RedeR: bridging the gap between network analysis and visualization.

Mauro A. A. Castro, Xin Wang, Florian Markowetz [*]
http://www.markowetzlab.org/software/networks.html
florian.markowetz@cancer.org.uk

July 15, 2011

Contents

[*]Cancer Research UK - Cambridge Research Institute, Robinson Way Cambridge, CB2 0RE, UK.

1 Overview

RedeR is an R-based package combined with a Java application for dynamic network visualization and manipulation. It implements a callback engine by using a low-level R-to-Java interface to build and run common plugins. In this sense, *RedeR* takes advantage of ***R*** to run robust statistics, while the R-to-Java interface bridge the gap between network analysis and visualization: for **R Developers**, it allows the development of Java plug-ins exclusively using R codes; for **Java Users**, it runs R methods implemented in a stand-alone application, and for **R Users** *RedeR* interactively displays R graphs using a robust Java graphic engine embedded in *R*.

RedeR use different strategies to link R to Java:

- Data interface: implements the callback engine to make calls from R via xml-rpc protocol. It sets *R* as client and *RedeR* as server.
- Graphic interface: implements the callback engine to make calls from Java via dynamic libraries. It wraps R graphics into RedeR classes.

The design of the software is depicted from Figure 1. One unique feature of this concept is how R methods can be wrapped and exported. For example, in a few lines of code *RedeR* sends R methods to the Java app using the *submitPlugin* function, which gives rise to a new Java plugin. Also, complex graphs with many attributes can be transferred from-and-to *R* using *addGraph* and *getGraph* functions.

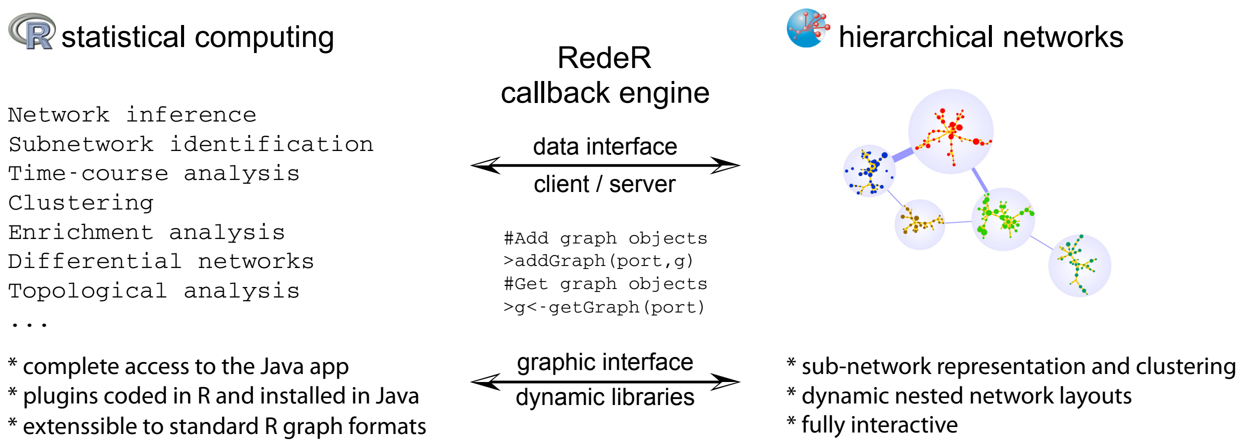


Figure 1: Schematic representation of RedeR calls. In the low-level interface, packages like XMLRPC[1] and rJava[2] are used to link R to Java.

