Magolego SNA - Lab 6

Contents

Motifs

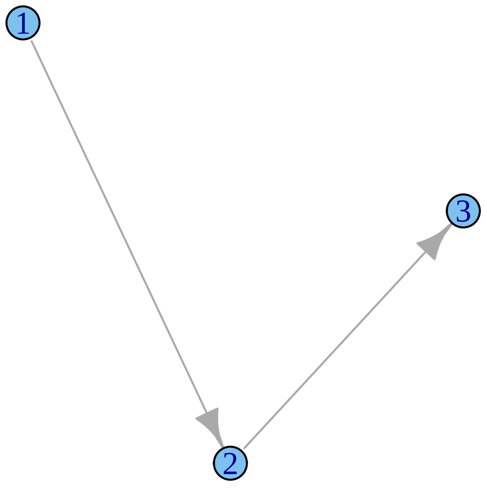
Motifs are often defined as recurrent and statistically significant sub-graphs or patterns. Here, we consider motifs as sub-graphs of a given graph, which are isomorphic to defined sample.

Here is our old beloved friend:

```r
library('igraph')
g = graph.famous("Zachary")
```

Let's define a simple sample motif:

```r
sample = graph(c(c(1,2), c(2,3)))
plot(sample)
```



Now let's calculate, if it is often in our graph:

```
isoclass_num = graph.isoclass(sample) # defining number of corresponding isoclass
motifs3 = graph.motifs(g, size = 3)
motifs3[isoclass_num] # returns number of motifs
```

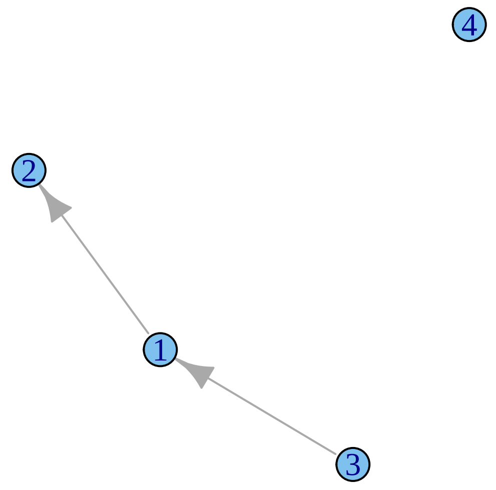
```
## [1] 45
```

Due to high computational time (isomorphism checks), `graph.motifs` are implemented for graps of sizes 3 and 4 only. However, we can easy check numbers for all motifs of size 4 to find more frequent patterns:

```
motifs4 = graph.motifs(g, size = 4)
```

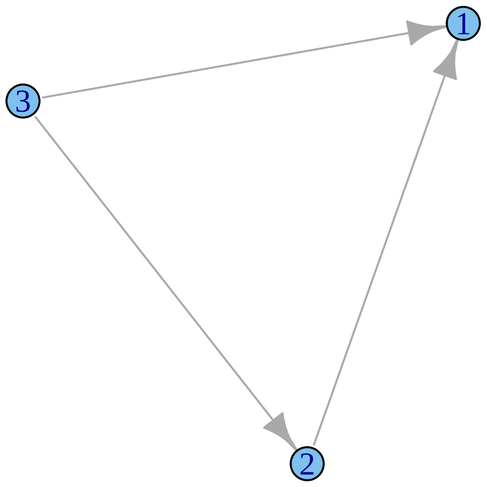
The most frequent pattern is fifth. Let's draw it.

```
plot(graph.isocreate(size=4, number=5))
```



Not what we're looking for.. Second most frequent (seventh):

```
plot(graph.isocreate(size=3, number=7))
```



That is certainly better.

Visualisation

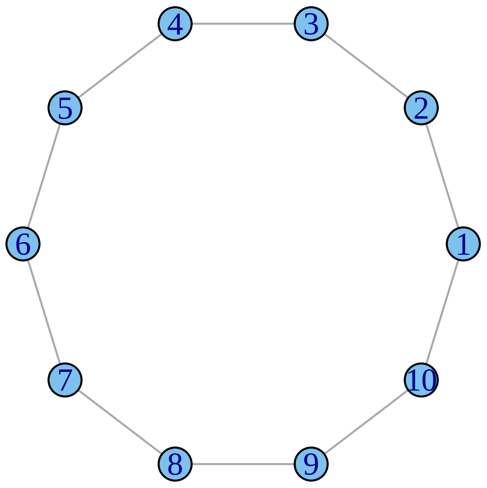
There are currently three different functions in the igraph package which can draw graph in various ways:

