

## PENDING CHANGES

- 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.

## R PROFILING AND OPTIMISATION

GÜNTHER SAWITZKI

## CONTENTS

Pending changes	1
Profiling facilities in R	2
L <sup>A</sup> T <sub>E</sub> X Layout Tools and Utilities	2
1. Profiling	3
1.1. Simple regression example	4
1.1.1. R basic	5
1.1.2. Package sprof	8
2. A better grip on profile information	10
2.1. The details	13
3. xxx	15
3.0.1. Plot	26
3.1. analysis	29
3.2. trimming	30
4. Standard output	36
4.1. Summary	36
4.2. Print	40
4.3. Plot	44
5. Graph	47
5.1. graph Package	48
5.1.1. igraph Package	49
5.1.2. network Package	50
5.1.3. Rgraphviz Package	51
Index	56

---

*Date:* Typeset: July 8, 2013.

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

*An R vignette for package sprof. Private Version*

## PROFILING FACILITIES IN R

R provides the basic instruments for profiling, both for time based samplers as for event based instrumentation. 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<sup>A</sup>T<sub>E</sub>X LAYOUT TOOLS AND UTILITIES

Print parameters used here:

---

Input

---

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

We want immediate warnings, if necessary. Set to level 2 to handle warnings as error.

---

Input

---

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

For larger tabels and data frames, we use a cludge to avoid long outputs.

---

Input

---

```
xcutdata.frame <- function(df, cut, margin){
  #! keep3, to add: margin top - random center - margin bottom
  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){
    if (cutfrom<cutto){
      df[cutfrom,] <- NA
      if (!is.null(rownames(df))) rownames(df)[cutfrom] <- "< cut >"
      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 (!missing(margin)) {
      if (length(margin)==1) margin <- c(margin,margin)
      cut <- c(margin[1]+1,nrow-margin[2])
      df <-cutrng(cut[1],cut[2]);
      return(df)}
    #   if (!missing(keep3)) { cut <- c(keep3[1]+1, keep3[1]+1,
    #                                   nrow-keep3[3]-1,nrow-keep3[3]-1)
    #   if (cut[3]-cut[4] > keep3[2]+2){delta<-(cut[3]-cut[2]) div 2; cut[3]<-0
    #   browser()
    #   } else df <- cutrng(cut[1],cut[4])
    #   cutrng(cut[1],cut[4]) return(df)}
  }

```

We use the R function `xtable()` for output and L<sup>A</sup>T<sub>E</sub>X `longtable`. A convenient wrapper to use this in our *Sweave* source is:

---

```

library(xtable)
prxt <- function (x, digits=3,          caption=NULL,
                  label=NULL, ...) {
  margin <- 10
  if (nrow(x)> 2*margin+3) x <-xcutdata.frame(x, margin=margin)
  print(
    xtable(x, digits=digits, caption=caption, label=label, ...),
    floating=FALSE, tabular.environment="longtable", NA.string="\vdots")
  }

```

---

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

## 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 functions. Most notably, the `apply()` family appears as function calls and 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.

---

*Input*

---

```
options(error = recover)
library(sprof)
```

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.

### 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.

---

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 invites us to get a summary.

---

Input

---

```

sumRprofRegressionExpl <- summaryRprof("RprofsRegressionExpl01.out")
str(sumRprofRegressionExpl, vec.len=3)

```

---

Output

---

```

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.

---

*Input*

---

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

	self.time	self.pct	total.time	total.pct
"lm.fit"	0.087	16.670	0.113	21.650
"[.data.frame"	0.057	10.920	0.099	18.970
"model.matrix.default"	0.051	9.770	0.069	13.220
"as.character"	0.043	8.240	0.043	8.240
"lm"	0.042	8.050	0.474	90.800
"summary.lm"	0.040	7.660	0.045	8.620
"structure"	0.032	6.130	0.033	6.320
"na.omit.data.frame"	0.026	4.980	0.114	21.840
"anyDuplicated.default"	0.022	4.210	0.022	4.210
"as.list.data.frame"	0.022	4.210	0.022	4.210
< cut >	:	:	:	:
"FUN"	0.001	0.190	0.007	1.340
"%in%"	0.001	0.190	0.004	0.770
"deparse"	0.001	0.190	0.002	0.380
"\$"	0.001	0.190	0.001	0.190
"as.list.default"	0.001	0.190	0.001	0.190
"as.name"	0.001	0.190	0.001	0.190
"coef"	0.001	0.190	0.001	0.190
"file"	0.001	0.190	0.001	0.190
"NCOL"	0.001	0.190	0.001	0.190
"terms.formula"	0.001	0.190	0.001	0.190

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

---

*Input*

---

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

	total.time	total.pct	self.time	self.pct
"<Anonymous>"	0.522	100.000	0.006	1.150
"Sweave"	0.522	100.000	0.000	0.000
"eval"	0.521	99.810	0.001	0.190
"doTryCatch"	0.521	99.810	0.000	0.000
"evalFunc"	0.521	99.810	0.000	0.000
"try"	0.521	99.810	0.000	0.000
"tryCatch"	0.521	99.810	0.000	0.000
"tryCatchList"	0.521	99.810	0.000	0.000
"tryCatchOne"	0.521	99.810	0.000	0.000

"withVisible"	0.521	99.810	0.000	0.000
< cut >	:	:	:	:
"as.list"	0.023	4.410	0.000	0.000
"anyDuplicated.default"	0.022	4.210	0.022	4.210
"as.list.data.frame"	0.022	4.210	0.022	4.210
"sapply"	0.014	2.680	0.001	0.190
"match"	0.011	2.110	0.001	0.190
"[[.data.frame"	0.008	1.530	0.001	0.190
"["	0.008	1.530	0.000	0.000
"rep.int"	0.007	1.340	0.007	1.340
"FUN"	0.007	1.340	0.001	0.190
"list"	0.005	0.960	0.005	0.960

---

Table 2: summaryRprof result: by.total, total time > 0.004s

1.1.2. *Package sprof*. In contrast, in our implementation we take a two step approach. First we read in the profile file to an internal representation.

---

Input

---

```
sprof01lm <- readRprof("RprofsRegressionExpl01.out")
sprof01 <- sprof01lm
```

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

---

Input

---

```
save(sprof01lm, file="sprof01lm.RData")
```

To run the vignette with a different profile, replace *ircodesprof01* by your example. You still have *ircodesprof01lm* 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 *ircodeinfo* 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.

---

Input

---

```
str(sprof01, max.level=2, vec.len=3, nchar.max=40)
```

---

Output

---

List of 4

```
$ info      : 'data.frame':      1 obs. of  8 variables:
..$ id       : Factor w/ 1 level "\RprofsRegressionExpl01.out\" 2013-06-\" | __truncated__: 1
..$ date     : POSIXct[1:1], format: "2013-07-08 18:02:17"
..$ nrnodes  : int 62
..$ nrstacks : int 50
..$ nrrecords: int 522
..$ firstline: Factor w/ 1 level "sample.interval=1000": 1
..$ ctllines : Factor w/ 1 level "sample.interval=1000": 1
..$ 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
..$ shortname : Factor w/ 50 levels "S<A>eFttCtCLtC0dTCwVeesleem.m..n.n...[\" | __truncated__,...: 2
```



```

..$ 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__,...: 2
$ 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 3 uses a permutation by total time, and a selection (compare to table 2 on page 7).

---

```

                                Input
nodes <- sprof01$nodes[order(sprof01$nodes$total.time, decreasing=TRUE),]
prxt(nodes[nodes$total.time>4,],
caption="splot result: by.total, total time > 0.004s",
      label="tab:prspbt")

```

---

	name	self.time	self.pct	total.time	total.pct
	10 <Anonymous>	6.000	1.150	522.000	6.790
	52 Sweave	0.000	0.000	522.000	6.790
	21 doTryCatch	0.000	0.000	521.000	6.780
	22 eval	1.000	0.190	521.000	6.780
	23 evalFunc	0.000	0.000	521.000	6.780
	55 try	0.000	0.000	521.000	6.780
	56 tryCatch	0.000	0.000	521.000	6.780
	57 tryCatchList	0.000	0.000	521.000	6.780
	58 tryCatchOne	0.000	0.000	521.000	6.780
	62 withVisible	0.000	0.000	521.000	6.780
< cut >	\vdots	⋮	⋮	⋮	⋮
	61 vapply	3.000	0.570	23.000	0.300
	13 anyDuplicated.default	22.000	4.210	22.000	0.290
	16 as.list.data.frame	22.000	4.210	22.000	0.290
	47 sapply	1.000	0.190	14.000	0.180
	31 match	1.000	0.190	11.000	0.140
	7 [[	0.000	0.000	8.000	0.100
	8 [[.data.frame	1.000	0.190	8.000	0.100
	25 FUN	1.000	0.190	7.000	0.090
	46 rep.int	7.000	1.340	7.000	0.090
	28 list	5.000	0.960	5.000	0.070

Table 3: splot result: by.total, total time > 0.004s

---

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

Input



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. To illustrate this, we encode the number the nodes by an arbitrary order and encode this by colour. Of course this is a poor information, and we use rainbow colours, which is about the poorest of all colour encodings. Just for demonstration.

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

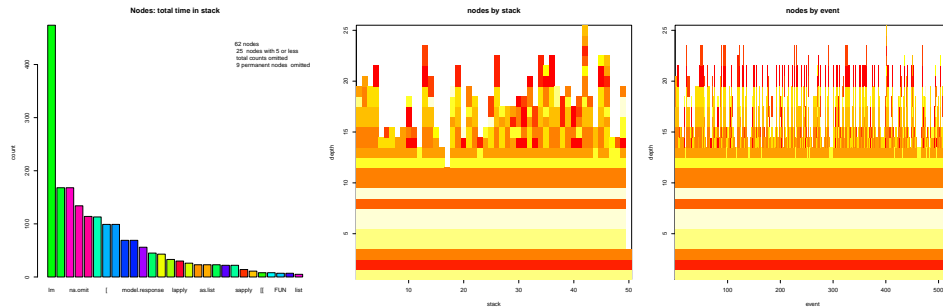
---

```

                                Input
oldpar <- par(mfrow=c(1,3))
#colp <- rainbow(sprof01$info$nrnodes)

#plot_nodes # 3
xnodes <- sprof01$nodes
src <- sprof01$info$src
mincount <- 5
    nrnodes <- dim(xnodes)[1]
    totaltime <- sum(xnodes$self.time)
colp <- rainbow(nrnodes)
colp <- colp[order(xnodes$total.time,decreasing=TRUE)]
    xnodes$colp <- colp
    xnodes <- xnodes[xnodes$total.time < totaltime,]
    #browser()
    if (mincount>0) xnodes <- xnodes[xnodes$total.time>=mincount,]
    trimmed <- nrnodes-dim(xnodes)[1]
    legnd <-function(trimmed=0, fulltime=0){
        ltext <- paste(nrnodes,"nodes\n")
        if (trimmed>0) ltext <- paste(ltext,trimmed," nodes with",mincount,"or less\n total co
        if (fulltime>0) ltext <- paste(ltext,fulltime,"permanent nodes omitted\n")
        legend("topright", legend=ltext, bty="n")
    }
totaltime <- sum(xnodes$self.time)
        fulltime <- dim(xnodes)[1]
        xnodes <- xnodes[xnodes$total.time < totaltime,]
        fulltime <- fulltime - dim(xnodes)[1]
        ordertotal<- order(xnodes$total.time,decreasing=TRUE);
        barplot(xnodes[ordertotal,]$total.time,
        main="Nodes: total time in stack",
        names.arg = xnodes[ordertotal,]$name, sub=src, ylab="count",
        col=xnodes$colp);
        legnd(trimmed=trimmed, fulltime=fulltime)
stacks_nodes <- list.as.matrix(sprof01$stacks$nodes)
#<<fig=TRUE, label=sREimgstacks>>=
image(x=1:ncol(stacks_nodes),y=1:nrow(stacks_nodes),
t(stacks_nodes),xlab="stack", ylab="depth", main="nodes by stack")
profile_nodes <- profiles_matrix(sprof01)
#<<fig=TRUE, label=sREimgprofiles>>=
image(x=1:ncol(profile_nodes),y=1:nrow(profile_nodes),
t(profile_nodes),xlab="event", ylab="depth", main="nodes by event")
par(oldpar)

```



The obvious message is that if seen by 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) resistant. Then in general we have some few steps where the algorithm splits, and then we have the finer details. This can be identified on the stack level.

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.

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 than as part of the general dynamics.

Stable framework effect can be detected automatically. “burn in” or “fade out” may need a closer look, and special stacks need an individual inspection on low frequency stacks.

