

R PROFILING AND OPTIMISATION

GÜNTHER SAWITZKI

PENDING CHANGES

Warning: this is under construction.

This vignette contains experimental material which may sink down to the package implementation, or vanish.

Known issues:

- Control information may be included as special stack in raw format.
- A list of profiles may become default. Only one profiling interval value per profile.
- Nodes may be implemented as *factor*. Work-around for the R factor handling needs to be added, i.e. *factor* as a data structure.
- changing timing interval is too expensive, as rle is not transparent to data frames. Implement profiles as a list, with a time interval attribute per list element.

CONTENTS

Pending changes	1
Profiling facilities in R	2
L ^A T _E X layout tools and R settings	3
1. Profiling	6
1.1. Simple regression example	7
1.1.1. R basic	8
1.1.2. Package sprof	11
1.1.3. Node classes	17
2. A better grip on profile information	20
2.1. The internal details	20
2.2. The free lunch	28
2.3. Cheap thrills	28
2.3.1. Trimming	32
2.3.2. Surgery	34

Date: May 2013. *Revised:* July 2013

Typeset, with minor revisions: August 24, 2013 from SVN *Revision* : 219 2013-08-24.

Key words and phrases. R programming, profiling, optimisation, R programming language.

An R vignette for package sprof.

URL: <http://sintro.r-forge.r-project.org/>

Private Version

2.4. Run length	37
3. Graph package	42
4. Standard output	51
4.1. Print	51
4.2. Summary	51
4.3. Plot	52
5. More Graphs	53
5.1. Example: regression	53
5.1.1. graph package	54
5.1.2. igraph package	55
5.1.3. network package	69
5.1.4. Rgraphviz package	74
6. Template	79
Index	80
7. xxx – lost & found	83
7.1. Trimmed example: regression	95
7.1.1. graph package	97
7.1.2. igraph package	101
7.1.3. network package	103
7.1.4. Rgraphviz package	109

PROFILING FACILITIES IN R

For the impatient: table 1 on page 9 and table 2 on page 10 give you information from profiling 100 runs of a simple linear regression. You get another view of the same information by fig. 16 on page 50. If you want to know more, please have some patience.

R provides the basic instruments for profiling, both for time based samplers as for event based instrumentation. Information on R profiling is in section 3.2 “Profiling R code for speed” and section 3.3. “Profiling R code for memory use” of “Writing R Extensions” <http://cran.r-project.org/doc/manuals/R-exprs.html>. Specific information on memory profiling is in <http://developer.r-project.org/memory-profiling.html>.

However this source of information seems to be rarely used.

Maybe the supporting tools are not adequate. The summaries provided by R reduce the information beyond necessity. Additional packages are available, but these are not sufficiently action oriented.

With package *sprof* we want to give a data representation that keeps the full profile information. Tools to answer common questions are provided. The data structure should make it easy to extend the tools as required.

The package is currently distributed at r-forge as part of the *sintro* material.

To install this package directly within R, type

```
install.packages("sprof", repos="http://r-forge.r-project.org")
```

To install the recent package from source directly within R, type
`install.packages("sprof", repos="http://r-forge.r-project.org", type="source")`

L^AT_EX LAYOUT TOOLS AND R SETTINGS

You may want to skip this section, unless you want to modify the vignette for your own purposes, or look at the internals.

To make sure we do not depend on packages collected along the way, we clean up packages. Calls to this function will be hidden.

Input

```
cleanpackages <- function()
{
  # make sure we do not get inherited methods or garbage here.
  try(detach("package:sna"), silent=TRUE)
  try(detach("package:igraph"), silent=TRUE)
  try(detach("package:network"), silent=TRUE)
  try(detach("package:graph"), silent=TRUE)
  try(detach("package:Rgraphviz"), silent=TRUE)
  try(detach("package:graph"), silent=TRUE)
}
```

The main library we are going to use is *sprof*.

Input

```
library(sprof)
```

We want immediate warnings, if necessary. So we set *warn* to level 1. Set *warn* to level 2 if you want to handle warnings as error.

Input

```
message("switching options(warn=1) -- immediate warning on")
options(warn=1)
```

We want a second chance on errors. So we install an error handler.

Input

```
#options(error = recover)
```

Print parameters used here:

Input

```
options(width = 72)
options(digits = 6)
```

For *str* output, we generally use these settings:

Input

```
strx <- function(x,
  max.level=2, vec.len=3,
  nchar.max=40,
  list.len=12,
```

```

width=70, strict.width="wrap",...)
{
  cat(paste("##strx:",deparse(substitute(x)),"\n"))
  str(x, max.level=max.level,
      vec.len=vec.len,
      nchar.max=nchar.max,
      list.len=list.len,
      width=width, strict.width=strict.width,...)
}

```

For larger tables and data frames, we use a kludge to avoid long outputs.

```

                                Input
xcutrows <- function(df, cut, margin=5)
{
  if (!is.data.frame(df)) return(df)
  nrow <- nrow(df)
  # cut a range if it is not empty.
  # Quiet noop else.
  # Does not cut single lines.
  cutrng <- function(cutfrom,cutto, cutname="<cut>"){
    if (cutfrom<cutto){
      df[cutfrom,] <- NA
      if (!is.null(rownames(df))) rownames(df)[cutfrom] <- cutname
      if (!is.null(df$name)) df$name[cutfrom] <- ""

      cutfrom <- cutfrom+1
      df[-(cutfrom:cutto),]
    }#if
  }

  if (!missing(cut)) {df <- cutrng(cut[1],cut[2]); return(df)}
  if (length(margin)==0) return(df)
  if (length(margin)==1) margin <- c(margin,margin)
  if (length(margin)==2) {
    cut <- c(margin[1]+1,nrow-margin[2])
    df <- cutrng(cut[1],cut[2]);
    return(df)
  }

  delta <- (nrow-sum(margin)) %/% 2
  if (delta<2) return(df)
  c1 <- margin[1]+1
  c2 <- c1 + delta -1
  c4 <- nrow- margin[3]
  c3 <- c4 - delta
  df <- cutrng(c3,c4, cutname="<cut hi>")
  df <- cutrng(c1,c2, cutname="<cut low>")
  return(df)
}

```

We use the R function `xtable()` for output and \LaTeX `longtable`. A convenient wrapper to use this in our *Sweave* source is given here. Among others, it adds `zero.print`.

ToDo: remove text
vdots from string/
name columns.
Note: this may be
a factor. Use empty
string.

```

                                Input
library(xtable)
prxt <- function (x, digits=2, cut=TRUE, caption=NULL,
                  label=NULL, zero.print=NULL, print.results=TRUE,...) {

    if (cut) {margin <- 10
    if (nrow(x)> 2*margin+3) x <- xcutrows(x, margin=margin)}

    #special sanitising for xtable
    xr <- rownames(x)
    #for (i in (1:length(xr))) {xr[i] <- sub(xr[i],"\\[","$[", fixed=TRUE)}
    #xr <- paste("$",xr,"$")
    xr <- gsub("[","{",xr, fixed=TRUE)
    xr <- gsub("_","\\_",xr, fixed=TRUE)
    xr <- gsub("^","\\^",xr, fixed=TRUE)
    rownames(x)<- xr

    pr <- print(
        xtable(x, digits=digits, caption=caption,
              label=label, ...),
        floating=FALSE,
        tabular.environment="longtable",
        caption.placement="top",
        zero.print = ".",
        NA.string="\\vdots",
        print.results=FALSE)
    #
        NA.string="", #NA.string="\\vdots",

    pr <- gsub( "$\\backslash$vdots","\\vdots",x=pr, fixed=TRUE)

    if(!is.null(zero.print))
        pr <- gsub( " 0 ",zero.print, x=pr, fixed=TRUE)

    if (print.results) cat(pr)
    invisible(pr)
}

```

This is to be used with `<<print=FALSE, results =tex, label=tab:prxx>>=`

The graph visualisation family is not friendly. We try to get control by using a wrapper which is at least used to the members of the *graphviz* clan. This will be used in later sections.

```

                                Input
plotviz <- function(x, layout="dot", main=NULL, sub=NULL,...)
{
    xid <- deparse(substitute(x))
    xsubid <- NULL

```

```

class1 <- NULL
if (inherits(x,"sprof")) {
  class1 <- paste0("class orig:",class(x))
  xsubid <- x$info$id
  sub <- paste0(sub, " ", x$info$id)
  x <- as(adjacency(x), "graphNEL")
}

if (!is.null(sub)) sub <- as.character(sub)
if (is.null(main))
  main <- paste0("plotviz( ", xid, ", ", layout, " )\n", xsubid) else
  main = paste0(main, "\n plotviz( ", xid, ", ", layout, " )\n", xsubid)

if (inherits(x,"Ragraph")) {
  plot(x=x, y=layout,
       cex.main=1.2,
       main=main,sub=sub,...)
} else {
  plot(x=x, y=layout,
       attrs=list(
         node=list(cex=4, fontsize=40, shape="ellipse")),
       cex.main=1.2,
       main=main,sub=sub,...)
}

# title(sub = paste0(sub, " ", as.character(class(x)))      )
  title(sub = "xx try sub xx")
legend("topleft",

      legend=c(class1,
               paste0("class: ",class(x)),
               paste0("layout: ",layout)),
      bg="#FFFFE040",
      seg.len=0
      )#"lightyellow" =#FFFFE0
}

```

1. PROFILING

The basic information provided by all profilers is a protocol of sampled stacks. For each recorded event, the protocol has one record, such as a line with a text string showing the sampled stack.

We use profiles to provide hints on the dynamic behaviour of programs. Most often, this is used to improve or even optimise programs. Sometimes, it is even used to understand some algorithm.

Profiles represent the program flow, which is considered to be laid out by the control structure of a program. The control structure is represented by the control graph, and this leads to the common approach to (re)construct the control graph, map the

profile to this graph, and used graph based methods for further analysis. The prime example for this strategy is the GNU profiler *gprof* (see <http://sourceware.org/binutils/docs/gprof/>) which is used as master plan for many common profilers.

It is only half of the truth that the control graph can serve as a base for the profiled stacks. In R, we have some peculiarities.

lazy evaluation: Arguments to functions can be passed as promises. These are only evaluated when needed, which may be at a later time, and may then lead to insertions in the stack. So we may have information resulting from the data flow, interspersed with the control flow.

memory management: Allocation of memory, and garbage collection, may interfere and leave their traces in the stack. While allocation is closely related to the visible control flow, garbage collection is a collective effect largely out of control of the code to execute.

primitives: Internal functions may escape the usual stack conventions and execute without leaving any identifiable trace on the stack.

control structures: In R, many control structures are implemented as function. Most notably, the *apply()* family appears as function calls and can lead to cliques in the graph representation that do not correspond to relevant structures. Since these functions are well known, they can have a special treatment.

So while the stack follows an overall well known dynamics, in R there are exceptions from regularity.

The general approach, by *summaryRprof()* and others, is to reduce the profile to node information, or to consider single transitions.

We take a different approach. We take the stacks, as recorded in the profiles as our basic information unit. From this, we ask: what are the actions we need to answer our questions? Representation in graphs may come later, if they can help.

If the stacks would come from the control flow only, we could make use of the sequential nature of stacks. But since we have to live with the R specific interferences, we stay with the raw stacks.

In this presentation, we will use a small list of examples. Since *Rprof* is not implemented on all systems, and since the profiles tend to get very large, we use some prepared examples that are frozen in this vignette and not included in the distribution, but all the code to generate the examples is provided.

ToDo: rearrange
stacks? detect
order?

1.1. Simple regression example.

Input

```
n <- 10000
x <- runif(n)
err <- rnorm(n)
y <- 2+ 3 * x + err
reg0data <- data.frame(x=x, y=y, err=err)
rm(x,y,err)
```

We will use this example to illustrate the basics. Of course the immediate questions are the variance between varying samples, and the influence of the sample size n . We keep everything fixed, so the only issue for now is the computational performance under strict iid conditions.

Still we have parameters to choose. We can determine the profiling granularity by setting the timing interval, and we can use repeated measurements to increase precision below the timing interval.

The timing interval should depend on the clock speed. Using for example 1ms amounts to some 1000 steps on a current CPU, per kernel.

If we use repeated samples, the usual rules of statistics applies. So taking 100 runs and taking the mean reduces the standard deviation by a factor $1/10$.

By the usual R conventions, seconds are used as time base for parameters. However report will use ms as a time base.

Here is an example how to take a profile, using basic R. See section 1.1.2 on page 11 how to use *sampleRprof* in package *sprof* for an easier solution.

ToDo: Can we calibrate times to CPU rate? Introduce cpu clock cycle as a time base

Input

```

profinterval <- 0.001
simruns <- 100
Rprof(filename="RprofsRegressionExpl.out", interval = profinterval)
  for (i in 1:simruns) xxx<- summary(lm(y~x, data=reg0data))
Rprof(NULL)

```

We now have the profile data in a file *RprofsRegressionExpl.out*. For this vignette, we use a frozen version *RprofsRegressionExpl01.out*.

1.1.1. *R basic*. The basic R functions invite us to get a summary.

Input

```

sumRprofRegressionExpl <- summaryRprof("RprofsRegressionExpl01.out")
#str(profile_nodes_rle, max.level=2, vec.len=3, nchar.max=40, list.len=6)
strx(sumRprofRegressionExpl)

```

Output

```

##strx: sumRprofRegressionExpl
List of 4
$ by.self : 'data.frame': 41 obs. of 4 variables:
..$ self.time : num [1:41] 0.087 0.057 0.051 0.043 0.042 0.04 0.032
  0.026 ...
..$ self.pct : num [1:41] 16.67 10.92 9.77 8.24 ...
..$ total.time: num [1:41] 0.113 0.099 0.069 0.043 0.474 0.045 0.033
  0.114 ...
..$ total.pct : num [1:41] 21.65 18.97 13.22 8.24 ...
$ by.total : 'data.frame': 62 obs. of 4 variables:
..$ total.time: num [1:62] 0.522 0.522 0.521 0.521 0.521 0.521 0.521
  0.521 ...
..$ total.pct : num [1:62] 100 100 99.8 99.8 ...
..$ self.time : num [1:62] 0.006 0 0.001 0 0 0 0 0 ...
..$ self.pct : num [1:62] 1.15 0 0.19 0 0 0 0 0 ...

```



```
$ sample.interval: num 0.001
$ sampling.time : num 0.522
```

The summary reduces the information contained in the profile to marginal statistics per node. This is provided in two data frames giving the same information, only in different order.

The file contains several spurious recordings: nodes that have been recorded only few times. It is worth noting these, but then they better be discarded. We use a time limit of 4ms, which given our sampling interval of 1ms means we require more than four observations.

```
prxt(sumRprofRegressionExpl$by.self,
      caption="summaryRprof result: by.self as final stack entry, all records",
      label="tab:prSRREbs")
```

Table 1: summaryRprof result: by.self as final stack entry, all records

	self.time	self.pct	total.time	total.pct
"lm.fit"	0.09	16.67	0.11	21.65
"{ }.data.frame"	0.06	10.92	0.10	18.97
"model.matrix.default"	0.05	9.77	0.07	13.22
"as.character"	0.04	8.24	0.04	8.24
"lm"	0.04	8.05	0.47	90.80
"summary.lm"	0.04	7.66	0.04	8.62
"structure"	0.03	6.13	0.03	6.32
"na.omit.data.frame"	0.03	4.98	0.11	21.84
"anyDuplicated.default"	0.02	4.21	0.02	4.21
"as.list.data.frame"	0.02	4.21	0.02	4.21
<cut>	:	:	:	:
"FUN"	0.00	0.19	0.01	1.34
"%in%"	0.00	0.19	0.00	0.77
"deparse"	0.00	0.19	0.00	0.38
"\$"	0.00	0.19	0.00	0.19
"as.list.default"	0.00	0.19	0.00	0.19
"as.name"	0.00	0.19	0.00	0.19
"coef"	0.00	0.19	0.00	0.19
"file"	0.00	0.19	0.00	0.19
"NCOL"	0.00	0.19	0.00	0.19
"terms.formula"	0.00	0.19	0.00	0.19

```
prxt(
  sumRprofRegressionExpl$by.total[
    sumRprofRegressionExpl$by.total$total.time>0.004,],
  caption="summaryRprof result: by.total, total time > 4ms",
  label="tab:prSRREbt")
```

Table 2: summaryRprof result: by.total, total time > 4ms

	total.time	total.pct	self.time	self.pct
"<Anonymous>"	0.52	100.00	0.01	1.15
"Sweave"	0.52	100.00	0.00	0.00
"eval"	0.52	99.81	0.00	0.19
"doTryCatch"	0.52	99.81	0.00	0.00
"evalFunc"	0.52	99.81	0.00	0.00
"try"	0.52	99.81	0.00	0.00
"tryCatch"	0.52	99.81	0.00	0.00
"tryCatchList"	0.52	99.81	0.00	0.00
"tryCatchOne"	0.52	99.81	0.00	0.00
"withVisible"	0.52	99.81	0.00	0.00
<cut>	:	:	:	:
"as.list"	0.02	4.41	0.00	0.00
"anyDuplicated.default"	0.02	4.21	0.02	4.21
"as.list.data.frame"	0.02	4.21	0.02	4.21
"sapply"	0.01	2.68	0.00	0.19
"match"	0.01	2.11	0.00	0.19
"{{{}}}.data.frame"	0.01	1.53	0.00	0.19
"{{{}}}"	0.01	1.53	0.00	0.00
"rep.int"	0.01	1.34	0.01	1.34
"FUN"	0.01	1.34	0.00	0.19
"list"	0.01	0.96	0.01	0.96

1.1.2. *Package sprof*. In contrast to the common R packages, in the *sprof* implementation we take a two step approach. First we read in the profile file to an internal representation. Analysis is done in later steps.

Input

```
sprof01<- readRprof("RprofsRegressionExpl01.out")
```

The data contain identification information for reference. This will be used in the functions of *sprof* and shown in the displays. Here is the summary of this section:

Input

```
str(sprof01$info)
```

	Output
'data.frame':	1 obs. of 9 variables:
\$ id	: chr "RprofsRegressionExpl01.out 2013-06-13 23:46:04"
\$ date	: POSIXct, format: "2013-08-24 22:15:44"
\$ nrnodes	: int 62
\$ nrstacks	: int 50
\$ nrrecords	: int 522
\$ sample.interval:	num 0.001
\$ sampling.time	: num 0.522
\$ ctllines	: chr "sample.interval=1000"
\$ ctllinenr	: num 1

For this vignette, we change the *id* information. So in this context:

Input

```
sprof01$info$id <- "sprof01"
```

We keep this example and use the copy *sprof01* of it extensively for illustration.

Input

```
save(sprof01, file="sprof01m.RData")
```

To run the vignette with a different profile, replace *sprof01* by your example. You still have the file for reference.

Package *sprof* provides a function *sampleRprof()* to take a sample and create a profile on the fly, as in

Input

```
sprof01temp <- sampleRprof(runif(10000), runs=100)
```

The basic data structure consists of four data frames. The *info* section collects global information from the input file, such as an identification strings and various global matrix. The *nodes* section initially gives the same information marginal information as *summaryRprof*. The *stacks* section puts the node information into their calling context as found in the input profile file. The *profiles* section gives the temporal context. It is implemented as a list, but conceptually it is a data frame. Implementing it as a list allows run length encoding of variables, which unfortunately is not allowed by R in data frames.

```
strx(sprof01)
```

Input

```
Output
```

```
##strx: sprof01
List of 4
$ info : 'data.frame': 1 obs. of 9 variables:
..$ id : chr "sprof01"
..$ date : POSIXct[1:1], format: "2013-08-24 22:15:44"
..$ nrnodes : int 62
..$ nrstacks : int 50
..$ nrrecords : int 522
..$ sample.interval: num 0.001
..$ sampling.time : num 0.522
..$ ctllines : chr "sample.interval=1000"
..$ ctllinenr : num 1
$ nodes : 'data.frame': 62 obs. of 5 variables:
..$ name : Factor w/ 62 levels "!", "..getNamespace", ...: 1 2 3 4 5 6 7
  8 ...
..$ self.time : num [1:62] 2 0 2 0 0 57 0 1 ...
..$ self.pct : num [1:62] 0.38 0 0.38 0 ...
..$ total.time: num [1:62] 2 1 4 26 99 99 8 8 ...
..$ total.pct : num [1:62] 0.03 0.01 0.05 0.34 1.29 1.29 0.1 0.1 ...
$ stacks : 'data.frame': 50 obs. of 7 variables:
..$ nodes :List of 50
.. .. [list output truncated]
..$ shortname : Factor w/ 50 levels
  "S<A>eFttCtCLtC0dTCwVeesleem.m..n.n...[\"| __truncated__,...: 27 17
  19 1 35 36 37 30 ...
..$ refcount : num [1:50] 1 5 26 55 13 43 51 87 ...
..$ stacklength : int [1:50] 19 20 19 21 14 15 15 14 ...
..$ stackheadnodes: int [1:50] 52 52 52 52 52 52 52 52 ...
..$ stackleafnodes: int [1:50] 27 28 41 6 39 14 38 30 ...
..$ stackssrc : Factor w/ 50 levels "!.data.frame [
  na.omit.data.frame na.\"| __truncated__,...: 27 28 39 5 37 13 36 30
  ...
$ profiles:List of 4
..$ data : int [1:522] 1 2 2 3 4 4 5 5 ...
..$ mem : NULL
..$ malloc : NULL
..$ timesRLE:List of 2
.. ..- attr(*, "class")= chr "rle"
- attr(*, "class")= chr [1:2] "sprof" "list"
```

The nodes do not come in a specific order. Access via a permutation vector is preferred. This allows different views on the same data set. For example, table 4 on page 14 uses a permutation by total time, and a selection (compare to table 2 on page 10). The only difference is that *sprof* works uses a millisecond (ms) base, whereas R in general uses seconds as a base.

```
nodes <- sprof01$nodes[order(sprof01$nodes$self.time,
decreasing=TRUE),]
```

Input

```
prxt(nodes[nodes$self.time>4,],
caption="sprof result: by.self, self time > 4ms",
label="tab:prspbtself")
```

Table 3: sprof result: by.self, self time > 4ms

	name	self.time	self.pct	total.time	total.pct
30	lm.fit	87.00	16.67	113.00	1.47
6	[.data.frame	57.00	10.92	99.00	1.29
38	model.matrix.default	51.00	9.77	69.00	0.90
14	as.character	43.00	8.24	43.00	0.56
29	lm	42.00	8.05	474.00	6.16
51	summary.lm	40.00	7.66	45.00	0.59
49	structure	32.00	6.13	33.00	0.43
41	na.omit.data.frame	26.00	4.98	114.00	1.48
13	anyDuplicated.default	22.00	4.21	22.00	0.29
16	as.list.data.frame	22.00	4.21	22.00	0.29
40	na.omit	20.00	3.83	134.00	1.74
39	model.response	13.00	2.49	56.00	0.73
36	model.frame.default	12.00	2.30	168.00	2.18
46	rep.int	7.00	1.34	7.00	0.09
10	<Anonymous>	6.00	1.15	522.00	6.79
28	list	5.00	0.96	5.00	0.07

At this level, it is helpful to note the expectations, and only then inspect the timing results. Since we are using a linear model, we are not surprised to see functions related to linear models on the top of the list. We may however be surprised to see functions related to data access and to character conversion very high on the list. The sizeable amount of time spent on NA handling is another aspect that is surprising.

```
Input
nodes <- sprof01$nodes[order(sprof01$nodes$total.time, decreasing=TRUE),]
prxt(nodes[nodes$total.time>4,],
caption="sprof result: by.total, total time > 4ms",
label="tab:prspbt")
```

Table 4: sprof result: by.total, total time > 4ms

	name	self.time	self.pct	total.time	total.pct
10	<Anonymous>	6.00	1.15	522.00	6.79
52	Sweave	0.00	0.00	522.00	6.79
21	doTryCatch	0.00	0.00	521.00	6.78
22	eval	1.00	0.19	521.00	6.78
23	evalFunc	0.00	0.00	521.00	6.78
55	try	0.00	0.00	521.00	6.78
56	tryCatch	0.00	0.00	521.00	6.78
57	tryCatchList	0.00	0.00	521.00	6.78

58	tryCatchOne	0.00	0.00	521.00	6.78
62	withVisible	0.00	0.00	521.00	6.78
<cut>	\vdots	⋮	⋮	⋮	⋮
61	vapply	3.00	0.57	23.00	0.30
13	anyDuplicated.default	22.00	4.21	22.00	0.29
16	as.list.data.frame	22.00	4.21	22.00	0.29
47	sapply	1.00	0.19	14.00	0.18
31	match	1.00	0.19	11.00	0.14
7	[[0.00	0.00	8.00	0.10
8	[[.data.frame	1.00	0.19	8.00	0.10
25	FUN	1.00	0.19	7.00	0.09
46	rep.int	7.00	1.34	7.00	0.09
28	list	5.00	0.96	5.00	0.07

ToDo: remove text
vdots from string/-
name columns

Given the sampling structure of the profiles, two aspects are common. The sampling picks up scaffold functions with a high, nearly constant frequency. And the sampling will pick up rare recordings that are near to detection range. The display functions hide these effects by default. In our example, about half of the nodes are cleared by this garbage collector.

Common rearrangements as by total time and by self time are supplied by the display functions.

`plot_nodes()`, for example, currently gives a choice of four displays for nodes, and supports trimming by default. Our profile starts with 62 nodes. The defaults cut off 34 nodes as uninformative, either because they are too rare, or ubiquitous.

Input

```
#8
oldpar <- par(mfrow=c(2,2))
plot_nodes(sprof01)
par(oldpar)
```

Basic information on node level: see fig. 1 on the facing page.

Information in the time scatterplots may sometimes be more accessible when using a logarithmic scale, so this is added.

If you prefer, you can have the bar charts in horizontal layout, giving more space for labels (Basic information on node level - horizontal bars: see fig. 2 on page 16).

Input

```
#8
oldpar <- par(mfrow=c(2,2))
plot_nodes(sprof01, horiz=TRUE)
par(oldpar)
```

We can add colour. To illustrate this, we encode the frequency of the nodes as colour. As a palette, we choose a heat map here.

