RnavGraph: an R package to visualize high dimensional data using graphs as navigational infrastructure

Adrian Waddell and Wayne Oldford

April 11, 2011

Contents

1 Introduction

RnavGraph provides interactive visualization tools for exploring high dimensional space through lower dimensional trajectories, based on the concepts first presented in Hurley and Oldford (2011).

1.1 Getting started

The RnavGraph package is available from the Comprehensive R Archive Network (CRAN). We used extensively S4 classes and the Tcl and Tk API via the tcltk R package. Most of the visualization, both for the graph and the 2d scatterplots, build upon the Tk canvas widget.

RnavGraph depends on the packages: methods, graphics, tcltk, graph and RBGL. However to be able to run all demos and examples we suggest to install the following packages: Rgraphviz, PairViz, scagnostics, rgl, grid, MASS, hexbin, RDRToolbox and, optionally, rggobi. These packages are available through the CRAN and/or Bioconductor R repositories.

Linux users should make sure that tcl and tk are installed. Further the package needs the following libraries (on a Ubuntu 10.04 system): libtk-img, libtk-img-dev, tcl-dev and tk-dev.

Once the package and all of its dependencies have been installed, RnavGraph is loaded using

```
> library(RnavGraph)
```

A closer look into RnavGraph reveals that a relatively large set of functions are available that provide a programmable user interface to the package.

```
> ls("package:RnavGraph")
```

```
 [1] "completegraph"       "linegraph"           "navGraph"
 [4] "newgraph"            "ng_2d"               "ng_2d_ggobi"
 [7] "ng_2d_myplot"        "ng_data"             "ng_get"
[10] "ng_graph"            "ng_image_array_gray" "ng_image_files"
[13] "ng_set"              "ng_set<-"            "ng_update"
[16] "ng_walk"             "plot"                "scagEdgeWeights"
[19] "scagGraph"           "scagNav"             "shortnames"
[22] "shortnames<-"
```

We try to explain all these functions within this vignette, though the R help function should also be used to get a more detailed description on any particular function, as needed. In addition, we provide the RnavGraph package with several demos that may be helpful, via

```
> demo(package = "RnavGraph")
```

The source code of these demos can be found in the system directory shown in the output from

```
> system.file("demo", package = "RnavGraph")
```

Many of the demos require data from the RnavGraphImageData package also written by us.

All said, though, this vignette should itself provide the best introduction to RnavGraph and its full functionality.

1.2 A simple example

We start with a simple example, using the famous Anderson Iris data, to quickly introduce the basic functionality of `RnavGraph` and to give you a feel for its interface. To begin, then, execute the following in R:

```
> library(RnavGraph)
> ng.iris <- ng_data(name = "iris", data = iris[,1:4],
+                    shortnames = c('s.L', 's.W', 'p.L', 'p.W'),
+                    group = iris$Species,
+                    labels = substr(iris$Species,1,2))
```

As the name suggests, `ng_data` sets up a "navgraph" or "ng" data object (more on this later). Its `data` argument takes a `data.frame` identifying the numeric variables to be explored. The Iris data consists of measurements of the `Sepal.Length`, `Sepal.Width`, `Petal.Length`, and `Petal.Width` on 50 flowers of each of three `Species` of Iris. (The above construction uses the argument `shortnames`, instead of the variable names, just to give more compact labelling in the subsequent displays.)

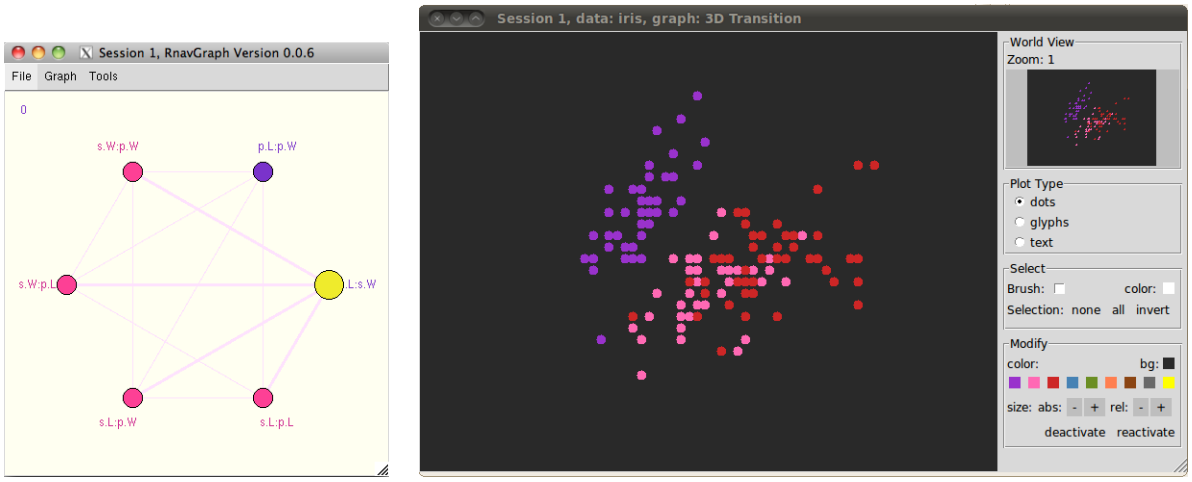
The Iris data provides a set of 150 points in a four-dimensional space; our objective is to visually explore the structure of this data. To begin this exploration, we simply call `navGraph` on the prepared data.

```
> navGraph(ng.iris)
```

Alternatively, if you have `rggobi` working in R and would like to visualize the data via `ggobi`, call

```
> navGraph(ng.iris, settings = list(defaultDisplay = "ggobi"))
```

Either of these calls will produce a *navigation graph* (or *navgraph* for short) as shown in Figure 1(a)



(a) The 3D transition graph (b) The associated display: 2D plot

Figure 1: navGraph on the Anderson Iris data.

and a "visualization" such as that of Figure 1(b). Here, Figure 1(b) is a `2D` display, native to `navGraph`, as produced by the first of the above `navGraph` calls (the second call would produce a `ggobi` session instead). The variates `Sepal.Length` and `Sepal.Width` determine the coordinates of the points in the scatterplot and the `group`, here `Iris$Species`, determine the point colours. If you

select "text" in the `2D` plot, the point symbols will be replaced by the text strings identified by the `labels` argument in the definition of the "ng data" `ng.iris`.

The graph of Figure 1(a), the *navigation graph*, is programmatically linked to the `2D plot` data visualization of Figure 1(b). The large yellow circle on the rightmost node of the graph is called the "bullet"; its position in the graph represents a well-defined location in the high-dimensional data space which, in turn, determines what is actually displayed in the visualization of Figure 1(b). In Figure 1(a), it sits on a node labelled `s.L:s.W` and so Figure 1(b) displays a scatterplot for the variates `Sepal.Length` and `Sepal.Width` ("shortnames" being `s.L` and `s.W`, respectively).

Note how the other nodes are coloured differently, and the edges to them highlighted, depending upon whether they share an edge with the bullet's current node or not. You can move the bullet to a connected node simply by double-clicking on the destination node, by selecting the destination node and scrolling, or by simply selecting and dragging the bullet itself. However you choose to do so, the bullet will dynamically traverse the edge from origin to destination node while, at the same time, the scatterplot display will "follow" the bullet's path smoothly updating its display.

Try it, moving the bullet from the original node `s.L:s.W` to the upper left node `s.W:p.W`. The effect in the scatterplot will be that of rotating the `s.L` axis into that of the `p.W` axis. The shared variate, `s.W` will remain in place.

Now make a number of connected moves in sequence along the graph. As you move the bullet along a path, the scatterplot will continually update via rigid rotations. Note also that each time the bullet arrives at a new node, its potential destination nodes and edges are highlighted anew, and those edges which have already been traversed are coloured slightly different from those which have not yet been visited. Your connected sequence of edges constitutes a path on the navigation graph which, when followed by the bullet, causes the 2D display to update, dynamically following various 3D trajectories through the four dimensional space of the Iris data.

The effect is made more apparent by explicitly choosing the entire path in advance. This is done using "shift-select". Simply hold the shift-key down continuously and select each node (single-click) in turn along a path of your choice. The path will be highlighted as you go. Double-click for the final destination and the bullet will immediately start to walk the path you have identified. This will allow you to focus on the movement in the scatterplot display as the path is walked.

Having just walked a path, you might care to re-run it. To do so, choose the "Tools" menu on the navgraph display and select the menu item "Paths". A new window will open where the path just walked is recorded as the "Active Path". This path may be viewed on the navgraph by pressing the "view" button, or walked again by pressing the "walk" button. The active path may also be saved by pushing the "save" button and, once saved, made active again by double-clicking on the path in the list of saved paths. It is also possible to select paths and type in comments related to the selected path. In this way, the user can record interesting paths, describe the interesting features found, and walk them again at a later date. Simply close the paths display when finished.

(The graphical user interface is described later in much more detail.)

1.3 Underlying graph theory

The graph theory underlying RnavGraph is given in detail in the paper by Hurley and Oldford (2011) and we direct the reader there to gain a fuller appreciation. In this section, we give only a brief summary of some of the ideas.

In the Iris example just considered, the nodes of Figure 1(a) are connected **only if** they share a variate. As a consequence of this, any travel along an edge is consistent with a a rigid rotation through a three dimensional subspace from one two-dimensional space (or scatterplot) to another. Because the edges are restricted in this way, Hurley and Oldford (2011) call such a graph, a *3D transition graph*.

The *complement* of a 3D transition graph is the graph having the same nodes but with edges appearing only between nodes that have **no** variates in common. That is only those edges missing between nodes in Figure 1(a) will appear in its complement. Hurley and Oldford (2011) call this graph the corresponding *4D transition graph*.

In our Iris example, choosing the "Graph" menu from the navgraph display of Figure 1(a) gives a menu of graphs to choose from. Selecting the menu item "iris: 4D" will cause the 4D transition graph to be displayed in place of the 3D transition graph and will also update the scatterplot display.

Since nodes no longer share a variable, moving the bullet along an edge in a 4D transition graph causes the 2D subspace (or scatterplot) at one node to smoothly transition to the 2D subspace (or scatterplot) at the destination node. This is effected by having both axes of one subspace simultaneously transformed into the axes of the other subspace along a geodesic in the four dimensional space defined by all 4 axes. In this way, the path is essentially like that followed in `ggobi` but restricted to the four variates. Walking a path on a 4D transition graph amounts to exploring the entire (possibly higher) dimensional space via four dimensional trajectories.

At this point, you might give the 4D transition a try. You will see that the 4D transitions are not rigid rotations and so might look a little unnatural.

Hurley and Oldford (2011) describe a variety of ways to construct meaningful 3D and 4D transition graphs. Fundamentally, these all begin with a graph on the variates themselves.

Hurley and Oldford (2011) define a variable graph G as any graph, whose nodes are variates, and whose edges indicate an interest (however defined) in the pair of variates each edge joins in the graph G.

For example, if we consider again the Iris data with $p = 4$ variates and assume that all pairs of variates are of equal interest, then G could be the *complete graph* on four variates given by the leftmost graph of Figure 2 (where $A = Sepal.Length$, $B = Sepal.Width$, $C = Petal.Length$, $D = Petal.Width$).

A handy constructor from graph theory is the *line graph* of a graph G denoted by $L(G)$. This and other constructors are given in detail in Hurley and Oldford (2011). Suffice to say here is that every edge in G becomes a node in $L(G)$ and nodes in $L(G)$ have edges between them if and only if the corresponding edges in G meet at a node in G. For our purposes, it is enough to know that $L(G)$

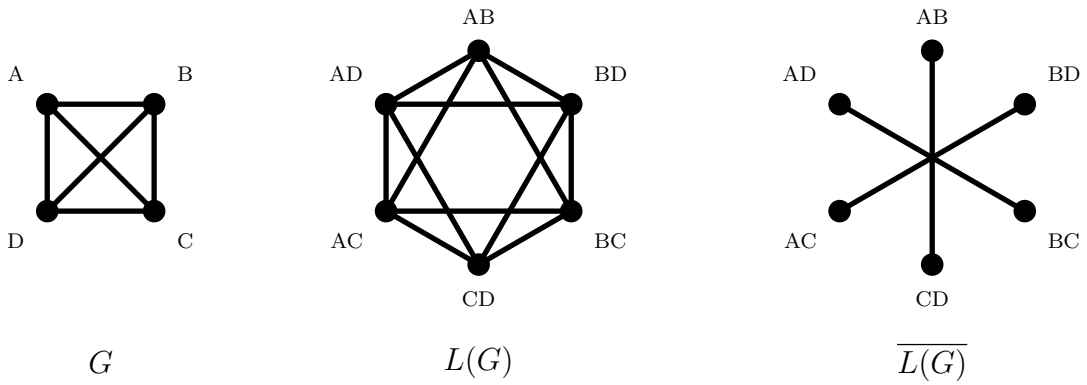


Figure 2: Complete graph G with the individual variables as nodes. Line graph $L(G)$ which is equivalent to the 3d-transition graph. $\overline{L(G)}$ the 4d transition graph.

will be a 3D transition graph, whatever the graph G (provided it is a variable graph). Moreover, the complement of $L(G)$ will be a 4D transition graph. These three graphs are shown for the Iris data in Figure 2.

Another means of constructing 3D and 4D transition graphs given by Hurley and Oldford (2011) is the use of various *graph products*. If the variables –from the data– separate into two sets, say $\mathcal{U} = \{U_1, U_2, ..., U_m\}$ and $\mathcal{V} = \{V_1, V_2, ..., V_n\}$, and for each set there is a corresponding variable graph which connects the pairs of interest, then the *Cartesian product* of the two graphs will produce a 3D transition graph preserving the pairs of interest and its complement, the *tensor product* on graphs will be the corresponding 4D transition graph. Figure 3 shows these, and other, graph products. Although RnavGraph does not yet implement these products, it will in the next release.

