

## Analysis of Multiplex Social Networks with R

Matteo Magnani  
InfoLab  
Uppsala University

Luca Rossi  
DssLab  
IT University of Copenhagen

Davide Vega  
InfoLab  
Uppsala University

---

### Abstract

Multiplex social networks are characterized by a common set of actors connected through multiple types of relations. The **multinet** package provides a set of R functions to analyze multiplex social networks within the more general framework of multilayer networks, where each type of relation is represented as a layer in the network. The package contains functions to import/export, create and manipulate multilayer networks, implementations of several state-of-the-art multiplex network analysis algorithms, e.g., for centrality measures, layer comparison, community detection and visualization. Internally, the package is mainly written in native C++ and integrated with R using the **Rcpp** (Eddelbuettel and François 2011) library.

*Keywords:* networks, multiplex, multilayer, social network analysis, R.

---

## 1. Introduction and background

Multiplex social networks are characterized by a common set of actors connected through multiple types of relations. In this article we introduce **multinet**, an R package to analyze multiplex social networks represented within the more general framework of multilayer networks. This article is based on version 3.2, available on The Comprehensive R Archive Network<sup>1</sup> (CRAN).

In the multilayer framework, each relation type is represented as a layer, so that for example a layer can be used to store friendship ties while another layer contains working ties among the same set of actors. Such a network can be used to study the relationships between these two types of social ties, for example counting how often colleagues are also friends, and also to study the relationships between actors and types of relations, for example whether a specific actor tends to befriend all her co-workers or to keep these two social contexts separated.

---

<sup>1</sup><https://cran.r-project.org>

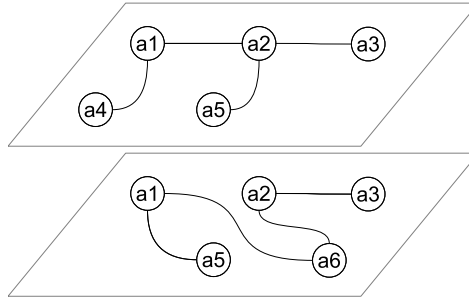


Figure 1: A multiplex network

Throughout this work we will follow the terminology described in (Dickison, Magnani, and Rossi 2016). In particular, we will use the term “multilayer social network” to indicate a network where vertices ( $V$ ) are organized into multiple layers ( $L$ ) and each vertex corresponds to an actor ( $A$ ), where the same actor can be mapped to vertices in different layers. This model, when used to describe multiplex networks, is a simplified version of (Magnani and Rossi 2011), where the same actor can correspond to multiple vertices in the same layer, and (Kivelä, Arenas, Barthelemy, Gleeson, Moreno, and Porter 2014), where layers can be identified by an array of features called aspects (for example, each layer may correspond to both a type of social relationship and a time).

**Definition 1.1 (Multiplex network)** *A multiplex network is a tuple  $(\mathcal{A}, \mathcal{L}, V, E)$  where  $\mathcal{A}$  is a set of actors,  $\mathcal{L}$  is a set of layers,  $V \subseteq \mathcal{A} \times \mathcal{L}$  and  $E \subseteq V \times V$  where  $\forall (a_1, l_1, a_2, l_2) \in E$   $l_1 = l_2$ .*

In Fig. 1 we have represented a multiplex network with two layers, six actors ( $\{a_1, \dots, a_6\}$ ), ten vertices and eight edges, four in each layer. As in multiplex networks each edge exists inside a specific layer, we can also represent an edge as an element of  $A \times A \times L$ , e.g.,  $(a_1, a_2, l)$ . However, **multinet** can also represent more general multilayer networks with edges connecting different layers: as shown in the next section, edges are represented as in the definition above, including two actors and two layers. Please notice that in **multinet** actors do not have to be present in all layers: for each actor we can specify the layers where it exists.

As we will see in the following, **multinet** allows the definition of directed layers, containing directed edges, and also attributes, that we can use to represent edge weights.

### 1.1. Related software

Several packages for network analysis are available in R. Notable examples are **statnet** (Handcock, Hunter, Butts, Goodreau, and Morris 2003), containing a collection of packages such as **sna** (Butts 2016), **network** (Butts 2015, 2008) and **ergm** (Hunter, Handcock, Butts, Goodreau, and Morris 2008), **igraph** (Csardi and Nepusz 2006) and **RSiena** (Ripley, Snijders, Bóda, Vörös, and Preciado 2018). **Multinet** complements this collection with several functions to analyze multiplex networks. In particular, the package provides functions focusing on the multilayer structure of the networks, for example to find how relevant some layers are for an actor or to discover communities spanning multiple layers. Individual layers of a multiplex network, each corresponding to a simple network, can instead be analyzed using the above-

mentioned packages, and in particular **multinet** contains functions to translate the layers into **igraph** objects. The methods provided by **multinet** are distinct from the ones provided by **multiplex** (Ostoic 2018). One can also be interested in MuxViz<sup>2</sup> and mully<sup>3</sup>.

## 2. Basic data management

The **multinet** package defines a class to represent multilayer networks (**Rcpp\_RMLNetwork**). Objects of this type are used as input or returned as output by most functions provided by the package.

Internally, all the objects constituting the network are stored in sets with logarithmic lookup and random access time, implemented as skip lists. This solution is (linearly) less efficient than using a set in the C++ standard library, but supports quick random access to the objects in the set, which is important when synthetic networks are generated. For efficiency reasons, most of the functions in the package are written in native C++ and integrated with R using the **Rcpp** (Eddelbuettel and François 2011) library. Storage requirements for the network class are on the order of the number of vertices plus the total number of edges (inter-layer and intra-layer).

The `ml_empty()` function returns an empty multilayer network, not containing any actor, layer, vertex or edge<sup>4</sup>. The function accepts an optional character argument `name`, indicating the name of the network.

```
R> ml_empty()
```

```
Multilayer Network [0 actors, 0 layers, 0 vertices, 0 edges (0,0)]
```

For convenience, the call to any of the network's constructors and readers returns an **S4** object compatible with the R `print` function. Otherwise, all the other functions' return types are, by design, either (i) a named list of elements (if the data is not relational) or (ii) a data frame.

### 2.1. Adding, retrieving and deleting network objects

Objects in an **Rcpp\_RMLNetwork** object can be queried using a set of utility functions. Built-in functions for retrieving and updating objects have the same signature name, following the pattern: `<op>_<objects>_ml`, where `<objects>` can be `actors`, `layers`, `vertices` or `edges`, and `<op>` is either blank, if we want to list the objects, or is the name of a specific operation: `num`, to compute the number of objects of the requested type, `add` or `delete`. If the number of actors is requested without specifying any layer, the total number of actors is returned, including those not present in any layer.

All the aforementioned functions require an **Rcpp\_RMLNetwork** object as first argument. Listing functions operating on actors and vertices also require an array of layer names: only the actors/vertices in the input layers are returned. If the array is empty, all the actors/vertices

<sup>2</sup><https://github.com/manlius/muxViz>

<sup>3</sup><https://github.com/frankkramer-lab/mully>

<sup>4</sup>Other ways to create networks, explained later, are the function `read_ml()` to load networks from files and the `grow_ml()` function to produce synthetic networks.

in the network are returned. Listing functions operating on edges, instead, require two parameters: one indicating the layer(s) from where the edges to be extracted start, and a second one with the layer(s) where the edges to be extracted end. If an empty list of starting layers is passed (default), all the layers are considered, while if an empty list of ending layers is passed (default), the ending layers are set as equal to those in the first parameter.

Now we can show a small example of how these functions work together. We start by creating an empty network with two layers, named UL (upper layer) and BL (bottom layer), respectively.

```
R> net <- ml_empty()
R> add_layers_ml(net, c("UL", "BL"))
R> layers_ml(net)
```

```
[1] "UL" "BL"
```

New layers are by default undirected, that is, edges added to them are treated as undirected. Directed layers are created by setting the `directed` parameter to `TRUE`, or using the `set_directed_ml()` function, which is necessary if we want to set directed intralayer edges. This function takes an `Rcpp_RMLNetwork` object and a `directionality` data frame as input. The next fragment of code changes the directionality of the inter-layer edges between the bottom and upper layers.

```
R> dir <- data.frame(layer1="UL", layer2="BL", dir=1)
R> set_directed_ml(net, dir)
R> is_directed_ml(net)
```

	layer1	layer2	dir
1	UL	UL	0
2	UL	BL	1
3	BL	UL	1
4	BL	BL	0

The next step to populate a network is to add vertices, where a pair actor-layer defines a vertex. Actors are identified by their name, and are created automatically when vertices are added to the network.

```
R> vertices <- data.frame(
+   actors = c("A", "B", "C", "A", "B", "C"),
+   layers = c("UL", "UL", "UL", "BL", "BL", "BL"))
R> vertices
```

	actors	layers
1	A	UL
2	B	UL
3	C	UL
4	A	BL
5	B	BL
6	C	BL

```
R> add_vertices_ml(net, vertices)
R> vertices_ml(net)
```

```
  actor layer
1     A    UL
2     B    UL
3     C    UL
4     A    BL
5     B    BL
6     C    BL
```

From the previous command you can see how the objects in a network are stored into (mathematical) sets, that is, they are unordered: we cannot assume that actor *A* will always be listed before actor *B*, and we have to sort the results if we want to keep a specific order.

