Modeling Package Dependencies Using Graphs

Introducing the pkgDepTools Package

by Seth Falcon

Introduction

Dealing with packages that have many dependencies, such as those in the Bioconductor repository, can be a frustrating experience for users. At present, there are no tools to list, recursively, all of a package's dependencies, nor is there a way to estimate the download size required to install a given package.

In this article we present the **pkgDepTools** package which provides tools for inspecting the dependency relationships among packages, generating the complete list of dependencies of a given package, and estimating the total download size required to install a package and its dependencies. The tools are built on top of the **graph** (Gentleman et al., 2006) package which is used to model the dependencies among packages in the BioC and CRAN repositories. Aside from this particular application to package dependencies, the approach taken is instructive for those interested in modeling and analyzing relationship data using graphs and the **graph** package.

Deciding how to represent the data in a graph structure and transforming the available data into a graph object are first steps of any graph-based analysis. We describe in detail the function used to generate a graph representing the dependency relationships among R packages as the general approach can be adapted for other types of data.

We also demonstrate some of the methods available in **graph**, **RBGL** (Carey and Long, 2006), and **Rgraphviz** (Gentry, 2006) to analyze and visualize graphs.

Modeling Package Dependencies

An R package can make use of functions defined in another R package by listing the package in the `Depends` field of its 'DESCRIPTION' file. The `available.packages` function returns a matrix of meta data for the packages in a specified list of CRAN-style R package repositories. Among the data returned are the dependencies of each package.

To illustrate our method, we use data from Bioconductor's package repositories as well as the CRAN repository. The Bioconductor project strongly encourages package contributors to use the dependency mechanism and build on top of code in other packages. The success of this code reuse policy can be measured by examining the dependencies of Bioconductor packages. As we will see, packages in the Bioconductor software repository have, on average, much richer dependency relationships than the packages hosted on CRAN.

A graph consists of a set of nodes and a set of edges representing relationships between pairs of nodes. The relationships among the nodes of a graph are binary; either there is an edge between a pair of nodes or there is not. To model package dependencies using a graph, let the set of packages be the nodes of the graph with directed edges originating from a given package to each of its dependencies. Figure 2 shows a part of the Bioconductor dependency graph corresponding to the **Category** package. Since circular dependencies are not allowed, the resulting dependency graph will be a directed acyclic graph (DAG).

The 'DESCRIPTION' file of an R package also contains a `Suggests` field which can be used by package authors to specify packages that provide optional features. The interpretation and use of the `Suggests` field varies, and the graph resulting from using this relationship in the Bioconductor repository is not a DAG; cycles are created by packages suggesting each other.

Building a Dependency Graph

To carry out the analysis, we need the **pkgDepTools** package along with its dependencies: **graph** and **RBGL**. We will also make use of **Biobase**, **Rgraphviz**, and **RCurl**. You can install these packages on your system using `biocLite` as shown below.

```
> u <- "http://bioconductor.org/biocLite.R"
> source(u)
> biocLite("pkgDepTools", dependencies=TRUE)
```

```
> library("pkgDepTools")
> library("RCurl")
> library("Biobase")
> library("Rgraphviz")
```

We now describe the `makeDepGraph` function that retrieves the meta data for all packages of a specified type (source, win.binary, or mac.binary) from each repository in a list of repository URLs and builds a `graph` instance representing the packages and their dependency relationships.

The function takes four arguments: 1) `repList` a character vector of CRAN-style package repository URLs; 2) `suggests.only` a logical value indicating whether the resulting graph should represent relations from the Depends field (FALSE, default) or the Suggests field (TRUE); 3) type a string indicating the type of packages to search for, the default is "source"; 4) keep.builtin which will keep packages that come with a standard R install in the dependency graph (the default is FALSE).

The definition of makeDepGraph is shown in Figure 1. The function obtains a matrix of package meta data from each repository using available.packages. A new graphNEL instance is created using new. A node attribute with name "size" is added to the graph with default value NA. When keep.builtin is FALSE (the default), a list of packages that come with a standard R install is retrieved and stored in baseOrRecPkgs.