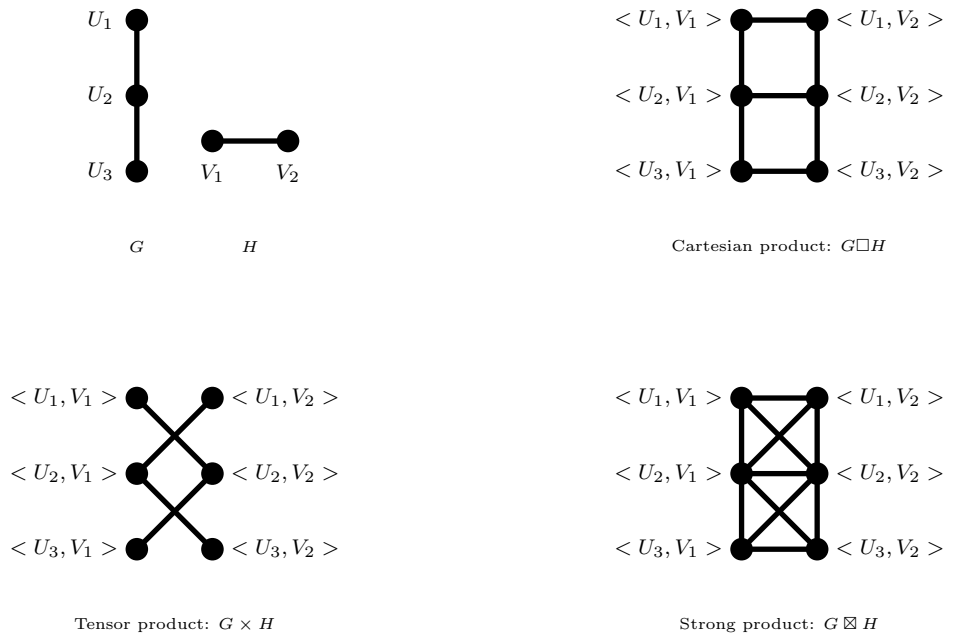


Figure 3: Graph products

Please see Hurley and Oldford (2011) for more details.

1.4 General Structure of `RnavGraph`

Before going further, it is important to realize that neither the underlying theory nor the implementation of `RnavGraph` presupposes that the only visualization of interest is that of rotating scatterplots.

The theory, so far (though Hurley and Oldford, 2011, go beyond 2D nodes), only takes it that each node of a transition graph represents 2D information of some sort and that an edge connects two sets of 2D information which either share a single dimension (a 3D transition) or share no dimension (a 4D transition). When imagining scatterplots at each node, it is easy to see that walking a path on the graph amounts to following a low dimensional trajectory through the higher dimensional space.

`RnavGraph` allows the user to define what visualization the vertices (or nodes) represent and what the transition (along the edges) represent. Moreover, moving a bullet along graph edge simply means updating the visualization with information on the current position between two nodes. What a visualization chooses to do with such information depends entirely on the visualization.

The implementation of RnavGraph is general in that it makes no assumptions about either the views being displayed or the nature of the transitions between them. `RnavGraph` integrates *data to be analyzed*, *a graph to navigate* and *visualization instructions* to be executed as the graph is traversed.

`RnavGraph` provides an S4 class for each of these components to encapsulate the corresponding information, namely the classes `NG_data`, `NG_graph` and `NG_Visualization`, respectively (their associated constructor functions will be explained later). The `navGraph()` function in R takes `graph`, `data` and `visualization` objects as arguments and starts up the corresponding graphical user interface. Figure 4 shows a stylized session.

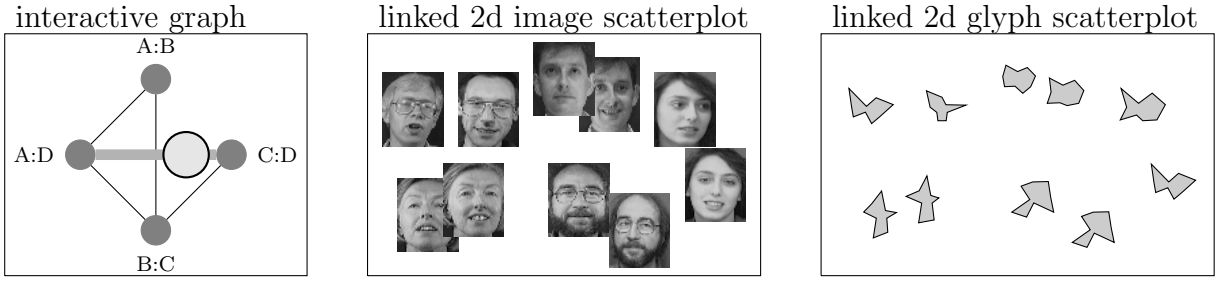


Figure 4: A possible session of `RnavGraph`. Note that here the ball is connected to two sets of visualization instructions. That is, to an scatterplot with images and a scatterplot with glyphs.

A `navGraph` function call returns a *navGraph* handler object which allows the user to control and receive certain states of the running navGraph GUI.

`navGraph` offers different ways to instantiate a session. The most straightforward way is to only define a `NG_data` object and then let `navGraph` set up the default environment with 2d scatterplots

and 3d- and 4d transition graphs. This is what we did in the introductory example.
Alternatively, the user can define the NG_data, NG_graph and NG_Visualization objects themselves and hence have a much finer control and broader set of functionality to access.

In the following subsection, we redo the simple example but, this time, with more detailed control over the constructions.

1.4.1 The simple example, in detail.

In the Iris data example, we simply relied on RnavGraph to fill in the details. Now, we revisit this example; this time filling in the details so as to demonstrate the fine control over graphs, data, and visualization instructions which RnavGraph offers to the user.

The following code may also be found in the ng_2d_iris demo.

```
> demo("ng_2d_iris", package = "RnavGraph")
```

Now, as before, the first step is to create the data object

```
> ng.iris <- ng_data(name = "iris", data = iris[,1:4],
+                    shortnames = c('s.L', 's.W', 'p.L', 'p.W'),
+                    group = iris$Species,
+                    labels = substr(iris$Species,1,2))
```

Note how the Species variable was passed on to the group argument and not to the data argument. Further, the shortnames argument takes alternative variable names. That is, we'd like to use these shortnames to name the vertices of the variable graph.

```
> V <- shortnames(ng.iris)
```

We now create the three graphs presented in the figure 2:

```
> G <- completegraph(V)
> LG <- linegraph(G)
> LGnot <- complement(LG)
```

These three objects are from the graph class

```
> class(G)
```

```
[1] "graphAM"
attr(,"package")
[1] "graph"
```

The RnavGraph package provides, as outlined earlier, a own graph class because navGraph needs additional information

```
> ng.lg <- ng_graph(name = "3D Transition", graph = LG, layout = "circle")
> ng.lgnot <- ng_graph(name = "4D Transition", graph = LGnot, layout = "circle")
```

Note that the names of the graph objects must be unique within all graph objects passed on to navGraph. The same holds for the data names, as navGraph uses these names to link a graph and data via a visualization object.

As the last step, we need to define visualization rules. For the 3d and 4d transition of 2d scatterplots we can use the ng_2d() function as follows

```
> viz3dTransition <- ng_2d(ng.iris, ng.lg)
```

navGraph takes multiple visualization instruction objects, so for demonstration purposes let's define another visualization instruction object

```
> viz4dTransition <- ng_2d(ng.iris, ng.lgnot)
```

In summary, we have now one data object ng.iris, two graph objects ng.lg and ng.lgnot, and two visualization instruction objects viz3dTransition and viz4dTransition. When dealing with multiple objects from the same class, we need to pack them into a list

```
> viz <- list(viz3dTransition, viz4dTransition)
> graphs <- list(ng.lg, ng.lgnot)
```

and finally we can pass them all to the navGraph GUI initializing function

```
> nav <- navGraph(graph = graphs, data = ng.iris, viz = viz)
```

where nav is the navGraph handler.

This example session shows essentially the work flow of using navGraph. The rest of this vignette discusses each step in detail. It also shows a variety of visualizations.

2 Data

Every NG_data object needs a unique name and a data set in the form of a data.frame with solely numeric variables. The shortnames, group and label argument are optional. It is important that the data.frame used only contains numeric variables, as this is the only data type supported for visualization at the moment and it simplifies some navGraph internal procedures. Hence, when trying to pass on non-numeric variables, ng_data() throws an error.

The name argument takes a string which must be unique between all NG_data objects passed on to navGraph. navGraph will, based on the data- and graph- names, link graph and data via the visualization instructions. A minimal working NG_data object can be created with

```
> ng.iris <- ng_data(name = "iris", data = iris[, 1:4])
```

NG_data objects, as most other objects provided by the RnavGraph package, show a summary if you print them with print or just enter their variable name into the R promt

```
> ng.iris
```

```
object from NG_data class.
  name: iris
  data: 150 x 4
    Variable Names | No Short Names
    ---------------------------
    Sepal.Length   |
    Sepal.Width    |
    Petal.Length   |
    Petal.Width    |
  group:  No group variable defined.
  label: labels weren't defined.
```

The variable names of `NG_data` objects, like the variable names of `data.frame` objects, can be displayed and changed with the `names` function:

```
> names(ng.iris)

[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"

> names(ng.iris) <- c("SepL", "SepL", "PetL", "PetW")
```

the same holds true for the shortnames:

```
> shortnames(ng.iris)

character(0)

> shortnames(ng.iris) <- c("s.L", "s.W", "p.L", "p.W")
```

let's see the output summary of `ng.iris`

```
> print(ng.iris)

object from NG_data class.
  name: iris
  data: 150 x 4
    Variable Names | Short Names
    ---------------------------
    SepL           | s.L
    SepL           | s.W
    PetL           | p.L
    PetW           | p.W
  group:  No group variable defined.
  label: labels weren't defined.
```

all the other data stored in a `NG_data` object can be accessed via the `ng_get` and modified via the `ng_set` function.

```
> ng_get(ng.iris)
```

```
Get what? Possible options are: name, data, group, labels
```

```
> ng_get(ng.iris, "data")[1:3, ]
```

```
  SepL SepL PetL PetW
1  5.1  3.5  1.4  0.2
2  4.9  3.0  1.4  0.2
3  4.7  3.2  1.3  0.2
```

```
> ng_get(ng.iris, "name")
```

```
[1] "iris"
```

```
> ng_get(ng.iris, "group")
```

```
numeric(0)
```

```
> ng_get(ng.iris, "labels")
```

```
character(0)
```

Modifying parts from the `NG_data` objects can be achieved using the `ng_set` function

```
> ng_set(ng.iris)
```

```
Replace what? Possible options are: name, data, group, labels
Use ng_set<- to set a value.
```

```
> ng_set(ng.iris, "labels") <- as.character(iris$Species)
```

2.1 group argument

The `group` argument of the `ng_data` function deserves its own subsection, as it maps to the color of the points, images, glyphs, etc. We herefore create a small toy example:

```
> x <- rep(1:30, each = 30)
> y <- rep(1:30, 30)
> ng.test <- ng_data(name = "test", data = data.frame(x = x, y = y,
+     z = 1:900), group = 1:900)
```

This produces a rectangular grid of 900 points where each point has a different group:

```
> nav <- navGraph(ng.test)
```

The output of `navGraph` is shown in figure 2.1. `navGraph` matches the first nine groups to the colors shown in its brushing menu and the rest to the colors returned by the R function `colors()`. Note that `navGraph` only distinguishes between as many groups as `colors()` knows colors. If more groups exist, the remaining points get mapped to the first color in the brushing menu.
If the data gets visualized with `ggobi`, the mapping from the `group` argument to the display items happens as shown in figure 2.1. Use the code

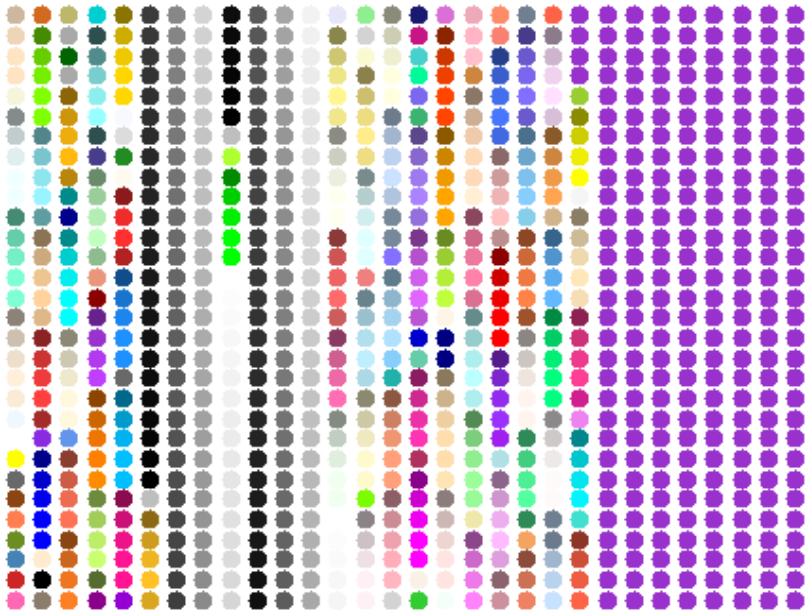


Figure 5: Mapping of the group argument to colors in the default tk2d window.

```
> nav <- navGraph(ng.test, settings = list(defaultDisplay = "ggobi"))
```

herefore.
For the tk2d display, the group variable could also be a vector of strings contained in the colors() output. For example

```
> x <- rep(1:3, each = 3)
> y <- rep(1:3, 3)
> ng.test <- ng_data(name = "test2", data = data.frame(x = x, y = y,
+     z = 1:9), group = c("red", "red", "green", "blue", "blue",
+     "blue", "blue", "orange", "orange"))
> nav <- navGraph(ng.test)
```

yield the output shown in figure 2.1.

