The Journal of Systems & Software 198 (2023) 111610

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

On the dependency heaviness of CRAN/Bioconductor ecosystem ☆

Zuguang Gu

National Center for Tumor Diseases (NCT), Im Neuenheimer Feld 460, Heidelberg, 69120, Germany

ARTICLE INFO

Article history:
Received 2 September 2022
Received in revised form 16 December 2022
Accepted 3 January 2023
Available online 5 January 2023

Dataset link: https://CRAN.R-project.org/package=pkgndep

Keywords:
CRAN
Bioconductor
Dependency graph
Dependency heaviness
Software ecosystem
Software engineering

ABSTRACT

The R package ecosystem is expanding fast and dependencies among packages are becoming more complex in the ecosystem. In this study, we explored the package dependencies from a new aspect. We applied a new metric named "dependency heaviness" which measures the number of additional strong dependencies that a package uniquely contributes to its child or downstream packages. We systematically studied how the dependency heaviness spreads from parent to child packages, and how it further spreads to remote downstream packages in the CRAN/Bioconductor ecosystem. We extracted top packages and key paths that majorly transmit heavy dependencies in the ecosystem. Additionally, the dependency heaviness analysis on the ecosystem has been implemented as a web-based database that provides comprehensive tools for querying dependencies of individual R packages.

1. Introduction

R has become a major programming language applied in many fields, including statistics, bioinformatics, geoinformatics, economics and general data science. It is widely used for a variety of tasks such as data processing (Grolemund and Wickham, 2016), visualization (Wilke, 2019), statistical modeling (Venables and Ripley, 2002), interactive web application development (Chang et al., 2022) and reproducible reporting (Xie, 2015). The reusable and extensible code implemented by developers are formatted as R packages and distributed on public repositories such as the Comprehensive R Archive Network (CRAN)[1] and Bioconductor[2] (Huber et al., 2015). CRAN is the major repository for R packages and Bioconductor specifically focuses on biology data analysis, especially on high-throughput omics data.[3] Both repositories perform manual validations on new packages and apply regular checks on packages to ensure their quality and usability. The number of packages in the two repositories increases almost exponentially by year (Hornik et al., 2021). Besides these two "standard repositories", there are also a huge number of R packages distributed on GitHub, which are self-maintained and mainly for experimental purposes.

Code reuse is an essential part of a programming language (Haefliger et al., 2008). An R package may inherit functionalities from other packages, which forms the dependency relations between packages. In other words, a package depends on the availability of other packages to work. R packages as well as their complex dependency relations construct the ecosystem. The continuously increasing number of R packages makes the dependency relations among packages even more complex (Mora-Cantallops et al., 2020a). By 2022-06-08, there are in total 22,076 CRAN and Bioconductor packages, including 194,351 direct dependency relations.[4] With such complex dependency relations, the ecosystem might be vulnerable to failures of only a small number of packages. For example, recently a compiling failure of the *isoband* package caused additional failures of 4,747 (almost 25%) of all packages on CRAN.[5] *isoband* is a dependency of the hub package *ggplot2* and it results in the failure being transmitted continuously to a huge number of downstream packages via *ggplot2*. Thus, it is important to explore the dependency structure in the R package ecosystem from the aspect of software engineering, which helps to understand the structure of the ecosystem,

☆ Editor: Dr. A. Bertolino.

E-mail address: z.gu@dkfz.de.

1 https://cran.r-project.org/.

2 http://bioconductor.org/.

3 Omics is a branch of biology science which studies biological systems from large-scale data. It includes a list of sub-branches, such as genomics which aims to reveal mutations and rearrangements on DNAs, and transcriptomics which quantifies mRNA levels of all genes in an organism.

4 The statistics were obtained with the R function `available.packages()`.

5 See a bug report on https://twitter.com/cjvanlissa/status/1577552826561171457. The failure was fixed very quickly within a week, https://github.com/wilkelab/isoband/issues/33.

https://doi.org/10.1016/j.jss.2023.111610
0164-1212/

to reveal top packages having major impacts and to study the vulnerability of the ecosystem.

The ecosystem contains complex dependency relations between packages, which can be naturally modeled as a graph (Fortuna et al., 2011). There are mainly two types of studies on the dependency graph. In the first category, researchers applied complex network analysis (CNA) approaches on the dependency graph, such as calculating various centrality metrics to explore the impacts of top packages, or applying network community methods to partition packages into densely connected subgraphs to further analyze their specific attributes (Mora-Cantallops et al., 2020b). In the second category, researchers developed tools for visualizing the complex dependency relations. These tools are mainly implemented as R packages, such as *deepdep* (Rafacz et al., 2021), *pkgnet* (Burns et al., 2021), *pkggraph* (Srikanth and Nikhil, 2018) and *miniCRAN* (de Vries, 2022). They give intuitive views of how dependencies are transmitted between packages. Nevertheless, they only work well on subgraphs showing local relations with small sizes, e.g., dependencies from a small number of R packages, while it becomes difficult to generate and to read when the size of the graph increases.

Network analysis on the dependency graph helps to understand the R ecosystem from the system's level. The degree centrality is a widely-studied metric which measures the number of dependency packages or the number of child packages that depend on a package (i.e., the dependents) (Korkmaz et al., 2019; Mora-Cantallops et al., 2020b). Indeed, degree is an important metric for revealing top packages that have significant impacts on the ecosystem. However, it is a local metric and we can still look at the system from new aspects. In practice, when a user installs a new package *P*, additional packages to be installed that he would notice are actually the total dependency packages upstream of *P*, while which packages are *P*'s direct dependencies are unobservable to the user. From a developer's perspective, the direct dependencies of his package provide no information of which parent brings more dependencies to it, while he needs to inspect the complete upstream of the dependency chain to find out parent packages contributing heavy dependencies. All these imply, the number of "total dependencies" instead of "direct dependencies" is a more practical metric. In fact, the number of total dependencies begins to be paid more attention in the R community. For example, Bioconductor and R universe[6] are listing the number of total dependencies as a basic metric for the packages hosted there.

When a package *P* has potentially more total dependencies, there are several consequences which affect the usability of *P*. We have listed the risks in our previous paper (Gu and Hübschmann, 2022): *"(i) Users have to install a lot of additional packages when installing P, which would bring the risk that installation failure of any upstream package stops the installation of P. (ii) The number of packages loaded into the R session after loading P will be huge, which increases the difficulty to reproduce a completely identical working environment on other computers. (iii) Dependencies of P will spread to all its child packages. (iv) On the platforms for continuous integration such as GitHub Action or Travis CI, automatic validation of P could easily fail due to the failures of its upstream packages."* Therefore, it is important to reveal packages contributing high total dependencies to better study their impacts on the vulnerability of the ecosystem.

Total dependency, or the transitive dependency, has already been investigated in several studies (Abate et al., 2009; Mora-Cantallops et al., 2020b), however, there still lacks a way to capture the unique transitive dependencies that a single package contributes. In our previous study (Gu and Hübschmann, 2022), we proposed a new metric named "*dependency heaviness*" that measures the number of dependencies that a parent uniquely brings to its child package and are not brought by any other parent. Simply speaking, this new metric helps to identify which parent package is heavier in the context of how it uniquely contributes the dependencies to its child package. Since now the dependency contribution of parent packages can be measured quantitatively, developers can easily identify heavy parents of their packages, then apply possible optimization to reduce the complexity of package dependencies and build more robust software. Of course, how to optimize the dependency depends on the specific uses of parent packages in the corresponding package. We recommended several solutions in Section 9.1.

Dependency heaviness can provide new insights for risk analysis on package ecosystems. Number of dependencies, especially transitive dependencies, is a commonly-used metric of how vulnerable a package is to code breaks in the ecosystem. Then, the dependency heaviness also measures accumulated risks from upstream to a package that are uniquely transmitted via a parent.

We have implemented the dependency heaviness metric in an R package *pkgndep*.[7] In our previous study (Gu and Hübschmann, 2022), we briefly described *pkgndep* as software with several use cases. In this study, we extended the definition of dependency heaviness to a broader range. Besides the dependency heaviness from a single parent to a single child package, we also studied how it is transmitted to remote downstream packages. We performed a systematic empirical study on the dependency transmission in the R package ecosystem. The contributions of this study are briefly listed as follows:

1. We studied how dependency heaviness spreads locally from parent to child packages, and we studied how dependency heaviness is simultaneously contributed by two parent packages.
2. We studied how dependency heaviness is transmitted remotely from upstream to downstream packages.
3. We applied CNA approaches and we extracted top packages and key paths that majorly transmit heavy dependencies in the ecosystem.
4. The dependency heaviness analysis on the ecosystem has been implemented as a web-based database which provides comprehensive tools for analyzing dependencies of individual R packages.

The remainder of this paper is organized as follows. Section 2 briefly summarizes current studies on the ecosystem of R as well as other programming languages. Section 3 provides technical background for understanding dependency relations in R. Section 4 introduces definitions of various heaviness metrics. Section 5 describes the new functionalities of the *pkgndep* package for this study as well as data processing. Section 6 raises various research questions. Section 7 describes the results of the analysis and answers the research questions. Section 8 describes the web-based database for the dependency heaviness analysis. Section 9 discusses how dependency heaviness analysis benefits developers. Section 10 summarizes the study. Section 11 discusses limitations of the analysis and proposes future plans. Section 12 encloses the paper with conclusions.

2. Current studies

2.1. Current studies on the R package ecosystem

There are a few studies applied on the R ecosystem where researchers analyzed the dependency relations as well as from

[6] https://r-universe.dev/.

[7] https://CRAN.R-project.org/package=pkgndep.

other aspects. In this section, we briefly described their analyses and findings.

R package dependencies can be modeled as a network with a complex structure. Mora-Cantallops et al. (2020b) performed complex network analysis on the CRAN ecosystem. Similar to social networks, they found the dependency network of CRAN also has the scale-free property. In a scale-free network, node degree follows a power-law distribution and the proportion of degrees of hub nodes is stable as the network size increases (Wang and Chen, 2003). This implies, in the R ecosystem, there are only a small number of hub packages that contribute huge amounts of dependencies to other packages. They also partitioned the global dependency network into modules and they found CRAN is modular where each module mainly corresponds to a specific analysis task.

Bommarito and Bommarito (2021) studied how packages contribute dependencies to their dependents, but aggregated by developers. Very interestingly, many of the most depended-on packages are maintained by the same developers. They found the top 10 developers are responsible for more than 50% of all packages in the ecosystem, and the percent would become higher if transitive dependency relations are also considered. This suggests that the R ecosystem is more vulnerable on the developer's level.

German et al. (2013) separated R packages into base packages, recommended packages, popular packages and contributed packages where the order of the four groups reflects the priority to be core packages in the ecosystem. They found that the more core a package is, the more dependents it has. Additionally, they explored other aspects such as code size, documentation, and community interest. One interesting finding was code size for base packages keeps increasing over years while in other categories code size is almost stably unchanged.

As there are also a huge amount of R packages hosted on GitHub, Decan et al. (2016b) studied the inter-repository package dependencies between CRAN and GitHub. They performed survival analysis and found that packages on GitHub are easier to break due to updates of their dependencies from CRAN. This may imply R packages on GitHub are less maintained and tested.

Mora-Cantallops et al. (2020a) studied the evolution of the R ecosystem. They revealed that in general packages are stably updated over the years, but the number of packages as well as the complexity of dependency relations are increasing. The increasing complexity is expected as a result of the scale-free property of the dependency graph where hub packages will preferably be linked to more new packages when the ecosystem evolves.

There are also studies focusing on other aspects of the R ecosystem. Claes et al. (2014) analyzed package errors on CRAN. They found Solaris and MacOS have more errors mainly because they are less used in development and packages are less tested on the two platforms. They also found the majority of the errors on CRAN are from external factors thus irrelevant to developers.

Korkmaz et al. (2019) measured the impacts of packages by the numbers of downloads, and they studied how various factors can predict package impacts by utilizing a generalized linear regression model. They used three groups of predictor variables: 1. package features such as number of commits and GitHub stars, 2. dependency networks centrality such as degree and closeness, 3. metrics from the co-developer network.[8] They found the number of dependents is a major factor that well correlates to package impacts. Another interesting finding in this study was if an author contributes to more packages, i.e., if an author is more active, his packages tend to have higher impacts.

[8] In a co-developer network, developers are nodes and two developers are connected if they contribute to the same package.

In a very recent paper (Vidoni, 2021), the author called for more software engineering studies on R programming language. She proposed various futural directions for both software engineers and R developers for better understanding the R ecosystem from the system's level.

2.2. Current studies on other ecosystems

Given parent–child dependency relations of packages, the construction of dependency graphs is similar for other programming languages. Methods applied in one ecosystem can almost be seamlessly applied to other ecosystems. We briefly described some of the current analyses and findings as follows.

Decan et al. (2016a) compared topology of dependency graphs in CRAN, npm (package repository of JavaScript) and PyPI (package repository of Python) and they found various network metrics are statistically different between ecosystems. This may reflect specific patterns of how packages are implemented in different programming languages. In particular, they found Python packages are more isolated where a large number of them only depend on the standard core packages.

Abate et al. (2009) studied the transitive dependency in the ecosystem. They proposed a metric "sensitivity" which measures the total number of downstream packages. They applied it to the Debian ecosystem and they revealed packages with small amounts of direct dependents but affecting majority of other packages in the ecosystem in an indirect manner. They claimed sensitivity is a more meaningful metric for measuring risks in the ecosystem.

Evolution of a package involves bug fixing, adding new features and changing interfaces, thus it may cause code breaks to downstream packages. Bogart et al. (2016) studied different responses to break changes, i.e., package updates that are not compatible with its dependents packages, in the Eclipse, npm and CRAN ecosystems. They found that different responses reflect the difference on policies and values of the repositories, such as that Eclipse requires backward compatibility, npm allows break changes via parallel versioning of the same package, and CRAN requires compatibility always for the newest versions.

Jafari et al. (2021) discussed strategies applied in dependent packages for getting rid of break changes from upstream. By analyzing the JavaScript ecosystem, they discovered seven bad habits for handling dependencies, which they named as "dependency smells", for example, depending on a fixed version, or depending directly on a remote repository with a URL. They argued these smells will produce potential risks to the ecosystem in the future. As a note, the dependency smell they categorized can mostly be applied to other ecosystems.

For a package P, Abate et al. (2009) proposed a metric "strong dominance" which measures the proportion of P's dependencies that are from an upstream package Q. The metric helps to identify a package that dominantly contributes dependencies to its downstream package P. It is similar to the dependency heaviness metric we applied in this paper, but the dependency heaviness metric is more focused on the unique dependencies that Q contributes to P in the ecosystem.

3. Background concepts

Reusable code in R is formatted as packages. R provides a flexible way for handling dependencies. In this section, we first introduce the definitions of various dependency relations. Next we categorize packages according to their relations in the dependency graph, and specific heaviness metrics will be defined for them later in Section 4.1.

```
Type: Package
Package: lubridate
Title: Make Dealing with Dates a Little Easier
Version: 1.8.0.9000
Maintainer: Vitalie Spinu
License: GPL (>= 2)
URL: https://lubridate.tidyverse.org,
    https://github.com/tidyverse/lubridate
Depends: methods, R (>= 3.2)
Imports: generics
Suggests: covr, knitr, rmarkdown,
    testthat (>= 2.1.0), vctrs (>= 0.4.1)
Enhances: chron, timeDate, tis, zoo
LinkingTo: cpp11 (>= 0.2.7)
...
```

Fig. 1. The DESCRIPTION file of the package *lubridate*. Only a fragment of the file is demonstrated. This is an example where all five dependency fields are specified. For most of the packages in CRAN/Bioconductor, only a subset of them is specified. Version requirements can also be specified for dependency packages.

3.1. Dependency relations

Dependency relations are declared in the fields of "Depends", "Imports", "LinkingTo", "Suggests" and "Enhances" in an R package's DESCRIPTION file, which locates under the root directory of a package (Fig. 1). Denote a package as *P*, dependency packages in its "Depends" are expected to be directly used by users if *P* is to be used, and they provide base functionalities for *P*. For example, it might be a good idea to set the package *ggplot2* as a "Depends" package for its extension packages. Dependency packages in "Depends" are attached to the search path[9] in the R session ahead of *P* when executing the command `library(P)`. All their public functions are visible to users. Dependency packages listed in "Imports" are internally used by *P* where specific functions, methods or classes from there are imported into *P*'s namespace based on the rules defined in *P*'s NAMESPACE file. The "Imports" packages are also loaded in the R session, but they are not attached to the search path, thus not visible to users. Dependency packages listed in "LinkingTo" contain header files to compile *P*'s C/C++ code. Packages listed in "Depends", "Imports" and "LinkingTo" are necessary for using *P* and they must be installed before installing *P*.

Fields "Suggests" and "Enhances" are similar. They contain dependency packages that are not necessary for using *P*, e.g., only used in examples or vignettes, or in the code that provides optional functionalities of *P*. Thus, these two types of dependencies are not mandatory to be installed when installing *P*. However, dependencies in "Suggests" are by default required for a complete `R CMD check`.[10] "Enhances" field is more flexible and packages listed are never checked.

In the CRAN/Bioconductor ecosystem, the proportions of the five types of dependency relations are 8.8% for "Depends", 54.4% for "Imports", 2.8% for "LinkingTo", 33.7% for "Suggests", and 0.4% for "Enhances" from 194,351 relations in 22,076 packages.[11]

Readers please refer to the official R manual "Writing R Extensions" (R Core Team, 2022) for more details on the dependency relations.