Before starting additional inspection, the data better be trimmed. 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.

---

#### Input

---

```
sprof02 <- sprof01
basetrim <- 13
sprof02$stacks$nodes <- sapply(sprof02$stacks$nodes, function(x){if (length(x)> basetrim) x[-(1:basetrim)]
#sprof02$stacks$nodes <- sprof02$stacks$nodes[!is.null(sprof02$stacks$nodes)]
```

---

#### Input

---

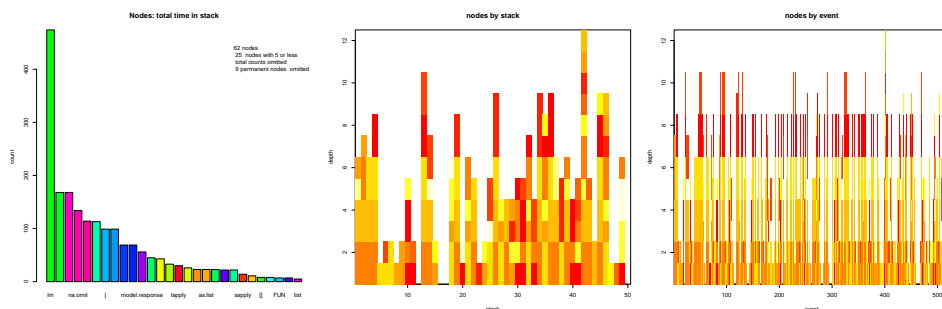
```
oldpar <- par(mfrow=c(1,3))
#colp <- rainbow(sprof02$info$nrnodes)

#plot_nodes # 3
xnodes <- sprof02$nodes
src <- sprof02$info$src
mincount <- 5
  nrnodes <- dim(xnodes)[1]
  totaltime <- sum(xnodes$self.time)
colp <- rainbow(nrnodes)
colp <- colp[order(xnodes$total.time,decreasing=TRUE)]
  xnodes$colp <- colp
  xnodes <- xnodes[xnodes$total.time < totaltime,]
#browser()
```

```

if (mincount>0) xnodes <- xnodes[xnodes$total.time>=mincount,]
trimmed <- nrmnodes-dim(xnodes)[1]
legnd <-function(trimmed=0, fulltime=0){
  ltext <- paste(nrmnodes,"nodes\n")
  if (trimmed>0) ltext <- paste(ltext,trimmed," nodes with",mincount,"or less\n total co
  if (fulltime>0) ltext <- paste(ltext,fulltime,"permanent nodes omitted\n")
  legend("topright", legend=ltext, bty="n")
}
totaltime <- sum(xnodes$self.time)
fulltime <- dim(xnodes)[1]
xnodes <- xnodes[xnodes$total.time < totaltime,]
fulltime <- fulltime - dim(xnodes)[1]
ordertotal<- order(xnodes$total.time,decreasing=TRUE);
barplot(xnodes[ordertotal,]$total.time,
main="Nodes: total time in stack",
names.arg = xnodes[ordertotal,]$name, sub=src, ylab="count",
col=xnodes$colp);
legnd(trimmed=trimmed, fulltime=fulltime)
stacks_nodes <- list.as.matrix(sprof02$stacks$nodes)
#<<fig=TRUE, label=sREimgstacks>>=
image(x=1:ncol(stacks_nodes),y=1:nrow(stacks_nodes),
t(stacks_nodes),xlab="stack", ylab="depth", main="nodes by stack")
profile_nodes <- profiles_matrix(sprof02)
#<<fig=TRUE, label=sREimgprofiles>>=
image(x=1:ncol(profile_nodes),y=1:nrow(profile_nodes),
t(profile_nodes),xlab="event", ylab="depth", main="nodes by event")
par(oldpar)

```



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 organized 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 a recursive version of `rle`, applied to the profile information.

---

Input

```
profile_nodes_rle<- rrle(profile_nodes)
```

**2.1. The 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.

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, 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 loose 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).

There have always been interrupts, and these show up in profiles. In R, this is related problem (GC)

The graph is a second instance that is (re)constructed from the stack recording.

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.

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.

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 consist 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.

3. xxx

---

*Input*

```
# xtable cannot handle posix
str(sprof01$info,
     caption="splot node info",
     label="tab:prSREinfo0")
```

---

*Output*

```
'data.frame':      1 obs. of  8 variables:
 $ id      : Factor w/ 1 level "\"RprofsRegressionExpl01.out\" 2013-06-13 23:46:04": 1
 $ date    : POSIXct, format: "2013-07-08 18:02:17"
 $ nrnodes : int 62
 $ nrstacks: int 50
 $ nrrecords: int 522
 $ firstline: Factor w/ 1 level "sample.interval=1000": 1
 $ ctllines : Factor w/ 1 level "sample.interval=1000": 1
 $ ctllinenr: num 1
```

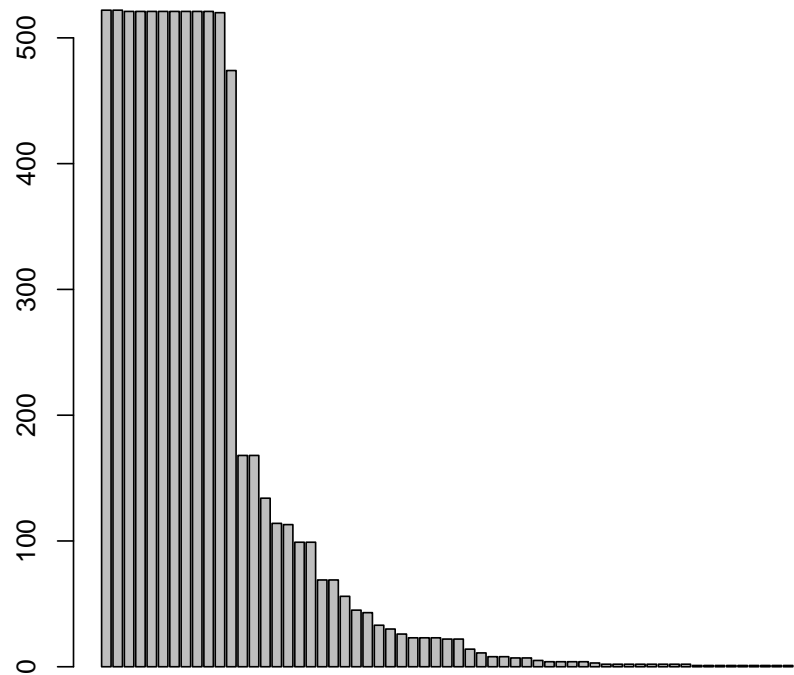
As a convention, we do not re-arrange items for ad-hoc choices, but use a permutation vector instead.

---

*Input*

```
rownames(sprof01$nodes) <- sprof01$nodes$names
nodesperm <- order(sprof01$nodes$total.time,decreasing=TRUE)
barplot(sprof01$nodes$total.time[nodesperm])
```





Selections are recorded as selection vectors, with reference to the original order. This needs some caution to align them with the order choices.

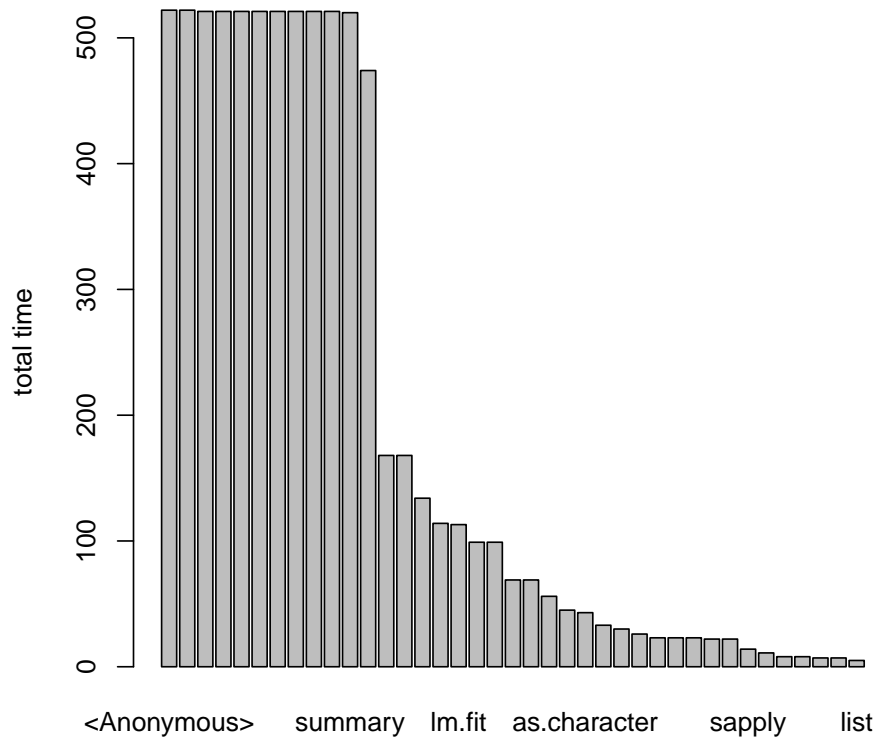
---

```

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")

```

### Nodes, by total time



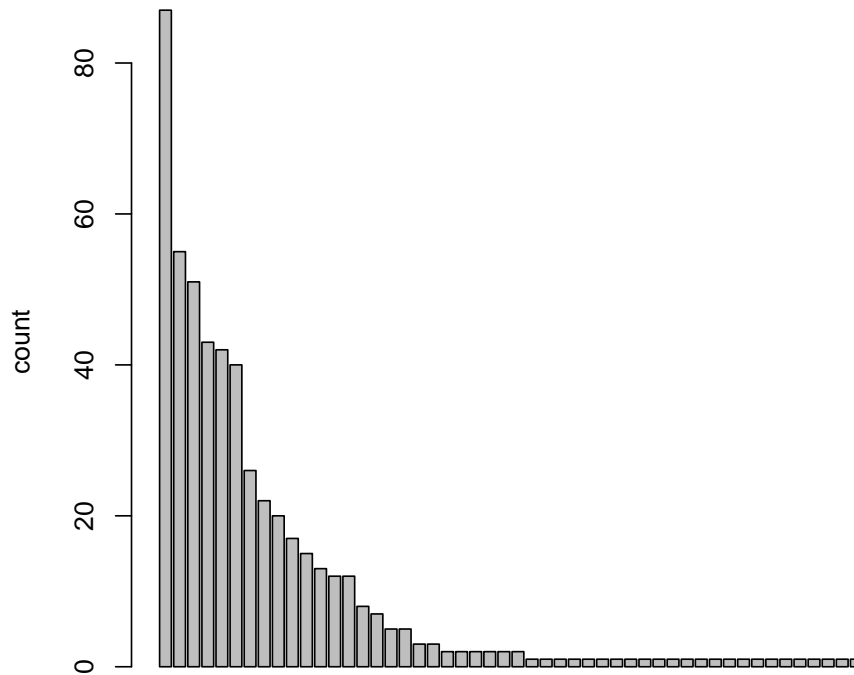

---

```

#rownames(sprof01$nodes) <- sprof01$nodes$names
stacksperm <- order(sprof01$stacks$refcount,decreasing=TRUE)
barplot(sprof01$stacks$refcount[stacksperm],main="Stacks, by reference count", ylab="count")

