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

TU Delft

Authors

Alexandru Dumitru

Supervisors

Georgios Gousios, Diomidis Spinellis

1 Background

- Importing packages that depend on other software leads to the appearance of **transitive dependencies.**
- "Fragile" packages may appear due to transitive dependencies, which can lead to high vulnerability issues [1]. Some famous examples: Log4J, Equifax
- Dependency network is hard to track when taking time into account.

2 Research Question

- **What would a graph data structure for package dependency look like?**
- **Does the introduction of time increase the precision of the data structure?**
- **What are the most critical software applications?**

3 Methodology

Main idea: Implement a time-dependenat graph to showcase dependencies between packages.

- Look for applications in the Debian package ecosystem and collect meta-data regarding them.
- For each package version, a node is created. Edges are created based on dependency requirements (Fig 1)
- Queries based on time intervals can be performed. The structure will select only the latest version of a package, depending on the time interval.

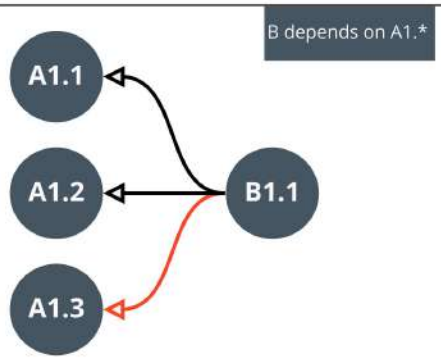

Fig 1 : At query time, only the latest version of A1.x will be chosen, based on the time interval given as input.

4 Results & Analysis

The graph is able to correctly resolve package versions.

Multiple packages were checked to see if the graph actually selects the latest version based on the timestamp, and the package requirement. (See Fig 2)

Dependency Requirement	*Resolved Versions*
`libc6 (>= 2.17)`	2.19-18+deb8u6 **2.31-13+deb11u3** 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)`	**7.74.0-1.3+deb11u1**
`zlib1g (>= 1:1.1.4)`	1:1.2.8.dfsg-5 1:1.2.11.dfsg-1 **1:1.2.11.dfsg-2** 1:1.2.7.dfsg-13 1:1.2.8.dfsg-2+b1

Fig 2 : Shows the dependency requirements of the package curl returned by Debian, and what are the resolved versions returned by the graph. Only the bold vaues are returned when asking for latest versions

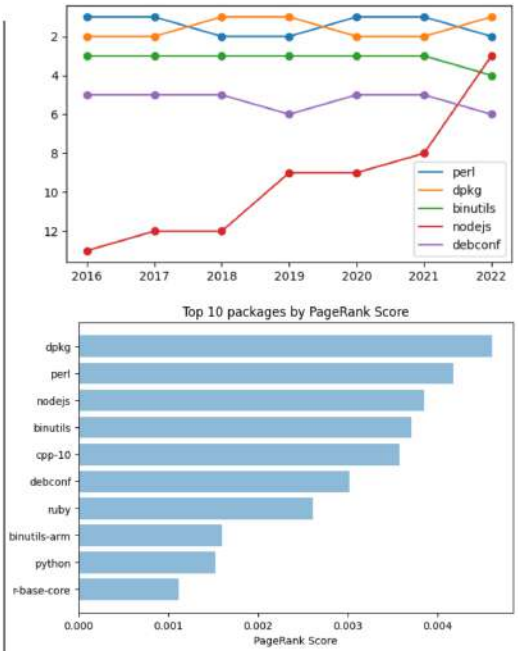

Fig 3&4: Top application ranked by PageRank score and how the ranked changed through time.

A custom criticality measurement was proposed to rank the most critical packages.

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:

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

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

Fig 5: Top application ranked by criticality score.

5 Conclusion

- A Graph structure is suitable to depict the dependencies between packages, as the mapping of a dependency can be depicted by a directed edge.
- The application dpkg ranked first both in PageRank and in the criticality score tables.

6 Future improvements

- Improve memory performance, the main bottleneck of the structure is that it requires a high amount of memory to store the nodes and the edges.
- Consider adding more Debian-specific relations, such as 'breaks', in order to ensure better results.

Related literature

[1] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. EmpiricalSoftware Engineering, 24(1):381–416, February 2018.