`geoGraph`: implementing geographic graphs for large-scale spatial modelling

Thibaut Jombart, Andrea Manica, François Balloux

June 28, 2010

Contents

1 Introduction **1**

2 First steps **2**
- 2.1 Installing the package . . . 2
- 2.2 Data representation . . . 3
  - 2.2.1 gGraph objects . . . 3
  - 2.2.2 gData objects . . . 5

3 Using geoGraph **6**
- 3.1 Importing geographic data . . . 6
- 3.2 Visualizing data . . . 11
- 3.3 Editing graphs . . . 11
- 3.4 Extracting information from GIS shapefiles . . . 11
- 3.5 Finding least-cost paths . . . 11

1 Introduction

This document describes the `geoGraph` package for the R software. `geoGraph` aims at implementing graph approaches for geographic data. In `geoGraph`, a given geographic area is modelled by a fine regular grid, where each vertice has a set of spatial coordinates and a set of attributes, which can be for instance habitat descriptors, or the presence/abundance of a given species. 'Travelling' within the geographic area can then be easily modelled as moving between connected vertices. The costs of moving from one vertex to another can be defined according to attribute values, which allows for instance to define *frictions* based on habitat.

`geoGraph` harnesses the full power of graph algorithms implemented in R by the `graph` and the `RBGL` (R Boost Graph Library) packages. In particular, RBGL is an interface between R and the impressive *Boost Graph*

Library in C++, proposing a wide range of algorithms with fast and efficient implementation. Therefore, once we have defined frictions for an entire geographic area, we can easily, for instance, find the least costs path from one node to another, or find the most parsimonious way of connecting a set of locations.

Once all data are set, calling upon RBGL routines is generally straightforward. However, interfacing spatial data and graphs can be a complicated task. The purpose of `geoGraph` is to simplify these 'preliminary' steps. This is achieved by defining new classes of objects which are essentially geo-referenced graphs with attributes (`gGraph` objects) and interfaced spatial data (`gData` objects). In this vignette, we show how to install `geoGraph`, show how to construct and handle `gGraph`/`gData` objects, and illustrate some basic features of graph algorithms.

2 First steps

2.1 Installing the package

What is tricky here is that a vignette is basically available once the package is installed. Assuming you got this document before installing the package, here are some clues about installing `geoGraph`.

First of all, `geoGraph` depends on the packages `methods` (base package), `graph` (on Bioconductor), and `RBGL` (on Bioconductor). These dependencies are mandatory, that is, you actually need to have these packages installed before using `geoGraph`. Also, it is better to make sure you are using the latest versions of these packages. While the `methods` package is part of the basic R release, `graph` and `RBGL` are no longer developped on CRAN, although some outdated versions still persist. To make sure you are using the right version, use the command `installDep.geoGraph()` while connected on the internet. Do NOT use `install.packages`, or related functionalities from the interactive menus. In all cases, the latest version of `geoGraph` can be found from http://r-forge.r-project.org/R/....

When loading the package, dependencies are also loaded:

```
> library(geoGraph)


        Note: polygon geometry computations in maptools
         depend on the package gpclib, which has a
         restricted licence. It is disabled by default;
         to enable gpclib, type gpclibPermit()

Checking rgeos availability as gpclib substitute:
FALSE

        ========================
        geoGraph 1.0-0 is loaded
        ========================
```

```
> search()
```

```
 [1] ".GlobalEnv"        "package:geoGraph"  "package:fields"
 [4] "package:spam"      "package:maptools"  "package:lattice"
 [7] "package:foreign"   "package:sp"        "package:RBGL"
[10] "package:graph"     "package:datasets"  "package:adegenet"
[13] "package:ade4"      "package:MASS"      "package:utils"
[16] "package:stats"     "package:graphics"  "package:grDevices"
[19] "package:methods"   "Autoloads"         "package:base"
```

The package is now ready to use.

2.2 Data representation

Two types of objects are used in `geoGraph`: `gGraph`, and `gData` objects. Both objects are defined as formal (S4) classes and often have methods for similar generic function (*e.g.*, getNodes is defined for both objects). Essentially, `gGraph` objects contain underlying layers of informations, including a spatial grid and possibly node attributes, and covering the area of interest. `gData` are sets of locations – like sampled sites, for instance – which have been interfaced to a `gGraph` object, to allow further manipulations such as finding paths on the grid between pairs of locations.