ToDo: apply colour
to selection?



```

# Input
freqrank01 <- rkindex(-sprof01$nodes$total.time, ties.method="random")
freqrankcol01 <- heat.colors(length(freqrank01))

```

```
#10
sprof01$nodes$icol <- freqrank01
oldpar <- par(mfrow=c(2,2))
```

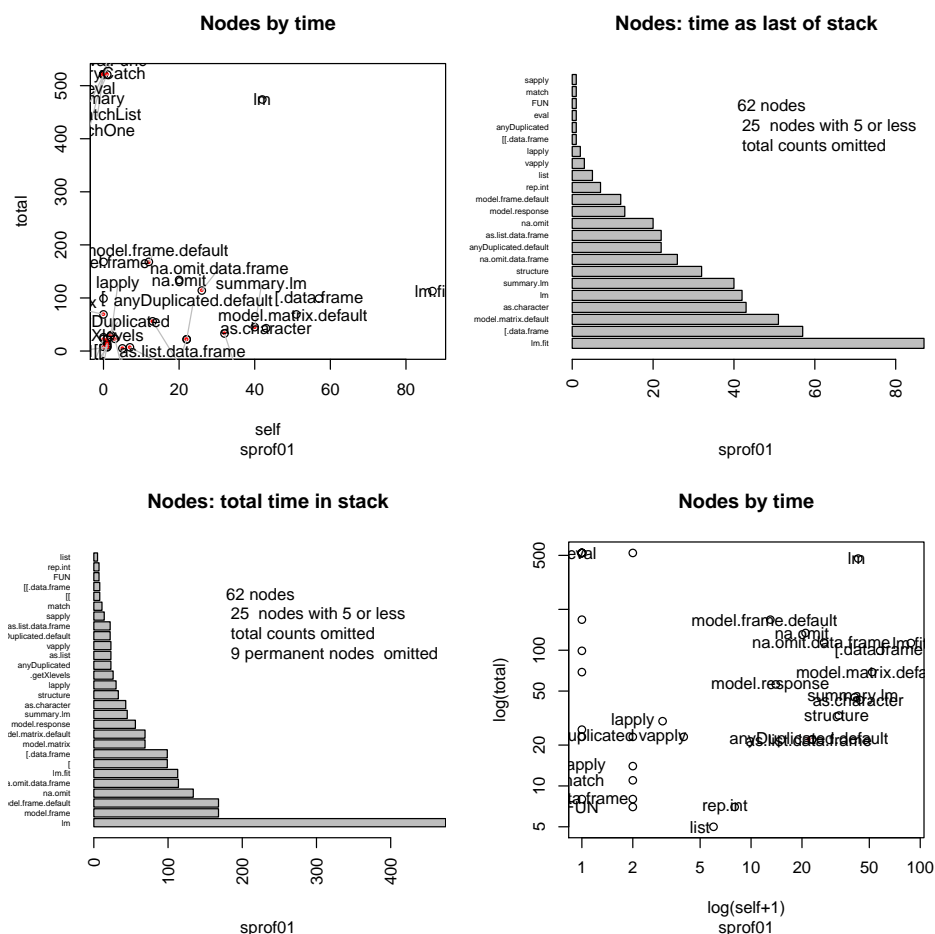
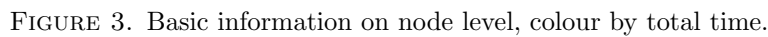


FIGURE 2. Basic information on node level - horizontal bars

```
plot_nodes(sprof01, col=freqrankcol01)
par(oldpar)
```

Colour is considered a volatile attribute. So you may need to pay some attention to keep colour indices (and colour palettes) aligned to your context. You may want to do experiments with colour, trying to find a good solution for your visual preferences. The recommended way is to use some stable colour index (the slot `icol` is reserved for this) and use this as an index to a choice of colour palettes. So `icol` becomes a part of the data structure, and the colour palette to be used is passed as a parameter. Package *RColorBrewer* may be a helpful source for colour palettes. You find more information and references on this in the vignette “Bertin matrices” of package *bertin*, <http://bertin.r-forge.r-project.org>.

ToDo: improve
colour: support
colour in a structure



- ToDo:** Move class attributes to package code
- ToDo:** add class by keyword

```
nodekeyword0 <- function(node)
{
}
```

```
nodepackages <- nodepackage(sprof01$nodes$name)
names(nodepackages) <- sprof01$nodes$name
table(nodepackages)
```

```
nodepackages
```

	base	stats	utils
<not found>	6	41	14
			1

```
sprof01$nodes$icol <- as.factor(nodepackages)
```

Nodes by package, colours by RColorBrewer: see fig. 4 on the next page.

```
x_apply <- c("apply", "lapply", "vapply", "sapply")
x_as <- c("as.list", "as.data.frame", "as.list.data.frame",
        "as.character", "as.list.default", "as.name")
```

(Extend as you need it) and then use, as for example:

```
nodeclass <- rep("x_nn", sprof01$info$nrnodes)
nodeclass[sprof01$nodes$name %in% x_apply] <- "x_apply"
nodeclass[sprof01$nodes$name %in% x_as] <- "x_as"
```

or use assignments on the fly

```
nodeclass[sprof01$nodes$name %in%
  c("eval", "evalFunc",
    "try", "tryCatch", "tryCatchList", "tryCatchOne",
    "doTryCatch", "withVisible")]
  ] <- "x_eval"
nodeclass[sprof01$nodes$name %in%
  c("model.frame", "model.matrix.default", "model.frame.default",
    "model.response", "model.matrix", "model.response")]
  ] <- "x_model"
nodeclass[sprof01$nodes$name %in%
  c("lm", "lm.fit", "summary.lm")]
  ] <- "x_lm"
nodeclass[sprof01$nodes$name == "<Anonymous>"] <- "x_Anon"
nodeclass[sprof01$nodes$name == "Sweave"] <- "x_Sweave"
nodeclass[sprof01$nodes$name %in% c("summary", "summary.lm")] <-
  "x_summary"
```

```
Input
```



```
sprof01$nodes$icol <- as.factor(nodeclass)
```

adds a sticky colour attribute. For interpretation, you should choose your preferred colour palette, for example

```
nodeclasscol <- c("red", "green", "blue", "yellow",
                  "cyan", "magenta", "purple",
                  "brown", "aquamarine")
```

ToDo: Defaults by class

```
# gold cyan4 aquamarine pink violet orchid hotpink salmon turquoise1
```

```
#8 12
oldpar <- par(mfrow=c(3,2))
plot_nodes(sprof01, which=1:6, col=nodeclasscol)
par(oldpar)
```

ToDo: classes need separate colour palette

Nodes by class, user defined colors: see fig. 5 on the facing page.

```
#8 12
oldpar <- par(mfrow=c(3,2))
plot_nodes(sprof01, which=1:6)
par(oldpar)
```

Nodes by class: default colour selection: see fig. 6 on page 22.

You can break down the frequency by classes of your choice. But beware of Simpson's paradox. The information you think you see may be strongly affected by your choices - what you see are reflections of conditional distributions. These may be very different from the global picture.

If package `wordcloud` is installed, a different view is possible. This is added in the plots above.

2. A BETTER GRIP ON PROFILE INFORMATION

The basic information provided by all profilers in R is a protocol of sampled stacks. The conventional approach is to break the information down to nodes and edges. The stacks provide more information than this. One way to access it is to use linking to pass information. This has already been used on the node level in section 1.1.2 on page 11.

ToDo: add attributes to stacks, and discuss scope

ToDo: sorting/ranging stacks

2.1. The internal details. For each recorded event, the protocol records one line with a text string showing the sampled stack (in reverse order: most recent first). The stack lines may be preceded by header lines with event specific information. The protocol may be interspersed with control information, such as information about the timing interval used. ?? on page ??.

We know that the structural information, static information as well as dynamic information, can be represented with the help of a graph. For a static analysis,



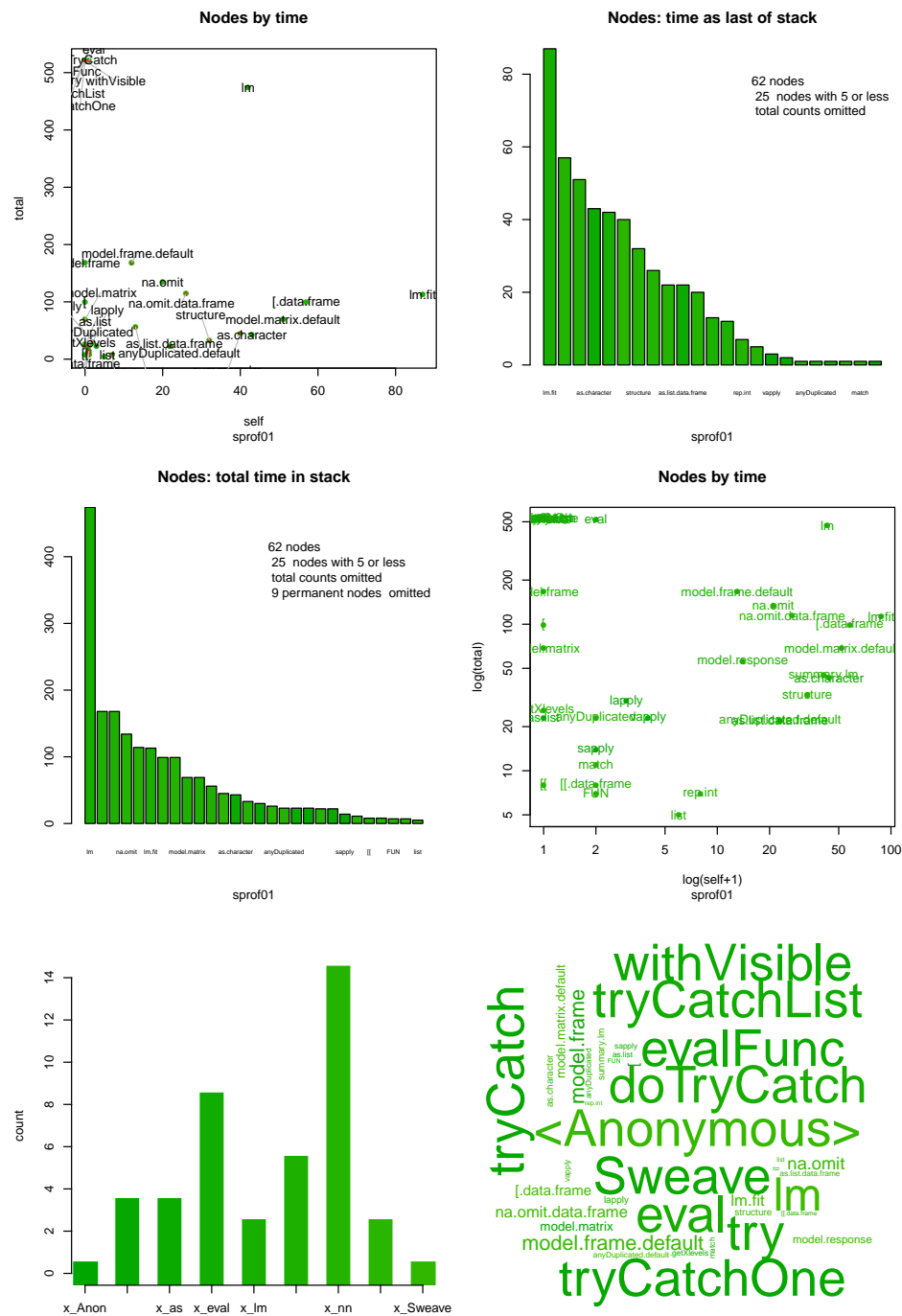


FIGURE 6. 8 12 Nodes by class: default colour selection

the graph representation may be the first choice. For a dynamic analysis, the stack information is our first information. A stack is a connected path in the program graph. If we start with nodes and edges, we lose information which is readily available in record of stacks.

As we know that we are working with stacks, we know that they have their peculiarities. Stacks tend to grow and shrink. Subsequent events will have extensions and shrinkages of stacks (if the recording is on a fine scale), or stack sharing common stumps (if the recording is on a coarser scale). We could exploit this information, but it does not seem worth the effort.

ToDo: re-think:
sort stacks

There have always been interrupts, and these show up in profiles. In R, there is a related problem: garbage collection (GC) may interfere and leave traces in the stack.

Stack information is first. The call graph is a second instance that is (re)constructed from the stack recording. The graph represents cumulated one-step information. Longer scale information contained in the stacks is lost in the graph.

Here is the way we represent the profile information:

The profile log file is sanitised:

- Control lines are extracted and recorded in a separate list.
- Head parts, if present, are extracted and recorded in a matrix that is kept line-aligned with the remainder
- Line content is standardised, for example by removing stray quotation marks etc.

After this, the sanitised lines are encoded as a vector of stacks, and references to this.

Note: after sanitising, stacks may have an empty node list. However they should not be removed as long as they may still be present in the profile.

If necessary, these steps are done by chunks to reduce memory load.

From the vector of stacks, a vector of nodes (or rather node names) is derived.

The stacks are now encoded by references to the nodes table. For convenience, we keep the (sanitised) textual representation of the stacks. (This may change.)

So far, texts are in reverse order. For each stack, we record the trailing leaf, and then we reverse order. The top of stack is now on first position.

Several statistics can be accumulated easily as a side effect.

Conceptually, the data structure consists of three tables (the implementation may differ, and is subject to change).

The profiles table is the representation of the input file. Control lines are collected in a special table. With the control lines removed, the rest is a table, one row per input line. The body of the line, the stack, is encoded as a reference to a stacks table (obligatory) and header information (optional).

The stacks table contains the collected stacks, each stack encoded as a list of references to the node table. This is obligatory. This list is kept in reverse order (root at position 1). A source line representing the stack information may be kept (optional).

The nodes table keeps the names at the nodes.

Sometimes, it is more convenient to use a simple representation, such as a matrix. Several extraction routines are provided for this, and the display routines make heavy use of this. See table 5.

ToDo: complete
matrix conversion

TABLE 5. Extraction and conversion routines

<code>profiles_matrix()</code>	incidence matrix: nodes by event
<code>stacks_matrix()</code>	incidence matrix: nodes by stack
<code>list.as.matrix()</code>	fill list to equal length and convert to matrix
<code>stackstoadj()</code>	stacks to (correspondence) adjacency matrix
<code>adjacency()</code>	sprof to (correspondence) adjacency matrix

We now can go beyond node level.

This is what we get for free from the node information on our three levels: node, stack, and profile.

ToDo: check and
stabilise colour linking

See fig. 7 on the facing page for a summary of nodes by stack and profile.

Input

```
#8 rainbow
sprof01$nodes$icol <- freqrank01; freqrankcol <- rainbow(62)
shownodes(sprof01, col=freqrankcol)
```

The obvious message is that if seen by stack level, there are different structures. Profiling usually takes place in a framework. So at the base of the stacks, we find entries that are (almost) persistent. Then usually we have some few steps where the algorithm splits, and then we have the finer details. These can be identified using information on the stack level, but of course they are not visible on the node or edge level in a graph representation. On the stack level, we see a socket. If we want a statistic, we can look at number of different nodes by level.

Input

```
stacks_nodes <- list.as.matrix(sprof01$stacks$nodes)
nrnodes <- apply(stacks_nodes,1,function(x) {length(unique(x))})
cat("nr unique nodes per stack level\n")
```

Output

```
nr unique nodes per stack level
```

Input

```
nrnodes
```

Output

```
[1] 1 1 2 2 2 2 2 2 2 2 2 2 4 11 12 10 10 16 9 8 6 8
[23] 3 2 2
```

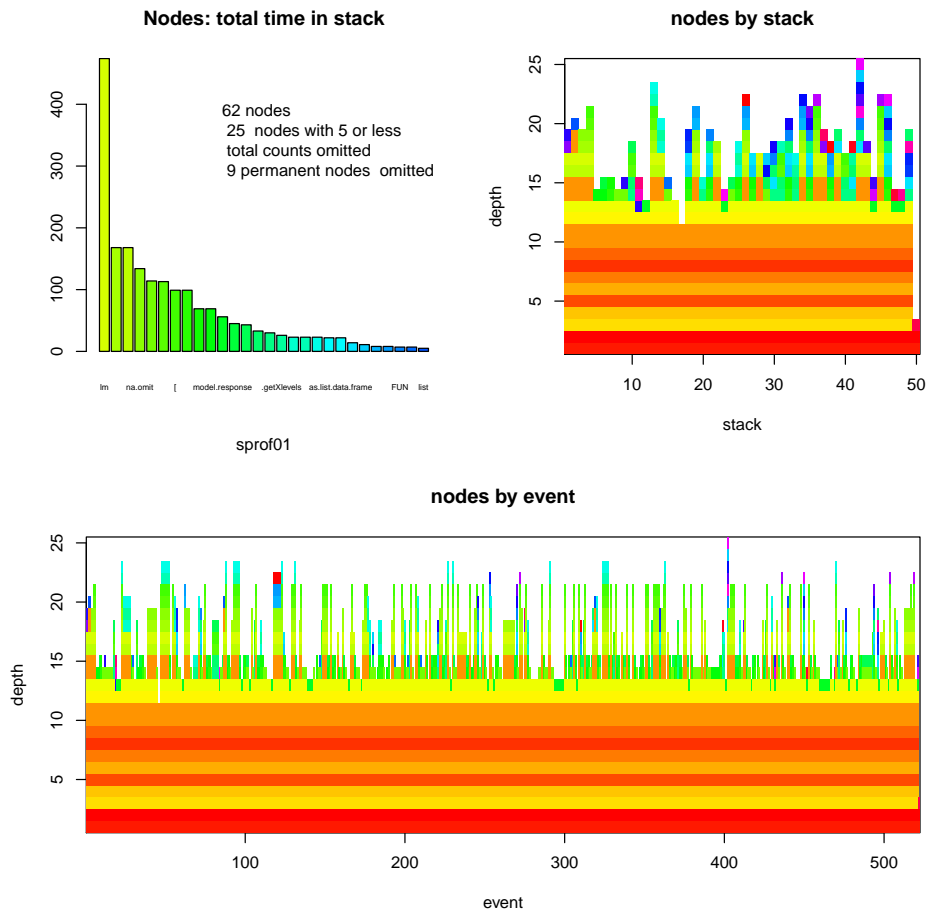



FIGURE 7. Nodes by stack and profile

Nr. of unique nodes by stack level: See fig. 8 on the next page.

```

Input
plot(x=nrnodes, y= 1:length(nrnodes),
      xlab="nr of unique nodes", ylab="stack level")
abline(h=2.5,col="green")
abline(h=12.5,col="green")

```

ToDo: check and synchronise

We will come to finer tools in section 2.4 on page 37 but for the moment the rough information should suffice to take a decision. In our example, it is only a matter of taste whether we cut off 12 levels, or we want to work with five components after cutting 13 levels three leaves us to start with five roots on the next level in our example.

Not so often, but a frequent phenomenon is to have some “burn in” or “fade out”. To identify this, we need to look at the profile level. The indicator to check is

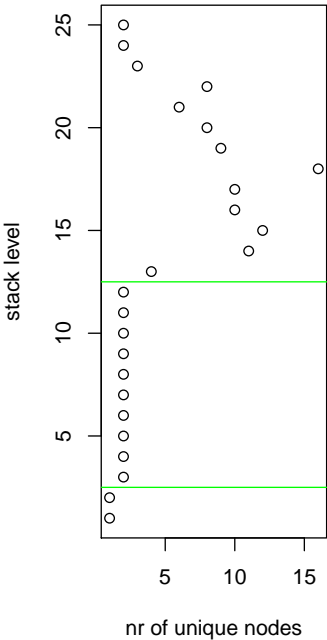


FIGURE 8. Nr of unique nodes by stack level.

to whether we have very low frequency stacks at the beginning or the end of our recording. The counts to be takes as reference can be seen from the summary.

```
summary(sprof01$stacks$refcount) Input
```

Output						
Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	
1.0	1.0	2.0	10.4	12.0	87.0	

ToDo: we could do smart smoothing of the stacks here

This summary has to be taken with caution. As the program runs, the stacks are build up und teared down, and we only take random samples. So in dynamic parts, we see images with some fluctuation, as one stack may be a snapshot of an other under construction. A better information is to cut off fluctuations and use this summary as a reference.

```
summary(sprof01$stacks$refcount[sprof01$stacks$refcount>2]) Input
```

Output						
Min.	1st Qu.	Median	Mean	3rd Qu.	Max.	
3.00	7.75	16.00	24.30	40.50	87.00	

Input

```
df <- data.frame(stack=sprof01$profiles$data,
  count=sprof01$stacks$refcount[sprof01$profiles$data])
prxt(df, caption="Stacks by event: burn in/fade out",
  label="tab:margin",
  digits=c(0,0,0) )
```

Table 6: Stacks by event: burn in/fade out

	stack	count
1	1	1
2	2	5
3	2	5
4	3	26
5	4	55
6	4	55
7	5	13
8	5	13
9	6	43
10	7	51
<cut>	:	:
513	3	26
514	3	26
515	3	26
516	3	26
517	4	55
518	3	26
519	36	2
520	16	42
521	44	2
522	50	1

Here at least one recording on either side is a candidate to be off. We may have a look at the next recordings and decide to go beyond and cut off events 1 : 3 and 519 : 522.

At a closer look, we may find stack patterns (maybe marked by specific nodes) that indicate administrative intervention and rather should be handled as separators between distinct profiles rather than as part of the general dynamics. Again we may use some indicator nodes to be used as marker for special stacks.

ToDo: example

Stable framework effects sometimes are obvious and can be detected automatically. “burn in” or “fade out” may need a closer look, and special stacks need and individual inspection on low frequency stacks. Tools for trimming are in section 2.3.1 on page 32.

ToDo: colours. re-colour. Propagate colour to graph.

2.2. The free lunch. What you have seen so far is what you get for free when using package *sprof*.

If you want to wrap up the information and look at it from a graph point of view, here is just one example. More are in section 3 on page 42 and `vrefsec:moregraph`. But before changing to the graph perspective, we recommend to see the next sections, not to skip them.

The preview, at this point, taking package *graph* as an example¹. *graph* on its side has an undocumented feature: it needs *Rgraphviz* to handle graph attributes². We have to take two steps. We extract the graph information from *sprof*. Using an adjacency matrix is a simple solution here. This is then converted to the "*graphNEL*" format which is shared by *graph* and *Rgraphviz*. *Rgraphviz* is hidden in the use of `plot()`. So here is a bare foot approach (Call graph derived from profile information: See fig. 9 on the next page.). A more sophisticated function implementation is in section 3 on page 42.

```
Input
```

```
#6 sprofadjNEL02
library(graph)
sprof01adjNEL <- as(adjacency(sprof01),"graphNEL")
plot(sprof01adjNEL, main="sprof01: graph layout example",
     sub=sprof01$info$id,
     attrs=list(node=list(cex=4, fontsize=40, shape="ellipse")),
     cex.main=2)
rm(sprof01adjNEL)
```

2.3. Cheap thrills. Before starting additional inspection, the data better be trimmed. Trimming routines are in section 2.3.1 on page 32, but the data structure is robust enough to allow manual intervention as used here.

ToDo: updateRprof needs careful checking. For now, we are including long listings here to provide the necessary information

```
Input
```

```
sprof02 <- sprof01; sprof02$info$id <- "sprof02: trimmed"
```

On the stack level, we take brute force to cut off the basic stacks.

```
Input
```

```
basetrim <- 13
sprof02$stacks$nodes <- sapply(sprof02$stacks$nodes,
                              function(x){if (length(x)> basetrim) x[-(1:basetrim)] })
```

We have noted burn in/fade out. This is on the profile level. Taking the big knife is not advisable, since time information and stack data must be synchronised. So we are more cautious, and instead of cutting off the stacks we replace them by an empty mark.

ToDo: should this be NA or NULL?

¹Package 'graph' was removed from the CRAN repository.

This package is now available from Bioconductor only.

See <http://www.bioconductor.org/packages/release/bioc/html/graph.html>.

²Package 'Rgraphviz' was removed from the CRAN repository.

This package is now available from Bioconductor only.

See <http://www.bioconductor.org/packages/release/bioc/html/Rgraphviz.html>.

sprof01: graph layout example

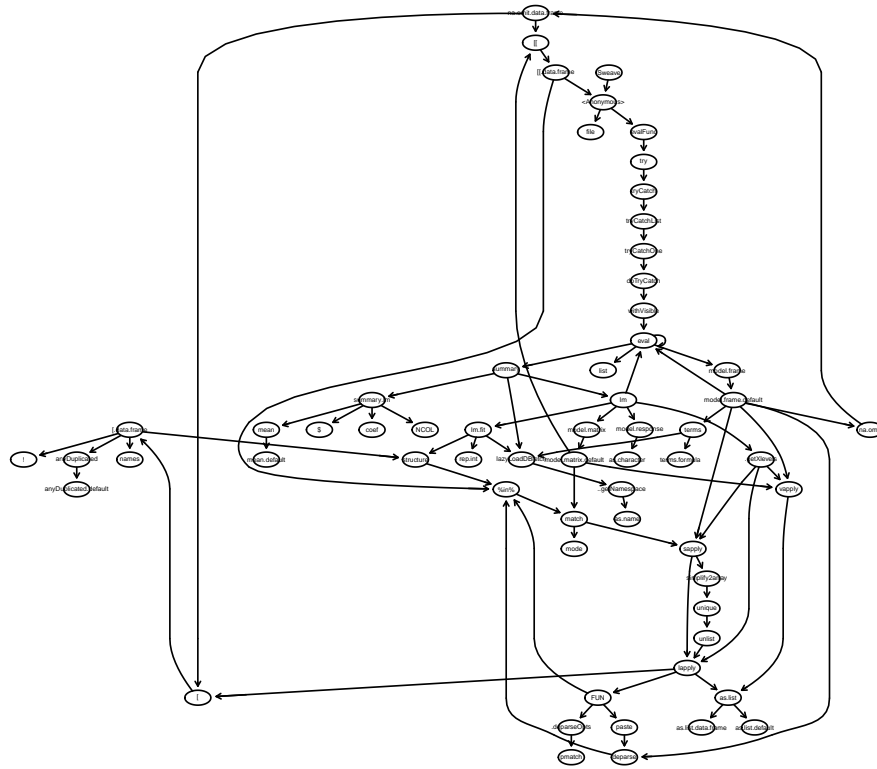


FIGURE 9. Call graph derived from profile information

sprof02\$profiles\$data[1:3] <- NA *Input*

sprof02\$profiles\$data[519:522] <- NA

At this point, it is a decision whether to adapt the timing information, or keep the original information. Since this decision does affect the structural information, it is not critical. But analysis is easier if unused nodes are eliminated. The *info* section is inconsistent at this point. Another reason to call *updateRprof()*.

