inetwork package vignette

Sun-Chong Wang

July 1, 2007

1 Introduction

The package implements functions for network analysis and plotting. It includes a function for the detection of communities or modules in the network. The algorithm for community-finding, also known in engineering as network partitioning, is based on a spectral method developed by M.E.J. Newman[1, 2]. Layout and visualization of networks are challenging when the topology of the networks is complex. The network plotting functions in this package purport to display complex networks in a compact and organized fashion by taking advantage of the community structure embedded in the networks.

2 Scheme

2.1 Hierarchical structure

Before plotting, one finds the communities by calling the network partitioning function `icommunity`:

```
> library(inetwork)
> data(cashflow)
> cluster5 <- icommunity(cf5, labelcf5)
```

The network `cf5`, which comes with the package in `cashflow`, has a total of 56 vertices. 14 of them are isolated, meaning they are not connected to any other vertices in the network. To get the communities among the 42 connected vertices, the function `icommunity` was called given the adjacency matrix `cf5`. The second input to `icommunity` is optional, providing the labels of the vertices. To show the detected communities, we use the other function in the package:

```
> ihierarchy(cluster5)
```

The plot shows the five detected communities, whose sizes are 8 in navy, 11 in blue, 6 in red, 1 in black and 16 in green. The vertical depths from the top in the plot indicate improvements in the *modularity*, a measure of clique, with divisions of the (sub)network into (sub)subnetworks. For example, in the first division, the original network, consisting of 42 connected vertices, was divided into two subnetworks: navy+blue on the left and red+black+green on the right in Fig. 1. The magnitude of the gain in modularity resulting from this division is the length of the vertical bar in the upper center. Further gain is achieved

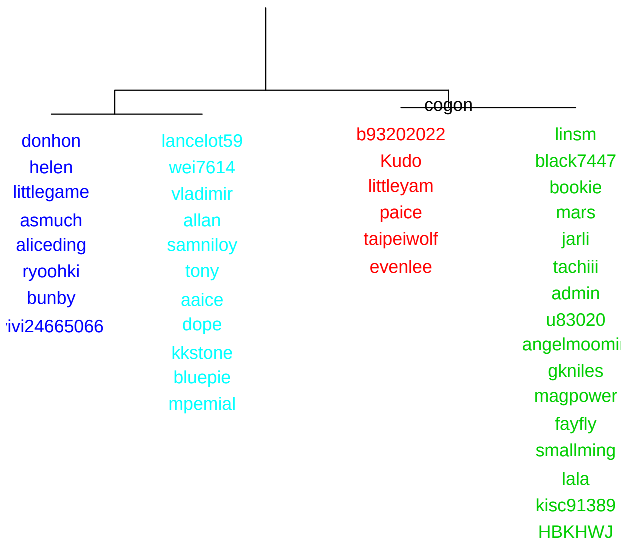

Figure 1: Communities, i.e. modules, are identified and shown in different colors

by dividing the 19 vertices on the left into 8 navy and 11 blue vertices. The magnitude of the gain is measured by the length of the left vertical bar. The algorithm also applied to the 23 vertices on the right. In this case, firstly, the vertex `cogon` was singled out without change in the modularity. The remaining 22 was then divided into 6 red and 16 green with gain in the modularity indicated by the depth of the vertical bar on the right (cf Fig. 1). Further divisions to the resulting subnetworks failed because of no gain in the modularity. The algorithm thus stopped and the results were output.

The order in which the vertices in a community appear in the column conveys information about the importance of the vertices in the community. For example, `donhon` in the navy community can be considered ringleader of that community in that if `donhon` is moved from the navy community to any other communities, the modularity of the network suffers more than if `helen` or any other member in the navy community are moved. Likewise, `helen` assumes a more important role than any others below her. Note however that the ringleadership scores of the members can tie so that, for example, `helen` is as important as `littlegame`, etc.

Sub-(sub)communities within (sub)communities reveal hierarchical structure in the network topology. A nice feature of such a hierarchical display is that if a partition of the network into two parties is enough, we know it is a 19-member left gang versus a 23-member right gang as in Fig. 1. In the case of voting, we know the left gang may try to recruit `cogon`, `evenlee`, `HBKHWJ`, ... etc in an effort to become majority.