2.2.1 gGraph objects

The content of the formal class `gGraph` can be obtained using:

```
> getClass("gGraph")
```

```
Class "gGraph" [package "geoGraph"]

Slots:

Name:      coords nodes.attr       meta      graph
Class:     matrix data.frame       list   graphNEL
```

and a new empty object can be obtained using the constructor:

```
> new("gGraph")
```

```
=== gGraph object ===

@coords: spatial coordinates of 0 nodes
     lon lat

@nodes.attr: 0 nodes attributes
data frame with 0 columns and 0 rows

@meta: list of meta information with 0 items

@graph:
A graphNEL graph with undirected edges
Number of Nodes = 0
Number of Edges = 0
```

The documentation `?gGraph` explains the basics about the object's content. In a nutshell, these objects are spatial grids with nodes and segments connecting neighbouring nodes, and additional informations on the nodes or on the graph itself. `coords` is a matrix of longitudes and latitudes of the nodes. `nodes.attr` is a data.frame storing attributes of the nodes, such as habitat descriptors; each row corresponds to a node of the grid, each column being a variable. `meta` is a list containing miscellanous informations on the graph itself. There is no contraint in the components of the list, but some components will be recognised by certain functions. For instance, you can specify plotting rules for representing a given node attribute by a given color by defining a component `$colors`. Similarly, you can associate costs to a given node attribute by defining a component `$costs`. An example of this can be found in already existing `gGraph` objects. For instance, `worldgraph.10k` is a graph of the world with approximately 10,000 nodes, and only on-land connectivity (*i.e.* no travelling on the seas).

```
> data(worldgraph.10k)
> worldgraph.10k


=== gGraph object ===

@coords: spatial coordinates of 10242 nodes
        lon       lat
1 -180.0000  90.00000
2  144.0000 -90.00000
3  -33.7806  27.18924
...

@nodes.attr: 1 nodes attributes
  habitat
1     sea
2     sea
3     sea
...

@meta: list of meta information with 2 items
[1] "$colors" "$costs"

@graph:
A graphNEL graph with undirected edges
Number of Nodes = 10242
Number of Edges = 6954


> worldgraph.10k@meta


$colors
           habitat       color
1              sea        blue
2             land       green
3         mountain       brown
4       landbridge light green
5 oceanic crossing  light blue
6  deselected land   lightgray

$costs
           habitat cost
1              sea  100
2             land    1
3         mountain   10
4       landbridge    5
5 oceanic crossing   20
6  deselected land  100
```

Lastly, the `graph` component is a `graphNEL` object, which is the standard class for graphs in the `graph` and `RBGL` packages. This object contains all information on the connections between nodes, and the weights (costs) of these connections.

Four main `gGraph` are provided with `geoGraph`: `rawgraph.10k`, `rawgraph.40k`, `worldgraph.10k`, and `worldgraph.40k`. These datasets are available using the command `data`. All these datasets are evenly spaced grids covering the entire earth, with some basic habitats descriptors provided as node attributes. The difference between rawgraphs and worldgraphs is that the first are entirely connected, while in the second connections occur only on land. Numbers '10k' and '40k' indicate that the grids are formed of roughly 10,000 and 40,000 nodes. For illustrative purposes, we shall use the 10k grids, since they are less heavy to handle. For most large-scale applications, the 40k versions should provide sufficient resolution. New `gGraph` can be constructed using the constructor (`new(...)`), but this topic is not documented in this vignette.

2.2.2 gData objects

`gData` are essentially sets of locations that have been interfaced with a `gGraph` object. During this operation, each location is assigned to the closest node on the grid of the `gGraph`, then allowing for travelling between locations on the grid. Then, for instance, it is possible to find the shortest path between two locations through some types of habitats, or using ecological costs.

Like for `gGraph`, the content of the formal class `gData` can be obtained using:

```
> getClass("gData")
```

```
Class "gData" [package "geoGraph"]

Slots:

Name:       coords    nodes.id        data gGraph.name
Class:      matrix   character         ANY   character
```

and a new empty object can be obtained using the constructor:

```
> new("gData")
```

```
=== gData object ===

@coords: spatial coordinates of 0 nodes
     lon lat

@nodes.id: nodes identifiers
character(0)

@data: data
NULL

Associated gGraph:
```

As before, the description of the content of these objects can be found in the documentation (`?gData`). `coords` is a matrix of xy (longitude/latitude) coordinates in which each row is a location. `nodes.id` is vector of characters giving the name of the nodes matching the locations; this is defined automatically when creating a new `gData`, or using the function `closestNode`. `data` is a slot storing data associated to the locations; it can be anything, but a data.frame should cover most requirements for storing data. Note that this object should be subsettable (i.e. the `[` operator should be defined), so that data can be subsetted when subsetting the `gData` object. Lastly, the slot `gGraph.name` contains the name of the `gGraph` object to which the `gData` has been interfaced.