We can now add some edges, in this case between all the vertices in the upper layer and between vertices *A* and *C* in the bottom one. Inter-layer edges, although supported, are not exemplified in this article focusing on multiplex networks.

```
R> edges <- data.frame(
+   actors_from = c("A", "A", "B", "A"),
+   layers_from = c("UL", "UL", "UL", "BL"),
+   actors_to   = c("B", "C", "C", "C"),
+   layers_to   = c("UL", "UL", "UL", "BL"))
R> edges
```

```
  actors_from layers_from actors_to layers_to
1           A          UL          B          UL
2           A          UL          C          UL
3           B          UL          C          UL
4           A          BL          C          BL
```

Now we can add these edges to the network, and observe the result.

```
R> add_edges_ml(net, edges)
R> edges_ml(net)
```

```
  from_actor from_layer to_actor to_layer dir
1           B          UL          C          UL  0
2           A          UL          C          UL  0
3           A          UL          B          UL  0
4           A          BL          C          BL  0
```

```
R> edges_ml(net, layers1 = "BL")
```

```
  from_actor from_layer to_actor to_layer dir
1           A          BL          C          BL  0
```

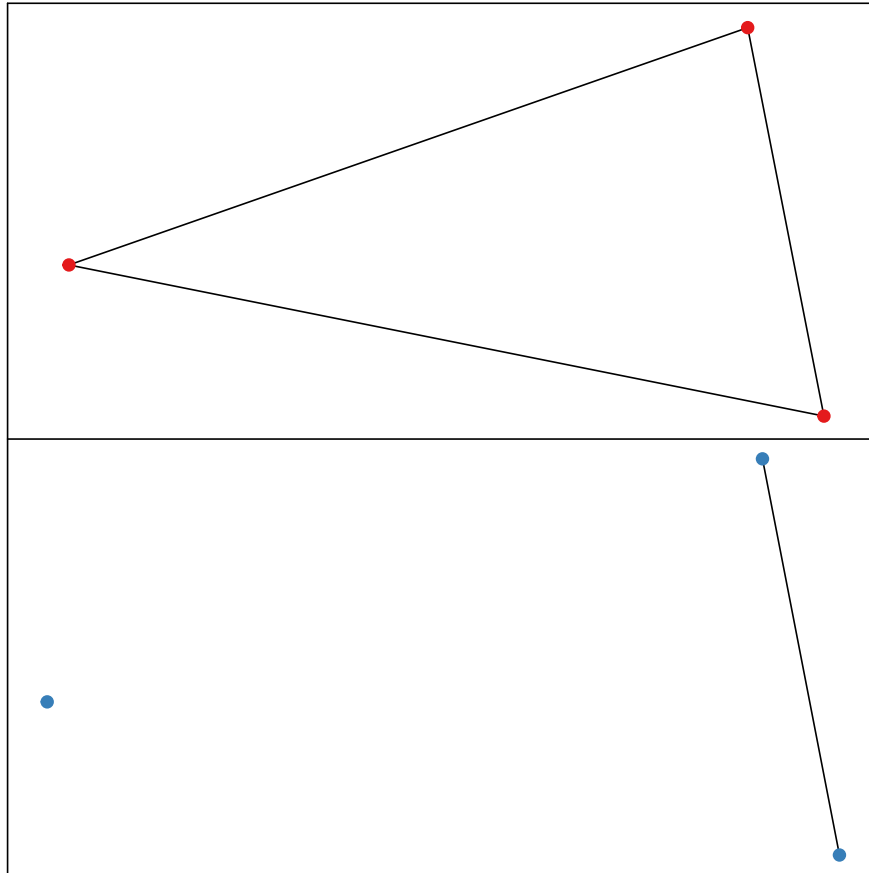


Figure 2: A visualization of our first multiplex network

In Figure 2 we can see a plot of the network we just created. More details on visualizing multiplex networks are provided later.

## 2.2. Handling attributes

When we study a multiplex network, we can be interested in representing different types of actors, add some categorical attribute to vertices or use a numerical value to represent the strength of the ties. The **multinet** package provides a set of functions to create attributes and add and retrieve attribute values. `attributes_ml()` returns a data frame with two columns, the **name** of the attribute and its **type**. As most of the functions in the package, the function accepts a filtering parameter, **target**, to limit the query to specific types of objects: “actor” (attributes attached to actors), “vertex” (attributes attached to vertices) or “edge” (attributes attached to edges). All the functions handling attributes use **target = "actor"** by default.

```
R> attributes_ml(net)
```

```
[1] name type
<0 rows> (or 0-length row.names)
```

The list of attributes of a newly created network is empty. We can create attributes by calling the `add_attributes_ml()` function and passing an `Rcpp_RMLNetwork` object, names of the `attributes`, types of the attributes (“string” or “numeric”) and the `target` as parameters. For example, the following code creates two string attributes for actors (notice that “actor” is the default target, and “string” is the default attribute type):

```
R> add_attributes_ml(net, c("name", "surname"))
R> attributes_ml(net)

  name  type
1  name string
2 surname string
```

Using the `add_attributes_ml()` function we can also specify different attributes for vertices and edges on individual layers, for which we must supply the `layer` parameter. If we want, instead, to manage inter-layer edges two parameters are needed, `layer1` and `layer2`, so that the attribute only applies to inter-layer edges from the first layer to the second and vice-versa. The example below shows how to use these parameters in practice to create a string attribute for the vertices in the bottom layer.

```
R> add_attributes_ml(net, "username", type = "string", target = "vertex",
+   layer = "BL")
R> attributes_ml(net, target = "vertex")

 layer  name  type
1     BL username string
```

At this point the `get_values_ml()` and `set_values_ml()` functions can be used to set and retrieve attribute values. Notice that if an attribute has not been set, then a default value is returned, in particular an empty string in the following example.

```
R> set_values_ml(net, "name", c("A", "B"), values = c("Alice", "Scronto"))
R> get_values_ml(net, "name", c("A", "C"))

 value
1 Alice
2
```

### 2.3. Input/output

In the previous sections we have introduced the `Rcpp_RMLNetwork` class and various methods to modify `Rcpp_RMLNetwork` objects. However, users would more often create `Rcpp_RMLNetwork` objects by reading them from a file, artificially generating them, or loading some of the datasets directly available in the package.

The **multinet** package provides two input/output functions: `read_ml()` and `write_ml()`. Networks can be read from files using a package-specific text-based format, and written to file

using the same format or the GraphML syntax<sup>5</sup>. GraphML is extensively used, e.g., by graph software such as iGraph, Gephy, yEd, as well as in the boost C++ libraries. The **multinet** format is not compatible with other packages, but it allows us to specify various details, such as the directionality of intra-layer edges and attributes, as in the following example:

```
#VERSION
3.0

#TYPE
multiplex

#LAYERS
research, UNDIRECTED
twitter, DIRECTED

#ACTOR ATTRIBUTES
affiliation,STRING

#VERTEX ATTRIBUTES
twitter, num_tweets, NUMERIC

#EDGE ATTRIBUTES
research, num_publications, NUMERIC

#ACTORS
Luca,ITU
Matteo,UU
Davide,UU

#VERTICES
Luca,twitter,53
Matteo,twitter,13

#EDGES

Luca,Matteo,research,9
Luca,Matteo,twitter
```

When we read this multiplex network we can also specify that we want all the actors to be present in all the layers, using the `align` parameter. The difference between the two obtained networks can be seen by checking the basic network statistics:

```
R> net <- read_ml(file = "example_io.mpx")
R> net
```

```
Multilayer Network [3 actors, 2 layers, 4 vertices, 2 edges (2,0)]
```

---

<sup>5</sup><http://graphml.graphdrawing.org>



```
R> aligned_net <- read_ml("example_io.mpx", align = TRUE)
R> aligned_net
```

Multilayer Network [3 actors, 2 layers, 6 vertices, 2 edges (2,0)]

Both `Rcpp_RMLNetwork` objects, `net` and `aligned_net`, have two layers and three actors; but the `align = TRUE` parameter in the second call to the `read_ml()` adds a new vertex to each layer for every actor in the input file.