3.2. Flexible control of dependencies

Being different from other programming languages, R provides a flexible control of dependencies. For example, in JavaScript, all dependencies should be available in advance of using a library, even if the functionality of a dependency is only rarely used by users. As a comparison, R allows specifying a set of core dependencies as well as a set of weak dependencies. Dependencies used in *P*'s source code can be explicitly specified in a form of `pkg::function(...)` where `pkg` is a parent of *P*. In this way, `function()` is not directly imported from `pkg`, thus availability of `pkg` is not mandatory for `R CMD check` while it is only checked when `pkg::function(...)` is executed in *P*. Although `pkg` contributes to *P*'s source code, it is used optionally. Thus it can be declared as a weak dependency and put in *P*'s "Suggests" field. This actually provides the possibility to optimize *P*'s dependencies. When a parent of *P* brings a large number of extra dependencies but it only provides limited functionalities which are rarely used by users, it is reasonable to specify it as a weak dependency in "Suggests".[12] Actually this is the motivation of us to develop the *pkgndep* package which helps developers easily identify parent packages that contribute heavy dependencies.

3.3. Dependency categories

Depending on different dependency relations, the following dependency categories for package *P* are defined as follows.

Strong parent packages: Dependency packages listed in the "Depends", "Imports", and "LinkingTo" fields of *P* (red box in Fig. 2). They are also called **strong direct dependency packages** of *P*. Strong parent packages are mandatory to be installed when installing *P*. To make it easy to discuss, we always referred to them as **parent packages** in the paper.

Weak parent packages: Dependency packages listed in the "Suggests" and "Enhances" fields of *P* (green box in Fig. 2). They are optionally required when installing *P*.

Strong dependency packages: Total dependency packages by recursively looking for parent packages (category A, B, as well as packages in red box in Fig. 2). They are also called **upstream packages**. Note strong dependency packages include parent packages. Strong dependency packages are mandatory to be installed when installing *P* and failure of any strong dependency package will prevent installation of *P*. In some other studies, they are also called **transitive dependency packages** (Decan et al., 2019; Mora-Cantallops et al., 2020b; Kikas et al., 2017).

All dependency packages: Total dependencies by recursively looking for parent packages, but on the level of *P*, its weak parents are also included (package category A, B, C and D, plus all packages listed in the red and green boxes in Fig. 2). It simulates when the full functionality of *P* is required, or when all weak parents become strong parents, the total number of strong dependency packages that *P* requires. In this paper, we did not discuss packages in this category, but for completeness, its definition is still given here.

Child packages: Packages whose parents include *P* (category E in Fig. 2). They are the packages on which *P* has a direct impact of dependencies. In some studies and package repositories, they are also called **direct dependents** (German et al., 2013).

[9] Search path is an ordered list of environments and package namespaces where a function is sequentially looked up when it is executed by a user. The R function `search()` returns the search path.

[10] `R CMD check` is a command that performs comprehensive checks on a package. It checks source code, documentations, examples, vignettes and it runs unit tests. A successful check is required for acceptance on CRAN/Bioconductor.

[11] The dependency relations of all CRAN/Bioconductor packages were obtained with the R function `available.packages()`. Data was collected on 2022-06-08.

[12] Developer can write helper code to check whether weak parent is already installed. If not, a friendly message can be printed to inform users to install it.

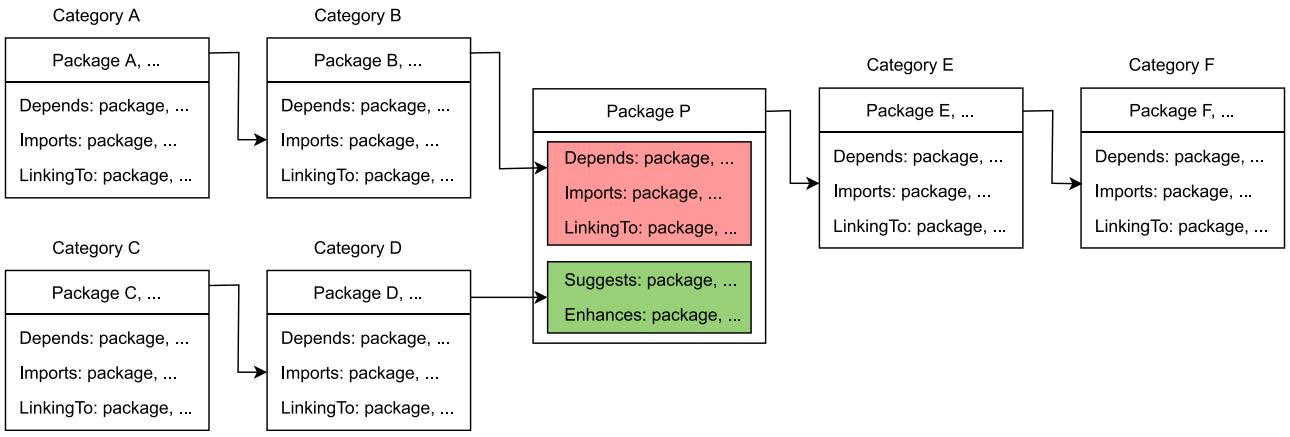


Fig. 2. Demonstration of dependency relations between R packages. Packages in categories A-D are in the upstream of package *P*. Packages in categories E and F are in the downstream of *P*.

Downstream packages: Total packages by recursively looking for child packages (category E and F in Fig. 2). *P* is required for the installation of any of its downstream packages. Note downstream packages include child packages.

Indirect downstream packages: Downstream packages excluding child packages (category F in Fig. 2), i.e., these with distance to *P* of at least 2 in the global dependency graph. These are the packages on which *P* has an indirect influence of dependencies.

Except explicitly clarifying, the term "dependency" always refers to "strong dependency" in the paper.

4. Methods

4.1. Definitions of dependency heaviness

The dependency heaviness in general measures the number of additional dependencies that a parent uniquely brings to its child packages or downstream packages in the ecosystem. Depending on different dependency categories defined in Section 3.3, there are various heaviness metrics listed in the following subsections.

Heaviness from a parent. If package *A* is a strong parent of *P*, the heaviness of *A* on *P* denoted as *h* is calculated as

$$h = n_1 - n_2 \tag{1}$$

where n_1 is the number of strong dependencies of *P* (Fig. 3A), and n_2 is the number of strong dependencies of *P* after changing *A* from a strong parent to a weak parent, i.e., by moving *A* to *P*'s "Suggests" (Fig. 3B). Thus, the heaviness measures the number of additionally required strong dependencies that *A* brings to *P* and are not brought by any other parent. In some discussions in the paper, we explicitly denoted it as $h^{A\to P}$ to indicate the parent–child relation, or h^A if only the parent is of interest.

If package *B* is a weak parent of *P*, n_2 is defined as the number of strong dependencies of *P* after changing *B* to a strong parent of *P*, i.e., by moving *B* to *P*'s "Imports" (Fig. 3E–F). In this scenario, the heaviness of the weak parent is calculated as $n_2 - n_1$.

From the aspect of dependency graph denoted as a directed graph $G = (V, E)$ where *V* is the set of all packages and *E* is the set of all strong parent–child dependency relations in the ecosystem, the heaviness of *A* on *P* is a score associated with an edge $e^{A\to P} \in E$. Now n_1 is the number of upstream packages of *P* ($n_1 = 9$ in Fig. 3C) and n_2 is the number of upstream package of *P* after removing the connection of $A \to P$ from *G* ($n_2 = 6$ in Fig. 3D), thus the heaviness of *A* on *P* is $n_1 - n_2 = 3$. When *B* is a weak parent of *P*, n_2 is calculated as the number of upstream packages after adding a new connection of $B \to P$ to *G* (Fig. 3G–H).

Since weak parents are not necessarily required for *P*, in this paper, we only discussed the heaviness from strong parents.

Max heaviness from parents. Normally, a package has multiple parents. Max heaviness from parents helps to reveal the parent that dominantly brings extra dependencies to package *P*. Assume *P* has K_p parents, the heaviness denoted as h_{max} is defined as

$$h_{max} = \max_{k\in\{1..K_p\}} h_k \tag{2}$$

where h_k is the heaviness of the *k*th parent on *P*.

Heaviness from an upstream package. If package *C* is an upstream package of *P* in the global dependency graph, let n_1 be the number of strong dependencies of *P*, and let n_2 be the number of strong dependencies of *P* after changing *C* to a weak parent of all *C*'s child packages, then the heaviness of *C* on *P* denoted as h_u is calculated as

$$h_u = n_1 - n_2. \tag{3}$$

From the aspect of the dependency graph, n_2 is the number of upstream packages of *P* after removing all edges which start from *C*. When *C* is only a parent of *P* (i.e., distance from *C* to *P* is one), h_u is not always identical to *h*. Assume *C* is a parent of *P* and *P* has another parent *A* where *C* is also a parent of *A*, i.e., with the relations of $C \to P$ and $C \to A \to P$. *h* just measures the local dependency effect which only removes $C \to P$, while h_u is a global dependency effect which removes all links from *C*, i.e., both $C \to P$ and $C \to A$, which results in general $h_u \geq h$. In some discussions in the paper, we denoted it explicitly as $h_u^{C\to P}$ to indicate the upstream–downstream relation.

Heaviness on child packages. Assume *P* has K_c child packages and the *k*th child is denoted as A_k. Denote the number of strong dependencies of A_k as n_{1k}, and denote the number of strong dependencies of A_k after changing *P* to a weak parent of A_k as n_{2k}, the heaviness of *P* on its child packages denoted as h_c is calculated as

$$h_c = \frac{1}{K_c}\sum_{k=1}^{K_c}(n_{1k} - n_{2k}). \tag{4}$$

$n_{1k} - n_{2k}$ is actually the heaviness of *P* on A_k, Eq. (4) can be rewritten as

$$h_c = \frac{1}{K_c}\sum_{k=1}^{K_c} h^{P\to A_k}. \tag{5}$$

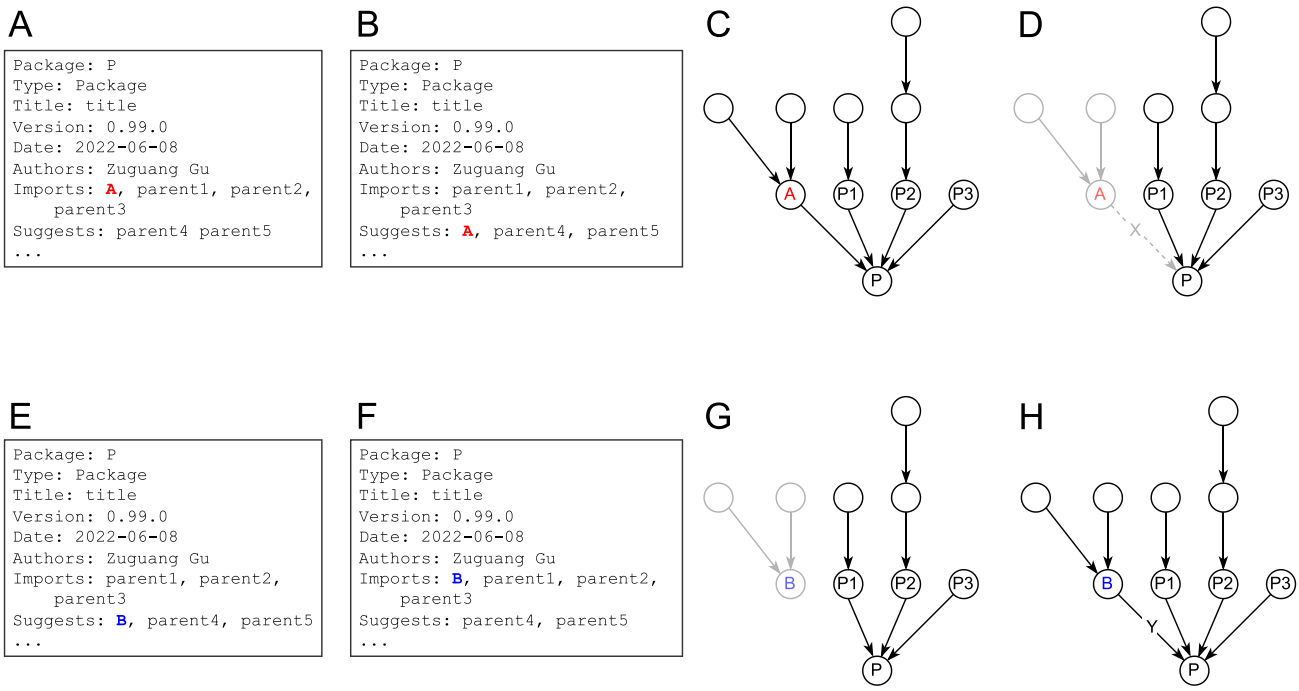


Fig. 3. Demonstration of the heaviness definition. (A) Fragment of the DESCRIPTION file of package *P* where package *A* is a strong parent. (B) Change *A* to a weak parent of *P*, i.e., by moving *A* to *P*'s "Suggests". (C) The dependency graph of all upstream packages of *P*. Note the graph only contains strong dependency relations. (D) Remove the relation between *A* and *P* from the dependency graph to simulate *A* has become a weak parent of *P*. (E) Fragment of the DESCRIPTION file of package *P* where package *B* is a weak parent. (F) Change *B* to a strong parent of *P*, i.e., by moving *B* to *P*'s "Imports". (G) The dependency graph of all upstream packages of *P*. Note since *B* is a weak parent of *P*, there is no connection between *B* and *P*. (H) Add a new connection between *B* and *P* to simulate *B* has become a strong parent of *P*.

The heaviness measures the average number of additional dependencies that *P* brings to its child packages.

Heaviness on downstream packages. The definition is similar to the heaviness on child packages. Assume *P* has K_d downstream packages and the *k*th downstream package is denoted as B_k. Denote the number of strong dependencies of B_k as n_{1k}, and denote the number of strong dependencies of B_k after changing *P* to a weak parent of all *P*'s child packages as n_{2k}. The heaviness of *P* on its downstream packages denoted as h_d is calculated as

$$h_\mathrm{d} = \frac{1}{K_\mathrm{d}} \sum_{k=1}^{K_\mathrm{d}} (n_{1k} - n_{2k}). \tag{6}$$

From the aspect of the dependency graph, n_{2k} is the number of upstream packages of B_k in a reduced graph where *P* is removed. Eq. (6) can be rewritten as

$$h_\mathrm{d} = \frac{1}{K_\mathrm{d}} \sum_{k=1}^{K_\mathrm{d}} h_\mathrm{u}^{P \to B_k} \tag{7}$$

where $h_\mathrm{u}^{P \to B_k}$ is the heaviness of *P* as an upstream package on B_k. In this way, h_c and h_d are not always identical if *P* only has child packages. h_c is a local measure while h_d is a global measure. They have the relation of $h_\mathrm{d} \geq h_\mathrm{c}$.

Heaviness on indirect downstream packages. The calculation is the same as h_d except here child packages are excluded from downstream packages. Denote the heaviness as h_id and denote the set of *P*'s child packages as S_c, h_id is defined as

$$h_\mathrm{id} = \frac{1}{K_\mathrm{d} - K_\mathrm{c}} \sum_{k=1}^{K_\mathrm{d}} (n_{1k} - n_{2k}) \cdot I(B_k \notin S_\mathrm{c}) \tag{8}$$

where K_c and K_d are the numbers of child and downstream packages respectively, and $I()$ is an indicator function. h_id is set to 0 if $K_\mathrm{c} = K_\mathrm{d}$, i.e., *P* has no indirect downstream packages. h_id measures the contribution of dependencies of *P* to the ecosystem in an indirect way.

4.2. Adjusted heaviness

In the Results section (Section 7), we performed dependency heaviness analysis on the CRAN/Bioconductor ecosystem. One of the aims is to prioritize packages which are significantly affected by upstream packages or affect their downstream packages in the ecosystem. If grouping packages by *K* which can be the number of parent, child or downstream packages depending on different types of heaviness metrics, distributions of heaviness values always have long tails, and tails are especially longer for smaller *K* (Figs. 6 and 9). Thus, if simply ranking packages based on the original heaviness values, top packages are preferably associated with small *K*. In general, packages with small *K* are of less interest because they only have very small impacts on the ecosystem. To prioritize packages with broader impacts on the ecosystem, the original definitions of various heaviness metrics are adjusted to decrease the weights of packages with smaller *K*. Please note, the designs of the adjusted heaviness metrics are empirical and the absolute values of adjusted heaviness are meaningless, which are only used for ranking packages.

Adjusted max heaviness from parents. When a package has more parents, dependencies from individual parents would have more overlap (i.e., dependencies from parent *A* overlap to dependencies from parent *B*). Since heaviness only measures the number of unique dependencies that a single parent brings in, or in other words, the number of dependencies that are mutually exclusive to those brought by all other parents, with more parents, the max heaviness from parents would decrease. For a package indexed as *k* in the ecosystem, the original max heaviness from

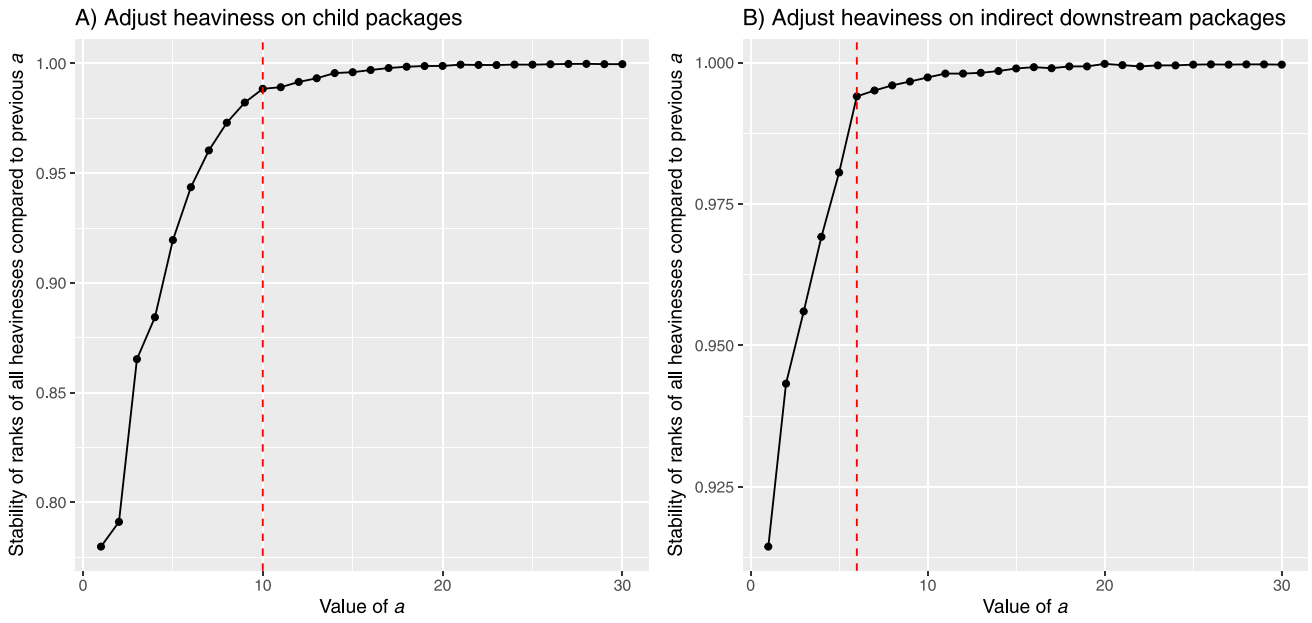


Fig. 4. Select a proper penalty value a for adjusting heaviness values. (A) Adjust heaviness on child packages. (B) Adjust heaviness on indirect downstream packages. Vertical dashed lines are the final selections of the penalty values.

parent denoted as $h_{\max,k}$ is adjusted to $h^{\text{adj}}_{\max,k}$, by multiplying a zooming factor denoted as a_k:

$$h^{\text{adj}}_{\max,k} = a_k \cdot h_{\max,k}. \tag{9}$$

a_k is defined as

$$a_k = (n_k + 30)/n_{\max} \tag{10}$$

where n_k is the number of parents of the kth package, and $n_{\max}$ is the maximal number of parents of all packages in the ecosystem. The value of 30 was selected empirically to balance the zooming rate on different n_k.