2 Quick start

2.1 Main callback methods

The first step is to build the server port, which will be required in all remote procedure calls. By default the constructor *RedPort* should set all details:

```
> library(RedeR)
> rdp <- RedPort()
```

Next, invoke RedeR using the method *calld*:

```
> calld(rdp)
```

Within an active interface, then the method 'addGraph' can easily send R graphs to the application. For example, the following chunk adds an *igraph*[3] object:

```
> g1 <- graph.lattice(c(5, 5, 5))
> addGraph(rdp, g1, layout.kamada.kawai(g1))
```

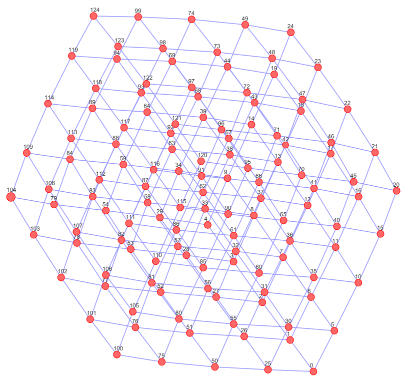


Figure 2: A toy example added to *RedeR* by the *addGraph* function.

Conversely, RedeR graphs can be transferred to R and wrapped in *igraph* objects:

```
> g2 <- getGraph(rdp)
> resetd(rdp)
```

The interface accepts additional graph attributes, as for example edge direction, edge width, edge weight, node shape, node size, node color etc. In *igraph* objects, vertex and edge attributes can be assigned as arbitrary R objects. In order to pass these extensible features to *RedeR* the attributes must be provided in a valid syntax. [1]

Another strategy is to wrap graphs into containers and then send it to the Java app. Next, the subgraphs g3 and g4 are assigned to different nested structures (Fig.3).

[1]See *getGraph* and *addGraph* specification for additional details.

```
> g3 <- barabasi.game(10)
> g4 <- barabasi.game(10)
> V(g3)$name <- paste("sn", 1:10, sep = "")
> V(g4)$name <- paste("sm", 1:10, sep = "")
> addGraph(rdp, g3, isNest = TRUE, gcoord = c(25, 25), gscale = 50)
> addGraph(rdp, g4, isNest = TRUE, gcoord = c(75, 75), gscale = 50)
```

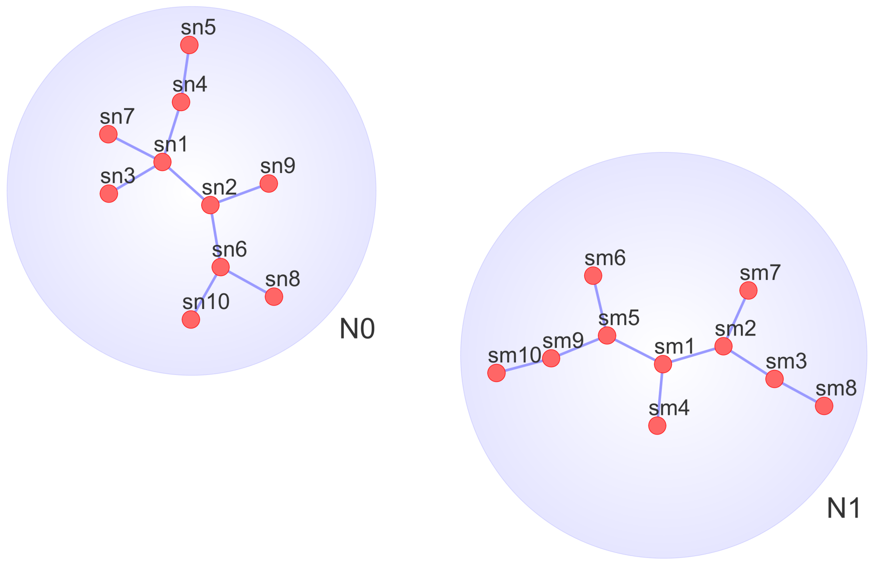


Figure 3: Graphs nested in *RedeR* by the command *addGraph*.

In this case, the subgraphs can be handled apart from each other. For example, the following chunk selects all nodes assigned to the container ”N0” and then gets back the subgraph (the selection step can also be done interactively!).

```
> selectNodes(rdp, "N0")
> g5 <- getGraph(rdp, status = "selected")
> resetd(rdp)
```

As a suggestion, try some RedeR features in the Java side (e.g. open samples s2 or s3 in the main panel and enjoy the dynamic layout options!).