```

### Stacks, by reference count




---

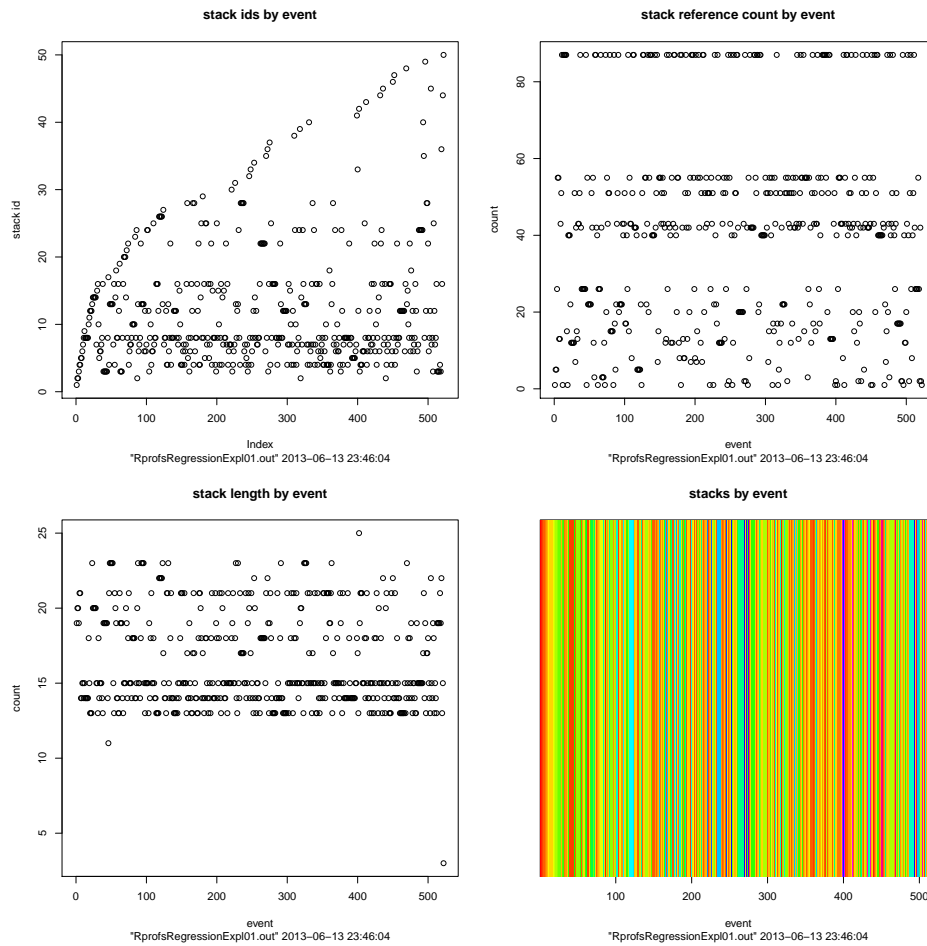
*Input*

```
stacksnrobsok <- sprof01$stacks$refcount > 4
sp4 <- sprof01$stacks$refcount[stacksperm][stacksnrobsok[stacksperm]]
names(sp4) <- sprof01$stacks$shortname[stacksperm][stacksnrobsok[stacksperm]]
barplot(sp4,
  main="Stacks, by reference count (4 obs. minimum)", ylab="count > 4")
```

---

```
oldpar<-par(mfrow=c(2,2))
plot_profiles(sprof01)
par(oldpar)
```

*Input*

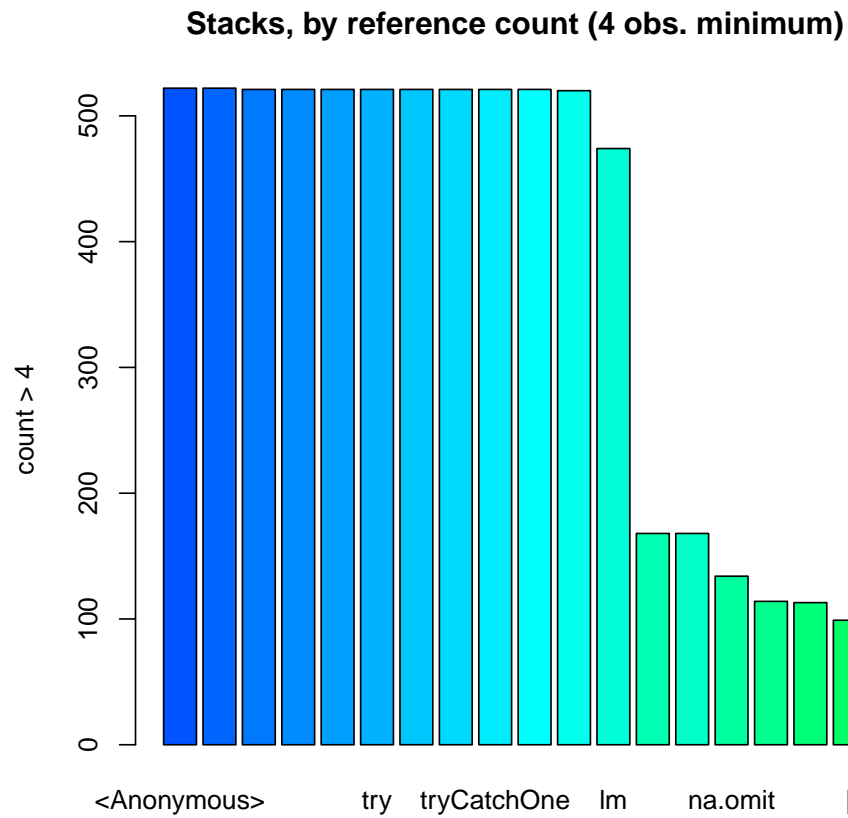


We now do a step down analysis. Aggregating the information from the profiling events, we have the frequency of stack references. On the stack level, we encode the frequency in colour, and linking propagates this to the profile level.

---

Input

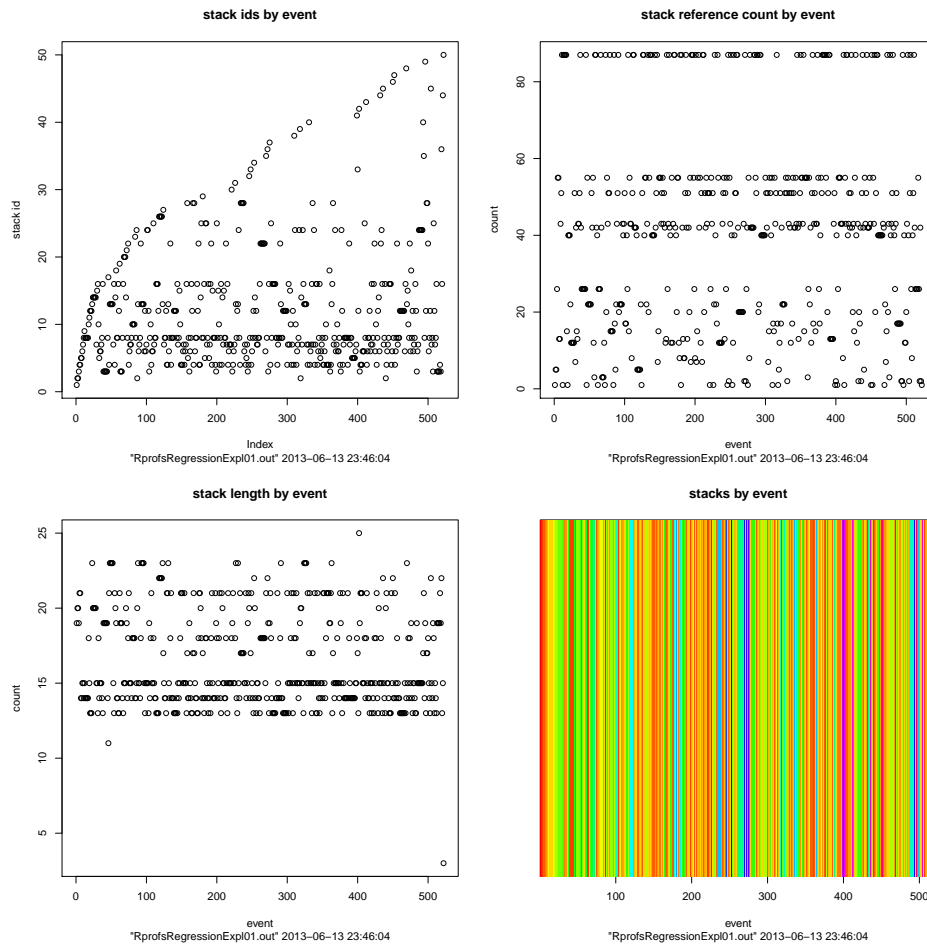
```
stackfreqscore <- rank(sprof01$stacks$refcount,ties.method="random")
stackfreqscore4<- stackfreqscore[stacksperm][stacksnrobsok[stacksperm]]
barplot(sp[stacksnrobsok[stacksperm]], main="Stacks, by reference count (4 obs. minimum)", ylab="count",
col=rainbow(80)[stackfreqscore4])
```



---

```
oldpar<- par(mfrow=c(2,2))  
plot_profiles(sprof01)  
par(oldpar)
```

Input



---

*Input*

---

```
prxt(sprof01$nodes,
     caption="nodes",
     label="tab:prSREnodes")
```

	name	self.time	self.pct	total.time	total.pct
1	!	2.000	0.380	2.000	0.030
2	..getNamespace	0.000	0.000	1.000	0.010
3	.deparseOpts	2.000	0.380	4.000	0.050
4	.getXlevels	0.000	0.000	26.000	0.340
5	[	0.000	0.000	99.000	1.290
6	[.data.frame	57.000	10.920	99.000	1.290
7	[[	0.000	0.000	8.000	0.100
8	[[.data.frame	1.000	0.190	8.000	0.100
9	%in%	1.000	0.190	4.000	0.050
10	<Anonymous>	6.000	1.150	522.000	6.790
< cut >	\vdots	:	:	:	:
53	terms	0.000	0.000	2.000	0.030
54	terms.formula	1.000	0.190	1.000	0.010
55	try	0.000	0.000	521.000	6.780
56	tryCatch	0.000	0.000	521.000	6.780
57	tryCatchList	0.000	0.000	521.000	6.780
58	tryCatchOne	0.000	0.000	521.000	6.780
59	unique	3.000	0.570	4.000	0.050
60	unlist	0.000	0.000	1.000	0.010
61	vapply	3.000	0.570	23.000	0.300
62	withVisible	0.000	0.000	521.000	6.780

Table 4: nodes



---

Input

---

```
str(sprof01$stacks, max.level=1)
```

---

Output

---

```
'data.frame':      50 obs. of  7 variables:
 $ nodes      :List of 50
 $ shortname   : Factor w/ 50 levels "S<A>eFttCtCLtC0dTCwVeesleem.m..n.n...[.",...: 27 17 19 1 35 36 3
 $ refcount    : num  1 5 26 55 13 43 51 87 1 15 ...
 $ stacklength : int  19 20 19 21 14 15 15 14 15 18 ...
 $ stackheadnodes: int  52 52 52 52 52 52 52 52 52 52 ...
 $ stackleafnodes: int  27 28 41 6 39 14 38 30 27 49 ...
 $ stackssrc    : Factor w/ 50 levels "! [.data.frame [ na.omit.data.frame na.omit model.frame.default
```

---

Input

---



---

Input

---

```
str(sprof01$profiles, max.level=1)
```

---

Output

---

```
List of 4
 $ data      : int [1:522] 1 2 2 3 4 4 5 5 6 7 ...
 $ mem       : NULL
 $ malloc    : NULL
 $ timesRLE:List of 2
  ..- attr(*, "class")= chr "rle"
```

---

Input

---

A summary is provided on request.

---

Input

---

```
sumsprof01 <- summary.sprof(sprof01)
str(sumsprof01, max.level=2)
```

---

Output

---

```
List of 4
 $ info      :'data.frame':      1 obs. of  8 variables:
  ..$ id      : Factor w/ 1 level "\"RprofsRegressionExpl01.out\"": 1
  ..$ date     : POSIXct[1:1], format: "2013-07-08 18:02:17"
  ..$ nrnodes  : int 62
  ..$ nrstacks : int 50
  ..$ nrrecords: int 522
  ..$ firstline: Factor w/ 1 level "sample.interval=1000": 1
  ..$ cttl原因 : Factor w/ 1 level "sample.interval=1000": 1
  ..$ cttl原因nr: num 1
 $ nodes     :'data.frame':      62 obs. of  5 variables:
  ..$ name     : Factor w/ 62 levels "!", "..getNamespace",...: 1 2 3 4 5 6 7 8 9 10 ...
  ..$ self.time : num [1:62] 2 0 2 0 0 57 0 1 1 6 ...
  ..$ self.pct  : num [1:62] 0.38 0 0.38 0 0 ...
  ..$ total.time: num [1:62] 2 1 4 26 99 99 8 8 4 522 ...
  ..$ total.pct : num [1:62] 0.03 0.01 0.05 0.34 1.29 1.29 0.1 0.1 0.05 6.79 ...
 $ stacks    :'data.frame':      50 obs. of  7 variables:
```

```

..$ nodes      :List of 50
..$ shortname   : Factor w/ 50 levels "S<A>eFttCtCLtCOdTCwVeesleem.m..n.n...[.",...: 27 17 19 1 35 ...
..$ refcount    : num [1:50] 1 5 26 55 13 43 51 87 1 15 ...
..$ stacklength : int [1:50] 19 20 19 21 14 15 15 14 15 18 ...
..$ stackheadnodes: int [1:50] 52 52 52 52 52 52 52 52 52 52 ...
..$ stackleafnodes: int [1:50] 27 28 41 6 39 14 38 30 27 49 ...
..$ stackssrc   : Factor w/ 50 levels "!" [.data.frame [ na.omit.data.frame na.omit model.frame.default]
$ profiles:List of 4
..$ data       : int [1:522] 1 2 2 3 4 4 5 5 6 7 ...
..$ mem        : NULL
..$ malloc     : NULL
..$ timesRLE:List of 2
.. ..- attr(*, "class")= chr "rle"
- attr(*, "class")= chr [1:2] "sprof" "list"

```

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.