ToDo: handle empty stacks and zero counts gracefully

ToDo: add a purge function

strx(sprof02\$info) *Input*

##strx: sprof02\$info

'data.frame': 1 obs. of 9 variables:

\$ id : chr "sprof02: trimmed"

```

$ date : POSIXct, format: "2013-08-24 22:15:44"
$ nrnodes : int 62
$ nrstacks : int 50
$ nrrecords : int 522
$ sample.interval: num 0.001
$ sampling.time : num 0.522
$ ctllines : chr "sample.interval=1000"
$ ctllinenr : num 1

```

```

prxt(sprof02$nodes,
     caption="sprof02, before update",
     label="tab:sprof02info1",
     digits=c(0,0,0,2,0,2,0),
     zero.print=" . "
)

```

Table 7: sprof02, before update

	name	self.time	self.pct	total.time	total.pct	icol
1	!	2	0.38	2	0.03	50
2	..getNamespace	.	0.00	1	0.01	55
3	.deparseOpts	2	0.38	4	0.05	43
4	.getXlevels	.	0.00	26	0.34	28
5	[.	0.00	99	1.29	19
6	[.data.frame	57	10.92	99	1.29	20
7	[[.	0.00	8	0.10	35
8	[[.data.frame	1	0.19	8	0.10	36
9	%in%	1	0.19	4	0.05	42
10	<Anonymous>	6	1.15	522	6.79	2
<cut>	\vdots	:	:	:	:	:
53	terms	.	0.00	2	0.03	48
54	terms.formula	1	0.19	1	0.01	59
55	try	.	0.00	521	6.78	10
56	tryCatch	.	0.00	521	6.78	5
57	tryCatchList	.	0.00	521	6.78	9
58	tryCatchOne	.	0.00	521	6.78	7
59	unique	3	0.57	4	0.05	41
60	unlist	.	0.00	1	0.01	56
61	vapply	3	0.57	23	0.30	31
62	withVisible	.	0.00	521	6.78	6

```

Input

```

```

sprof02 <- updateRprof(sprof02)
sprof02$info$id <- "sprof02 updated"

```

Input

```
strx(sprof02$info)
```

Output

```
##strx: sprof02$info
'data.frame':      1 obs. of  10 variables:
 $ id : chr "sprof02 updated"
 $ date : POSIXct, format: "2013-08-24 22:15:44"
 $ nrnodes : int 62
 $ nrstacks : int 50
 $ nrrecords : int 522
 $ sample.interval: num 0.001
 $ sampling.time : num 0.522
 $ ctllines : chr "sample.interval=1000"
 $ ctllinenr : num 1
 $ date_updated : POSIXct, format: "2013-08-24 22:15:49"
```

Input

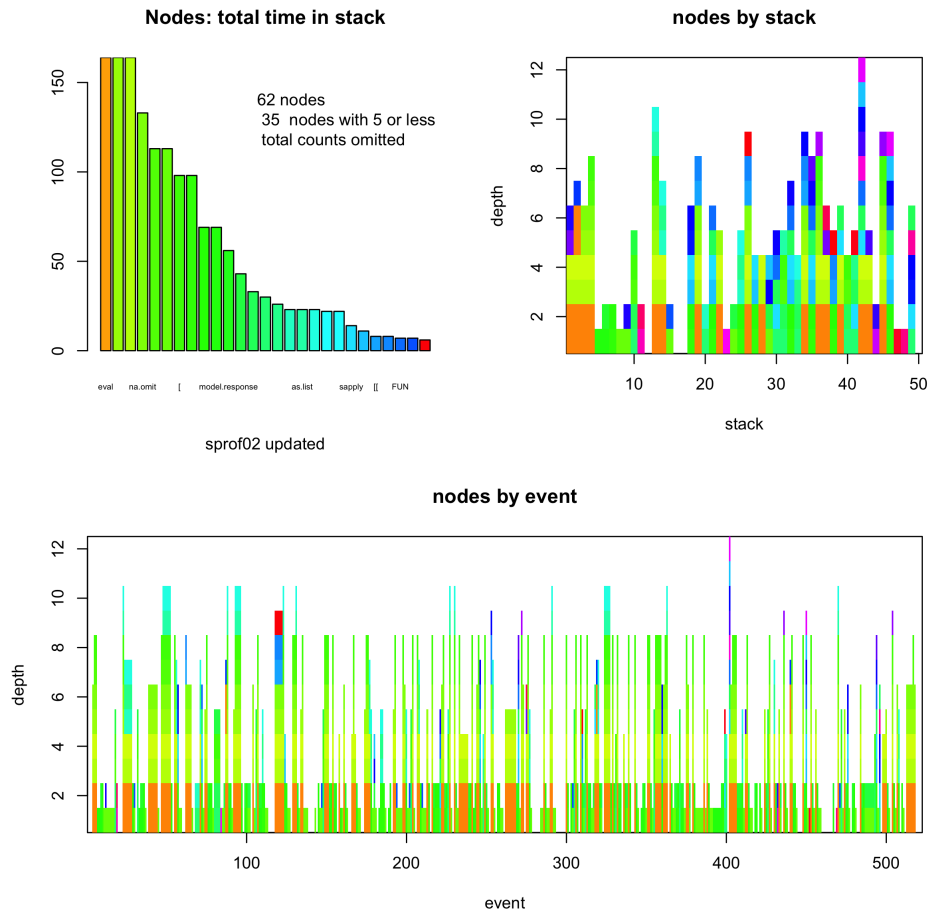
```
prxt(sprof02$nodes,
     caption="sprof02, after update",
     label="tab:sprof02info2",
     digits=c(0,0,0,2,0,2,0),
     zero.print=" . "
)
```

Table 8: sprof02, after update

	name	self.time	self.pct	total.time	total.pct	icol
1	!	1	0.23	1	0.06	50
2	..getNamespace	.	0.00	1	0.06	55
3	.deparseOpts	2	0.46	4	0.25	43
4	.getXlevels	.	0.00	26	1.64	28
5	[.	0.00	98	6.17	19
6	[.data.frame	57	13.16	98	6.17	20
7	[[.	0.00	8	0.50	35
8	[[.data.frame	1	0.23	8	0.50	36
9	%in%	1	0.23	4	0.25	42
10	<Anonymous>	6	1.39	6	0.38	2
<cut>	\vdots	:	:	:	:	:
53	terms	.	0.00	1	0.06	48
54	terms.formula	1	0.23	1	0.06	59
55	try	.	0.00	.	0.00	10
56	tryCatch	.	0.00	.	0.00	5
57	tryCatchList	.	0.00	.	0.00	9
58	tryCatchOne	.	0.00	.	0.00	7
59	unique	3	0.69	4	0.25	41
60	unlist	.	0.00	1	0.06	56
61	vapply	3	0.69	23	1.45	31
62	withVisible	.	0.00	.	0.00	6

Input

#8 8

shownodes(sprof02)FIGURE 10. `sprof02`: Nodes by stack and profile

Nodes by stack and profile: see fig. 10.

2.3.1. *Trimming*. Note: trimming may be supported by the graph packages. If you are more familiar with your graph package, you may prefer to handle trimming there.

To decide about trimming, we can use the displays shown above, or we can use some statistics. Global trimming works on the stack level. To make life easy, we can imbed the stack information in a matrix and use marginals.

Input

```
stackm01 <- list.as.matrix(sprof01$stacks$nodes)
nr_uniquenodes01 <- apply(stackm01,1, function(x) { length(unique(x))})
names( nr_uniquenodes01)<- rownames(nr_uniquenodes01,FALSE,"L")
nr_uniquenodes01
```

Output

L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13	L14	L15	L16	L17	L18
1	1	2	2	2	2	2	2	2	2	2	2	4	11	12	10	10	16
L19	L20	L21	L22	L23	L24	L25											
9	8	6	8	3	2	2											

Input

```
rm(stackm01, nr_uniquenodes01)
```

ToDo: This section needs to be reworked

Input

```
rsprof01Tr <- trimstacks(sprof01, level=11)
```

After trimming, it is worth to inspect the result and to see whether additional trimming is helpful. Function `roots_sprof` is handy.

Input

```
roots_sprof(sprof01,stacks=rsprof01Tr)
```

Output

summary	<NA>
50	NA

Here after trimming we have one root node. This may be esthetically pleasing, but it is more informative to note that `summary` is a common root, and then cut it off (thus fragmenting the graph into components).

Input

```
rsprof02Tr <- trimstacks(sprof01, level=12)
roots_sprof(sprof01,stacks=rsprof02Tr)
```

Output

lm	lazyLoadDBfetch	summary.lm	<NA>
29	27	51	NA

Now this is revealing. We know how our test example was constructed. So we are prepared to find `lm` and `summary.lm` at prominent positions. But it is surprising to find `lazyLoadDBfetch` as dominant node, and with a frequency comparable to that of `lm`.

ToDo: move to other section

There is no statistics on profiles. Profiles are our elementary data. However we can link to our derived data to get a more informative display. For example, going one step back we can encode stacks and use these colour codes in the display of a profile.

Or going two steps back, we can encode nodes in colour, giving coloured stacks, and use these in the display of profile data.

2.3.2. *Surgery*. Note: surgery may be supported by the graph packages. Use the implementation you are more familiar with. Surgery can also be done conveniently on the source level, using simple replace statements.

Looking at nodes gives you a point-wise horizon. Looking at edges gives you a one step horizon. The stacks give a wider horizon, typically a step size of 10 or more. The stacks we get from R have peculiarities, and we can handle with this broader perspective. These are not relevant if we look point-wise, but may become dominating if we try to get a global picture. We take a look ahead (details to come in section 3 on page 429 and have a preview how our example is represented as a graph. Left is the original graph as recovered from the edge information, right the graph after we have cut off the scaffold effects.

ToDo: cut next level

Control structures may be represented in R as function, and these may lead to concentration points. Using information from the stacks, we can avoid these by introducing substitute nodes on the stack level. For example, `lapply` is appearing in various contexts and may be confusing any graph representation. We can avoid this by replacing a short sequence.

```
"[" "lapply" ".getXlevels" -> "<.getXlevels_>"
```

Other candidates are:

```
"as.list" "vapply" "model.frame.default" -> "<model_as.list>"
```

or

```
"as.list" "vapply" "model.matrix.default" -> "<model_matrix_as.list>"
```

If the node does not exist, we want to add it to our global variable. For now, we do it using expressions on the R basic level and avoid tricks like simulating “call by reference”.

ToDo: Implement. Currently best handled on source=text level

ToDo: function `addnode` using “call by reference” to be added

```

                                Input
addnode <- function(nodes, newnode, warn = options("warn"))
{
  i <- match(newnode, nodes$name, nomatch=0)
  if (i==0){
    nodes$name <- as.character(nodes$name)
    nodes <- rbind(nodes,NA)
    i <- length(nodes$name)
    nodes$name[i] <- newnode
    rownames(nodes) <- nodes$name
    if (as.logical(warn))
      message("addnode: node added. An updateRprof() may be necessary.")
  }
  return(nodes)
}

```

```

                                Input
sprof03 <- sprof02; sprof03$info$id <- "sprof03: surgery"
nodes <- addnode(sprof03$nodes, "<.getXlevels_>", warn=FALSE)
nodes <- addnode(nodes, "<model_as.list>", warn=FALSE)
sprof03$nodes <- addnode(nodes, "<model_matrix_as.list>", warn=FALSE)

```

ToDo: xreplacenodes: improve implement

ToDo: clean up factor handling

So far, we use factor indices only.

```

xwhere <- function(oldseq, x){
  x <- unlist(x)
  firstpos <- 1; lastpos <- length(x) - length(oldseq) +1

  if (lastpos <1) return(0)
  l <- length(oldseq)-1
  while (firstpos <= lastpos) {
    if ( isTRUE (
      all.equal(oldseq , x[firstpos:(firstpos+1)], check.attributes=FALSE)
    ) )
      {return(firstpos) }
    firstpos <- firstpos+1
  }
  return(0)
}

```

```

xreplace <- function(oldseq, newseq, x){
  #! handle multiple replacements
  wh <- xwhere(oldseq,x)
  if (wh) {
    l <- length(oldseq)-1
    if (wh>1) x1 <- c(x[1:(wh-1)], newseq) else x1 <- newseq
    if (wh+1 < length(x)) x <- c(x1, x[ -(1:(wh + 1))]) else x <- x1
    return(x)
  } else return(x)
}

```

```

nodenames2index <- function(names, sprof){
  if (is.character(names))
    { sapply(names, function(x) {match(x,sprof$nodes$name)})
      #if (is.na(trimnode)) return(ts)
    } else names
}

```

```

xreplacenodes <- function(sprof, oldseq, newseq)
{
  if (is.character(oldseq)) oldseq <- nodenames2index(oldseq,sprof)
  if (is.character(newseq)) newseq <- nodenames2index(newseq,sprof)
  stacks <- sprof$stacks$nodes
  sapply(stacks, function(x){xreplace(oldseq, newseq, unlist(x))})
}

```

```

xfactor <- function(x, from) {
  if (is.factor(from)) {
    levels <- levels(from)
    labels <- labels(from)
    ordered <- is.ordered(from)
  }
}

```

ToDo: stacksrc,
collstacksdictrv etc.
now out of date

```

    } else if (is.character(from)) {
      levels <- 1: length(from)
      labels <- from
      ordered = is.ordered(from)
    }

    if (is.list(x)) {
      sapply(x, factor,
             levels,
             labels)
    } else {
      factor(x,
             levels=1: length(from),
             labels=from)
    }
  }
}

```

```

#xfactor(sprof03$stacks$nodes, sprof03$nodes$name)

```

```

# Input
sprof03$stacks$nodes <- xreplacenodes(sprof03,
  c(".getXlevels", "lapply", "["),
  "<.getXlevels_>")
sprof03$stacks$nodes <- xreplacenodes(sprof03,
  c("model.frame.default", "vapply", "as.list"),
  "<model_as.list>")
sprof03$stacks$nodes <- xreplacenodes(sprof03,
  c("model.matrix.default", "vapply", "as.list"),
  "<model_matrix_as.list>")
sprof03 <- updateRprof(sprof03)

```

```

# Input
#xfactor(sprof03$stacks$nodes, sprof03)

```

ToDo: warn about undefined vars, e.g. `rl`, `class`, ...

ToDo: replace `sprof03` by

ToDo: fix null name

This surgery gives no additional information on the profile data. It only helps the graph representation, by extending the one-step information given by the edges to two or omre step information at some critical points.

We use a prepared sanitised version of our data set.

```

# Input
sprof04 <- readRprof("RprofsRegressionExpl03.out", id="sprof04")

```

Applying our old classification:

```

# Input
nodeclass <- rep("x_nn", length(sprof04$nodes$name))
nodeclass[sprof04$nodes$name %in% x_apply] <- "x_apply"
nodeclass[sprof04$nodes$name %in% x_as] <- "x_as"
nodeclass[sprof04$nodes$name %in%
  c("eval", "evalFunc",

```

```

      "try", "tryCatch", "tryCatchList", "tryCatchOne",
      "doTryCatch", "withVisible")
    ] <- "x_eval"
  nodeclass[sprof04$nodes$name %in%
    c("model.frame", "model.matrix.default", "model.frame.default",
      "model.response", "model.matrix", "model.response")
    ] <- "x_model"
  nodeclass[sprof04$nodes$name %in%
    c("lm", "lm.fit", "summary.lm")
    ] <- "x_lm"
  nodeclass[sprof04$nodes$name == "<Anonymous>"] <- "x_Anon"
  nodeclass[sprof04$nodes$name == "Sweave"] <- "x_Sweave"
  nodeclass[sprof04$nodes$name %in% c("summary", "summary.lm")] <-
    "x_summary"
  sprof04$nodes$icol <- as.factor(nodeclass)

```

now gives

Input

```

#8 12
oldpar <- par(mfrow=c(3,2))
plot_nodes(sprof04, which=1:6, col=nodeclasscol)
par(oldpar)

```

Sanitised nodes by class, user defined colours: see fig. 11 on the following page.

2.4. Run length. For a visual inspection, runs of the same node and level in the profile are easily perceived. For an analytical inspection, we have to reconstruct the runs from the data. In stacks, runs are organised hierarchically. On the root level, runs are just ordinary runs. On the next levels, runs have to be defined given (within) the previous runs. So we need `rrle()`, a recursive version of `rle`, applied to the profile information. This gives a detailed information about the presence time of each node, by stack level.

ToDo: use `sprof02`
or `sprof03`?

Input

```

profile_nodes <- profiles_matrix(sprof02)
profile_nodes_rle <- rrle(profile_nodes, collapseNA=FALSE)
#!NA needs special case in run length handling.

strx(profile_nodes_rle, list.len=5)

```

Output

```

##strx: profile_nodes_rle
List of 12
 $ :List of 2
  ..$ lengths: int [1:365] 1 1 1 3 3 1 7 1 ...
  ..$ values : int [1:365] NA NA NA 22 39 37 30 4 ...
  ..- attr(*, "class")= chr "rle"
 $ :List of 2
  ..$ lengths: int [1:411] 1 1 1 3 1 1 1 1 ...
  ..$ values : int [1:411] NA NA NA 22 NA NA 14 38 ...
  ..- attr(*, "class")= chr "rle"
 $ :List of 2

```

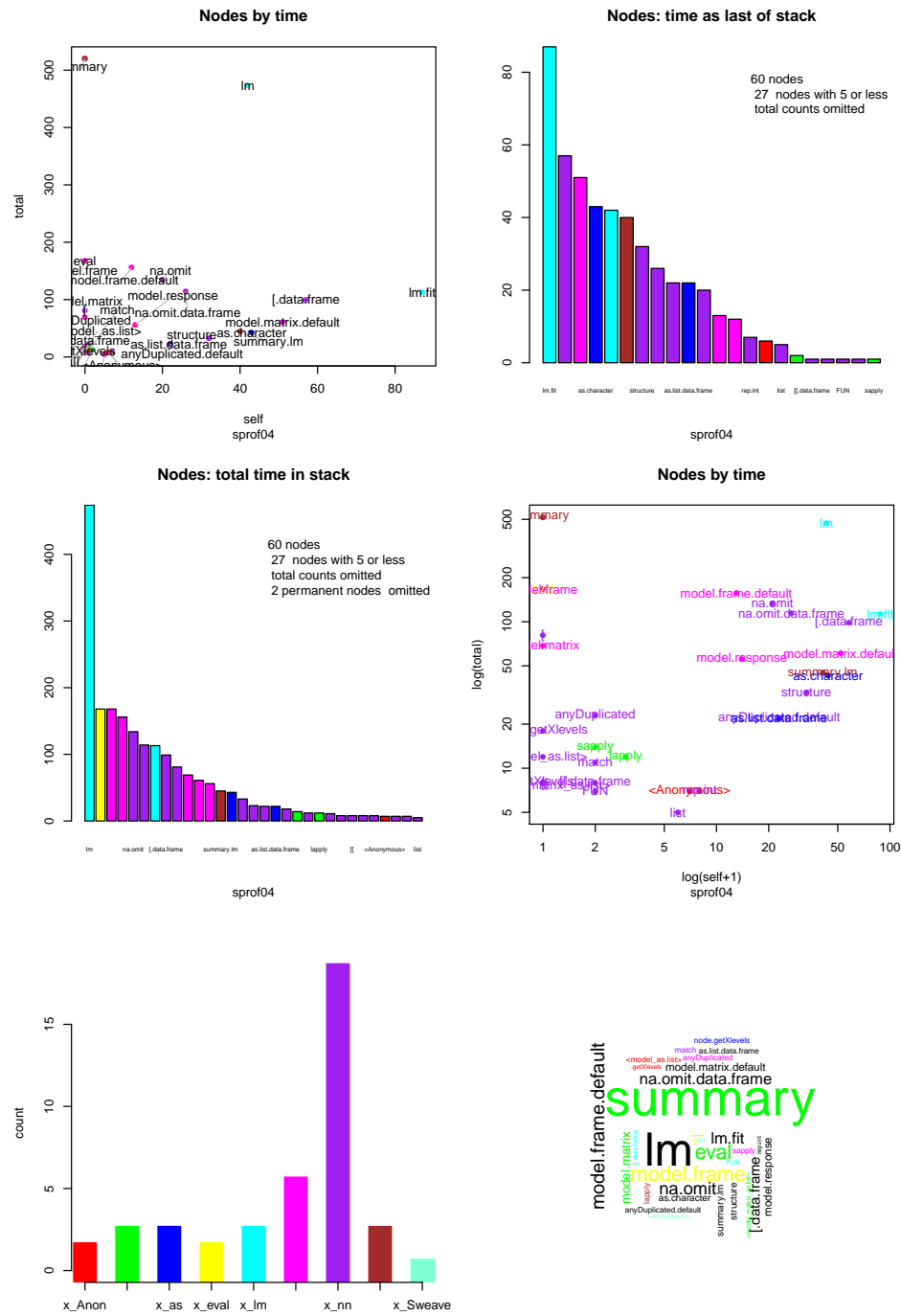


FIGURE 11. Sanitised. Nodes by class. User defined colours.

```

..$ lengths: int [1:431] 1 1 1 3 1 1 1 1 ...
..$ values : int [1:431] NA NA NA 35 NA NA NA NA ...
..- attr(*, "class")= chr "rle"
$ :List of 2
..$ lengths: int [1:431] 1 1 1 3 1 1 1 1 ...
..$ values : int [1:431] NA NA NA 36 NA NA NA NA ...
..- attr(*, "class")= chr "rle"
$ :List of 2
..$ lengths: int [1:452] 1 1 1 3 1 1 1 1 ...
..$ values : int [1:452] NA NA NA 40 NA NA NA NA ...
..- attr(*, "class")= chr "rle"
[list output truncated]

```

Input

On a given stack level, the run length is the best information on the time used per call, and the run count of a node is the best information on the number of calls. So this is a prime starting point for in-depth analysis.

If you need it, you can represent the run length information by level as a matrix. This is expanding a sparse matrix to full and should be avoided.

```

profile_nodes_rlearray <- nodesprofile(sprof02)
strx(profile_nodes_rlearray)

```

Output

```

##strx: profile_nodes_rlearray
num [1:61, 1:12, 1:7] 0 1 0 17 0 0 0 0 ...
- attr(*, "dimnames")=List of 3
..$ node : chr [1:61] "!" "..getNamespace" ".deparseOpts" ...
..$ level : chr [1:12] "1" "2" "3" ...
..$ run_length: chr [1:7] "1" "2" "3" ...

```

This allows us to extract marginal from `provlev[node, level, run length]`.

```

nn <- profile_nodes_rlearray["model.frame", , ]
print.table(addmargins(nn), zero.print = ".")

```

Output

	run_length							
level	1	2	3	4	5	6	7	Sum
1
2
3	40	17	7	4	2	6	1	77
4
5
6
7
8
9
10

ToDo: keep as factor. This is a sparse cube with margins node, stack level, run length. Nodes are mostly concentrated on few levels.

ToDo: Warning: data structure still under discussion

ToDo: hack. replace by decent vector/array based implementation

ToDo: add summary for NA

ToDo: add marginals and conditionals.

Provide function `node_summary`.

11
12
Sum	40	17	7	4	2	6	1	77

```
amt <- nodesrunlength(sprof02)
prxt(amt,
      caption=paste0("Marginal statistics on nodes by run length, ",
                      "sorted by total time used"),
      label="tab:pramt4",
      digits=c(rep(0,dim(amt)[2]) ,2),
      zero.print=" . ") #dim(amt)[2]-1, +1 for rownames
```

Table 9: Marginal statistics on nodes by run length, sorted by total time used

	1	2	3	4	5	6	7	nr_runs	total_time	avg_time
eval	86	34	14	8	4	12	2	160	334	2.09
model.frame	40	17	7	4	2	6	1	77	164	2.13
model.frame.default	40	17	7	4	2	6	1	77	164	2.13
na.omit	46	10	5	4	1	4	1	71	133	1.87
lm.fit	46	18	3	1	1	1	1	71	113	1.59
na.omit.data.frame	43	7	4	5	.	4	.	63	113	1.79
{}	59	4	3	4	.	1	.	71	98	1.38
{().data.frame	59	4	3	4	.	1	.	71	98	1.38
model.matrix	55	4	2	61	69	1.13
model.matrix.default	55	4	2	61	69	1.13
model.response	35	3	3	.	.	1	.	42	56	1.33
as.character	34	3	1	38	43	1.13
structure	21	1	.	1	.	1	.	24	33	1.38
lapply	26	.	.	1	.	.	.	27	30	1.11
.getXlevels	17	1	1	1	.	.	.	20	26	1.30
anyDuplicated	10	.	.	2	1	.	.	13	23	1.77
as.list	16	1	.	.	1	.	.	18	23	1.28
vapply	13	1	1	.	1	.	.	16	23	1.44
anyDuplicated.default	9	.	.	2	1	.	.	12	22	1.83
as.list.data.frame	15	1	.	.	1	.	.	17	22	1.29
sapply	14	14	14	1.00
match	12	12	12	1.00
{(){}()	3	.	.	.	1	.	.	4	8	2.00
{(){}().data.frame	3	.	.	.	1	.	.	4	8	2.00
FUN	7	7	7	1.00
rep.int	7	7	7	1.00
<Anonymous>	1	.	.	.	1	.	.	2	6	3.00
.deparseOpts	4	4	4	1.00
%in%	4	4	4	1.00
simplify2array	4	4	4	1.00
unique	4	4	4	1.00
list	3	3	3	1.00
deparse	2	2	2	1.00

mode	2	2	2	1.00
names	2	2	2	1.00
pmatch	2	2	2	1.00
!	1	1	1	1.00
..getNamespace	1	1	1	1.00
\$	1	1	1	1.00
as.list.default	1	1	1	1.00
as.name	1	1	1	1.00
coef	1	1	1	1.00
lazyLoadDBfetch	1	1	1	1.00
mean	1	1	1	1.00
mean.default	1	1	1	1.00
NCOL	1	1	1	1.00
paste	1	1	1	1.00
terms	1	1	1	1.00
terms.formula	1	1	1	1.00
unlist	1	1	1	1.00

See table 9: Marginal statistics on nodes by run length.

We can add this information to our base and use it to enhance graphical displays. See: nodes marked by run length and run count, fig. 16 on page 50.

ToDo: hack. keep length in nodesrunlength

`eval` is the base of all evaluation in R, so we should not be surprised to find it at the top of the list. The next two entries, `model.frame` and `model.frame.default` are seen 77 times. We know that in our example we had a loop with 100 repetitions, so a frequency of 100 would not be surprising. A closer look shows that both functions occur as isolated events 40 times, but frequency decreases with run length until we see another high at run length 6. We are sampling, and of course sampling can miss some function calls. But this pattern is the oopposite. This is what if occurs if we do not miss a function call, but we miss a gap. So several function calls are joined and appear as some longer runs, typically a multiple of the original run length.

We can improve the hit rate by increasing the sampling rate. But of course this is at the expense of using more space for the log files, and increased time for the overhead. In our case, 1 ms seems to be a good compromise, but 0.1 ms might be another feasible choice.

As we walk down the list, `na.omit` is next, followed closely by `na.omit.data.frame`. In our problem, there are no missing data. But R does not have something like a “vanilla mode”. The overhead used for the handling of potentially missing values would need a restructuring of the linear model algorithm, e.g. by introducing a “has.na” flag.

Walking further down, we see other candidates such as `as.character` that may be avoidable in this problem.

```

Input
nr <- nodesrunlength(sprof02, clean=FALSE)
## nr <- rbind(nr,0) ## hack. keep length in nodesrunlength

```

ToDo: table: node
#runs min median
run length max

```
rownames(nr) <- sprof02$nodes$name
sprof02$nodes <- cbind(sprof02$nodes, nr)
rownames(sprof02$nodes) <- sprof02$nodes$name
```

3. GRAPH PACKAGE

What we have achieved so far can be seen from the graph representations. For our purposes, an edge table is most convenient. To allow for edge attributes, we can use an R `data.frame` as provided by

We can make use of any graph mapping package. Unfortunately, each seem to have its own calling convention. So we have to do some translation.

Input

```
#12 6
library(graph)
oldpar <- par(mfrow=c(1,2))
plotviz(sprof01)
plotviz(sprof02)
par(oldpar)
```

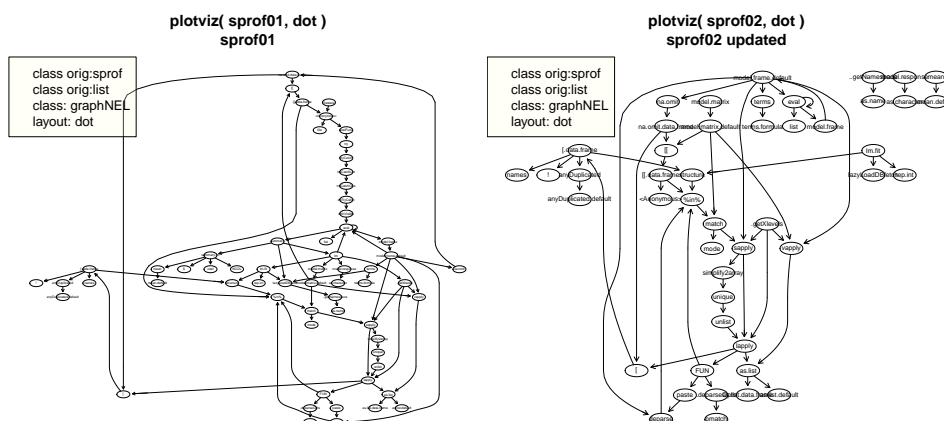


FIGURE 12. Sprof graph, before and after trimming.

See fig. 12 for a comparison before and after trimming. The scaffold effect are removed from the picture on the right side. This cuts off the uninformative spine, and induces minor changes in the body of the graph. You can do additional trimming, if you want.

R is function based, and control structures in general are implemented as functions. In a graph representation, they appear as nodes, concentrating and seeding to unrelated paths. We can detect these on the stack level and replace them by surrogates, introducing new nodes. This is a case for surgery.

We use a prepared sanitised version of our data set. (Sprof graph, after trimming, and after trimming and some sanitising, see fig. 13.)