Contrary to `gGraph` objects, `gData` objects will frequently be constructed. In the next sections, we shall illustrate how we can build and use `gData` objects from a set of locations.

3 Using geoGraph

An overview of the material implemented in the package is summarized the package's manpage, accessible via:

```
> `?`(geoGraph)
```

The html version of this manpage may be preferred to browse more easily the content of `geoGraph`; it is accessible by typing:

```
> help("geoGraph", package = "geoGraph", html = TRUE)
```

To revert help back to text mode, simply type:

```
> options(htmlhelp = FALSE)
```

In the following, we go through various tasks that can be achieve using `geoGraph`.

3.1 Importing geographic data

Geographic data consist of a set of locations, possibly accompanied by additional information. For instance, one may want to study the migrations amongst a set of biological populations with known geographic coordinates. In `geoGraph`, geographic data are stored in `gData` objects. These objects match locations to the closest nodes on a grid (a `gGraph` object), and store additional data if needed.

As a toy example, let us consider three locations: Bordeaux (France), London (UK), Malaga (Spain), and Zagreb (Croatia). Since we will be working with a crude grid (10,000 nodes), locations need not be exact. We enter the longitudes and latitudes (in this order, that is, xy coordinates) of these cities in decimal degrees, as well as approximate population sizes:

```
> Bordeaux <- c(-1, 45)
> London <- c(0, 51)
> Malaga <- c(-4, 37)
> Zagreb <- c(16, 46)
> cities.dat <- rbind.data.frame(Bordeaux, London, Malaga, Zagreb)
> colnames(cities.dat) <- c("lon", "lat")
> cities.dat$pop <- c(1e+06, 1.3e+07, 5e+05, 1200000)
> row.names(cities.dat) <- c("Bordeaux", "London", "Malaga", "Zagreb")
> cities.dat
```

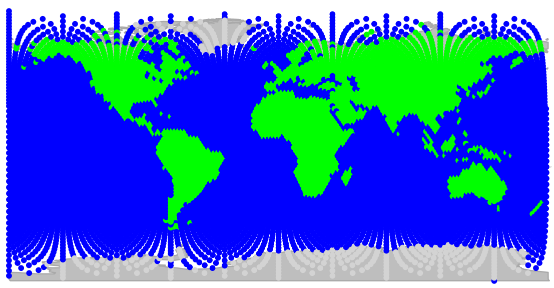
```
         lon lat     pop
Bordeaux  -1  45 1.0e+06
London     0  51 1.3e+07
Malaga    -4  37 5.0e+05
Zagreb    16  46 1.2e+06
```

We load a `gGraph` object which contains the grid supporting the data:

```
> data(worldgraph.10k)
> worldgraph.10k
```

```
=== gGraph object ===

@coords: spatial coordinates of 10242 nodes
       lon       lat
1 -180.0000  90.00000
2  144.0000 -90.00000
3  -33.7806  27.18924
...

@nodes.attr: 1 nodes attributes
  habitat
1     sea
2     sea
3     sea
...

@meta: list of meta information with 2 items
[1] "$colors" "$costs"

@graph:
A graphNEL graph with undirected edges
Number of Nodes = 10242
Number of Edges = 6954
```

```
> plot(worldgraph.10k)
```

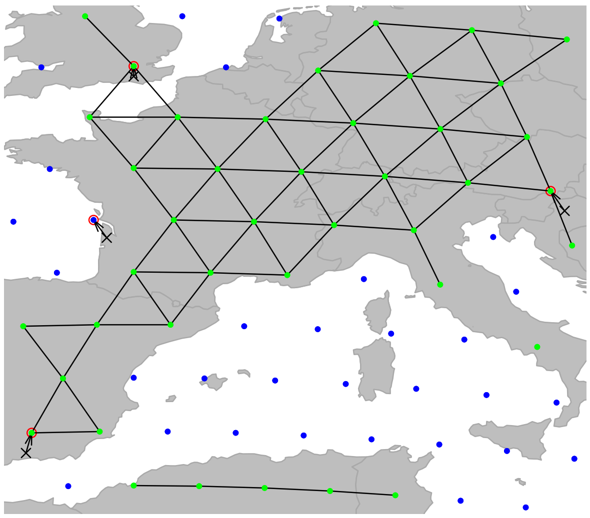


(we could use `worldgraph.40k` for a better resolution). On this figure, each node is represented with a color depending on the habitat type, either 'sea' (blue) or 'land' (green). We are going to interface the cities data with this grid; to do so, we create a `gData` object using `new` (see `?gData` object):

```
> cities <- new("gData", coords = cities.dat[, 1:2], data = cities.dat[,
+     3, drop = FALSE], gGraph.name = "worldgraph.10k")
> cities
```