3 Graph

Defining an NG_graph object happens in two stages. First, a graph object from the graph package has to be created. Second, a NG_graph object gets created from the graph object. This has the advantage, that a R standard graph gets created using the full flexibility provided by other packages building on the graph package such as the RGBL and Rgraphviz package.

3.1 The graph package

We recommend to take a brief look at the following vignettes in order to get an overview of some of the functionality of R handling graphs.

```
> vignette(package = "graph")
> vignette(package = "RBGL")
> vignette(package = "Rgraphviz")
```

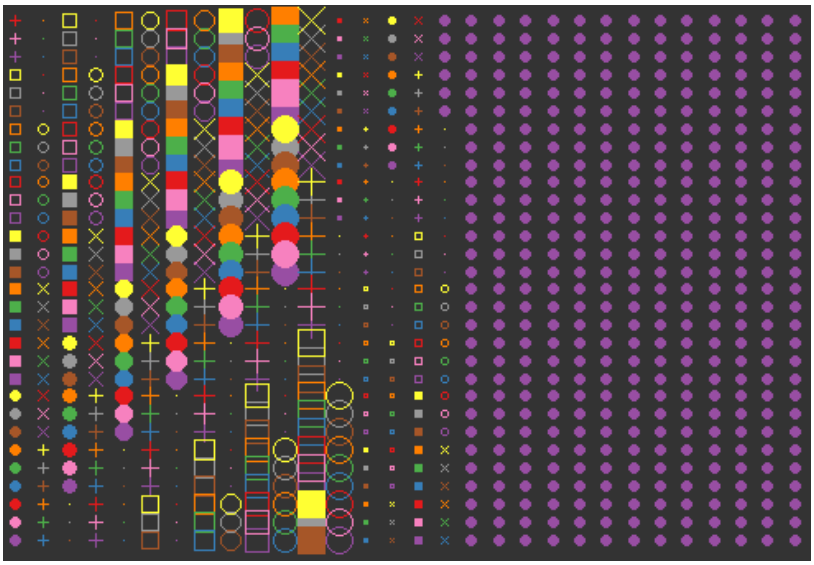


Figure 6: Mapping of the `group` argument to colors in the ggobi dispaly.

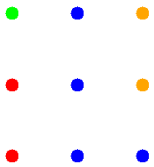


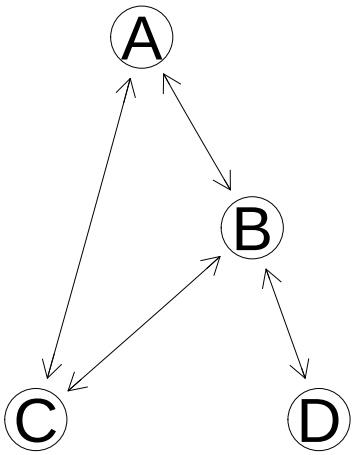
Figure 7: Mapping of the `group` argument containing color names.

The `graph()` package does, to our understanding, not deliver any simple creator functions, that is the user is left to use the `new()` creator function.

```
> adjM <- matrix(c(0, 4, 1, 0, 2, 0, 3, 2, 2, 2, 0, 0, 0, 2, 0,
+     0), ncol = 4)
> rownames(adjM) <- c("A", "B", "C", "D")
> colnames(adjM) <- c("A", "B", "C", "D")
> G <- new("graphAM", adjMat = adjM, edgemode = "directed")
> G

A graphAM graph with directed edges
Number of Nodes = 4
Number of Edges = 8
```

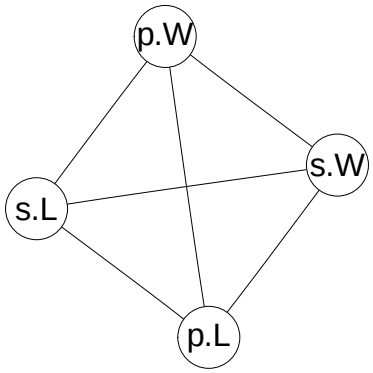
If the `Rgraphviz` package is installed, one can plot a `graph` object

```
> library(Rgraphviz)
> plot(G)
```



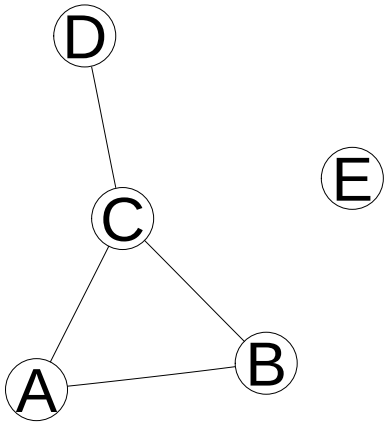
We have written some functions which ease the graph creation part. Often the users want to start from a complete variable graph. We provide the `completegraph` function which takes a vector of node names as the argument

```
> V <- c("s.L", "s.W", "p.L", "p.W")
> G <- completegraph(V)
> plot(G, "neato")
```



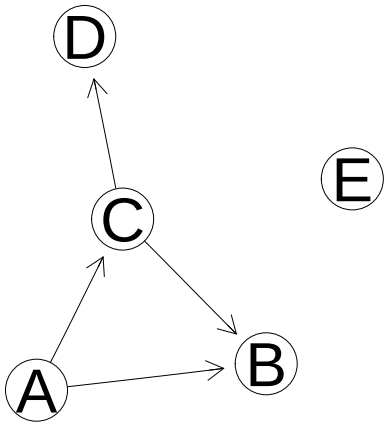
We also introduce the `newgraph` R function which either takes an adjacency- or a from-to-edge matrix to create a graph

```
> from <- c("A", "A", "C", "C")
> to <- c("B", "C", "B", "D")
> ftEmat <- cbind(from, to)
> G <- newgraph(nodeNames = LETTERS[1:5], mat = ftEmat)
> plot(G, "neato")
```



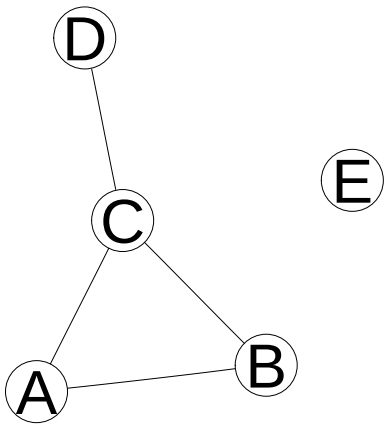
Note how the node "E" was added to the graph. Graphs can also be directed, however `RnavGraph` does not constrain the bullet in direction if an edge exists. Hence this feature is for the current `RnavGraph` version not from importance, however for completeness

```
> G <- newgraph(nodeNames = LETTERS[1:5], mat = ftEmat, directed = TRUE)
> plot(G, "neato")
```



The from-to-edge matrix can also be numeric

```
> from <- c(1, 1, 3, 3)
> to <- c(2, 3, 2, 4)
> ftEmat <- cbind(from, to)
> G <- newgraph(nodeNames = LETTERS[1:5], mat = ftEmat)
> plot(G, "neato")
```



Weights can also added to Graphs via the weights argument. Note however that `navGraph` does not visualize them, however they might be important for greedy path finding algorithms

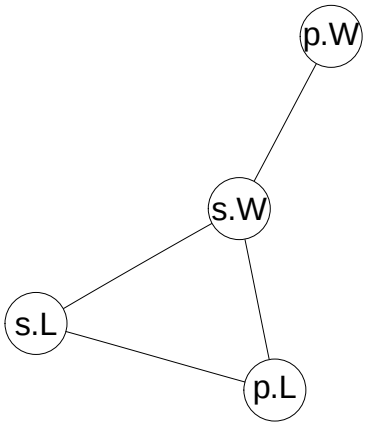
```
> weights <- c(2, 1, 3, 4)
> G <- newgraph(nodeNames = LETTERS[1:5], mat = ftEmat, weights = weights)
> edgeData(G, attr = "weight")

$`A|B`
[1] 2

$`A|C`
[1] 1

$`B|A`
[1] 2

$`B|C`
[1] 3

$`C|B`
[1] 3

$`C|D`
[1] 4

$`C|A`
[1] 1

$`D|C`
[1] 4
```

Or alternatively you can add weights after the graph creation process

```
> G <- newgraph(nodeNames = LETTERS[1:5], mat = ftEmat, weights = weights,
+     directed = TRUE)
> edgeData(G, attr = "weight")

$`A|B`
[1] 2

$`A|C`
[1] 1

$`C|B`
[1] 3

$`C|D`
[1] 4

> edgeData(G, from = "A", to = "B", attr = "weight")

$`A|B`
[1] 2

> edgeData(G, from = "A", to = "B", attr = "weight") <- 8
```

As mentioned earlier, the `newgraph` function takes also adjacency matrices as an argument, use the argument `isAdjacency=TRUE`

```
> adjM <- matrix(c(0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0,
+     0), ncol = 4)
> all(adjM == t(adjM))

[1] TRUE

> G <- newgraph(nodeNames = V, mat = adjM, isAdjacency = TRUE)
> plot(G, "neato")
```



Note that all the graphs from the `graph` class have the function `nodes` and `edges` defined

```
> nodes(G)
[1] "s.L" "s.W" "p.L" "p.W"
> edges(G)
$s.L
[1] "s.W" "p.L"

$s.W
[1] "s.L" "p.L" "p.W"

$p.L
[1] "s.L" "s.W"

$p.W
[1] "s.W"
```

If you use an adjacency matrix to create a graph, you can also pass on a weight matrix

```
> adjM <- matrix(c(0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0,
+     0), ncol = 4)
> weightsM <- matrix(c(0, 0, 5, 0, 2, 0, 1, 3, 0, 0, 0, 0, 0, 7,
+     0, 0), ncol = 4)
> G <- newgraph(nodeNames = V, mat = adjM, weights = weightsM,
+     directed = TRUE, isAdjacency = TRUE)
> edgeData(G, attr = "weight")
$`s.L|s.W`
[1] 2

$`s.W|p.W`
[1] 7

$`p.L|s.L`
[1] 5

$`p.L|s.W`
[1] 1

$`p.W|s.W`
[1] 3
```

Further the `ftM2adjM` function converts a from-to-edge matrix into a adjacency matrix

```
> ftM2adjM(ftEmat)
  1 3 2 4
1 0 1 1 0
3 0 0 1 1
2 0 0 0 0
4 0 0 0 0
```

3.1.1 linegraph and its complement

After creating a variable graph, getting its linegraph and the complement of the linegraph yields the 3d- and 4d transition graph. We provide the `linegraph` and the `graph` package provides the `complement` function herefore. The linegraph has the separator `sep` argument to distinguish the node names. You must chose a string that does not occur within the node names.

```
> G <- completegraph(V)
> LG <- linegraph(G, sep = "::")
> nodes(LG)
```

```
[1] "s.L::s.W" "s.L::p.L" "s.L::p.W" "s.W::p.L" "s.W::p.W" "p.L::p.W"
```

Notice, if G had edge weights, then they are lost now. Also the line graph is only defined for undirected graphs at the moment.

3.2 The `NG_data` object

The user can get his graphs in whatever ways he want. In the end, the node names must be in sync with the shortnames or names of the `NG_data` object, and the graph must be from the `graph` class and subsequently the `NG_graph` class.

It stays to create an `NG_data` object

```
> ng.LG <- ng_graph(name = "3D Transition", graph = LG, sep = "++",
+     layout = "circle")
```

```
[ng_graph]: warning, sep does not occur in some node names
```

```
> ng.LG
```

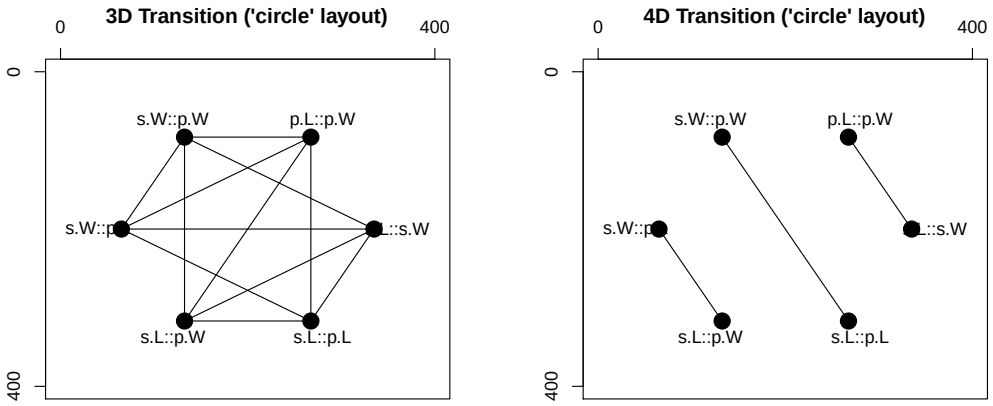
```
NG_graph object from ng_graph()
name: 3D Transition
layout: circle
```

Note that you have to specify the `sep` argument again. The reason herefore is that you could pass on any graph with any dimensionality of node names (e.g. "A:B:C:D" for four dimensions). In addition, you can specify the graph layout. Currently `circle`, `random` (currently not working), `kamadaKawaiSpring` and `fruchtermanReingold` are possible options for the layout. We use the `RBGL` package to get the latter two layouts. The complement does not need any special considerations

```
> LGnot <- complement(LG)
> ng.LGnot <- ng_graph(name = "4D Transition", graph = LGnot, sep = "::",
+     layout = "circle")
```

The name of each `NG_graph` object, as for the `NG_data` object, must be unique within all graphs passed on to the `navGraph` function. `NG_graph` objects can be plotted

```
> par(mfrow = c(1, 2))
> plot(ng.LG)
> plot(ng.LGnot)
```



As for the `NG_data` object, you can access or modify with the `ng_get` and `ng_set` function, respectively.

```
> ng_get(ng.LG)
```

```
Get what? Possible options are: name, graph, visitedEdges, layout
```

```
> ng_set(ng.LG, "name") <- "3d transition graph"
```

Careful, some things are linked: graph, visitedEdges and layout. Changing one should affect the whole object (which it does not at the moment).