ToDo: fix null name

Input

```
#12 6
sprof04 <- readRprof("RprofsRegressionExpl03.out", id="sprof04")
oldpar <- par(mfrow=c(1,2))
plotviz(sprof02)
plotviz(sprof04)
par(oldpar)
```

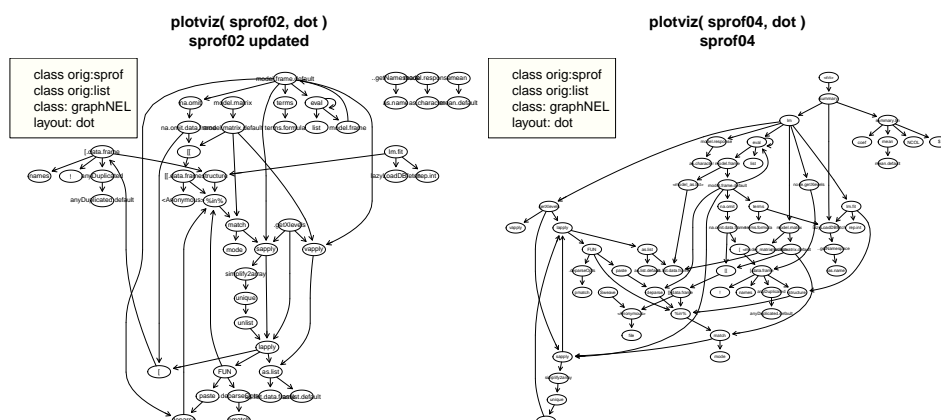


FIGURE 13. Sprof graph, after trimming, and after trimming and some sanitising.

ToDo: cut top levels

Now the structure becomes obvious. Cutting off may be taken two levels deeper. This would completely separate the `lm()` branch from the `summary.lm()`. In the `lm()` branch, there are tree nodes (`lapply()`, `%in%` and `[.data.frame]`) that are a cases for additional surgery to avoid focusing affects in the graph display. It is your choice to remove them, or live with them.

We know ho to create standard graph displays from this. The next step is to encode additional information we have from the profiles as attribute to the graph.

The derived edge frequency is the first bit of information. Implicitly, it can be used as weight in the graph placement routines. We make this explicit by giving a choice whether to use it or not. Irrespective of this choice, we encode reference counts as line width of the edges.

The functions in this chapter are not included in the package to avoid dependency of `sprof` on `graph` and other graph packages.

```
library(graph)
library(Rgraphviz)
```

This is a common routine for the `graph` and `Rgraphviz` package.

```

as_graphNEL_sprof <- function(sprof, Inputweight=TRUE){

  a04<-adjacency(sprof)

  rnames <- rownames(a04)

  if (!weight) {
    dimold <- dim(a04); a04 <- as.numeric(a04); dim(a04) <- dimold
    rownames(a04)<- rnames; colnames(a04)<- rnames;
  } #! define lwd first

  el04 <- edgedf(a04)
  el04$lwd <- rindex(el04$count, maxindex=6, ties.method="min")

  a04NEL <- as(a04,"graphNEL")
  nodeDataDefaults(a04NEL, "shape") <- "ellipse"
  nodeDataDefaults(a04NEL, "cex") <- 0.6
  nodeDataDefaults(a04NEL, "weight") <- 1
  nodeDataDefaults(a04NEL, "fill") <- "green"
  nodeDataDefaults(a04NEL, "col") <- "yellow"

  a04NEL <- layoutGraph(a04NEL)

  nodeRenderInfo(a04NEL) <- list(shape="ellipse")
  nodeRenderInfo(a04NEL) <- list(cex=0.6, shape="ellipse")
  nodeRenderInfo(a04NEL) <- list(weight=1)
  #nodeRenderInfo(a04NEL) <- list(color="yellow")
  nodeRenderInfo(a04NEL) <- list(fill="yellow", col="blue")

  edgeDataDefaults(a04NEL,"lwd") <- 1
  edgeDataDefaults(a04NEL,"col") <- "grey"

  #nodeRenderInfo(a04NEL) <- list(weight=1)

  #edgeRenderInfo(a04NEL) <- list(lwd=el04$lwd)
  #edgeRenderInfo(a04NEL)$lwd <- el04$lwd
  for (i in 1:length(el04$lwd))
  {edgeRenderInfo(a04NEL)$lwd[i] <- el04$lwd[i]}
  a04NEL
}

```

```

Input
plot_graphNEL_sprof <- function(sprof04,
#   mode="dot", nodeattrs = c("default", "time", "runs"),
  layout = "dot", nodeattrs,
  fill_list = NULL,
  main = NULL, sub = NULL,...)
{
  main1 <- deparse(substitute(sprof04))
  xsubid <- NULL

```

```

y<-NULL
class1 <- NULL

if (missing(nodeattrs)) nodeattrs <- "default"
#nodeattrs <- match.arg(nodeattrs)
# cat(nodeattrs)

if (inherits(sprof04, "sprof")) {
  class1 <- paste0("class orig:", class(sprof04))
  main1 <- sprof04$info$id
  graphNEL <-
    as_graphNEL_sprof(sprof04, weight=FALSE)
  y <- layout
} else graphNEL <- sprof04

if (!is.null(sub)) sub <- as.character(sub)

if (is.null(main)) {
  main <-paste0("plot_graphNEL_sprof( ",
    main1 , " , ",
    layout, " )\n",
    xsubid) } else
{
  main=paste0(main, "\n plot_graphNEL_sprof( ",
    main1 , " , ",
    layout, " )\n", xsubid)
}

# functions
legattributes <- function(legnd){
  legend("bottomright",
    legend= legnd,
    bg="#FFFFE040",
    seg.len=0,
    bty="n",
    text.font=3)
  #bg="#00004040",
}

#nDD0 <- (nodeRenderInfo(graphNEL))
#graphNEL <- layoutGraph(graphNEL)
#nDD1 <- (nodeRenderInfo(graphNEL))# fill & col corrupted

#nodeDataDefaults(graphNEL, "cex") <- 1.0
#nodeDataDefaults(graphNEL, "weight") <- 1
#nodeDataDefaults(graphNEL, "fill") <- "green"
#nodeDataDefaults(graphNEL, "col") <- "yellow"

#nodeDataDefaults(graphNEL, "shape") <- "ellipse"
nDD2 <- (nodeRenderInfo(graphNEL))

graphNEL <- layoutGraph(graphNEL)

nodeDataDefaults(graphNEL, "shape") <- "ellipse"

```

```

if (nodeattrs == "runs"){
  amt04 <- nodesrunlength(sprof04, clean=FALSE)

  # node attributes
  #sprof04$nodes$self.time -> fill
  fill_list <- heat.colors(12)[
    rkindex(-amt04[, "avg_time"],
      pwr=0.5, maxindex=12, ties.method="min")]
  names(fill_list) <- sprof04$nodes$name

  #sprof04$nodes$total.time -> lwd
  lwd_list <- rkindex(amt04[, "nr_runs"],
    pwr=0.5, maxindex=6, ties.method="min")
  names(lwd_list) <- sprof04$nodes$name

  #strx(nodeRenderInfo(graphNEL))
  nDD3 <- (nodeRenderInfo(graphNEL))

  nodeDataDefaults(graphNEL, "shape") <- "ellipse"

  nodeRenderInfo(graphNEL) <- list(lwd=lwd_list,
    fill=fill_list,
    col="#0000FF80",
    shape="ellipse",
    weight=1)
  #strx(nodeRenderInfo(graphNEL))
} else if (nodeattrs == "time"){
  # node attributes
  #sprof04$nodes$self.time -> fill
  fill_list <- heat.colors(12)[
    rkindex(-sprof04$nodes$self.time,
      pwr=0.5, maxindex=12, ties.method="min")]
  names(fill_list) <- sprof04$nodes$name

  #sprof04$nodes$total.time -> lwd
  lwd_list <- rkindex(sprof04$nodes$total.time,
    pwr=0.5, maxindex=6, ties.method="min")
  names(lwd_list) <- sprof04$nodes$name

  #strx(nodeRenderInfo(graphNEL))
  nDD3 <- (nodeRenderInfo(graphNEL))

} else {
  graphNEL <- layoutGraph(graphNEL)
  fill_list <- NULL
  lwd_list <- NULL
}

nodeDataDefaults(graphNEL, "shape") <- "ellipse"

nodeRenderInfo(graphNEL) <- list(
  lwd=lwd_list,
  fill=fill_list,

```

```

        col="#0000FF80",
        shape="ellipse",
        weight=1)

# edge attributes
# strx(edgeRenderInfo(graphNEL))
nER01 <- edgeRenderInfo(graphNEL)
edgeRenderInfo(graphNEL) <- list(col= "#80808080")
# strx(edgeRenderInfo(graphNEL))

renderGraph(graphNEL)

legend("topleft",
      legend=c( class1, paste0("class: ", class(graphNEL)),
        paste0("layout: ", layout)),
      bg="#FFFFE040",
      seg.len=0
      )#"lightyellow" =#FFFFE0

if (nodeattrs == "runs"){
  title(main=paste0("nodes (run length)\n", main)      )
  legattributes( c("node color: avg time, pwr=0.5",
    "node lwd: nr runs, pwr=0.5",
    "edge lwd: frequency"))
} else if (nodeattrs == "time"){
  title(main=paste0("nodes (time)\n", main))
  legattributes( c("node color: self.time, pwr=0.5",
    "node lwd: total.time, pwr=0.5",
    "edge lwd: frequency"))
} else {
  title(main=main)
  legattributes( c("#node color: ??",
    #"node lwd: ??",
    "edge lwd: frequency"))
}
title( sub=sprof04$info$id, col.sub=grey(0.5))
invisible(graphNEL)
}# %: plot_graphNEL_sprof(sprof02)

```

`plot_graphNEL_sprof(sprof04)` *Input*

Rgraphviz/graph basic plot: see fig. 14 on the next page.

To use attributes on nodes and edges, we need **Rgraphviz**.

`#6 6`
`plot_graphNEL_sprof(sprof04, nodeattrs="time")`

Rgraphviz/graph plot with attributes: see fig. 15 on page 49. This plot gives us the traditional view, highlighting nodes and edges with overall time presence.

ToDo: remove
 global colour; imple-
 ment local colour
ToDo: merge with
 as_graphNEL_sprof

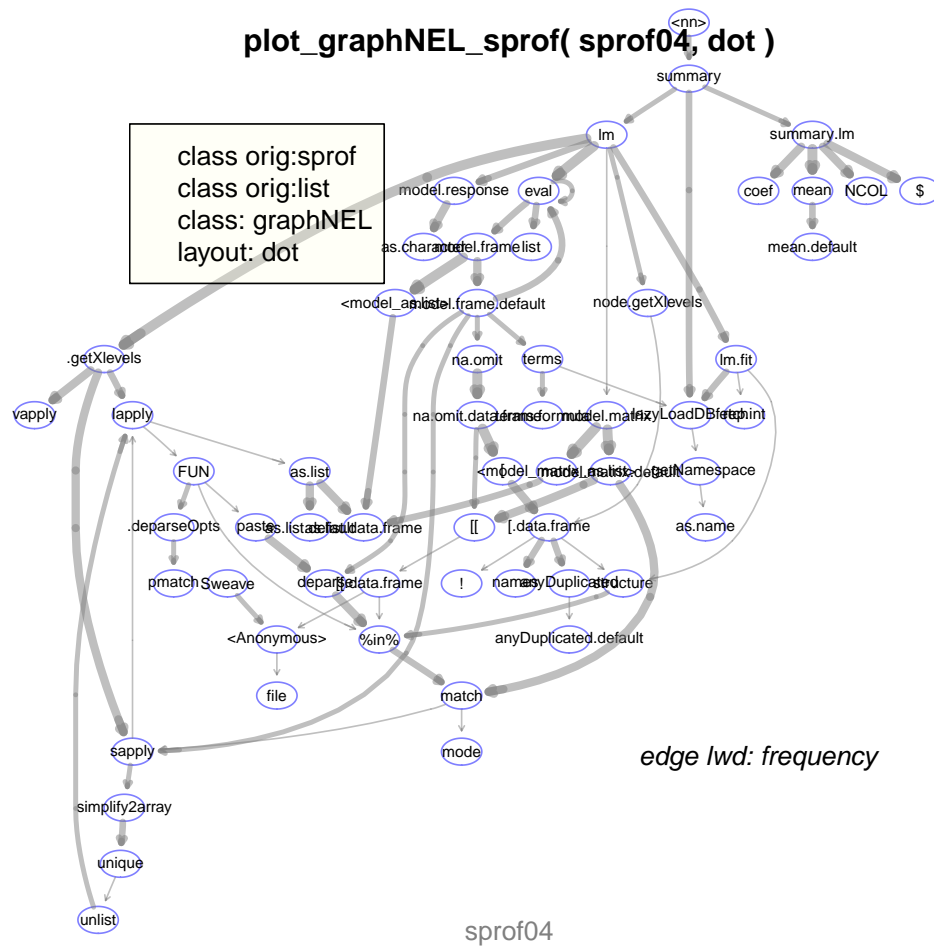


FIGURE 14. Rgraphviz/graph basic plot

We should not overload the plot. We could use colour encoding for the edges, but this would conflict visually with the colour encoding of the nodes. We could use different shapes for classes of nodes, but then we would need an additional display to explain the shape information.

But within the choice of attributes used, we still can select the information shown. To close this round, instead of showing the node information from the rough summary, we can show the information from the run length discussed in section 2.4 on page 37.

Nodes marked by run length and run count: see fig. 16 on page 50.

Input

```
#6 6
library(Rgraphviz)
plot_graphNEL_sprof(sprof04, nodeattrs="runs")
```

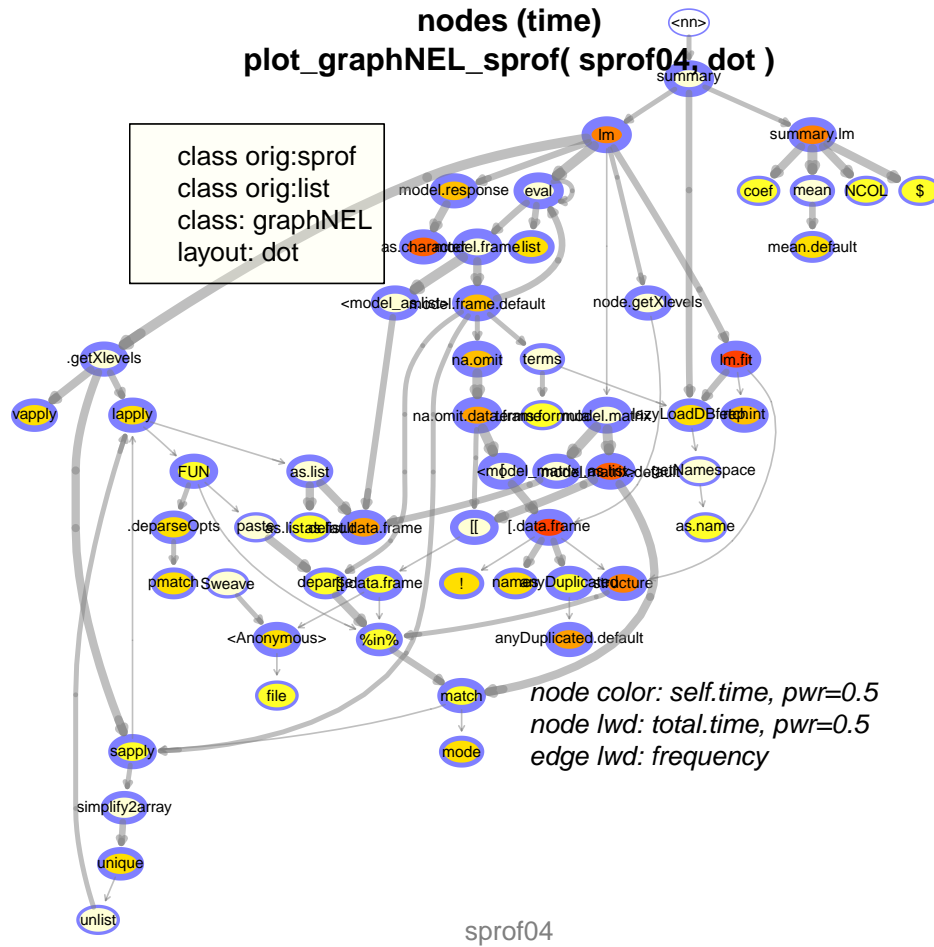



FIGURE 15. Rgraphviz/graph plot with attributes

This plot gives us a more informative view, highlighting nodes by number of runs and average run length and edges with overall time presence. It puts the information in place, showing us the context where the resources are consumed. Some are consumed with good reason, since here work is done. Others are very doubtful and seem to be mere administration. These are candidates for improvement.

The bottom line is: graph layout ist best left to graph representation packages. You can help by trimming and surgery. As far as attributes are concened, the current recommendation is to start with run time lengths and run time frequencies by node and level:

```
plot_graphNEL_sprof3( x, nodeattrs = "runs")
```

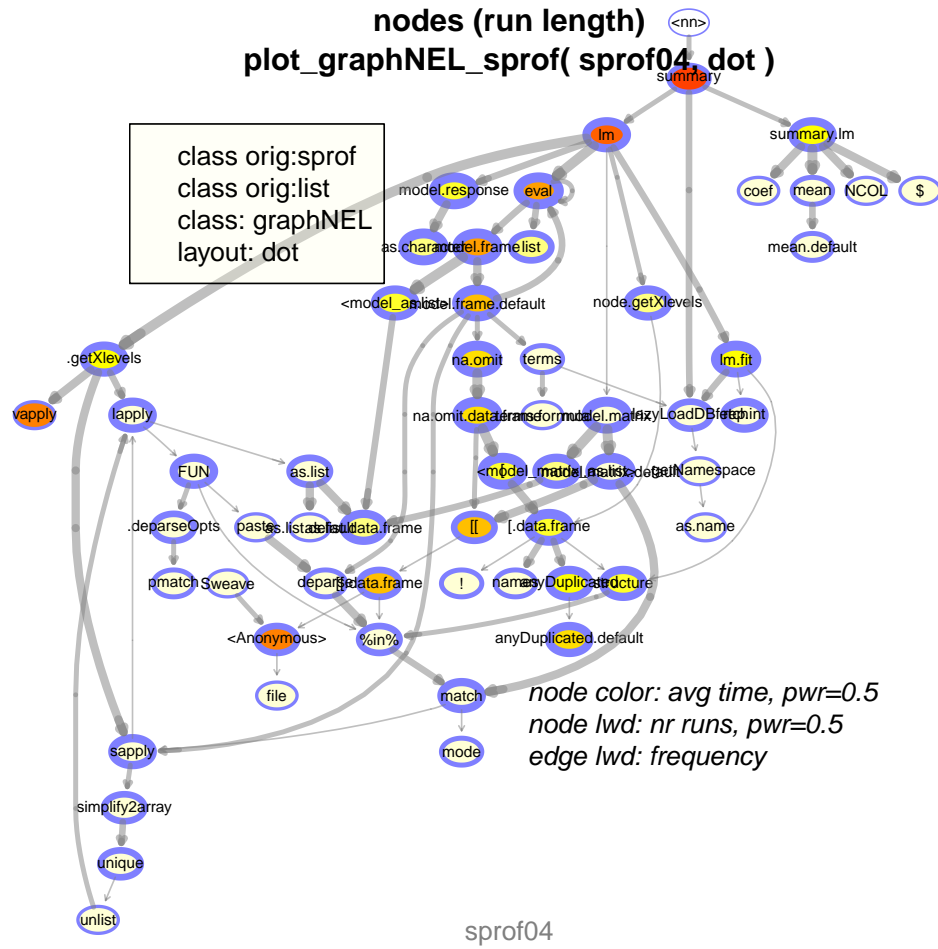


FIGURE 16. nodes (run length)

4. STANDARD OUTPUT

For a reference, here are the standard functions.

```
sprof <- sprof01
```

4.1. **Print.** We omit the (lengthy) print output here and just give the commands as a reference.

```
print_nodes(sprof)
```

```
print_stacks(sprof)
```

```
print_profiles(sprof)
```

The `print()` method for *sprof* objects concatenates these three functions.

4.2. **Summary.**

```
summary_nodes(sprof)
```

```
summary_stacks(sprof)
```

```
summary_profiles(sprof)
```

The `summary()` method for *sprof* objects concatenates these three functions.

ToDo: Clarify: "print prints its argument and returns it invisibly (via `invisible(x)`)."

Return the argument, or some print representation?

ToDo: is there a `print=FALSE` variant to postpone printing to e.g. `xtable`?

4.3. **Plot.** Examples are given in the reference manual for *sprof*.

plot_nodes(sprof) Input

plot_stacks(sprof) Input

plot_profiles(sprof) Input

The `plot.sprof()` method for *sprof* objects concatenates these three functions, see fig. 17. Using the plot functions above allows better control and will be preferred. `shownodes()` may be a sufficient summary, see fig. 7 on page 25.

#12 16 Input
`oldpar <- par(mfrow=c(3,4))`
`plot.sprof(sprof04)`
`par(oldpar)`

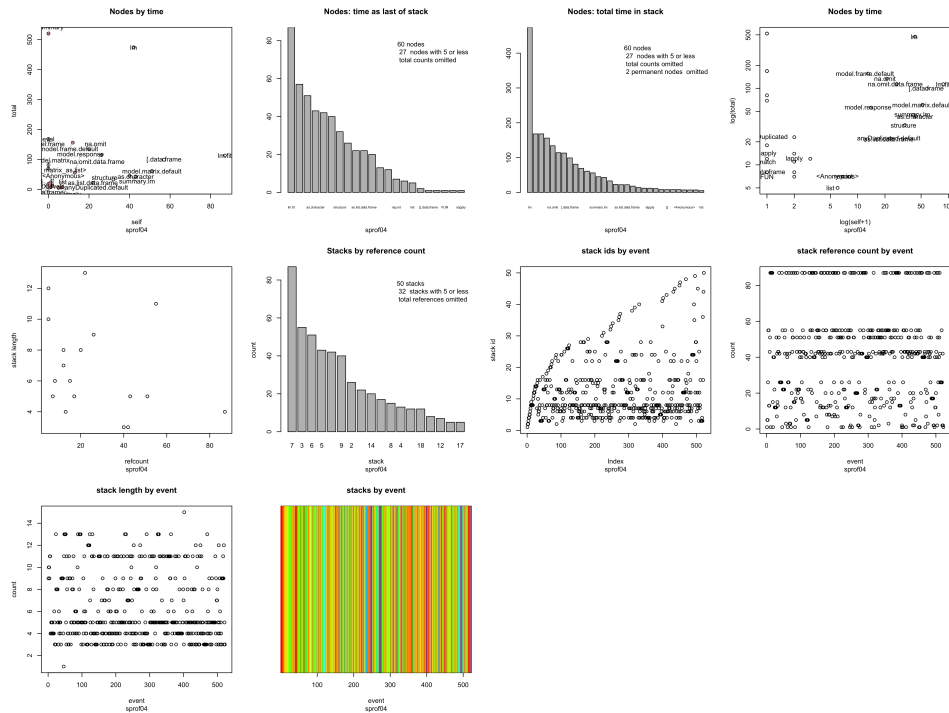


FIGURE 17. `plot.sprof(sprof04)`

5. MORE GRAPHS

Note: This section is collecting experiments with various graph packages. So far, none of the experiments looks too promising. This section is only of interest for you, if you have a preference for some graphic package and want to look up a wrapper here. On the other side: contributions and suggestions are welcome.

Graph layout is a theme of its own. Proposals are readily available, as are their implementation. For some of them, there are R interfaces or re- implementations in R. Their usefulness in our context has to be explored, and the answers will vary with personal preferences.

For some graph layout packages we illustrate an interface here and show a sample result. We use the original profile data here. This is a nasty graph with some R stack peculiarities. The corresponding results for the trimmed profile data are shown in the next section 7.1 on page 95. This is a more realistic example of the kind of graphs you will have to work with.

5.1. Example: regression. In this section, we use the recent version of our example, *sprof02* for demonstration. You can re-run it, using your *sprof* data by modifying this instruction by replacing *sprof02* in the next statement with your profile information.

```
sprof <- sprof02
```

To interface *sprof* to a graph handling package, *adjacency()* can extract the adjacency matrix from the profile.

There are various packages for finding a graph layout, and the choice is open to your preferences. The R packages for most of these are just wrapper

```
sprofadj02 <- adjacency(sprof)
```

This is a format any graph package can handle (maybe). To be on the save side, we provide an (extended) edge list. The added component *lwd* is a proposal for the line width in the graph rendering.

```
sprofedge102 <- edgedf(sprofadj02)
sprofedge102$lwd <- rindex(sprofedge102$count,
  maxindex=12,
  ties.method="min")
```

ToDo: by graph package: preferred input format?

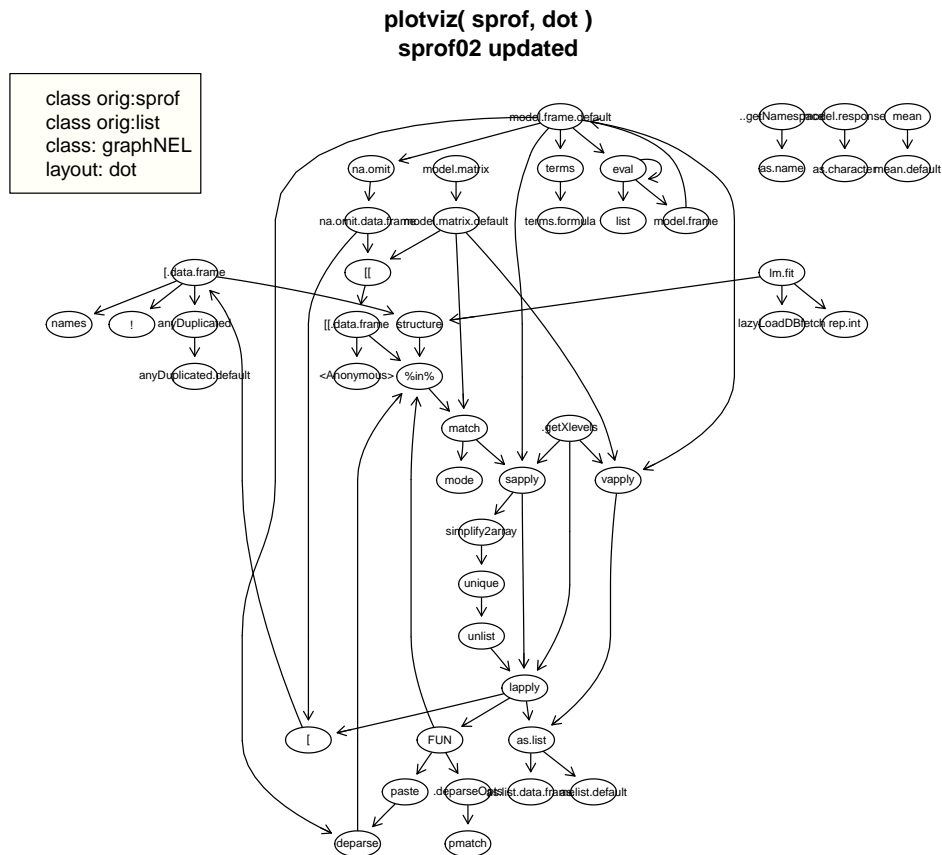
ToDo: use attributes. Edge width should be easy.

ToDo: include information from stack connectivity.

ToDo: add usage of *sprofedge102*

This package is now available from Bioconductor only, see <http://www.bioconductor.org/packages/release/bioc/html/graph.html>.

Input



5.1.2. *igraph* package. Attributes for *igraph* are documented in `help(igraph.plotting)`.