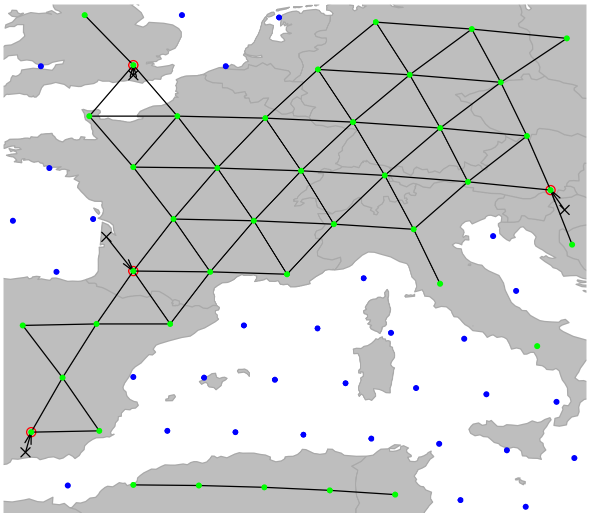
```
=== gData object ===

@coords: spatial coordinates of 4 nodes
  lon lat
1  -1  45
2   0  51
3  -4  37
...

@nodes.id: nodes identifiers
     1      2      3
"5774" "6413" "4815"
...

@data: 4 data
             pop
Bordeaux 1.0e+06
London   1.3e+07
Malaga   5.0e+05
...

Associated gGraph: worldgraph.10k
```

```
> plot(cities, type = "both", reset = TRUE)
> plotEdges(worldgraph.10k)
```



This figure illustrates the matching of original locations (crosses) to nodes of the grid (red circles). As we can see, an issue occured for Bordeaux, which has been assigned to a node in the sea (in blue). Locations can be re-assigned to nodes with restrictions for some node attribute values using `closestNode`; for instance, here we constrain matching nodes to have an `habitat` value (defined as node attribute in `worldgraph.10k`) equalling `land` (green points):

```
> cities <- closestNode(cities, attr.name = "habitat", attr.value = "land")
> plot(cities, type = "both", reset = TRUE)
> plotEdges(worldgraph.10k)
```



Now, all cities have been assigned to a node on the grid (again, better accuracy will be gained on 40k or finer grids - we use 10k for illustrative purposes only). Content of `cities` can be accessed via various accessors (see `?gData`). For instance, we can retrieve original locations, assigned nodes, and additional data using:

```
> getCoords(cities)
```

```
     lon lat
5775  -1  45
6413   0  51
4815  -4  37
7699  16  46
```

```
> getNodes(cities)
```

```
  5775   6413   4815   7699
"5775" "6413" "4815" "7699"
```

```
> getData(cities)
```

```
             pop
Bordeaux 1.0e+06
London   1.3e+07
Malaga   5.0e+05
Zagreb   1.2e+06
```

We can also get the exact coordinates of the matching nodes using:

```
> getCoords(cities, original = FALSE)
```

```
               lon      lat
5775  1.001791e-05 43.73025
6413  1.001791e-05 51.37555
4815 -3.787658e+00 37.74879
7699  1.547808e+01 46.73633
```

More interestingly, we can now retrieve all the geographic information contained in the underlying grid (i.e., `gGraph` object) as node attributes:

```
> getNodesAttr(cities)
```

```
     habitat
5775    land
6413    land
4815    land
7699    land
```

In this example, the information stored in `worldgraph.10k` is rather crude: it only distinguishes land from sea. However, more complex habitat information could be incorporated, for instance from GIS shapefiles (see dedicated section below).

3.2 Visualizing data

An essential aspect of spatial analysis lies in visualizing the data. In `geoGraph`, the spatial grids (`gGraph`) and spatial data (`gData`) can be plotted and browsed using a variety of functions.

3.2.1 Plotting `gGraph` objects

Displaying a `gGraph` object is done through `plot` and `points` functions. The first opens a new plotting region, while the second draws in the current plotting region; functions have otherwise similar arguments (see `?plot.gGraph`).