Adjusted heaviness on child packages. Generally, heaviness on child packages has a trend that distribution tails are shortened when the numbers of child packages increase (Fig. 9). This is mainly because if package P has more child packages, its child packages may have more other parents which dilute the heaviness from P. To decrease the weights of packages with small numbers of child packages, a positive penalty term denoted as a is added to K_c as in Eq. (11) where K_c is the number of child packages and h^{adj}_c is the adjusted heaviness of a package on its child packages. Note a is set to the same value for all packages.

$$h^{\text{adj}}_c = \frac{1}{K_c + a}\sum_{k}^{K_c}(n_{1k} - n_{2k}) = \frac{K_c}{K_c + a} \cdot h_c. \tag{11}$$

It is easy to see that a decreases h_c faster for smaller K_c than larger K_c. To select an optimized value for a, we took a as integers in the set $\{1, 2, \ldots, 29, 30\}$; and for a specific package indexed as k and a value of a, we calculated the adjusted heaviness on its child packages denoted as $h^{\text{adj}}_{c,k,a}$, and the vector for all packages is denoted as $h^{\text{adj}}_{c,a}$. a is selected as the value by which the ranking of adjusted heaviness of all packages becomes stable. To measure the stability of the ranking in $h^{\text{adj}}_{c,a}$ compared to $h^{\text{adj}}_{c,a-1}$, we calculated the stability score denoted as s_a as

$$s_a = \frac{1}{N}\sum_{k}^{N} I(|R_{k,a} - R_{k,a-1}| \le 50) \tag{12}$$

where N is the total number of packages in the ecosystem, $R_{k,a}$ and $R_{k,a-1}$ are the ranks of package k's adjusted heaviness in the two vectors $h^{\text{adj}}_{c,a}$ and $h^{\text{adj}}_{c,a-1}$ respectively, and $I()$ is the indicator function.

s_a, or its general denotation s, measures the fraction of packages whose ranking differences of adjusted heaviness are no larger than 50 between two neighboring values of a (50 is a small value compared to the total number of R packages in the ecosystem, which is 22,076 in this study). When s becomes stable with a, we could conclude increasing a will not greatly change the ranking in s. In Fig. 4A, we calculated s on a list of a in $\{1, 2, \ldots, 29, 30\}$. By observing the trend of the curve, we can see when $a = 10$ (red vertical in Fig. 4A), s starts to increase slowly. Thus, a was empirically selected to 10.

Adjusted heaviness on indirect downstream packages. The definition is the same as the adjusted heaviness on child packages. For a package P, the adjusted heaviness denoted as $h^{\text{adj}}_{\text{id}}$ is calculated as:

$$h^{\text{adj}}_{\text{id}} = \frac{K_{\text{id}}}{K_{\text{id}} + a} \cdot h_{\text{id}} \tag{13}$$

where K_{id} is the number of indirect downstream packages, i.e., $K_{\text{id}} = K_d - K_c$. The penalty a was selected in a similar way as for child packages. It was selected as $a = 6$ (Fig. 4B).

The heaviness on all downstream packages can be adjusted in the same way. However, in this study, we only looked at the heaviness on indirect downstream packages instead of total downstream packages, thus, here we omitted the definition of adjustment of heaviness on all downstream packages.

4.3. Co-heaviness from parent pairs

Heaviness from parent A on P only measures the number of additional dependencies that A uniquely brings to P. However, there are scenarios where multiple parents import a similar set of dependencies, which results in heaviness from individual parents being very small. Here we define the co-heaviness that measures the number of additional dependency packages simultaneously brought by two parent packages (Fig. 5). Let A and B be two parents of P, let S_A be the set of reduced dependency packages when only changing A to a weak parent of P (Fig. 5D), let S_B be the set of reduced dependency packages when only changing B to a weak parent of P (Fig. 5E), and let S_{AB} be the set of

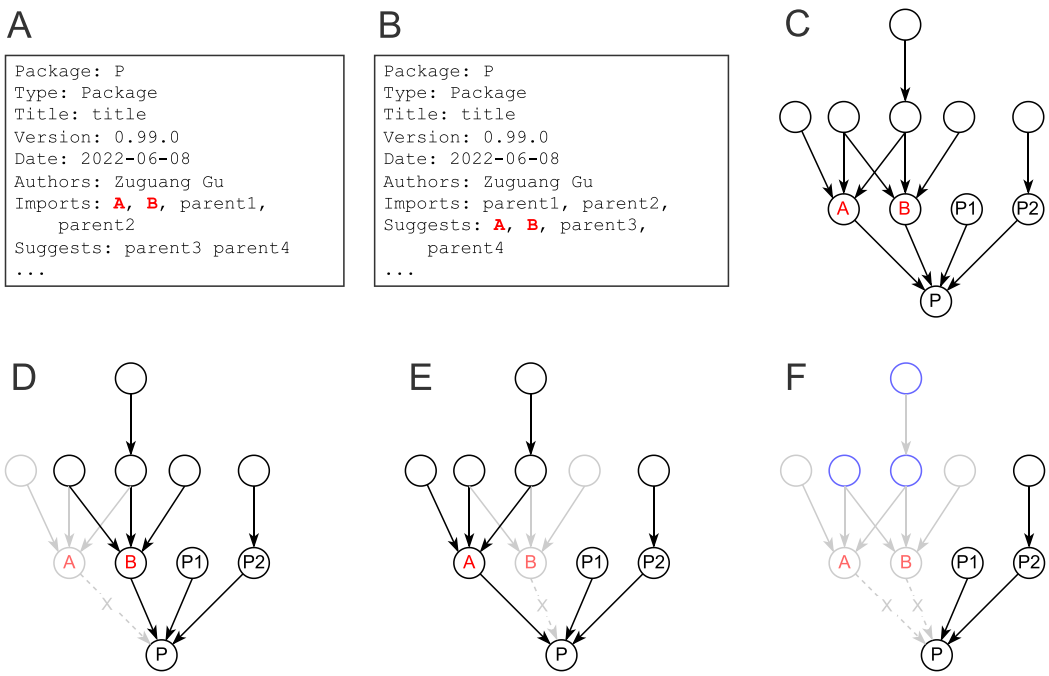


Fig. 5. Demonstration of the co-heaviness definition. (A) Fragment of the DESCRIPTION file of package P where package A and B are two strong parents. (B) Change both A and B to weak parents of P, i.e., by moving A and B to P's "Suggests". (C) The dependency graph of all upstream packages of P. Note the graph only contains strong dependency relations. (D) Remove the relation between A and P from the dependency graph to simulate A has become a weak parent of P. (E) Remove the relation between B and P from the dependency graph to simulate B has become a weak parent of P. (F) Remove the relations from both A and B to P from the dependency graph to simulate A and B have become two weak parents of P. The number of blue nodes in Figure F corresponds to the value of co-heaviness of A and B on P.

reduced dependency packages when changing both A and B to weak parents of P, then the co-heaviness of A and B on P denoted as h_{co} is defined as

$$h_{\text{co}} = |S_{AB} \setminus \cup(S_A, S_B)| \tag{14}$$

where $X \setminus Y$ is the set of elements in X but not in Y, and $|X|$ is the number of elements in set X (Fig. 5F). The co-heaviness measures the number of reduced packages only caused by co-action of A and B. In some discussions in the paper, we denoted it explicitly as $h_{\text{co}}^{(A,B)\to P}$ to indicate the relations.

With the previous denotations, there are

$$h^A = |S_A| \tag{15}$$

$$h^B = |S_B| \tag{16}$$

$$S_A \cap S_B = \emptyset \tag{17}$$

where h^A is the heaviness of A on P, h^B is the heaviness of B on P, and S_A and S_B are mutually exclusive. Then there is the following relation:

$$|S_{AB}| = h_{\text{co}} + h_A + h_B. \tag{18}$$

Thus, the number of reduced dependencies by moving both A and B to P's weak parents is the sum of heaviness of A and B on P individually and the co-heaviness of A and B on P (Fig. 5F).

5. Tools and materials

In our previous study (Gu and Hübschmann, 2022), we have developed an R package *pkgndep* which performs dependency heaviness analysis on single packages. For a given R package whose dependency packages are only from CRAN and Bioconductor, *pkgndep* calculates the dependency heaviness from every of its parent package, additionally with an intuitive heatmap visualization as well as an HTML report, which helps developers to easily reveal heavy parents. To be a companion tool for the study described in this paper, we have updated *pkgndep* to version 1.2.0 with new functionalities for analyzing the R package ecosystem.[13] Given a package P, there are the following functions for querying package dependencies in various dependency categories:

- `parent_dependency()`
- `upstream_dependency()`
- `child_dependency()`
- `downstream_dependency()`

And there are the following functions for calculating the corresponding dependency heaviness metrics:

- `heaviness()`
- `co_heaviness()`
- `heaviness_from_upstream()`
- `heaviness_on_children()`
- `heaviness_on_downstream()`

The function names are self-explanatory. We believe these new functions will be convenient tools for researchers who wish to perform software engineering studies on the R package ecosystem.

The analysis on the R package ecosystem is bound to a certain snapshot of CRAN and Bioconductor. In this study, dependency relations of all CRAN and Bioconductor packages (we call it the "package database") were obtained with the function `available.packages()` on 2022-06-08.[14] This resulted in 18,638 R packages from CRAN, 3,438 packages from Bioconductor (bioc version 3.15), and 124,251 strong dependency relations in the two ecosystems. Various dependency analyses in this study were

[13] https://CRAN.R-project.org/package=pkgndep.

[14] `available.packages()` always returns the metadata of the newest versions of all packages hosted on CRAN/Bioconductor.

applied with the *pkgndep* package with its aforementioned new functions. The result is represented as a table where rows are R packages and columns are various heaviness metrics. The table can be obtained by the function `all_pkg_stat_snapshot()` in *pkgndep*. Network analysis was applied with the package *igraph* (Csardi and Nepusz, 2006) and visualized with Cytoscape (Shannon et al., 2003) and the R package *RCy3* (Gustavsen et al., 2019).

For reproducibility of this study, we have integrated the script for calculating dependency metrics for all packages in the *pkgndep* package and it can be accessed with the command `system.file("extdata", "analysis.R", package = "pkgndep")`. The scripts for the figures in this paper are available at https://github.com/jokergoo/pkgndep_global. *pkgndep* version 1.2.* can be used to reproduce the complete analysis in this study. We plan to regularly update the package database in *pkgndep* to ensure the dependency analysis on R packages is always up-to-date.

6. Research questions

Dependency heaviness is a directional measure. It measures the amount of dependencies uniquely transmitted from a package to its single or total downstream packages. In the context of the complex dependency graph, we separated our research questions (RQs) into three categories according to different dependency directions.

Packages have certain numbers of parents and each parent may bring additional dependencies transitively. We studied the dependency flow from parent to child packages and we first asked the following two research questions:

- **RQ1: What is the general pattern of dependency heaviness from parents?** This includes the following sub questions: 1. What is the proportion of packages suffering from heavy parents and what are they? 2. How is the dependency heaviness accumulated from remote upstream packages?
- **RQ2:** It is very common that a package depends on multiple parents. The second research question is **how do two parents contribute dependencies synergistically?**

For hub packages such as *ggplot2* in the ecosystem, their dependencies are all passed to their downstream packages. With regards to the impacts on downstream packages, we asked the next following two research questions:

- **RQ3: What is the general pattern of dependency heaviness of a package contributing to all its child packages?** This also includes several sub questions: 1. What is the distribution of dependency heaviness on child packages? 2. What are the top packages that spread the highest amount of dependencies directly to their children?
- **RQ4:** A package may transmit dependencies to its remote downstream packages. We next asked **what are the differences between the dependency heaviness transmission to indirect downstream packages and to direct child packages?**

CNA on the dependency graph can reveal interesting structures of dependency transmission in the ecosystem. From the aspect of network analysis, we asked the last research question:

- **RQ5. How are the dependency flows transmitted throughout the ecosystem?** This includes the following sub questions: 1. What is the global attribute and structure of the dependency graph if taking dependency heaviness as the weight ? 2. How deep can the dependency heaviness be transmitted? 3. Does there exist a core graph that transmits the majority of the dependencies in the ecosystem?

When answering these research questions, we also examined the difference between CRAN and Bioconductor to reveal ecosystem-specific patterns.

7. Results

7.1. RQ1: Heaviness from parent packages

A package may have multiple parents. Here we only studied the max heaviness from its parents (abbreviated as MHP) which measures the number of unique dependencies that a package maximally inherits from its parents. In Section 7.1.3, we demonstrated that if a package suffers heavy dependency from its parents, it is very likely that one or only a very few parents contribute heavy dependencies to it. Therefore, MHP is a reasonable metric for studying the general pattern of heaviness from parent packages.

7.1.1. Distribution of MHP

In general, MHP has a long-tail distribution where 82.4% of all packages have a heaviest parent contributing heaviness <30, whereas there are only 878 (4.0%) packages having a heaviest parent contributing heaviness ≥ 60. Therefore, only a small fraction of packages have heavy parents in the ecosystem. For 878 packages with extremely heavy parents (MHP $\geq$ 60), 785 (89.4%) of them only have less than 15 parents, which indicates packages with small numbers of parents are more likely to have heavy parents. Also interestingly, 720 (82.0%) of them do not have child packages, which indicates when packages have heavier parents, they are more likely to be leaf packages in the dependency graph and no other package depends on them.

Distribution tails of MHP are shortened when numbers of parents increase (Fig. 6A). As has been explained in Section 4.1, MHP measures the number of unique dependencies that a single parent maximally brings in. Thus, with more parents, dependencies from multiple parents would have more chances to overlap, which results in decreasing MHP. In Fig. 6A, it can be straightforwardly observed that, on the top edge of the point clouds, there is a clear trend where MHP drops as the number of parents increases.

Globally, Bioconductor packages have heavier parents than CRAN packages. The mean MHP for all Bioconductor packages is 24.6 while it is only 13.3 for all CRAN packages (Table 1. The median values are 16 vs. 6). Nevertheless, the difference of MHP in the two repositories becomes smaller when the number of parents increases. E.g., when only considering packages with parents >20 (580 packages left), the mean MHP values are 27.2 vs. 22.8 for CRAN and Bioconductor packages, and the median values are 23 vs. 19.

Bioconductor packages suffer heavier parents than CRAN packages. The reason might be that Bioconductor packages are mainly for biological data analysis and many of them integrate various analyses and annotation resources from upstream packages (average numbers of parents are 5.1 and 8.4 for CRAN and Bioconductor, Table 1). Thus, it is easier for dependency heaviness to be accumulated from upstream on Bioconductor (average numbers of strong dependencies are 30.8 and 66.1 for CRAN and Bioconductor, Table 1). For example, the Bioconductor package *miRspongeR* listed as a top package in Fig. 6A has an upstream package *SPONGE* which integrates analysis from a list of heavy parents for different analysis aims (*tidyverse* for data processing, *ggpubr* for visualization, *biomaRt* for obtaining biological annotation data, *caret* for statistical modeling).