As the size of the network grows, laying out vertices under a roof can become clumsy as seen in the following example,

```
> ihierarchy(icommunity(cf9, labelcf9))
```

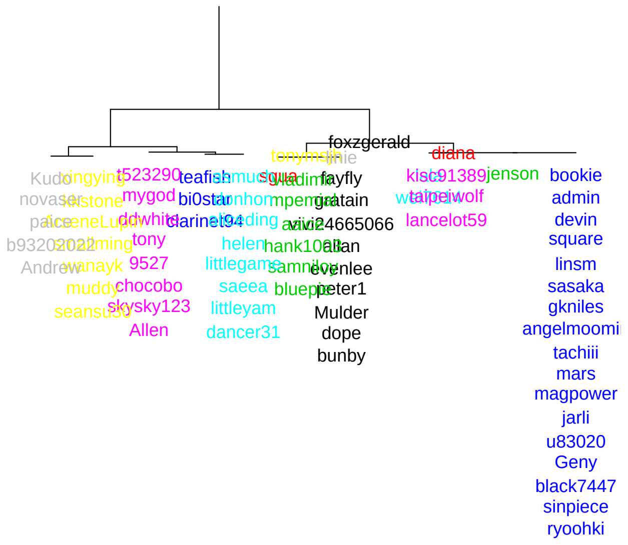

Figure 2: A larger network with more communities

The function provides an option `fan=TRUE` to turn on spreading of the levels of same depths onto concentric arcs:

```
> ihierarchy(icommunity(cf9, labelcf9), fan = TRUE, spread = 0.5)
```

The option `spread` sets the opening of the arc. For example, increasing to spread=1.5 in the present case turns the hierarchy into a circle:

```
> ihierarchy(icommunity(cf9, labelcf9), fan = TRUE, spread = 1.5)
```

2.2 Edges

The `ihierarchy` function displays network vertices in a useful way. It however displays no edges linking the vertices. Seeing is believing, so oftentimes we desire to witness the interconnectivity among vertices. This section introduces a function that draws both vertices and edges.

Again, let's start with an example network.

```
> partite7 <- icommunity(cf7, labelcf7, partite = TRUE)
```

Notice the option `partite=TRUE` in the clustering function `icommunity`. It is to partition the network into communities within which the densities of edges are smaller and between which the densities of edges are larger than average. The default is `partite=FALSE`, giving the traditional communities in terms of links among nodes as in the examples of the previous subsection. The point is that the definition of a community would depend on the context. Section 4 illustrates with examples. Let's continue with network plotting.

The cashflow network now consists of seven communities whose sizes are held in the array:

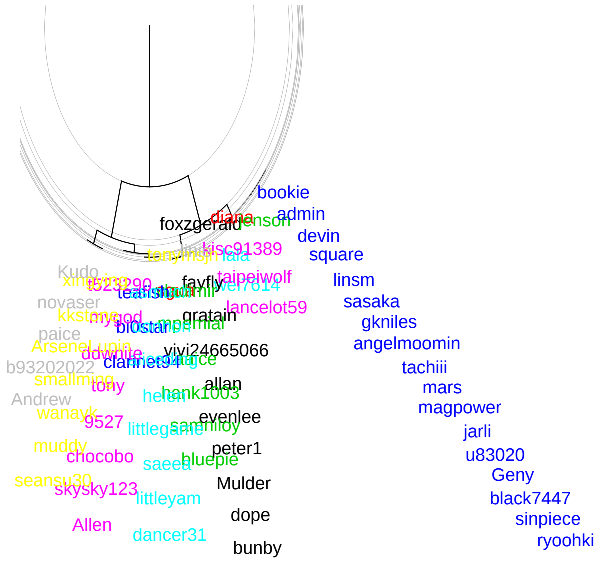

Figure 3: The structure of Fig. 2 but displayed along concentric arcs

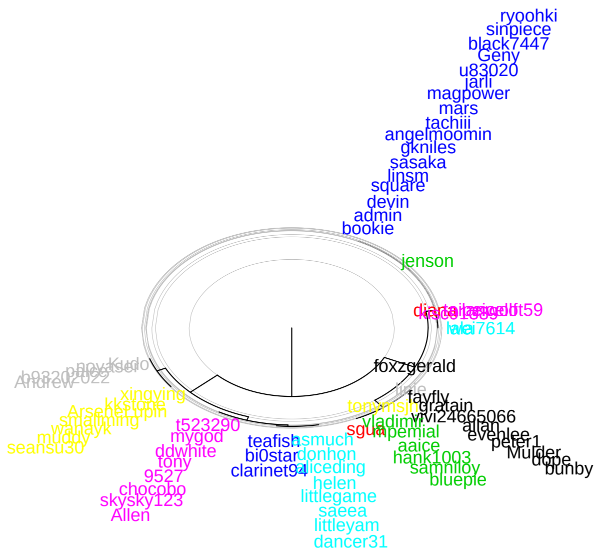

Figure 4: The spreading of Fig. 3 increases to become concentric circles

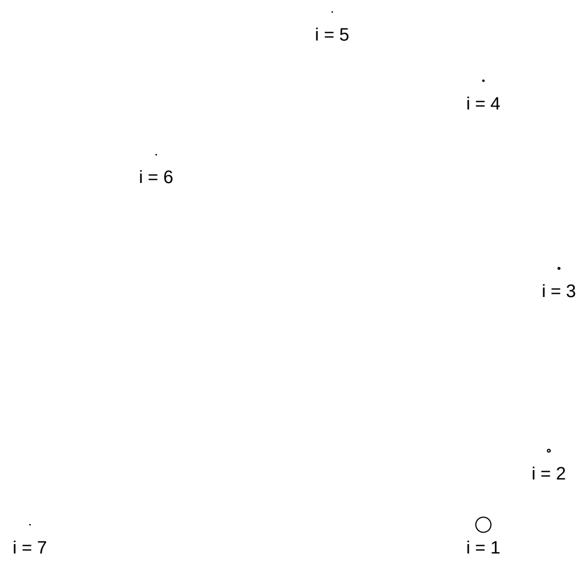

Figure 5: Identified communities are placed along a spiral

```
> partite7$sizes

[1]  3  1  6  1  3 34
```

The problem now is to organize the communities and their vertices on a two-dimension graph for display. We propose to arrange the communities along a spiral (cf. Fig. 5): $spiral = r\theta$ in polar coordinates. Its x and y components in Cartesian coordinates are,

$$\begin{aligned} spiral_{i_x} &= R_i \cos(i\Delta\theta) \\ spiral_{i_y} &= R_i \sin(i\Delta\theta) \end{aligned} \tag{1}$$

where the subscript i indicates the ith community. The `inetplot` function starts plotting the community of the largest size, then the 2nd largest community, ..., etc along the spiral. So, in the `cf7` example, i runs from 1 to 7 with $i = 1$ for the community consisting of 33 vertices, and $i = 2$ for the community having 6 vertices, ..., etc. Organization is this way reveals the modular and hierarchical structure in the network while simplifying the algorithm as to how deep (i.e. how many levels) to display the hierarchy.

The pitch between two neighboring communities on the spiral is set by the `theta` option. That is, `theta=30` dictates $\Delta\theta = 30$ in Eq. (1). The `cf7` example in Fig. 5 has seven communities and thus six pitches crossing 180 degrees at `theta=30`. A larger $\Delta\theta$ spreads the communities farther apart along the spiral. The R_i in Eq. (1) increases with the angle $i\Delta\theta$, i.e. proportional to i by definition: $R_i = i - 1$.