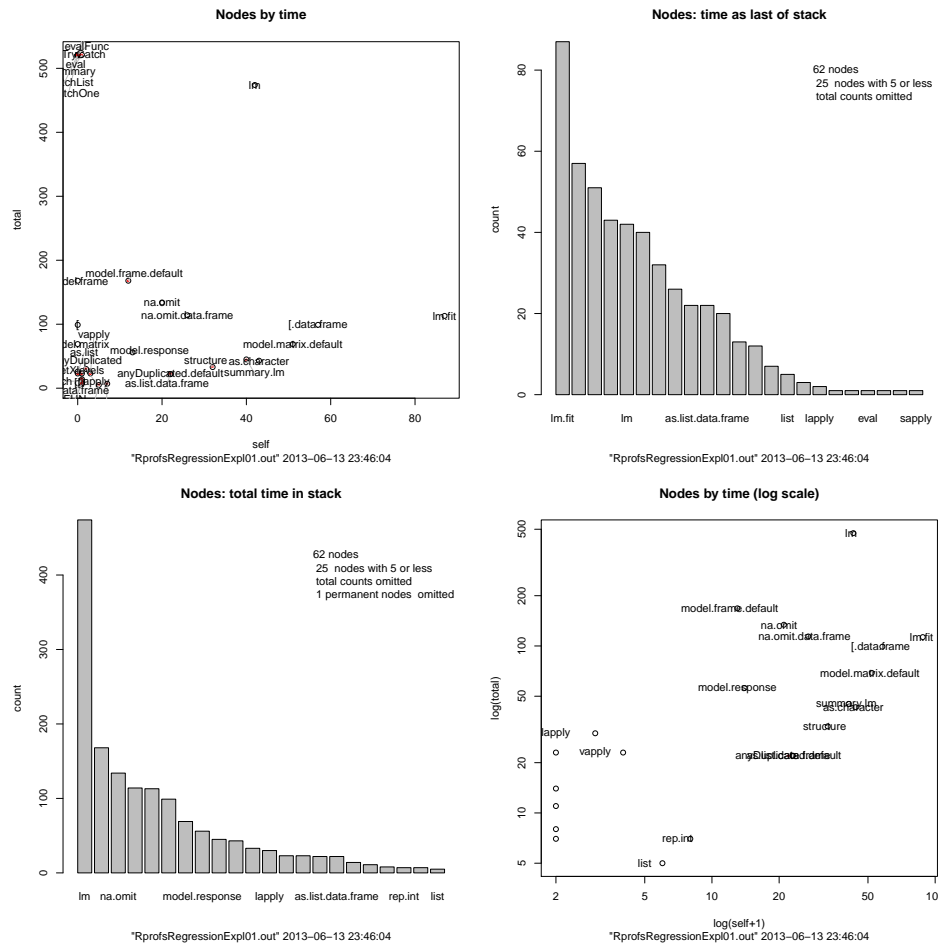
3.0.1. *Plot*. Looking at lists of numbers is not too informative. We get a first impression by plotting the data.

---

```

#plot_nodes(sprof01, col=nodescol[nodescore])
oldpar <- par(mfrow=c(2,2))
plot_nodes(sprof01)
par(oldpar)

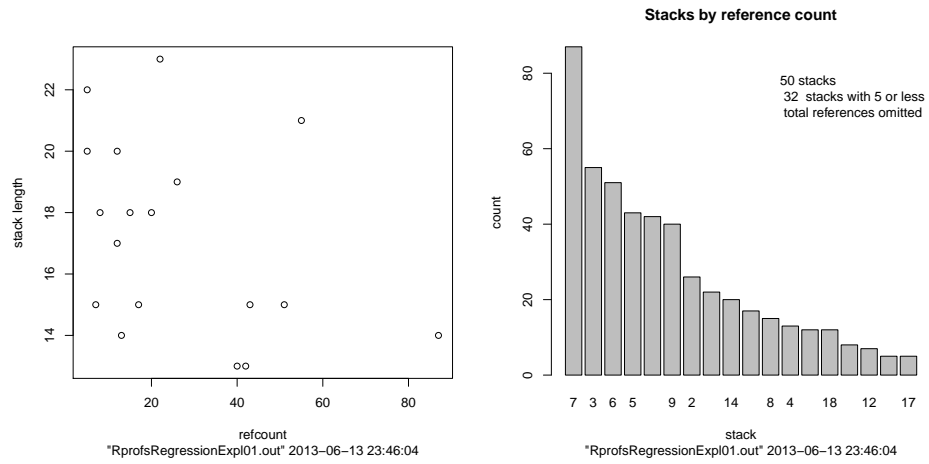
```




---

*Input*

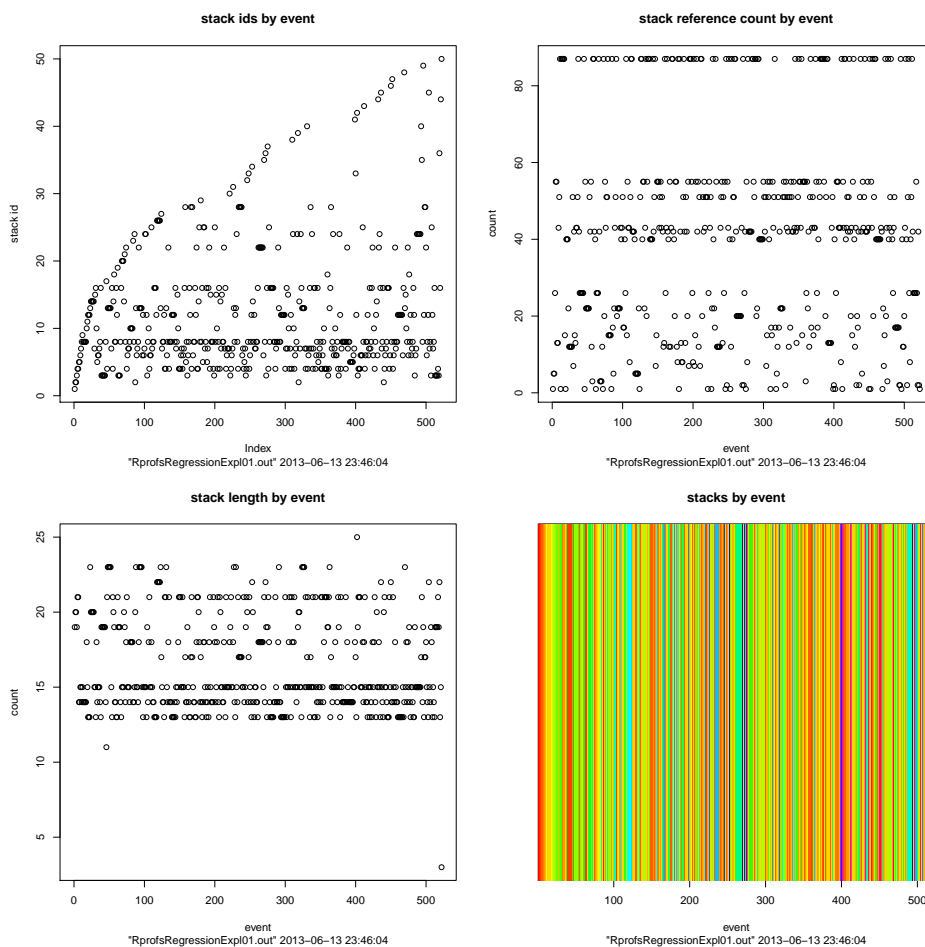
```
oldpar <- par(mfrow=c(1,2))
plot_stacks(sprof01)
par(oldpar)
```




---

*Input*

```
oldpar <-par(mfrow=c(2,2))
plot_profiles(sprof01)
par(oldpar)
```



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

### 3.1. analysis.

---

Input

```
profile_nodes_rrle <- rrle(profile_nodes)
str(profile_nodes_rrle)
```

---

Output

---

```
List of 12
 $ :List of 2
  ..$ lengths: int [1:361] 6 3 1 7 1 1 1 1 1 6 ...
  ..$ values : int [1:361] 22 39 37 30 4 2 NA NA NA 22 ...
  ..- attr(*, "class")= chr "rrle"
 $ :List of 2
  ..$ lengths: int [1:407] 6 1 1 1 1 1 1 1 1 1 ...
  ..$ values : int [1:407] 22 NA NA 14 38 NA 27 NA NA NA ...
  ..- attr(*, "class")= chr "rrle"
 $ :List of 2
  ..$ lengths: int [1:427] 6 1 1 1 1 1 1 1 1 1 ...
```

```

..$ values : int [1:427] 35 NA NA NA NA NA NA NA NA NA ...
..- attr(*, "class")= chr "rle"
$:List of 2
..$ lengths: int [1:427] 6 1 1 1 1 1 1 1 1 ...
..$ values : int [1:427] 36 NA NA NA NA NA NA NA NA NA ...
..- attr(*, "class")= chr "rle"
$:List of 2
..$ lengths: int [1:450] 1 2 3 1 1 1 1 1 1 ...
..$ values : int [1:450] 53 22 40 NA NA NA NA NA NA NA ...
..- attr(*, "class")= chr "rle"
$:List of 2
..$ lengths: int [1:466] 1 2 3 1 1 1 1 1 1 ...
..$ values : int [1:466] 27 22 41 NA NA NA NA NA NA NA ...
..- attr(*, "class")= chr "rle"
$:List of 2
..$ lengths: int [1:489] 1 2 1 2 1 1 1 1 1 ...
..$ values : int [1:489] NA 28 NA 5 NA NA NA NA NA NA ...
..- attr(*, "class")= chr "rle"
$:List of 2
..$ lengths: int [1:494] 1 1 1 1 2 1 1 1 1 ...
..$ values : int [1:494] NA NA NA NA 6 NA NA NA NA NA ...
..- attr(*, "class")= chr "rle"
$:List of 2
..$ lengths: int [1:508] 1 1 1 1 1 1 1 1 1 ...
..$ values : int [1:508] NA NA NA NA NA NA NA NA NA ...
..- attr(*, "class")= chr "rle"
$:List of 2
..$ lengths: int [1:512] 1 1 1 1 1 1 1 1 1 ...
..$ values : int [1:512] NA NA NA NA NA NA NA NA NA ...
..- attr(*, "class")= chr "rle"
$:List of 2
..$ lengths: int [1:522] 1 1 1 1 1 1 1 1 1 ...
..$ values : int [1:522] NA NA NA NA NA NA NA NA NA ...
..- attr(*, "class")= chr "rle"
$:List of 2
..$ lengths: int [1:522] 1 1 1 1 1 1 1 1 1 ...
..$ values : int [1:522] NA NA NA NA NA NA NA NA NA ...
..- attr(*, "class")= chr "rle"

```

### 3.2. trimming.

---

```

                                Input
trimstacks <- function(sprof, level){
  lapply(sprof$stacks$nodes, function(x) {x[-(1:level)]})
}

```

---

```

                                Input
sprof01Tr <- trimstacks(sprof01, 11)
#profile_nodesTr <- profiles_matrix(sprof01Tr)
#image(x=1:ncol(profile_nodesTr),y=1:nrow(profile_nodesTr), t(profile_nodesTr),xlab="event", ylab="dep

```

---

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.

