Under New Memory Management

by Luke Tierney

R 1.2 contains a new memory management system based on a generational garbage collector. This improves performance, sometimes only marginally but sometimes by a factor of two or more. The workspace is no longer statically sized and both the vector heap and the number of nodes can grow as needed. They can shrink again, but never below the initially allocated sizes.

Generational Garbage Collection

Garbage collection is the colorful name usually used for the process of reclaiming unused memory in a dynamic memory management system. Many algorithms are available; a recent reference is Jones and Lins (1996). A simple garbage collection system works something like this: The system starts with a given amount of heap memory. As requests for chunks of memory come in, bits of the heap are allocated. This continues until the heap is used up. Now the system examines all its variables and data structures to determine which bits of allocated memory are still in use. Anything that is no longer in use is garbage (hence the name) and can be reclaimed for satisfying the next set of allocation requests.

The most costly step in this process is determining the memory still in use. Generational collection, also called ephemeral collection, is designed to speed up this process. It is based on the observation that there tend to be two kinds of allocated objects. Objects representing built-in functions or library functions and objects representing data sets being analyzed tend to be needed for long periods of time and are therefore present at many successive collections. Intermediate results of computations, on the other hand, tend to be very short lived and often are only needed for one or two collections.

A generational collector takes advantage of this observation by arranging to examine recently allocated objects more frequently than older objects. Objects are marked as belonging to one of three generations. When an object is initially allocated, it is placed in the youngest generation. When a collection is required, the youngest generation is examined first. Objects still in use are moved to the second generation, and ones no longer in use are recycled. If this produces enough recycled objects, as it usually will, then no further collection is needed. If not enough recycled objects are produced, the second generation is collected. Once again, surviving objects are moved to the next, the final, generation, and objects no longer in use are recycled. On very rare occasions even this will not be enough and a collection of the final generation is needed. However for most collections examining the youngest generation is sufficient, and collections of the youngest generation tend to be very fast. This is the source of the performance improvement brought about by the generational collector.

Limiting Heap Size

Since the size of the R workspace is adjusted at runtime, it is no longer necessary to specify a fixed workspace size at startup. However, it may be useful to specify a limit on the heap size as a precaution in settings where attempting to allocate very large amounts of memory could adversely affect other users or the performance of the operating system.

Many operating systems provide a mechanism for limiting the amount of memory a process can use. Unix and Linux systems have the `limit` and `ulimit` commands for `csh` and `sh` shells, respectively. The Windows version of R also allows the maximal total memory allocation of R to be set with the command line option '`--max-mem-size`'. This is analogous to using the `limit` command on a Unix system.

For somewhat finer control R also includes command line arguments '`--max-nsize`' and '`--max-vsize`' for specifying limits at startup and a function `mem.limits` for setting and finding the limits at runtime. These facilities are described in more detail in `?Memory`.

If R hits one of these limits it will raise an error and return to the R top level, thus aborting the calculation that hit the limit. One possible extension currently under consideration is a more sophisticated error handling mechanism that might allow the option of asking the user with a dialog whether the heap limits should be raised; if the user agrees, then the current calculation would be allowed to continue.

API Changes

Implementing the generational collector required some changes in the C level API for accessing and modifying internal R data structures.

For the generational collector to work it has to be possible to determine which objects in the youngest generations are still alive without examining the older generations. An assignment of a new object to an old environment creates a problem. To deal with this problem, we need to use a *write barrier*, a mechanism for examining all assignments and recording any references from old objects to new ones. To insure accurate recording of any such references, all assignments of pointers into R objects must go through special assignment functions. To allow the correct use of these assignment functions to be checked reliably by the C compiler, it was necessary to also require that all reading of pointer fields go through

special accessor functions or macros. For example, to access element `i` of vector `x` you need to use

```
VECTOR_ELT(x, i)
```

and for assigning a new value `v` to this element you would use

```
SET_VECTOR_ELT(x, i, v)
```

These API changes are the main reason that packages need to be recompiled for 1.2. Further details on the current API are available in *"Writing R Extensions"*.

Future Developments

There are many heuristics used in the garbage collection system, both for determining when different generations are collected and when the size of the heap should be adjusted. The current heuristics seem to work quite well, but as we gain further experience with the collector we may be able to improve the heuristics further.

One area actively being pursued by the R core team is interfacing R to other systems. Many of these systems have their own memory management systems that need to cooperate with the R garbage collector. The basic tools for this cooperation are a finalization mechanism and weak references. A preliminary implementation of a finalization mechanism for use at the C level is already part of the collector in R 1.2. This will most likely be augmented with a weak reference mechanism along the lines described by Peyton Jones, Marlow and Elliott (1999).

References

Richard Jones and Rafael Lins (1996). *Garbage Collection*. Wiley. 10

Simon Peyton Jones, Simon Marlow, and Conal Elliott (1999). Stretching the storage manager: weak pointers and stable names in Haskell. http://www.research.microsoft.com/Users/simonpj/Papers/papers.html. 11

Luke Tierney
University of Minnesota, U.S.A.
luke@stat.umn.edu

On Exact Rank Tests in R

by Torsten Hothorn

Linear rank test are of special interest in many fields of statistics. Probably the most popular ones are the Wilcoxon test, the Ansari-Bradley test and the Median test, therefore all implemented in the standard package **ctest**. The distribution of their test statistics under the appropriate hypothesis is needed for the computation of P-values or critical regions. The algorithms currently implemented in R are able to deal with untied samples only. In the presence of ties an approximation is used. Especially in the situation of small and tied samples, where the exact P-value may differ seriously from the approximated one, a gap is to be filled.

The derivation of algorithms for the exact distribution of rank tests has been discussed by several authors in the past 30 years. A popular algorithm is the so called network algorithm, introduced by Mehta and Patel (1983). Another smart and powerful algorithm is the shift algorithm by Streitberg and Röhmel (1986). In this article, we will discuss the package **exactRankTests**, which implements the shift algorithm. The computation of exact P-values and quantiles for many rank statistics is now possible within R.

Using ExactRankTests

The algorithm implemented in package **exactRankTests** is able to deal with statistics of the form

$$T = \sum_{i=1}^{m} a_i$$

where $a = (a_1, \ldots, a_N)$ are positive, integer valued scores assigned to N observations. We are interested in the distribution of T under the hypothesis that all permutations of the scores a are equally likely. Many rank test can be regarded this way, e.g. the Wilcoxon test is a special case with $a_i = i$ and the Ansari-Bradley test has scores $a_i = \min(i, N - i + 1)$. For details about the algorithm we point to the original articles, for example Streitberg and Röhmel (1986) and Streitberg and Röhmel (1987).

Package **exactRankTests** implements the functions `dperm`, `pperm`, and `qperm`. As it is standard in R/S they give the density, the probability function and the quantile function. Additionally, the function `pperm2` computes two-sided P-values. Consider e.g. the situation of the Wilcoxon test. Let `x` and `y` denote two vectors of data, possibly tied. First, we compute the ranks over all observations and second we compute the Wilcoxon statistic, which is the sum over the ranks of the `x` sample.

```
R> ranks <- rank(c(x,y))
```