Layouts available for *igraph*:

```
layout.auto(graph, dim=2, ...)
layout.random(graph, params, dim=2)
layout.circle(graph, params)
layout.sphere(graph, params)
layout.fruchterman.reingold(graph, ..., dim=2, params)
layout.kamada.kawai(graph, ..., dim=2, params)
layout.spring(graph, ..., params)
layout.reingold.tilford(graph, ..., params)
layout.fruchterman.reingold.grid(graph, ..., params)
layout.lgl(graph, ..., params)
layout.graphopt(graph, ..., params=list())
layout.svd(graph, d=shortest.paths(graph), ...)
layout.norm(layout, xmin = NULL, xmax = NULL, ymin = NULL, ymax = NULL,
zmin = NULL, zmax = NULL)
```

Input

```
library(igraph)
search()
```

Output

[1] ".GlobalEnv"	"package:igraph"
[3] "package:stringr"	"package:grid"
[5] "package:wordcloud"	"package:RColorBrewer"
[7] "package:Rcpp"	"package:xtable"
[9] "package:sprof"	"package:stats"
[11] "package:graphics"	"package:grDevices"
[13] "package:utils"	"package:datasets"
[15] "package:methods"	"Autoloads"
[17] "package:base"	

Input

```
as_igraph_sprof <- function(sprof, layoutfun, params=NULL,...){
  adj <- adjacency(sprof)
  adj[adj!=0] <- 1
  sprof_igraph <- graph.adjacency(adj)
  sprof_igraph <- set.graph.attribute(sprof_igraph, "layout",
    layoutfun(sprof_igraph,params=params,...),...)
  V(sprof_igraph)$color <- "yellow"
  E(sprof_igraph)$color <- "#0000FF20"
  E(sprof_igraph)$width <- c(1,2)
  return(sprof_igraph)
```

```
}
```

```

Input
sprof_igraph <- graph.adjacency(adjacency(sprof))
sprof_igraph <- set.graph.attribute(sprof_igraph, "layout", layout.kamada.kawai(sprof_igraph))
V(sprof_igraph)$color <- "yellow"
#E(sprof_igraph)$width <- c(1,2)
E(sprof_igraph)$color <- "#0000FF40"
E(sprof_igraph)$width <- c(1,2)
# see

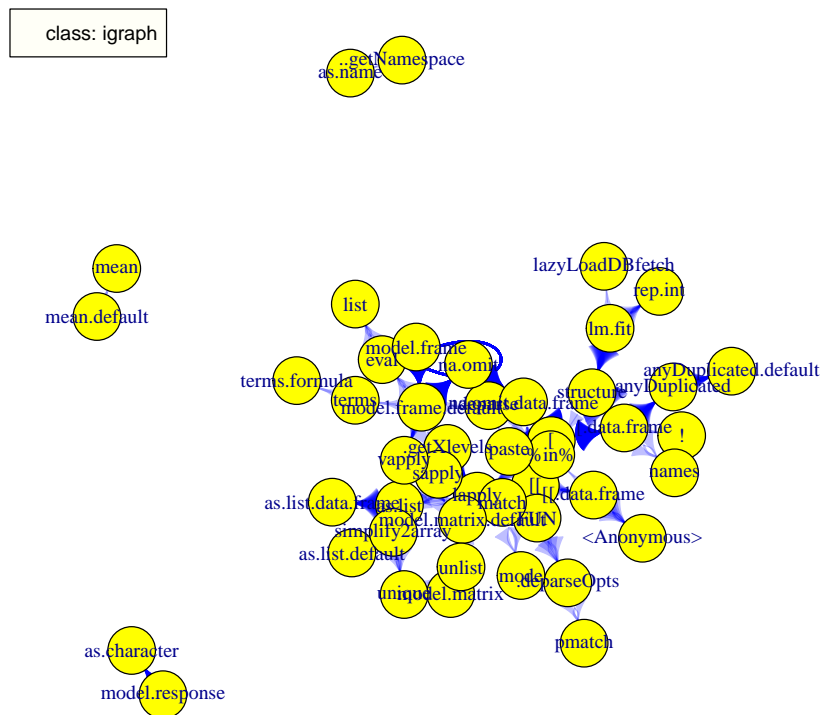
```

```

Input
plot(sprof_igraph, main=paste0("igraph kamada.kawai layout\n", sprof$info$id))
  legend("topleft",
    legend= paste0("class: ",class(sprof_igraph)),
    bg="#FFFFE040",
    seg.len=0
  )

```

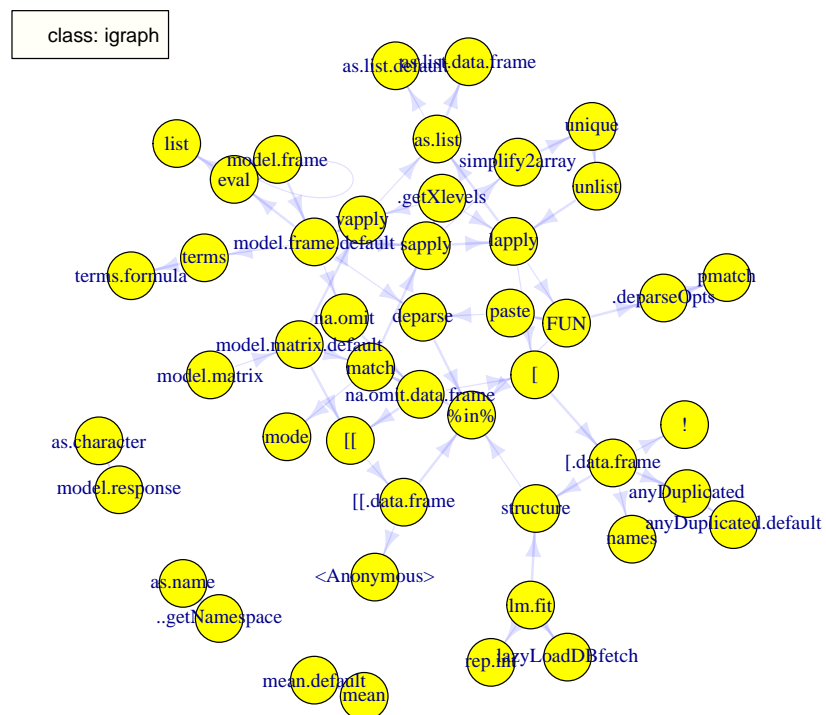
igraph kamada.kawai layout sprof02 updated



Experiments with various layouts for *igraph* follow.

#8 8

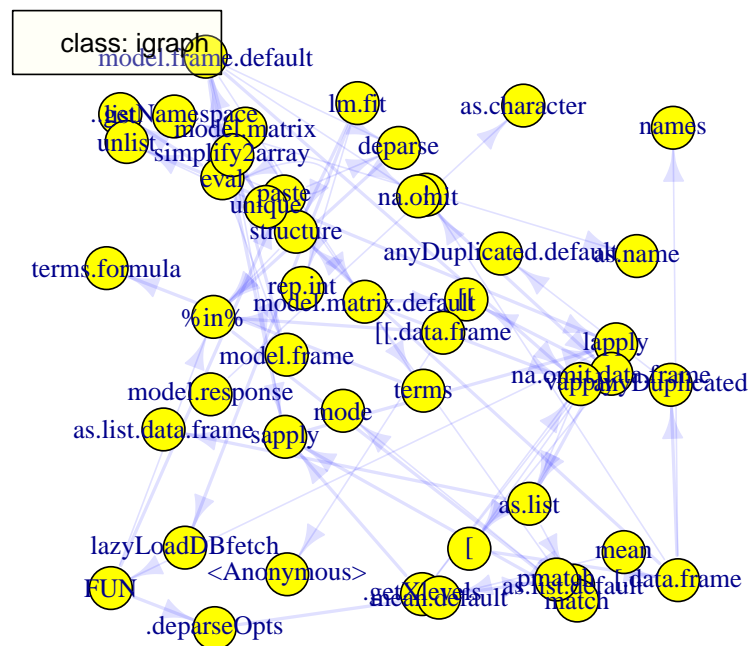
igraph auto layout
sprof02 updated



 Input

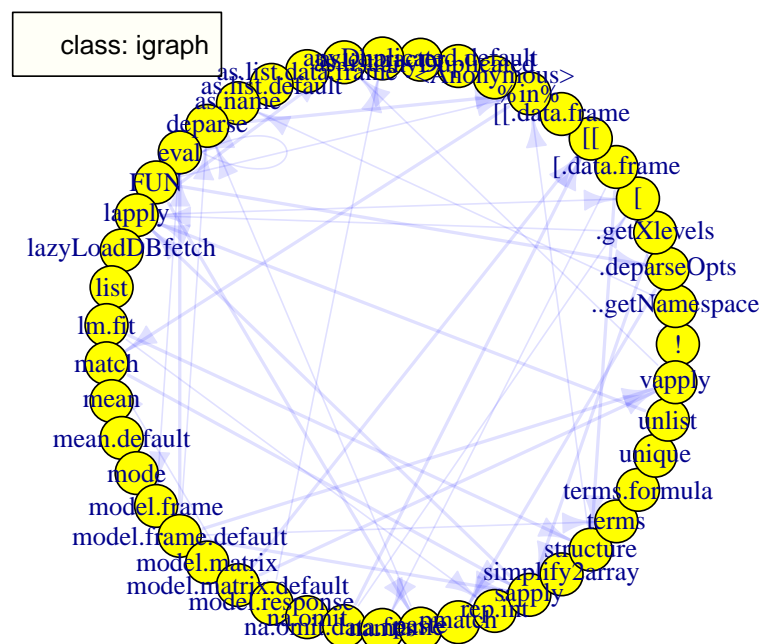
```
#6 6
sprof_ig_random <- as_igraph_sprof(sprof, layout.random)
plot(sprof_ig_random, main=paste0("igraph random layout\n", sprof$info$id))
  legend("topleft",
        legend= paste0("class: ",class(sprof_ig_random)),
        bg="#FFFFE040",
        seg.len=0
        )
```

igraph random layout sprof02 updated



#6 6

igraph circle layout
sprof02 updated



Input

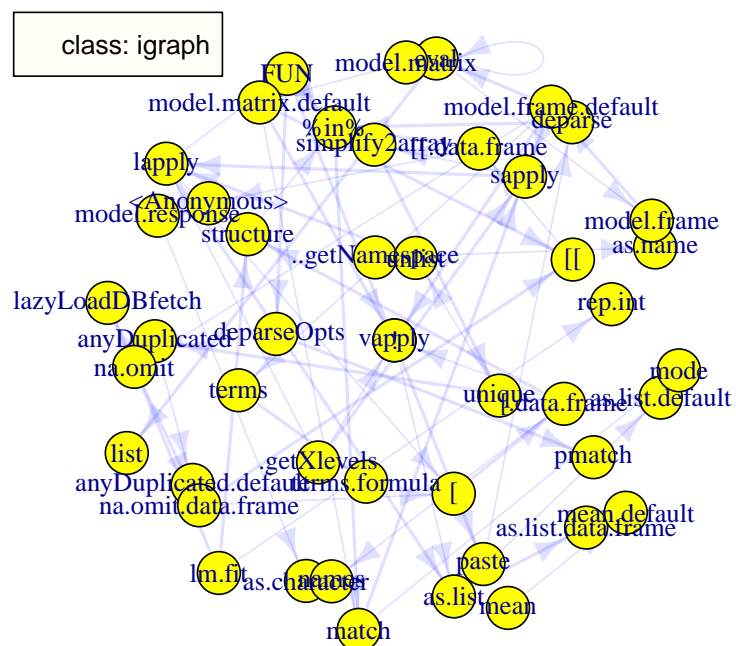
#6 6

```

sprof_ig_auto <- as_igraph_sprof(sprof, layout.sphere)
plot(sprof_ig_auto, main=paste0("igraph sphere layout\n", sprof$info$id))
  legend("topleft",
        legend= paste0("class: ", class(sprof_ig_auto)),
        bg="#FFFE040",
        seg.len=0
        )

```

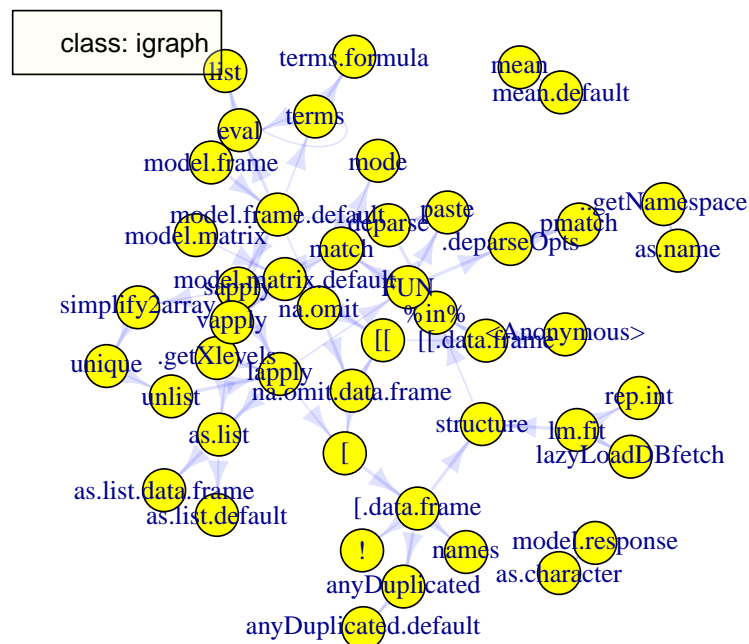
igraph sphere layout sprof02 updated



Input

```
#6 6
sprof_ig_auto <- as_igraph_sprof(sprof, layout.fruchterman.reingold)
plot(sprof_ig_auto, main=paste0("igraph fruchterman.reingold layout\n", sprof$info$id))
  legend("topleft",
        legend= paste0("class: ",class(sprof_ig_auto)),
        bg="#FFFFE040",
        seg.len=0
  )
```

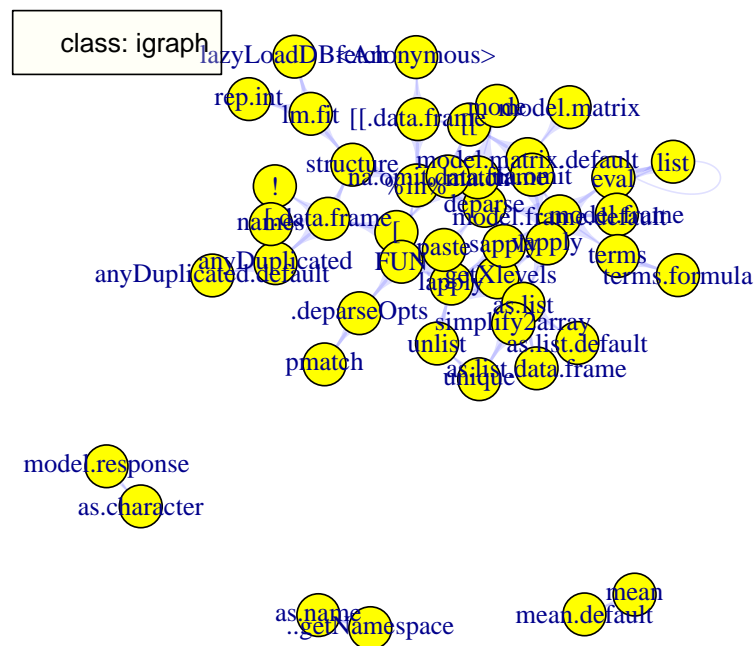
igraph fruchterman.reingold layout
sprof02 updated



 Input

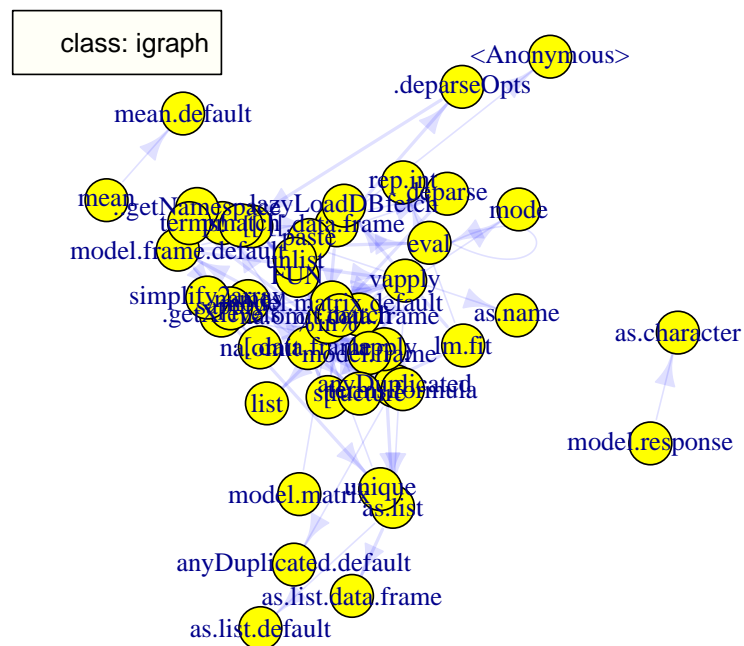
```
#6 6
sprof_ig_auto <- as_igraph_sprof(sprof, layout.kamada.kawai)
plot(sprof_ig_auto, main=paste0("igraph kamada.kawai layout\n", sprof$info$id))
  legend("topleft",
        legend= paste0("class: ", class(sprof_ig_auto)),
        bg="#FFFFE040",
        seg.len=0
  )
```

igraph kamada.kawai layout sprof02 updated



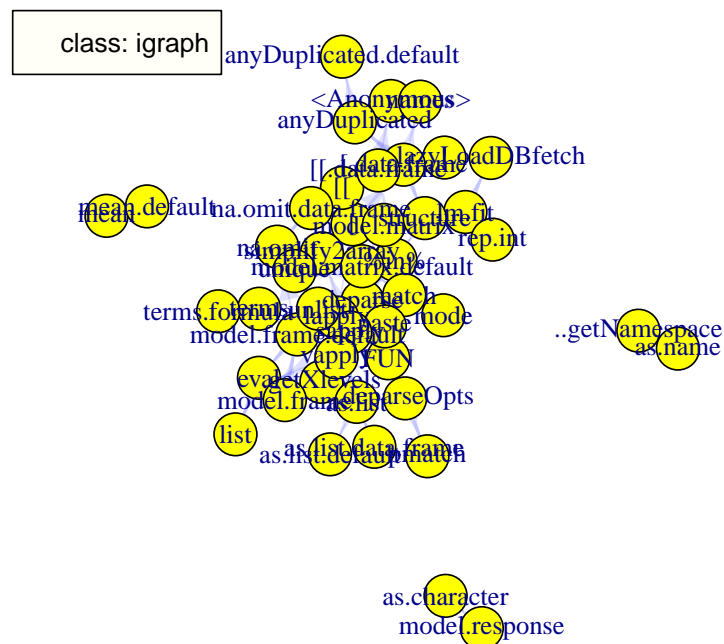
```
#6 6
sprof_ig_auto <- as_igraph_sprof(sprof, layout.spring)
plot(sprof_ig_auto, main=paste0("igraph spring layout\n", sprof$info$id))
  legend("topleft",
        legend= paste0("class: ",class(sprof_ig_auto)),
        bg="#FFFE040",
        seg.len=0
  )
```

igraph spring layout sprof02 updated



```
#6 6
sprof_ig_auto <- as_igraph_sprof(sprof, layout.fruchterman.reingold.grid)
plot(sprof_ig_auto, main=paste0("igraph fruchterman.reingold.grid layout\n", sprof$info$id))
  legend("topleft",
        legend= paste0("class: ", class(sprof_ig_auto)),
        bg="#FFFFE040",
        seg.len=0
  )
```

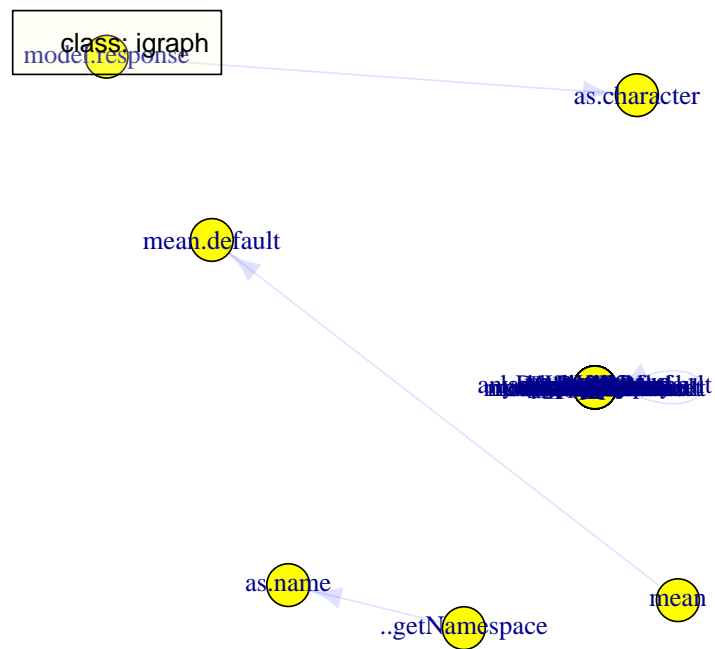
igraph fruchterman.reingold.grid layout sprof02 updated



 Input

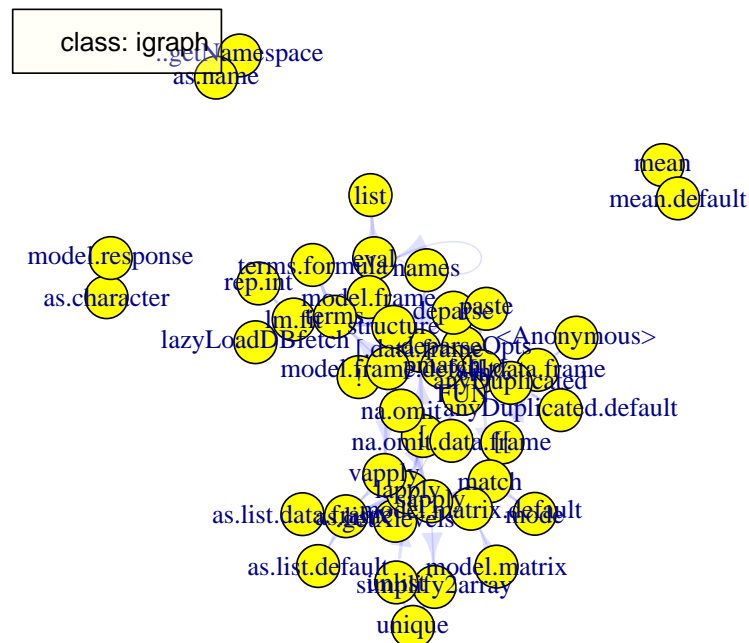
```
#6 6
sprof_ig_auto <- as_igraph_sprof(sprof, layout.lgl)
plot(sprof_ig_auto, main=paste0("igraph lgl layout\n", sprof$info$id))
  legend("topleft",
        legend= paste0("class: ",class(sprof_ig_auto)),
        bg="#FFFFE040",
        seg.len=0
  )
```

igraph lgl layout sprof02 updated

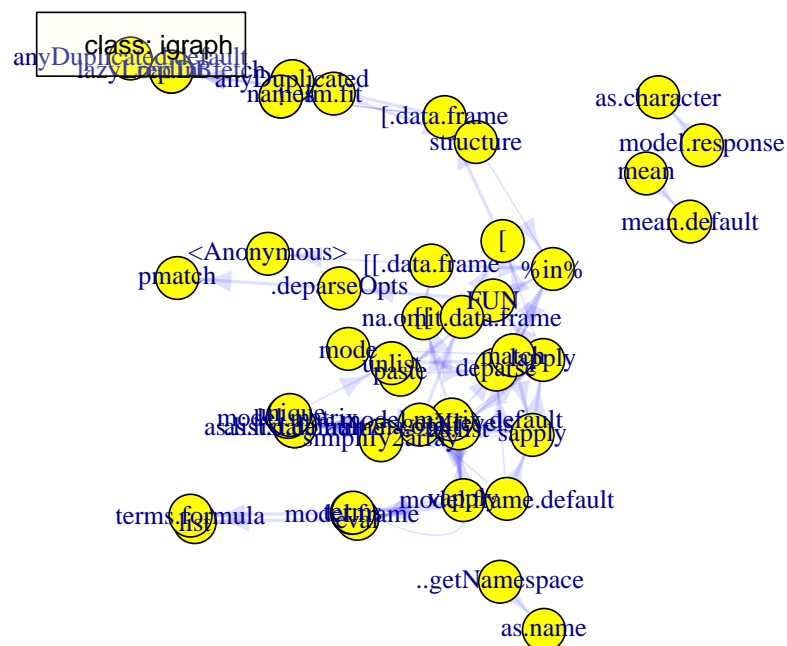


```
#6 6
sprof_ig_auto <- as_igraph_sprof(sprof, layout.graphopt)
plot(sprof_ig_auto, main=paste0("igraph graphopt layout\n", sprof$info$id))
  legend("topleft",
        legend= paste0("class: ",class(sprof_ig_auto)),
        bg="#FFFE040",
        seg.len=0
  )
```

igraph graphopt layout sprof02 updated



igraph svd layout
sprof02 updated



5.1.3. *network* package.

Input

```
detach("package:igraph")
#detach("package:sna")

search()
```

Output

```
[1] ".GlobalEnv"      "package:stringr"
[3] "package:grid"    "package:wordcloud"
[5] "package:RColorBrewer" "package:Rcpp"
[7] "package:xtable"  "package:sprof"
[9] "package:stats"   "package:graphics"
[11] "package:grDevices" "package:utils"
[13] "package:datasets" "package:methods"
[15] "Autoloads"       "package:base"
```

Input

```
library(network)
search()
```

Output

```
[1] ".GlobalEnv"      "package:network"
[3] "package:stringr"  "package:grid"
[5] "package:wordcloud" "package:RColorBrewer"
[7] "package:Rcpp"     "package:xtable"
[9] "package:sprof"    "package:stats"
[11] "package:graphics" "package:grDevices"
[13] "package:utils"    "package:datasets"
[15] "package:methods"  "Autoloads"
[17] "package:base"
```

ToDo: propagate

Input

```
as_network_sprof <- function(sprof) {
  sprofadj02 <- adjacency(sprof)
  nwsprof02 <- as.network(sprofadj02)
  network.vertex.names(nwsprof02) <-
    rownames(sprofadj02) # not honoured by plot
  return(nwsprof02)
}
```

Input

```
plot_network_sprof <- function( nwsprof,
  mode = "fruchtermanreingold", main=NULL, label=NULL, sub=NULL,...)
{
  classnwsprof <- class(nwsprof)
  xid <- deparse(substitute(nwsprof))
  xsubid <- NULL

  if (inherits(nwsprof, "sprof")) {
    xsubid <- nwsprof$info$id
    nwsprof <-
```

```

        as_network_sprof(nwsprof)
    }
    if (!is.null(label))
        warning("explicit label supplied, but will use vertex names")

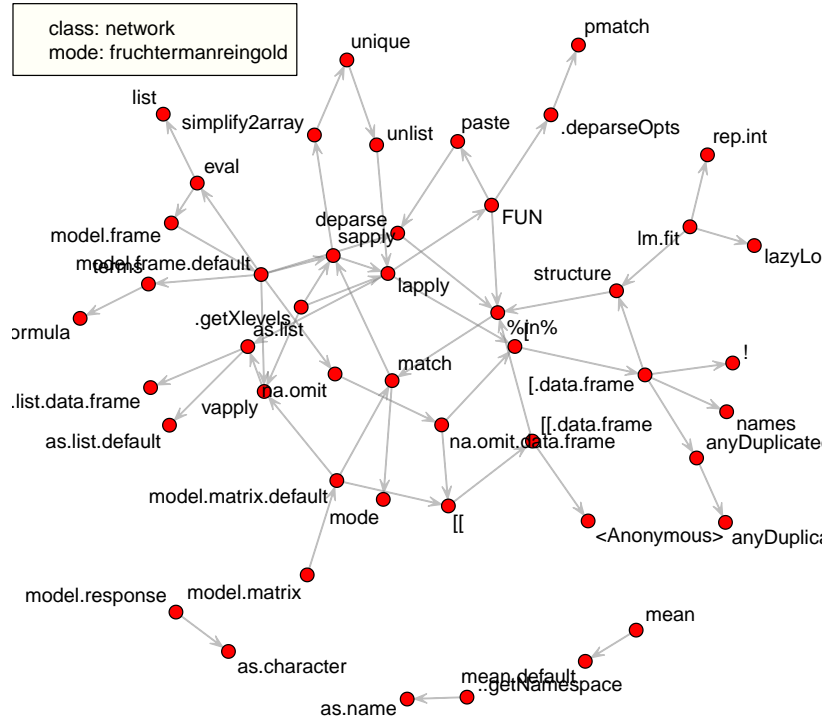
    if (!is.null(sub)) sub <- as.character(sub)
    if (is.null(main))
        main <-
            paste0("plot_network_sprof( ", xid, ", ", mode, " )\n", xsubid) else
        main <-
            paste0(main, "\n plot_network_sprof( ", xid, ", ", mode, " )\n", xsubid)

    plot( nwsprof,
          label = network.vertex.names(nwsprof),
          main= main,
          mode = mode,
          edge.len=2,
          edge.col="#80808080",
          sub= sub,
          cex.main=1.5,...)

    legend("topleft",
          legend=c(paste0("class: ",classnwsprof),paste0("mode: ",mode)),
          seg.len=0
          )
}

```

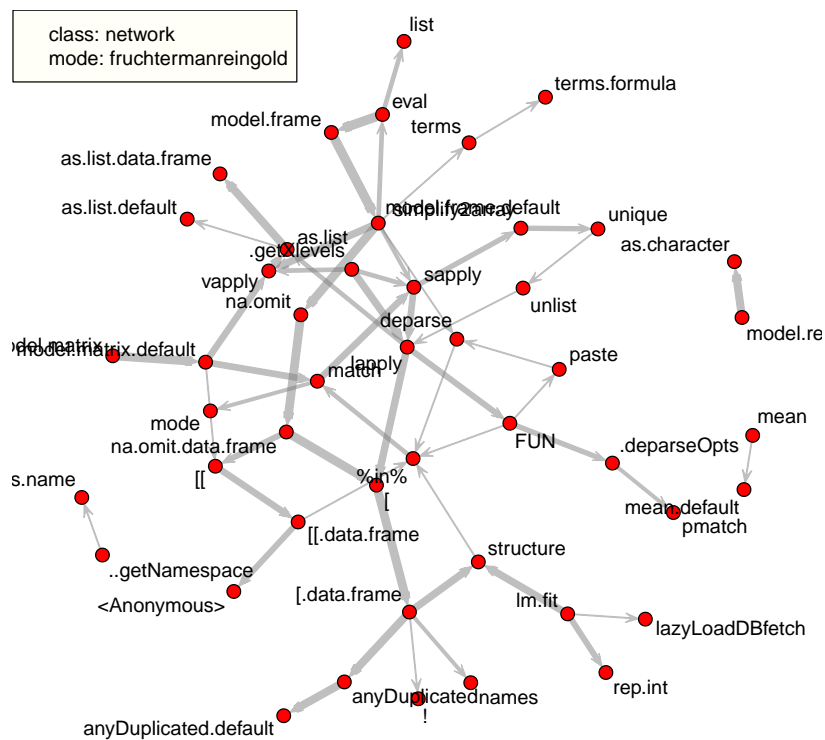

plot_network_sprof(nwsprof02, fruchtermanreingold)



Experiments to include weight.