4 Visualization Instructions

Once the `NG_data` and `NG_graph` objects have been defined, they have to be connected with some visualization instructions. That is, the visualization instructions tell `navGraph` what and how `navGraph` should visualize when the ball gets moved along the edges of the graph. Conceptually, `navGraph` allows for any visualization of the data, as long the user can implement them. That is, there are no constrains to dimensionality of data represented by a node or the "morphing" defined along an edge. This feature is not documented yet, however you can achieve this with the `ng_2d_myplot` function. However the `RnavGraph` package implements the 2d-scatterplot example for the moment and provides an interface for the user to implement their own –possibly new– ideas.

For the 2d- scatterplot example, we need a device that plots some objects at the x and y euclidean coordinate. The objects could be either dots, images, glyphs, text or anything else you can imagine (and implement). There are many plotting devices available in `R` (e.g. traditional graphics, grid, rgl and ggobi) and `navGraph` can deal with most of them. However when it comes to speed and double buffering, only few of them produce satisfactory output. In addition, different devices allow for different objects to be plotted. We have implemented a new device, lets call it `tk2d`, that builds upon the tk canvas widget (so does the interactive graph). Alternatively to `tk2d` one can use the `rggobi` package and `ggobi` to visualize the scatterplots. `ggobi` however does only allow to visualize dots, rectangles and crosses in different colors. `Ggobi` has not been maintained for a while and hence does not run well on all to my knowledge? current operating systems. For what follows, we will give a detailed explanation of each device. Most likely you only want to read the `tk2d` and `ggobi` subsection and then return to this section once you have more custom needs.

4.1 tk2d

The tk2d device can either display dots, images, glyphs or text (strings).

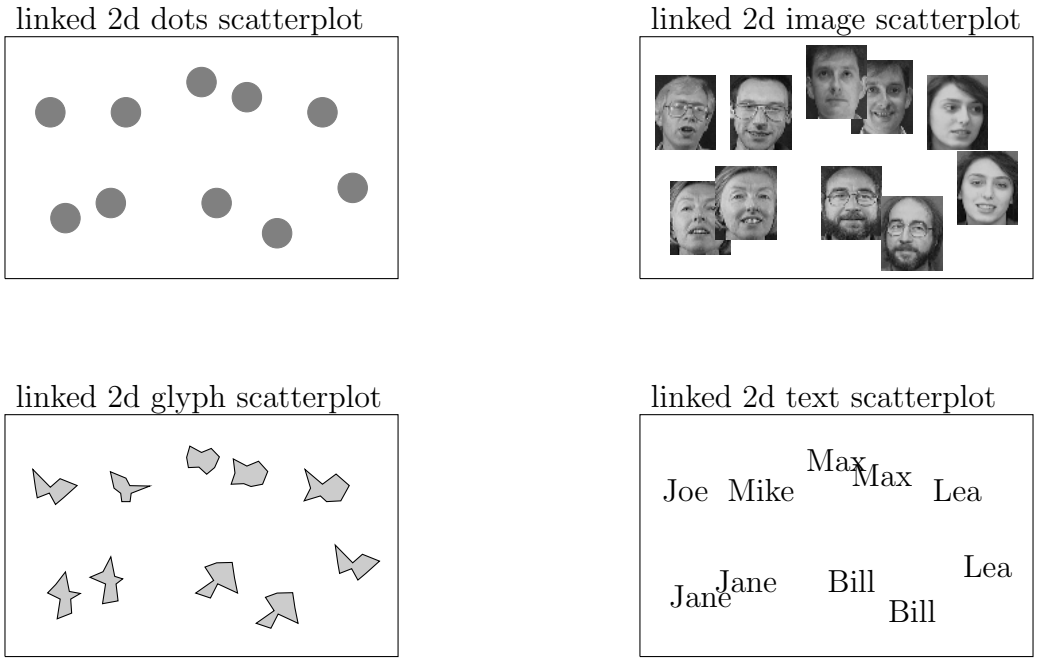


Figure 8: tk2d can visualize dots, images, glyphs and text.

Once the tk2d device is started, there is a whole set of interaction tools like brushing, changing colors, resizing images etc. We will discuss them in the "Starting a navGraph Session" section. For now lets define some visualization instructions that use the tk2d device and only show dots

```
> viz1 <- ng_2d(data = ng.iris, graph = ng.lg)
> viz1

tk2d scatter plot: ng_2d()
Graph: 3D Transition
Data: iris
```

Notice the output of the viz1 object. It says that it addresses the tk2d device which displays a scatterplot and that it connects the NG_graph object with the name "3D Transition" with the NG_data object with the name "iris". Hence if you weren't to choose the graph and data names unique, navGraph would not know how to link them correctly. navGraph has no bound on the number of visualization instructions you can pass on as an argument. So lets define another tk2d visualization object

```
> viz2 <- ng_2d(data = ng.iris, graph = ng.lgnot)
```

4.1.1 Working with Images (from some text source)

The main reason we implemented the tk2d device is because we wanted the RnavGraph package to be useful to explore image data. All images get imported as tcl image objects since they later will

be displayed on a tk canvas widget. This has the implication that the user needs a running `Img` tcl extension set up for the tcltk version R connects to via the `tcltk` R package. If the images however eventually exist as an R object of some sort (e.g. a matrix), the tcl `Img` extension is not needed. For the examples in this vignette and for some of the demos, we require the `RnavGraphImageData` package.

```
> library(RnavGraphImageData)
```

you can get an overview of the data provided by the package with

```
> data(package = "RnavGraphImageData")
```

Much of this data is from the webpage of Sam Roweis http://www.cs.nyu.edu/~roweis/data.html.

Lets start with the USPS `digits` image data

```
> data(digits)
> dim(digits)
```

```
[1]   256 11000
```

from the help documentation (not done yet) for the data set

```
> help("digits")
```

we get that the `digits` data consists of 16×16 8-bit grayscale images of "0" through "9"; 1100 examples of each class. That is, one image is stored in one column (which is often the case in the machine learning field). For example the data of one handwritten 8 is

```
> matrix(digits[, 7 * 1100 + 1], ncol = 16, byrow = FALSE)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12] [,13] [,14] [,15] [,16]
 [1,]    0    0    0    7   71  156  156  156  156   125    37     0     0     0     0     0
 [2,]    0   19  170  217  255  255  255  252  252   255   173     8     0     0     0     0
 [3,]    8  255  255  247  186  120   87   66   66    87   144   161     0     0     0     0
 [4,]    7  244  255  127    0    0    0    0    0     0    44    34     0     0    24    13
 [5,]    0  100  255  246  166   27    0    0    0     0     0     0    32   173   214    88
 [6,]    0    1   85  240  255  224   41    0    0    30   119   174   138   121    15     0
 [7,]    0    0    0   61  255  255  190   94  121   242   255   220    91     5     0     0
 [8,]    0    0    0    1  117  255  255  255  255   185   105    18     0     0     0     0
 [9,]    0    0    1   79  237  250  253  255  228    32     0     0     0     0     0     0
[10,]    0   34  129  255  200   31   86  238  255   131     0     0     0     0     0     0
[11,]    4  193  253  163   13    0    0  129  255   213    21     0     0     0     0     0
[12,]   93  255  195    0    0    0    0   20  213   255    81     0     0     0     0     0
[13,]  131  255  123    0    0    0    0    8  202   255   153     0     0     0     0     0
[14,]   89  255  249  188  124   88   88  178  255   228    30     0     0     0     0     0
[15,]    0   17  172  255  255  255  255  255  243    49     0     0     0     0     0     0
[16,]    0    0    5   67  154  154  154  130   40     0     0     0     0     0     0     0
```

Hence the gray scales are coded from 0 to 255. Since visualizing 11000 digits is computationally expensive, we will continue with a sub-sample of the data

```
> sel <- sample(x = 1:11000, size = 600)
> p.digits <- digits[, sel]
```

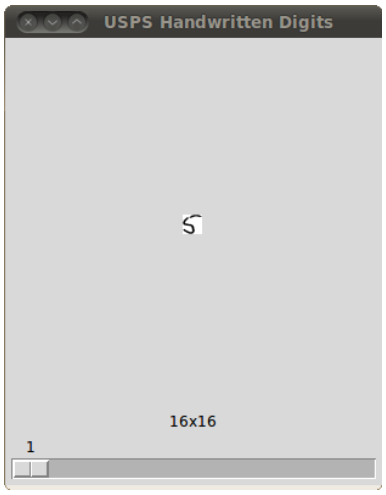
In order for `tk2d` to be able to display the images, they need to be wrapped into a `NG_image` object. Now, contrary to all the `NG_...` objects we have encountered so far, the `NG_image` class does not wrap the image data within itself. The `NG_image` object contains only the names (character strings) of the images that (the names) tcl did automatically assign. That is, when importing an image into tcl, tcl names them consecutively, i.e. `image1`, `image2`, ..., so that they can readily be displayed when needed. Coming back to our `digits` example, the `ng_image_array_gray` R function creates a `NG_image` object from a `data.frame` object with values ranging from 0 to 255.

```
> ng.i.digits <- ng_image_array_gray("USPS Handwritten Digits",
+     p.digits, 16, 16, invert = TRUE, img_in_row = FALSE)
```

the images can be seen with

```
> ng.i.digits
```

which would yield something like (it's random because of the sampling!)



where the scroll bar can be used to navigate through all the images. The name you specified does not have to be unique since here the image ids are the only important part. The name is only used for the `print` output of the `NG_image` object. We also need data on the images. Many scenario are possible, i.e. we could have measured data for each image or we could perform some kind of feature extraction for each image or we reduce the dimensionality (number of pixel) of each image to something manageable, say 4 or 5 dimension –rather than the commonly conveniently chosen two dimensions–. For the rest of this vignette, we choose the dimensionality reduction approach using isomap provided in the `vegan` (or also by the `RDRToolbox`) R package.

```
> library(vegan)
```

We need to transpose the `digit` data first, since each image is saved as one column and not within a row (the statistics- and computer science field seem to disagree on a common convention).

```
> p.digitsT <- t(p.digits)
```

Next, we need to generate a distance matrix (euclidean distance of one image to another) and perform isomap on the distances

```
> dise <- vegdist(p.digitsT, method = "euclidean")
> ord <- isomap(dise, k = 8, ndim = 6, fragmentedOK = TRUE)
```

Next we create an `NG_data` object

```
> digits_group <- rep(c(1:9, 0), each = 1100)
> ng.iso.digits <- ng_data(name = "ISO_digits", data = data.frame(ord$points),
+     shortnames = paste("i", 1:6, sep = ""), group = digits_group[sel],
+     labels = as.character(digits_group[sel]))
```

and the NG_graph objects

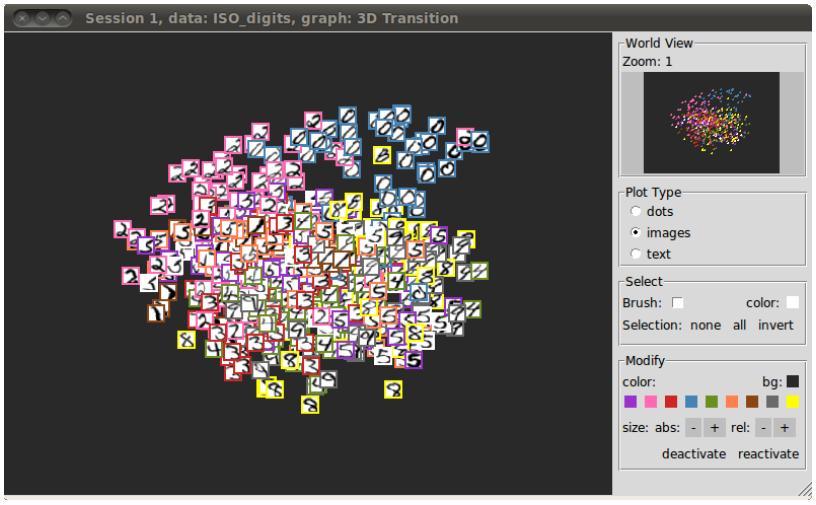
```
> V <- shortnames(ng.iso.digits)
> G <- completegraph(V)
> LG <- linegraph(G)
> LGnot <- complement(LG)
> ng.LG <- ng_graph(name = "3D Transition", graph = LG)
> ng.LGnot <- ng_graph(name = "4D Transition", graph = LGnot)
```

And finally, the visualization instruction allow for adding the images to the tk2d plot

```
> vizDigits1 <- ng_2d(data = ng.iso.digits, graph = ng.LG, images = ng.i.digits)
> vizDigits2 <- ng_2d(data = ng.iso.digits, graph = ng.LGnot, images = ng.i.digits)
```

Starting a navGraph session is simple (but in more detail discussed later on)

```
> nav <- navGraph(data = ng.iso.digits, graph = list(ng.LG, ng.LGnot),
+     viz = list(vizDigits1, vizDigits2))
```



For other examples of importing greyscale images from txt data file see the ng_2d_images_alpha_letter, ng_2d_images_alpha_letter and ng_2d_images_umist_faces demos.