2.2 Working interactively

The next chunk generates a scale-free graph according to the Barabasi-Albert model[3] and sends the graph to RedeR without any layout information.

```
> g6 <- barabasi.game(500)
> addGraph(rdp, g6)
```

Then using the dynamic function available in the app you can layout the graph as presented in Figure 4.

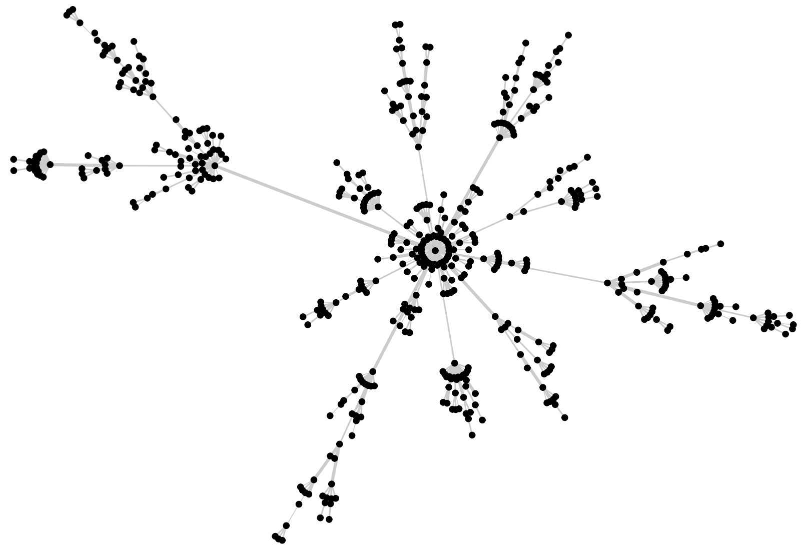
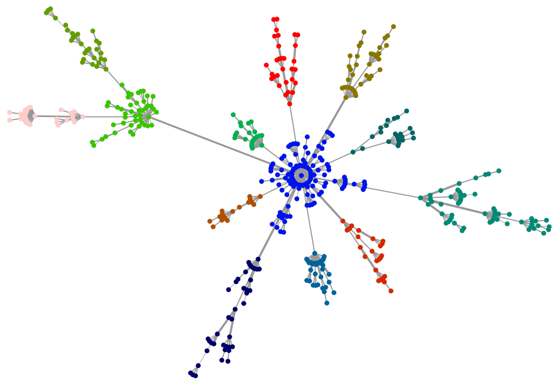


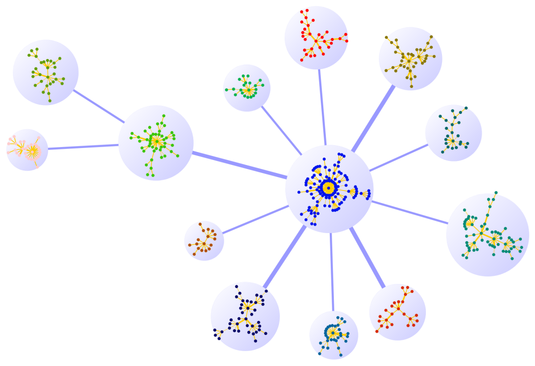
Figure 4: Scale-free graph according to the Barabasi-Albert model[3].

In Figure 5a the same graph is used to exemplify the community structure mapped by the edge-betweenness function available in RedeR. In Figure 5b these communities are nested to containers, which are objects of the same class of the nodes but with additional behaviors: it can be hidden and anchored to the main panel (Fig.5c). You can build these containers either using *R* or *Java* functions (see options available in the *clustering* main menu and in the shortcuts of nested objects).