When no special information is needed, e.g., there are no attributes, no isolated nodes and all edges are undirected, the format becomes as simple as a list of layer-annotated edges:

```
Luca,Matteo,research
Davide,Matteo,research
Luca,Matteo,friendship
```

A multiplex network can also be created starting from **igraph** objects, where each graph represents a layer. For this to be possible, the vertices of the graphs must have a `name` attribute indicating the name of the corresponding actor.

For example, consider the following graphs:

```
R> l1 <- read.graph("example_igraph1.dat", format = "ncol")
R> l1
```

```
IGRAPH 820640d UN-- 3 3 --
+ attr: name (v/c)
+ edges from 820640d (vertex names):
[1] A--B A--C B--C
```

```
R> l2 <- read.graph("example_igraph2.dat", format = "ncol")
R> l2
```

```
IGRAPH f305b09 UN-- 2 1 --
+ attr: name (v/c)
+ edge from f305b09 (vertex names):
[1] A--C
```

They can be added as layers of a multiplex network as follows:

```
R> n <- ml_empty()
R> add_igraph_layer_ml(n, l1, "layer1")
R> add_igraph_layer_ml(n, l2, "layer2")
R> n
```

Multilayer Network [3 actors, 2 layers, 5 vertices, 4 edges (4,0)]

```
R> edges_ml(n)
```

```

  from_actor from_layer to_actor to_layer dir
1          A   layer2      C   layer2  0
2          A   layer1      B   layer1  0
3          B   layer1      C   layer1  0
4          A   layer1      C   layer1  0

```

## 2.4. Generation

The package also provides basic functionality to generate synthetic multiplex networks, following the approach proposed by [Magnani and Rossi \(2013a\)](#). This problem is approached by allowing layers to evolve at different rates, based on internal or external dynamics. Internal dynamics can be modelled using existing network models (for example, preferential attachment), assuming that how the layer grows can be explained only looking at the layer itself. External dynamics involve importing edges from other layers. Within this perspective the intuition is that relations existing on a layer might naturally expand over time into other layers (e.g., co-workers starting to add each other as friends on Facebook). The package also allows different growing rates for different layers.

In the following example we create a multiplex network with 3 layers based on the Preferential Attachment ([Barabási and Albert 1999](#)) and the Erdos-Renyi models ([Erdos and Rényi 1960](#)). The first and last layers will only evolve according to their internal models (`pr.external = 0`), while the second will have a probability of .8 of evolving according to external dynamics, that is, importing edges from other layers (`pr.external = .8`). Note that all the probability vectors must have the same number of fields, one for each layer. By defining `pr.internal` and `pr.external`, we are also implicitly defining `pr.no.action` (1 minus the other probabilities, for each field/layer). In the example, the third layer grows at a lower speed than the others, having an (implicitly defined) `pr.no.action = .1`.

```

R> models_mix <- c(evolution_pa_ml(3, 1), evolution_er_ml(100),
+   evolution_er_ml(100))
R> pr.internal <- c(1, .2, .9)
R> pr.external <- c(0, .8, 0)

```

The probability to import edges from the other layers in case external events happen is specified using a dependency matrix. The following matrix specifies that the second layer should import edges from the first layer with probability 1 if an external evolutionary event is triggered. It is expected that the values on each row of the matrix add to 1.

```

R> dependency <- matrix(c(1, 1, 0, 0, 0, 0, 0, 0, 1), 3, 3)
R> dependency

```

```

      [,1] [,2] [,3]
[1,]    1    0    0
[2,]    1    0    0
[3,]    0    0    1

```

We can now generate the network, with 100 actors and 100 growing steps.

```
R> ml_generated_mix <- grow_ml(100, 100, models_mix, pr.internal, pr.external,  
+ dependency)
```

```
R> num_edges_ml(ml_generated_mix, layers1 = "l0")
```

```
[1] 100
```

```
R> num_edges_ml(ml_generated_mix, layers1 = "l1")
```

```
[1] 64
```

```
R> num_edges_ml(ml_generated_mix, layers1 = "l2")
```

```
[1] 89
```

## 2.5. Predefined data

Another way to obtain network data without having to manually construct it is to load some well-known networks already available inside the package. These are loaded using functions beginning with “ml”, followed by the name of the network, e.g., `ml_florentine()`.

In the remainder of the article we will use the AUCS network, included in the current version of the **multinet** package as an example dataset and often used in the literature to test new methods. The data, described by [Dickison \*et al.\* \(2016\)](#), were collected at a university research department and include five types of online and offline relations. The population consists of 61 employees, including professors, postdocs, PhD students and administrative staff.

```
R> net <- ml_aucs()
```

```
R> net
```

```
Multilayer Network [61 actors, 5 layers, 224 vertices, 620 edges (620,0)]
```

```
R> layers_ml(net)
```

```
[1] "lunch"      "coauthor"  "leisure"  "work"      "facebook"
```

## 3. Data exploration

**Multinet** provides a basic visualisation function. The `multiforce` layout, used for all graph visualizations in this article, is described in ([Fatemi, Magnani, and Salehi 2018](#)) and works as shown in [Fig. 3](#). Inside each layer the algorithm uses a force-based approach as in the commonly used Fruchterman-Reingold layout, to keep adjacent nodes close to each other, but in addition it also applies forces to the vertices so that they remain aligned to vertices with the same actor on other layers. These two intra-layer and inter-layer forces can also be weighted. In addition, in case the network contains multiple components it is possible to

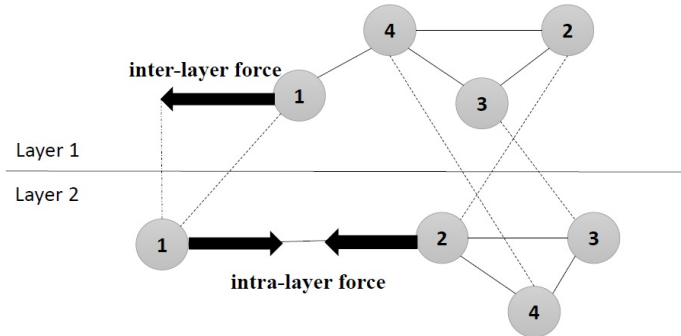


Figure 3: An explanation of multiforce

specify a `gravity` force, that attracts the vertices towards the center of their layer so that the distance between the components is controlled.

If the graph is very small we can produce a default visualization just by executing `plot(net)`, which in turns would use the `multiforce` layout with default settings, but to make the plot of the AUCS network more readable we shall add a few details. In particular: (1) we explicitly compute a layout that draws each layer independently of the others, as declared by setting interlayer weights (`w_inter`) to 0, (2) we plot the layers on two rows, to better use the space on the page (`grid`), (3) we remove the labels from the vertices, to increase readability (`vertex.labels = ""`), and (4) we add a legend with the names of the layers<sup>6</sup>. The result of the following command is shown in Fig. 4.

```
R> l1 <- layout_multiforce_ml(net, w_inter = 0, gravity = 1)
R> plot(net, layout = l1, grid = c(2, 3), vertex.labels = "",
+       legend.x = "bottomright", legend.inset = c(.05, .05)
+ )
```

As another example, we may want to align all the actors with respect to their layout in the first layer. To achieve this we can set to 0 all the intra-layer weights in the other layers, so that they do not affect the layout. However, to obtain a good layout we may have to repeat the process and sometimes update some parameters. First, the iterative way in which forces are computed may result in small variations with respect to the position of the same actor on different layers. Second, the results of different executions are different in general, because the initial positions of the vertices are chosen at random. Finally, one should consider that not all actors exist in all layers; for example, if we set intra-layer forces to 0 all actors that are only present on that layer will end up in a random position. The result of the following layout is shown in Fig. 5; please notice that not all actors are present on all layers in this network.

```
R> l2 <- layout_multiforce_ml(net, w_inter = 1,
+   w_in = c(1, 0, 0, 0, 0),
+   gravity = c(1, 0, 0, 0, 0)
+ )
```

<sup>6</sup>An aligned layout, with the same actor visualized in approximately the same position in all layers, can be achieved by setting a high value of `w_inter`.