- `plot.igraph` does simple non-interactive 2D plotting to R devices. Actually it is an implementation of the plot generic function, so you can write plot(graph) instead of plot.igraph(graph). As it used the standard R devices it supports every output format for which R has an output device. The list is quite impressing: PostScript, PDF files, XFig files, SVG files, JPG, PNG and of course you can plot to the screen as well using the default devices, or the good-looking anti-aliased Cairo device. See plot.igraph for some more information.
- `tkplot` does interactive 2D plotting using the tcltk package. It can only handle graphs of moderate size, a thousend vertices is probably already too many. Some parameters of the plotted graph can be changed interactively after issuing the tkplot command: the position, color and size of the vertices and the color and width of the edges. See tkplot for details.
- `rglplot` is an experimental function to draw graphs in 3D using OpenGL. See rglplot for some more information.

Let's draw a graph-ring using different three methods.

```r
library(igraph)
library(rgl)
```

```
g <- graph.ring(10)
g$layout <- layout.circle
plot(g)
```



```
#doesn't work in R Markdown
tkplot(g)
```

```
## Loading required package: tcltk
```

```
## [1] 1
```

```
rglplot(g)
```

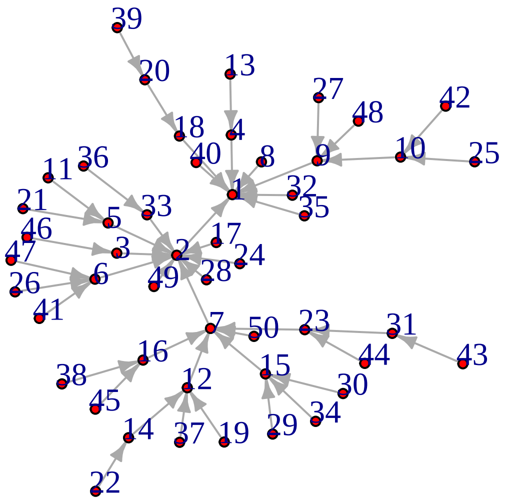
Layout

Either a function or a numeric matrix. It specifies how the vertices will be placed on the plot.
Let's demonstrate how it works on some graph. For example graph which based on barabashi model.

```
g <- barabasi.game(50)
```

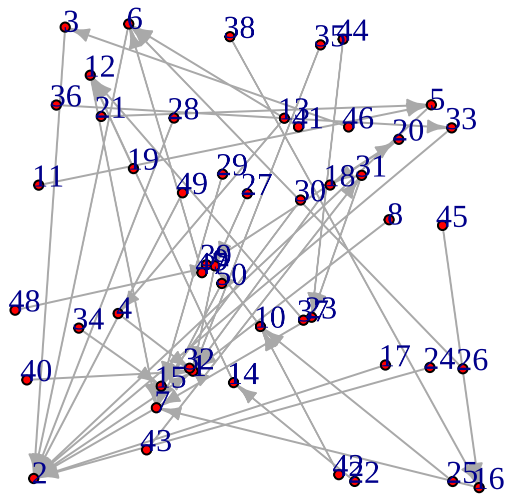
- `layout.auto` - tries to choose an appropriate layout function for the supplied graph, and uses that to generate the layout.

```r
plot(g, layout=layout.auto, vertex.size=4,
     vertex.label.dist=0.5, vertex.color="red", edge.arrow.size=0.5)
```



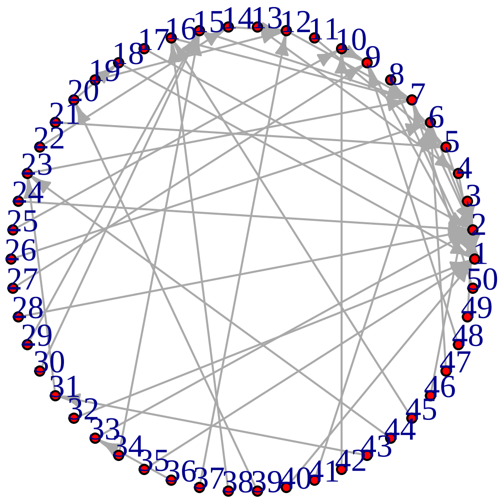
- `layout.random` - simply places the vertices randomly on a square.

```r
plot(g, layout=layout.random, vertex.size=4,
     vertex.label.dist=0.5, vertex.color="red", edge.arrow.size=0.5)
```