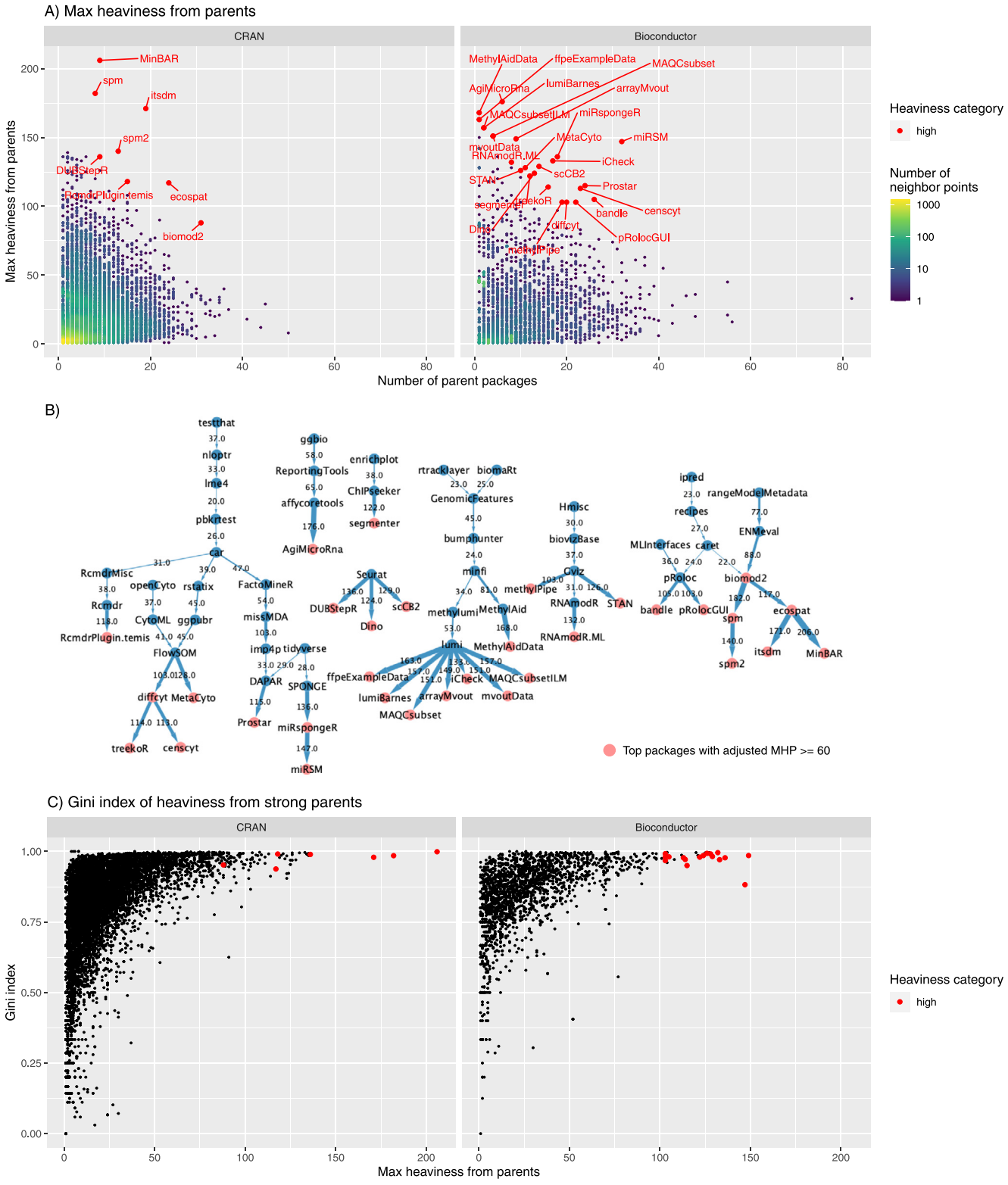


Fig. 6. Heaviness from parent packages. (A) Distribution of MHP. (B) Upstream of packages with adjusted MHP $\geq$ 60. Only edges with heaviness $\geq$20 are included. Values on the edges are the dependency heaviness of a parent on its child package. (C) Gini index of the heaviness from parents. In Figure A, since points may overlap, we visualized the two-dimensional distribution densities where numbers of neighbor points were based on circular areas with radius of 1% of ranges on *x*-axis and *y*-axis respectively. The packages highlighted in red in Figure A are the same as in Figure B and C, which are the top packages with adjusted MHP $\geq$ 60. MHP: max heaviness from parents.

7.1.2. Top packages with the highest MHP

If a package has more upstream dependencies, it is more vulnerable to corruptions from its upstream packages. We identified top packages with the highest MHP and these top packages have the most risky parents in the ecosystem from the aspect of how they uniquely contribute risks to their child packages.

Packages with smaller numbers of parents have longer distribution tails of MHP. To capture top packages with the heaviest

Table 1
Average values of various metrics of packages on CRAN and Bioconductor. N_{child}: number of child packages; N_{indirect}: number of indirect downstream packages.

Metrics averaged in the ecosystem	CRAN	Bioconductor
Number of strong dependencies	30.8	66.1
Number of parents	5.1	8.4
Max heaviness from parents	13.3	24.6
Max co-heaviness from parents	4.5	12.2
Number of children	4.7	3.5
Number of children (with $N_{\text{child}} > 0$)	18.2	15.2
Heaviness on child packages (with $N_{\text{child}} > 0$)	7.8	14.8
Number of indirect downstream	29.0	11.5
Number of indirect downstream (with $N_{\text{indirect}} > 0$)	256.8	136.5
Heaviness on indirect downstream packages (with $N_{\text{indirect}} > 0$)	4.4	8.3

parents but not biased by the small numbers of parents, the original MHP was adjusted. Top packages that have extremely heavy parents are filtered by adjusted MHP $\geq$ 60 and they are marked in red in Fig. 6A. We found there are 8 CRAN packages and 24 Bioconductor packages that have extremely heavy parents. This also indicates Bioconductor packages may suffer more from heavy parents. Among them, the package *MinBAR* has the highest MHP with the heaviest parent named *ecospat* which uniquely contributes 206 additional dependencies to *MinBAR*.

When a package suffers from an extremely heavy parent, the next question naturally to ask is how are the dependencies accumulated from the upstream of the heavy parent? Fig. 6B illustrates upstream dependencies of the top 32 packages with adjusted MHP $\geq$ 60. To keep the graph small and compact, parent–child with dependency heaviness $\geq$20 are only included in the graph. In other words, the graph in Fig. 6B contains major dependency flows from upstream to the packages suffering heavy parents. Interestingly, for most packages, high dependencies are accumulated from upstream in very long ranges. The longest transmission path in the graph has a length of 9, e.g., from *testthat* to *treekoR* or *censcyt*. As a comparison, the average distance in the global dependency graph is only 2.6. We also found heavy dependencies can be inherited from the same parents, such as *Seurat* transmitting on average 130 unique dependencies to its 3 child packages and *lumi* transmitting on average 152 unique dependencies to its 7 child packages. The transmission of heaviness will be further discussed in Section 7.3 on the short-range transmissions and in Section 7.5 on the long-range transmissions.

7.1.3. Uniqueness of the heaviest parents

A package may have multiple parents. We observed a general pattern that only a small number of parents contribute large heaviness while the majority of other parents only contribute very small heaviness to their child packages. The Gini index was used to quantitatively measure the dispersion of the heaviness distribution from a package's parents. When the Gini index is close to 1, there is a uniquely high heaviness value; and when the Gini index is close to zero, the heaviness values approximately follow a uniform distribution. Fig. 6C shows that there is a clear trend that when MHP increases, i.e., when the parents get heavier, the Gini indices increase as well. For the top packages with extremely high MHP, the corresponding Gini indices are very close to 1. This indicates these heaviest parents play unique and dominant roles in contributing dependencies to their child packages.

Answer to RQ1

Only a small fraction of R packages in the ecosystem suffer from heavy parents. The heavy dependencies on them are normally accumulated from far upstream. If a package suffers from heavy parents, it is very likely the heaviest parent plays a unique and dominant role in contributing dependencies to it.

7.2. RQ2: Co-heaviness from two parent packages

Heaviness discussed in Section 7.1 only measures the number of dependencies that a single parent uniquely brings in. However, there are scenarios where multiple parents import similar sets of dependencies, which results in heaviness from individual parents being very small. Taking the package *DESeq2* (version 1.36.0) as an example, its two parent packages *geneplotter* and *genefilter* import 51 and 53 dependencies respectively, among which 50 are the same.[15] Due to the high overlap, the heaviness of *geneplotter* and *genefilter* on *DESeq2* are only 1 and 2. In this section, we studied the dependency heaviness simultaneously imported by two strong parents, i.e., the co-heaviness. In the analysis, for a package, we only looked at the maximal co-heaviness from all its parent pairs (abbreviated as MCoHP). Empirically, MCoHP is dramatically higher than the co-heaviness from other pairs if a package has an obvious heavy parent pair. For the ease of discussion, we name the parent pair which contributes the highest co-heaviness as the *MCoHP parents* in the paper.

7.2.1. Relations of parent pairs

In the CRAN/Bioconductor ecosystem, we found 92.6% of all packages have MCoHP $<$ 20, and only 533 (2.4%) of them (143 from CRAN and 390 from Bioconductor) have MCoHP $\geq$ 40, which indicates the proportion of packages whose heavy dependencies are simultaneously inherited from two parents are extremely small in the ecosystem. Denote two parents of package P as A and B, the relations of parent pairs can be summarized into the following four categories:

1. **Parent–child**. E.g., B is a parent of A where A inherits all dependencies from B.
2. **Upstream–downstream**. E.g., B is an indirect upstream package of A where A also inherits all dependencies from B.
3. **Common-upstream**. There exists a common upstream package C of A and B where C contributes heavy dependencies to both A and B, defined as $h_{\text{u}}^{C\to A} > 0.75h_{\text{co}}^{(A,B)\to P}$ and $h_{\text{u}}^{C\to B} > 0.75h_{\text{co}}^{(A,B)\to P}$. This means the co-heaviness of A and B mainly comes from C.
4. **No clear relation**. The dependencies of A and B are accumulated from their upstream packages independently; or A and B may have common upstream, but the upstream packages do not contribute significantly high heaviness on them.

For the 533 top packages with MCoHP $\geq$ 40, in 486 (91.2%) packages, MCoHP parents are in parent–child relation; in 4 (0.75%) packages, MCoHP parents are in upstream-downstream relation; in 13 (2.4%) packages, MCoHP parents are in common-upstream relation; and in 30 (5.6%) packages, MCoHP parents have no clear relation (Fig. 7).

[15] The heaviness analysis on *DESeq2* can be accessed in the heaviness database introduced in Section 8.

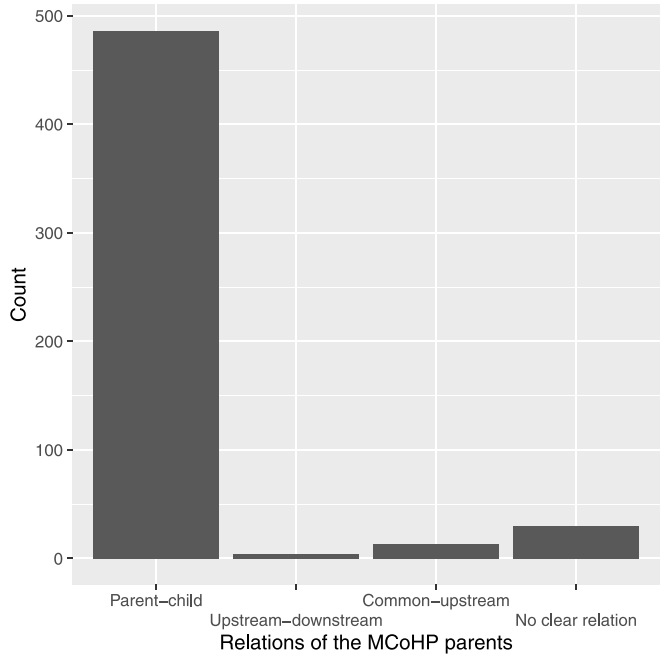


Fig. 7. Relations of the MCoHP parents of the 533 packages with MCoHP $\geq$ 40. MCoHP: max co-heaviness from parents.

7.2.2. Parent pairs that contribute high co-heaviness

Since most MCoHP parents are in parent–child relations, we next studied whether there is preference of selecting a member of the MCoHP parents. Again only in the 533 packages with MCoHP $\geq$ 40, we found in the Bioconductor ecosystem, the package *AnnotationDbi* dominantly contributes high co-heaviness to its child packages as a MCoHP parent (Fig. 8A–B). Top four companion packages co-working with *AnnotationDbi* are *org.Hs.eg.db* (affecting 79 child packages), *org.Mm.eg.db* (affecting 51 child packages), *GenomicFeatures* (affecting 48 child packages) and *org.Rn.eg.db* (affecting 35 child packages) (Fig. 8A). *AnnotationDbi* is a base Bioconductor package which defines the general database interface, and a large number of other packages storing specific biological annotation data are implemented with this interface. To use a specific annotation package from Bioconductor, methods from *AnnotationDbi* should also be used for extracting corresponding data. Therefore, the data packages (e.g., *org.Hs.eg.db* for human, *org.Mm.eg.db* for mouse and *org.Rn.eg.db* for rat) and *AnnotationDbi* are normally used together. Similarly, *AnnotationHub* and *ExperimentHub* provide similar functionalities for storing external annotation and experimental data. *ExperimentHub* inherits the same set of function interface from *AnnotationHub*, thus any package depending on *ExperimentHub* would also depend on *AnnotationHub*. These two "**Hub*" packages are becoming standard ways for handling external data and they are used more and more frequently in the Bioconductor system. Fig. 8B shows the number of child packages of the MCoHP parent packages. Besides the aforementioned three "*org.*.eg.db*" packages, *AnnotationDbi* co-works with in total 18 packages on Bioconductor and it affects 235 child packages with mean co-heaviness of 44. There are other examples of heavy MCoHP parents (Fig. 8B). The package *GenomicFeatures* co-works with 9 other packages where *GenomicFeatures* provides tools for manipulating gene/transcript annotations and the companion packages are organism-specific annotation packages. The package *BSgenome* co-works with 30 other packages where *BSgenome* provides classes and methods for dealing with genome sequences, and the companion packages are also organism-specific annotation packages.

As has been demonstrated previously, top packages contributing strong co-heaviness in the Bioconductor ecosystem are all annotation-related packages. We next explored Bioconductor by excluding the eight top annotation-related packages listed in Fig. 8B. We aimed to study the co-heavines patterns for the "software packages" on Bioconductor. Fig. 8C shows now there is almost no dominant parent pair contributing strong co-heaviness, except the pair *CAMERA* and *xcms*. These two packages are both for mass spectrometry data analysis and they contribute a mean co-heaviness of 77 to their 5 child packages. *CAMERA* and *xcms* provide core functionalities for mass spectrometry data analysis and additional packages in the field might need to depend on both of them. Fig. 8D lists top Bioconductor software packages as heavy co-parents. Notably, the package *Gviz* which provides a general visualization framework for genomics data affects 6 child packages as a co-parent with other 4 comparison packages. It contributes on average a co-heaviness of 88 which is much higher than other co-parent packages.

For CRAN packages, there are less package pairs that contribute strong co-heaviness to their child packages (average MCoHP are 4.5 and 12.2 for CRAN and Bioconductor, Table 1). There is only one parent pair *Hmisc* and *rms* that contribute strong co-heaviness on their 9 child packages with average co-heaviness of 57 (Fig. 8E). Also, *Hmisc* affects 20 child packages as a co-parent with other 9 companion packages (Fig. 8F) where *Hmisc* is a parent of all its companion packages.

Answer to RQ2

Only a small fraction of R packages in the ecosystem inherit dependencies uniquely and simultaneously from two parents. The two parents contributing high co-heaviness mostly have the relation of parent–child, thus they provide a similar set of dependencies. There are more Bioconductor packages having high co-heaviness from parents where one parent provides an interface for manipulating biological data and the other parent provides data for specific organisms with that interface. On CRAN, there are very few packages suffering from high co-heaviness from parents.

7.3. RQ3: Heaviness on child packages

We studied how a package *P* contributes dependency heaviness to its child packages (abbreviated as HC) by looking at the average number of dependencies uniquely brought by *P*.

7.3.1. Distribution of HC

Fig. 9A illustrates the distribution of HC for 5,593 (25.3%) packages with at least one child. In general, distributions of HC have very long tails for packages with small numbers of children, and distribution tails are shortened dramatically as numbers of children increase. This is mainly because when *P* has more child packages, the heaviness on child packages may be diluted by their other parents.

On average, CRAN packages have slightly more children than Bioconductor (18.2 vs. 15.2, Table 1), but with smaller average HC (7.8 vs. 14.8, Table 1).[16] In Section 7.1, we have demonstrated Bioconductor packages have more parents and inherit more strong dependencies, thus it is expected that they on average transmit more dependency heaviness to their children.

[16] Statistics are based on packages with at least one child package.

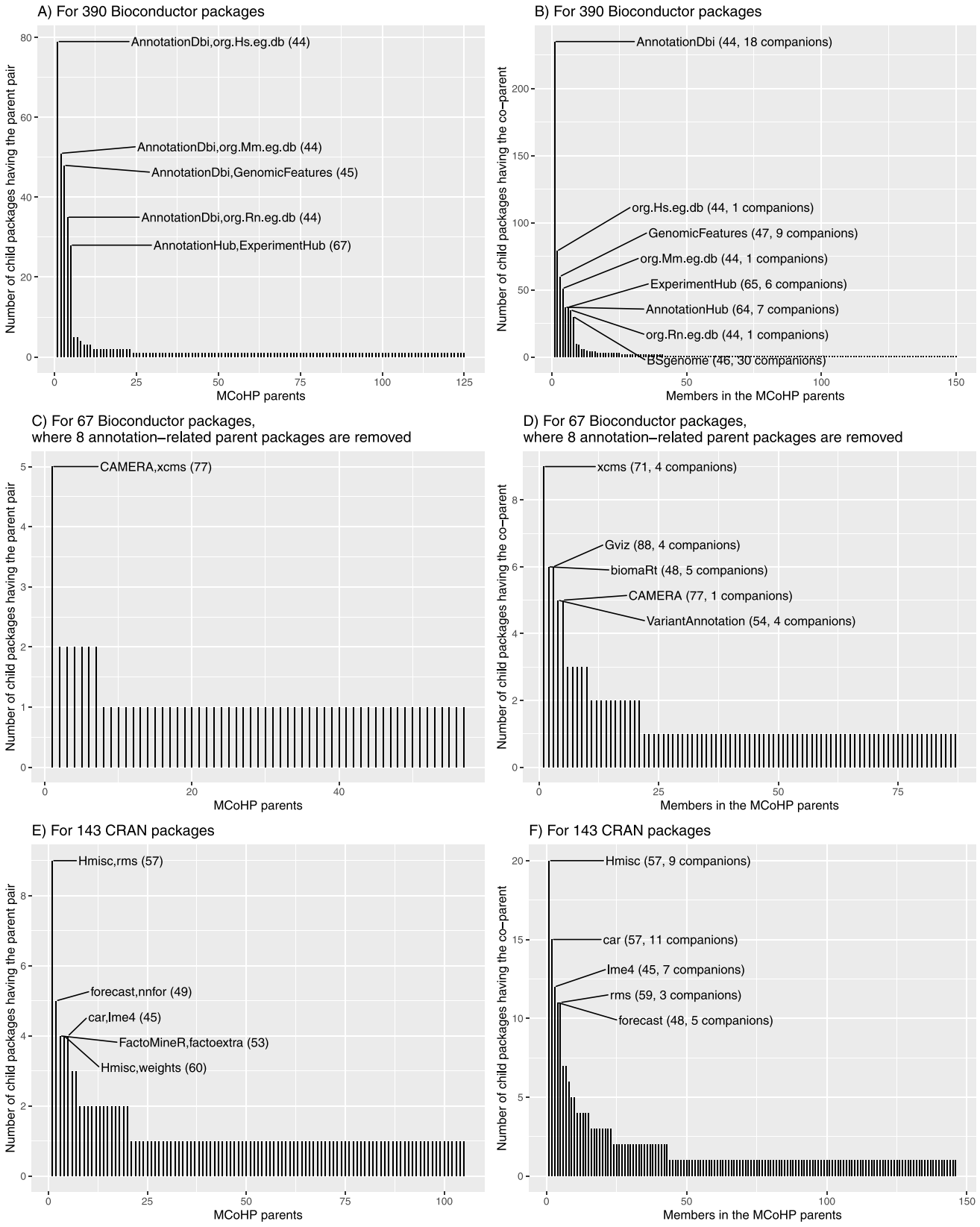


Fig. 8. Co-heaviness in the CRAN/Bioconductor ecosystem. (A, C, E) Number of child packages for each MCoHP parent pair. A: all 390 top Bioconductor packages; C: the top Bioconductor packages of which MCoHP parents do not include the annotation packages listed in Figure B; E: all 143 top CRAN packages. In Figure A, C, E, numbers in the parentheses are the average co-heaviness. (B, D, F) Number of child packages for packages as a member of MCoHP parents. In Figure B, D, F, numbers in the parentheses are the average co-heaviness and numbers of companion packages. All analyses are restricted to 533 packages with MCoHP $\geq$ 40. MCoHP: max co-heaviness from parents.