4.1.2 Working with color Images (jpeg, png, etc...)

If your image data source is a bmp, gif, ico, jpeg, pcx, pixmap, png, ppm, postscript, sgi, sun, tga, tiff, window, xbm or xpm file, the ng_image_files R function helps you to import the images into the tcl layer provided the Img tcl extension works properly. If you need to precess the image data in R, the R packages 4.1.2 provide some functionality, however not for all the formats the Img tcl extension is capable of importing. Note however with the shell R command and an installed Imagemagik you can convert any images to almost any format you could wish for.

```
> shell("convert image.png image.jpg")
```

We work with a data set from the Library of Amsterdam http://staff.science.uva.nl/~aloi/. The download is the "Quater resolution (192×144)" which whose size is 60MB and contains $1000 - 250$ objects with each three different viewing angles 2250 images. We have attached a resized version ($38x29$ pixels) in the RnavGraphImageData package in order to save space.

Function	Package	Image Format	Returned Object
`read.pnm`	`pixmap`	pbm, pgm, ppm	objects of diverse pixmap classes
`read.jpeg`	`rimage`	jpeg	image.matrix object
`readRiff`	`rtiff`	tiff	pixmap objcet
`readPNG`	`png`	png	array

Table 1: Table (almost) directly from the useR book "Morphometrics with R" page 33.

The path to the images is

```
> imgPath <- system.file("aloi_small", package = "RnavGraphImageData")
> aloi_images <- list.files(path = imgPath, full.names = TRUE)
> length(aloi_images)
```

```
[1] 2250
```

```
> aloi_images[1:5]
```

```
[1] "/usr/local/lib/R/site-library/RnavGraphImageData/aloi_small/1000_c.png"
[2] "/usr/local/lib/R/site-library/RnavGraphImageData/aloi_small/1000_l.png"
[3] "/usr/local/lib/R/site-library/RnavGraphImageData/aloi_small/1000_r.png"
[4] "/usr/local/lib/R/site-library/RnavGraphImageData/aloi_small/251_c.png"
[5] "/usr/local/lib/R/site-library/RnavGraphImageData/aloi_small/251_l.png"
```

We sample 400 images, so that we do not have to deal with all 2250 images.

```
> sel <- sample(1:length(aloi_images), replace = FALSE)
> p.aloi_images <- aloi_images[sel]
```

Note that the `rimage` package needs the `fftw-dev` (in ubuntu) package to perform Fast Fourier Transformations.

Now, lets import the images first into the tcl layer, herefore the universal function (for jpg,png, ...) is `ng_image_files`:

```
> ng.i.objects <- ng_image_files(name = "ALOI objects", path = p.aloi_images)
```

and again, you can see the images using

```
> ng.i.objects
```

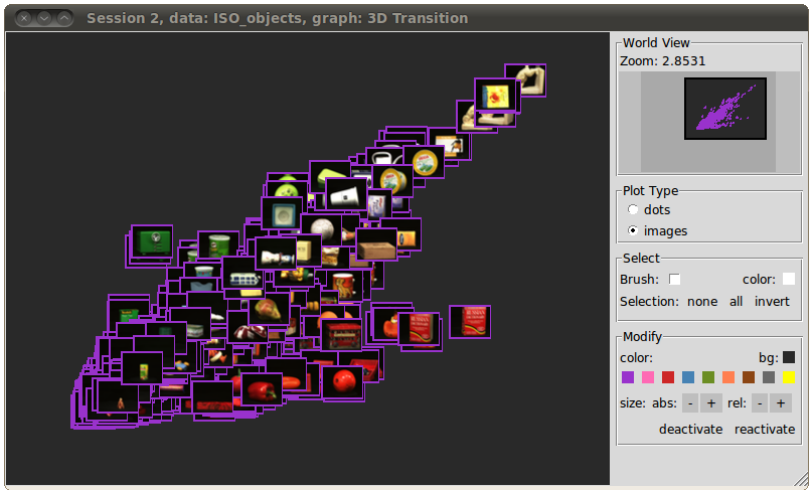
In order to work with the image date we use the `readPNG` function from the `png` package. `readPNG` returns a three dimensional array with the red-blue-green contents for each pixel. For the ease of our demonstration we just get for each image the sum of each, red,green,blue, content

```
> library(png)
> imgData <- t(sapply(p.aloi_images, FUN = function(path) {
+     x <- readPNG(path)
+     r <- sum(x[, , 1])
+     g <- sum(x[, , 2])
+     b <- sum(x[, , 3])
+     return(c(r, g, b))
+ }))
```

Initializing a `navGraph` session on this data with the images stays the same as in the previous example

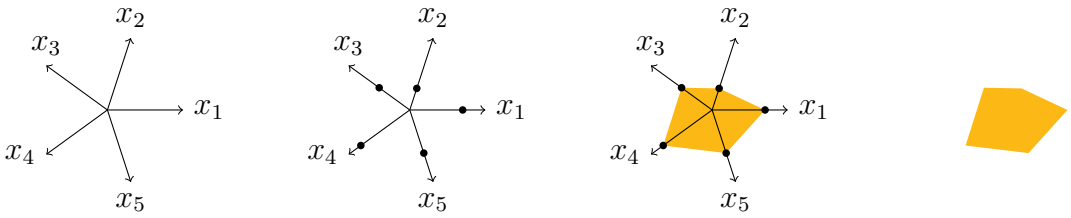
```
> ng.iso.objects <- ng_data(name = "ISO_objects", data = data.frame(imgData),
+     shortnames = paste("i", 1:3, sep = ""))
> V <- shortnames(ng.iso.objects)
> G <- completegraph(V)
> LG <- linegraph(G)
> LGnot <- complement(LG)
> ng.LG <- ng_graph(name = "3D Transition", graph = LG)
> ng.LGnot <- ng_graph(name = "4D Transition", graph = LGnot)
> vizObjects1 <- ng_2d(data = ng.iso.objects, graph = ng.LG, images = ng.i.objects)
> vizObjects2 <- ng_2d(data = ng.iso.objects, graph = ng.LGnot,
+     images = ng.i.objects)
> nav <- navGraph(data = ng.iso.objects, graph = list(ng.LG, ng.LGnot),
+     viz = list(vizObjects1, vizObjects2))
```

and you should expect to see something like



4.1.3 Working with Star Glyphs

A star glyph is a visual representation of one data entity. If the entity has p-dimensional data associated, say $x_i = (x_{11}, x_{12}, ..., x_{1p})$, the glyph gets constructed as the following graphic outlines:



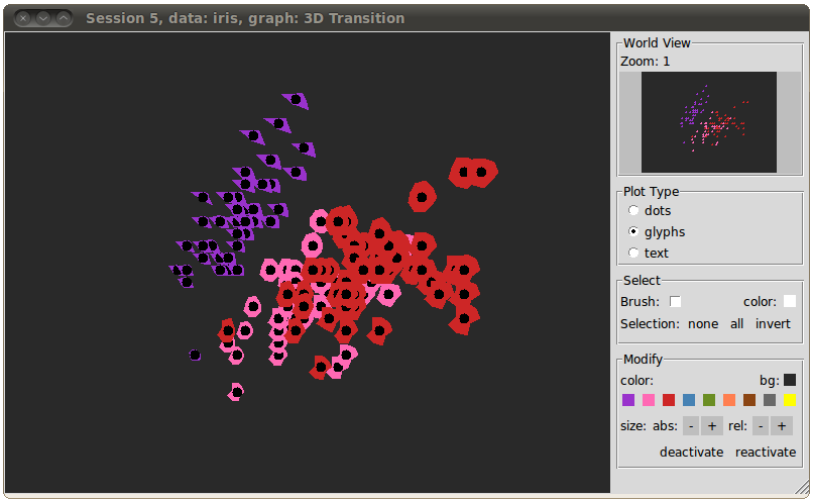
We decided to use the last image as our star glyph visualization. We take a look at glyphs using the `iris` data example. Data and graph object have been defined previous in this vignette. It just stays to define the visualization instructions with the glyph definition

```
> vizGlyph1 <- ng_2d(data = ng.iris, graph = ng.lg, glyph = names(ng.iris)[c(1,
+     2, 3, 4, 3, 2, 3, 4, 1)])
> vizGlyph2 <- ng_2d(data = ng.iris, graph = ng.lgnot, glyph = shortnames(ng.iris)[c(1,
+     2, 3, 4, 3, 2, 3, 4, 1)])
> vizGlyph2
```

```
tk2d scatter plot: ng_2d()
Graph: 4D Transition
Data: iris
```

Note that you can either use the `shortnames` or the `names` to define the glyph sequence. It stays to call `navGraph`

```
> nav <- navGraph(ng.iris, list(ng.lg, ng.lgnot), list(vizGlyph1,
+     vizGlyph2))
```



4.1.4 Working with Text

When defining the `NG_data` object, one can specify the `labels` argument (character) which directly maps to the text radiobutton in the `tk2d` display.

4.2 ggobi

Ggobi allows one to use different elements of data visualization and interaction then the `tk2d` display (e.g. parallel coordinate plots). Ggobi is superior to the `tk2d` display if one deals with large data sets. However ggobi does not visualize images nor glyphs. Visualization instructions which address the ggobi device can be created with the `ng_2d_ggobi` command. Let's demonstrate this using the `iris` data example

```
> vizGgobi1 <- ng_2d_ggobi(data = ng.iris, graph = ng.lg)
> vizGgobi2 <- ng_2d_ggobi(data = ng.iris, graph = ng.lgnot)
```

and finally start `navGraph`

```
> nav <- navGraph(ng.iris, list(ng.lg, ng.lgnot), list(vizGgobi1,
+     vizGgobi2))
```

or as shown earlier

```
> nav <- navGraph(ng.iris, settings = list(defaultDisplay = "ggobi"))
```

4.3 Custom visualization instructions

4.3.1 traditional graphic device

If you want to create your own visualization from scratch instead of using for example the `tk2d` display, you can define a display function that takes any subset of the following as arguments

argument	description
x	x-coordinate
y	y-coordinate
group	group slot from `NG_data` object
labels	labels slot from `NG_data` object
order	order of points. In 3d rigid rotation, the order increases with the distance of a the point from the viewer.
from	node name the bullet moves from
to	node name the bullet moves to
percentage	in between percentage of bullet
data	data name of `NG_data` object

Let's give an example with the traditional graphic device. First of all, a plotting function is needed

```
> myPlot <<- function(x, y, group, labels) {
+     plot(x, y, col = group, pch = 19)
+ }
```

note that the order argument only gives an order while the bullet is traversing a 3d rotation (no order at a node).
then, a visualization instruction has to be defined using `ng_2d_myplot`

```
> viz1 <- ng_2d_myplot(ng.iris, ng.lg, fnName = "myPlot", device = "base")
> viz1
```

```
2D Axis Plot: nd_2d_myplot()
Plot function: myPlot
Graph: 3D Transition
Data: iris
```

and finally, you can call `navGraph` as used

```
> nav <- navGraph(ng.iris, ng.lg, viz1)
```

Note, that by default the base graphics system stores all the plotting instructions onto a device in order to be able to redraw the device if it gets moved or scaled etc... In context with `navGraph` were we are to plot and re-plot the data many times, this behavior is a nuisance and slows down your computer. Hence you can turn this behavior of the current display of (once it exists) with the following command

```
> dev.control(displaylist = "inhibit")
```

Note that OSX has a weird implementation of double buffering. You need to let the bullet rest for a while before the display gets refreshed. Hence animations wont work.

Further, see the demos: `ng_2d_myplot_base` and `ng_2d_myplot_base`.

4.3.2 grid

See demo: `ng_2d_myplot_grid`.

4.3.3 rgl

See demo: `ng_2d_myplot_grid`.

5 Starting a navGraph Session

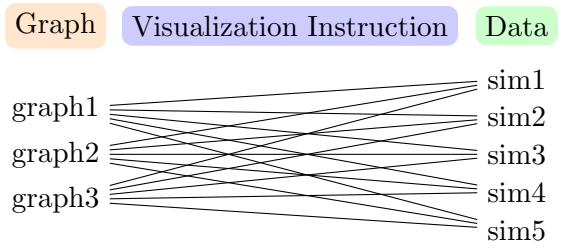
The `navGraph` function is the heart of the `RnavGraph` package. Once the `NG_data`, `NG_graph` and `NG_visualization` objects have been correctly defined, one can pass them to the `navGraph` function which in turn starts the graphical user interface consisting of the graph with a bullet and the visualizations.

We will first discuss the different scenarios the `navGraph` function accommodates. Then we move on to the graphical user interface description. And finally we show how one can communicate with the graphical user interface from the R prompt via the navGraph handler.

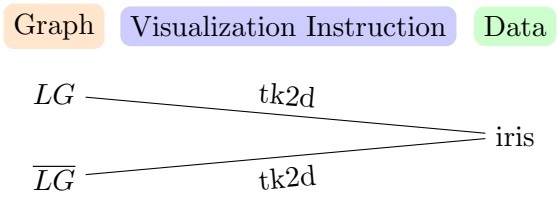
5.1 Calling navGraph

The `navGraph` R function has four arguments: `data`, `graph`, `viz` and `settings`, where the latter three are optional. Except the `settings` argument, all of them accommodate either their corresponding `NG_` object or a list of their corresponding `NG_` objects. A special case poses the `data` argument which also accepts a navGraph handler in order to continue a saved session. More about that later. There are no constraints about how graphs and data sets are connected via visualization instructions as long they are consistent. For example, say we run 5 simulations and we would like

to compare their results via some visualization with `navGraph`. Further there might be three graphs we are interested in traversing to explore the difference in the simulation results. This scenario would look like



where each connecting line represents an `NG_visualization` instruction object. One could also only choose a subset of these visualization instructions. Once the graphical user interface is started, the user can then choose the graph he wants to explore and `navGraph` switches to the visualizations connected to this graph. Hence, for a single `navGraph` the user "looks" form a single graph perspective at all the data connected via their visualization instructions. For the iris example we have the following scenario



If the `navGraph` only gets a `NG_data`- or a list of `NG_data` objects, it creates the graphs and visualization instructions for a 3d and 4d transition graphs connected to the data with the `tk2d` display.