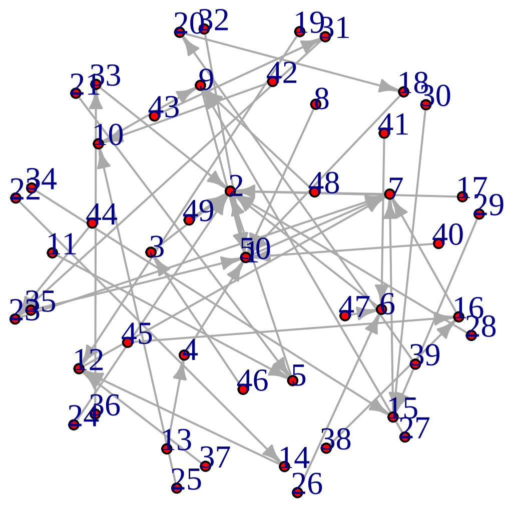
- `layout.circle` - places the vertices on a unit circle equidistantly.

```
plot(g, layout=layout.circle, vertex.size=4,
     vertex.label.dist=0.5, vertex.color="red", edge.arrow.size=0.5)
```



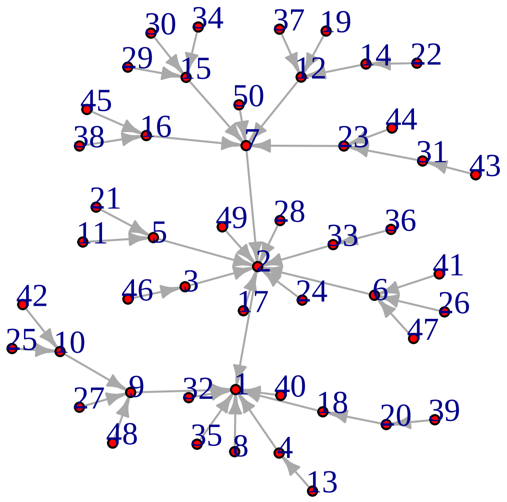
- `layout.sphere` - places the vertices (approximately) uniformly on the surface of a sphere, this is thus a 3d layout.

```
plot(g, layout=layout.sphere, vertex.size=4,
     vertex.label.dist=0.5, vertex.color="red", edge.arrow.size=0.5)
```



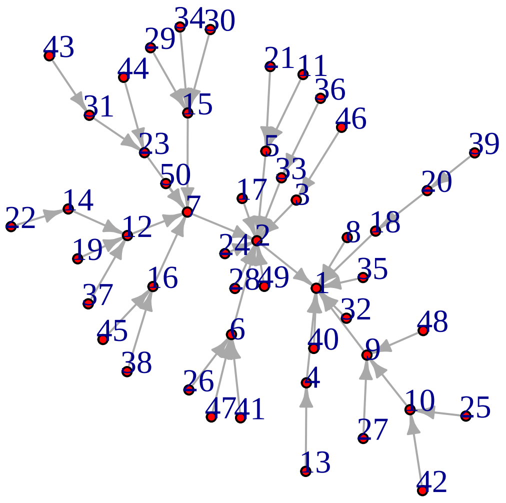
- `layout.fruchterman.reingold` uses a force-based algorithm proposed by Fruchterman and Reingold.

```
plot(g, layout=layout.fruchterman.reingold, vertex.size=4,
     vertex.label.dist=0.5, vertex.color="red", edge.arrow.size=0.5)
```



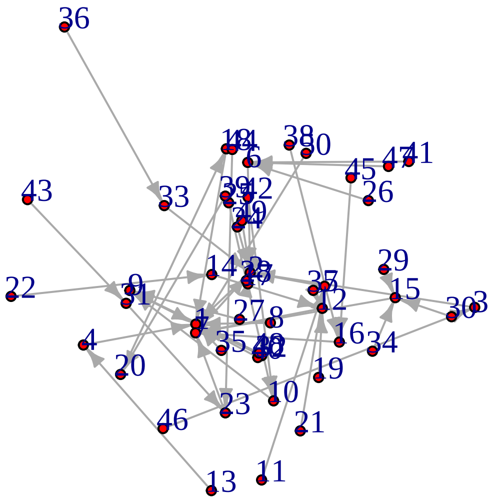
- `layout.kamada.kawai` is another force based algorithm.

```
plot(g, layout=layout.kamada.kawai, vertex.size=4,
     vertex.label.dist=0.5, vertex.color="red", edge.arrow.size=0.5)
```