Input

```
plot_network_sprof( nwsprof02, fruchtermanreingold )
```



5.1.4. *Rgraphviz package.* Package ‘Rgraphviz’ was removed from the CRAN repository.

This package is now available from Bioconductor only, see <http://www.bioconductor.org/packages/release/bioc/html/Rgraphviz.html>.

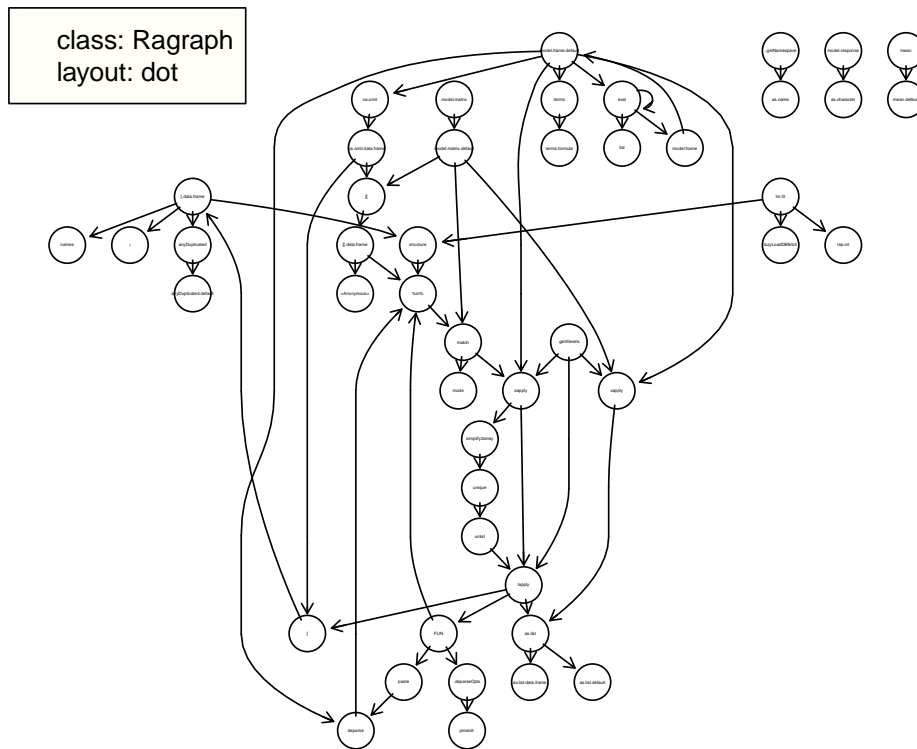
	<i>Input</i>
<code>library(Rgraphviz)</code>	
<code>search()</code>	

	<i>Output</i>
[1] ".GlobalEnv"	"package:Rgraphviz"
[3] "package:graph"	"package:stringr"
[5] "package:grid"	"package:wordcloud"
[7] "package:RColorBrewer"	"package:Rcpp"
[9] "package:xtable"	"package:sprof"
[11] "package:stats"	"package:graphics"
[13] "package:grDevices"	"package:utils"
[15] "package:datasets"	"package:methods"
[17] "Autoloads"	"package:base"

	<i>Input</i>
<code>sprofadjNEL02 <- as(sprofadj02, "graphNEL")</code>	
<code>sprofadjRag02 <- agopen(sprofadjNEL02, name="Rprof Example")</code>	

6 6

```
plotviz( sprofadjRag02, dot )
```

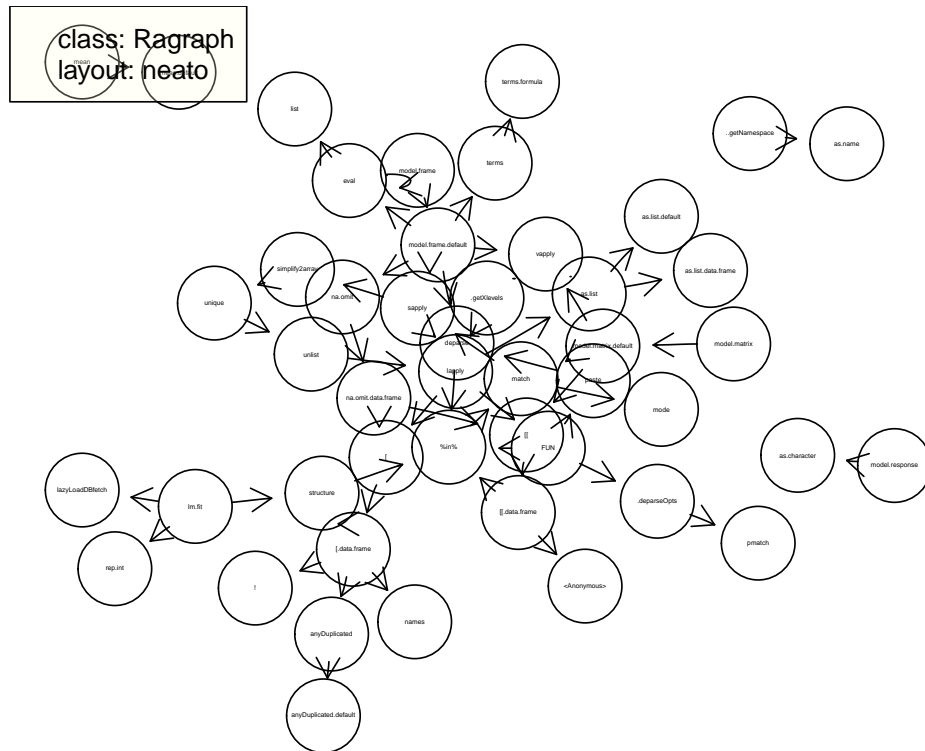


Input

#6 6

```
plotviz(sprofadjRag02,"neato",
        sub=sprof$info$id)
```

```
plotviz( sprofadjRag02, neato )
```

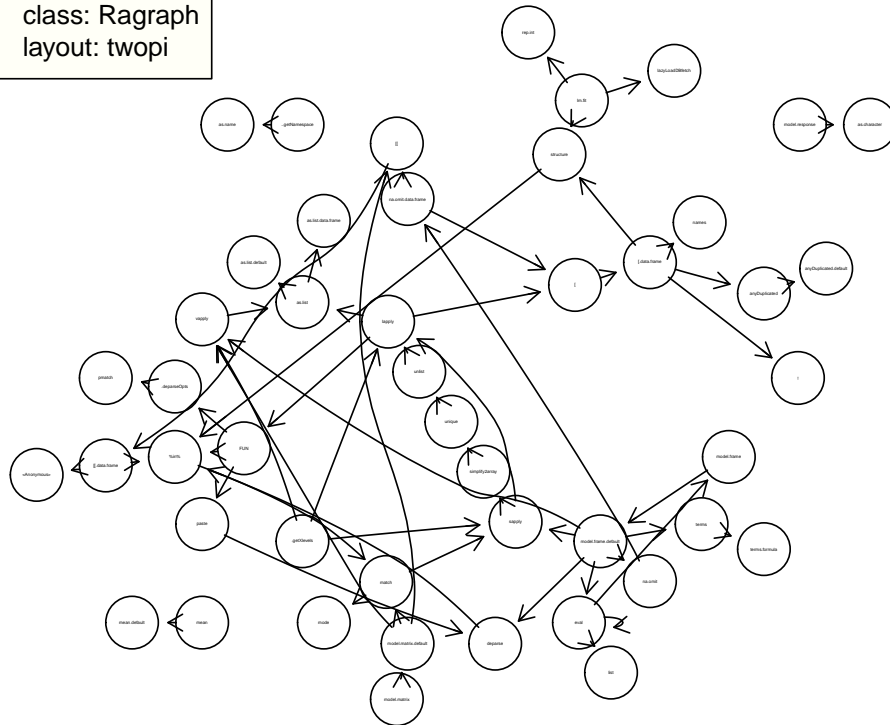


#6 6 *Input*

```
plotviz(sprofadjRag02,"twopi",
        sub=sprof$info$id)
```

plotviz(sprofadjRag02, twopi)

class: Ragraph
layout: twopi

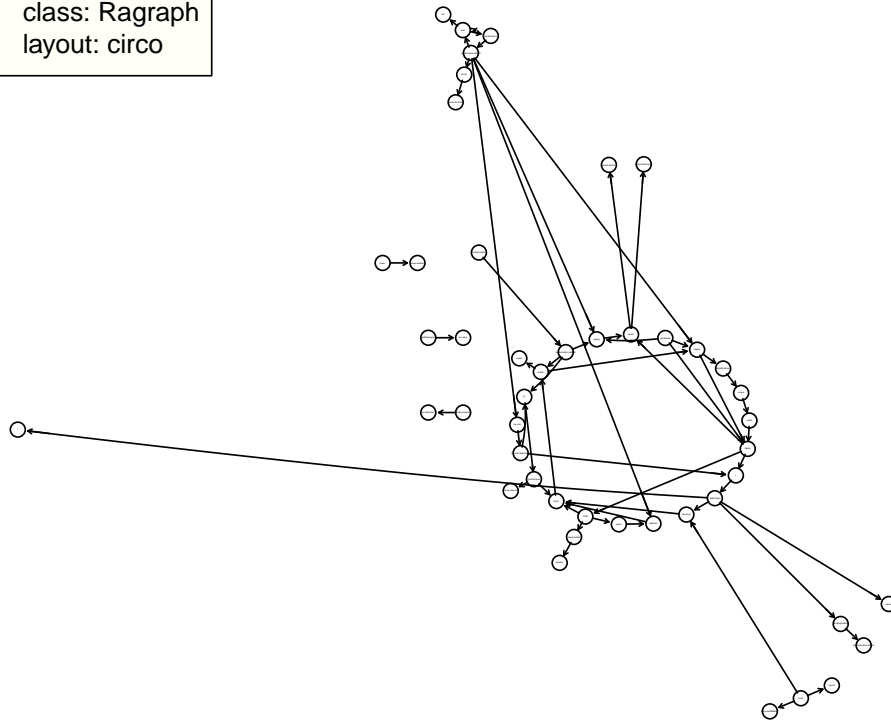


Input

```
#6 6  
plotviz(sprofadjRag02,"circo",  
        sub=sprof$info$id)
```

plotviz(sprofadjRag02, circo)

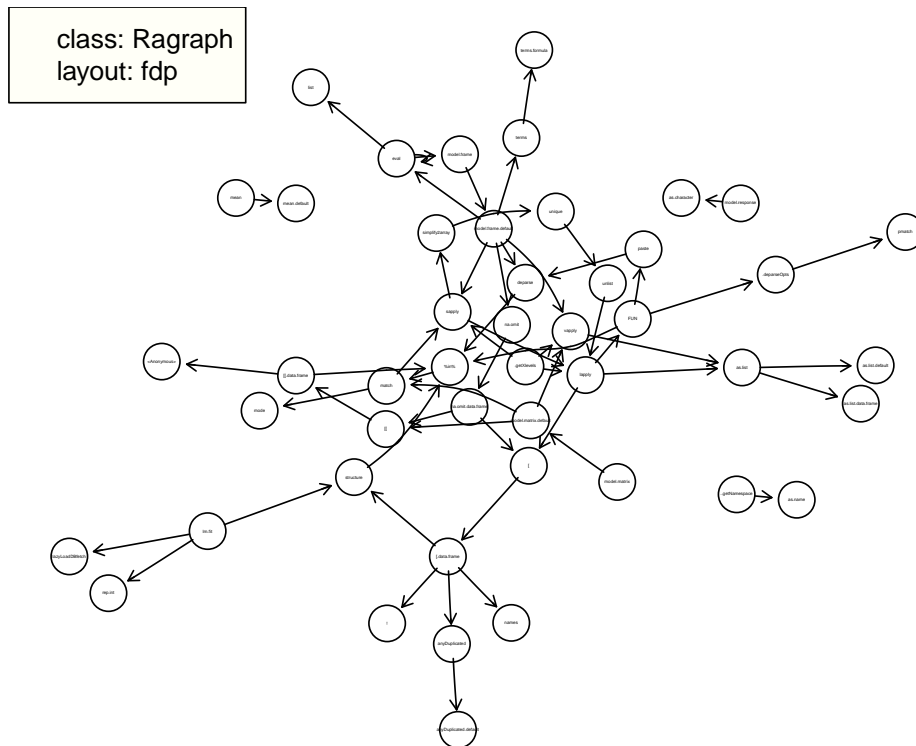
class: Ragraph
layout: circo



#6 6 *Input*

```
plotviz(sprofadjRag02,"fdp",
        sub=sprof$info$id)
```

plotviz(sprofadjRag02, fdp)



6. TEMPLATE

- Run a profiling routine to profile your functions. You can do it on the fly
- Read in the profile
- Get a survey
- Trim base level and burn-in/fade-out
- Get a revised survey
- Use a graph display
- Think!

INDEX

- Topic **adjacency**
 until, 93
- Topic **hplot**
 plot.sprof, 52
 plot_nodes, 14
 shownodes, 52
- Topic **manip**
 adjacency, 24
 list.as.matrix, 24
 profiles_matrix, 24
 rrle, 37
 stacks_matrix, 24
 stackstoadj, 24
- Topic **misc**
 apply, 7
 print, 51
 sampleRprof, 11
 summary, 51
 summaryRprof, 7
- Topic **util**
 adjacency, 53
- ToDo
- 0: remove text vdots from string/name columns. Note: this may be a factor. Use empty string., 5
 - 1: Can we calibrate times to CPU rate? Introduce cpu clock cycle as a time base, 8
 - 1: Defaults by class, 20
 - 1: Move class attributes to package code, 17
 - 1: add class by keyword, 17
 - 1: apply colour to selection?, 14
 - 1: classes need separate colour palette, 20
 - 1: colour by class – redo. Bundle colour index with colour?, 17
 - 1: improve colour: support colour in a structure, 16
 - 1: rearrange stacks? detect order?, 7
 - 1: remove text vdots from string/name columns, 14
 - 2: stackssrc, collstacksdictrv etc. now out of date, 35
 - 2: Implement. Currently best handled on source=text level, 34
 - 2: This section needs to be reworked, 33
 - 2: Warning: data structure still under discussion, 39
 - 2: add a purge function, 29
 - 2: add attributes to stacks, and discuss scope, 20
 - 2: add marginals and conditionals. Provide function node_summary., 39
 - 2: add summary for NA, 39
 - 2: check and stabilise colour linking, 24
 - 2: check and synchronise, 25
 - 2: clean up factor handling, 34
 - 2: colours. recolour. Propagate colour to graph., 28
 - 2: complete matrix conversion, 24
 - 2: cut next level, 34
 - 2: example, 27
 - 2: fix null name, 36
 - 2: function addnode using “call by reference” to be added, 34
 - 2: hack. keep length in nodesrunlength, 41
 - 2: hack. replace by decent vector/array based implementation, 39
 - 2: handle empty stacks and zero counts gracefully, 29
 - 2: keep as factor. This is a sparse cube with margins node, stack level, run length. Nodes are mostly concentrated on few levels., 39
 - 2: move to other section, 33
 - 2: re-think: sort stacks, 23
 - 2: replace by sprof03, 36
 - 2: should this be NA or NULL?, 28
 - 2: sorting/arranging stacks, 20
 - 2: table: node #runs min median run length max, 42
 - 2: updateRprof needs careful checking. For now, we are including long listings here to provide the necessary information, 28
 - 2: use sprof02 or sprof03?, 37
 - 2: warn about undefined vars, e.g. rle, class, ..., 36
 - 2: we could do smart smoothing of the stacks here, 26
 - 2: xreplacenodes: improve implement, 34
 - 3: cut top levels, 43
 - 3: fix null name, 43
 - 3: merge with as_graphNEL_sprof, 47
 - 3: remove global colour; implement local colour, 47
 - 4: Clarify: print prints its argument and returns it invisibly (via invisible(x)). Return the argument, or some print representation?, 51
 - 4: is there a print=FALSE variant to postpone printing to e.g. xtable?, 51
 - 5: add usage of sprofedgel02, 53
 - 5: by graph package: preferred input format?, 53
 - 5: include information from stack connectivity., 53
 - 5: maximum edge.lwd?, 73
 - 5: propagate, 69

5: use attributes. Edge width should be
easy., 53
7: use `as_rkindex`, 85
7: use stack colours, 85

`adjacency`, 24, 53
`apply`, 7

`Index01`, 79

`list.as.matrix`, 24

`plot.sprof`, 52
`plot_nodes`, 14
`print`, 51
`profiles_matrix`, 24

`rrle`, 37

`sampleRprof`, 11
`shownodes`, 52
`stacks_matrix`, 24
`stackstoadj`, 24
`summary`, 51
`summaryRprof`, 7

`until`, 93

R session info:

- R version 3.0.1 (2013-05-16), x86_64-apple-darwin10.8.0
- Locale: en_GB.UTF-8/en_GB.UTF-8/en_GB.UTF-8/C/en_GB.UTF-8/en_GB.UTF-8
- Base packages: base, datasets, graphics, grDevices, grid, methods, stats, utils
- Other packages: graph 1.38.3, RColorBrewer 1.0-5, Rcpp 0.10.3, Rgraphviz 2.4.0, sprof 0.0-6, stringr 0.6.2, wordcloud 2.4, xtable 1.7-1
- Loaded via a namespace (and not attached): BiocGenerics 0.6.0, igraph 0.6.5-2, network 1.7.2, parallel 3.0.1, slam 0.1-28, sna 2.3-1, stats4 3.0.1, tools 3.0.1

L^AT_EX information:

textwidth: 4.9823in linewidth:4.9823in
 textheight: 8.0824in

Svn repository information:

\$HeadURL: svnssh://gsawitzki@svn.r-forge.r-project.org/svnroot/sintro/pkg/sprof/vignettes/sprofiling.Rnw +
 \$Source: /u/math/j40/cvsroot/lectures/src/insider/profile/Rnw/profile.Rnw,v \$
 \$Id: sprofiling.Rnw 219 2013-08-23 19:52:30Z gsawitzki \$
 \$Revision: 219 \$
 \$Date: 2013-08-23 21:52:30 0200(*Fri*, 23*Aug*2013)+
 \$Name: \$
 \$Author: gsawitzki \$

7. XXX – LOST & FOUND

```

                                Input
nodefreq <- rep(0,length(sprof01$nodes$name))
for (i in (1:length(sprof01$stacks$nodes))){
  nodefreq <- nodefreq +
    table( factor(sprof01$stacks$nodes[[i]],
                  levels <- 1:length(sprof01$nodes$name),
                  ordered=FALSE))
}
names(nodefreq) <- sprof01$nodes$name

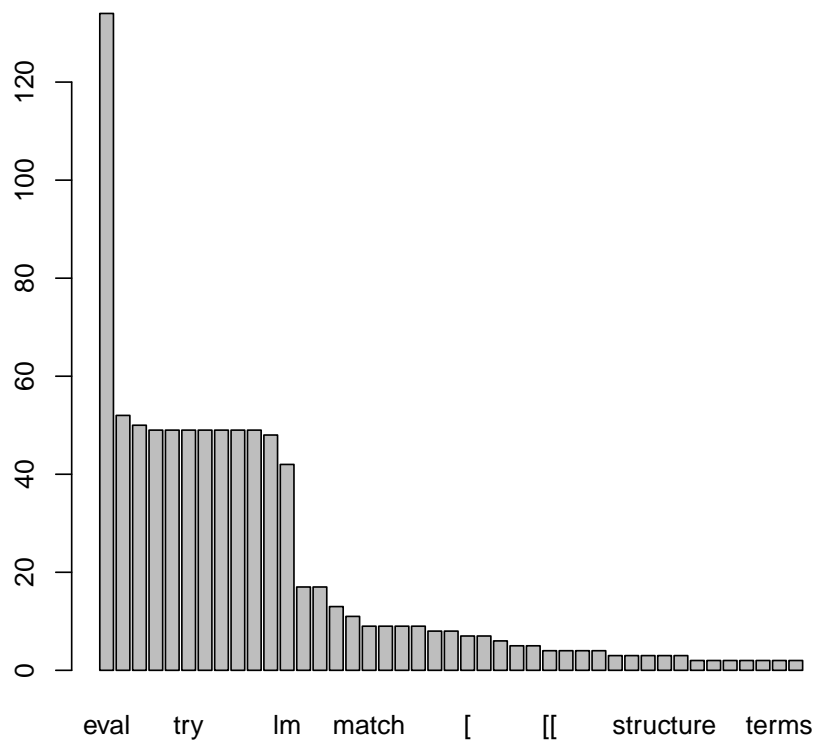
```

Top frequent nodes.

```

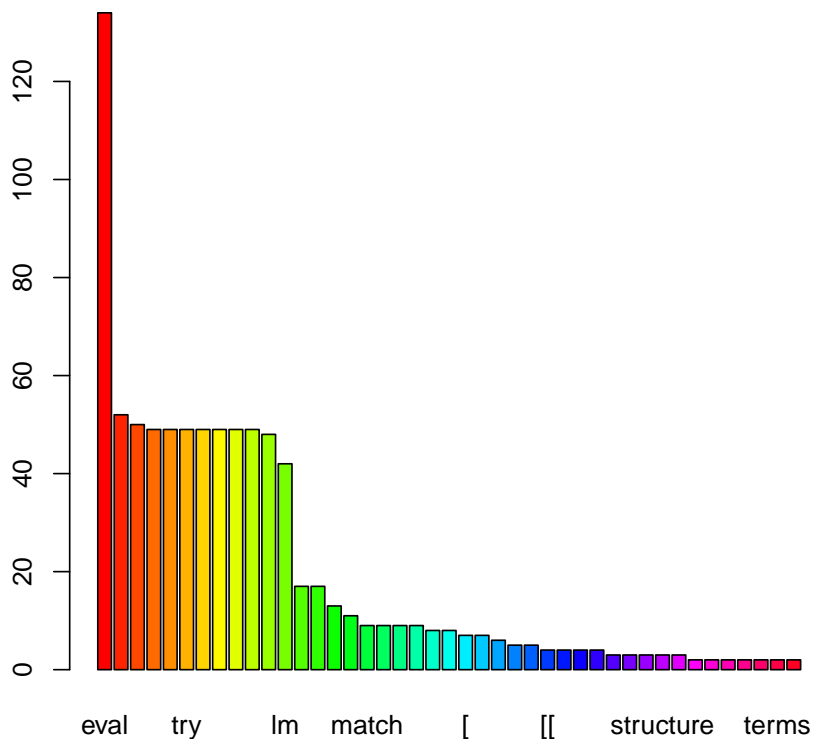
                                Input
ndf <- nodefreq[nodefreq>1]
ondf <- order(ndf,decreasing=TRUE)
barplot(ndf[ondf])

```



Input

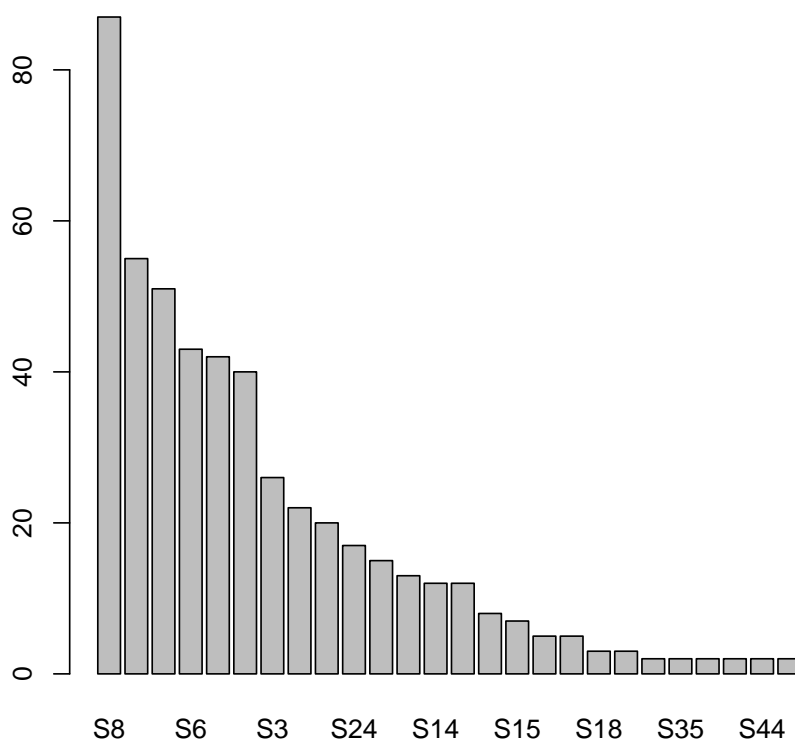
```
barplot(ndf[ondf], col=rainbow(length(ondf)))
```



Top frequent stacks.