5.1.1 Settings

The `settings` argument of the `navGraph` function determines the look and feel of the graph interaction interface. Currently on can not control the look and feel of the `tk2d` display. The `settings` argument has to be a list optionally containing other named lists. The grand scheme of what can be modified looks like

color (char)	**interaction** (num)	**display**	**tk2d** (num)
background	bulletRadius	NSteps	bg (char)
bullet	nodeRadius	animationTime	brush_colors (vect char)
bulletActive	lineWidth	dragSelectRadius	brush_color (char)
nodes	highlightedLineWidth	labelDistRadius	linked (logical)
nodesActive			
adjNodes			
adjNodesActive			
notVisitedEdge			
visitedEdge			
edgeActive			
labels			
labelsActive			
adjLabels			
adjLabelsActive			
path			

where the `color` elements must be character string (recognized as a color) and the `interaction` and `display` elements must be numeric. For example the following settings object would work

```
> navGraph(..., settings = list(color = list(background = "green"),
+     interaction = list(bulletRadius = 4, nodeRadius = 3)))
```

Note that everything is case sensitive according to the table.

Additionally, there is an additional option that can be specified in settings. If only a `NG_data` object gets passed to `navGraph` one can choose between the `tk2d` and `ggobi` as a display.

```
> navGraph(ng.iris, settings = list(defaultDisplay = "ggobi"))
```

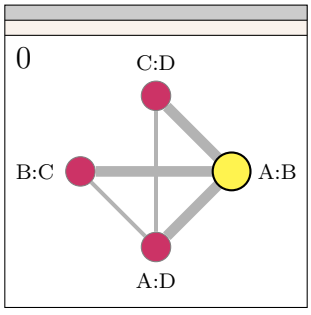
or

```
> navGraph(ng.iris, settings = list(defaultDisplay = "tk2d"))
```

default is the `tk2d` display.

5.2 Graphical User Interface

Once `navGraph` has initialized the all displays, you will see one window with a graph similar to

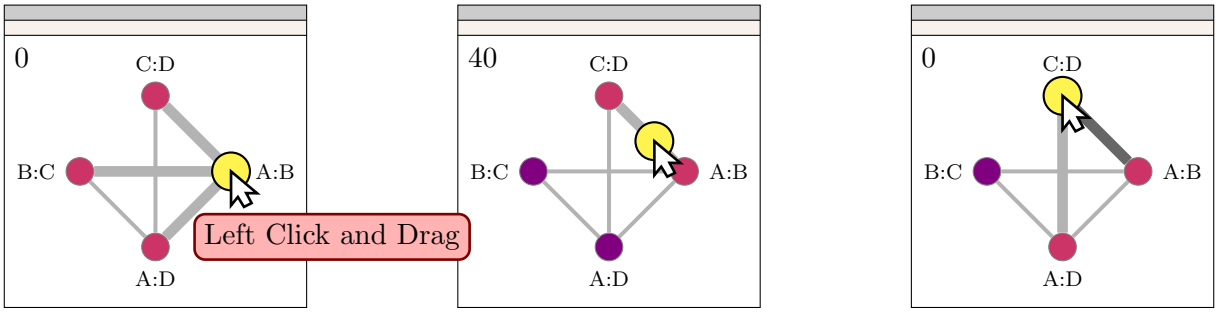


and all the data visualization displays connected to the current graph. We will first cover how to interact with the graph display and then move on to the interaction with the `tk2d` display.

Further, in the sections that follow, we will show some stripes of graphics about the interactions. We refer to them via state1, state2, etc... (reset the numbers every time you see new stripe of graphics).

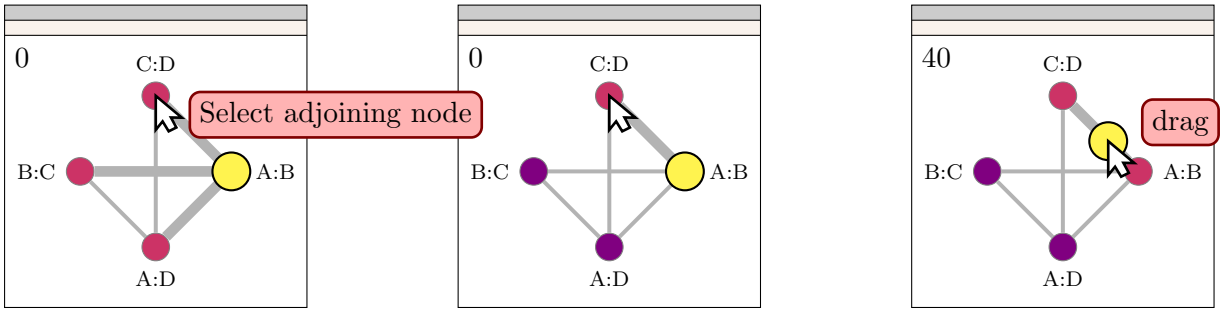
5.2.1 Move the bullet

The bullet can be dragged in a intuitive way along the edges of the graph.



1. Bullet on A:B
    - All adjoining nodes and their connecting edges are highlighted via line with or color.
    - The number in the upper left corner indicates the percentage the bullet progressed towards another node
2. Bullet between A:B and C:D
    - now only the active edge and nodes are highlighted
3. Bullet arrived on C:D
    - Same as in 1) but the edge A:B to C:D has a different color to indicate that we have traversed along this edge.

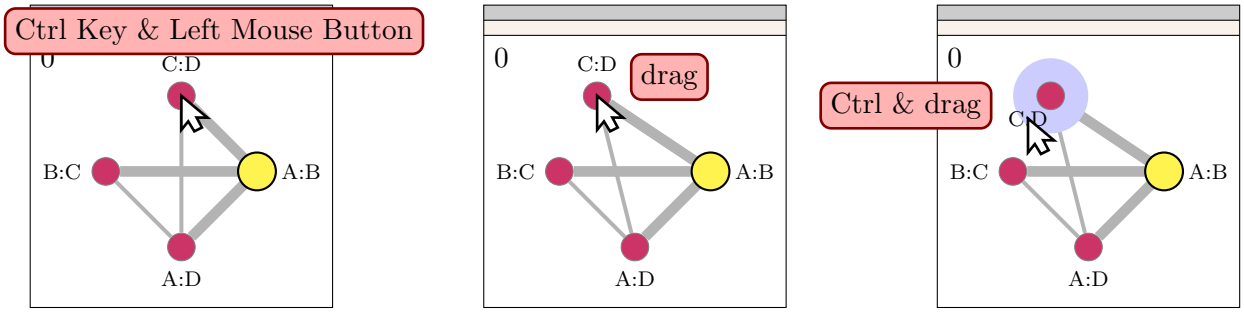
When you drag the bullet in State 1 towards a node, `navGraph` won't constrain the bullets move for some small radius around the current node position. This way, as soon as you cross the prespecified radius, `navGraph` calculates which edge is closest to the direction you moved the bullet and constrains the bullet to move along this edge. If your graph is so large that it is very hard for you to land on the desired edge, you can also select an adjoining node while in state 1. If the node you select is not adjoining, the bullet jumps to this non adjoining node.



Note that once you move along an edge, you can also use your scroll wheel on the mouse to move the bullet. This is conceptually correct, however `navGraph` also has an "active" (mouse over) node and edge state with a different color. (not shown in the images above).

5.2.2 Modify the Graph Layout

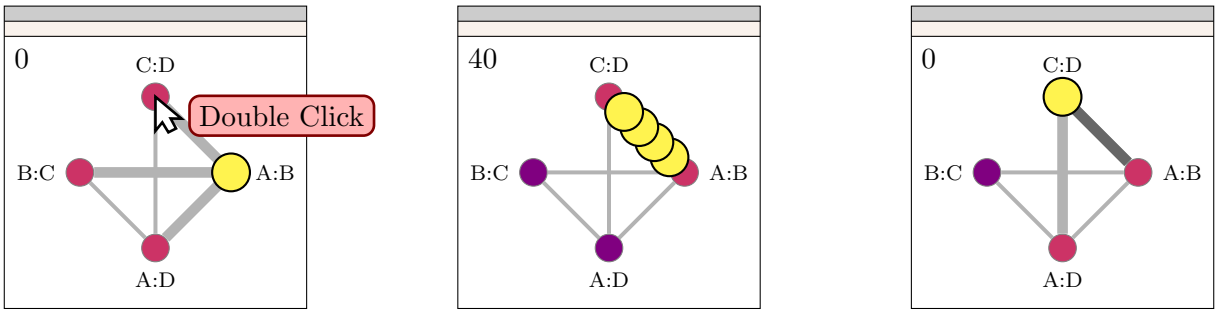
The user can also modify the layout of the graph by dragging nodes on the canvas while pressing the `CTRL` key.



There are no constraints for the nodes to be moved around even the canvas border!. However the labels of the nodes must be moved within the defined `labelDistRadius` in the interaction settings (colored blue here).

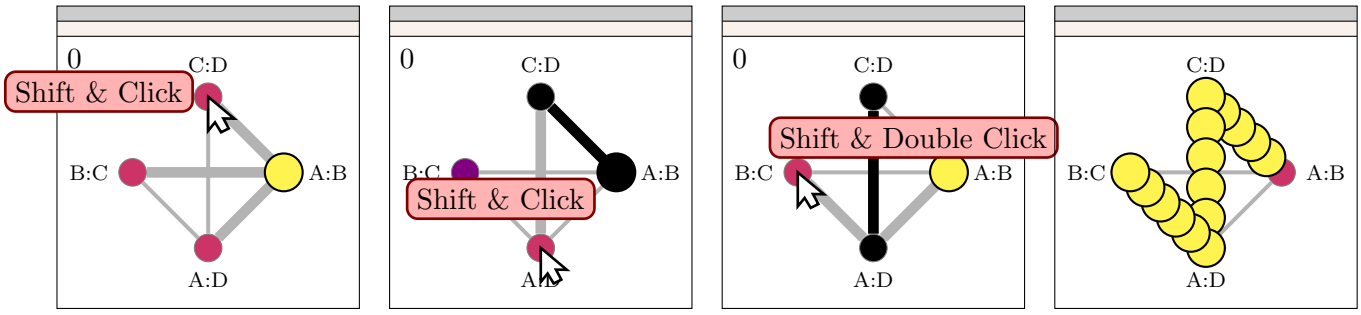
5.2.3 Animate

Once you move the bullet on an edge, you can let the bullet move towards one of the two nodes that define the edge by double clicking on that node. If you click on another node, the bullet jumps there. For the case of the bullet being on one node, you can double click on an adjoining node, the first click will act as edge selection and the second click as an animation command. The animation speed is determined by the `animationTime` variable in the interaction settings. That is, `animationTime/NSteps` is the pause before the bullet moves an increment towards the node. Note however that the time from one node to the other is not equal `animationTime`, since the updating displays and doing other "housekeeping" in `navGraph` also take up time. (More precisely as long as if you select `animationTime`=0).



5.2.4 Paths

Instead of guiding the bullet by dragging it along edges, you can also select a path in order to later animate the path (in the sense of animation above). You create a path by selecting adjoining nodes while having the shift key pressed down. Once you are done, you can double click on the last node and `navGraph` will animate or walk along the just specified path



Note that as soon as you release the shift key, you are out of the path mode. However if you go to the Tools > Paths menu on the graph display window, you will find you path as `Active Path` saved. The Paths toolbox has some bugs. That is, the scroll bar in the Saved Paths list does not work (if you ever happen to add so many paths). This seems a tk issue, and we plan to address this at a later time point. The V button stands for "view path", the W button for "walk path" (animate) and the R button for record paths you explore while dragging the bullet NOT IMPLEMENTED.

You can also animate a path with the `ng_walk` function. The `ng_walk` function takes as an argument a navGraph handler and a path either in the form of a vector of node names or in the form of a single string where the sequence of node names is separated by a space. Lets see an example

```
> nav <- navGraph(ng.iris)
> ng_walk(nav, "s.L:s.W s.L:p.L p.L:p.W s.L:p.W")
```

or the path as a vector

```
> ng_walk(nav, c("s.L:s.W", "s.L:p.L", "p.L:p.W", "s.L:p.W"))
```

Note that every path sent to `ng_walk` gets stored as the active path in the path tool.

Currently, a path is considered to be a sequence of adjoining nodes (no gaps).