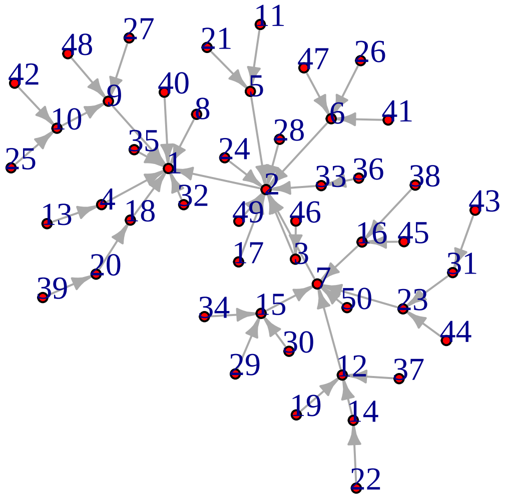
- `layout.spring` is a spring embedder algorithm.

```
plot(g, layout=layout.spring, vertex.size=4,
     vertex.label.dist=0.5, vertex.color="red", edge.arrow.size=0.5)
```



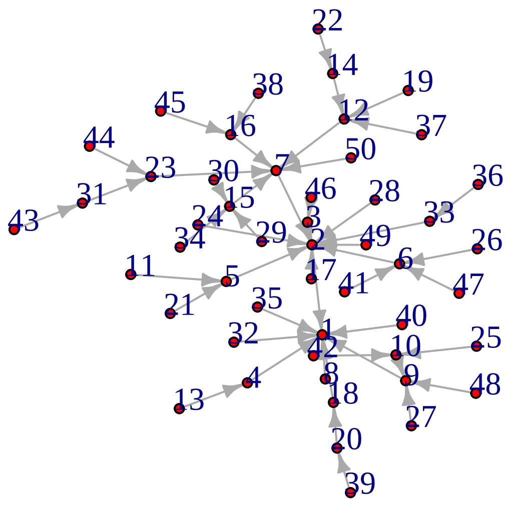
- `layout.fruchterman.reingold.grid` is similar to layout.fruchterman.reingold but repelling force is calculated only between vertices that are closer to each other than a limit, so it is faster.

```
plot(g, layout=layout.fruchterman.reingold.grid, vertex.size=4,
     vertex.label.dist=0.5, vertex.color="red", edge.arrow.size=0.5)
```



- `layout.lgl` is for large connected graphs, it is similar to the layout generator of the Large Graph Layout software http://lgl.sourceforge.net/.

```
plot(g, layout=layout.lgl, vertex.size=4,
     vertex.label.dist=0.5, vertex.color="red", edge.arrow.size=0.5)
```



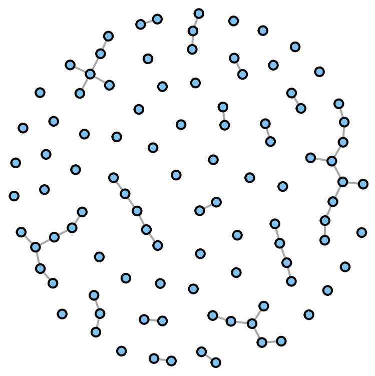
- `layout.graphopt` is a port of the graphopt layout algorithm by Michael Schmuhl. graphopt version 0.4.1 was rewritten in C and the support for layers was removed (might be added later) and a code was a bit reorganized to avoid some unneccessary steps is the node charge (see below) is zero.
- `layout.svd` is a currently experimental layout function based on singular value decomposition.
- `layout.norm` normalizes a layout, it linearly transforms each coordinate separately to fit into the given limits.
- `layout.drl` is another force-driven layout generator, it is suitable for quite large graphs.
- `layout.reingold.tilford` generates a tree-like layout, so it is mainly for tree.

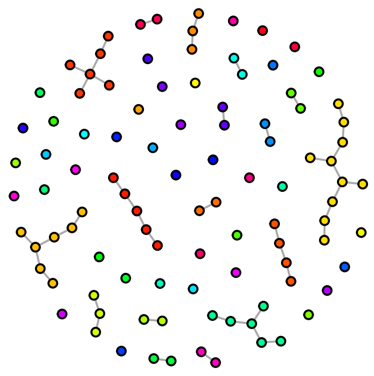
Highlight components

Let's make different color for each graph component

```r
g <- erdos.renyi.game(100, 1/100)

l <- layout.fruchterman.reingold(g)

op = par(mfrow = c(1,2))
plot(g, layout=l, vertex.size=5, vertex.label=NA)

comps <- clusters(g)$membership
colbar <- rainbow(max(comps)+1)
```

```
V(g)$color <- colbar[comps+1]

plot(g, layout=l, vertex.size=5, vertex.label=NA)
```