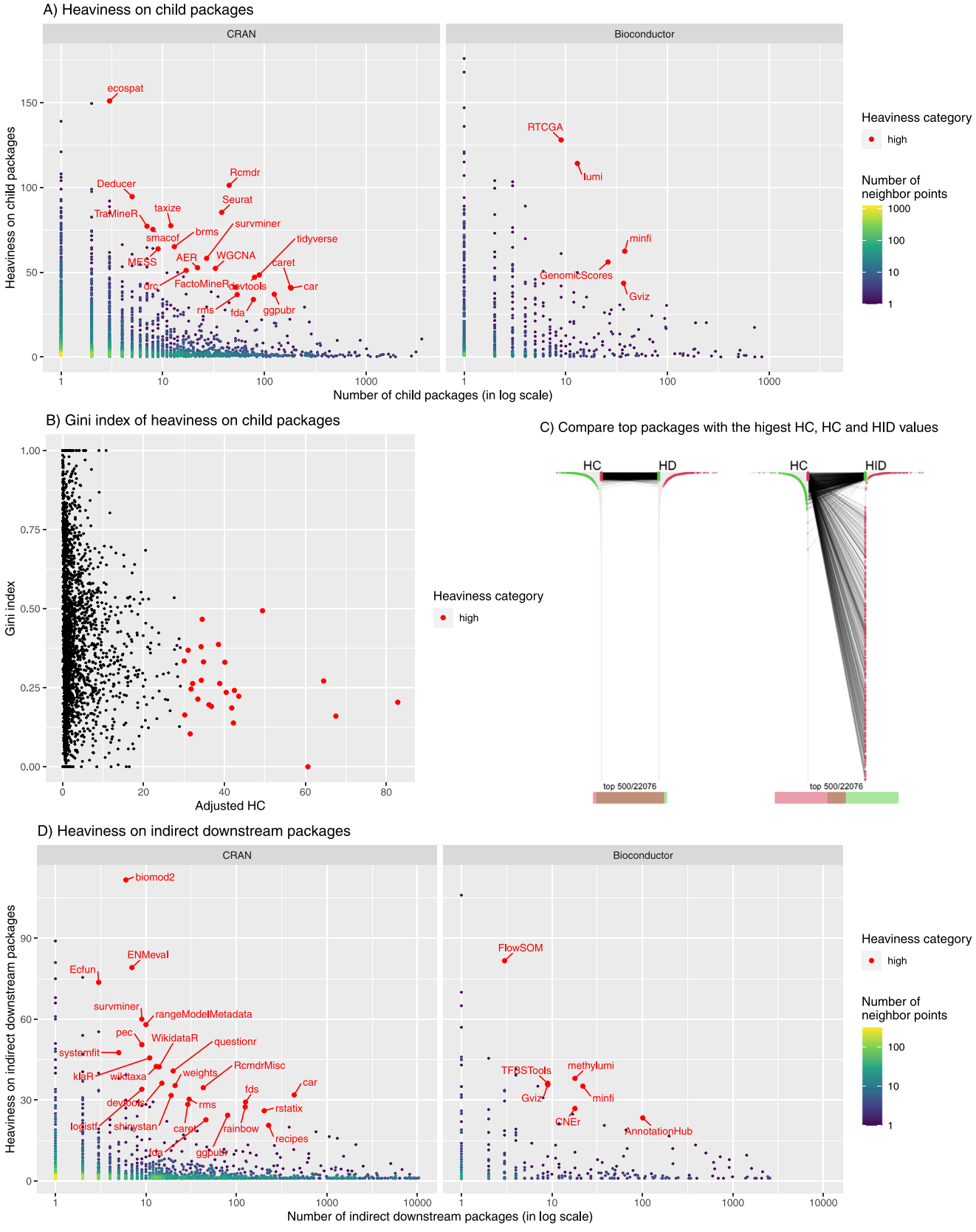


Fig. 9. Heaviness analysis on child and indirect downstream packages. (A) Heaviness on child packages. (B) Gini indices of individual heaviness values on child packages. (C) Compare top 500 packages with the highest HC, HD and HID values. In each plot, packages are sorted by two metrics, e.g., HC and HD in the left plot. Top 500 packages by either of the two metrics are highlighted by segments that connect the same packages in the two sorted lists. At the bottom is a Venn diagram showing the overlap of the top 500 packages from the two metrics. (D) Heaviness on indirect downstream packages. HC: heaviness on child packages; HD: heaviness on downstream packages; HID: heaviness on indirect downstream packages.

Table 2
Top packages with adjusted HC $\geq$ 30. N_{strong}: number of strong dependencies; N_{child}: number of child packages; HC: heaviness on child packages.

Package	N_{strong}	N_{child}	HC	Repository	Package	N_{strong}	N_{child}	*HC*	Repository
ecospat	232	3	151.0	CRAN	GenomicScores	98	26	56.0	Bioconductor
RTCGA	127	9	128.0	Bioconductor	AER	92	22	52.6	CRAN
lumi	162	13	114.2	Bioconductor	WGCNA	108	33	52.3	CRAN
Rcmdr	135	45	101.2	CRAN	drc	96	17	51.1	CRAN
Deducer	107	5	94.6	CRAN	tidyverse	107	89	48.4	CRAN
Seurat	145	38	85.3	CRAN	devtools	76	80	47.0	CRAN
taxize	127	12	77.4	CRAN	Gviz	142	37	43.5	Bioconductor
TraMineR	100	7	77.1	CRAN	FactoMineR	104	52	41.2	CRAN
smacof	122	8	75.2	CRAN	caret	81	180	41.0	CRAN
brms	123	13	65.1	CRAN	car	87	183	40.6	CRAN
MESS	84	9	63.8	CRAN	ggpubr	96	125	37.0	CRAN
minfi	141	38	62.4	Bioconductor	rms	78	54	36.8	CRAN
survminer	115	27	58.2	CRAN	fda	60	78	33.9	CRAN

7.3.2. Top packages with the highest HC

From the perspective of R package developers, HC is especially useful because it also measures the expected number of additional dependencies if *P* is added as a new strong parent of their packages. Thus it is important to list top packages that contribute high heaviness in the ecosystem. Similarly, we also adjusted original HC and we set adjusted HC $\geq$ 30 to extract top packages that broadly affect a large number of child packages in the ecosystem (Fig. 9A and Table 2). Not surprisingly, these top packages already inherit large amount of dependencies from their parents (the column N_{strong} in Table 2), and most of these dependencies are transmitted to their child packages uniquely (the column N_{child} in Table 2). To list a few, the package *lumi* has 162 dependencies from its parents, and on average 114 (70.4%) of them are uniquely contributed to its child packages; the package *RTCGA* has 127 dependencies from its parents, and all of them uniquely go to its child packages. Also as listed in Table 2, there are several packages transmitting large heaviness to broader sets of child packages, such as the package *caret* contributing on average 41 additional dependencies to its 180 child packages, the package *car* contributing on average 40.6 additional dependencies to its 183 child packages, and the package *ggpubr* contributing on average 37.0 additional dependencies to its 125 child packages. These packages play major roles in contributing heavy dependencies to their child packages in the ecosystem.

Expectedly, many of the top packages listed in Table 2 are popular and widely applied in various fields such as general statistical modeling (*car*, *caret*, *fda*, *rms* and *FactoMineR*), specific analysis approaches on biological data (*WGCNA* and *Seurat*), data visualization (*ggpubr*, *Gviz* and *survminer*) and infrastructure-related applications (*devtools* and *tidyverse*), thus they are depended-on by a great number of child packages. However, developers who depend on these top packages should be aware of the potential risks that they introduce.

7.3.3. How do package dependencies transmit to children?

Package *P* may have multiple child packages. We explored the distribution of individual heaviness values of *P* on all its child packages. In Fig. 9B, when packages have high adjusted HC values, i.e., ranked as top packages affecting their child packages, Gini indices of the heaviness values get close to 0.2. It indicates that top heavy packages contribute dependencies almost evenly to their child packages.

Answer to RQ3

In general, the average unique dependencies that a package contributes to all its child packages decrease when it has more children. HC metric is important especially for developers because it also measures the expected number of additional dependencies if *P* is added as a new strong parent of their packages. We found many of the top packages with extremely high HC are already popular in use and we suggest developers pay more attention to these packages if they want to add them as new parents of their packages.

7.4. RQ4: heaviness on indirect downstream packages

HC is a metric of local relation, i.e., the direct parent–child relation. We next studied how dependency heaviness is uniquely transmitted to remote downstream of the global dependency graph.

7.4.1. Remove child packages from downstream

We observed that, for top packages with the highest heaviness on downstream packages (abbreviated as HD), they tend to also have high HC values. In the left panel of Fig. 9C, 478 packages are common in the top 500 packages with the highest HC values and the top 500 packages with the highest HD values. This implies, for these 478 packages, the downstream packages are mainly composed of child packages. For this reason, to study the long-range dependency transmission, we removed child packages from downstream packages, i.e., we only studied the heaviness on indirect downstream packages (abbreviated as HID). In the right panel of Fig. 9C, now there are only 129 packages in common in the top 500 packages by HC and by HID, and these 129 packages contribute strong heaviness to both their direct child packages and remote downstream packages.

7.4.2. Compare packages with top HC and HID

Fig. 9D illustrates the distributions of HID for 2,396 (10.9%) packages with at least one indirect downstream package. Similarly, distributions also have long tails, which indicates only a small fraction of packages spread large dependencies to the ecosystem in the long ranges. In Fig. 9D, we highlighted packages with adjusted HID $\geq$ 20. Compared to Fig. 6A, the list of top packages changes. Table 3 lists top packages with adjusted HC $\geq$ 30 or adjusted HID $\geq$ 20. To reduce the length of the table, we additionally set the numbers of downstream packages (i.e., numbers of child packages + numbers of indirect downstream packages) $\geq$30. In the 21 packages listed in Table 3, they can be summarized into three categories: (i) having both top HC and HID values (9 packages); (ii) only having top HID values (7 packages); (iii) only having top HC values (5 packages). In the first category,

Table 3
Top packages with adjusted HC $\geq$ 30 or adjusted HID $\geq$ 20. Additionally packages listed in the table have numbers of downstream packages (i.e., numbers of child packages + numbers of indirect downstream packages) $\geq$30. N_{child}: number of child packages; HC: heaviness on child packages; N_{id}: number of indirect downstream packages; HID: heaviness on indirect downstream packages.

Package	N_{child}	HC	N_{id}	HID	Is it a top HC package?	Is it a top HID package?
car	183	40.6	435	31.9	y	y
recipes	26	9.9	227	20.6		y
rstatix	10	12.1	203	26.0		y
caret	180	41.0	29	28.4	y	y
ggpubr	125	37.0	80	24.4	y	y
AnnotationHub	97	16.7	101	23.4		y
rainbow	5	21.0	125	27.5		y
fds	2	37.0	126	29.3		y
fda	78	33.9	46	22.7	y	y
devtools	80	47.0	15	36.3	y	y
tidyverse	89	48.4	5	38.0	y	
FactoMineR	52	41.2	37	18.5	y	
rms	54	36.8	30	30.3	y	y
minfi	38	62.4	22	35.2	y	y
RcmdrMisc	7	19.3	43	34.6		y
Gviz	37	43.5	9	35.7	y	y
Rcmdr	45	101.2	1	81.0	y	
Seurat	38	85.3	2	54.0	y	
survminer	27	58.2	9	60.0	y	y
WGCNA	33	52.3	2	24.5	y	
Weights	12	30.6	21	35.4		y

packages not only affect their direct child packages but also affect indirect downstream packages. An example is the package *car* which directly affects 183 child packages (with HC of 40.6) and additionally 435 indirect downstream packages (with HID of 31.9, Fig. 10A). In the second category, packages have very few direct child packages, but child packages play as hub packages to transmit the heaviness to a great number of indirect downstream packages. One typical example is the package *rstatix*, which only has 10 child packages, but affects 203 indirect downstream packages. Fig. 10B demonstrates the heaviness is mainly transmitted from *rstatix* via a hub child package *ggpubr* which spreads the dependency flow to 96.2% of downstream packages of *rstatix*. In the third category, packages mainly contribute heaviness only till their child packages and majority of the transmission do not go further deep in the dependency graph, e.g., the package *tidyverse* (Fig. 10C). The downstream dependency graph for any package on CRAN/Bioconductor can be accessed in the heaviness database in *pkgndep* which will be introduced in Section 8.

Similar to HC, CRAN packages have more indirect downstream packages than Bioconductor (256.8 vs. 136.5, Table 1), but CRAN packages on average transmit less dependency heaviness to their indirect downstream packages (4.4 vs. 8.3, Table 1).[17] Nevertheless, there are more packages with top HID (adjusted HID $\geq$ 20) on CRAN than on Bioconductor (25 vs. 7).

Answer to RQ4

Only a small fraction of packages spread large dependencies in the ecosystem in the long ranges. There are three modes of the dependency transmission: (i) Dependency heaviness is broadly transmitted to both direct child packages and remote downstream packages; (ii) Dependency heaviness is transmitted to remote downstream via hub child packages; (iii) Dependency heaviness is mainly transmitted locally where their child packages are the ends of the dependency transmission.

7.5. RQ5: Dependency graph analysis

In previous sections, the analysis was focused on different heaviness metrics on individual packages. In this section, we studied the dependency relations from the aspect of the global dependency graph by applying the complex network analysis approaches on it.

7.5.1. The general patterns of heaviness spreads in the dependency graph

Denote the global dependency graph as $G = (V, E)$, where V is the set of all packages in the CRAN/Bioconductor ecosystem, E is the set of strong dependency relations, and G is directed, then the heaviness from a parent on a child is a score associated to an edge in the graph. Fig. 11A illustrates the distribution of heaviness which can be approximated as a stretched exponential distribution (Elton, 2018) fitted as

$$\Pr(H = h) = 0.46 \cdot \exp(-1.66h^{0.37}) \tag{19}$$

where H is the random variable of the heaviness. The model implies in the ecosystem, there are only a very small amount of dependency transmissions that are heavy from parent to child packages. The 95th percentile of all heaviness values is 20.

7.5.2. Total dependency heaviness to the ecosystem

In Sections 7.3 and 7.4, we studied the average heaviness on child packages and indirect downstream packages. Here we looked at the total heaviness that a package contributes to the whole ecosystem, i.e., to all its downstream packages. Using the same denotation in Eq. (6), the total heaviness of package P on all its downstream packages is calculated as $h_d \cdot K_d$. The total heaviness also measures the number of reduced dependencies in the whole ecosystem if P is removed. A package with small HD may have a large total effect simply because it has a huge number of downstream packages. For example, the package *stringr* only has a small HD of 2.3, but it affects 5,276 downstream packages, which makes it a top package contributing dependencies in the ecosystem. Therefore, we only looked at packages with HD $\geq$ 10 (1,247 packages left). Fig. 11B illustrates *ggplot2* is the most influential package that affects 5622 downstream packages with a HD value of 14.7. It uniquely contributes in total 82,830 dependencies to the ecosystem. Another example is the package *car* which affects 618 downstream packages but with a higher HD value of 35.6, uniquely contributing in total 20,001 dependencies. All of the top 20 packages (filtered by total heaviness on downstream $\geq$5000) listed in Fig. 11B affect a large number of downstream packages, with an average number of 882. The top 20 packages

[17] Statistics are based on packages with at least one indirect downstream package.