The layout in Figure 5 may have one drawback: when the number of communities goes up, the origin of the spiral populates the majority of the vertices while the tail of the spiral is rather sparse. To make a better use of the space,

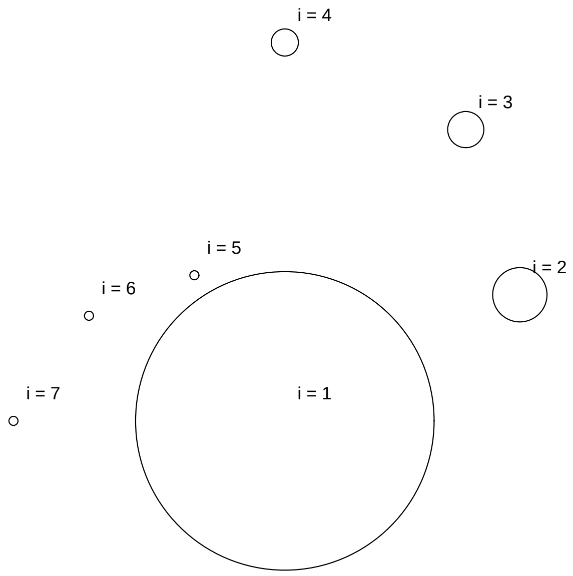

Figure 6: The same as Fig. 5 but the distances R_i's to the spiral origin are weighted by their sizes relative to the largest community

we scale back the distance of community i to community 1 by a factor equal to `sizes[i]/sizes[1]`, i.e. the relative size. Such a shrink in the radius R_i gives rise to a better use of the given plotting area in the sense that the relative sizes of the communities become discernible as demonstrated in Fig. 6. Figures 7 and 8 show further operations on the spiral; the pitch is shorten in Fig. 7 to `theta=15` and the radii R_i's are inflated by a factor of 2 in Fig. 8.

The general idea of network plotting has been given: communities are identified first. They are sorted in a decreasing order according to their sizes and then laid out around a spiral trajectory. Next, we consider how to place the vertices in the designated areas, i.e. within the circles in Figs. 6-8. we provide two ways of organizing members of the community: i) circle and ii) spiral. In the first option `circle=TRUE` which is default, community members are simply placed evenly on the circumference of the circle, whose center has already been determined according the general rule above (cf Fig. 9). The radius of the circle is proportional to the size of the community. The other option `circle=FALSE` is to arranging the members again along a spiral originating at its circular center. The pitch of these component spirals varies concurrently with the value set by the `theta` option for the backbone spiral as illustrated in Fig. 10. Similar to the shrinking factor reining in the backbone spiral, we refrain the component spiral with $|r_j| = \log_2(j)$ where j is the jth vertices in the community. The schema of component spirals along a backbone spiral makes the resulting graph *self-similar*. The spiral option can be advantageous when the community sizes are large as seen in the examples to follow.