5.2.5 The navGraph Handler

The navGraph handler links the graphical user interface with the R prompt. A good example its use was the `ng_walk` function we introduced in the last section. The navGraph handler contains all the data, graphs and settings but not the images of the `navGraph` session in progress. Hence if you work with large data, you want to avoid having a many navGraph handlers in your workspace. The navGraph handler gets returned from a `navGraph` function call

```
> library(RnavGraph)
> ng.iris <- ng_data(name = "iris", data = iris[, 1:4], shortnames = c("s.L",
+     "s.W", "p.L", "p.W"), group = iris$Species, labels = substr(iris$Species,
+     1, 2))
> nav <- navGraph(ng.iris)

Session 1, Data iris is new!

> nav
```

```
RnavGraph handler:
---
created       : Mon Apr 11 13:04:50 2011
last updated  : not
---
graphs        : iris : 3D, iris : 4D
data          : iris
```

You can now brush data and change color and size of objects and add paths. You can save the current state of a `navGraph` session with `ng_update`

```
> nav <- ng_update(nav)
> nav

RnavGraph handler:
---
created       : Mon Apr 11 13:04:50 2011
last updated  : Mon Apr 11 13:04:50 2011
---
graphs        : iris : 3D, iris : 4D
data          : iris
```

Note that the last updated field changed. Updating the navGraph handler is particular useful for getting the new groups (according to color and size of the `tk2d` or `ggobi` display).

```
> ng_get(nav)

possible options are: graphs, paths, data, ggobi, viz
```

the objects graphs,data and viz are lists of their corresponding `NG_object`. However if only one element is in the list, `ng_get` unlists the object

```
> ng_get(nav, "data")

object from NG_data class.
  name: iris
  data: 150 x 4
    Variable Names | Short Names
    ---------------------------
    Sepal.Length   | s.L
    Sepal.Width    | s.W
    Petal.Length   | p.L
    Petal.Width    | p.W
  group:  3 groups.
  labels: se, ve, vi.

> ng_get(ng_get(nav, "data"), "group")[1:5]

[1] "cdarkorchid;s5" "cdarkorchid;s5" "cdarkorchid;s5" "cdarkorchid;s5"
[5] "cdarkorchid;s5"
```

the coding for the `tk2d` display is "c<color>;s<size>". Getting the group for each data point only works with the `tk2d` and `ggobi` data display. For any other custom display the user must provide his own way to get the new group classifiers whenever he incorporates a brushing tool. The navGraph handler can also be used to restart a navGraph session

```
> nav1 <- navGraph(nav)
```

however note that this way the images get lost in the new `tk2d` display. This is because the images are stored in the `tcl` layer, and the images are usually too large to be saved in navGraph handlers.

Again the example with the `ng_walk` function

```
> ng_walk(nav, "s.L:s.W s.L:p.L p.L:p.W s.L:p.W")
> ng_walk(nav, c("s.L:s.W", "s.L:p.L", "p.L:p.W", "s.L:p.W"))
```

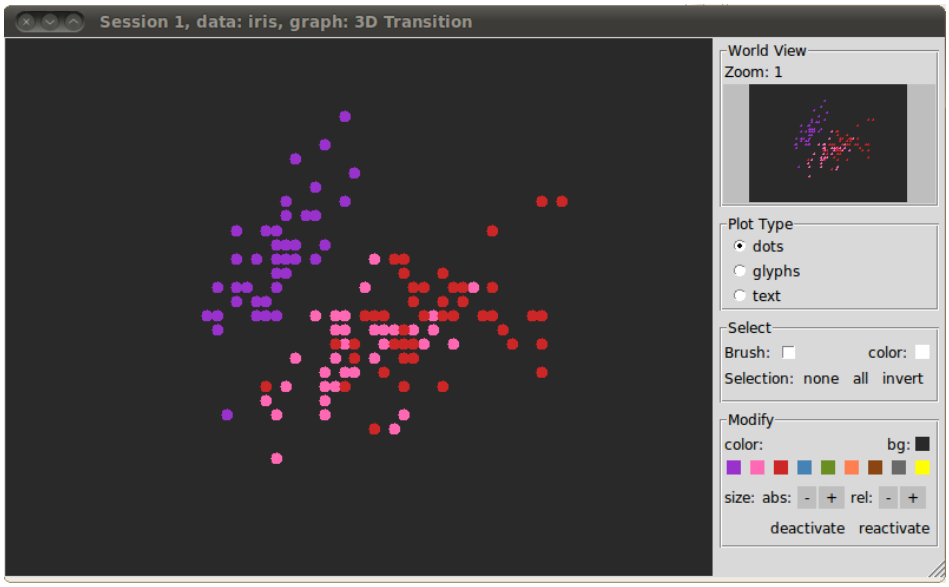
Theoretically, the navGraph handler can be saved using `save` and restored in a new R session using the `load` command. However we recommend to initialize a new `navGraph` session from scratch if possible.

If a `ggobi` and `tk2d` display show the same data, then the `ng_update` function will ask you whether you want to save the grouping of the `tk2d` or `ggobi` display.

5.3 The `tk2d` Display

The `tk2d` display was programmed by us for the `RnavGraph` package. It accommodates displaying dots, images, glyphs and text. It also provides the user with a lot of interactivity possibilities such as brushing, zooming and resizing of the window. Further if for a particular graph two `tk2d` windows display the same data, then the data points are linked between the windows. This means that if you for example change the color and size of some points on one display, the same points in the other window also change color and size.

The `tk2d` window for the iris data looks as follows



The display is designed to be intuitive. So getting familiar with it should not pose any big problems.

5.3.1 Zooming and Moving the viewing region

The `tk2d` displays the data always twice, in the "main view" on the left and in the "world view" in the upper right corner. The "world view" in the upper right corner always displays all the data points and provides as its name suggest a view of "your whole data world". The "main view" allows the user to zoom in and out, brush points and deactivate and reactivate points. The zoom factor of the "main view" is displayed above the map view. The mouse scroll wheel is used to zoom in and out either while the mouse is placed within the "world view" or "main view". If the user zooms in or out in the "main view" the center of zooming is where the mouse cursor points at, where in the "world view" the zooming center is always in the middle of the viewing rectangle. When you zoom in, your current view at the data is shown in the "world view" with a rectangle of the same color as background color of the "main view".

If zoomed in, you can move your current view in the "main view" by pressing the right mouse button and dragging your view to the desired place. In the "world view" you can just drag and drop the viewing rectangle (left mouse button).

5.3.2 Brushing and Selecting

The `tk2d` display allows to change the color, size and hiding (deactivate) of each object. We use size and color since they are (the only) two attributes that are shared between points, glyphs, images and text. Currently the text size however does not change!. You can select points objects only in the "main view" by either selecting and deselecting them with a left mouse click or by using the brush tool. Selecting them with a mouse click deselects the previously with a mouse click selected point. If you want to select points permanently you need to press the Shift key. When you check the brush checkbox, a gray rectangle appears in the upper right corner. You can resize the brush rectangle by dragging the solid rectangle on its right lower corner. You can move the rectangle by clicking anywhere in the "main view" and the brush rectangle jumps to your mouse cursor and also moves with it while your mouse button is still pressed. By default, the points below the brush rectangle don't get brushed permanently. If you want to brush permanently you need to press the Shift key while brushing. Points can only be deselected individually by clicking on them with the shift key pressed, or all together by pressing the non button.

5.3.3 Deactivating and Reactivating points

Once you have grouped a set of points by assigning them different color and size, you might want to hide them so you can group the rest of the points without the distraction of the already grouped points. You can do this by selecting these points and then press "deactivate" in the modify menu. The button will then stay highlighted to indicate that not all the data is shown. You can deactivate different data points in multiple steps by selecting them and repeatedly pressing the "deactivate" button. Once you press the "reactivate" button all point will appear again.

5.3.4 Changing Color and Size

Once you have some points selected, you can change their size and color. You can change the size of the points immediately pressing the "abs" or "rel" + or -. The `tk2d` gives each object a size attribute. absolute (abs) resizing changes all selected objects to the minimum of the object's sizes plus or minus one size. The relative (rel) resizing increments or decrements all sizes by one. If you were to select five objects with the sizes $\{2, 3, 3, 4, 6\}$ and press the rel - button, your new sizes would

be $\{1, 2, 2, 3, 5\}$ and if you were to press the rel - button again you would get $\{0, 1, 1, 2, 5\}$. Note that you can get negative sizes which theoretically would not show on the "main view" anymore, we chose however to display them with minimal possible display size. This might be confusing at the beginning. If you have a set of points selected with the sizes $\{2, 3, 3, 4, 6\}$ and you were to press the abs + button you would get $\{3, 3, 3, 3, 3\}$ or the abs - button you would get $\{1, 1, 1, 1, 1\}$. Size changes take effect immediately.

Changing the color of objects takes place once you have some points brushed and select a colorbox. However you wont notice the change until you deselect all the points. (Use Selection none, invert or all).

5.3.5 Linking Data between two `tk2d` displays

If you display twice (or more) the same data in two `tk2d` displays that are controlled from within the same or different `navGraph` sessions, modifying one point in a particular `tk2d` display modifies all the points from the same data in all `tk2d` displays. Such an example session could be

```
> V <- shortnames(ng.iris)
> G <- completegraph(V)
> LG <- linegraph(G)
> LGnot <- complement(LG)
> ng.lg <- ng_graph(name = "3D Transition", graph = LG, layout = "circle")
> ng.lgnot <- ng_graph(name = "4D Transition", graph = LGnot, layout = "circle")
> viz1 <- ng_2d(ng.iris, ng.lg, glyphs = V[c(1, 2, 3, 4, 1, 3,
+     2, 4)])
> viz2 <- ng_2d(ng.iris, ng.lg)
> viz <- list(viz1, viz2)
> graphs <- list(ng.lg, ng.lgnot)
> nav <- navGraph(graph = graphs, data = ng.iris, viz = viz)
```

Note how `ng.lg` points twice to the `ng.iris` data.

Alternatively you can also start `navGraph` twice

```
> nav1 <- navGraph(ng.iris)
> nav2 <- navGraph(ng.iris)
```

note how a message pops up in the R prompt saying the iris data has been linked. You can tell `navGraph` that it should not link the data

```
> nav3 <- navGraph(ng.iris, settings = list(tk2d = list(linked = FALSE)))
```

Currently, if you wish you had a complete new state, close R and start it again.

6 Scagnostics and RnavGraph

Scagnostics allows one to find graphs that have certain properties, such as the nodes representing clumpy, stringy or convex scatterplots. Scagnostics is.... We will demonstrate how you can use scagnostics to the define the variable graphs. The example data in this section is the olive data, provided in the `RnavGraph` package.

```
> library(scagnostics)
> data(olive)
> ng.olive <- ng_data(name = "Olive", data = olive[, -c(1, 2)],
+     shortnames = c("p1", "p2", "s", "ol", "l1", "l2", "a", "e"),
+     group = as.numeric(olive$Area) + 1)
```

6.1 The quick way

The `scagNav` R function initializes a `navGraph` session, with 3d and 4d transition graphs that have nodes which satisfy the desired scagnostic property defined with the `scag` argument. When calling the `scagnostics` function in the `scagnostics` R package, `scagnostics` returns a matrix with all the possible satterplot combinations in the column and all the scagnostic measures in the rows

```
> scagMat <- scagnostics(olive)
> rownames(scagMat)
```

```
[1] "Outlying"  "Skewed"    "Clumpy"    "Sparse"    "Striated"  "Convex"
[7] "Skinny"    "Stringy"   "Monotonic"
```

The names of these scagnostic measures in combination with a preceding "Not" can be used for the `scag` argument. If you, for example, look for `NotClumpy`, a variable $1 - Clumpy$ gets generated. Further, you can choose a top fraction of scatterplots showing a certain measure most to be displayed in the graph using the `topFrac` argument. `scagNav` will generate for each `scag` element a 3d and 4d transition graph, except if the `combineFn` arguments gets specified. The `combineFn` argument takes any function defined on a vector of scagnostic measure weights such as sum or max. This is for example useful, if you would like to create a graph that either displays clumpy or stringy patterns in its 2d scatterplot nodes.

```
> nav <- scagNav(data = ng.olive, scags = c("Skinny", "Sparse",
+     "NotConvex"), topFrac = 0.2, combineFn = max, glyphs = shortnames(ng.olive)[1:8],
+     sep = ":")
```

or

```
> nav <- scagNav(data = ng.olive, scags = c("Skinny", "Sparse",
+     "NotConvex"), topFrac = 0.2, glyphs = shortnames(ng.olive)[1:8],
+     sep = "+")
```

Careful, `scags` is case sensitive.

6.2 The detailed way

The `scagNav` function generates 3d and 4d transition graphs with their corresponding `tk2d` visualization. We now show how to do each step so that the user can work with the graphs and visualization instructions and finally call the `navGraph` function. For every `navGraph` session needs to know the data, graphs and the visualization instruction (this should be your mantra for the next few days). We have already defined the `ng.olive` data object, we now want to create a variable graph or a set of variable graphs who's edges carry weights from the `scagnostics` function. Hence we first have to extract the weights of interest. This is done using our `scagEdgeWeights` function

```
> edgeWts <- scagEdgeWeights(data = ng.olive, scags = c("Clumpy",
+     "Skinny"))
> edgeWts$fromToEdgeMatrix[1:3, ]

     from to     Clumpy    Skinny
[1,]    1  2 0.01425229 0.5928461
[2,]    1  3 0.01101357 0.5185595
[3,]    1  4 0.02674608 0.5791038

> edgeWts$nodeNames

[1] "p1" "p2" "s"  "ol" "l1" "l2" "a"  "e"
```

the numbers in the `from` and `to` column correspond with the order of the `nodeNames`. Note that the `data` argument can either be a `NG_data` object or a `data.frame`. There is also a `combineFn` argument which takes a function:

```
> edgeWts <- scagEdgeWeights(data = ng.olive, scags = c("Clumpy",
+     "Skinny"), combineFn = max)
> edgeWts$fromToEdgeMatrix[1:3, ]

     from to combined weights
[1,]    1  2        0.5928461
[2,]    1  3        0.5185595
[3,]    1  4        0.5791038
```

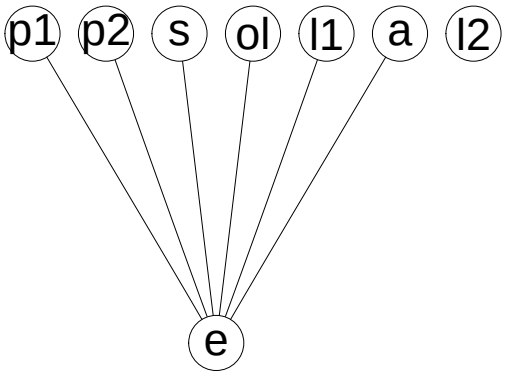
or

```
> edgeWts <- scagEdgeWeights(data = ng.olive, scags = c("Clumpy",
+     "Skinny"), combineFn = function(x) {
+     2 * x[1] + 3 * x[2]
+ })
> edgeWts$fromToEdgeMatrix[1:3, ]

     from to combined weights
[1,]    1  2         1.807043
[2,]    1  3         1.577706
[3,]    1  4         1.790803
```

Currently, the `combineFn` only takes a function which returns a single value.