Iterating through each package's meta data, the appropriate field (either Depends and Imports or Suggests) is parsed using a helper function cleanPkgField. If the user has not set keep.builtin to TRUE, the packages that come with R are removed from deps, the list of the current package's dependencies. Then for each package in deps, addNode is used to add it to the graph if it is not already present. addEdge is then used to create edges from the package to its dependencies. The size in megabytes of the packages in the current repository is retrieved using getDownloadSizesBatched and is then stored as node attributes using nodeData. Finally, the resulting graphNEL, depG is returned. A downside of this iterative approach to the construction of the graph is that the addNode and addEdge methods create a new copy of the entire graph each time they are called. This will be inefficient for very large graphs.

Definitions for the helper functions cleanPkgField, makePkgUrl, and getDownloadSizesBatched are in the source code of the **pkgDepTools** package.

Here we use makeDepGraph to build dependency graphs of the BioC and CRAN packages. Each dependency graph is a graphNEL instance. The out-edges of a given node list its direct dependencies (as shown for package **annotate**). The node attribute "size" gives the size of the package in megabytes.

```
> biocUrl <- biocReposList()["bioc"]
> cranUrl <- biocReposList()["cran"]
> biocDeps <- makeDepGraph(biocUrl)
> cranDeps <- makeDepGraph(cranUrl)
```

```
> biocDeps

A graphNEL graph with directed edges
Number of Nodes = 256
Number of Edges = 389
```

```
> cranDeps

A graphNEL graph with directed edges
Number of Nodes = 908
Number of Edges = 403
```

```
> edges(biocDeps)["annotate"]

$annotate
[1] "Biobase"
```

```
> nodeData(biocDeps, n = "annotate",
      attr = "size")

$annotate
[1] 1.451348
```

The degree and connectedComp methods can be used to compare the BioC and CRAN dependency graphs. Here we observe that the mean number of direct dependencies (out degree of nodes) is larger in BioC than it is in CRAN.

```
> mean(degree(biocDeps)$outDegree)

[1] 1.519531
```

```
> mean(degree(cranDeps)$outDegree)

[1] 0.4438326
```

A subgraph is connected if there is a path between every pair of nodes. The **RBGL** package's connectedComp method returns a list of the connected subgraphs. Examining the distribution of the sizes (number of nodes) of the connected components in the two dependency graphs, we can see that the BioC graph has relatively fewer length-one components and that more of the graph is a part of the largest component (87% of packages for BioC vs 50% for CRAN). The two tables below give the size of the connected components (top row) and the number of connected components of that size found in the graph (bottom row).

```
> table(listLen(connectedComp(cranDeps)))

  1   2   3   4   5   7  14 245
521  34  10   2   3   1   1   1
```

```
> table(listLen(connectedComp(biocDeps)))

  1   2   3 195
 32  10   3   1
```

Both results demonstrate the higher level of interdependency of packages in the BioC repository.

```
makeDepGraph <- function(repList, suggests.only=FALSE,
                         type=getOption("pkgType"),
                         keep.builtin=FALSE, dosize=TRUE)
{
    pkgMatList <- lapply(repList, function(x) {
        available.packages(contrib.url(x, type=type))
    })
    if (!keep.builtin)
      baseOrRecPkgs <- rownames(installed.packages(priority="high"))
    allPkgs <- unlist(sapply(pkgMatList, function(x) rownames(x)))
    if (!length(allPkgs))
      stop("no packages in specified repositories")
    allPkgs <- unique(allPkgs)
    depG <- new("graphNEL", nodes=allPkgs, edgemode="directed")
    nodeDataDefaults(depG, attr="size") <- as.numeric(NA)
    for (pMat in pkgMatList) {
        for (p in rownames(pMat)) {
            if (!suggests.only) {
                deps <- cleanPkgField(pMat[p, "Depends"])
                deps <- c(deps, cleanPkgField(pMat[p, "Imports"]))
            } else {
                deps <- cleanPkgField(pMat[p, "Suggests"])
            }
            if (length(deps) && !keep.builtin)
              deps <- deps[!(deps %in% baseOrRecPkgs)]
            if (length(deps)) {
                notFound <- ! (deps %in% nodes(depG))
                if (any(notFound))
                  depG <- addNode(deps[notFound], depG)
                deps <- deps[!is.na(deps)]
                depG <- addEdge(from=p, to=deps, depG)
            }
        }
        if (dosize) {
            sizes <- getDownloadSizesBatched(makePkgUrl(pMat))
            nodeData(depG, n=rownames(pMat), attr="size") <- sizes
        }

    }
    depG
}
```

