Building a simple graph sampler again

Alun Thomas

2023-05-10

Introduction

The `rviewgraph` package provides functions for creating, changing, and viewing a graph. It originally contained only one function called `rViewGraph()` that allowed viewing a fixed graph using an animated graphical user interface (GUI). A newer function called 'vg()' is now available and allows for viewing a graph the structure of which can change dynamically. To illustrate this we will implement a simple Markov chain Monte Carlo (MCMC) sampler with a given target distribution.

Creating the graph viewer

First ensure that `rviewgraph` is available, and attempt to start both an old version of the GUI using `rViewGraph()`, and new version of the GUI using `vg()`. For the old version we need to specify a graph in advance so we generate some random edge end points.

```
library(rviewgraph)

oldgui = rViewGraph(sample(1:10,10,TRUE),sample(1:10,10,TRUE))
#> [1] "rViewGraph only runs in interactive mode."
is.null(oldgui)
#> [1] TRUE
if (interactive() && !is.null(oldgui))
{
    oldgui.stop()
    oldgui.hide()
}

v = vg()
#> [1] "Running without a visible GUI."
is.null(v)
#> [1] FALSE
```

We see that `rViewGraph()` returns `NULL` and prints an error message. This is because the old programs can only be run in interactive mode. The new version, `vg()`, however, will return a list of functions that can be run non-interactively, but, of course, without a visible GUI. As this vignette was produced non-interactively, there is no GUI, but if you are replicating it from a live `R` session you should see a new `Java` window that has appeared with a blank canvas on which the graph will be viewed. There will also be a GUI showing the random graph generated in the call to `rViewGraph()`, so we remove that.

The returned value `v` is a list of functions that we will use to display the graph as we run the sampler.

Defining the elements of the sampler

First we define a probability mass function on graphs, in fact we need only specify this up to a constant of proportionality. This simple example depends only on the number of edges in the graph, and as such it would be more efficient to track this count as we iterate through the MCMC scheme, however, this will illustrate how the structure of the graph can be queried, and also makes the MCMC scheme more easily adapted to different probability distributions. For numerical stability, we work with logs of probabilities.

This probability distribution favours more sparsely edged graphs if we set `a > 0`, and more densely edged ones if `a < 0`. We can query the graph for a two dimensional array that specifies the edges. Because this graph is undirected, each edge is returned twice, once in each direction, so we divide to get the number of edges.

```
edges = function(g)
{
    dim(g$getEdges())[1] / 2
}

logProb = function(g,a)
{
    -edges(g)*a
}
```

Here is a function that flips an edge in a graph. That is, if a pair of vertices are connected it disconnects them, if disconnected, it connects them. This will be our Metropolis proposal scheme, and it is reversible. The argument `g` is assumed to be a list that was returned by calling `vg()`.

```
flip = function(g,x,y)
{
    if (g$connects(x,y))
        g$disconnect(x,y)
    else
        g$connect(x,y)
}
```

Now we define a Metropolis MCMC proposal and acceptance step. The calls to `Sys.sleep()` just slow the process down to make the process easier to follow in the viewer when used interactively. The colour changes will show the proposals being made.

```
step = function(g, a, t = 1)
{
    # Find the probability of the current graph.
    delta = logProb(g,a)

    # Choose a pair of random vertices to flip, and display them.
    x = sample(g$getVertices(),2,replace=FALSE)
    g$colour(x,"red")
    Sys.sleep(t/3)

    # Then make the flip.
    flip(g,x[1],x[2])
    Sys.sleep(t/3)

    # Find the ratio of the probabilities of the proposed new graph
    # and the previous incumbent.
    delta = logProb(g,a) - delta
```

```
    # Then with probability equal to the minimum of 1 and this ratio
    # accept the new graph by doing nothing, otherwise reject it by
    # undoing the flip.
    if (log(runif(1)) > delta)
        flip(g,x[1],x[2])

    # Reset the colours, and pause before the next step.
    g$colour(x)
    Sys.sleep(t/3)
}
```

The complete Metropolis MCMC scheme

We can now combine these functions to make a complete MCMC scheme using here a graph of `n = 20` vertices. As an example of an output, we can calculate the mean number of edges over the simulation. If we set the penalty to 0, we get a random walk over graphs of size `n`, so the simulated mean number of edges should be about `n(n-1)/4`.

If this is run in interactive mode, this is primarily to illustrate how the GUI shows the progress of the sampler, so we set the sleep parameter to give one step per second and make 50 updates. If run non-interactively, we don't need to slow down the sampler to see the progress, so we make more iterations and include a burn in period to allow the sampler to move away from the initial state and obtain better estimates of the mean number of edges.

The scheme is run using the viewer that was created at the beginning of this vignette.

```
    # To make the simulation reproducible.
    set.seed(1)

    n = 20
    v$add(1:n)

    pen = 0

    if (interactive())
    {
        wait = 1
        s = 50
    } else
    {
        wait = 0
        s = 1000
        for (i in 1:s)
            step(v,pen,wait)
    }

    medge = 0
    for (i in 1:s)
    {
        step(v,pen,wait)
        medge = medge + edges(v)
    }
    c(medge/s, n*(n-1)/4)
#> [1] 98.164 95.000
```

A positive penalty should decrease the mean number of edges.

```
    pen = 1
    medge = 0
    for (i in 1:s)
    {
        step(v,pen,wait)
        medge = medge + edges(v)
    }
    medge/s
#> [1] 58.688
```

And a negative one should increase the mean number of edges.

```
    pen = -1
    medge = 0
    for (i in 1:s)
    {
        step(v,pen,wait)
        medge = medge + edges(v)
    }
    medge/s
#> [1] 127.254
```