We now could manually create a variable graph using `newgraph` where we only draw an edge if the `combined weight` of an edge lies within the top 20% quantile

```
> weights <- edgeWts$fromToEdgeMatrix[, "combined weights"]
> ii <- weights > quantile(weights, 0.8)
> G <- newgraph(nodeNames = edgeWts$nodeNames, mat = edgeWts$fromToEdgeMatrix[ii,
+     c(1, 2)], weights = weights[ii])
> plot(G)
```



where the weights are part of the graph `G`, see for example

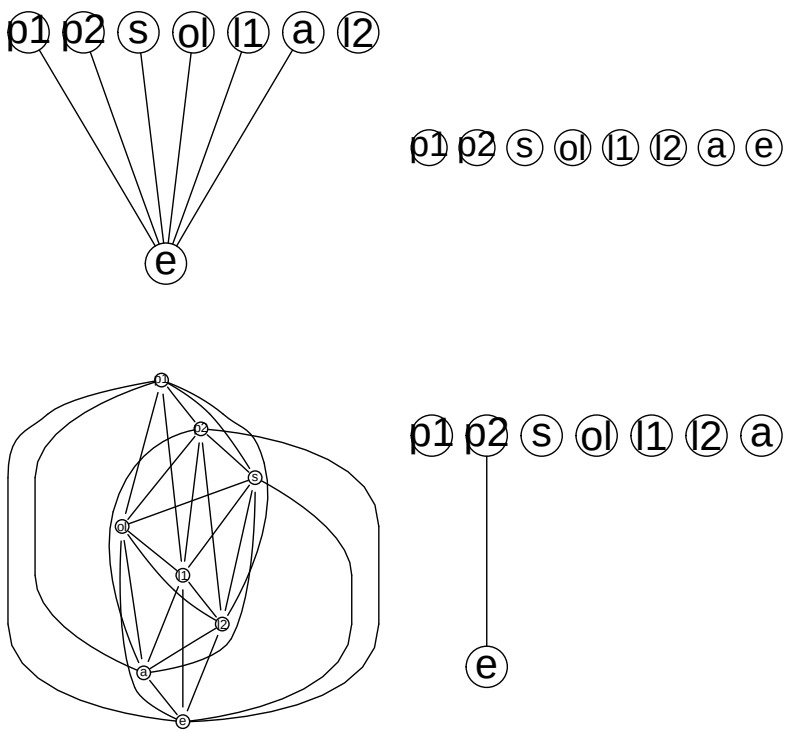
```
> edgeData(G, attr = "weight")$"p1|e"

[1] 2.968342
```

and hence we could run `navGraph`

```
> ng.lg <- ng_graph("3d olive", linegraph(G))
> viz <- ng_2d(ng.olive, ng.lg)

> nav <- navGraph(ng.olive, ng.lg, viz)
```

Now this becomes tedious if you have multiple scagnostic measures and hence you would have to create graph for each measure. The `scagGraph` function simplifies this task. Note how `scagGraph` creates `graph` object or a list of `graph` objects with one single call

```
> par(mfrow = c(2, 2))
> G_1 <- scagGraph(edgeWts, topFrac = 0.2)
> plot(G_1)
> edgeData(G_1, attr = "weight")$"p1|e"

[1] 2.968342

> G_1 <- scagGraph(edgeWts, topFrac = 0)
> plot(G_1)
> G_1 <- scagGraph(edgeWts, topFrac = 1)
> plot(G_1)
> G_1 <- scagGraph(edgeWts, topFrac = 1e-04)
> plot(G_1)
```



And for multiple scagnostic measures

```
> edgeWts <- scagEdgeWeights(data = ng.olive, scags = c("Clumpy",
+     "NotClumpy", "Monotonic"), combineFn = NULL)
> graphList <- scagGraph(edgeWts, topFrac = 0.2)
> graphList

$Clumpy
A graphNEL graph with undirected edges
Number of Nodes = 8
Number of Edges = 6

$NotClumpy
A graphNEL graph with undirected edges
Number of Nodes = 8
Number of Edges = 6

$Monotonic
A graphNEL graph with undirected edges
Number of Nodes = 8
Number of Edges = 6
```

Now start navGraph

```
> nav <- navGraph(ng.olive, graphList)
```

7 Example sessions on different data

In this last section, we will give example code for navGraph sessions using different well known data sets. Some of them are provided by the MASS library

```
> library(MASS)
```

7.1 Iris

```
> ng.iris <- ng_data(name = "iris", data = iris[, 1:4], shortnames = c("s.L",
+     "s.W", "p.L", "p.W"), group = as.numeric(iris$Species), labels = substr(iris$Specie
+     1, 2))
> nav <- navGraph(ng.iris)
```

7.2 Olive

```
> library(PairViz)
> data(olive)
> d.olive <- data.frame(olive[, -c(1, 2)])
> ng.olive <- ng_data(name = "Olive", data = d.olive, shortnames = c("p1",
+     "p2", "s", "oleic", "l1", "l2", "a", "e"), group = as.numeric(olive[,
+     "Area"]), labels = as.character(olive[, "Area"]))
> ng.olive
> G <- completegraph(shortnames(ng.olive))
> LG <- linegraph(G)
> ng.lg <- ng_graph("3d olive", LG, layout = "kamadaKawaiSpring")
> ng.lgnot <- ng_graph("4d olive", complement(LG), layout = "kamadaKawaiSpring")
> nav <- navGraph(ng.olive, list(ng.lg, ng.lgnot), list(ng_2d(ng.olive,
+     ng.lg), ng_2d(ng.olive, ng.lgnot)))
> ng_walk(nav, eulerian(as(LG, "graphNEL"))[1:7])
```

7.3 US Judge Ratings

```
> library(MASS)
> ng.data <- ng_data(name = "US Judge Ratings", data = USJudgeRatings)
> p <- ncol(USJudgeRatings)
> adjM <- matrix(0, ncol = p, nrow = p)
> adjM[c(1:8, 11), c(9, 10, 12)] <- 1
> adjM[c(9, 10, 12), c(1:8, 11)] <- 1
> G <- newgraph(names(ng.data), adjM, isAdjacency = TRUE)
> ng.lg <- ng_graph("3d Us Judge", linegraph(G), layout = "fruchtermanReingold")
> nav <- navGraph(ng.data, ng.lg, ng_2d(ng.data, ng.lg))
```

7.4 Storm Tracks

```
> library(rggobi)
> names(stormtracks)
> storms <- stormtracks[, c(2:9, 11)]
```

```
> ng.storms <- ng_data(name = "Storm tracks", data = stormtracks[,
+     c(2:9, 11)], group = as.numeric(stormtracks[, "type"]), labels = stormtracks[,
+     "type"])
> p <- ncol(ng_get(ng.storms, "data"))
> adjM <- matrix(0, ncol = p, nrow = p)
> adjM[c(1, 2, 4, 5, 6), c(7, 8, 9)] <- 1
> adjM[c(7, 8, 9), c(1, 2, 4, 5, 6)] <- 1
> adjM[c(7, 8, 9), c(7, 8, 9)] <- 1
> adjM[7, 7] <- adjM[8, 8] <- adjM[9, 9] <- 0
> adjM[c(5, 6), c(5, 6)] <- 1
> adjM[5, 5] <- adjM[6, 6] <- 0
> G <- newgraph(names(ng.storms), adjM, isAdjacency = TRUE)
> LG <- linegraph(G)
> ng.lg <- ng_graph("3d storm tracks", LG, layout = "kamadaKawaiSpring")
> ng.lgnot <- ng_graph("4d storm tracks", complement(LG), layout = "kamadaKawaiSpring")
> viz1 <- ng_2d(ng.storms, ng.lg)
> viz2 <- ng_2d(ng.storms, ng.lgnot)
> nav <- navGraph(ng.storms, list(ng.lg, ng.lgnot), list(viz1,
+     viz2))
```

7.5 US cereal

```
> ng.data <- ng_data(name = "UScereal", data = UScereal[, c(2:8,
+     10)], shortnames = c("cal", "prot", "fat", "sod", "fib",
+     "carb", "sug", "pt"), group = UScereal[, 1], labels = UScereal[,
+     1])
> nav <- navGraph(ng.data)
> nav <- scagNav(ng.data, scags = "Outlying", topFrac = 0.2)
```

7.6 Boston Housing

```
> ng.data <- ng_data(name = "Boston", data = Boston[, -9], shortnames = names(Boston[,
+     -9]), group = Boston[, "rad"])
> nav <- navGraph(ng.data)
> nav <- scagNav(ng.data, scags = "Clumpy", topFrac = 0.2)
```

7.7 Birth Weight

```
> ng.data <- ng_data(name = "Birth Weight Data", data = birthwt[,
+     c(1:3, 5:10)], group = birthwt[, 4])
> p <- ncol(ng_get(ng.data, "data"))
> adjM <- matrix(0, ncol = p, nrow = p)
> adjM[c(1:8), c(9)] <- 1
> adjM[c(9), c(1:8)] <- 1
> G <- newgraph(names(ng.data), adjM, isAdjacency = TRUE)
> LG <- linegraph(G, sep = "++")
> ng.lg <- ng_graph("3d birth weight", LG, sep = "++")
```

```
> ng.lgnot <- ng_graph("34 birth weight", complement(LG), sep = "++")
> nav <- navGraph(ng.data, list(ng.lg, ng.lgnot), list(ng_2d(ng.data,
+     ng.lg), ng_2d(ng.data, ng.lgnot)))
```

7.8 Swiss bank note data

```
> require(alr3)
> data(banknote)
> names(banknote[, 1:6])
> ng.data <- ng_data(name = "Swiss bank note Data", data = banknote[,
+     1:6], shortnames = names(banknote[, 1:6]), group = banknote[,
+     7])
> nav <- navGraph(ng.data)
> nav <- scagNav(ng.data, scags = "Clumpy", topFrac = 0.2)
```

7.9 Body Dimensions Data

```
> require(gclus)
> data(body)
> names(body[, 1:10])
> ng.data <- ng_data(name = "Body Dimensions", data = body[, 1:24],
+     group = body[, 25])
> nav <- navGraph(ng.data)
> nav <- scagNav(ng.data, scags = "Clumpy", topFrac = 0.1)
```

7.10 Ozone Data

```
> require(gclus)
> data(ozone)
> ng.data <- ng_data(name = "Ozone data", data = ozone)
> nav <- navGraph(ng.data)
> p <- ncol(ng_get(ng.data, "data"))
> adjM <- matrix(0, ncol = p, nrow = p)
> adjM[c(1), c(2, 4, 5, 6)] <- 1
> adjM[c(2, 4, 5, 6), c(1)] <- 1
> G <- newgraph(names(ng.data), adjM, isAdjacency = TRUE)
> LG <- linegraph(G)
> ng.lg <- ng_graph("3d ozone", LG, layout = "circle")
> ng.lgnot <- ng_graph("4d ozone", complement(LG), layout = "circle")
> nav <- navGraph(ng.data, list(ng.lg, ng.lgnot), list(ng_2d(ng.data,
+     ng.lg), ng_2d(ng.data, ng.lgnot)))
```

7.11 Swiss fertility

```
> ng.data <- ng_data(name = "SwissFertility", data = swiss, shortnames = c("Fer",
+     "Agri", "Exam", "Edu", "Cath", "IM"))
> p <- ncol(swiss)
```

```
> adjM <- matrix(0, ncol = p, nrow = p)
> adjM[1:5, 6] <- 1
> adjM[6, 1:5] <- 1
> G <- newgraph(shortnames(ng.data), adjM, isAdjacency = TRUE)
> LG <- linegraph(G)
> ng.lg <- ng_graph("3d fertility", LG, layout = "circle")
> ng.lgnot <- ng_graph("4d fertility", complement(LG), layout = "fruchtermanReingold")
> nav <- navGraph(ng.data, list(ng.lg, ng.lgnot), list(ng_2d(ng.data,
+     ng.lg), ng_2d(ng.data, ng.lgnot)))
```

7.12 Challenger

```
> require(alr3)
> data(challeng)
> ng.data <- ng_data(name = "Challenger Data", data = challeng[,
+     c(1:3, 5:7)])
> p <- ncol(ng_get(ng.data, "data"))
> adjM <- matrix(0, ncol = p, nrow = p)
> adjM[c(1, 2), c(3:6)] <- 1
> adjM[c(3:6), c(1, 2)] <- 1
> G <- newgraph(names(ng.data), adjM, isAdjacency = TRUE)
> LG <- linegraph(G)
> ng.lg <- ng_graph("3d challenger", LG)
> ng.lgnot <- ng_graph("4d challenger", complement(LG))
> nav <- navGraph(ng.data, list(ng.lg, ng.lgnot), list(ng_2d(ng.data,
+     ng.lg), ng_2d(ng.data, ng.lgnot)))
```

7.13 Animal

```
> library(PairViz)
> require(cluster)
> data(animals)
> names(animals)
> ng.data <- ng_data(name = "Animal Data", data = animals)
> nav <- navGraph(ng.data)
> ng_walk(nav, eulerian(as(ng_get(ng_get(nav, "graphs")[[1]], "graph")),
+     "graphNEL"))
```

References

- Hurley, C. and R.W. Oldford (2011), "Graphs as navigational infrastructure for high dimensional data spaces", (*Computational Statistics*, to appear).