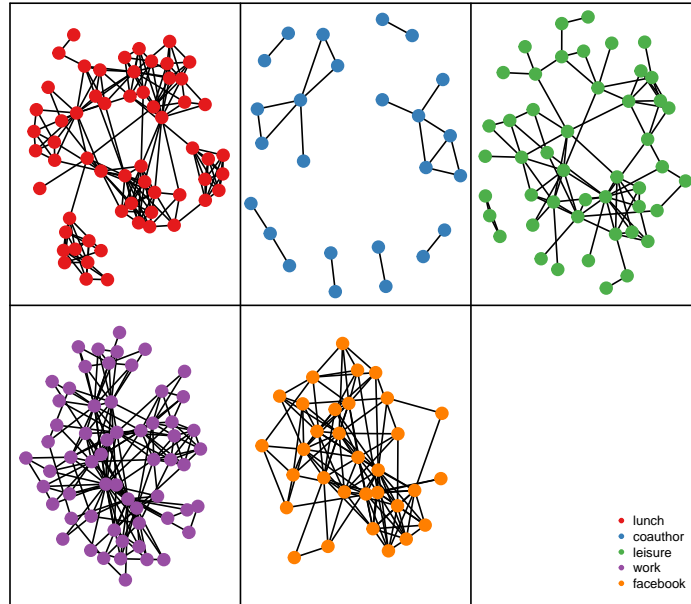


Figure 4: A basic visualization of the AUCS network

We can also use the attributes to inspect the relationship between the role of the actors and the topology of the network. We start by retrieving the role of each vertex (`vertex_roles`).

```
R> roles <- get_values_ml(net, actors = vertices_ml(net)[[1]], attribute = "role")
R> head(roles)
```

```
      value
1      PhD
2     Admin
3      PhD
4 Professor
5 Associate
6      PhD
```

The utility function `values2graphics()` processes the attribute values and generates information that can be used to plot these values in the network drawing. The result is shown in Fig. 6.

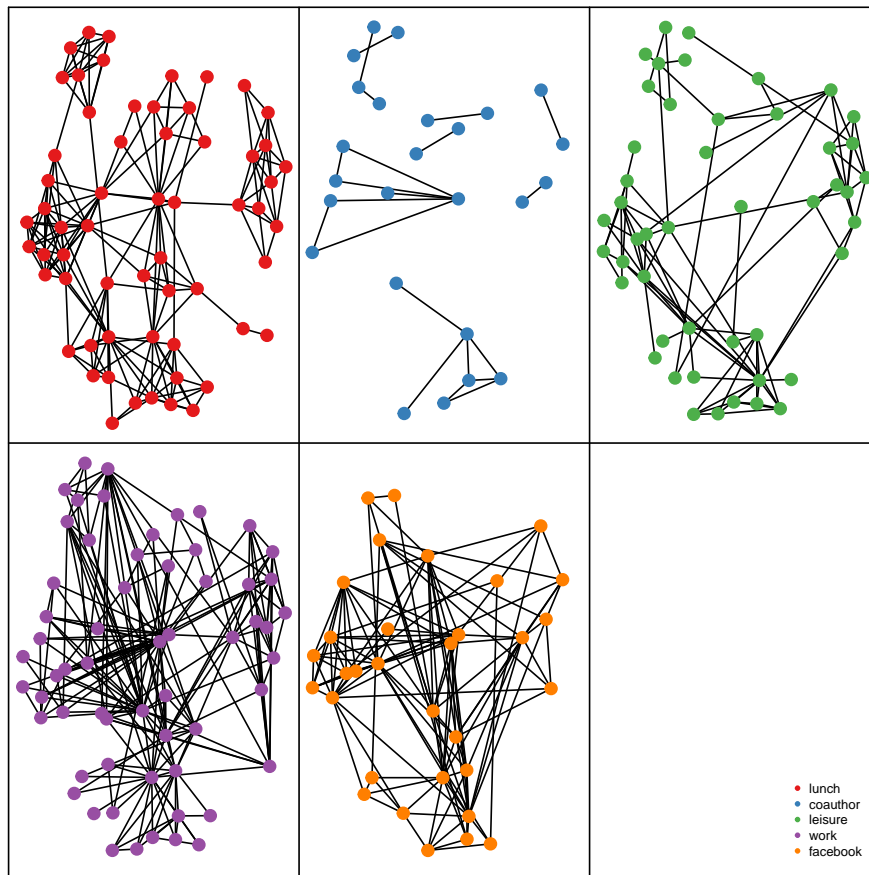


Figure 5: An aligned visualization of the AUCS network, following the layout in the first layer

```
R> gr <- values2graphics(roles)
R> plot(net, layout = 1, grid = c(2,3), vertex.labels = "", vertex.color = gr$color)
R> legend("bottomright", legend = gr$legend.text, col = gr$legend.col,
+       pt.bg = gr$legend.col, pch = gr$legend.pch, bty = "n",
+       pt.cex = 1, cex = .5, inset = c(0.05, 0.05)
+ )
R> legend("bottomright", legend = layers_ml(net), bty = "n", pch=20,
+       pt.cex = 1, cex = .5, inset = c(0.2, 0.05)
+ )
```

## 4. Measuring a network

A traditional way of measuring a multiplex network is to focus on each layer at a time, considering it as an independent graph. For example, the `summary()` function computes a selection of measures on all the layers, and also on the flattened network.

```
R> summary(net)
```

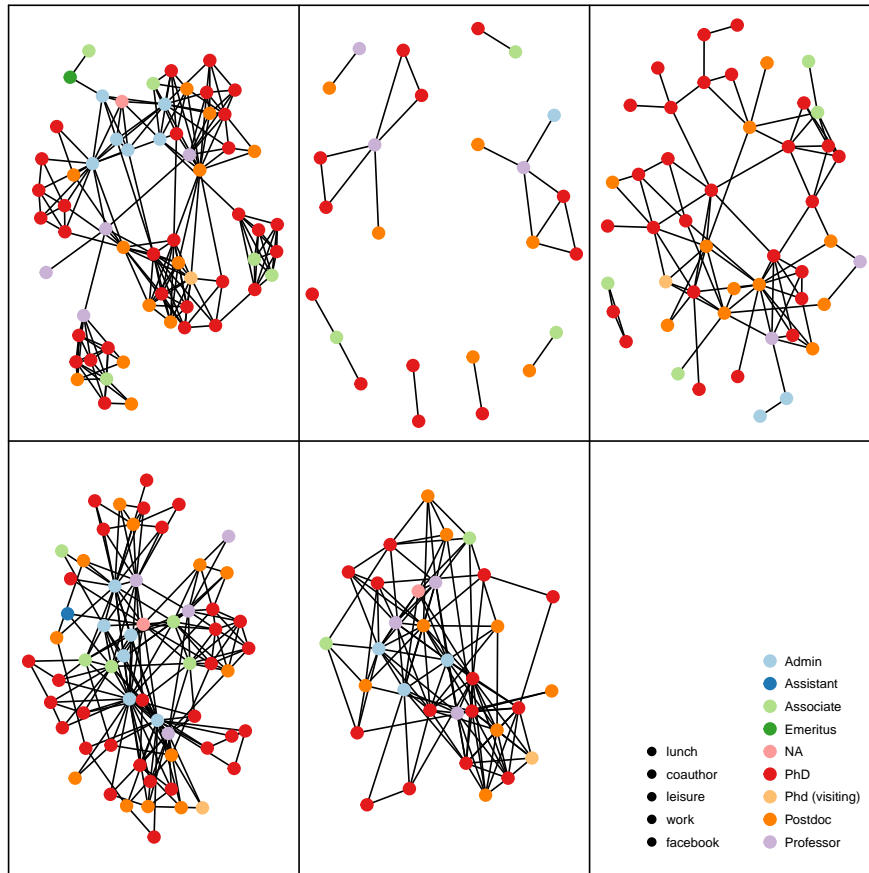


Figure 6: A visualization of the AUCS network where vertex colors represent roles

	n	m	dir	nc	slc	dens	cc	apl	dia
_flat_	61	620	0	1	61	0.33879781	0.4761508	2.062842	4
coauthor	25	21	0	8	6	0.07000000	0.4285714	1.500000	3
facebook	32	124	0	1	32	0.25000000	0.4805687	1.955645	4
leisure	47	88	0	2	44	0.08140611	0.3430657	3.115911	8
lunch	60	193	0	1	60	0.10903955	0.5689261	3.188701	7
work	60	194	0	1	60	0.10960452	0.3387863	2.390395	4

The columns indicate:

1. **n** order (number of vertices)
2. **m** size (number of edges)
3. **dir** directionality: In case of directed relations (e.g. sending a message to someone) the value will be 1.
4. **nc** number of connected components: The number of subgraphs in which any two vertices are connected to each other by paths. In case of directed layers, edge directionality

will be taken into account, thus the number of strong components will be reported.