---

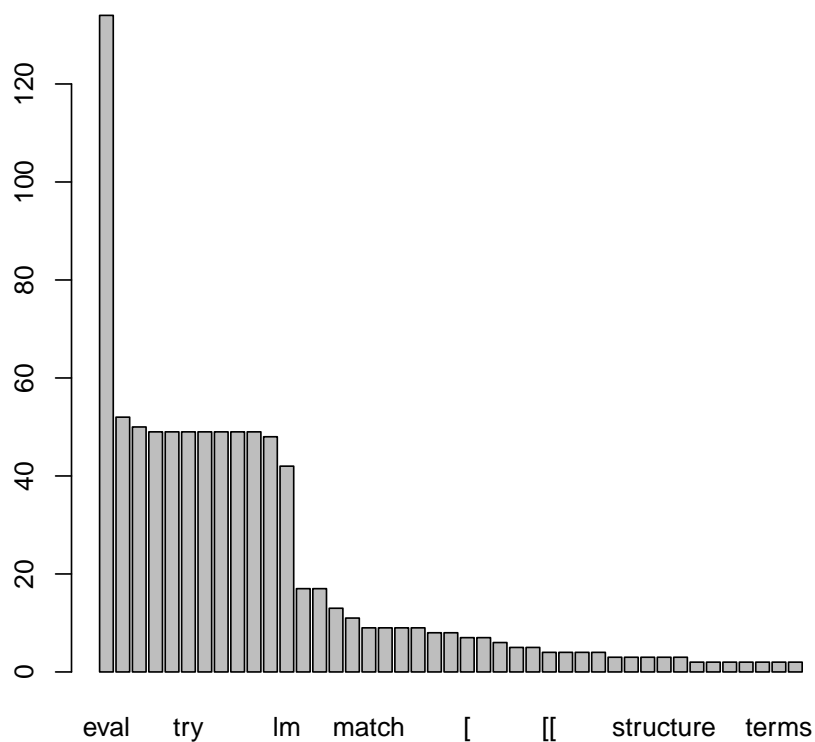
Input

---

```

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

```




---

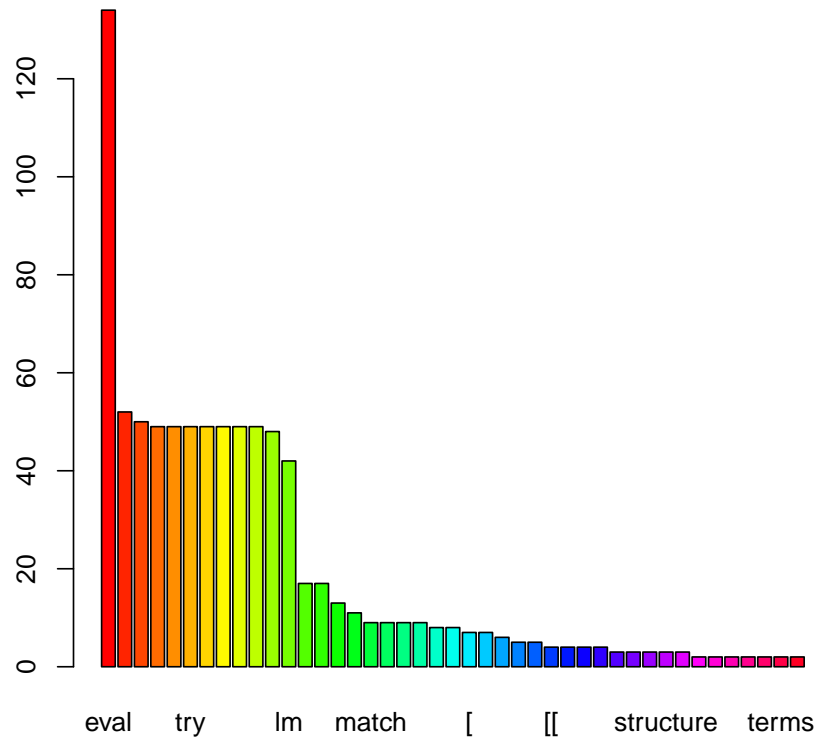
Input

---

```

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

```



Top frequent stacks.

---

Input

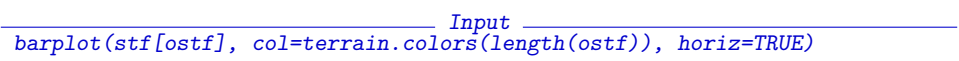
---

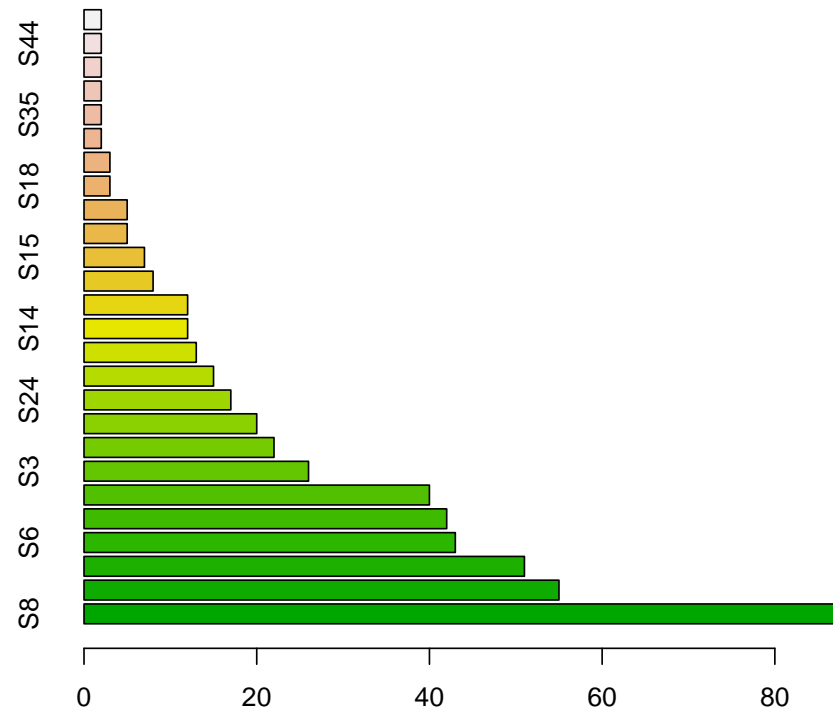
```

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])

```

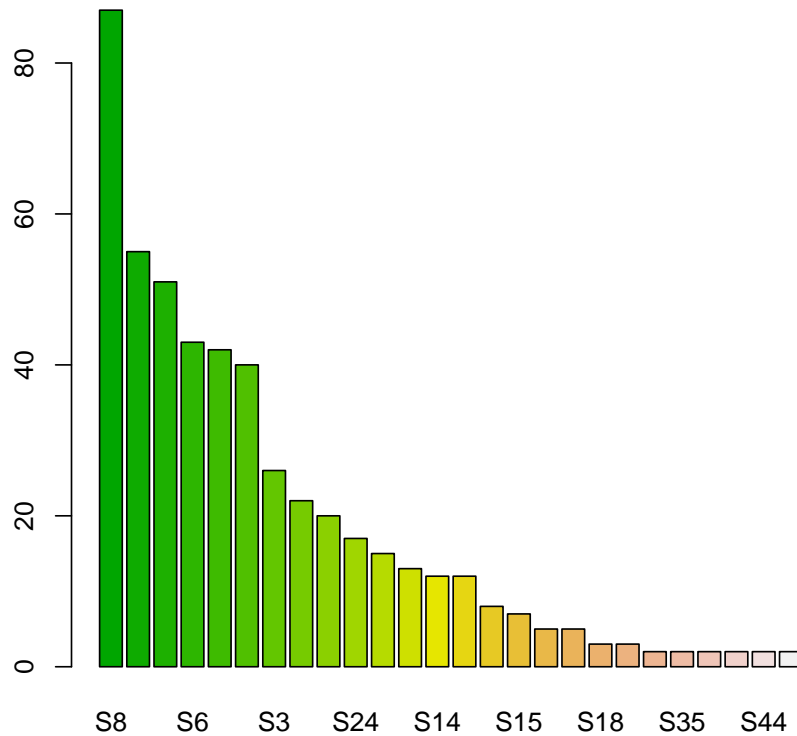






---

*Input*  
`barplot(stf[ostf], col=terrain.colors(length(ostf)))`



There is no statistics on profiles. Profiling 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.

#### 4. STANDARD OUTPUT

##### 4.1. Summary.

Input						
<code>summary_nodes(rpo)</code>						
		Output		self.time	self.pct	total.time
shortname	root	leaf				
<code>.deparseOpts</code>	<code>.dp0</code>	-	LEAF	1	Inf	0
<code>.getXlevels</code>	<code>.gtX</code>	-	-	0	NaN	0
<code>.row_names_info</code>	<code>.r__</code>	-	LEAF	1	Inf	0

[	[	-	-	0	NaN	0
[.data.frame	[.d.	-	LEAF	48	Inf	0
[[	[[	-	-	0	NaN	0
[[.data.frame	[[..	-	LEAF	2	Inf	0
%in%	%in%	-	LEAF	1	Inf	0
<Anonymous>	<An>	-	-	0	NaN	0
\$	\$	-	LEAF	2	Inf	0
anyDuplicated	anyD	-	LEAF	1	Inf	0
anyDuplicated.default	anD.	-	LEAF	9	Inf	0
as.character	as.c	-	LEAF	35	Inf	0
as.list	as.l	-	LEAF	1	Inf	0
as.list.data.frame	a...	-	LEAF	33	Inf	0
as.vector	as.v	-	-	0	NaN	0
c	c	-	LEAF	1	Inf	0
cat	cat	-	LEAF	1	Inf	0
chol2inv	chl2	-	LEAF	2	Inf	0
deparse	dprs	-	-	0	NaN	0
doTryCatch	dTrC	-	-	0	NaN	0
eval	eval	-	-	0	NaN	0
evalFunc	evlF	-	-	0	NaN	0
FUN	FUN	-	LEAF	1	Inf	0
getOption	gtOp	-	-	0	NaN	0
lapply	lppl	-	-	0	NaN	0
lazyLoadDBfetch	lLDB	-	LEAF	7	Inf	0
list	list	-	LEAF	14	Inf	0
lm	lm	-	LEAF	43	Inf	0
lm.fit	lm.f	-	LEAF	114	Inf	0
match	mtch	-	LEAF	2	Inf	0
match.fun	mtc.	-	-	0	NaN	0
mean	mean	-	-	0	NaN	0
mean.default	mn.d	-	LEAF	1	Inf	0
mode	mode	-	LEAF	1	Inf	0
model.frame	mdl.f	-	-	0	NaN	0
model.frame.default	mdl.f.	-	LEAF	16	Inf	0
model.matrix	mdl.m	-	-	0	NaN	0
model.matrix.default	mdl.m.	-	LEAF	55	Inf	0
model.response	mdl.r	-	LEAF	2	Inf	0
na.omit	n.mt	-	LEAF	15	Inf	0
na.omit.data.frame	n...	-	LEAF	23	Inf	0
names	nams	-	LEAF	1	Inf	0
options	optn	-	LEAF	1	Inf	0
paste	past	-	-	0	NaN	0
pmatch	pmtc	-	LEAF	1	Inf	0
rep.int	rp.n	-	LEAF	1	Inf	0
sapply	sppl	-	-	0	NaN	0
simplify2array	smp2	-	-	0	NaN	0
structure	strc	-	LEAF	33	Inf	0
summary	smmr	-	-	0	NaN	0
summary.lm	smm.	-	LEAF	36	Inf	0
Sweave	Swev	ROOT	-	0	NaN	0
terms	trms	-	-	0	NaN	0
terms.formula	trm.	-	LEAF	1	Inf	0
try	try	-	-	0	NaN	0

tryCatch	tryC	-	-	0	NaN	0
tryCatchList	trCL	-	-	0	NaN	0
tryCatchOne	trCO	-	-	0	NaN	0
unique	uniq	-	-	0	NaN	0
unique.default	unq.	-	LEAF	1	Inf	0
unlist	unls	-	LEAF	1	Inf	0
vapply	vppl	-	-	0	NaN	0
withVisible	wthV	-	-	0	NaN	0
	total.pct					
.deparseOpts	NaN					
.getXlevels	NaN					
.row_names_info	NaN					
[	NaN					
[/data.frame	NaN					
[[	NaN					
[[/data.frame	NaN					
%in%	NaN					
<Anonymous>	NaN					
\$	NaN					
anyDuplicated	NaN					
anyDuplicated.default	NaN					
as.character	NaN					
as.list	NaN					
as.list.data.frame	NaN					
as.vector	NaN					
c	NaN					
cat	NaN					
chol2inv	NaN					
deparse	NaN					
doTryCatch	NaN					
eval	NaN					
evalFunc	NaN					
FUN	NaN					
getOption	NaN					
lapply	NaN					
lazyLoadDBfetch	NaN					
list	NaN					
lm	NaN					
lm.fit	NaN					
match	NaN					
match.fun	NaN					
mean	NaN					
mean.default	NaN					
mode	NaN					
model.frame	NaN					
model.frame.default	NaN					
model.matrix	NaN					
model.matrix.default	NaN					
model.response	NaN					
na.omit	NaN					
na.omit.data.frame	NaN					
names	NaN					
options	NaN					

paste	NaN
pmatch	NaN
rep.int	NaN
sapply	NaN
simplify2array	NaN
structure	NaN
summary	NaN
summary.lm	NaN
Sweave	NaN
terms	NaN
terms.formula	NaN
try	NaN
tryCatch	NaN
tryCatchList	NaN
tryCatchOne	NaN
unique	NaN
unique.default	NaN
unlist	NaN
vapply	NaN
withVisible	NaN