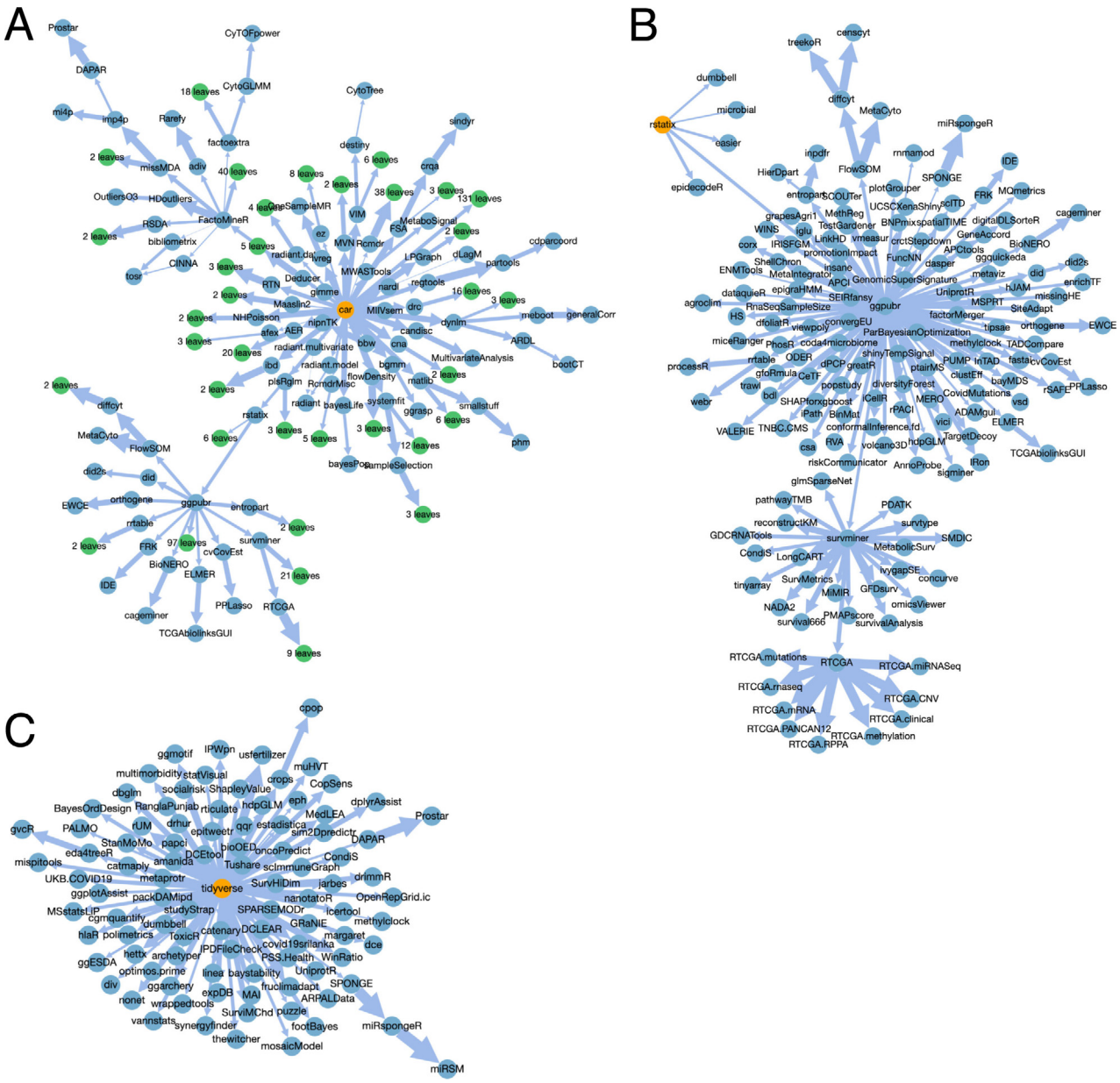


Fig. 10. Three examples of dependency graphs of how packages transmit dependencies heaviness to their downstream packages. (A) The package *car*. (B) The package *rstatix*. (C) The package *tidyverse*. The three packages are colored in orange in the graphs. In Figure A, to reduce the size of the graph for visualization, leaf nodes sharing the same parent are grouped and colored in green. Note the three graphs are directed where *car*, *rstatix* and *tidyverse* are the root nodes.

(0.1% of all packages) contribute 18.9% unique dependencies to the whole ecosystem. This suggests if developers of these top packages can manage to reduce dependencies of their packages, it will greatly reduce the risks they bring to the whole ecosystem.

7.5.3. Depth of the dependency heaviness transmission

In Section 7.1, we observed dependency heaviness on top packages with the highest MHP are accumulated from very remote upstream. Here we studied dependency heaviness transmission for all packages in the ecosystem. For each package, we calculated its depth in the graph as the maximal distance from all its upstream packages where the distance between two packages is the length of the shortest path in the directed graph. Fig. 11C illustrates the distribution of MHP at each depth, and Fig. 11D illustrates the number of packages at each depth. The two figures clearly show, when a package locates more downstream in the graph, it receives larger dependency heaviness from its parent.

7.5.4. Core graph

To study how the dependency heaviness is transmitted in the global dependency graph, we constructed a core graph where heaviness from a parent on a child is $\geq$30. The core graph includes 4,302 packages (19.5%), 3,950 strong dependency relations (3.2%) and 44.2% heaviness flows of the complete graph (measured as the fraction of total heaviness in the core graph and in the global graph).

The core graph is visualized in Fig. 12A. It can be easily observed that there are several hub packages that transmit large heaviness to their direct child packages. The top two packages *ggplot2* and *AnnotationDbi* contribute heavy dependencies to their 442 and 273 child packages directly. We also observed there are a huge number of isolated and small graph components (i.e., maximally connected subgraphs) where heaviness is only transmitted locally. Out of the total 379 graph components, 352 components only have size $\leq$10; however, the other 27 components include 77.0% of packages in the core graph. The largest

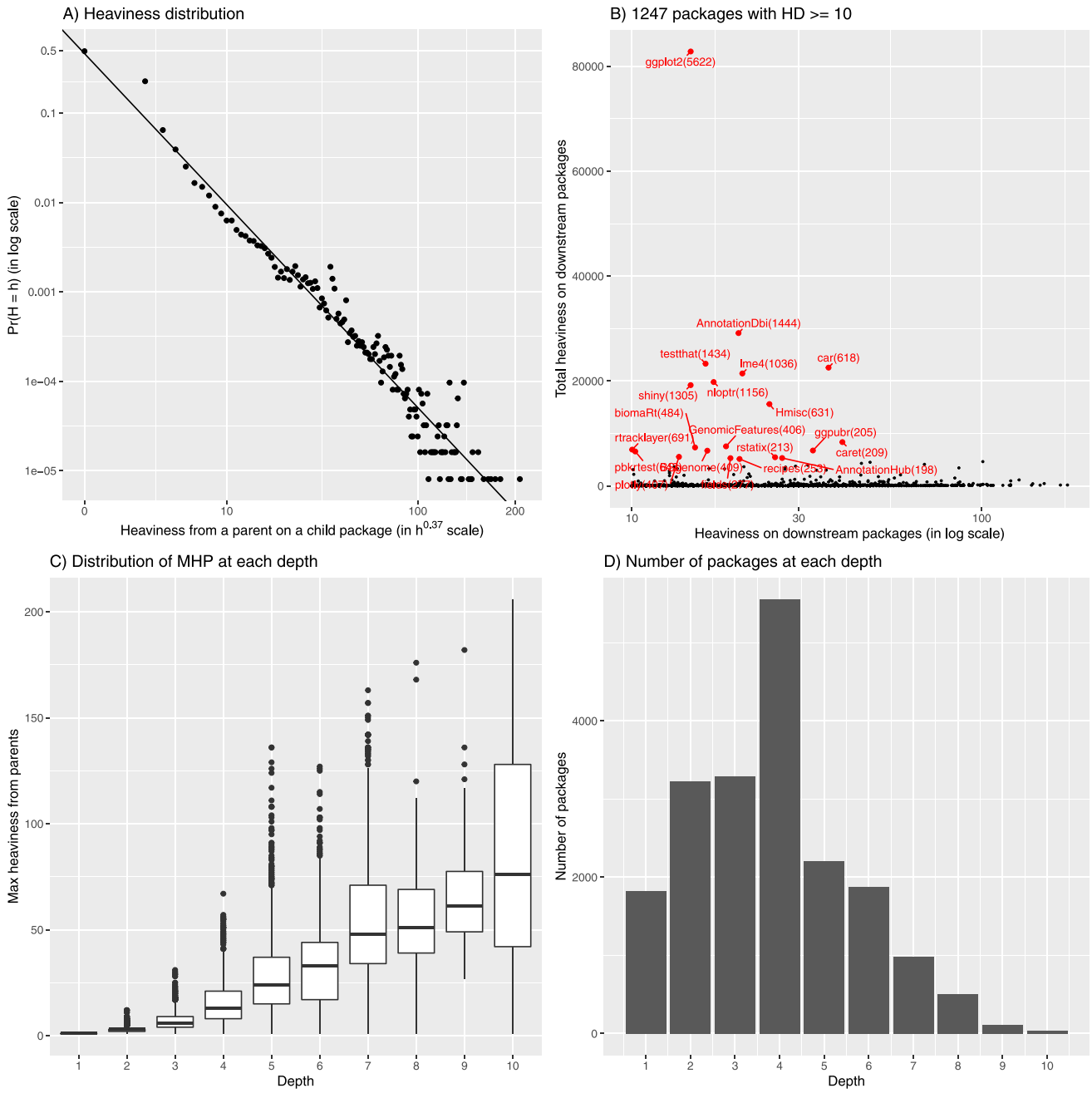


Fig. 11. Heaviness analysis on the global dependency graph. (A) Distribution of heaviness from a parent on a child package. The line is the fit from a stretched exponential distribution ($R^2 = 0.952$). (B) Total heaviness on downstream packages for the packages with HD $\geq$ 10. Packages in red have total heaviness $\geq$5000. Values in the parentheses are the numbers of downstream packages. (C) MHP distribution of packages at each depth in the dependency graph. (D) Number of packages at each depth. In Figure C and D, packages with depth = 0 (709 packages, i.e., packages with no dependencies) and depth $\geq$11 (1 package) are removed for the plot. HD: heaviness on downstream packages; MHP: max heaviness from parents.

component leaded by *ggplot2* includes 2,082 packages (48.4% of all packages in the core graph). The size of the graph component approximately follows a power-law distribution (Fig. 12B).

We next studied how deep a package P can transmit heavy dependencies to downstream of the core graph. For this, we defined a metric named "dependency transmission length". Assume P is reachable to N_{leaf} leaf packages[18] where a leaf package has out-degree of zero in the graph, let d_i be the distance from P to the ith leaf package which is the length of the shortest path from P to the leaf package, then the dependency transmission length denoted as l for P is calculated as

$$l = \max_{i \in \{1..N_{\text{leaf}}\}} d_i. \tag{20}$$

Fig. 12C illustrates the distribution of dependency transmission length. It shows the majority of the transmission from P only has a length of 1, which can be easily confirmed by Fig. 12A. But there also exist long paths where heavy dependencies can be continually transmitted to P's deep downstream packages (examples in Fig. 12D). Note the depth analysis here is different from that in Section 7.5.3. In Section 7.5.3, we studied how the dependency heaviness P inherits from its upstream, while in this section we studied how the dependency heaviness is transmitted to P's downstream.

[18] Package A is reachable to package B in the dependency graph when the distance from A to B is finite.

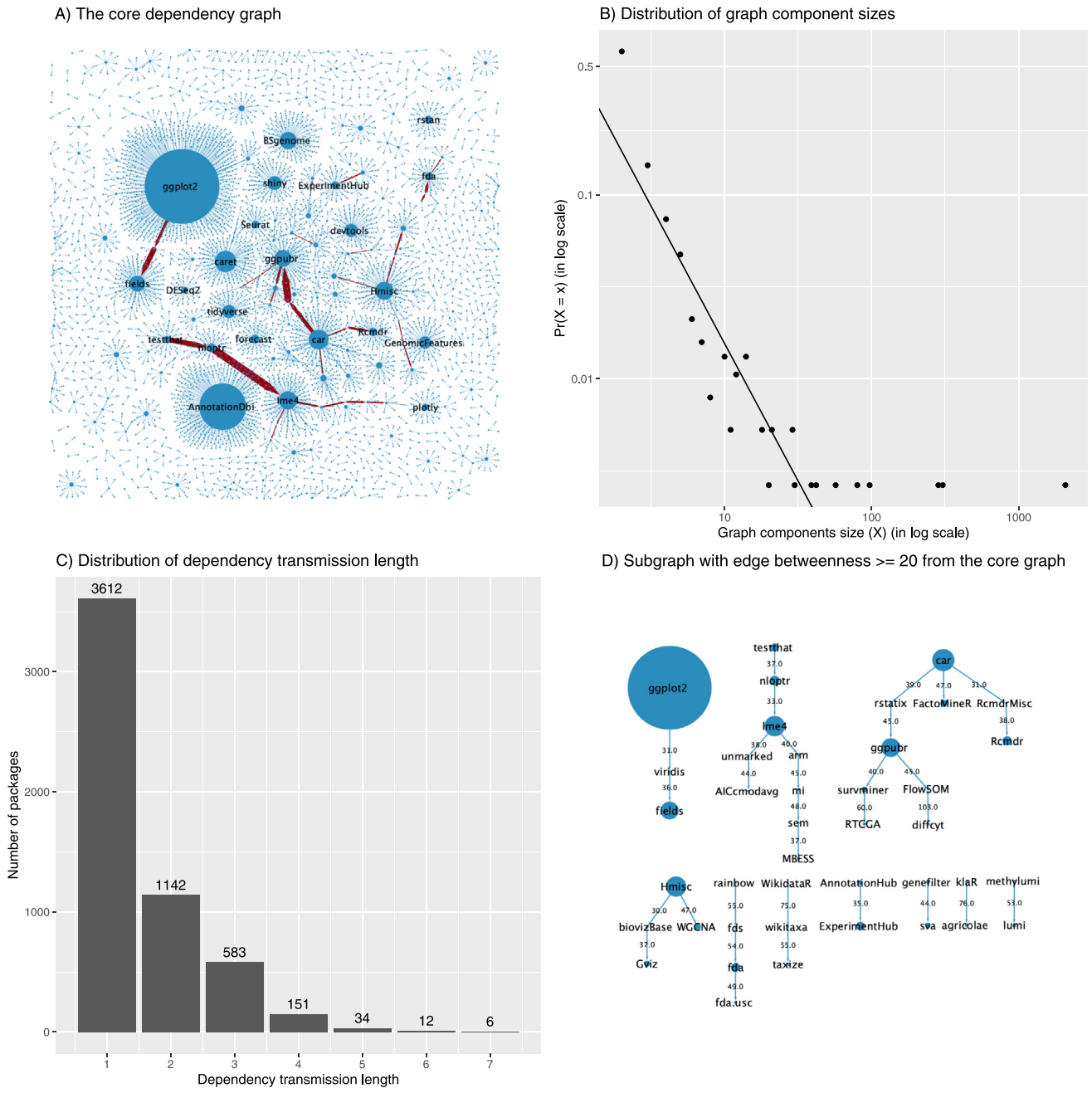


Fig. 12. Analysis on the core dependency graph. (A) Network visualization of the core graph. Node size is mapped to out-degree, i.e., number of child packages. Labels of hub packages with out-degree ≥30 are added. Edge width is mapped to the betweenness and edges with betweenness ≥20 are highlighted in red. (B) Distribution of graph component sizes. The line is the fit of the power-law distribution ($R^2 = 0.845$). The largest five data points are removed from fitting. (C) Distribution of dependency transmission length of packages in the core graph. (D) The subgraph that only contains high betweenness edges from Figure A. Node size is mapped to the out-degree calculated from the core graph. Values on edges are the heaviness from parents to corresponding child packages.

7.5.5. Key dependency paths

The betweenness of an edge measures the number of shortest paths in the graph that pass through the edge. In the context of the dependency graph, the edge betweenness measures the amount of heaviness flows that are transmitted via a parent–child pair. In Fig. 12A, edges with betweenness ≥20 are highlighted in red and a subgraph that only contains these high betweenness edges is induced in Fig. 12D. We named the subgraph as "key dependency paths". The key paths transmit 25.2% of the total heaviness flows (measured as the fraction of total betweenness in the key paths and in the core graph) while only including 0.8% edges and 1.0% packages from the core graph. In most cases, in the key paths, the heaviness is transmitted from hub packages, however there are exceptions. For example, *rstatix* only has a few child packages (10 in the global graph and 4 in the core graph), but it connects two hub packages *car* and *ggpubr* as a bridge to continue the dependency transmission (Fig. 12D). Another similar example is *viridis* that connects *ggplot2* and *fields* (Fig. 12D). Besides that, we also found there are long paths such as from *testthat* to *MBESS* with length of 6, and from *car* to *diffcyt* or *RTCGA* with length of 5, where the heavy dependencies can be continually transmitted through.

Answer to RQ5

We constructed a core graph which transmits heavy dependencies in the ecosystem. We revealed graph components and key paths that transmit major heaviness in the ecosystem. We found that hub packages mainly transmit dependency heaviness only to their child packages, thus locally. When a package locates more deep in the downstream of the dependency graph, it preferably inherits larger dependency heaviness from the ecosystem.

8. The open database

We have integrated the dependency heaviness analysis for the CRAN/Bioconductor ecosystem in the package *pkgndep* as a web-based database (We call it the heaviness database, Fig. 13). It provides detailed analysis reports on the dependency heaviness both from direct parent–child relations and remote upstream-downstream relations. The heaviness database can be simply accessed with the function `dependency_database()` from the *pkgndep* package.

The database has two parts: a summary table of all packages and analyses of individual packages. Fig. 13A illustrates the global table of the heaviness analysis of all packages. Columns in the table are separated into two groups highlighted in yellow and blue, which correspond to metrics from upstream (e.g., MHP) and on downstream (e.g., HC, HD and HID). Packages with adjusted HC $\geq$ 30 are highlighted in red to emphasize that they have high impacts on the ecosystem. If a package only imports a limited number of functions from the heaviest parent, the package is marked as 'reducible' in purple, which implies possibility to reduce its upstream dependencies for developers (see our suggestions in Section 9.1).