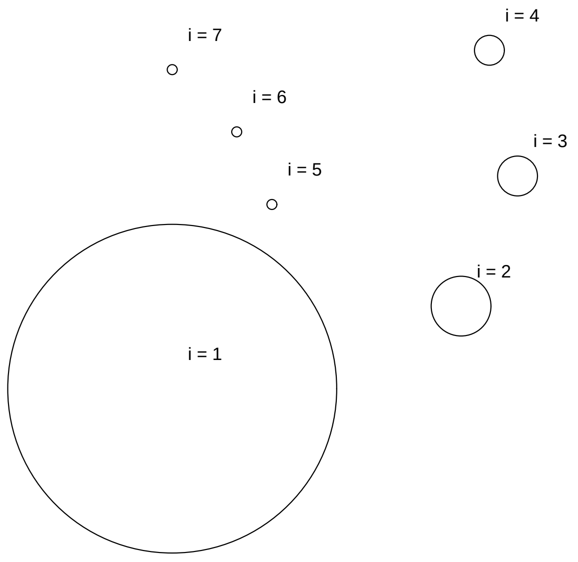

Figure 7: Same as Fig. 6 except that the pitch angle $\Delta\theta$ is reduced

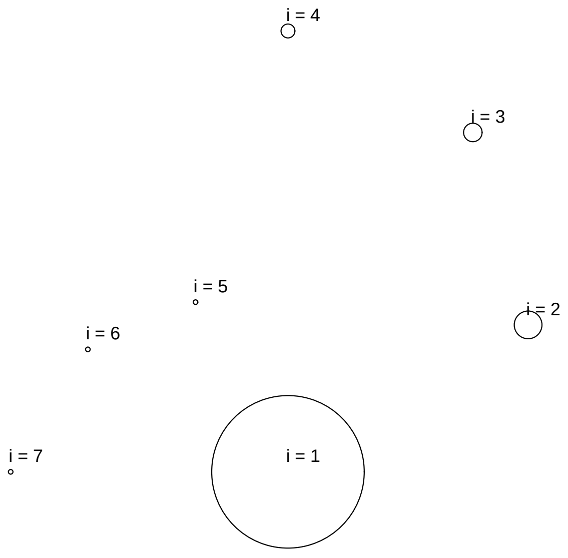

Figure 8: Same as Fig. 6 except that the R_i's are inflated by a common factor

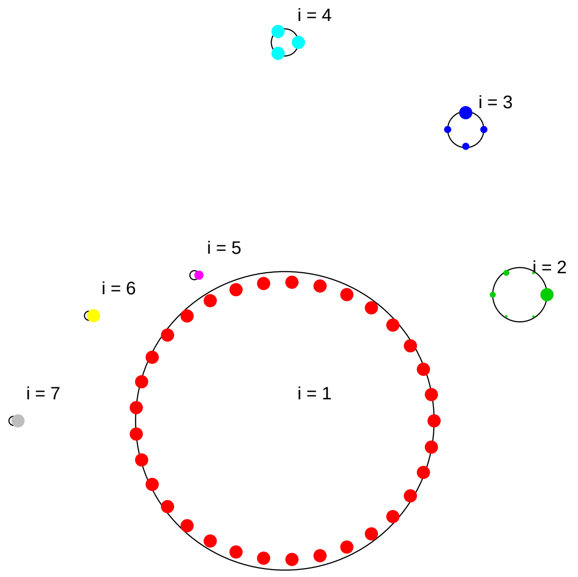

Figure 9: Arrange of community members on circumferences

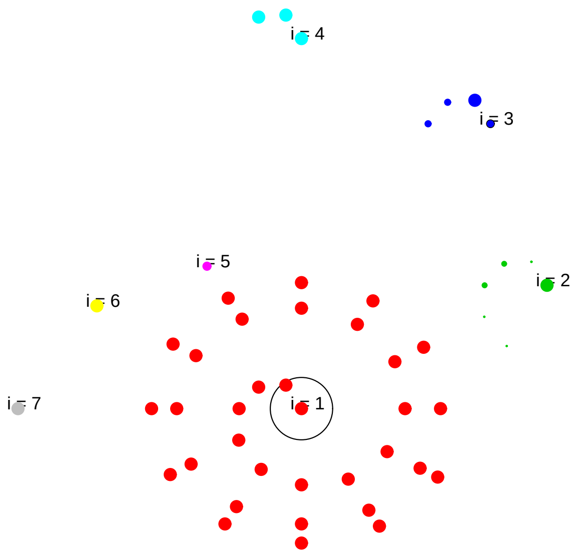

Figure 10: Arrange of community members along spirals

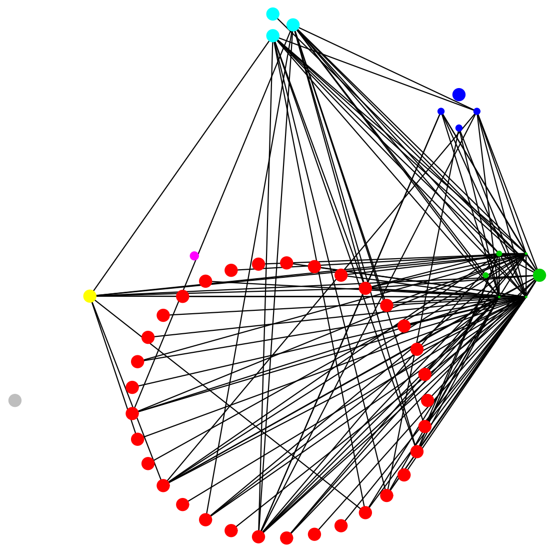

Figure 11: Turn on the edges (cf Fig. 9)

3 Examples

Let's take a look at some outputs from the plotting function.

```
> inetplot(partite7, shaft = 10, circle = TRUE, labels = FALSE)
```

and the component spiral option:

```
> inetplot(partite7, shaft = 50, circle = FALSE, labels = TRUE,
+     points = FALSE)
```

which does not clutter the vertex labels once they are turned on. Note that in this case, we have lengthened the R_i's radii by setting `shaft=50` in comparison with `shaft=10, circle=TRUE` options.