Input

```
x <- sprof01
xsrc <- as.matrix(x$stacks$refcount)
rownames(xsrc) <- rownames(xsrc, do.NULL=FALSE, prefix="S")
#stf <- x$stacks$refcount[x$stacks$refcount>1]
#names(stf) <- x$stacks$shortname[x$stacks$refcount>1]
stf <- xsrc[xsrc>1]
names(stf) <- rownames(xsrc)[xsrc>1]
ostf <- order(stf,decreasing=TRUE)
barplot(stf[ostf])
```



```

# xtable cannot handle posix, so we use print output here
# str(profile_nodes_rle, max.level=2, vec.len=3, nchar.max=40, list.len=6)
strx(sprof01$info)

```

```
##strx: sprof01$info
'data.frame':      1 obs. of  9 variables:
 $ id : chr "sprof01"
 $ date : POSIXct, format: "2013-08-24 22:15:44"
 $ nrnodes : int 62
 $ nrstacks : int 50
 $ nrrecords : int 522
 $ sample.interval: num 0.001
 $ sampling.time : num 0.522
 $ cttlines : chr "sample.interval=1000"
 $ cttlinese : chr "1"
```

```

# Input
rownames(sprof01$nodes) <- sprof01$nodes$names
nodesperm <- order(sprof01$nodes$total.time,decreasing=TRUE)
nodesnrobsok <- sprof01$nodes$total.time > 4
sp <- sprof01$nodes$total.time[nodesperm][nodesnrobsok[nodesperm]]
names(sp) <- sprof01$nodes$name[nodesperm][nodesnrobsok[nodesperm]]
barplot(sp,
  main="Nodes, by total time", ylab="total time")

```

A bar chart showing the total time for various functions. The y-axis is labeled 'total time' and ranges from 0 to 500. The x-axis lists functions: <Anonymous>, summary, lm.fit, as.character, supply, and list. The bars for <Anonymous> are the tallest, reaching over 500. The bars for summary, lm.fit, and as.character are significantly shorter, around 100-150. The bars for supply and list are the shortest, near 0.

On the first look, information on the profile level is not informative. Profile records are just recordings of some step, taken at regular intervals. We get a minimal information, if we encode the stacks in colour.

ToDo: use stack colours

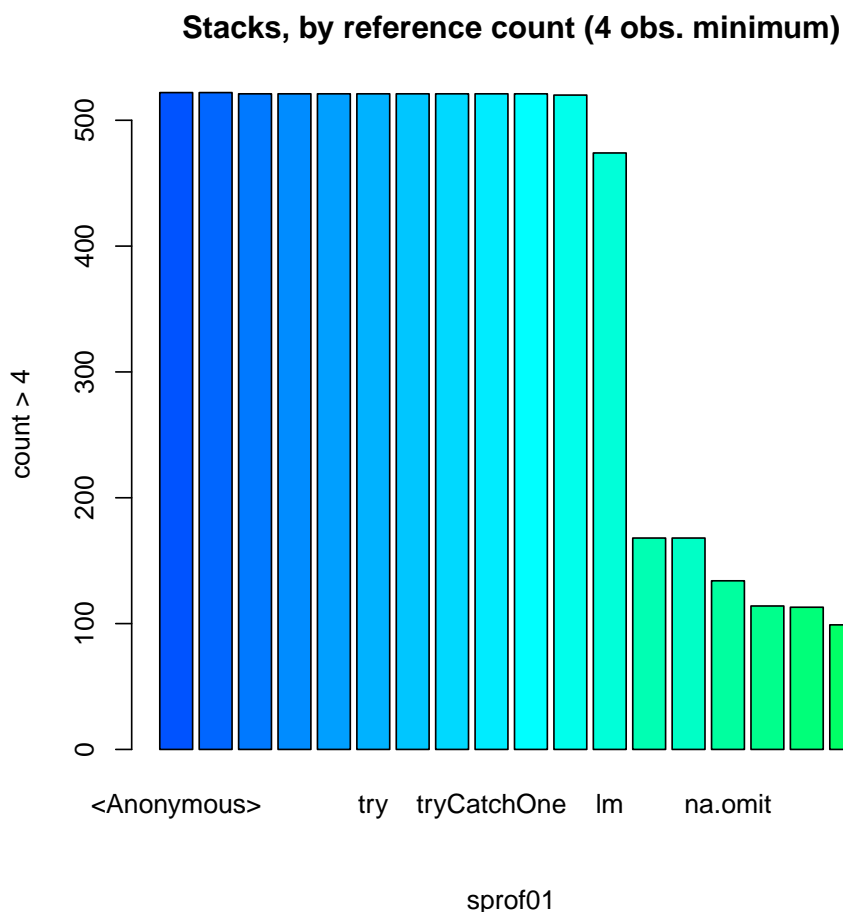
We now do a step down analysis. Aggregating the information from the profiling events, we have the frequency of stack references. On the stack level, we encode the frequency in colour, and linking propagates this to the profile level.

ToDo: use as_rkindex

```

Input
stackfreqscore <- rank(sprof01$stacks$refcount,ties.method="random")
stacksperm <- order(sprof01$stacks$refcount,decreasing=TRUE)
stacksnrobsok <- sprof01$stacks$refcount > 4
stackfreqscore4<- stackfreqscore[stacksperm][stacksnrobsok[stacksperm]]
barplot(sp[stacksnrobsok[stacksperm]],
        main="Stacks, by reference count (4 obs. minimum)", ylab="count > 4",
        col=rainbow(80)[stackfreqscore4], sub=sprof01$info$id)

```



Input

```
prxt(sprof01$nodes,
     caption="nodes",
     label="tab:prSREnodes",
     max.level=2, vec.len=3,nchar.max=40,
     digits=c(0,0,0,2,0,2,0))
```

Table 10: nodes

	name	self.time	self.pct	total.time	total.pct	icol
1	!	2	0.38	2	0.03	50
2	..getNamespace	0	0.00	1	0.01	55
3	.deparseOpts	2	0.38	4	0.05	43
4	.getXlevels	0	0.00	26	0.34	28
5	[0	0.00	99	1.29	19
6	[.data.frame	57	10.92	99	1.29	20
7	[[0	0.00	8	0.10	35
8	[[.data.frame	1	0.19	8	0.10	36
9	%in%	1	0.19	4	0.05	42
10	<Anonymous>	6	1.15	522	6.79	2
<cut>	\vdots	⋮	⋮	⋮	⋮	⋮
53	terms	0	0.00	2	0.03	48
54	terms.formula	1	0.19	1	0.01	59
55	try	0	0.00	521	6.78	10
56	tryCatch	0	0.00	521	6.78	5
57	tryCatchList	0	0.00	521	6.78	9
58	tryCatchOne	0	0.00	521	6.78	7
59	unique	3	0.57	4	0.05	41
60	unlist	0	0.00	1	0.01	56
61	vapply	3	0.57	23	0.30	31
62	withVisible	0	0.00	521	6.78	6

Input

```
#str(sprof01$stacks, max.level=2, vec.len=6,
#      nchar.max=40, list.len=20,width=70, strict.width="wrap"
strx(sprof01$stacks)
```

Output

```
##strx: sprof01$stacks
'data.frame':      50 obs. of  7 variables:
 $ nodes :List of 50
 ..$ : int 52 10 23 55 56 57 58 21 ...
 ..$ : int 52 10 23 55 56 57 58 21 ...
 ..$ : int 52 10 23 55 56 57 58 21 ...
 ..$ : int 52 10 23 55 56 57 58 21 ...
 ..$ : int 52 10 23 55 56 57 58 21 ...
 ..$ : int 52 10 23 55 56 57 58 21 ...
 ..$ : int 52 10 23 55 56 57 58 21 ...
 ..$ : int 52 10 23 55 56 57 58 21 ...
 ..$ : int 52 10 23 55 56 57 58 21 ...
 ..$ : int 52 10 23 55 56 57 58 21 ...
 ..$ : int 52 10 23 55 56 57 58 21 ...
 ..$ : int 52 10 23 55 56 57 58 21 ...
 .. [list output truncated]
 $ shortname : Factor w/ 50 levels
   "S<A>eFttCtCLtC0dTCwVeesleem.m..n.n...[ "|" __truncated__,...: 27 17
   19 1 35 36 37 30 ...
 $ refcount : num 1 5 26 55 13 43 51 87 ...
 $ stacklength : int 19 20 19 21 14 15 15 14 ...
 $ stackheadnodes: int 52 52 52 52 52 52 52 52 ...
 $ stackleafnodes: int 27 28 41 6 39 14 38 30 ...
 $ stackssrc : Factor w/ 50 levels "!" [.data.frame [
   na.omit.data.frame na."| __truncated__,...: 27 28 39 5 37 13 36 30
   ...
```

A summary is provided on request.

Input

```
sumsprof01 <- summary.sprof(sprof01)
```

Output

```
$id
[1] "Profile Summary Sat Aug 24 22:16:12 2013"

$len
[1] 522

$uniquestacks
[1] 50

$nr_runs
[1] 396

$nrstacks
[1] 50
```

```
$stacklength
```

```
[1] 3 25
```

```
$nrnodesperlevel
```

```
[1] 1 1 2 1 1 1 1 1 1 1 1 1 3 10 11 9 9 15 8 7 5 7
```

```
[23] 2 1 1
```

	shortname	root	leaf	self.time	self.pct
!	!	-	LEAF	2	0.383142
..getNamespace	..gN	-	-	0	0.000000
.deparseOpts	.dpO	-	LEAF	2	0.383142
.getXlevels	.gtX	-	-	0	0.000000
[[-	-	0	0.000000
[.data.frame	[.d.	-	LEAF	57	10.919540
[[[[-	-	0	0.000000
[[.data.frame	[[..	-	LEAF	1	0.191571
%in%	%in%	-	LEAF	1	0.191571
<Anonymous>	<An>	-	LEAF	6	1.149425
\$	\$	-	LEAF	1	0.191571
anyDuplicated	anyD	-	LEAF	1	0.191571
anyDuplicated.default	anD.	-	LEAF	22	4.214559
as.character	as.c	-	LEAF	43	8.237548
as.list	as.l	-	-	0	0.000000
as.list.data.frame	a...	-	LEAF	22	4.214559
as.list.default	as..	-	LEAF	1	0.191571
as.name	as.n	-	LEAF	1	0.191571
coef	coef	-	LEAF	1	0.191571
deparse	dprs	-	LEAF	1	0.191571
doTryCatch	dTrC	-	-	0	0.000000
eval	eval	-	LEAF	1	0.191571
evalFunc	evlF	-	-	0	0.000000
file	file	-	LEAF	1	0.191571
FUN	FUN	-	LEAF	1	0.191571
lapply	lppl	-	LEAF	2	0.383142
lazyLoadDBfetch	lLDB	-	LEAF	2	0.383142
list	list	-	LEAF	5	0.957854
lm	lm	-	LEAF	42	8.045977
lm.fit	lm.f	-	LEAF	87	16.666667
match	mtch	-	LEAF	1	0.191571
mean	mean	-	-	0	0.000000
mean.default	mn.d	-	LEAF	2	0.383142
mode	mode	-	LEAF	2	0.383142
model.frame	mdl.f	-	-	0	0.000000
model.frame.default	mdl.f.	-	LEAF	12	2.298851
model.matrix	mdl.m	-	-	0	0.000000
model.matrix.default	mdl.m.	-	LEAF	51	9.770115
model.response	mdl.r	-	LEAF	13	2.490421
na.omit	n.mt	-	LEAF	20	3.831418
na.omit.data.frame	n...	-	LEAF	26	4.980843
names	nams	-	LEAF	2	0.383142
NCOL	NCOL	-	LEAF	1	0.191571
paste	past	-	-	0	0.000000
pmatch	pmtc	-	LEAF	2	0.383142

rep.int	rp.n	- LEAF	7	1.340996
sapply	sppl	- LEAF	1	0.191571
simplify2array	smp2	- -	0	0.000000
structure	strc	- LEAF	32	6.130268
summary	smmr	- -	0	0.000000
summary.lm	smm.	- LEAF	40	7.662835
Sweave	Swev	ROOT -	0	0.000000
terms	trms	- -	0	0.000000
terms.formula	trm.	- LEAF	1	0.191571
try	try	- -	0	0.000000
tryCatch	tryC	- -	0	0.000000
tryCatchList	trCL	- -	0	0.000000
tryCatchOne	trCO	- -	0	0.000000
unique	uniq	- LEAF	3	0.574713
unlist	unls	- -	0	0.000000
vapply	vppl	- LEAF	3	0.574713
withVisible	wthV	- -	0	0.000000
	total.time	total.pct		
!	2	0.383142		
..getNamespace	1	0.191571		
.deparseOpts	4	0.766284		
.getXlevels	26	4.980843		
[99	18.965517		
[.data.frame	99	18.965517		
[[8	1.532567		
[[.data.frame	8	1.532567		
%in%	4	0.766284		
<Anonymous>	522	100.000000		
\$	1	0.191571		
anyDuplicated	23	4.406130		
anyDuplicated.default	22	4.214559		
as.character	43	8.237548		
as.list	23	4.406130		
as.list.data.frame	22	4.214559		
as.list.default	1	0.191571		
as.name	1	0.191571		
coef	1	0.191571		
deparse	2	0.383142		
doTryCatch	521	99.808429		
eval	521	99.808429		
evalFunc	521	99.808429		
file	1	0.191571		
FUN	7	1.340996		
lapply	30	5.747126		
lazyLoadDBfetch	3	0.574713		
list	5	0.957854		
lm	474	90.804598		
lm.fit	113	21.647510		
match	11	2.107280		
mean	2	0.383142		
mean.default	2	0.383142		
mode	2	0.383142		
model.frame	168	32.183908		

model.frame.default	168	32.183908
model.matrix	69	13.218391
model.matrix.default	69	13.218391
model.response	56	10.727969
na.omit	134	25.670498
na.omit.data.frame	114	21.839080
names	2	0.383142
NCOL	1	0.191571
paste	1	0.191571
pmatch	2	0.383142
rep.int	7	1.340996
sapply	14	2.681992
simplify2array	4	0.766284
structure	33	6.321839
summary	520	99.616858
summary.lm	45	8.620690
Sweave	522	100.000000
terms	2	0.383142
terms.formula	1	0.191571
try	521	99.808429
tryCatch	521	99.808429
tryCatchList	521	99.808429
tryCatchOne	521	99.808429
unique	4	0.766284
unlist	1	0.191571
vapply	23	4.406130
withVisible	521	99.808429

Input

```
#str(profile_nodes_rle, max.level=2, vec.len=3, nchar.max=40, list.len=6)
strx(sumsprof01)
```

Output

```
##strx: sumsprof01
'data.frame':      62 obs. of  7 variables:
 $ shortname : Factor w/ 62 levels "!", "..gN", ".dp0", ...: 1 2 3 4 5 6 7
   8 ...
 $ root : Factor w/ 2 levels "-", "ROOT": 1 1 1 1 1 1 1 ...
 $ leaf : Factor w/ 2 levels "-", "LEAF": 2 1 2 1 1 2 1 2 ...
 $ self.time : num 2 0 2 0 0 57 0 1 ...
 $ self.pct : num 0.383 0 0.383 0 ...
 $ total.time: num 2 1 4 26 99 99 8 8 ...
 $ total.pct : num 0.383 0.192 0.766 4.981 ...
```

Input

```
#str(sumsprof01, max.level=2, vec.len=3,
#      nchar.max=40, list.len=6,
#      width=70, strict.width="wrap")
```

The classical approach hides the work that has been done. Actually it breaks down the data to record items. This figure is not reported anywhere. In our case, it can be reconstructed. The profile data have 8456 words in 524 lines.

In our approach, we break down the information. Two lines of control information are split off. We have 522 lines of profile with 50 unique stacks, referencing 62 nodes. Instead of reducing it to a summary, we keep the full information. Information is always kept on its original level.

On the profiles level, we know the sample interval length, and the id of the stack recorded. On the stack level, for each stack we have a reference count, with the sample interval lengths used as weights. This reference count is added up for each node in the stack to give the node timings.

Cheap statistics are collected as they come by. For example, from the stacks table it is cheap to identify root and leaf nodes, and this mark is propagated to the nodes table. These are some attempts to recover the factor structures.

Input

```

xfi <- levels(sprof02$nodes$name)
profile_nodes_rlefac <- lapply(profile_nodes_rle,
  function(xl) {xl$values <- factor(xl$values,
    levels=1:62,
    labels=xfi); xl}) # seems ok
profile_nodes_rletfac <- lapply(profile_nodes_rle,
  function(x) table(x,dnn=c("run length","node")) ) #factors lost again
colnames(profile_nodes_rletfac[[1]]) <-
  sprof02$nodes$name[ as.integer(colnames(profile_nodes_rletfac[[1]]))]
profile_nodes_rletfac1 <- lapply(profile_nodes_rletfac,
  function(xl) {colnames(xl) <-
    sprof02$nodes$name[ as.integer(colnames(xl))];
    xl} )
invisible(lapply(profile_nodes_rletfac1,
  function(x) print.table(t(x),zero.print = ".") ))

```

Output

```

      run length
node   1  2  3  4  5  6  7
<NA>   1  .  .  .  .  .  .
<NA>  17  1  1  1  .  .  .
<NA>   1  .  .  .  .  .  .
<NA>   1  .  .  .  .  .  .
<NA>  40 17  7  4  2  6  1
<NA>  46 18  3  1  1  1  1
<NA>   1  .  .  .  .  .  .
<NA>  55  4  2  .  .  .  .
<NA>  35  3  3  .  .  1  .
<NA>   1  .  .  .  .  .  .

      run length
node           1  2  3  4  5  6  7
as.character    34  3  1  .  .  .  .
as.name          1  .  .  .  .  .  .
eval            40 17  7  4  2  6  1
lapply           16  .  .  1  .  .  .
lazyLoadDBfetch   1  .  .  .  .  .  .
mean.default      1  .  .  .  .  .  .
model.matrix.default 55  4  2  .  .  .  .

```

```

rep.int          7 . . . . .
sapply           3 . . . . .
structure       10 1 . . . 1 .
vapply           . . 1 . . . .

run length
node            1 2 3 4 5 6 7
[              14 . . 1 . . .
[[             1 . . . . . .
%in%           1 . . . . . .
as.list        2 . . . . . .
lapply         2 . . . . . .
match          9 . . . . . .
model.frame   40 17 7 4 2 6 1
simplify2array 1 . . . . . .
vapply        6 1 . . . . .

run length
node            1 2 3 4 5 6 7
[.data.frame   14 . . 1 . . .
[[.data.frame  1 . . . . . .
as.list        7 1 . . . . .
as.list.data.frame 2 . . . . .
FUN            1 . . . . . .
match          1 . . . . . .
model.frame.default 40 17 7 4 2 6 1
sapply         9 . . . . . .
unique         1 . . . . . .

run length
node            1 2 3 4 5 6 7
.deparseOpts   1 . . . . . .
<Anonymous>    1 . . . . . .
anyDuplicated   1 . . . . . .
as.list.data.frame 6 1 . . . . .
as.list.default 1 . . . . . .
deparse        1 . . . . . .
eval           3 . . . . . .
lapply         5 . . . . . .
na.omit        46 10 5 4 1 4 1
sapply         2 . . . . . .
simplify2array  3 . . . . . .
structure      11 . . 1 . . .
terms          1 . . . . . .
unlist         1 . . . . . .
vapply         7 . . . 1 . .

run length
node            1 2 3 4 5 6
as.list        7 . . . 1 .
eval           3 . . . . .
FUN            5 . . . . .
lapply         3 . . . . .
na.omit.data.frame 43 7 4 5 . 4
terms.formula  1 . . . . .
unique         3 . . . . .

run length

```

```

node          1  2  3  4  5  6
  .deparseOpts    3  .  .  .  .  .
    [          45  4  3  3  .  1
    [[          2  .  .  .  1  .
  %in%          1  .  .  .  .  .
  as.list.data.frame 7  .  .  .  1  .
    FUN          1  .  .  .  .  .
    list         3  .  .  .  .  .

      run length
node          1  2  3  4  5  6
  [.data.frame 45  4  3  3  .  1
  [[.data.frame 2  .  .  .  1  .
    match       1  .  .  .  .  .
    paste       1  .  .  .  .  .
    pmatch      2  .  .  .  .  .

      run length
node          1  4  5
  !           1  .  .
  %in%        1  .  .
  <Anonymous> .  .  1
  anyDuplicated 9  2  1
  deparse      1  .  .
  mode         1  .  .
  names        2  .  .

      run length
node          1  4  5
  %in%         1  .  .
  anyDuplicated.default 9  2  1
      run length
node          1
  match 1
      run length
node          1
  mode 1

```

7.1. Trimmed example: regression. In this section, we use the reduced version of our example, *sprof04* for demonstration. Except for the change of the data set, this is just a copy of the previous chapter, collecting the various layouts for easy reference.

Note: This section will be replaced by the template section. For internal use only. Skip it.

Some experiments may have found their way to this chapter. They will be expelled.

You can re-run it, using your *sprof* data by modifying this instruction:

```
sprof <- sprof04 Input
```

To interface *sprof* to a graph handling package, *until()* can extract the adjacency matrix from the profile.

sprofadj02 <- *adjacency(sprof)* *Input*

This is a format any graph package can handle (maybe).

7.1.1. *graph* package.

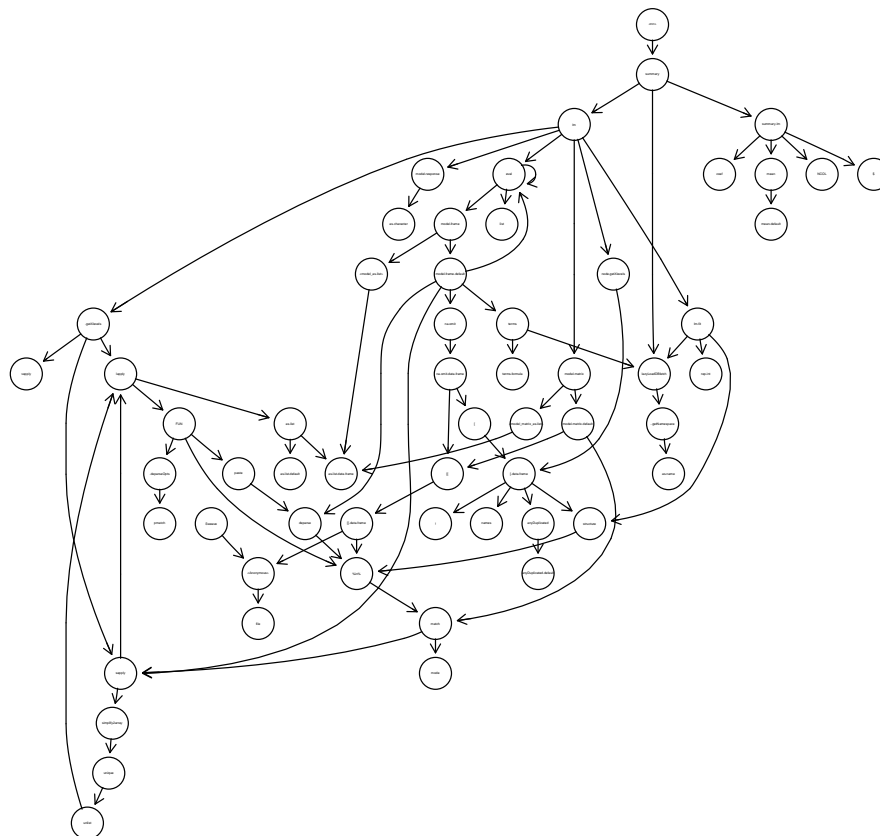
```
library(graph)
search()
```

	Output
[1] ".GlobalEnv"	"package:graph"
[3] "package:stringr"	"package:grid"
[5] "package:wordcloud"	"package:RColorBrewer"
[7] "package:Rcpp"	"package:xtable"
[9] "package:sprof"	"package:stats"
[11] "package:graphics"	"package:grDevices"
[13] "package:utils"	"package:datasets"
[15] "package:methods"	"Autoloads"
[17] "package:base"	

Some tests for scaling ...

```
sprofadjNEL02 <- as(sprofadj02, "graphNEL")
```

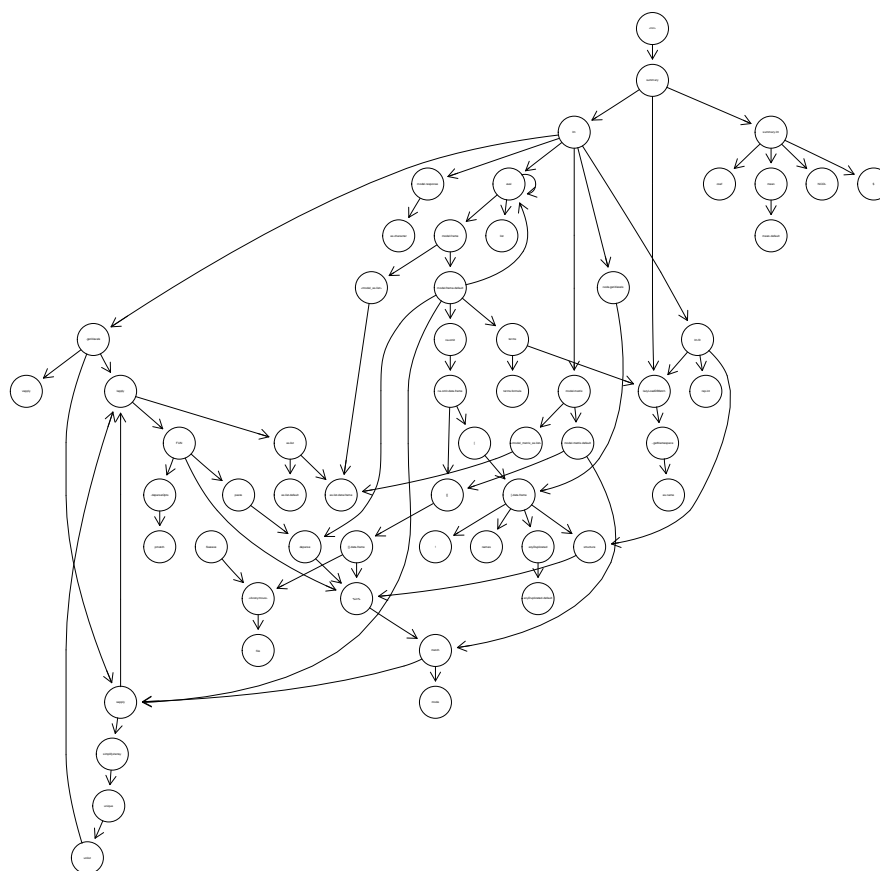
```
#8
plot(sprofadjNEL02, main="sprof04: graph layout",)
#detach("package:graph")
```

sprof04: graph layout

Input

```
#18
plot(sprofadjNEL02, main="sprof04: graph layout",)
#detach("package:graph")
```

sprof04: graph layout

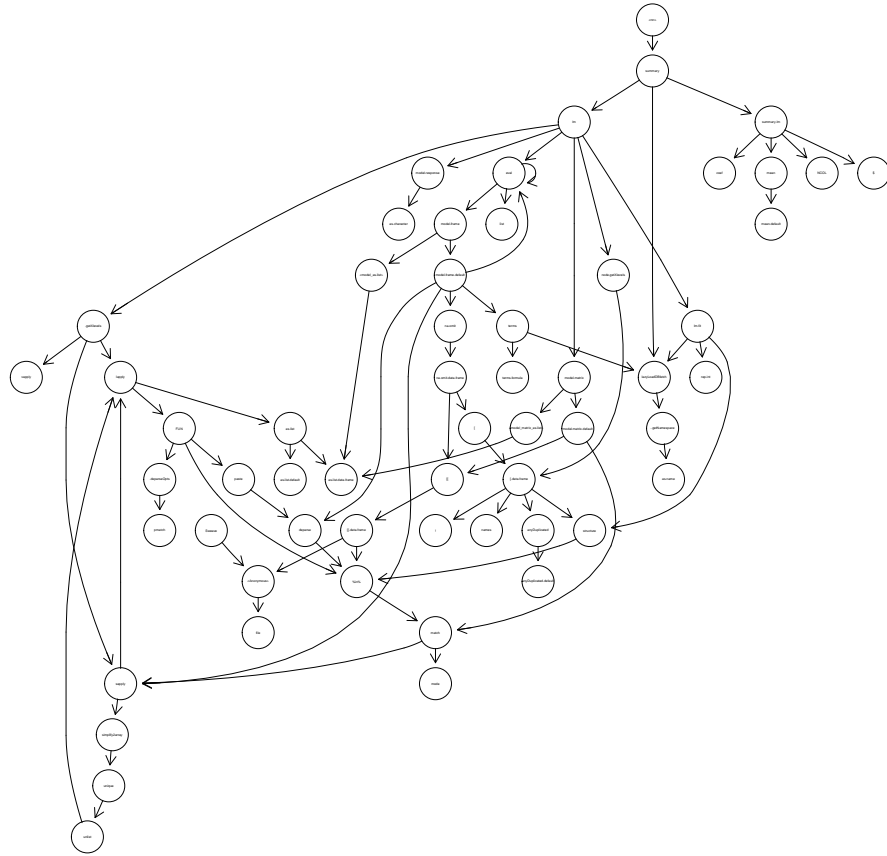


Input

#12

`plot(sprofadjNEL02, main="sprof04: graph layout")``#detach("package:graph")`

sprof04: graph layout



7.1.2. *igraph* package.