Figure 1: The definition of the `makeDepGraph` function.

Using the Dependency Graph

The dependencies of a given package can be visualized using the graph generated by `makeDepGraph` and the **Rgraphviz** package. The graph in Figure 2 was produced using the code shown below. The `acc` method from the **graph** package returns a vector of all nodes that are accessible from the given node. Here, it has been used to obtain the complete list of **Category**'s dependencies.

```
> categoryNodes <- c("Category",
      names(acc(biocDeps, "Category")[[1]]))
> categoryGraph <- subGraph(categoryNodes,
      biocDeps)
> nn <- makeNodeAttrs(categoryGraph,
      shape = "ellipse")
> plot(categoryGraph, nodeAttrs = nn)
```

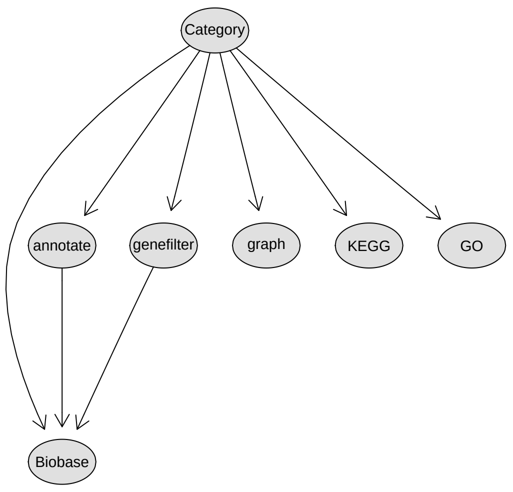

Figure 2: The dependency graph for the **Category** package.

In R , there is no easy to way to preview a given package's dependencies and estimate the amount of data that needs to be downloaded even though the `install.packages` function will search for and install package dependencies if you ask it to by specifying `dependencies=TRUE`. Next we show how to build a function that provides such a "preview" by making use of the dependency graph.

Given a plot of a dependency graph like the one for **Category** shown in Figure 2, one can devise a simple strategy to determine the install order. Namely, find the packages that have no dependencies, the leaves of the graph, and install those first. Then repeat that process on the graph that results from removing the leaves from the current graph. The download size is easily computed by retrieving the "size" node attribute for each package in the dependency list.

```
basicInstallOrder <- function(pkg, depG) {
    allPkgs <- c(pkg,
                 names(acc(depG, pkg)[[1]]))
    if (length(allPkgs) > 1) {
        pkgSub <- subGraph(allPkgs, depG)
        toInst <- tsort(pkgSub)
        if (!is.character(toInst))
          stop("depG is not a DAG")
        rev(toInst)
    } else {
        allPkgs
    }
}
```

Figure 3: Code listing for the `basicInstallOrder` function.

Figure 3 lists the definition of `basicInstallOrder`, a function that generates the complete dependencies for a given package using the strategy outlined above. The `tsort` function from **RBGL** performs a topological sort of the directed graph. A topological sort on a DAG gives an ordering of the nodes in which node *a* comes before *b* if there is an edge from *a* to *b*. Reversing the topological sort yields a valid install order.

The `basicInstallOrder` function can be used as the core of a "preview" function for package installation. The **pkgDepTools** package provides such a preview function called `getInstallOrder`. This function returns the *uninstalled* dependencies of a given package in proper install order along with the size, in megabytes, of each package. In addition, the function returns the total expected download size. Below we demonstrate the `getInstallOrder` function.