5. **dens** density: The ratio between the number of edges and the number of possible edges.
6. **cc** clustering coefficient: The ratio between the triangles and the connected triples in the layer. In case of directed layers, edges will be treated as undirected.
7. **apl** average path length: The average graph-distance between all pairs of vertices in the layer.
8. **dia** diameter: The longest graph distance between any two vertices in the layer.

To compute other functions or perform another type of layer-by-layer analysis we can convert the layers into **igraph** objects, using the `as.igraph()` function, for a single (group of) layer(s), or the `as.list()` function to obtain a list with all the layers as **igraph** objects in addition to the flattened network. Once the **igraph** objects have been generated, all the network measures available in **igraph** can be computed. The following code, for example, uses **igraph** to compute the degree centralization of the **facebook** layer:

```
R> layers <- as.list(net)
R> names(layers)

[1] "_flat_" "coauthor" "facebook" "leisure" "lunch" "work"

R> centralization.degree(layers[[3]])$centralization

[1] 0.233871
```

As another example of layer-by-layer analysis, Fig. 7 shows the degree distribution of each layer, and also the degree distribution of the flattened network.

#### 4.1. Layer comparison

In addition to a layer-by-layer analysis, we can compare layers using several different approaches. All the methods mentioned in this section are explained and evaluated in (Brodka, Chmiel, Magnani, and Ragozini 2018).

For example, to quantify the difference between the degree distributions in different layers we can use the `layer_comparison_ml()` function to produce a table with pair-wise comparisons. The following code computes the dissimilarity between degree distributions, computed using the Jeffrey dissimilarity function (the higher the values, the most dissimilar the two layers). In this case it is possible to observe how the degree distribution of **work** and **coauthor** layers are remarkably similar pointing at similar network structures (probably with few highly connected hubs) and how they are both dissimilar from e.g. **lunch** that shows a different network structure, probably shaped by different social dynamics.

```
R> layer_comparison_ml(net, method = "jeffrey.degree")
```



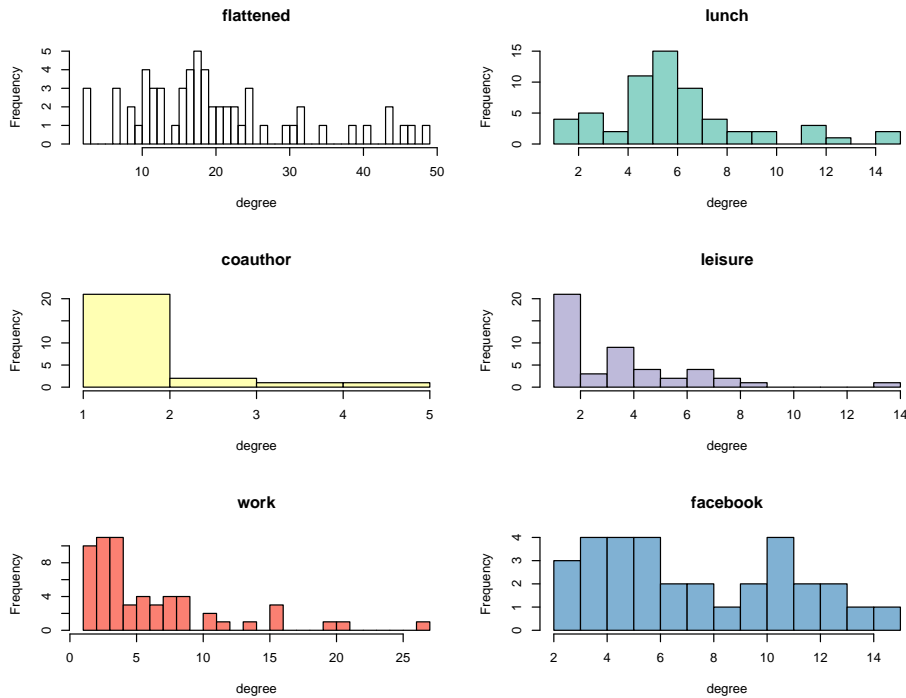


Figure 7: Frequency distribution for vertices degree on each layer.

	lunch	coauthor	leisure	work	facebook
lunch	0.000000	2.8966530	1.3288250	0.8372414	0.4207678
coauthor	2.8966530	0.0000000	0.4521076	0.5917494	2.0214010
leisure	1.3288250	0.4521076	0.0000000	0.2118452	1.0177980
work	0.8372414	0.5917494	0.2118452	0.0000000	0.7106788
facebook	0.4207678	2.0214010	1.0177980	0.7106788	0.0000000

The `layer_comparison_ml()` function can also be used to compute multiplex-specific comparisons considering the fact that the same actors may be present on the different layers. In fact, one important comparison can be made to check to what extent this is true:

```
R> layer_comparison_ml(net, method = "jaccard.actors")
```

	lunch	coauthor	leisure	work	facebook
lunch	1.0000000	0.4166667	0.7833333	0.9672131	0.5333333
coauthor	0.4166667	1.0000000	0.4117647	0.4166667	0.2954545
leisure	0.7833333	0.4117647	1.0000000	0.7833333	0.5192308
work	0.9672131	0.4166667	0.7833333	1.0000000	0.5333333
facebook	0.5333333	0.2954545	0.5192308	0.5333333	1.0000000

The function returns 0 if there are no common actors between the pair of layers, and 1 if the same actors are present in the two layers. If there is a strong overlapping between the actors, then we can ask whether actors having a high (or low) degree on one layer behave similarly in other layers. To do this we can compute the correlation between the degrees:

```
R> layer_comparison_ml(net, method = "pearson.degree")
```

	lunch	coauthor	leisure	work	facebook
lunch	1.0000000	0.1486368	0.28151667	0.24647515	0.3125598
coauthor	0.1486368	1.0000000	0.48084471	0.42719422	0.5472774
leisure	0.2815167	0.4808447	1.00000000	0.06805041	0.3781743
work	0.2464752	0.4271942	0.06805041	1.00000000	0.5406011
facebook	0.3125598	0.5472774	0.37817432	0.54060113	1.0000000

The Pearson (or linear) correlation between the degree of actors in the two layers is in the interval  $[-1, 1]$ . The smallest value (-1) indicates that high-degree actors in one layer are low-degree in the other and vice versa, while the largest value (1) is returned if high-degree (resp., low-degree) actors in one layer are high-degree (resp., low-degree) actors in the other. It is important to note that the correlation only depends on the number of incident edges for each pair (actor, layer), and not on which actors are adjacent: they can be the same or different actors.

We can also check to what extent actors are adjacent to the same other actors in different layers, by checking the amount of overlapping between edges in the two layers, which will be 0 if no actors that are adjacent in one layer are also adjacent in the other and 1 if all pairs of actors are either adjacent in both layers or in none.

As an example of how to use these functions we can look at the relation between the layers `lunch` and `work`. From the Jeffrey dissimilarity function we know that `lunch` and `work` show a very different degree distribution. Nevertheless, looking at Jaccard overlapping on the actors we also see that the two layers share many actors. This clearly rises the question if those actors, that are present in both layers, behave in a similar way. Observing the Jaccard overlapping of the edges we can see that while several edges actually exist in both layers (edge overlap between `lunch` and `work` is actually the highest in the multilayer network) vertices that are highly connected on a layer are not necessarily highly connected on the other layer. This is compatible with the intuition obtained from the analysis of the degree distribution similarity: the two layers, while composed largely by the same actors and with several edges in common, are actually organized according to different social dynamics.

```
R> layer_comparison_ml(net, method = "jaccard.edges")
```

	lunch	coauthor	leisure	work	facebook
lunch	1.00000000	0.06467662	0.2772727	0.33910035	0.17843866
coauthor	0.06467662	1.00000000	0.1010101	0.09137056	0.05839416
leisure	0.27727273	0.10101010	1.00000000	0.20512821	0.15846995
work	0.33910035	0.09137056	0.2051282	1.00000000	0.18656716
facebook	0.17843866	0.05839416	0.1584699	0.18656716	1.00000000

The package provides additional similarity functions, listed in Table 1.