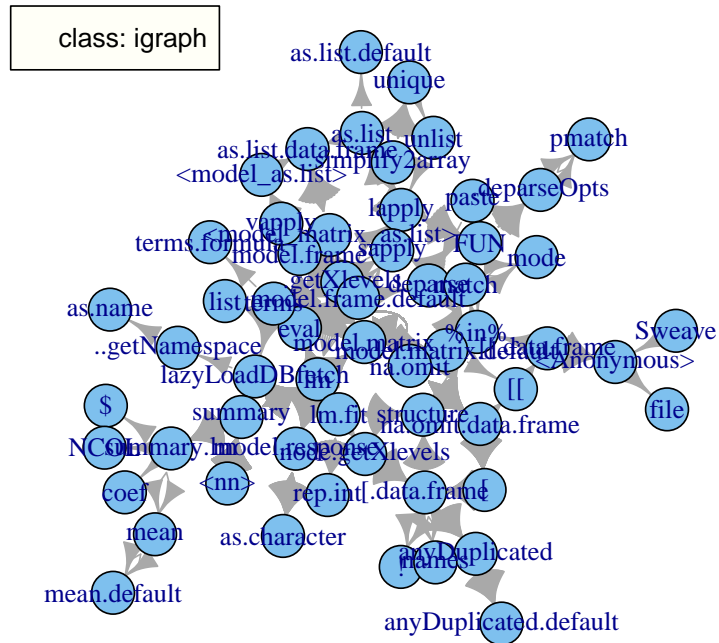
<i>library(igraph)</i> <i>search()</i>	<i>Input</i>
---	--------------

	<i>Output</i>
[1] ".GlobalEnv"	"package:igraph"
[3] "package:stringr"	"package:grid"
[5] "package:wordcloud"	"package:RColorBrewer"
[7] "package:Rcpp"	"package:xtable"
[9] "package:sprof"	"package:stats"
[11] "package:graphics"	"package:grDevices"
[13] "package:utils"	"package:datasets"
[15] "package:methods"	"Autoloads"
[17] "package:base"	

<i>sprof_igraph <- graph.adjacency(sprofadj02)</i>	<i>Input</i>
---	--------------

<i>#6</i> <i>plot(sprof_igraph, main="sprof04: igraph layout: trimmed data")</i> <i> legend("topleft",</i> <i> legend=c(paste0("class: ",class(sprof_igraph))),</i> <i> bg="#FFFFE040",</i> <i> seg.len=0</i> <i>)"lightyellow" =#FFFFE0</i> <i>detach("package:igraph")</i>	<i>Input</i>
---	--------------

sprof04: igraph layout: trimmed data



7.1.3. *network* package.

Input

```
library(network)
search()
```

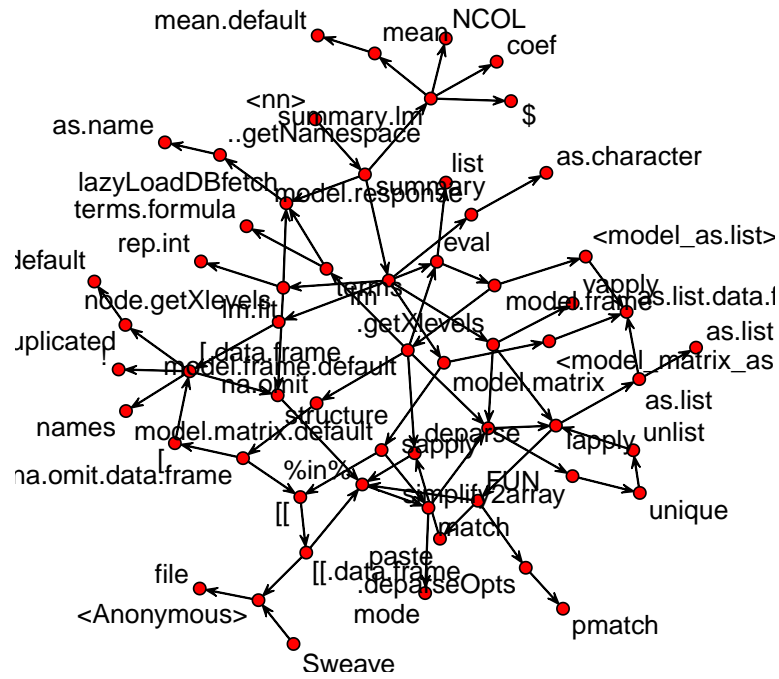
Output

```
[1] ".GlobalEnv"      "package:network"
[3] "package:stringr"  "package:grid"
[5] "package:wordcloud" "package:RColorBrewer"
[7] "package:Rcpp"     "package:xtable"
[9] "package:sprof"    "package:stats"
[11] "package:graphics" "package:grDevices"
[13] "package:utils"    "package:datasets"
[15] "package:methods"  "Autoloads"
[17] "package:base"
```

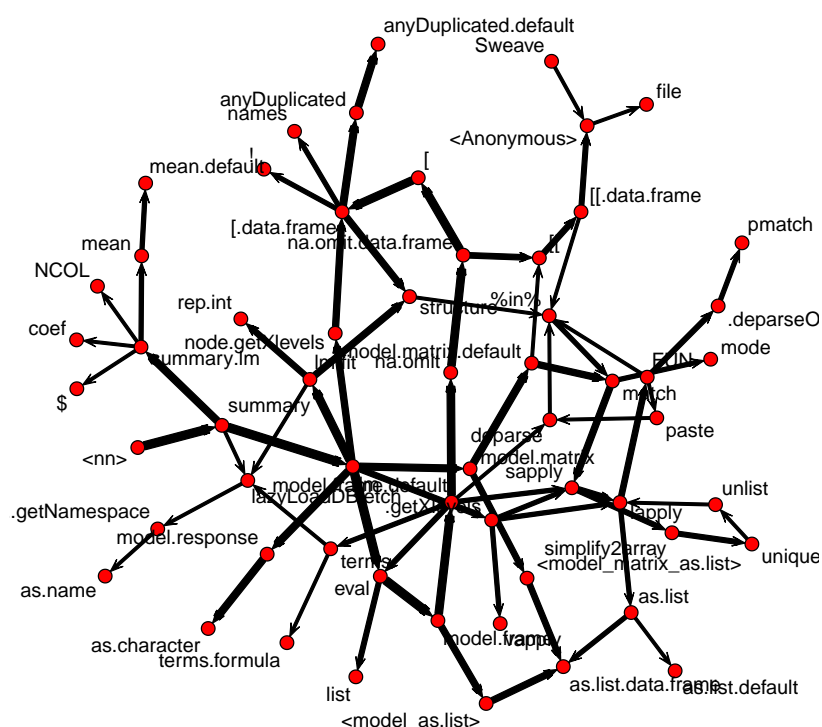
Input

```
#6
nwsprof02 <- as.network(sprofadj02) # names is not imported
network.vertex.names(nwsprof02) <- rownames(sprofadj02) # not honoured by plot
plot(nwsprof02, label=rownames(sprofadj02),
     main="sprof04: network layout: trimmed data", cex.main=2)
```

sprof04: network layout: trimmed dat



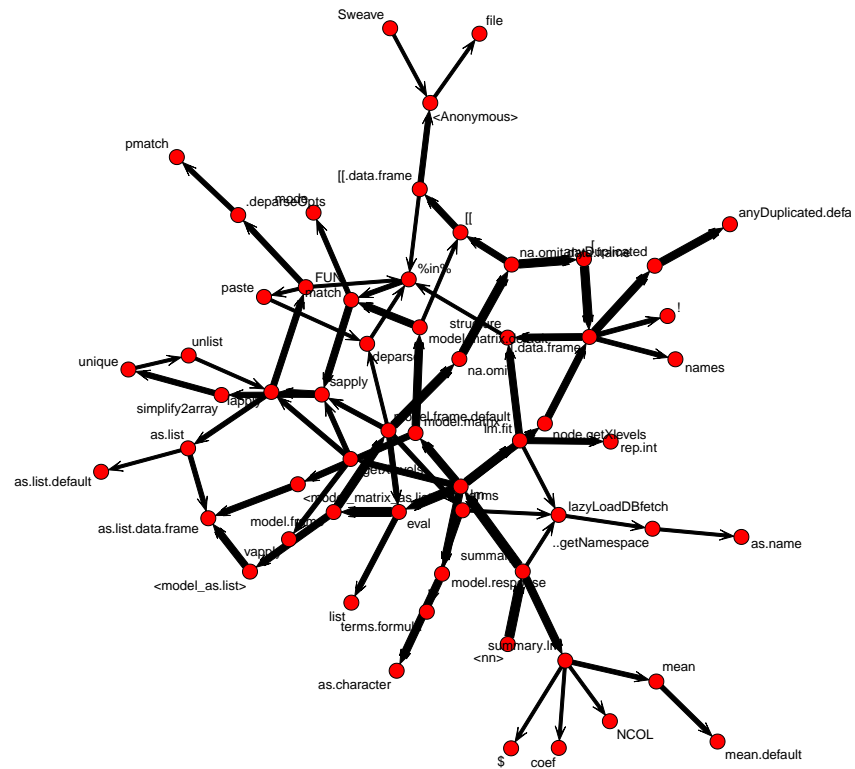

```
#8
edge.lwd<-sprofadj02
edge.lwd[edge.lwd>0]<- rkindex(edge.lwd[edge.lwd>0], maxindex=12,
  ties.method="max")
plot(nwsprof02, label=rownames(sprofadj02),
  main="sprof04: network layout: trimmed data",
  cex.main=2, edge.lwd=edge.lwd)
```

sprof04: network layout: trimmed data

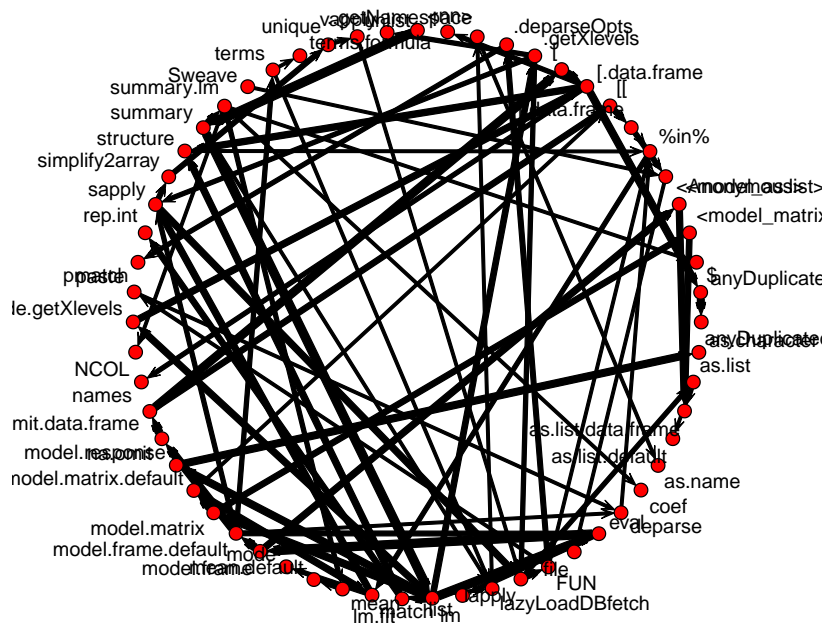
#12

```
plot(nwsprof02, label=rownames(sprofadj02),
main="sprof04: network kamadakawai layout: \n trimmed data",
mode="kamadakawai",
cex.main=2, edge.lwd=edge.lwd)
```

**sprof04: network kamadakawai layout:
trimmed data**



**sprof04: network circle layout:
trimmed data**

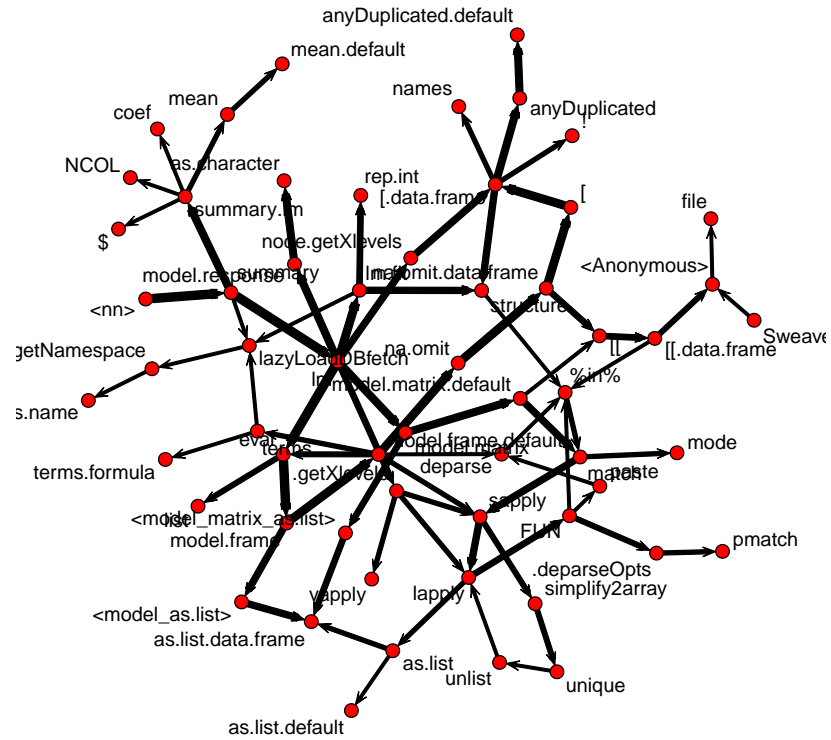


```

      Input
plot(nwsprof02, label=rownames(sprofadj02),
     main="sprof04: network fruchtermanreingold layout: \n trimmed data",
     mode="fruchtermanreingold",
     cex.main=2, edge.lwd=edge.lwd)
detach("package:network")

```

trimmed data



7.1.4. *Rgraphviz* package.

Input

```
library(Rgraphviz)
search()
```

	Output
[1] ".GlobalEnv"	"package:Rgraphviz"
[3] "package:graph"	"package:stringr"
[5] "package:grid"	"package:wordcloud"
[7] "package:RColorBrewer"	"package:Rcpp"
[9] "package:xtable"	"package:sprof"
[11] "package:stats"	"package:graphics"
[13] "package:grDevices"	"package:utils"
[15] "package:datasets"	"package:methods"
[17] "Autoloads"	"package:base"

Input

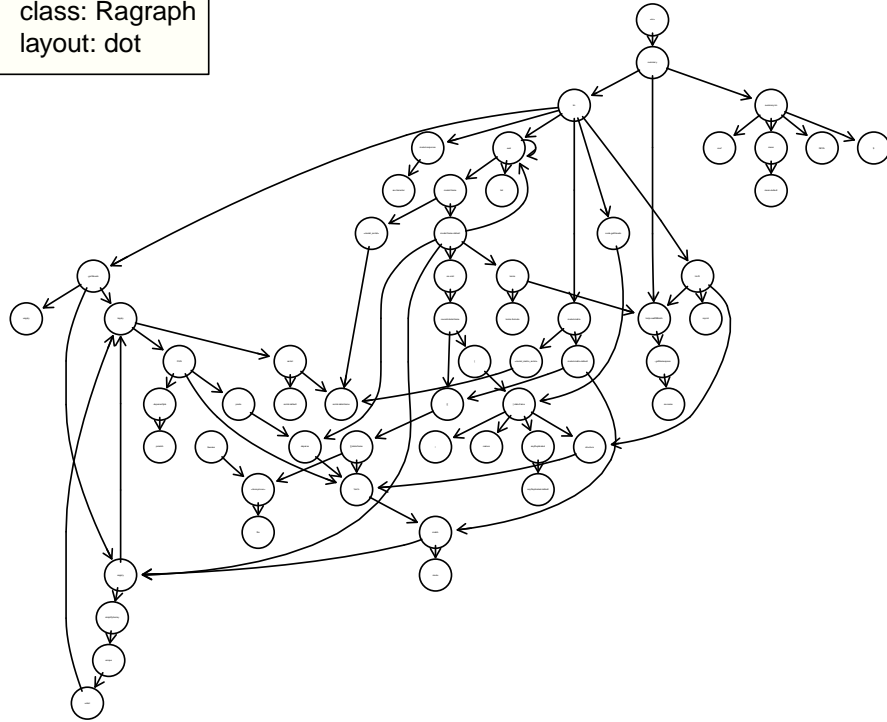
```
sprofadjRag02 <- agopen(sprofadjNELO2, name="Rprof Example")
```

Input

```
#6 6
plotviz(sprofadjRag02, sub=as.character(sprof$info$id))
```

plotviz(sprofadjRag02, dot)

class: Ragraph
layout: dot

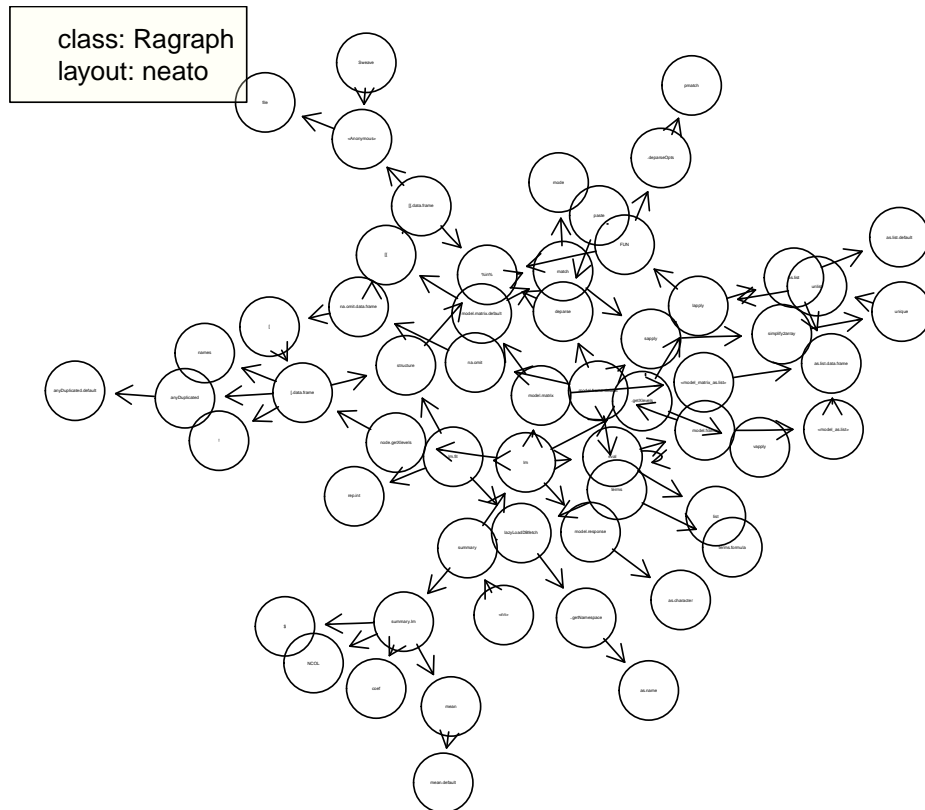


Input

#6

```
plotviz(x=sprofadjRag02, layout="neato")
```

plotviz(sprofadjRag02, neato)



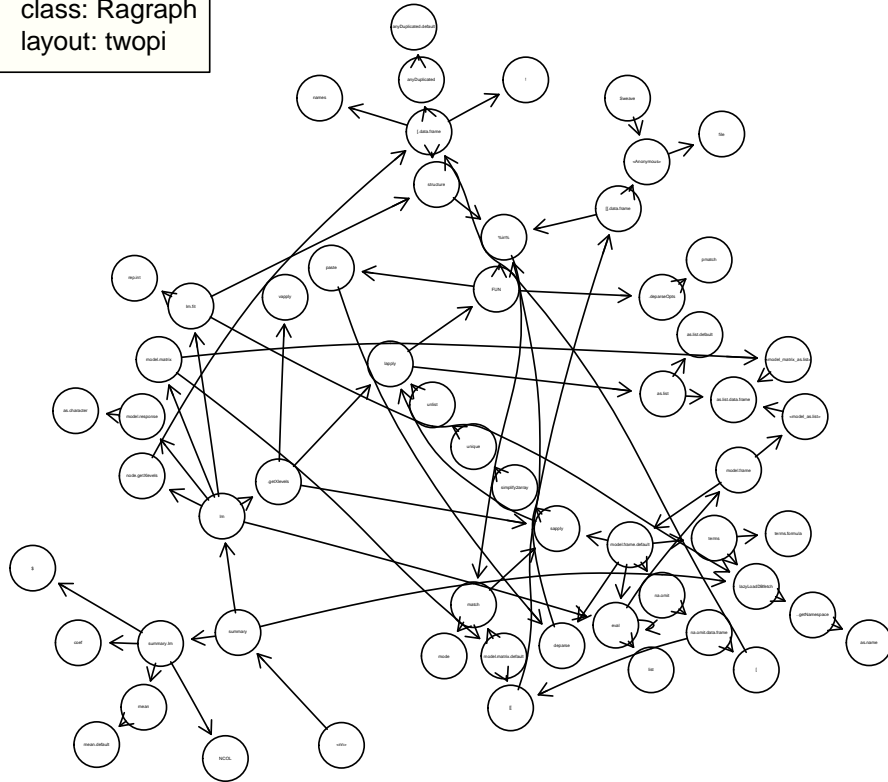
Input

#6

```
plotviz(sprofadjRag02, layout="twopi")
```

plotviz(sprofadjRag02, twopi)

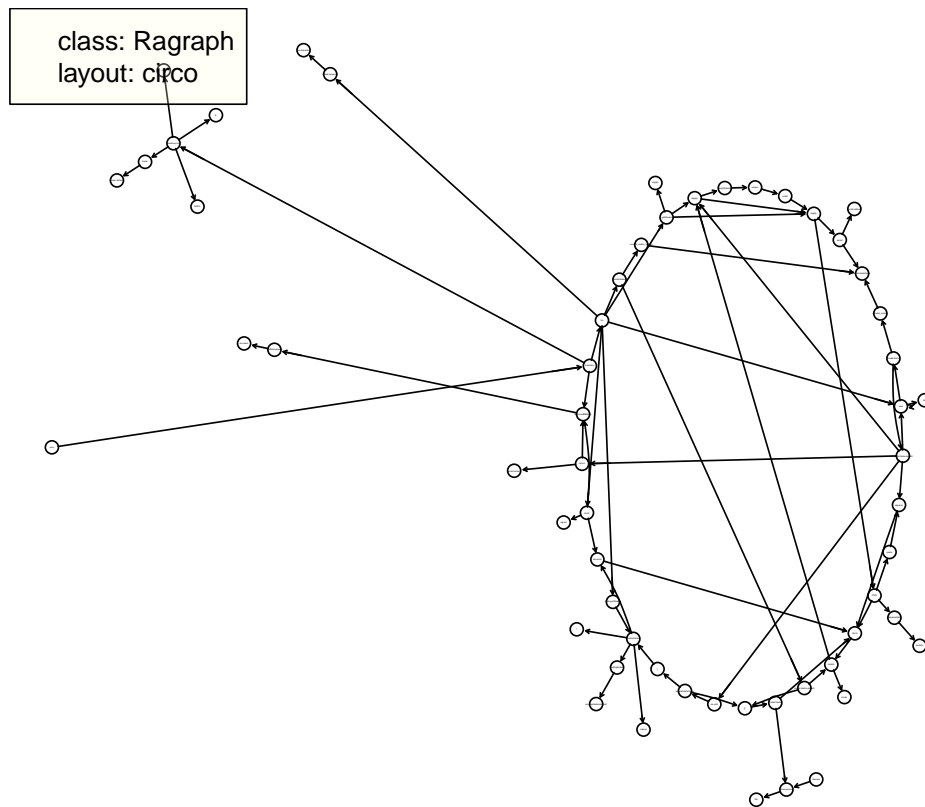
class: Ragraph
layout: twopi



Input

```
#6  
plotviz(sprofadjRag02, layout="circo")
```

plotviz(sprofadjRag02, circo)

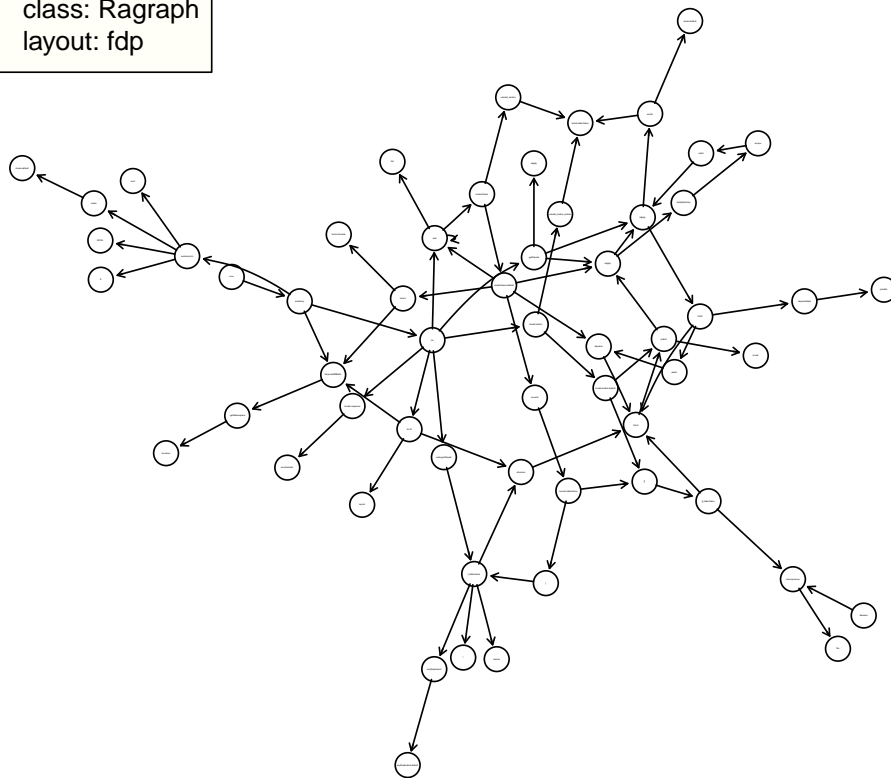


Input

#6

`plotviz(sprofadjRag02, layout="fdp")`**plotviz(sprofadjRag02, fdp)**

class: Ragraph
layout: fdp



GÜNTHER SAWITZKI
STATLAB HEIDELBERG
IM NEUENHEIMER FELD 294
D 69120 HEIDELBERG

E-mail address: gs@statlab.uni-heidelberg.de

URL: <http://sintro.r-forge.r-project.org/>