For the next example you will need to reproduce in *RedeR* app the graph from Figure 5b (or any graph with containers), then select one of the communities and run the chunk below: a simple degree distribution should be plotted in the R side (Fig.6). This is the first step to illustrate how to build an interactive plugin.

```
> g <- getGraph(rdp, status = "selected")
> if (vcount(g) > 0) plot(degree.distribution(g), xlab = "k", ylab = "P(k)", pch = 19)
```


(a) Communities


(b) Graphs into containers

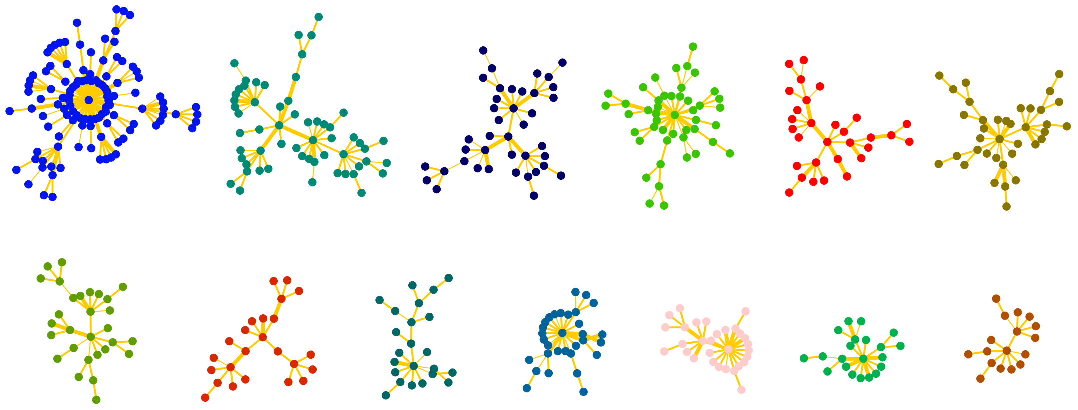

(c) Subnetworks

Figure 5: Community structure: (a) subgraphs detected based on edge betweenness; (b) nested communities into containers; (c) subnetworks in hidden containers.

2.3 Plugin builder

RedeR plug-ins have two main sections: methods and add-ons. The 'methods' section can be regarded as the plug-in trigger. When installed in the Java app, this trigger is used to start a given analysis by unfolding the R expressions wrapped in the methods. Add-ons use the same strategy, but remains hidden in the app – and it is optional. Formal functions can be passed to add-ons as additional arguments. Prior to the main call, all functions are automatically loaded in R, making the source code available to the subsequent analysis.

A simple example

- Wrap methods in functions

  Here, the degree distribution is used in the same way as illustrated in the previous section but – prior to execute the analysis – the plugin requires two commands: *RedPort* to set the interface, and *dynwin* to wrap R graphics in RedeR Java classes.

```
> mt1 <- function() {
+     rdp <- RedPort("MyPort")
+     dynwin(rdp)
+     g <- getGraph(rdp, status = "selected")
+     if (vcount(g) > 0)
+         plot(degree.distribution(g), xlab = "k", ylab = "P(k)", pch = 19)
+ }
```

- Initiate plugin skeleton:

  Now the method *mt1* is added to the *PluginBuilder*, and is ready to be sent to the application.

```
> plugin <- PluginBuilder(title = "MyPlugin", allMethods = list(mt1 = mt1))
```

- Submit new plugins to RedeR:

  The submission is straightforward: in the Java side, the plugin will be displayed in the main menu provided that the interface is properly set to find *R*.

```
> submitPlugin(rdp, plugin)
> updatePlugins(rdp)
> exitd(rdp)
```

As a suggestion, try the heatmap plugin in RedeR. This is the first default plugin of the software; hopefully others will be available soon!