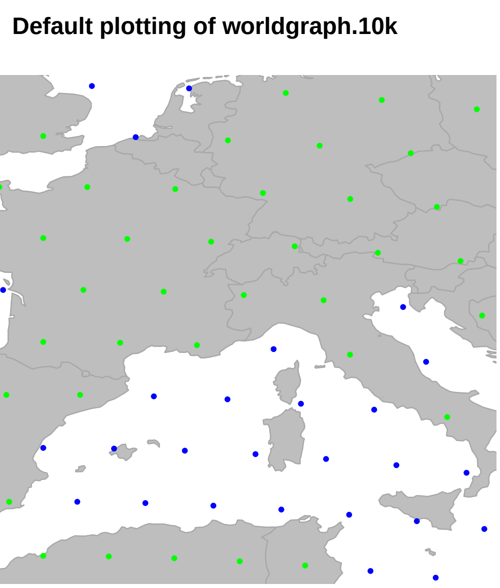
By default, plotting a `gGraph` displays the grid of nodes overlaying a shapefile (by default, the landmasses). Edges can be plotted at the same time (argument `edges`), or added afterwards using `plotEdges`. If the `gGraph` object possesses an adequately formed `@meta$colors` component, the colors of the nodes are chosen according to the node attributes and the color scheme specified in `@meta$colors`. Alternatively, the color of the nodes can be specified via the `col` argument in `plot/points`.

Here is an example using `worldgraph.10k`:

```
> data(worldgraph.10k)
> worldgraph.10k@meta$colors
```

```
          habitat       color
1             sea        blue
2            land       green
3        mountain       brown
4      landbridge light green
5 oceanic crossing  light blue
6  deselected land   lightgray
```

```
> head(getNodesAttr(worldgraph.10k))

  habitat
1     sea
2     sea
3     sea
4     sea
5     sea
6     sea

> table(getNodesAttr(worldgraph.10k))

deselected land            land             sea 
            290            2632            7320 

> plot(worldgraph.10k)
> title("Default plotting of worldgraph.10k")
```



It may be worth noting that plotting `gGraph` objects involves plotting a fairly large number of points and edges. On some graphical devices, the resulting plotting can be slow. For instance, one may want to disable *cairo* under linux: this graphical device yields better graphics than *Xlib*, but at the expense of increase computational time. To switch to *Xlib*, type:

```
> X11.options(type = "Xlib")
```

and to revert to *cairo*, type:

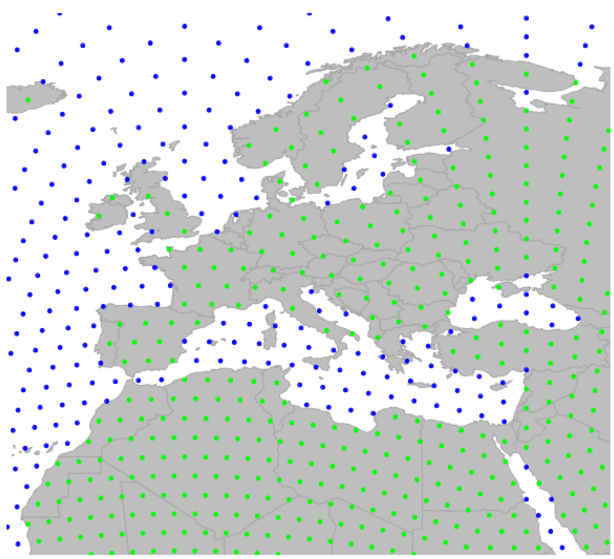
```
> X11.options(type = "cairo")
```

3.2.2 Zooming in and out, sliding, etc.

In practice, it is often useful to be able to peer at specific regions, and more generally to navigate inside the graphical representation of the data. For this, use the interactive functions `geo.zoomin`, `geo.zoomout`, `geo.slide`, `geo.back`, `geo.bookmark`, and `geo.goto`. The zoom and slide functions will require you to left-click on the graphics to zoom in, zoom out, or slide to adjacent areas; in all cases, a right click ends the function. Also note that `geo.zoomin` can accept an argument specifying a rectangular region, which will be adapted by the function to fit best a square area with similar position and centre, and zoom to this area (see `?geo.zoomin`). `geo.bookmark` and `geo.goto` respectively set and go to a bookmark, i.e. a tagged area. This is most useful when one has to zoom to distant areas repeatedly.