## 4.2. Degree and degree deviation

Various functions can be used to measure individual actors. As a starting point, the following is the list of highest-degree actors on the whole multiplex network:

Overlapping	Distribution dissimilarity	Correlation
jaccard.actors	dissimilarity.degree	pearson.degree
jaccard.edges	KL.degree	rho.degree
jaccard.triangles	jeffrey.degree	
coverage.actors		
coverage.edges		
coverage.triangles		
sm.actors		
sm.edges		
sm.triangles		
rr.actors		
rr.edges		
rr.triangles		
kulczynski2.actors		
kulczynski2.edges		
kulczynski2.triangles		
hamann.actors		
hamann.edges		
hamann.triangles		

Table 1: Similarity functions provided in the package.

```
R> deg <- degree_ml(net)
R> top_degrees <- head(deg[order(-deg)])
R> top_actors <- head(actors_ml(net)[order(-deg)])
R> top_degrees
```

```
[1] 49 47 46 44 44 41
```

However, in a multiplex context degree becomes a layer-specific measure. We can no longer just ask “who is the most central actor” but we should ask “who is the most central actor on this layer?” Let us see how the most central actors look like when we “unpack” their centrality on the different layers:

```
R> data.frame(
+   facebook = degree_ml(net, actors = top_actors, layers = "facebook"),
+   leisure = degree_ml(net, actors = top_actors, layers = "leisure"),
+   lunch = degree_ml(net, actors = top_actors, layers = "lunch"),
+   coauthor = degree_ml(net, actors = top_actors, layers = "coauthor"),
+   work = degree_ml(net, actors = top_actors, layers = "work"),
+   flat = top_degrees)
```

```
   facebook leisure lunch coauthor work flat
1         12         1   15         NA   21   49
2         13         2   12         NA   20   47
3         14        14    7          3    8   46
4         11        NA    6         NA   27   44
```

5	15	7	13	NA	9	44
6	9	7	7	4	14	41

From the above result we can see how neighbors may not be equally distributed across the layers. Actor U4, for example, has the largest degree within the 6 actors analyzed in both the facebook layer and the flattened network. However, it has no presence in the coauthor layer and a very small degree in the leisure layer. If we want to quantify to what extent actors have similar or different degrees on the different (combinations of) layers, we can compute the standard deviation of the degree:

```
R> sort(degree_deviation_ml(net, actors = top_actors))
```

```
[1] 3.310589 4.261455 5.230679 7.418895 8.133880 9.987993
```

However, degree deviation should be used with care, because high variability may be due to differing densities in different layers.

### 4.3. Neighborhood and exclusive neighborhood

The neighbors of an actor  $a$  are those distinct actors that are adjacent to  $a$  on a specific input layer, or on a set of input layers. While on a single layer degree and neighborhood have the same value, they can be different when multiple layers are taken into account, because the same actors can be adjacent on multiple layers leading to a higher degree but not a higher neighborhood.

**Definition 4.1 (Neighbors)** Let  $M = (\mathcal{A}, \mathcal{L}, V, E)$  be a multiplex network. The neighbors of  $a \in \mathcal{A}$  on layers  $L \subseteq \mathcal{L}$  are defined as follows:

$$\text{neighbors}(a, L) = \{a' \in \mathcal{A} \mid (a, l, a', l) \in E \text{ and } l \in L\}$$

**Definition 4.2 (Neighborhood centrality)** Let  $M = (\mathcal{A}, \mathcal{L}, V, E)$  be a multiplex network. The neighborhood of  $a \in \mathcal{A}$  on layers  $L \subseteq \mathcal{L}$  is defined as follows:

$$\text{neighborhood}(a, L) = |\text{neighbors}(a, L)|$$

```
R> degree_ml(net, actors = "U4", layers = c("work", "lunch"))
```

```
[1] 36
```

```
R> neighborhood_ml(net, actors = "U4", layers = c("work", "lunch"))
```

```
[1] 21
```

The `xneighborhood_ml()` function (exclusive neighborhood) counts the neighbors that are adjacent to a specific actor only on the input layer(s) (Berlingerio, Coscia, Giannotti, Monreale, and Pedreschi 2012). A high exclusive neighborhood on a layer (or set of layers) means that the layer is important to preserve the connectivity of the actor: if the layer disappears, those neighbors would also disappear.

**Definition 4.3 (Exclusive neighborhood)** *Exclusive neighborhood is defined as:*

$$xneighborhood(a, L) = |neighbors(a, L) \setminus neighbors(a, \mathcal{L} \setminus L)|$$

where  $\setminus$  indicates the set difference operation.

```
R> neighborhood_ml(net, actors = "U91", layers = c("facebook", "leisure"))
```

```
[1] 22
```

```
R> xneighborhood_ml(net, actors = "U91", layers = c("facebook", "leisure"))
```

```
[1] 13
```

In this case, studying actor U191 within the multilayer network, we can see that out of 22 neighbors on `facebook` and `leisure`, 13 are exclusively present in these two layers, thus removing those layers will substantially impact the actor's connectivity.

#### 4.4. Relevance

Based on the concept of neighborhood, we can define a measure of layer relevance for actors (Berlingerio, Pinelli, and Calabrese 2013). `relevance_ml()` computes the ratio between the neighbors of an actor on a specific layer (or set of) and the total number of her neighbors. Every actor could be described as having a specific "signature" represented by her presence on the different layers.

**Definition 4.4 (Relevance)** *Let  $M = (\mathcal{A}, \mathcal{L}, V, E)$  be a multiplex network. Relevance is defined as follows:*

$$relevance(a, L) = \frac{neighborhood(a, L)}{neighborhood(a, \mathcal{L})}$$

```
R> data.frame(
+   facebook = relevance_ml(net, actors = "U123", layers = "facebook"),
+   leisure = relevance_ml(net, actors = "U123", layers = "leisure"),
+   lunch = relevance_ml(net, actors = "U123", layers = "lunch"),
+   coauthor = relevance_ml(net, actors = "U123", layers = "coauthor"),
+   work = relevance_ml(net, actors = "U123", layers = "work"))
```

```
   facebook leisure   lunch coauthor   work
1 0.3793103      NA 0.2068966      NA 0.9310345
```

Similarly to neighborhood also relevance can be extended using the concept of exclusive neighbor. The `xrelevance_ml()` function measures how much the connectivity of an actor (in terms of neighbors) would be affected by the removal of a specific layer (or set of layers).

**Definition 4.5 (Exclusive layer relevance)** *Let  $M = (\mathcal{A}, \mathcal{L}, V, E)$  be a multiplex network. Exclusive relevance is defined as follows:*

$$xrelevance(a, L) = \frac{xneighborhood(a, L)}{neighborhood(a, \mathcal{L})}$$

```
R> data.frame(
+   facebook = xrelevance_ml(net, actors = "U123", layers = "facebook"),
+   leisure = xrelevance_ml(net, actors = "U123", layers = "leisure"),
+   lunch = xrelevance_ml(net, actors = "U123", layers = "lunch"),
+   coauthor = xrelevance_ml(net, actors = "U123", layers = "coauthor"),
+   work = xrelevance_ml(net, actors = "U123", layers = "work"))

      facebook leisure lunch coauthor      work
1 0.06896552      NA     0      NA 0.5172414
```

Relevance and Exclusive relevance provide a simple way to estimate the relation between the actors and the multilayer structure. That can be useful for several reasons: one can use these functions to identify users who are extremely connected on a specific layer or combination of layers (e.g. users that are extremely active on Facebook) or to identify users that are connected uniquely through a layer or set of layers to part of their network. Relevance and Exclusive relevance can be fruitfully observed together. For example looking at the results visualized above for U123 one can observe how U123 has a presence on the network that is largely based on the `work` layer, containing 93% of the actor's neighbors with more than half being present only there.

#### 4.5. Distances

In addition to single-actor measures, the package can also be used to compute multilayer distances between pairs of actors. In a multiplex network, when a path passes from a vertex it can continue on any layer where the corresponding actor is present. As a consequence a path can traverse multiple layers.

**Definition 4.6 (Multiplex path length)** *The multiplex length of a path  $p$  on layers  $\mathcal{L} = \{l_1, \dots, l_m\}$  is a vector  $pl$  where  $pl[i]$  indicates the number of edges traversed in layer  $l_i$ .*

Distances are defined by [Magnani and Rossi \(2013b\)](#) as sets of lengths of Pareto-optimal multidimensional paths. As an example, if two actors are adjacent on two layers, both edges would qualify as Pareto-optimal paths from one actor to the other, as edges on different layers are considered incomparable (that is, it is assumed that it makes no sense in general to claim that two adjacent vertices on Facebook are closer or further than two adjacent vertices on the co-author layer). Pareto-optimal paths can also span multiple layers.