---

*summary\_stacks(rpo)* Input

---

	len	refcount	root	leafs
1	13	1	53	27
2	18	1	53	27
3	17	16	53	37
4	19	23	53	42
5	21	1	53	27
6	21	48	53	5
7	20	1	53	27
8	15	1	53	27
9	18	25	53	15
10	15	55	53	39
11	14	1	53	27
12	14	114	53	30
13	15	23	53	50
14	13	43	53	29
15	18	10	53	50
16	14	1	53	27
17	14	2	53	19
18	13	36	53	52
19	25	1	53	35
20	15	35	53	13
21	18	15	53	41
22	20	1	53	1
23	20	7	53	15
24	14	2	53	40
25	20	1	53	62
26	23	9	53	12
27	20	14	53	28

Output

```

28 19      1  53  24
29 22      1  53  17
30 15      1  53  47
31 14      2  53  10
32 21      1  53   7
33 17      1  53  15
34 16      1  53  43
35 15      1  53  34
36 22      1  53  11
37 19      1  53  55
38 21      1  53  46
39 16      1  53   7
40 18      1  53   3
41 24      1  53  31
42 16      1  53   8
43 19      1  53  44
44 19      1  53  31
45 17      1  53  14
46 20      1  53  61
47  3      1  53  18

```

---

Input

---

```
summary_profiles(rpo)
```

---

Output

---

```

$nid
[1] "Profile Summary Mon Jul  8 18:02:20 2013"

$len
[1] 508

$uniquestacks
[1] 47

$nr_runs
[1] 373

```

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

#### 4.2. Print.

---

Input

---

```
print_nodes(rpo)
```

---

Output

---

	shortname	root	leaf	self.time	self.pct	total.time
.deparseOpts	.dp0	-	LEAF	1	Inf	0
.getXlevels	.gtX	-	-	0	NaN	0
.row_names_info	.r__	-	LEAF	1	Inf	0
[	[	-	-	0	NaN	0
[[.data.frame	[[.d.	-	LEAF	48	Inf	0
[[	[[	-	-	0	NaN	0
[[.data.frame	[[..	-	LEAF	2	Inf	0

%in%	%in%	- LEAF	1	Inf	0
<Anonymous>	<An>	- -	0	NaN	0
\$	\$	- LEAF	2	Inf	0
anyDuplicated	anyD	- LEAF	1	Inf	0
anyDuplicated.default	anD.	- LEAF	9	Inf	0
as.character	as.c	- LEAF	35	Inf	0
as.list	as.l	- LEAF	1	Inf	0
as.list.data.frame	a...	- LEAF	33	Inf	0
as.vector	as.v	- -	0	NaN	0
c	c	- LEAF	1	Inf	0
cat	cat	- LEAF	1	Inf	0
chol2inv	chl2	- LEAF	2	Inf	0
deparse	dprs	- -	0	NaN	0
doTryCatch	dTrC	- -	0	NaN	0
eval	eval	- -	0	NaN	0
evalFunc	evlF	- -	0	NaN	0
FUN	FUN	- LEAF	1	Inf	0
getOption	gtOp	- -	0	NaN	0
lapply	lppl	- -	0	NaN	0
lazyLoadDBfetch	lLDB	- LEAF	7	Inf	0
list	list	- LEAF	14	Inf	0
lm	lm	- LEAF	43	Inf	0
lm.fit	lm.f	- LEAF	114	Inf	0
match	mtch	- LEAF	2	Inf	0
match.fun	mtc.	- -	0	NaN	0
mean	mean	- -	0	NaN	0
mean.default	mn.d	- LEAF	1	Inf	0
mode	mode	- LEAF	1	Inf	0
model.frame	mdl.f	- -	0	NaN	0
model.frame.default	mdl.f.	- LEAF	16	Inf	0
model.matrix	mdl.m	- -	0	NaN	0
model.matrix.default	mdl.m.	- LEAF	55	Inf	0
model.response	mdl.r	- LEAF	2	Inf	0
na.omit	n.mt	- LEAF	15	Inf	0
na.omit.data.frame	n...	- LEAF	23	Inf	0
names	nams	- LEAF	1	Inf	0
options	optn	- LEAF	1	Inf	0
paste	past	- -	0	NaN	0
pmatch	pmtc	- LEAF	1	Inf	0
rep.int	rp.n	- LEAF	1	Inf	0
sapply	sppl	- -	0	NaN	0
simplify2array	smp2	- -	0	NaN	0
structure	strc	- LEAF	33	Inf	0
summary	smmr	- -	0	NaN	0
summary.lm	summ.	- LEAF	36	Inf	0
Sweave	Swev	ROOT -	0	NaN	0
terms	trms	- -	0	NaN	0
terms.formula	trm.	- LEAF	1	Inf	0
try	try	- -	0	NaN	0
tryCatch	tryC	- -	0	NaN	0
tryCatchList	trCL	- -	0	NaN	0
tryCatchOne	trCO	- -	0	NaN	0
unique	uniq	- -	0	NaN	0



unique.default	unq.	- LEAF	1	Inf	0
unlist	unls	- LEAF	1	Inf	0
vapply	vppl	- -	0	NaN	0
withVisible	wthV	- -	0	NaN	0
	total.pct				
.deparseOpts	NaN				
.getXlevels	NaN				
.row_names_info	NaN				
[	NaN				
[.data.frame	NaN				
[[	NaN				
[[.data.frame	NaN				
%in%	NaN				
<Anonymous>	NaN				
\$	NaN				
anyDuplicated	NaN				
anyDuplicated.default	NaN				
as.character	NaN				
as.list	NaN				
as.list.data.frame	NaN				
as.vector	NaN				
c	NaN				
cat	NaN				
chol2inv	NaN				
deparse	NaN				
doTryCatch	NaN				
eval	NaN				
evalFunc	NaN				
FUN	NaN				
getOption	NaN				
lapply	NaN				
lazyLoadDBfetch	NaN				
list	NaN				
lm	NaN				
lm.fit	NaN				
match	NaN				
match.fun	NaN				
mean	NaN				
mean.default	NaN				
mode	NaN				
model.frame	NaN				
model.frame.default	NaN				
model.matrix	NaN				
model.matrix.default	NaN				
model.response	NaN				
na.omit	NaN				
na.omit.data.frame	NaN				
names	NaN				
options	NaN				
paste	NaN				
pmatch	NaN				
rep.int	NaN				
sapply	NaN				

simplify2array	NaN
structure	NaN
summary	NaN
summary.lm	NaN
Sweave	NaN
terms	NaN
terms.formula	NaN
try	NaN
tryCatch	NaN
tryCatchList	NaN
tryCatchOne	NaN
unique	NaN
unique.default	NaN
unlist	NaN
vapply	NaN
withVisible	NaN

---

*print\_stacks(rpo)* Input

---

	len	refcount	root	leafs
1	13	1	53	27
2	18	1	53	27
3	17	16	53	37
4	19	23	53	42
5	21	1	53	27
6	21	48	53	5
7	20	1	53	27
8	15	1	53	27
9	18	25	53	15
10	15	55	53	39
11	14	1	53	27
12	14	114	53	30
13	15	23	53	50
14	13	43	53	29
15	18	10	53	50
16	14	1	53	27
17	14	2	53	19
18	13	36	53	52
19	25	1	53	35
20	15	35	53	13
21	18	15	53	41
22	20	1	53	1
23	20	7	53	15
24	14	2	53	40
25	20	1	53	62
26	23	9	53	12
27	20	14	53	28
28	19	1	53	24
29	22	1	53	17
30	15	1	53	47
31	14	2	53	10

Output

```

32 21      1  53    7
33 17      1  53   15
34 16      1  53   43
35 15      1  53   34
36 22      1  53   11
37 19      1  53   55
38 21      1  53   46
39 16      1  53    7
40 18      1  53    3
41 24      1  53   31
42 16      1  53    8
43 19      1  53   44
44 19      1  53   31
45 17      1  53   14
46 20      1  53   61
47  3      1  53   18

```

---

*Input*

---

```
print_profiles(rpo)
```

---

*Output*

---

```

$tid
[1] "Profile Summary Mon Jul  8 18:02:20 2013"

$len
[1] 508

$uniquestacks
[1] 47

$nr_runs
[1] 373

```

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

#### 4.3. Plot.

---

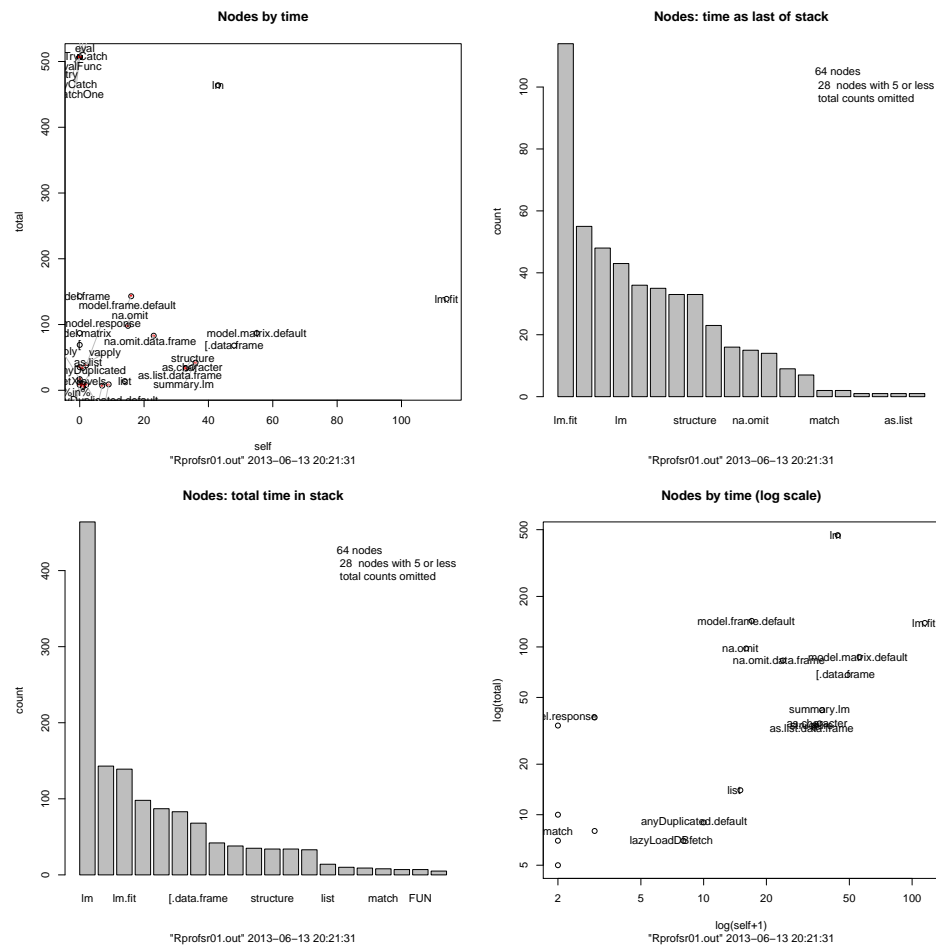
*Input*

---

```

oldpar<- par(mfrow=c(2,2))
plot_nodes(rpo)
par(oldpar)

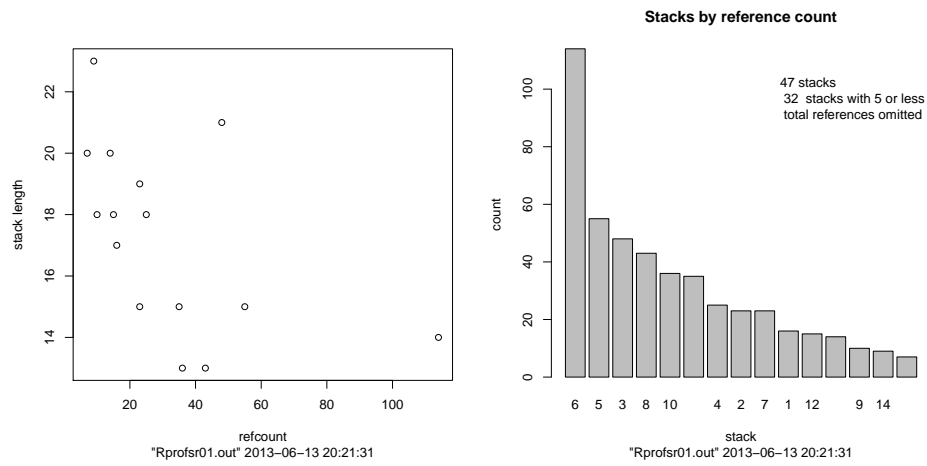
```




