Learning heavily increased between 2015 and 2022, the usage of torch didn't increase enough to make it stand out. This is most likely due to the fact that this analysis focuses on what packages are being used by other packages in the graph, while torch is a package that is used mostly in applications rather than packages/libraries.
- numpy was always one of the most used packages, but its usage climbed a lot between 2018 and 2019. Even now, in 2022, its usage is one of the highest.
- Django peaked in 2017. The decrease in popularity might be attributed to FastAPI entering the web framework scene in 2018.
- typing-extensions had a boom in popularity from 2020 to 2021. This strange trend was also confirmed by it still being one of the most used packages in 2022. This might indicate that the Python community is becoming more aware of the importance of having types in their code.

There are also some interesting interactions that can be visualized in this data. Requests has urllib3 as a dependency. This is why their scores are weakly correlated. This correlation is more visible in recent years. From 2020 to 2021 when urllib3's score decreased, so did requests' score. In the 2021-2022 period, the exact effect occurs, only this time both increased in popularity. Pandas depends upon numpy. As such, the same trend that is seen in the urllib3-requests relationship can be seen here. The difference is that urllib3 and requests compete in the same space (providing http operations) while numpy and pandas don't. This can be seen when looking at the two pairs in parallel. When requests' score dropped in 2019, urllib3's increased by a similar amount. Meanwhile, the scores of numpy and pandas fluctuate together.

5 Responsible Research and Reproducibility

Responsible Research

Due to the nature of this research, the ethical risk that is generated is minimal. Still, there are a few points that need to be addressed.

All of the data that was used for this project has been gathered from reputable sources and in accordance with all of the mentioned licensing agreements. The only personal information that can be gathered from the data sources that were mentioned in this report is the name of the author (only if using the PyPI Metadata Cache source - see 3.1), but authors are informed about this possibility when they agree to upload their project on PyPI. If any operation was disallowed, that approach would be stopped. A good example of this happening is the discovery that PyPI disallows scripts from accessing some of its API's (see 3.1).

When it comes to the results that were created during this project, they and the methods used to generate them are fully explained. Also, the results come from objective calculations, as to avoid any bias from corrupting them.

Reproducibility

When it comes to reproducing this research, utmost attention was paid to writing Section 3 as clearly as possible. This in turn should allow other researchers to follow the exact steps that were taken to reach the conclusion. The final data that was used for the generation of the results (both in raw and processed form), as well as the results generated during this project are publicly available [22]. All of the sources that were explored are also clearly mentioned in Section 3. The code repositories that were used for this research project are (and will remain) Open Source [17]. This was done to maintain transparency and to allow future researchers to start from where we left off instead of having to redo all of this work.

6 Discussion

This section will focus on comparing the results of this research with other studies. It will also include a list of limitations that impacted the creation of this work.

6.1 Results

Graph Structure

Since the graph structure is heavily inspired by the work of Kikas et. al [9], a similar discussion about it can be held. Because the research community has not settled on a singular method for generating dependency graphs, it is hard to conclude if the chosen representation is the best way of structuring said data structure.

While the work of Kikas et. al [9] predicted that only using package metadata (compared to using package version metadata) will not be sufficient in the future, we still haven't reached that point. For example, the work of Bagmar et. al [12] occurred in 2021 (data collected in 2019), 4 years after the publication of the study conducted by Kikas et al. The work of Bagmar et al. is mentioned because it is one of the more recent analysis of the Python ecosystem.

When discussing the structure from a time-based graph standpoint, no research that was encountered during the writing of this paper mentioned storing the timestamp inside of the nodes. The work of Wang et al. [7] focuses on having time information in the edges, while the work of Cattuto et al. [8] focuses on creating an entire framework built around the idea of time. Fortunately, the core ideas of the time-based dependency graph are abstract enough that they can be applied to a different graph representation.