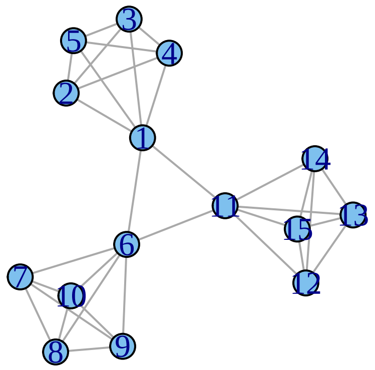


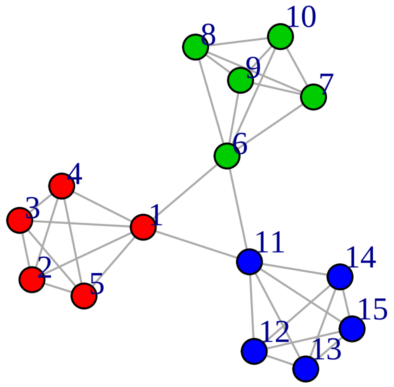


Highlight communities in graph

Let's make different color for each community

```
g <- graph.full(5) %du% graph.full(5) %du% graph.full(5)
g <- add.edges(g, c(1,6, 1,11, 6,11))

op = par(mfrow = c(1,2))
plot(g, layout = layout.kamada.kawai)

com <- spinglass.community(g, spins=5)
V(g)$color <- com$membership+1
g <- set.graph.attribute(g, "layout", layout.kamada.kawai(g))
plot(g, vertex.label.dist=1.5)
```

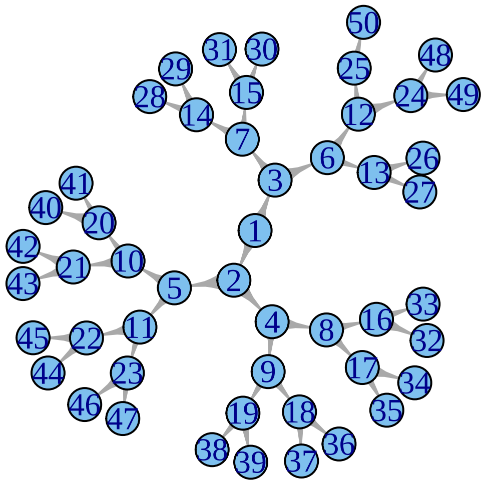




```
par(op)
```

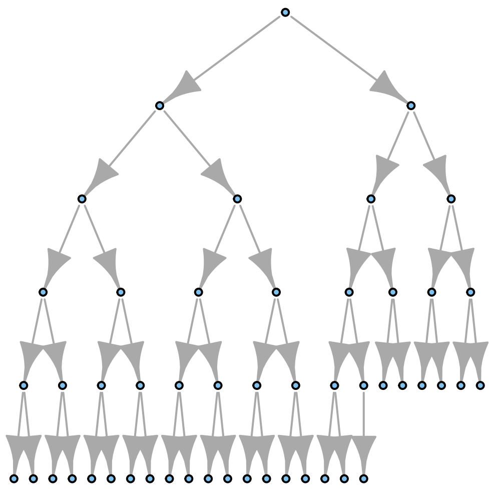
Trees Visualization

```
plot(graph.tree(50, 2))
```



We can use `layout = layout.reingold.tilford` to draw tree

```
plot(graph.tree(50, 2), vertex.size=3, vertex.label=NA, layout=layout.reingold.tilford)
```



```r
tkplot(graph.tree(50, 2, mode="undirected"), vertex.size=10,
       vertex.color="green")
```

```
## [1] 2
```

Assortative Mixing

Assortative Mixing coefficient shows whether nodes with the same attribute values tend to form connections. Download Caltech.gml (or Caltech.mat) friendship network. Inspect nodes attributes and compute assortativity coefficients with `assortativity` function

```r
# Your code here
g <- read.graph(file = 'Caltech.gml', format = 'gml')
assortativity.nominal(graph = g, types = V(g)$dorm+1, directed = F)
```

```
## [1] 0.3491531
```

Basic network analysis pipeline

The basic pipeline for exploratory graph analysis consists of:

Loading

See Seminar 1.

Cleaning

After loading graph with all necessary attributes, it is often recommended to:

- delete empty nodes:

```
g = delete.vertices(g, degree(g) == 0)
```

- delete self-cycles and multiple edges - to make graph simple. `simplify` does all the work; some parameters could be tuned:

```
simplify(g)
```

```
## IGRAPH U--- 769 16656 --
## + attr: id (v/n), status (v/n), gender (v/n), magor (v/n),
##   sndmagor (v/n), dorm (v/n), year (v/n), school (v/n)
```

```
# is.simple(simplify(g, remove.multiple=FALSE))
```

Obtaining main characteristics

See Seminars 2, 4.

Clustering

See Seminar 5.

Visualization

See previous section + Seminars 1, 4.