**Definition 4.7 (shorter-than relation)** *Let  $pl_1$  and  $pl_2$  be two multilayer path lengths.  $pl_1$  is shorter than  $pl_2$  iff  $\forall i \ pl_1[i] \leq pl_2[i] \wedge \exists i \ pl_1[i] < pl_2[i]$ .*

The distance between two actors  $a_1$  and  $a_2$  is the set of lengths of the shortest paths between them.

```
R> distance_ml(net, "U91", "U4")

      from to lunch coauthor leisure work facebook
1   U91 U4     2         0         0         0
```

2	U91	U4	1	0	1	0	0
3	U91	U4	0	1	2	0	0
4	U91	U4	0	0	1	1	0
5	U91	U4	1	0	0	1	0
6	U91	U4	0	0	0	2	0
7	U91	U4	0	0	0	0	1
8	U91	U4	0	0	3	0	0
9	U91	U4	0	2	0	1	0
10	U91	U4	1	2	0	0	0

This concept of distance allows us to provide a more qualitative description of the relation between two actors than what we would get using a single number. For example, we can see whether two actors are closer on some layers than others, and we can also see if specific combinations of layers lead to shorter paths than those on single layers, indicating how switching layer (for example discussing a work-related issue over lunch) may have an impact on phenomena such as the spreading of information in the network.

## 5. Community detection

A common network mining task is the identification of communities. An imprecise but generally accepted definition of community is as a subgroup of actors who are more densely connected among themselves than with the rest of the network.

The function `glouvain_ml()` uses the algorithm described by [Mucha, Richardson, Macon, Porter, and Onnela \(2010\)](#) to find community structures across layers, where vertices in different layers can belong to the same or a different community despite corresponding to the same actor. This method belongs to the class of community detection methods based on modularity optimization, that is, it tries to find an assignment of the vertices to communities so that the corresponding value of modularity is as high as possible. Multilayer modularity is a quality function that is high if most of the edges are between vertices in the same community and if vertices corresponding to the same actors are also often in the same community. Modularity as used in the package is defined as:

$$Q_m = \frac{1}{2\mu} \sum_{ijsr} [(a_{ijs} - \frac{k_{is}k_{js}}{2m_s})\delta(s,r) + \omega\delta(i,j)]\delta(\gamma_{i,s}, \gamma_{j,r}) \quad (1)$$

where  $i, j$  are actors,  $s, r$  are layers,  $a_{ijs}$  is 1 if  $i, j$  are adjacent on layer  $s$ ,  $k_{is}$  is the degree of actor  $i$  on layer  $s$ ,  $\mu$  is the number of pairs of vertices either adjacent on a layer or corresponding to the same actor,  $\delta$  is the Kronecker delta.

The function `glouvain_ml()` accepts one main parameter: `omega`, corresponding to  $\omega$  in the formula of modularity.

```
R> ml_clust <- glouvain_ml(net)
R> head(ml_clust)
```

```
  actor  layer cid
1  U19  lunch  0
```

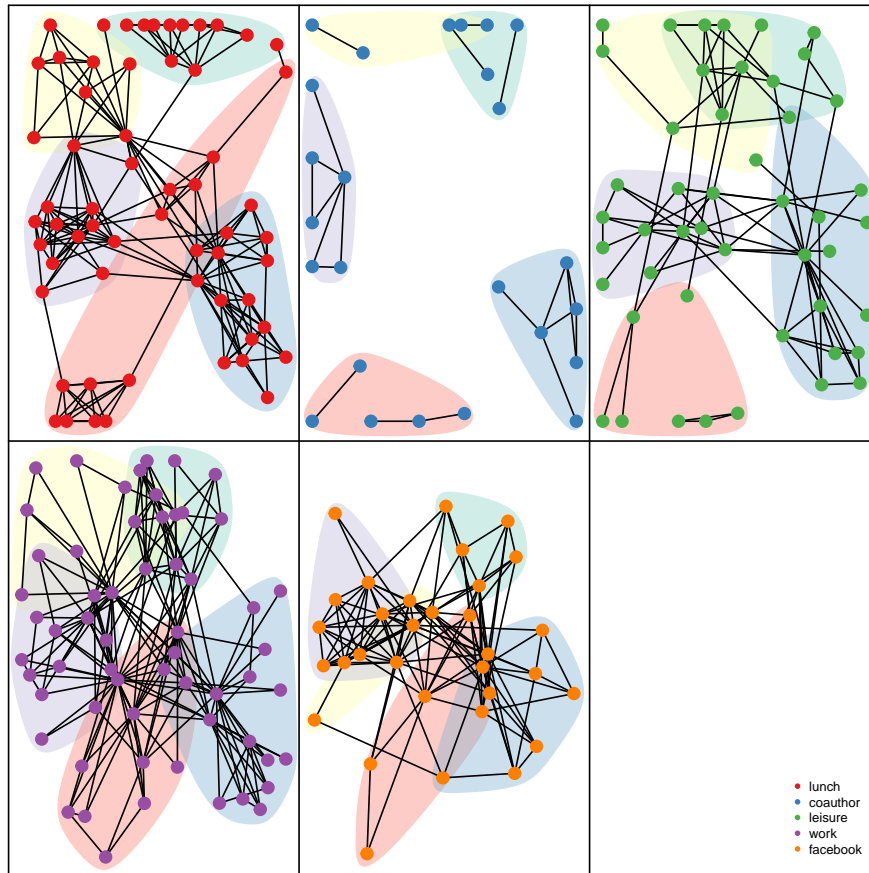


Figure 8: Multilayer representation of communities in the AUCS network detected using the generalized Louvain (`glouvain`) method.

```

2 U19 leisure 0
3 U19 work 0
4 U73 lunch 0
5 U73 leisure 0
6 U73 work 0

```

The result of the function is a data frame with two columns identifying a vertex, as a pair (actor, layer), and a third column with a numeric value (`cid`) identifying the community to which the vertex belongs. The corresponding communities are visualized in Figure 8, where we can see how five main groups of actors are identified (roughly corresponding to the research groups at the department) and how connectivity on all layers is partially explainable by this grouping.

The availability of community detection algorithms allows us to study their behaviours varying their parameters. For example, we can study the impact of the parameter `omega`. Higher values will result in communities spanning multiple layers, because including the same actor on different layers in the same community increases the value of modularity. With `omega` set



to 0, having the same actors on different layers in the same community does not contribute to modularity. In the following we can observe the impact of  $\omega$  on the number of communities and the number of layers spanned by each community (code to count communities and layers not shown):

```
R> clus <- glouvain_ml(net, omega=0)
R> num_communities(clus)
```

```
[1] 27
```

```
R> avg_layers_per_community(clus)
```

```
[1] 1
```

```
R> clus <- glouvain_ml(net, omega=0.01)
R> num_communities(clus)
```

```
[1] 7
```

```
R> avg_layers_per_community(clus)
```

```
[1] 3.857143
```

```
R> clus <- glouvain_ml(net, omega=1)
R> num_communities(clus)
```

```
[1] 5
```

```
R> avg_layers_per_community(clus)
```

```
[1] 5
```

The package provides other community detection algorithms: multilayer clique percolation (ML-CPM) ([Afsarmanesh and Magnani 2018](#)), ABACUS ([Berlingerio \*et al.\* 2013](#)) (for overlapping and partial community detection) and Infomap ([De Domenico, Lancichinetti, Arenas, and Rosvall 2015](#)) (for partitioning/overlapping community detection on undirected or directed networks):

```
R> c1 <- abacus_ml(net, 4, 2)
R> c2 <- clique_percolation_ml(net, 4, 2)
R> c3 <- glouvain_ml(net)
R> c4 <- infomap_ml(net)
```

We can now compare these community detection methods by computing some statistics about (1) the number of communities generated, (2) the average community size, (3) the percentage of vertices included in at least one cluster (which is 1 for complete community detection methods), (4) the percentage of actors included in at least one cluster (which is 1 for complete community detection methods) and (5) the ratio between the number of actor-community pairs and the number of clustered actors, indicating the level of overlapping (which is 1 for partitioning community detection methods and higher for overlapping methods). The corresponding statistics for the AUCS network are the following (code to build the dataframe not shown):

```
R> com_stats_auc
```

	num	avg_s	clust_vertices	clust_actors	actor_overl
abacus	22	6.318182	0.5446429	0.6557377	3.47500
clique p.	11	5.181818	0.3750000	0.5081967	1.83871
louvain	5	12.200000	1.0000000	1.0000000	1.00000
infomap	5	12.200000	1.0000000	1.0000000	1.00000

The same comparison can be performed on some of the other datasets included in the library, for example the bank wiring network (code to build the dataframe not shown):

```
R> com_stats_bank
```

	num	avg_s	clust_vertices	clust_actors	actor_overl
abacus	6	5	0.4677419	0.5714286	3.125
clique p.	2	4	0.2580645	0.5714286	1.000
louvain	2	7	1.0000000	1.0000000	1.000
infomap	1	14	1.0000000	1.0000000	1.000

The interpretation of these results is largely dependent on the interpretation of the result obtained by the specific community detection method. All the methods that are implemented in the package are based on pre-existing methods to detect community on single-layer networks. The ideal definition of community in a single-layer context, that is different for each method, was then extended, following different assumptions, into a multilayer context. For this reason, while a simple and straightforward interpretation of the results is impossible, it is surely possible to observe how different methods (thus different underlying definitions of community) return very different results. An in-depth analysis, description and comparison of these algorithms is available in (Hanteer, Interdonato, Magnani, Tagarelli, and Rossi 2019).