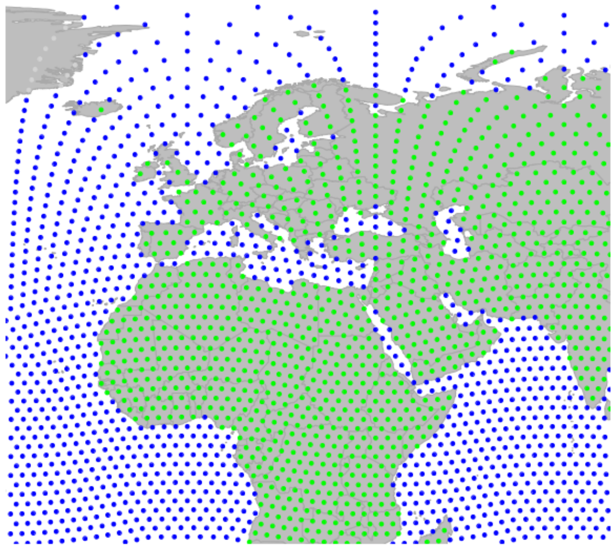
Here are some examples based on the previous plotting of `worldgraph.10k`, but you are best playing with these functions by yourself. Zooming in:

```
> geo.zoomin()
```



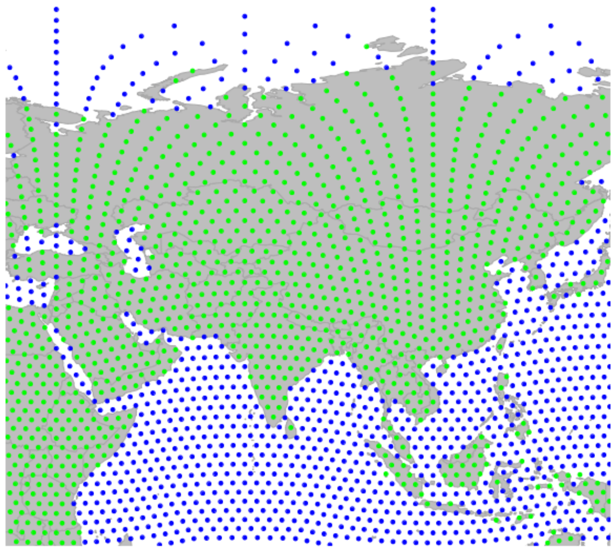
Zooming out:

```
> geo.zoomout()
```



Sliding to the east:

```
> geo.slide()
```



One important thing which makes plotting gGraph objects different from most other plotting in R is that the geoGraph keeps the changes to the plotting area in memory. This allows to undo one or several moves using geo.back. Moreover, even if the graphical device is killed, plotting again a gGraph will use the old parameters by default. To disable this behavior, set the argument reset=TRUE when calling upon plot. Technically, this 'plotting memory' is implemented by storing plotting information in an environment defined as the hidden variable .geoGraphEnv:

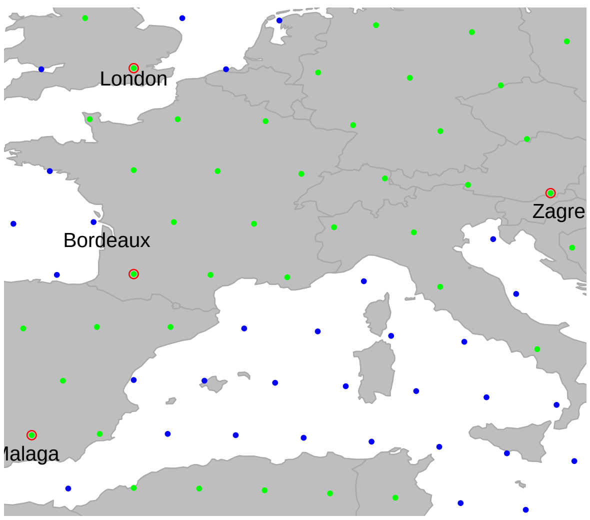
```
> .geoGraphEnv

<environment: 0xc6e8578>

> ls(env = .geoGraphEnv)

[1] "bookmarks"       "last.plot"       "last.plot.param" "last.points"
[5] "psize"           "sticky.points"   "usr"             "zoom.log"

> get("last.plot", .geoGraphEnv)

plot(worldgraph.10k)
```

However, it is recommended not to modify these objects, unless you really know what you are doing. In any case, plotting a gGraph object with argument reset=TRUE will remove previous plotting history and undo possible wrong manipulations.