Package Popularity

While Ma et al. [11] and Bagmar et. al [12] used different metrics than this research to evaluate the importance of a package to the Python ecosystem, some similarities did occur. In both studies, just like in this one (Figure 5), numpy, requests and six were in the top 10. This gives a degree of validity to the findings produced by this paper. An interesting point is that in the results of Ma et al. [11], pandas doesn't get a place in the top 20. This is due to the fact that at the time when this research was conducted (2016) Pandas had just been released. This also confirms the findings of this paper (Figure 7). Meanwhile, typing-extensions is not listed in any of the studies. This also support the results of this paper, as it can be clearly seen (Figure 7) that it rose in popularity only recently (the data for the studies was collected in 2016 and 2019 respectively).

6.2 Limitations

This section will focused on the limitations of this research paper. It will mention any ideas for circumventing these limitations.

Memory Usage

The implementation created for this paper [17] uses an extreme amount of RAM. For loading the Project Metadata Table [15], without any extra requirements 56 GB of memory are required (69 GB for also loading the extra requirements). This amount was recorded after an attempted optimization of the graph library that was used - removing one of the maps that was used for storing the edges. This optimization theoretically halved the memory usage. Overall, the amount of memory used by the code is still abnormally high. A suggestion could be switching to a different programming language for the implementation. In the provided repository, the basic data structure is also written in Rust. The memory usage was way lower than when using the Go implementation. This might prove to be a better approach than the current one.

Dependencies with Special Requirements

Some packages have special dependency requirements depending on the platform they are being installed on. These dependencies are tagged similarly to how extra requirements are tagged. For this paper, all platform specific requirements were treated as regular requirements. But, future research might want to focus on platform specific requirements. This would require minimal change in the repository, but the provided dataset would then become invalid.

Data not Clean Enough

Even though processing was applied to the data, it is still not in its cleanest form. As it can be seen in Figure 3, there are a lot of free floating nodes (not connected to any other node). This is due to the fact that packages with no dependencies and no dependents were not remove from the dataset. While it doesn't impact the results drastically, it might save quite a bit of processing time and memory to not even consider these packages.

Project Metadata Table Requires a Google Account

This is not the biggest limitation of this research but it has to be mentioned. Due to the fact that the main dataset used for this paper comes from Google BigQuery [15], a Google account is required to access it. Unfortunately, to be able to download it, a Google Cloud Project needs to be created and a lot of steps need to be taken. To alleviate some of this nuisance, the dataset used for this project was included in the Zenodo upload [22]. But if new data needs to be acquired, the process will take a bit of setup.

7 Conclusions and Future Work

The main contributions of this paper are: (i) a data structure that introduces the time component into dependency graphs; (ii) an analysis of the time-based dependency graph constructed for Python packages. The data structure represents a step into the domain of time-based dependency graphs. Its formalization enables it to be implemented in any programming language. It also allows properties to be derived from it. When it comes to the analysis part, multiple trends were discovered. It also became apparent that the Python ecosystem has evolved over the years, with developers using top packages now more than in 2015. The analysis also showed that from a popularity of usage standpoint there aren't any packages that, once removed, would collapse the entire ecosystem.

This research opens up the way to improve multiple other venues. One of the first recommendations would be the reimplementation of the existing code base in a different, more memory efficient graph structure or even a different programming language. The general goal of future research should

be to optimize the current prototype to make it viable for real world applications. More in depth research could also be conducted on the specifics of particular ecosystems at specific times.

References

[1] "Pypi," [Online]. Available: https://pypi.org/.

[2] "Npm," [Online]. Available: https://www.npmjs.com.

[3] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," *Empirical Software Engineering*, vol. 12, no. 5, pp. 471–516, 2007. DOI: 10.1007/s10664-007-9040-x.