There may be isolated vertices (i.e. nodes without edges to others in the network). The isolated vertices are not shown by default. They can be turned on by setting the `singlets=TRUE` option,

```
> inetplot(partite7, shaft = 10, points = FALSE, singlets = TRUE)
```

with the singlets evenly placed on the circumference of a circle that encloses the connected vertices.

4 Communities and Anti-communities

This section shows the difference between the two options of the logical argument `partite` to the clustering function `icommunity`. In a social network where an edge between two vertices indicates acquaintance between the two persons represented by the two vertices. The natural way to define communities in this

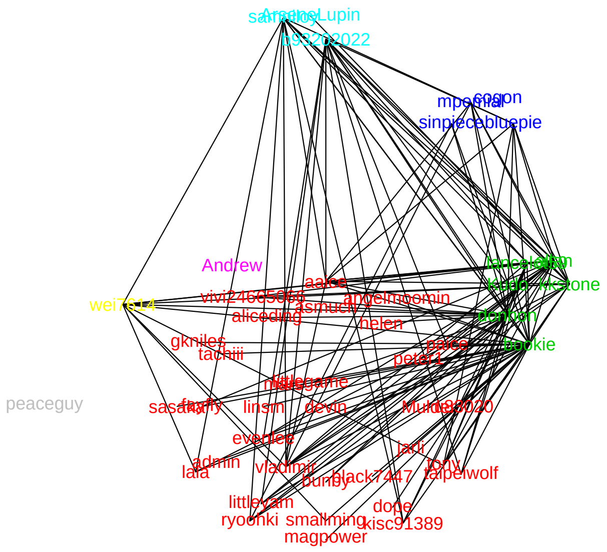

Figure 12: Turn on the vertex labels (cf Fig. 10)

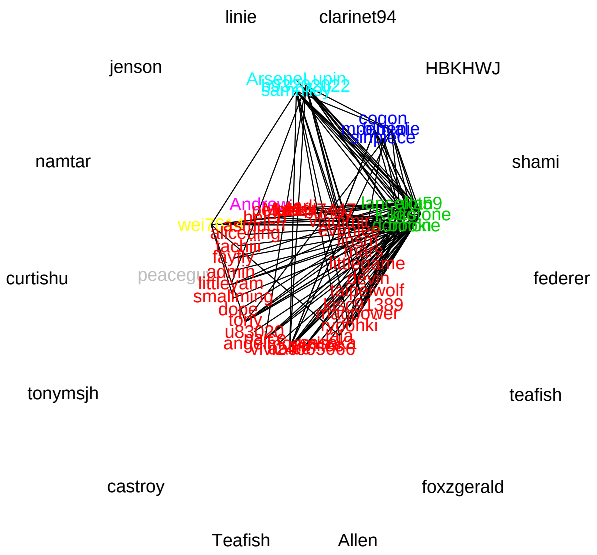

Figure 13: Showing the isolated vertices (cf Fig. 11)

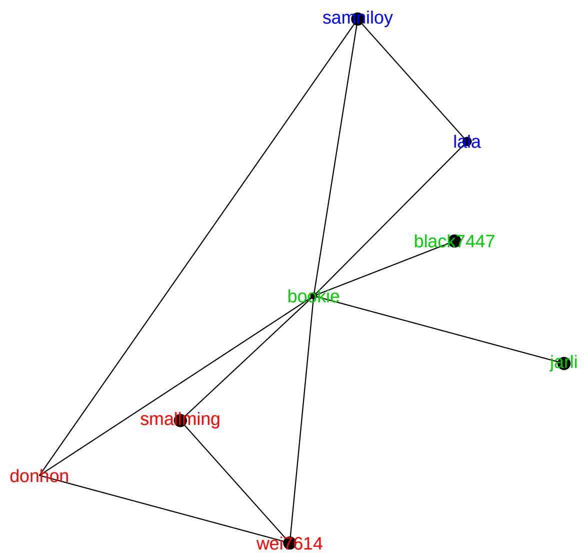

Figure 14: Partition the network in the way that the edge densities within the found communities are maximized