3.2.3 Plotting gData objects

gData objects are by default plotted in addition to the corresponding gGraph. For instance, using the cities example from above:

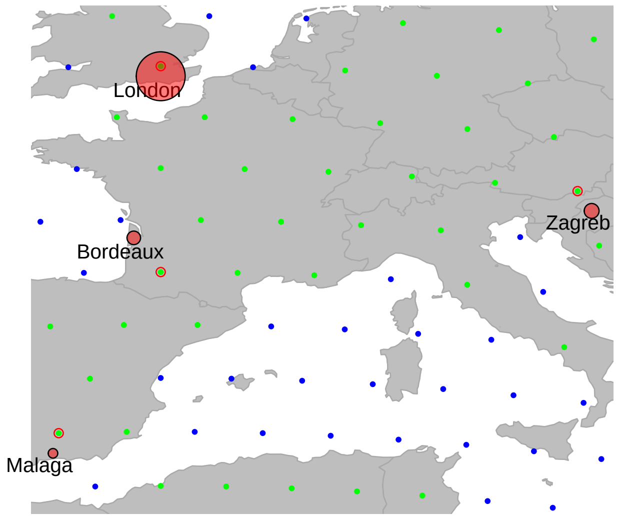
```
> plot(cities, reset = TRUE)
> text(getCoords(cities), rownames(getData(cities)))
```



Note the argument reset=TRUE, which tells the plotting function to adapt the plotted area to the geographic extent of the dataset.

To plot additional information, it can be useful to extract the spatial coordinates from the data. This is achieved by getCoords. This method takes an extra argument original, which is TRUE if original spatial coordinates are seeked, or FALSE for coordinates of the nodes on the grid. We can use this to represent, for instance, the population sizes for the different cities:

```
> plot(cities, reset = TRUE)
> par(xpd = TRUE)
> text(getCoords(cities) + -0.5, rownames(getData(cities)))
> symbols(getCoords(cities)[, 1], getCoords(cities)[, 2], circ = sqrt(unlist(getData(cities))),
+     inch = 0.2, bg = transp("red"), add = TRUE)
```



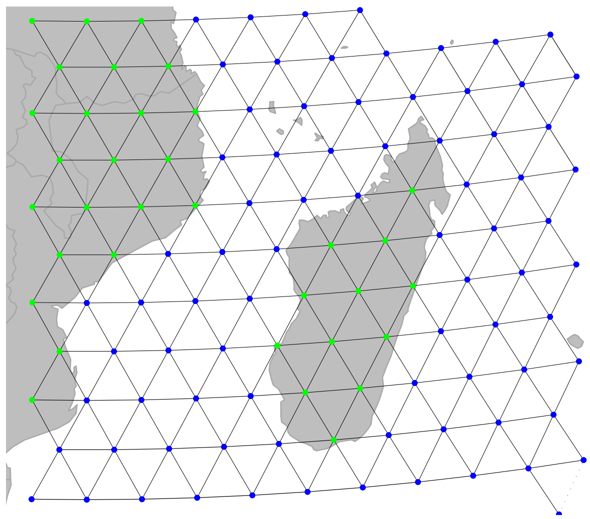
3.3 Editing gGraphs

Editing graphs is an essential task in `geoGraph`. While available `gGraph` objects provide a basis to work with (see `?worldgraph.10k`), one may want to adapt a graph to a specific case. For instance, connectivity should be defined according to biological knowledge of the organism under study. `gGraph` can be modified in different ways: by changing the connectivity, and by changing costs and attribute values.

3.3.1 Changing the connectivity of a gGraph

There are two main ways of chaning the connectivity of a `gGraph`, which match two different objectives. The first approach is to perform global and systematic changes of the connectivity of the graph. Typically, one will want to remove all connections over a given type of landscape, which is uncrossable by the organism under study. Let's assume one is interested in sea fishes. To model fish dispersal, we have to define a graph which connects only nodes falling in seas. We load the `gGraph` object `rawgraph.10k`, and zoom in to a smaller area (Madagascar) to illustrate changes in connectivity:

```
> data(rawgraph.10k)
> geo.zoomin(c(35, 54, -26, -10))
> plotEdges(rawgraph.10k)
```



We shall set a bookmark for this area, in case we would want to get back to this place later on:

```
> geo.bookmark("madagascar")
```

```
Bookmark ' madagascar  'saved.
```

What we now want to do is remove all but sea-sea connections. To do so, the easiest approach is to i) define costs for edges based on habitat, with land being given large costs and ii) remove all edges with large costs.