First, we create a single dependency graph for all CRAN and BioC packages.

```
> allDeps <- makeDepGraph(biocReposList())
```

Calling `getInstallOrder` for package **GOstats**, we see a listing of only those packages that need to be installed. Your results will be different based upon your installed packages.

```
> getInstallOrder("GOstats", allDeps)

$packages
    1.45MB     0.22MB      1.2MB
"annotate" "Category"  "GOstats"

$total.size
[1] 2.87551
```

When `needed.only=FALSE`, the complete dependency list is returned regardless of what packages are currently installed.

```
> getInstallOrder("GOstats", allDeps,
      needed.only = FALSE)
```

```
$packages
      0.31MB       17.16MB         1.64MB
     "graph"          "GO"      "Biobase"
      1.45MB        1.29MB         0.23MB
  "annotate"        "RBGL"         "KEGG"
      0.23MB        0.22MB          1.2MB
"genefilter"    "Category"      "GOstats"

$total.size
[1] 23.739
```

Wrap Up

We have shown how to generate package dependency graphs and preview package installation using the **pkgDepTools** package. We have described in detail how the underlying code is used and the process of modeling relationships with the **graph** package.

These tools can help identify and understand interdependencies in packages. A very similar approach can be applied to visualizing class hierarchies in R such as those implemented using the S4 (Chambers, 1998) class system or Bengtsson's **R.oo** (Bengtsson, 2006) package.

The **graph**, **RBGL**, and **Rgraphviz** suite of packages provides a very powerful means of manipulating, analyzing, and visualizing relationship data.

Bibliography

H. Bengtsson. *R.oo: R object-oriented programming with or without references*, 2006. URL http://www.braju.com/R/. R package version 1.2.3.

V. Carey and L. Long. *RBGL: Interface to boost C++ graph lib*, 2006. URL http://bioconductor.org. R package version 1.10.0.

J. M. Chambers. *Programming with Data: A Guide to the S Language*. Springer-Verlag New York, 1998.

R. Gentleman, E. Whalen, W. Huber, and S. Falcon. *graph: A package to handle graph data structures*, 2006. R package version 1.12.0.

J. Gentry. *Rgraphviz: Provides plotting capabilities for R graph objects*, 2006. R package version 1.12.0.

E. Hartuv and R. Shamir. A clustering algorithm based on graph connectivity. *Information Processing Letters*, 76(4–6):175–181, 2000. URL citeseer.ist.psu.edu/hartuv99clustering.html.

M. Kanehisa and S. Goto. KEGG: Kyoto encyclopedia of genes and genomes. *Nucleic Acids Res*, 28: 27–30, 2000.

Seth Falcon
Program in Computational Biology
Fred Hutchinson Cancer Research Center
Seattle, WA, USA
emailsfalcon@fhcrc.org

Image Analysis for Microscopy Screens

Image analysis and processing with EBImage

by Oleg Sklyar and Wolfgang Huber

The package **EBImage** provides functionality to perform *image processing* and *image analysis* on large sets of images in a programmatic fashion using the R language.

We use the term *image analysis* to describe the extraction of numeric features (*image descriptors*) from images and image collections. Image descriptors can then be used for statistical analysis, such as classification, clustering and hypothesis testing, using the resources of R and its contributed packages.

Image analysis is not an easy task, and the definition of image descriptors depends on the problem. Analysis algorithms need to be adapted correspondingly. We find it desirable to develop and optimize such algorithms in conjunction with the subsequent statistical analysis, rather than as separate tasks. This is one of our motivations for writing the package.

We use the term *image processing* for operations that turn images into images, with the goals of enhancing, manipulating, sharpening, denoising or similar (Russ, 2002). While some image processing is often needed as a preliminary step for image analysis, image processing is not the primary aim of the package. We focus on methods that do not require interactive user input, such as selecting image regions with a pointer etc. Whereas interactive methods can be extremely effective for small sets of images, they tend to have limited throughput and reproducibility.

EBImage uses the `Magick++` interface to the ImageMagick (2006) image processing library to implement much of its functionality in image processing and input/output operations.