## 6. Conclusion

In this article we have presented the **multinet** package and some of its functions to create and analyze multiplex networks. The package provides a wide range of network analysis methods to analyze individual actors, identify groups (communities) and compare layers, in addition to functions to explore and generate network data. **multinet** is also integrated with **igraph**,

so that single layers or flattened sets of layers can also be analyzed using more traditional methods.

Future developments of the package that would extend its usability include: more support for weighted and directed networks, that are not considered by all the algorithms, the addition of more network simplification/preprocessing methods (Interdonato, Magnani, Perna, Tagarelli, and Vega 2020), such as sampling, the integration with other network libraries, the porting to other programming languages and the addition of functions generating multiplex and multilayer networks from other types of data, such as data extracted using social media platforms' APIs.

## Acknowledgments

We thank Mikael Dubik for the implementation of the generalized Louvain method, and several people who participated in our training workshops or contacted us to suggest features and report bugs. The **multinet** package includes the following external code: **eclat**<sup>7</sup> (for association rule mining), **Eigen**<sup>8</sup> and **spectra**<sup>9</sup> (for matrix manipulation), **Infomap**<sup>10</sup> (for the Infomap community detection method) and Howard Hinnant's date and time library<sup>11</sup>.

This work was partially supported by the European Community through the project "Values and ethics in Innovation for Responsible Technology in Europe" (Virt-EU) funded under Horizon 2020 ICT-35-RIA call Enabling Responsible ICT-related Research and Innovation.

## References

- Afsarmanesh N, Magnani M (2018). "Partial and Overlapping Community Detection in Multiplex Social Networks." In *Social Informatics*.
- Barabási AL, Albert R (1999). "Emergence of Scaling in Random Networks." *Science*, **286**(5439), 509–512. ISSN 0036-8075. doi:10.1126/science.286.5439.509. <http://science.sciencemag.org/content/286/5439/509.full.pdf>, URL <http://science.sciencemag.org/content/286/5439/509>.
- Berlingerio M, Coscia M, Giannotti F, Monreale A, Pedreschi D (2012). "Multidimensional Networks: Foundations of Structural Analysis." *World Wide Web*. ISSN 1386-145X. doi:10.1007/s11280-012-0190-4. URL <http://link.springer.com/10.1007/s11280-012-0190-4>.
- Berlingerio M, Pinelli F, Calabrese F (2013). "ABACUS: Apriori-Based Community Discovery in Multidimensional Networks." *Data Mining and Knowledge Discovery*, **27**.
- Brodka P, Chmiel A, Magnani M, Ragozini G (2018). "Quantifying Layer Similarity in Multiplex Networks: A Systematic Study." *Royal Society open science*, **5**(8).

<sup>7</sup><http://www.borgelt.net/eclat.html>

<sup>8</sup><http://eigen.tuxfamily.org>

<sup>9</sup><https://spectralib.org>

<sup>10</sup><http://www.mapequation.org>

<sup>11</sup><https://github.com/HowardHinnant/date>

- Butts CT (2008). “**network**: A Package for Managing Relational Data in R.” *Journal of Statistical Software*, **24**(2). URL <http://www.jstatsoft.org/v24/i02/paper>.
- Butts CT (2015). **network**: *Classes for Relational Data*. The Statnet Project (<http://statnet.org>). R package version 1.13.0.1, URL <http://CRAN.R-project.org/package=network>.
- Butts CT (2016). **sna**: *Tools for Social Network Analysis*. R package version 2.4, URL <https://CRAN.R-project.org/package=sna>.
- Csardi G, Nepusz T (2006). “The **igraph** Software Package for Complex Network Research.” *InterJournal, Complex Systems*, 1695. URL <http://igraph.org>.
- De Domenico M, Lancichinetti A, Arenas A, Rosvall M (2015). “Identifying Modular Flows on Multilayer Networks Reveals Highly Overlapping Organization in Interconnected Systems.” *Physical Review X*, **5**.
- Dickison ME, Magnani M, Rossi L (2016). *Multilayer Social Networks*. Cambridge University Press. ISBN 978-1107438750.
- Eddelbuettel D, François R (2011). “**Rcpp**: Seamless R and C++ Integration.” *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08. URL <http://www.jstatsoft.org/v40/i08/>.
- Erdos P, Rényi A (1960). “On the Evolution of Random Graphs.” *Publ. Math. Inst. Hung. Acad. Sci.*, **5**(1), 17–60.
- Fatemi Z, Magnani M, Salehi M (2018). “A Generalized Force-Directed Layout for Multiplex Sociograms.” In *Social Informatics*.
- Handcock MS, Hunter DR, Butts CT, Goodreau SM, Morris M (2003). **statnet**: *Software Tools for the Statistical Modeling of Network Data*. Seattle, WA. URL <http://statnetproject.org>.
- Hanteer O, Interdonato R, Magnani M, Tagarelli A, Rossi L (2019). “Community Detection in Multiplex Networks.” *ArXiv*, **abs/1910.07646**.
- Hunter DR, Handcock MS, Butts CT, Goodreau SM, Morris M (2008). “**ergm**: A Package to Fit, Simulate and Diagnose Exponential-Family Models for Networks.” *Journal of Statistical Software*, **24**(3), 1–29.
- Interdonato R, Magnani M, Perna D, Tagarelli A, Vega D (2020). “Multilayer network simplification: approaches, models and methods.” *Computer Science Review (to appear)*.
- Kivelä M, Arenas A, Barthelemy M, Gleeson JP, Moreno Y, Porter MA (2014). “Multi-layer Networks.” *Journal of Complex Networks*, **2**(3), 203–271. doi:doi:10.1093/comnet/cnu016.
- Magnani M, Rossi L (2011). “The ML-Model for Multi-layer Social Networks.” In *ASONAM*, pp. 5–12. IEEE Computer Society. ISBN 9781612847580.

Magnani M, Rossi L (2013a). “Formation of Multiple Networks.” In *Social Computing, Behavioral-Cultural Modeling and Prediction*, pp. 257–264. Springer-Verlag Berlin Heidelberg. ISBN 978-3-642-37209-4.

Magnani M, Rossi L (2013b). “Pareto Distance for Multi-layer Network Analysis.” In AM Greenberg, WG Kennedy, ND Bos (eds.), *Social Computing, Behavioral-Cultural Modeling and Prediction*, volume 7812 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg. ISBN 978-3-642-37209-4. doi:10.1007/978-3-642-37210-0. URL <http://link.springer.com/10.1007/978-3-642-37210-0>.

Mucha PJ, Richardson T, Macon K, Porter MA, Onnela JP (2010). “Community Structure in Time-Dependent, Multiscale, and Multiplex Networks.” *Science*, **328**(5980), 876–878. ISSN 0036-8075. doi:10.1126/science.1184819. <http://science.sciencemag.org/content/328/5980/876.full.pdf>, URL <http://science.sciencemag.org/content/328/5980/876>.

Ostoic AR (2018). **multiplex**: *Algebraic Tools for the Analysis of Multiple Social Networks*. R package version 2.9, URL <https://CRAN.R-project.org/package=multiplex>.

Ripley RM, Snijders TAB, Bóda Z, Vörös A, Preciado P (2018). “Manual for **Siena** version 4.0.” *Technical report*, Oxford: University of Oxford, Department of Statistics; Nuffield College. R package version 1.2-12., URL <https://www.cran.r-project.org/web/packages/RSiena/>.

**Affiliation:**

Matteo Magnani  
InfoLab  
Department of Information Technology  
Uppsala University  
Sweden  
E-mail: [matteo.magnani@it.uu.se](mailto:matteo.magnani@it.uu.se)

*and*

Luca Rossi  
DssLab  
IT University of Copenhagen  
Denmark  
E-mail: [lucr@itu.dk](mailto:lucr@itu.dk)

*and*

Davide Vega  
InfoLab  
Department of Information Technology  
Uppsala University  
Sweden  
E-mail: [davide.vega@it.uu.se](mailto:davide.vega@it.uu.se)