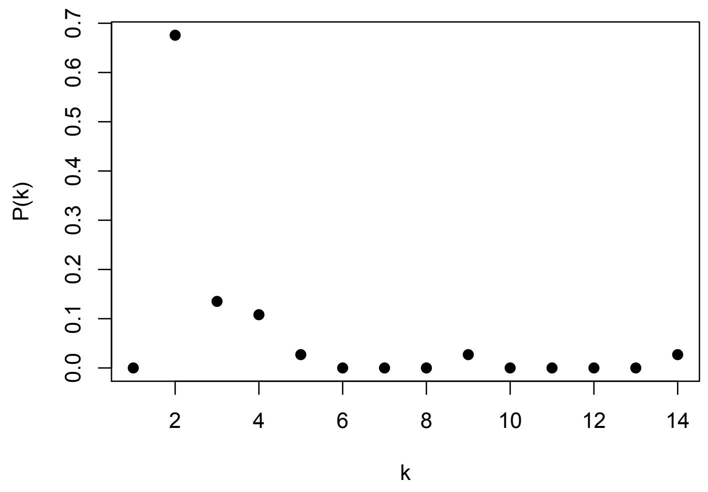


Figure 6: RedeR plugins: a simple example to illustrate the concept (degree distribution of selected subgraphs). One subgraph from Figure 5b was interactively selected in *RedeR* and then plotted either in the R side or in the Java side of the interface using the plugin described above.

3 Installation

3.1 The RedeR package

The RedeR package is freely available from Bioconductor at `http://www.bioconductor.org`. In addition, you might need to install other dependencies (e.g. RCurl and XML; section 3.3 shows a typical R session).

3.2 The RedeR application

The RedeR jar file is already included in the R package and, as usual, to run Java applications your system must have a copy of the JRE (Java Runtime Environment, version>= 5). The RedeR software can be used as a stand-alone application, or even embedded in other softwares, but in order to use plugins the interface must be set properly in the Java side to find R (e.g. path to R home). This should be a simple task:

- Using RedeR embedded in R: it is automatic ..but if your R environment deviates a lot from the default installation then it might be necessary to add some path manually. The function *calld* accepts additional arguments for these cases.
- Using RedeR as stand-alone application: follow the settings in the main menu and add R paths if required. A script file is generated to do all the job.

If you find any difficulty, please contact us.

3.3 Session information

```
R version 2.12.2 (2011-02-25)
Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] RedeR_0.99.3   igraph_0.5.5-2 XMLRPC_0.2-4

loaded via a namespace (and not attached):
[1] RCurl_1.4-3  XML_3.2-0    tools_2.12.2
```

References

[1] Duncan Temple Lang. *XMLRPC: Remote Procedure Call (RPC) via XML in R*, 2010. R package.

[2] Simon Urbanek. *rJava: Low-level R to Java interface*, 2010. R package.

[3] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

4 Supplements

The R chunks below can be used to reproduce the pre-processed dataset available at ”RedeR.data” object. This dataset integrates a case study that soon is going to appear in a new communication!

- Pre-processing pipelines for differential expression, clustering and co-expression analysis:

  ```
  > source("casePart1.R")
  > deaPipeline()
  > deaFigures()
  > clustPipeline(nboot = 1000)
  > clustFigures()
  > correl <- ceaPipeline()
  > ceaFigures(correl)
  > pvchip <- pvchipPipeline()
  > pvchipFigures(pvchip)
  > dataPipeline4RedeR(correl, pvchip)
  > graphPipeline4RedeR()
  ```

  p.s.1 source ’casePart1.R’ is available in the package ’script’ directory.
  p.s.2 requirements: RedeR, limma, snow, pvclust and gplots packages.
  p.s.3 Snow package might need adjustments to run parallel computing!

- Pre-processing pipelines for enrichment analyses and subnetwork identification

  ```
  > source("casePart1.R")
  > analysesPipeline()
  > gscoPipeline4RedeR()
  > ppiPipeline4RedeR()
  ```

  p.s.1 source ’casePart2.R’ is available in the package ’script’ directory.
  p.s.2 requirements: RedeR, BioNet, DLBCL, ALL, limma, graph, org.Hs.eg.db, KEGG.db, GO.db, AnnotationDbi, snow, biomaRt and HTSanalyzeR packages.
  p.s.3 Snow package might need adjustments to run parallel computing!