The database contains comprehensive tools for querying dependencies for individual packages. For a package *P*, the database allows to explore how the dependencies are inherited from parent or upstream to *P*, and how the dependencies are transmitted from *P* to its child or downstream. In the direct parent and child dependency results, there are tables showing heaviness-related metrics. It also lists the "Imports" information (i.e., how classes and methods are imported from parents to *P* or from *P* to its children) which is automatically parsed from the NAMESPACE files of corresponding packages. In the upstream dependency results, the dependency path from each upstream package to *P* is listed, which is the shortest path from the upstream package to *P* in the global dependency graph (Fig. 13B). There is also an interactive graph that shows how the heaviness is accumulated (Fig. 13B). Similarly, in the downstream dependency results, the dependency path from *P* to every downstream package is listed, which can be further filtered by the depth to the downstream packages (Fig. 13C). The downstream dependency graph is also visualized as an interactive graph (Fig. 13D). Normally, the downstream dependency graph is large. In order to reduce the graph size for visualization, leaf packages are grouped into a single node if they have the same parent (Fig. 13D). Additionally, edges with high betweenness in the downstream dependency graph are highlighted in red, which correspond to the key paths that transmit major dependency heaviness from *P*.

9. Considerations for developers

The heaviness analysis is especially useful for developers. As this study was motivated and accumulated from the author's experience as an active developer of R packages,[19] we proposed the following considerations from three different aspects from a developer's perspective. Note the three aspects are associated while not isolated. In the following subsections, we discussed each aspect with several examples to demonstrate how heaviness analysis benefits developers. The dependency heaviness analysis for all example packages mentioned in this section can be accessed in the heaviness database in *pkgndep*.

9.1. How to properly handle the dependency of a package?

It is a good practice to keep package dependency as simple as possible. However, there is always a balance between the compactness of dependencies and the comprehensiveness of a package's functionalities. If a package has a parent showing high heaviness (e.g., with high MHP), it is a sign that reduction of the dependency complexity should be considered. We have the following three suggestions that developers may consider.

First, if package *P* only imports one or a small amount of simple functions from its parent *A*, heavy dependencies from *A* can be avoided by directly implementing functions with the same functionalities as the original ones. For example, as we have demonstrated in our previous study (Gu and Hübschmann, 2022), the package *mapstats* has a heavy parent *Hmisc* with heaviness of 49 where only a single function `capitalize()` is imported to *mapstats*. `capitalize()` is an extremely simple function that only capitalizes the first letter of a word. It can be easily re-implemented by developer's own to get rid of the 49 unnecessary dependencies.

Second, on CRAN/Bioconductor, it is common that there are several packages providing the same functionalities for an analysis task. Then if *P* depends on a heavy parent, the developer can look for a light dependency package which provides the same functionality as the heavy one. For example, the package *biovizBase* has a heavy parent *Hmisc* with heaviness of 30 where a single function `bezier()` is imported to *biovizBase*. `bezier()` is for generating Bézier curves and the use of *Hmisc* can be replaced with other lighter packages that also generate Bézier curves, e.g., a package called *bezier* but with zero additional dependency. As a note, this optimization on *biovizBase* is even more meaningful because the reduction of the 30 extra dependencies on *biovizBase* can additionally reduce the dependencies of its 631 downstream packages with an average reduced dependencies of 20 for each.

Third, Some packages aim to be a "toolkit" to provide comprehensive analysis by integrating many other packages. This increasing comprehensiveness also brings the expansion of dependencies. For example, the package *singleCellTK* provides comprehensive tools for analyzing single-cell RNASeq data by depending on 82 parents and in total 369 strong dependencies.[20] This makes it very vulnerable to failures from upstream packages and it is not friendly for users to install. Nevertheless, for such toolkit packages, there are always core functionalities that are more frequently used by users and optional functionalities that are less used. The huge dependencies of such packages can be reduced by moving parents that only provide optional functionalities to weak parents, then it dramatically reduces the total strong dependencies (see methods in Section 3.2). Take the package *cola* which is also developed by the author as an example. *cola* provides consensus clustering analysis as its core functionality, and it also provides comprehensive functions for downstream analysis. According to our experience, we found some downstream analyses that depend on heavy parents are rarely used by users, thus, we set them as weak parents. With this

[19] https://jokergoo.github.io/software/.

[20] *singleCellTK* is the package with the largest number of parents and the second largest number of strong dependencies on CRAN/Bioconductor.

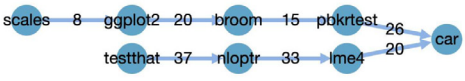
A

Package	Repository	Number of strong dependency packages	Number of all dependency packages	Number of parent packages	Max heaviness from parent packages	Max co-heaviness from parent packages	Heaviness on child packages	Number of child packages	Heaviness on downstream packages	Number of downstream packages
Rcmdr	CRAN	135	157	14	38	32	101.2	45	100.8	46
Seurat	CRAN	145	266	50	reducible 8	11	85.3	38	86.1	40
lumi	Bioconductor	162	170	19	53	47	114.2	13	83.8	18
RTCGA	Bioconductor	127	180	15	60	5	128.0	9	128.0	9
minfi	Bioconductor	141	165	34	24	16	62.4	38	68.7	60
tidyverse	CRAN	107	120	29	reducible 17	5	48.4	89	48.0	94
survminer	CRAN	115	139	18	40	4	58.2	27	58.6	36
taxize	CRAN	127	145	24	55	8	77.4	12	76.7	13
devtools	CRAN	76	119	23	12	5	47.0	80	45.3	95
GenomicScores	Bioconductor	98	154	17	26	40	56.0	26	56.3	27
WGCNA	CRAN	108	113	17	reducible 47	28	52.3	33	50.7	35
caret	CRAN	81	200	16	reducible 27	15	41.0	180	40.0	209
car	CRAN	87	125	14	26	39	40.6	183	36.5	618
brms	CRAN	123	187	24	reducible 37	12	65.1	13	65.1	13
AER	CRAN	92	170	7	reducible 78	4	52.6	22	54.3	22
ecospat	CRAN	232	255	24	reducible 117	49	151.0	3	151.0	3

B

Upstream package	Dependency path	Dependency depth	Heaviness on car
pbkrtest	pbkrtest → car (26)	2	26
lme4	lme4 → car (20)	2	20
nloptr	nloptr → lme4 → car (33, 20)	3	15
testthat	testthat → nloptr → lme4 → car (37, 33, 20)	4	14
scales	scales → ggplot2 → broom → pbkrtest → car (8, 20, 15, 26)	5	7

Dependency paths in the table are merged into the following graph with 8 nodes and 16 edges. Reset graph | Vertical layout



C

Adjust range of the dependency depth: 1 5

Depth = 1: 166 paths
Depth = 2: 221 paths
Depth = 3: 144 paths
Depth = 4: 37 paths
Depth = 5: 12 paths

Downstream package	Dependency path	Dependency depth	Heaviness of car on downstream package
DJL	car → DJL (88)	2	88
DistatisR	car → DistatisR (88)	2	88
GEWIST	car → GEWIST (88)	2	88

D

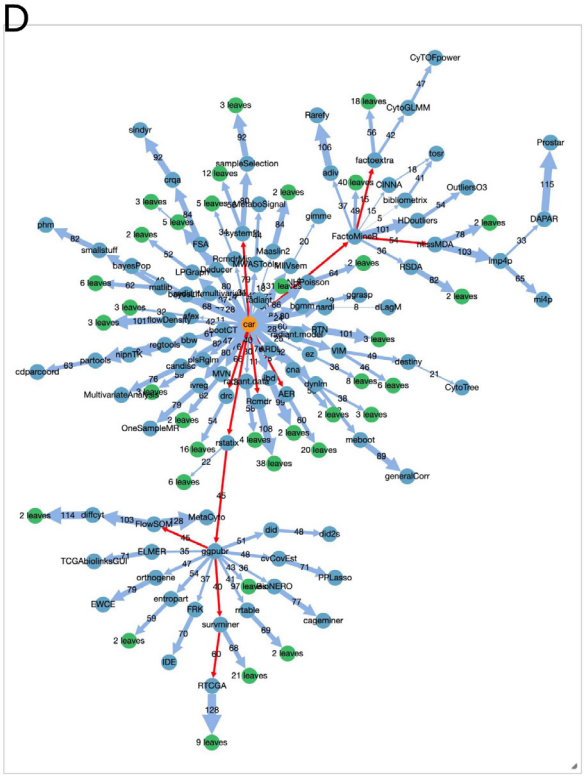


Fig. 13. Web-based database of the dependency heaviness analysis for all R packages on CRAN/Bioconductor. (A) The global heaviness table of all packages. (B) Upstream of a package. Dependency path from every upstream package is listed in a table and visualized in a graph. (C) Downstream of a package. Dependency path to every downstream package is listed in a table. The table can be filtered by selecting a range of depths of the dependency paths. (D) A graph of the downstream dependency graph. Green nodes represent groups of leaf packages that connect to the same parent package. Edges with high betweenness are highlighted in red. The cutoff of betweenness is selected as the 'elbow' of the curve of sorted betweenness of all edges in the graph. Values on edges are the heaviness from parents to corresponding child packages.

strategy, the strong dependency of *cola* was dramatically reduced from 252 to 64.

Nevertheless, there are also packages with no parent contributing significantly high dependency heaviness. In this case, optimizing the dependencies is difficult. An example is the package *Seurat* which has 50 parents and 145 strong dependencies, but the heaviest parent only contributes 8 additional dependencies (MHP = 8), thus optimization on one or only a few parents will not dramatically reduce dependencies of *Seurat*. Besides that, there are also scenarios where reduction of heavy parents could not be performed: (i) A heavy parent provides core functionality to its child package; (ii) S4 methods or S4 classes[21] are imported from a parent package; (iii) A child package depends on the C/C++ headers from a parent.[22]

21 S4 is an object oriented system in R.

22 Then the parent must be put in the "LinkingTo" field of the child package.

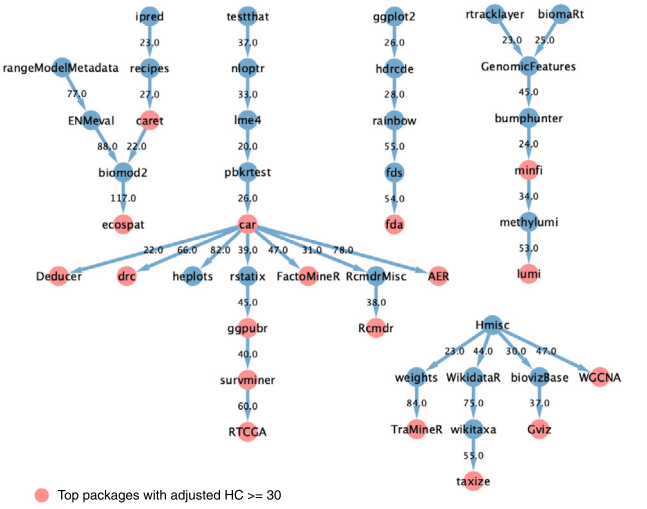


Fig. 14. Upstream of packages with adjusted HC $\geq$ 30. For simplicity, only edges with heaviness $\geq$20 and graph components with size $\geq$5 are kept in the figure. Values on edges are the heaviness values from parents to corresponding child packages. HC: heaviness on child packages.

9.2. How is dependency heaviness accumulated to a package from upstream?

Dependencies from remote upstream cannot be directly controlled by the developer, but it is still useful for understanding how the heaviness is accumulated to his package. Here we take top packages with the highest adjusted HC (Section 7.3) as examples, because they also inherit huge dependency heaviness from their upstream and they play important roles in the ecosystem. Fig. 14 illustrates upstream packages that transmit major dependency heaviness to top HC packages. In general, dependencies are transmitted in the long ranges, which agrees with the results in Section 7.1. We observed there are three different modes of dependency accumulation.

1. **Heaviness is accumulated from multiple heavy parents.** For example, package *ecospat* inherits unique dependencies from *ENMeval* (with a heaviness of 74 uniquely transmitted to *ecospat*) and *caret* (with a heaviness of 19 uniquely transmitted to *ecospat*) separately.
2. **A group of packages inherit heavy dependencies all from the same heavy upstream package.** For example, package *TraMineR*, *taxize*, *Gviz* and *WGCNA* all inherit huge unique dependencies from the same upstream package *Hmisc*.
3. **The heaviness transmission can be traced back to a remote upstream package.** As a typical example, dependency transmission to package *RTCGA* can be traced back to the package *car* which additionally affects several other packages. If tracing further to *car*'s upstream, we can see the heavy dependencies are actually from the package *testthat*.

Once we have revealed the upstream source of the heaviness, this leads us to the question which is asked in the next section: how does the dependency heaviness spread in the ecosystem from the "source"? In particular, we will explore the impacts of *testthat* and *Hmisc* on their downstream packages.

9.3. How to reduce the dependency heaviness spreading to the downstream?

When a package becomes popular in the ecosystem, it is very important for the developer to carefully manage its dependency size. If the developer introduces new features that need extra dependencies, he should be aware of the risks that are also brought to the ecosystem. We suggest the developer also paying attention to the current set of parents of his package, and trying to compact its dependency size.

According to our observation, there are still a large number of packages on CRAN/Bioconductor whose dependency sizes can be reduced. Here we take the package *AnnotationDbi* as an example. *AnnotationDbi* is a hub package that contributes dependency heaviness to 1444 downstream packages in the ecosystem (with HD of 20.2, mainly on Bioconductor). It was also mentioned as a high impact package in Sections 7.2.2 and 7.5.4. By exploring how *AnnotationDbi*'s parents contribute their functionalities to *AnnotationDbi*, we found a package *KEGGREST* contributes 20 unique dependencies to *AnnotationDbi* and these extra dependencies are transmitted to all downstream packages of *AnnotationDbi*. A close inspection shows only one function `KeggList()` is imported to *AnniotaitonDbi* and this functionality is very rarely used by its massive downstream packages. If *KEGGREST* can be set as a weak parent of *AnnotationDbi*, on average 9.5 dependencies can be reduced for every of its 1,444 downstream packages. This actually implies even a small reduction of dependencies of a hub package will have a huge impact on the whole ecosystem.

It is also interesting to explore how dependencies from hub packages are transmitted to downstream packages in the ecosystem. This helps to reveal the "problematic sites" of the dependency transmission and later developers can propose possible solutions to fix them. We found the following two typical scenarios where heavy dependencies are improperly transmitted to the ecosystem and they can actually be avoided.

Inefficient use of a parent package. In Section 9.2, we revealed a long dependency heaviness transmission originated from *testthat*. A closer look shows *testthat* behaves as a hub package in the ecosystem which has 130 child packages and 1,304 indirect downstream packages, with HC of 18.2 and HID of 15.8 respectively.[23] This is quite unexpected because *testthat* is a package mainly for package unit testing purpose and it is normally put in a package's "Suggests" field.[24] We found there are two reasons for its high impact in the ecosystem. First, majority of the heaviness from *testthat* are transmitted via its child package *nloptr* to downstream. In total 18,933 extra unique dependencies (81.3% of all from *testthat*) are transmitted to its 1,156 (80.6%) downstream packages via *nloptr*. *nloptr* declares *testthat* as its strong parent because *nloptr* performs unit testing on its C++ code which requires a header file from *testthat*, thus *testthat* must be put in its "LinkingTo" field. If by some means, *nloptr* can get rid of the strong dependency on *testthat*, e.g., by using another unit testing tool on C++ code, on average 16.4 dependency heaviness for each of its 1,156 downstream packages can be reduced.

Secondly, we also observed *testthat* is used inefficiently in its 129 other child packages (excluding *nloptr*). In these packages, developers use *testthat* to perform object validation in the source code, e.g., to compare whether two objects are equal (by `expect_equal()`) or to validate a text output (e.g., by `expect_match()`). *testthat* provides comprehensive tools for unit testing on packages. while it would be too heavy if it is directly used in package's source code just for simple validation. Actually, developers can replace `expect_*()` functions from *testthat* with self-implemented code very easily. For example `expect_equal(x, y)` can be replaced by `x == y` (assuming `x` and `y` are two scalars), and `expect_match(text, regexp)` can

[23] The downstream dependency graph of *testthat* can be accessed in the heaviness database in *pkgndep*.

[24] https://r-pkgs.org/testing-basics.html.

be replaced by `grepl(regexp, text)`. In this way, large amount of extra dependencies of *testthat* can be reduced.

A hub package providing a wide range of functionalities. If a hub package provides a wide range of functionalities, it is very likely that it also inherits a large number of packages from upstream (We have partially discussed it in Section 9.1). All the dependencies of the hub package are transmitted to the downstream even though child packages may only import a limited number of functions from it. Here we take the package *devtools* and *Hmisc* as two examples. *devtools* provides functionalities for package development, but in the ecosystem it has 80 child packages with HC of 47 and 15 indirect downstream with HID of 36.3. A deep inspection shows most of its child packages import the function `install_github()` to install dependencies that are directly from their development branches on GitHub. There are two possible optimizations. First, dependency on *devtools* can be set as a weak parent because *devtools* does not contribute to the functionalities of its child packages. And second, the installation functionality in *devtools* can be separated into a new and light package. Actually, `install_github()` has already been moved to a new package *remotes*,[25] and developers can consider to migrate from *devtools* to *remotes*.

As a second example, also as we have mentioned in Sections 7.2.2 and 9.2, *Hmisc* behaves as a heavy hub packages with 248 child packages (with HC of 29.4) and 383 indirect downstream packages (with HID of 20). *Hmisc*, as its name tells, provides a huge collection of functions for *"data analysis, high-level graphics, utility operations, functions for computing sample size and power, simulation, importing and annotating datasets, imputing missing values, advanced table making, variable clustering, character string manipulation, conversion of R objects to LaTeX and HTML code, and recording variables"*.[26] It has 18 parents and in total 67 strong dependencies from upstream. The dependency heaviness analysis on *Hmisc* reveals *ggplot2*, *viridis* and *htmlTable* contribute majority (59.7%) of dependencies to *Hmisc*.[27] The first two are for data visualization and the last one is for report generation. A deep inspection of how *Hmisc* is used in its child packages shows the visualization and reporting are very rarely used. Thus, similar as *devtools*, if *Hmisc* can separate its visualization and reporting parts out as a separated package, it can save on average 18.5 extra dependencies for every of its 631 downstream packages.