context is to partition the network so that a community has a denser edges among members within than between communities. In this case, we set the partite option to false, `partite=FALSE`.

```
> cluster3 <- icommunity(cf3, labelcf3, partite = FALSE)
> inetplot(cluster3, shaft = 2, circle = FALSE, theta = 33)
```

On other hand, in some cases, a community is better defined where the within-community edges are sparser than between-community ones. Examples include the trading network of a market. In this case, a transaction (edge) between two traders (vertices) indicates that one considers the price of the asset too low while the other thinks the opposite. A transaction is thus matched between them. People sharing the same belief about the trend of the asset thus make no or few trades. They form a community. If this is the case, we set the partite option to `TRUE`. The corresponding example is shown below:

```
> partite3 <- icommunity(cf3, labelcf3, partite = TRUE)
> inetplot(partite3, shaft = 10, circle = FALSE, theta = 33)
```

The above partitioning was on the same network (i.e. adjacency matrix). Note that vertices belonging to a community are in one color. Comparing the two partitioning of Figs. 14 and 15, we see that the difference lies in whether the within- or between-community edges are maximized.

The algorithm[1, 2] works by partitioning the original network into two, each of which is then subject to further division, and so on. A branching process in the algorithm is imagined, and thus a recurrent call in the function `icommunity` is implemented. A nice feature of the algorithm is that the within- or between-community edge densities are compared to what is expected from the empirical

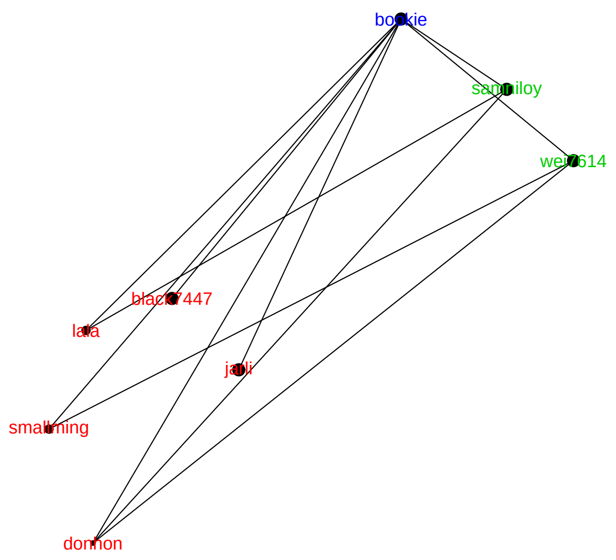

Figure 15: The same data as Fig. 14 but the partition was performed to maximize between-community edge densities

edge distribution. When the the density of edges falls below the average of the empirical distribution, the algorithm stops dividing. As a consequence, the number of communities to be found in the network does not have to be pre-set; the algorithm finds the optimal number of communities on its way. This feature is one of the advantages of the algorithm[1, 2]. The division in each iteration is according to the signs of the elements in the leading eigenvector of the so-called modularity matrix. Readers are referred to Newman (2006) for details[1, 2].

Performance of the algorithm implemented in R depends on the numbers of vertices and edges in the network. As a radical example, it takes 10 minutes for `icommunity` to return the communities in a scale-free network with 600 vertices and 8 edges per vertex on an Intel laptop (Centrino Duo / 1.66 GHz / 1 GB RAM) running Windows XP.

References

[1] M.E.J. Newman, Modularity and community structure in networks. In *Proc. Natl. Acad. Sci. U.S.A.* 103 (2006) 8577–8582.

[2] M.E.J. Newman, Finding community structure in networks using the eigenvectors of matrices. In *Phys. Rev. E* 74 (2006) 036104-1–19.