---

Input

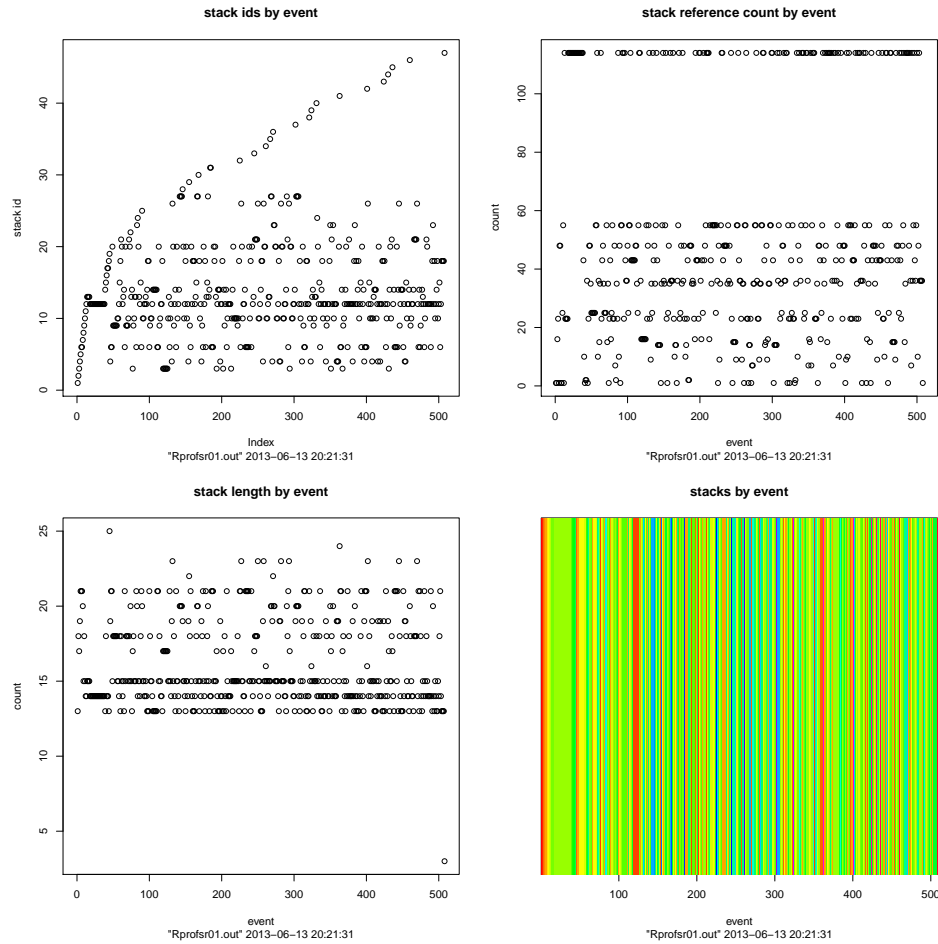
```
oldpar<- par(mfrow=c(1,2))
plot_stacks(rpo)
par(oldpar)
```




---

*Input*

```
oldpar<- par(mfrow=c(2,2))
plot_profiles(rpo)
par(oldpar)
```



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

## 5. GRAPH

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:

---

```
sprof <- sprof02 Input
```

---

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

---

```
sprofadj <- adjacency(sprof) Input
```

---

This is a format any graph package can handle.

## 5.1. graph Package.

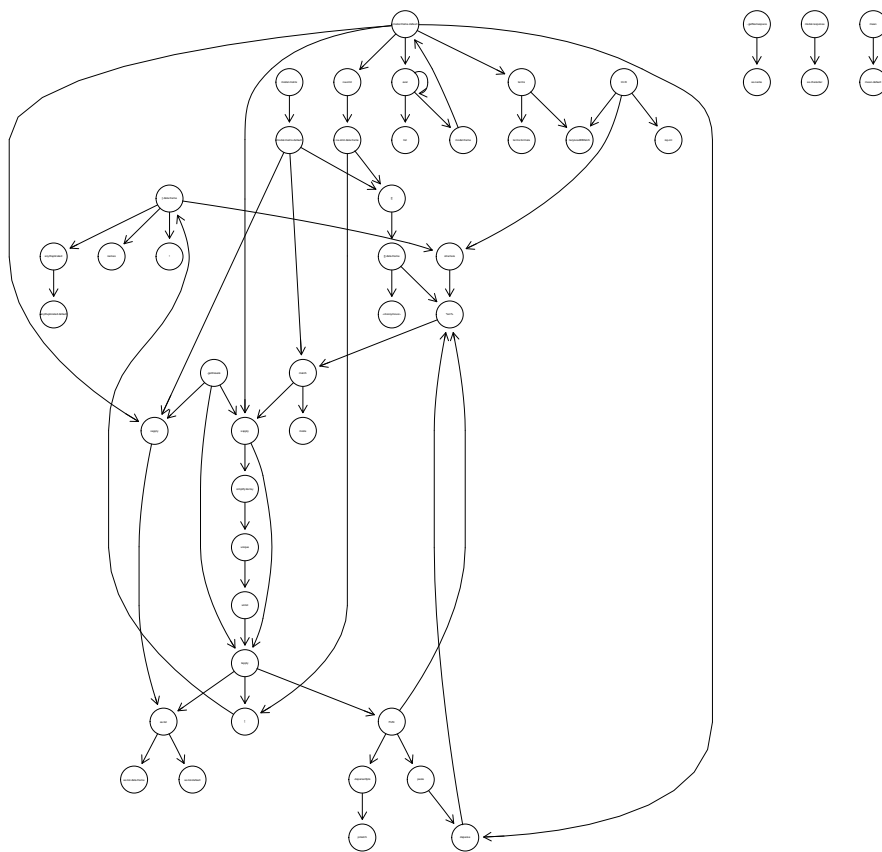
---

```
library(graph)
sprofadjNEL <- as(sprofadj, "graphNEL")
```

---

```
plot(sprofadjNEL, main="graph layout", cex.main=5)
#detach("package:graph")
```

---

**graph layout**

5.1.1. *igraph Package.*

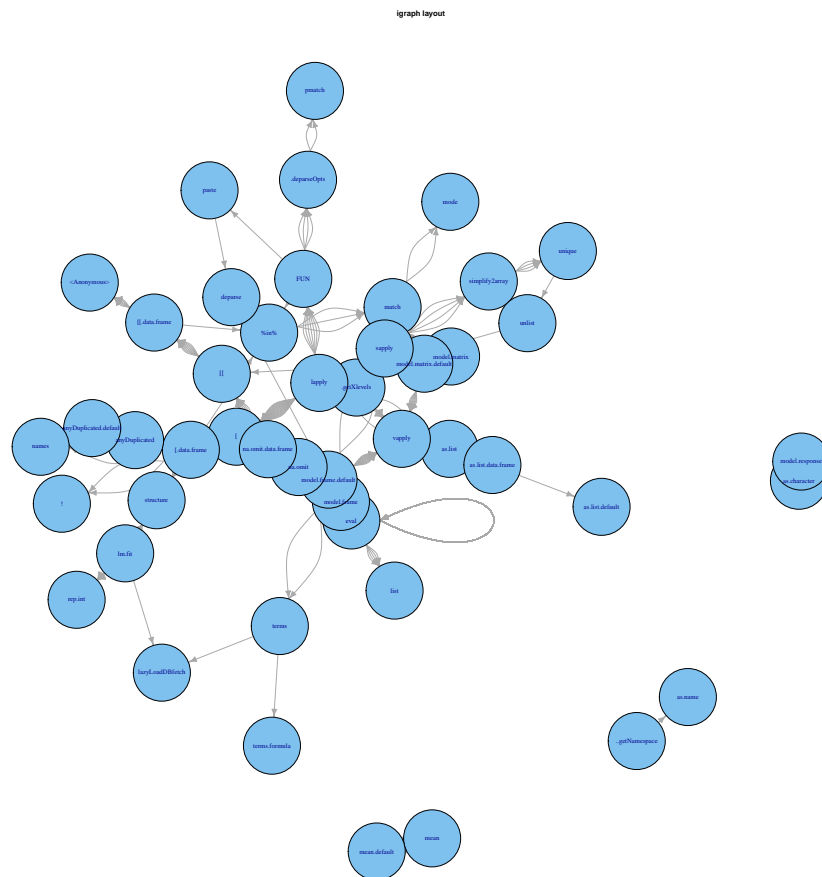

---

```
library(igraph)
sprofing <- graph.adjacency(sprofadj)
```

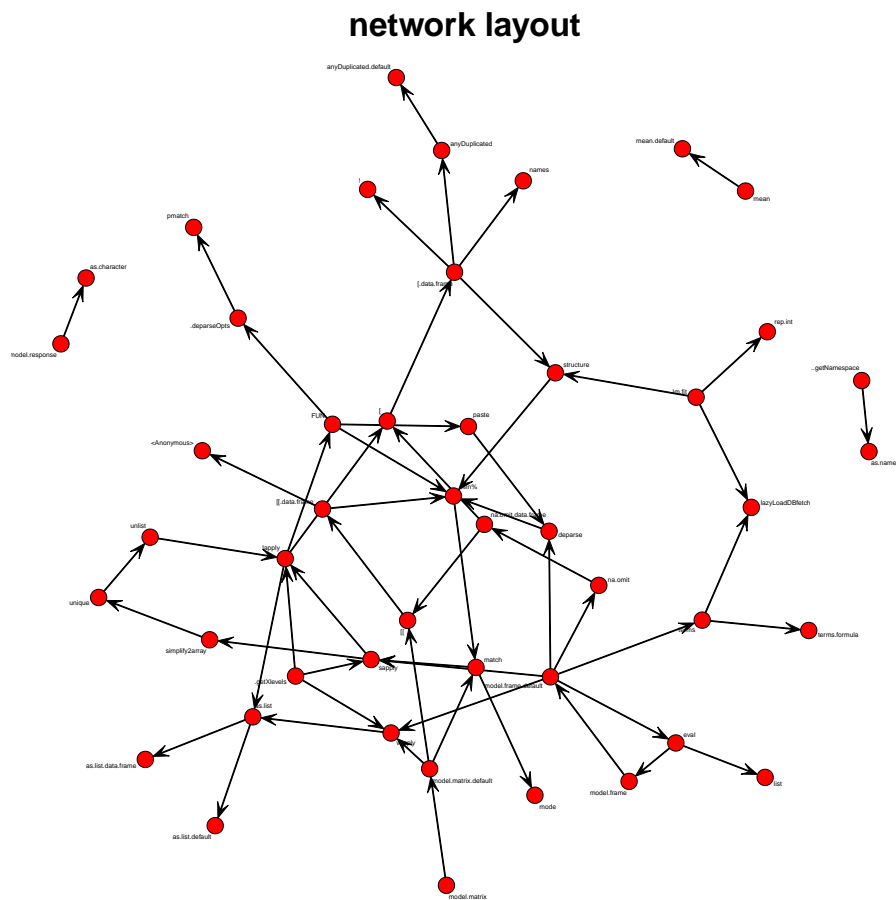
---

```
#plot(sprofing, main="igraph layout", cex.main=5)
plot(sprofing, main="igraph layout")
detach("package:igraph")
```