10. Discussion

Dependency analysis is an important topic for studying package ecosystems. One of the aims is to discover top packages that have major impacts on the dependency transmission in the ecosystem. Number of dependents is a widely-used metric that measures the local impact of how important a package is in contributing to other packages' functionalities (Mora-Cantallops et al., 2020b; Korkmaz et al., 2019). By taking the ecosystem as a whole, researchers studied the vulnerability of packages to the failures caused by recursive dependencies from upstream. Then, a more useful metric, the number of transitive dependencies, is proposed Mora-Cantallops et al. (2020b) and Decan et al. (2019). Besides the transitive effects accumulated from upstream, researchers also looked at the number of transitive dependents to study the indirect influences on downstream of the ecosystem (Decan et al., 2019; Abate et al., 2009). Although these metrics are useful for understanding the attributes of the ecosystem, they are used as descriptive statistics in current studies (Decan et al., 2019). These analyses are usually global and they are limited for developers because they provide almost no practical help on how to manage dependencies of their packages. After high impact packages are discovered from the ecosystem, naturally there will be the follow-up questions to ask, such as how the dependencies are transmitted from upstream to the high impact packages or how the dependencies are transmitted from the high impact packages to downstream? This may bring more questions such as can we find the most important part of the dependency transmission in the ecosystem? As a package may have multiple dependencies, either direct or transitive, it is quite common that individual dependencies have different levels of influences on the package. This implies, to understand the ecosystem deeper, we need to shift the focus from package-centered to dependency relation-centered, i.e., to find which dependency relation is more important with regard to transmitting dependencies. For this purpose, we proposed a new metric named "dependency heaviness" which quantitatively measures the unique contribution of dependency from a parent to a child. Dependency heaviness is also based on transitive dependencies, but it measures from a different aspect. With this metric, we can easily identify which parents are heavy with regards to how they contribute dependencies to a child package.

Based on the direct dependency heaviness from a parent to a child, we extended the heaviness definitions to a broader range to study the patterns of unique dependency flows in the ecosystem. We aimed to answer the question of how the dependencies are uniquely transmitted through the ecosystem. We first explored the heavy dependency inheritance from parent and upstream packages. This analysis might be less interesting for software engineering because top packages with this metric are basically special cases. They are preferably located at the end of the dependency transmission chain and with no dependents, thus having no large impact on the ecosystem. But as they inherit heavy dependencies from upstream, it is still interesting to explore how the dependencies are accumulated. We found the deeper a package is located in the ecosystem, the more likely it inherits heavier dependencies.

We next explored how packages transmit unique dependencies to the downstream of the ecosystem with two metrics of HC and HID. This analysis is more important because top packages with the highest HC or HID have major impacts on the dependency transmission in the ecosystem. HC is a more practical metric because it also measures the expected number of additional dependencies if a package is included as a new parent of a developer's package. HC, although it is also summarized from all its child packages, can generate different results from current studies. For example, the package *Rcpp* has 2,795 child packages and it is the package with the third most child packages in the ecosystem. As a hub package, code breaks of *Rcpp* will affect a large number of other packages. In this sense, *Rcpp* can be treated as the source of the "risk" in the ecosystem. However, from the aspect of dependency heaviness, it only has a HC of 0.58, which means it is an extremely light package. HC and HID focus more on the role of a package as an intermediate package receiving dependencies from upstream and transmitting to downstream. So they are more like bi-directional metrics of the dependency transmission. Additionally, HC and HID focus more on the influences on individual packages while not on the whole set of dependents.

We applied network analysis on the dependency graph. Being different from network analysis in current studies which take the graph as unweighted (Mora-Cantallops et al., 2020b), we studied the graph by weighting edges with dependency heaviness. In this way, we can distinguish which relations are more important in the graph. Based on the edge weights, we extracted a core graph and key paths that transmit major dependencies in the

[25] The first version of *remotes* was released in 2016.

[26] https://CRAN.R-project.org/package=Hmisc.

[27] Heaviness analysis of *Hmisc* can be performed in the heaviness database in *pkgndep*.

ecosystem. A deeper analysis on the dependency graph revealed heavy dependencies can be accumulated from very far upstream, and hub packages transmit heavy dependencies most locally.

We have implemented the complete analysis as a web-based database that can be easily accessed from the *pkgndep* package. The database provides detailed statistics for various heaviness metrics both from upstream and downstream packages. It also provides comprehensive dependency analysis for individual packages. It helps developers understand how the dependencies are accumulated to their packages from the ecosystem and how the dependencies spread to the downstream of their packages. We have extensively used it to study the dependency chains of example packages in Section 9.

11. Limitations and future work

In Section 7.1, we studied patterns of maximal heaviness from parents (MHP) in the ecosystem. As we have demonstrated, if a package suffers heavy dependencies from its parents, in many cases, the heaviest parent dominantly contributes unique dependencies to it. Nevertheless, there are still cases where there are more than one dominantly heavy parents. For example, the package *pathwayTMB*[28] has two heaviest parents of *clusterProfiler* and *survminer* which contribute dependency heaviness of 62 and 55 mutually exclusively. One reasonable hypothesis is that if package P has more than one heavy parent, these heavy parents are responsible for different analysis tasks in P. Indeed, for the two heavy parents of *pathwayTMB*, *clusterProfiler* performs gene set enrichment analysis and *survminer* performs survival analysis, thus they introduce dependencies from different sources. Nevertheless, such scenarios where P has multiple heavy parents should not be often, but it provides a complementary view on the ecosystem.

In Section 4.3, we defined the co-heaviness metric from two parents. We can define co-heaviness from more than two parents in a similar way. However, as co-heaviness only measures the unique number of dependencies that a group of parents simultaneously bring in, with considering more parents, the co-heaviness value will decrease and it would not be a proper metric for studying the ecosystem.

Our study was only focused on the CRAN and Bioconductor ecosystems. There are also a great number of R packages only hosted on GitHub. Since CRAN and Bioconductor packages are not allowed to depend on GitHub packages, it would be interesting to study how the dependencies heaviness is transmitted to GitHub packages. Since GitHub packages are more for experimental purposes, one hypothesis is that GitHub packages may suffer more from heavy parents.

Evolution of the package ecosystem is also a popular topic in software engineering (German et al., 2013; Kikas et al., 2017; Mora-Cantallops et al., 2020a) which studies the longitudinal change of packages as well as their dependency relations over time. In particular, we think the following two topics might be worth exploring. 1. We can study the dependency changes after a high impact package was introduced to the ecosystem. For example, *tidyverse* was introduced to CRAN in 2016. Since then, it has become a core package for data analysis. *tidyverse* is a heavy package with HC of 48.4 and it would be interesting to study how it changes the dependency structure of the ecosystem. 2. Specific for Bioconductor, it would be interesting to study the dependency structure change along with the evolution of high-throughput technologies, e.g., in the era of microarray, genomics, single cell transcriptomics, and multi-omics.[29] This might help to answer the question: does more advanced technology make the corresponding tools more complex? Finally, we hope the study presented in this paper as well as the *pkgndep* package can give developers new insights to properly maintain and optimize dependencies of their packages, then to build a healthier and more robust R ecosystem in the future.

In Section 9, we manually explored that heavy dependencies of a group of packages can be traced back to the upstream package *nloptr* which inherits large dependencies from *testthat*. This analysis is important because it helps to find out the "source" of the heavy dependency transmission. Then the question is how to reveal such "source packages" systematically and automatically? To answer this question, we can look at a parent package A and its child P from two aspects. First, the dependency transmission from A to P is influential in the ecosystem where A contributes a huge amount of dependency heaviness to the downstream via P. Let us denote the total amount of dependency heaviness from A to its downstream via P as $h_{\mathrm{d,total}}^{A\to(P)}$. Using the same denotations in Eq. (6), it can be calculated as

$$h_{\mathrm{d,total}}^{A\to(P)} = \sum_{k=1}^{K_{\mathrm{d}}} (n_{1k} - n_{2k}) \cdot I(A_k \in S_{\mathrm{d}}^{P}) \tag{21}$$

where A_k is A's kth downstream package and S_{d}^{P} is the set of P's downstream packages. Second, heavy dependencies transmitted to A's downstream via P are not originated from A's parents. This means A is the source of the heavy dependency chain while very few dependencies are accumulated from A's parents. Let us denote A's MHP parent as B, then we can quantitatively measure the level of $A \to P$ being a source of the heaviness transmission denoted as $s^{A\to(P)}$ by

$$s^{A\to(P)} = h_{\mathrm{d,total}}^{A\to(P)} - h_{\mathrm{d,total}}^{B\to(P)}. \tag{22}$$

In this way, if A is the source of heaviness transmission via P, $h_{\mathrm{d,total}}^{B\to(P)}$ will be small which makes $s^{A\to(P)}$ being a large value; while if most of the heaviness is still from A's parent B, $h_{\mathrm{d,total}}^{A\to(P)}$ and $h_{\mathrm{d,total}}^{B\to(P)}$ would be similar which makes $s^{A\to(P)}$ being a small value.

The dependency heaviness analysis can be extended to ecosystems in other programming languages. Theoretically, definitions of various heaviness metrics have no assumption of which package ecosystem to use, and they can be universally applied as long as a global dependency graph is available. Similar as *pkgndep*, we plan to implement a general purpose tool that supports many other package ecosystems also with a web-based analysis platform for them.

12. Conclusion

We performed a systematic analysis on the dependency heaviness landscape of the R package ecosystem. We revealed the general patterns of the dependency transmission locally from parent to child packages, also remotely from upstream to downstream packages. Using network analysis approaches, we revealed top packages and key paths that play significant roles in transmitting dependencies in the ecosystem. The complete analysis has been implemented as a web-based database and we believe it will facilitate researchers as well as R package developers to better understand the R package ecosystem and to build more robust software.

CRediT authorship contribution statement

Zuguang Gu: Conceptualization, Formal analysis, Investigation, Methodology, Software, Visualization, Writing – original draft, Writing – review & editing.

[28] *pathwayTMB* inherits 215 strong dependencies from its 16 parent packages.

[29] An example of the timeline can be found at https://carpentries-incubator.github.io/bioc-project/02-introduction-to-bioconductor/index.html.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

All data as well as the web-based dependency database are available in the *pkgndep* R package (https://CRAN.R-project.org/package=pkgndep). How to access the data is described in Section 5. The use of the heaviness database is described in Section 8.

Acknowledgments

This study did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

References

Abate, P., Di Cosmo, R., Boender, J., Zacchiroli, S., 2009. Strong dependencies between software components. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement. ESEM '09, IEEE Computer Society, USA, pp. 89–99. http://dx.doi.org/10.1109/ESEM.2009.5316017.

Bogart, C., Kästner, C., Herbsleb, J., Thung, F., 2016. How to break an API: Cost negotiation and community values in three software ecosystems. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2016, Association for Computing Machinery, New York, NY, USA, pp. 109–120. http://dx.doi.org/10.1145/2950290.2950325.

Bommarito, M.J., Bommarito, E., 2021. An empirical analysis of the R package ecosystem. http://dx.doi.org/10.2139/ssrn.3788978.

Burns, B., Lamb, J., Qi, J., 2021. Pkgnet: get network representation of an R package. URL: https://CRAN.R-project.org/package=pkgnet.

Chang, W., Cheng, J., Allaire, J., Sievert, C., Schloerke, B., Xie, Y., Allen, J., McPherson, J., Dipert, A., Borges, B., 2022. Shiny: Web application framework for R. URL: https://CRAN.R-project.org/package=shiny.

Claes, M., Mens, T., Grosjean, P., 2014. On the maintainability of CRAN packages. In: 2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). pp. 308–312. http://dx.doi.org/10.1109/CSMR-WCRE.2014.6747183.

Csardi, G., Nepusz, T., 2006. The igraph software package for complex network research. InterJ. Complex Syst. 1695, URL: https://igraph.org.

de Vries, A., 2022. Minicran: create a mini version of CRAN containing only selected packages. URL: https://CRAN.R-project.org/package=miniCRAN.

Decan, A., Mens, T., Claes, M., 2016a. On the topology of package dependency networks: A comparison of three programming language ecosystems. In: Proccedings of the 10th European Conference on Software Architecture Workshops. ECSAW '16, Association for Computing Machinery, New York, NY, USA, pp. 1–4. http://dx.doi.org/10.1145/2993412.3003382.

Decan, A., Mens, T., Claes, M., Grosjean, P., 2016b. When GitHub meets CRAN: An analysis of inter-repository package dependency problems. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering. SANER, 1, pp. 493–504. http://dx.doi.org/10.1109/SANER.2016.12.

Decan, A., Mens, T., Grosjean, P., 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empir. Softw. Eng. 24 (1), 381–416. http://dx.doi.org/10.1007/s10664-017-9589-y.

Elton, D.C., 2018. Stretched exponential relaxation. http://dx.doi.org/10.48550/ARXIV.1808.00881.

Fortuna, M.A., Bonachela, J.A., Levin, S.A., 2011. Evolution of a modular software network. Proc. Natl. Acad. Sci. 108 (50), 19985–19989. http://dx.doi.org/10.1073/pnas.1115960108.

German, D.M., Adams, B., Hassan, A.E., 2013. The evolution of the R software ecosystem. In: 2013 17th European Conference on Software Maintenance and Reengineering. pp. 243–252. http://dx.doi.org/10.1109/CSMR.2013.33.

Grolemund, G., Wickham, H., 2016. R for Data Science. O'Reilly Media, Inc., Sebastopol, CA.

Gu, Z., Hübschmann, D., 2022. Pkgndep: a tool for analyzing dependency heaviness of R packages. Bioinformatics 38 (17), 4248–4251. http://dx.doi.org/10.1093/bioinformatics/btac449.

Gustavsen, Julia, A., Pai, Shraddha, Isserlin, Ruth, Demchak, Barry, Pico, Alexander, R., 2019. RCy3: Network biology using cytoscape from within R. f1000research. http://dx.doi.org/10.12688/f1000research.20887.3.

Haefliger, S., von Krogh, G., Spaeth, S., 2008. Code reuse in open source software. Manage. Sci. 54 (1), 180–193. http://dx.doi.org/10.1287/mnsc.1070.0748.

Hornik, K., Ligges, U., Zeileis, A., 2021. Changes on CRAN 2021-07-01 to 2021-12-31. R J. 13 (2), URL: https://journal.r-project.org/archive/2021-2/.

Huber, W., Carey, V.J., Gentleman, R., Anders, S., Carlson, M., Carvalho, B.S., Bravo, H.C., Davis, S., Gatto, L., Girke, T., Gottardo, R., Hahne, F., Hansen, K.D., Irizarry, R.A., Lawrence, M., Love, M.I., MacDonald, J., Obenchain, V., Oleś, A.K., Pagès, H., Reyes, A., Shannon, P., Smyth, G.K., Tenenbaum, D., Waldron, L., Morgan, M., 2015. Orchestrating high-throughput genomic analysis with Bioconductor. Nature Methods 12 (2), 115–121. http://dx.doi.org/10.1038/nmeth.3252.

Jafari, A.J., Costa, D.E., Abdalkareem, R., Shihab, E., Tsantalis, N., 2021. Dependency smells in JavaScript projects. IEEE Trans. Softw. Eng. 1. http://dx.doi.org/10.1109/tse.2021.3106247.

Kikas, R., Gousios, G., Dumas, M., Pfahl, D., 2017. Structure and evolution of package dependency networks. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories. MSR, pp. 102–112. http://dx.doi.org/10.1109/MSR.2017.55.

Korkmaz, G., Kelling, C., Robbins, C., Keller, S., 2019. Modeling the impact of Python and R packages using dependency and contributor networks. Soc. Netw. Anal. Min. 10 (1), 7. http://dx.doi.org/10.1007/s13278-019-0619-1.

Mora-Cantallops, M., Sánchez-Alonso, S., García-Barriocanal, E., 2020b. A complex network analysis of the comprehensive R archive network (CRAN) package ecosystem. J. Syst. Softw. 170, 110744. http://dx.doi.org/10.1016/j.jss.2020.110744.

Mora-Cantallops, M., Sicilia, M.-A., Garcí a Barriocanal, E., Sánchez-Alonso, S., 2020a. Evolution and prospects of the comprehensive R archive network (CRAN) package ecosystem. J. Softw. Evol. Process. 32 (11), e2270. http://dx.doi.org/10.1002/smr.2270.

R Core Team, 2022. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria, URL: https://www.R-project.org/.

Rafacz, D., Baniecki, H., Maksymiuk, S., Bakala, L., 2021. Deepdep: visualise and explore the deep dependencies of R packages. URL: https://CRAN.R-project.org/package=deepdep.

Shannon, P., Markiel, A., Ozier, O., Baliga, N.S., Wang, J.T., Ramage, D., Amin, N., Schwikowski, B., Ideker, T., 2003. Cytoscape: A software environment for integrated models of biomolecular interaction networks. Genome Res. 13 (11), 2498–2504. http://dx.doi.org/10.1101/gr.1239303.

Srikanth, K., Nikhil, S., 2018. Pkggraph: A consistent and intuitive platform to explore the dependencies of packages on the comprehensive R archive network like repositories. URL: https://CRAN.R-project.org/package=pkggraph.

Venables, W.N., Ripley, B.D., 2002. Modern Applied Statistics with S. Springer New York, NY, New York.

Vidoni, M., 2021. Software engineering and R programming: A call for research. R J. 13 (2), 6–14. http://dx.doi.org/10.32614/RJ-2021-108.

Wang, X.F., Chen, G., 2003. Complex networks: small-world, scale-free and beyond. IEEE Circuits Syst. Mag. 3 (1), 6–20. http://dx.doi.org/10.1109/MCAS.2003.1228503.

Wilke, C.O., 2019. Fundamentals of Data Visualization. O'Reilly Media, Inc., Sebastopol, CA.

Xie, Y., 2015. Dynamic Documents with R and Knitr. Chapman and Hall/CRC, Boca Raton, Florida.