Costs of a given node attribute (here, 'habitat') are indicated in the `@meta$costs` slot:

```
> rawgraph.10k@meta$costs
```

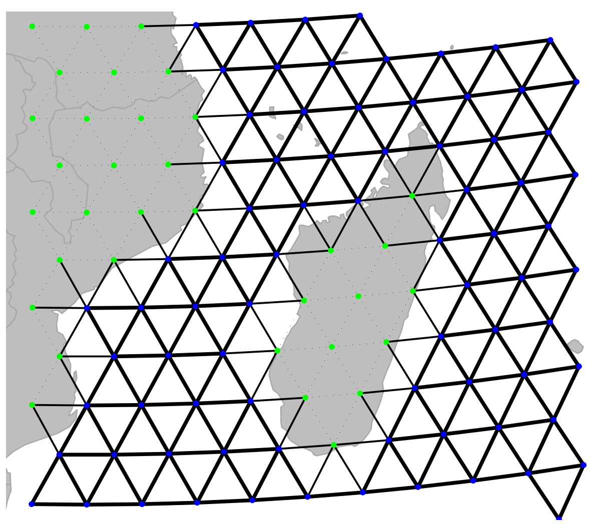
```
          habitat cost
1             sea  100
2            land    1
3        mountain   10
4      landbridge    5
5 oceanic crossing   20
6  deselected land  100
```

```
> newGraph <- rawgraph.10k
> newGraph@meta$costs[2:6, 2] <- 100
> newGraph@meta$costs[1, 2] <- 1
> newGraph@meta$costs
```

```
          habitat cost
1             sea    1
2            land  100
3        mountain  100
4      landbridge  100
5 oceanic crossing  100
6  deselected land  100
```

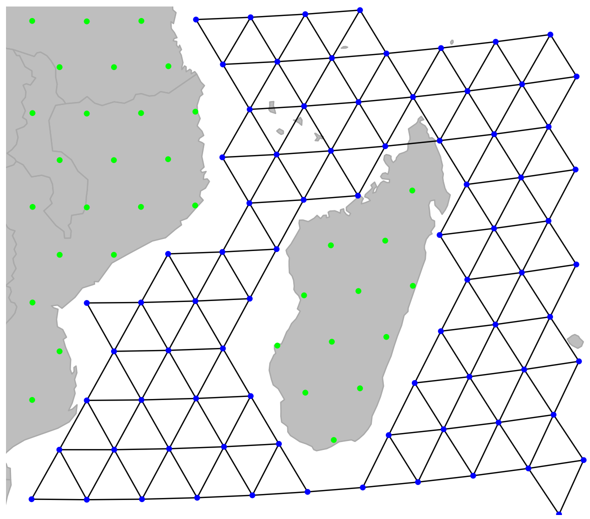
We have just changed the costs associated to habitat type, but this change is not yet effective on edges between nodes. To do so, we use `setCosts`, which sets the cost of an edge to the average of the costs of the nodes:

```
> newGraph <- setCosts(newGraph, attr.name = "habitat")
> plot(newGraph, edge = TRUE)
```



On this new graph, we represent the edges with a width inversely proportional to the associated cost; that is, bold lines for easy travelling and light edges/dotted lines for more costly mouvement. This is not enough yet, since travelling on land is still possible. However, we can tell `geoGraph` to remove all edges associated to too strong a cost, as defined by a given threshold (using `dropDeadEdges`). Here, only sea-sea connections shall be retained, that is, edges with cost 1.

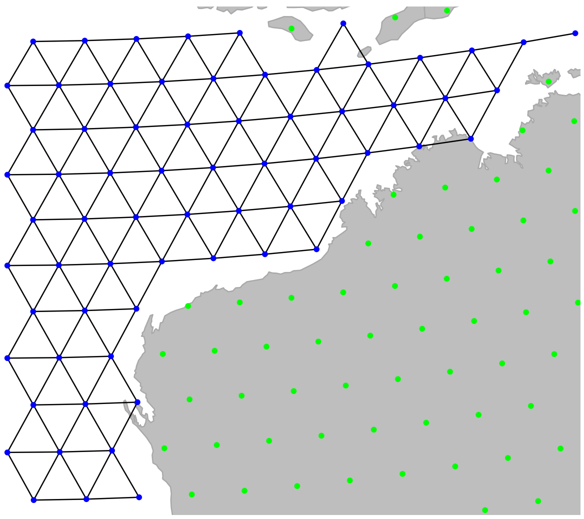
```
> newGraph <- dropDeadEdges(newGraph, thres = 1.1)
> plot(newGraph, edge = TRUE)
```



Here we are: `newGraph` only contains connections in the sea. Note that, despite we restrained the plotting area to Madagascar, this change is effective everywhere. For instance, travelling to the Australian coasts:

```
> geo.zoomin(c(110, 130, -27, -12))
> geo.bookmark("australia")
```

```
Bookmark ' australia  'saved.
```



The second approach is to refine the graph by hand, adding or removing locally some connections. This can be necessary to connect components such as islands to the main landmasses. Apart from adding or removing connections, other possible operations will concern changing costs properties as well as changing attribute values.

3.4 Extracting information from GIS shapefiles

3.5 Finding least-cost paths