[4] J. Hejderup, A. van Deursen, and G. Gousios, "Software ecosystem call graph for dependency management," English, in *ICSE-NIER'18 Proceedings of 40th International Conference on Software Engineering*, R2 - Software Engineering in Other Domains Accepted author manuscript; ICSE 2018 : 40th International Conference on Software Engineering ; Conference date: 27-05-2018 Through 03-06-2018, United States: Association for Computing Machinery (ACM), 2018, pp. 101–104, ISBN: 978-1-4503-5662-6. DOI: 10.1145/3183399.3183417. [Online]. Available: https://www.icse2018.org/.

[5] A. Sharma. "Dev corrupts npm libs 'colors' and 'faker' breaking thousands of apps." (2022), [Online]. Available: https://www.bleepingcomputer.com/news/security/dev-corrupts-npm-libs-colors-and-faker-breaking-thousands-of-apps/ (visited on 05/19/2022).

[6] "Libraries.io," [Online]. Available: https://libraries.io/.

[7] Y. Wang, Y. Yuan, Y. Ma, and G. Wang, "Time-Dependent Graphs: Definitions, Applications, and Algorithms," *Data Science and Engineering*, vol. 4, no. 4, pp. 352–366, 2019. DOI: 10.1007/s41019-019-00105-0.

[8] C. Cattuto, M. Quaggiotto, A. Panisson, and A. Averbuch, "Time-varying social networks in a graph database," *First International Workshop on Graph Data Management Experiences and Systems*, 2013. DOI: 10.1145/2484425.2484442.

[9] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and Evolution of Package Dependency Networks," *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017. DOI: 10.1109/msr.2017.55.

[10] J. Wang, Q. Wu, Y. Tan, J. Xu, and X. Sun, "A graph method of package dependency analysis on Linux Operating system," *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, 2015. DOI: 10.1109/iccsnt.2015.7490780.

[11] W. Ma, L. Chen, Y. Zhou, and B. Xu, "What Are the Dominant Projects in the GitHub Python Ecosystem?" *2016 Third International Conference on Trustworthy Systems and their Applications (TSA)*, 2016. DOI: 10.1109/tsa.2016.23.

[12] A. Bagmar, J. C. Wedgwood, D. Levin, and J. Purtilo, "I know what you imported last summer: A study of security threats in the python ecosystem," *ArXiv*, vol. abs/2102.06301, 2021.

[13] "Pypi api," [Online]. Available: https://warehouse.pypa.io/api-reference/json.html.

[14] M. M. Hossain, N. Mahmoudi, and C. Lin, *An Analysis of Dependency Network Evolution in PyPI*, version 0.1, Zenodo, Apr. 2019. DOI: 10.5281/zenodo.2620607. [Online]. Available: https://doi.org/10.5281/zenodo.2620607.

[15] Python Software Foundation, *Python Package Index (PyPI)*, Google Cloud Platform. [Online]. Available: https://console.cloud.google.com/marketplace/product/gcp-public-data-pypi/pypi.

[16] "Repology cache," [Online]. Available: https://pypicache.repology.org/.

[17] A. Purcaru, A. Dumitru, A. Brands, D. Corlade, T. Dobrev. "SoftwareThatMatters Code Repository." (2022), [Online]. Available: https://github.com/AndreiPurcaru/SoftwareThatMatters.

[18] D. S. Nick Coghlan, "Version identification and dependency," PEP 440, 2013. [Online]. Available: https://peps.python.org/pep-0440/.

[19] "Graphia," [Online]. Available: https://graphia.app/ (visited on 05/17/2022).

[20] "Pipdeptree," [Online]. Available: https://pypi.org/project/pipdeptree/ (visited on 06/16/2022).

[21] D. F. Gleich, "PageRank Beyond the Web," *SIAM Review*, vol. 57, no. 3, pp. 321–363, 2015. DOI: 10.1137/140976649.

[22] A. Purcaru, *SoftwareThatMatters - Analyzing the effect of introducing time as a component in Python dependency graphs*, version 1.0.1, Zenodo, Jun. 2022. DOI: 10.5281/zenodo.6659483. [Online]. Available: https://doi.org/10.5281/zenodo.6659483.