---







5.1.3. *Rgraphviz Package.*

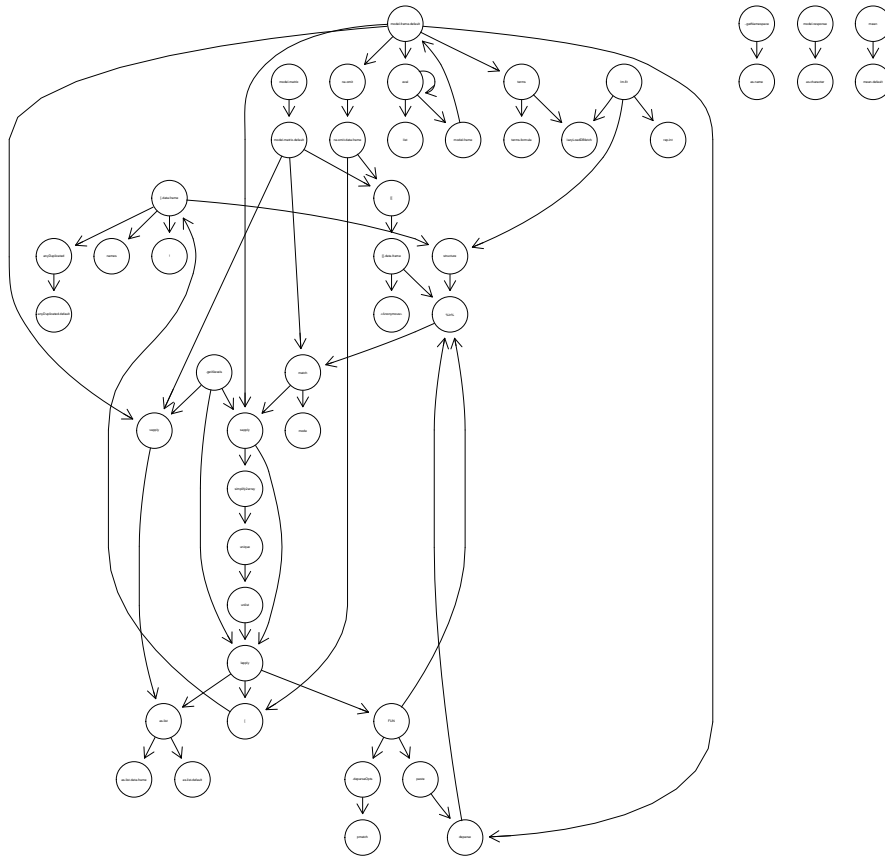

---

```
library(Rgraphviz)
sprofadjRag <- agopen(sprofadjNEL, name="Rprof Example")
```

---

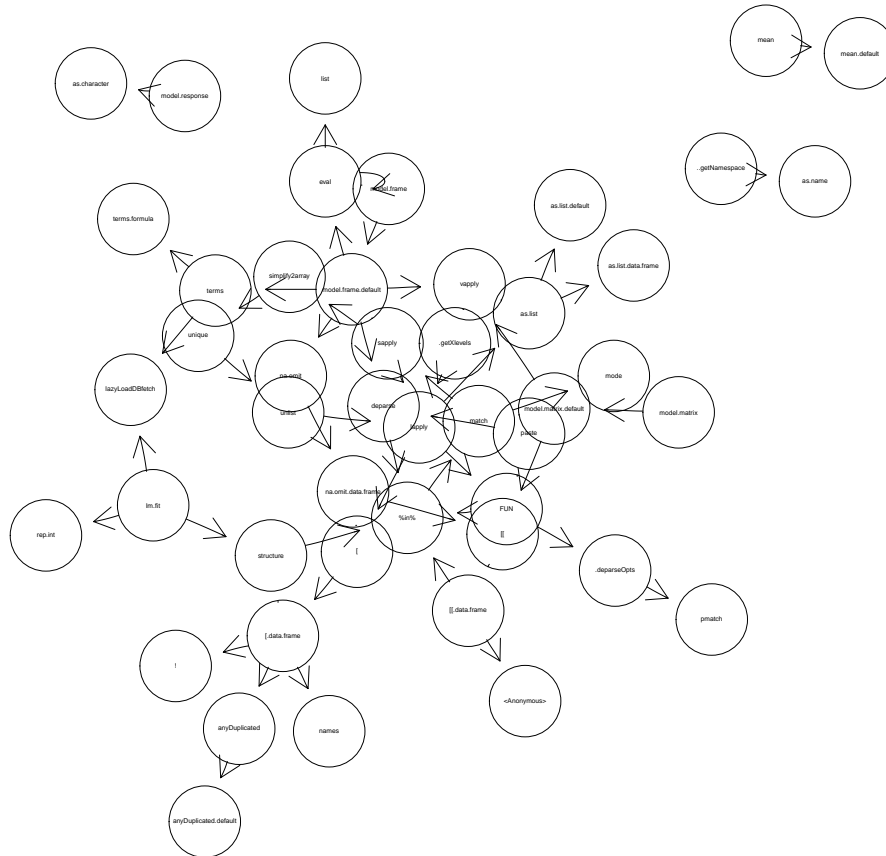
```
plot(sprofadjRag, main="Graphviz dot layout", cex.main=5)
```

---

**Graphviz dot layout**

*Input*

## Graphviz neto layout



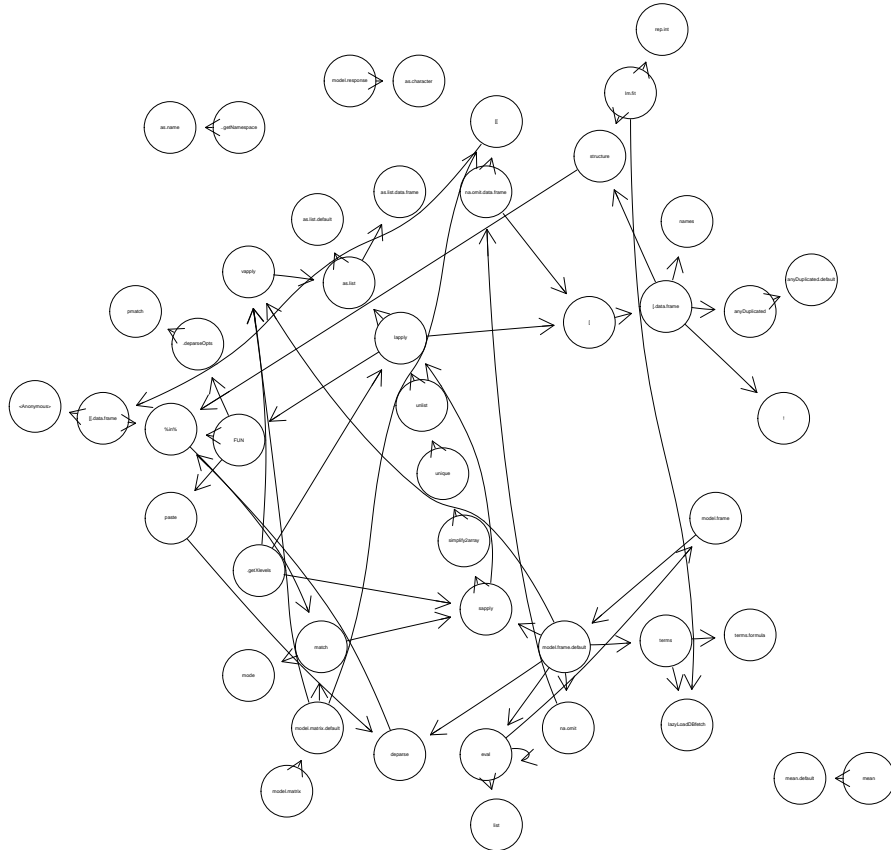
---

*Input*

```
plot(sprofadjRag,"twopi", main="Graphviz twopi layout", cex.main=5)
```

---

## Graphviz twopi layout

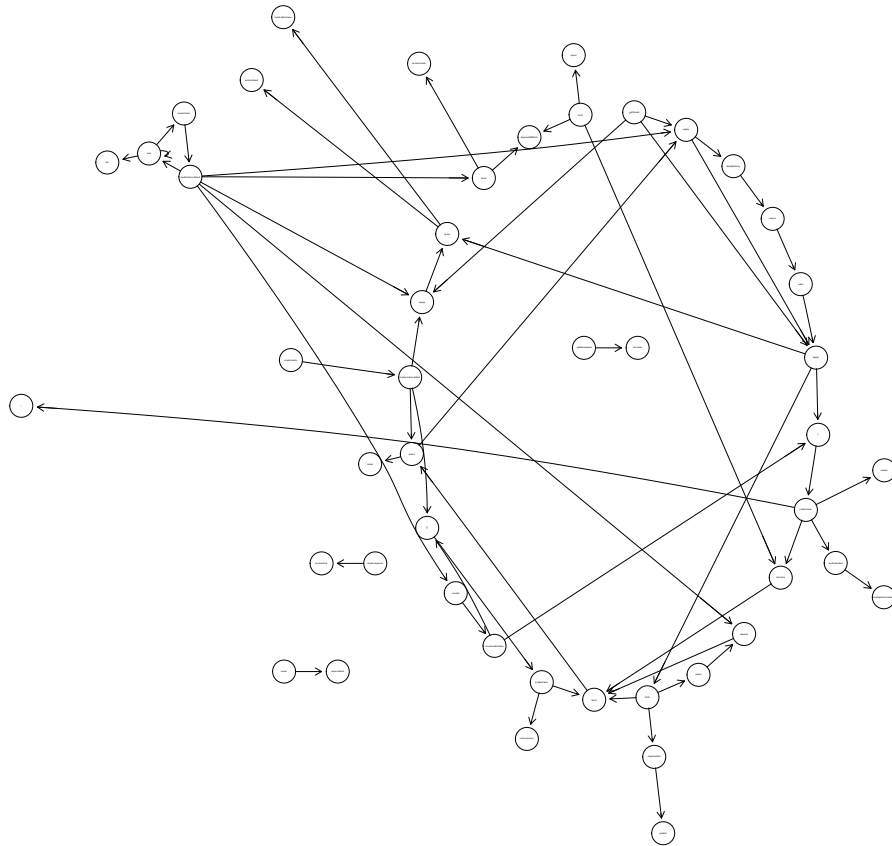


---

*Input*  
`plot(sprofadjRag,"circo", main="Graphviz circo layout", cex.main=5)`

---

### Graphviz circo layout

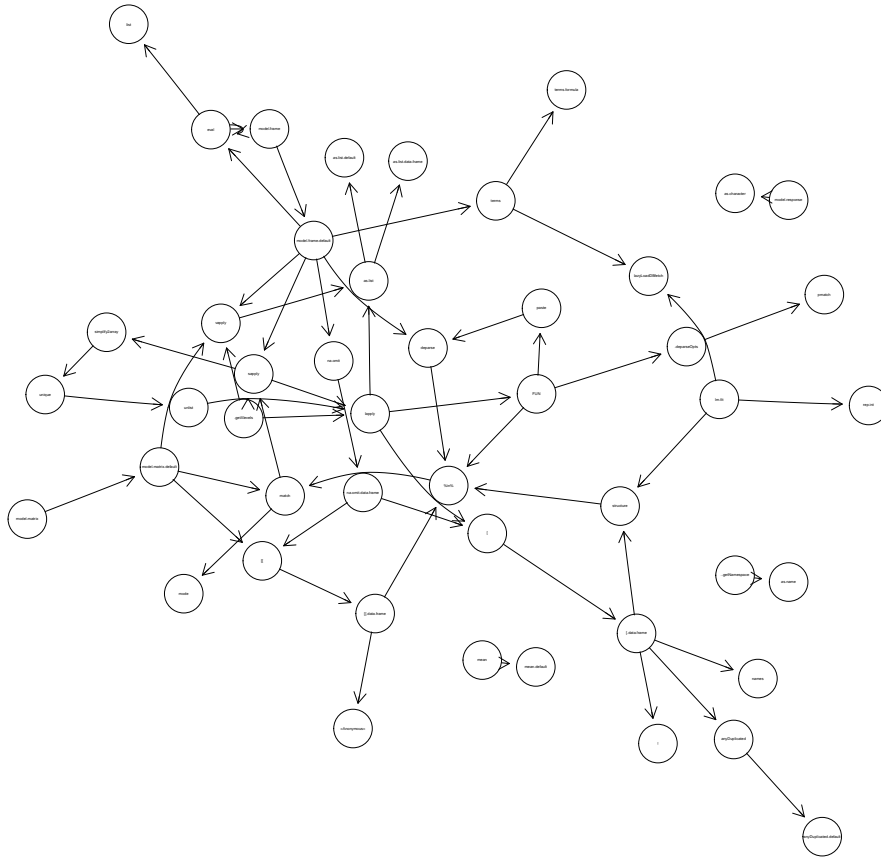


---

*Input*  
`plot(sprofadjRag,"fdp", main="Graphviz fdp layout", cex.main=5)`

---

### Graphviz fdp layout



## INDEX

Topic **adjacency**

until, 47

Topic **misc**

apply, 4

plot, 29, 47

print, 44

sampleRprof, 8

summary, 40

summaryRprof, 4

apply, 4

Index01, 55

plot, 29, 47

print, 44

sampleRprof, 8

summary, 40

summaryRprof, 4

until, 47

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 1.38.2, RColorBrewer 1.0-5, Rcpp 0.10.3, Rgraphviz 2.4.0, sprof 0.0-4, 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, stats4 3.0.1, tools 3.0.1

svn repository info:

```
$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 161 2013-07-07 12:30:36Z gsawitzki $
$Revision: 161 $
$Date: 2013-07-07 14:30:36 0200(Sun, 07 Jul 2013)+
$name: $
$Author: gsawitzki $
```

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/`