Gábor Csárdi
Tamás Nepusz
Edoardo M. Airoldi

Statistical Network Analysis with igraph

April 25, 2016

Springer

Contents

Chapter 1
Igraph

1.1 About igraph

For the purposes of this book, igraph is an extension package for R. It is a collection or R functions to explore, create, manipulate and visualize graphs. Originally most of igraph was optimized for running time, to perform all of its computations as quickly as possible. While this is still an important goal, in recent years the development shifted and we tried to make it easier to use. It often hard to meet these goals together, so we were (and still are) trying to find a good balance between them.

Igraph is open source software. All of its source code is available online at the igraph homepage: `http://igraph.org`. It also includes and builds on several other open source packages, without which it would not have been possible to create it.

All the program code in this book is available at the igraph homepage, and also in the *`igraphbook`* R package. The reader can simply load this package into R and follow the code interactively while reading the book.

The igraph homepage is at `http://igraph.org`. It contains detailed documentation about each igraph function, tutorials and demos. It is a great place to keep track of what's going on around igraph.

1.1.1 About the code in this book

This book uses igraph version 1.1.0 and R version 3.2.3. Because of the dynamically changing nature of both R and igraph, it is inevitable that some of the program code from the book will not work with future igraph and/or R versions. We will keep the program code in the *`igraphbook`* package, and on the igraph homage up to date, and make sure that they work with the latest R and igraph versions.

In the book, blocks of text are interweaved with chunks of program code. All code chunks are numbered, within chapters. The same numbering will be kept in future versions of the *igraphbook* R package and on the homepage. While the individual code chunks build on the results of previous chunks, each chapter is self-contained. To run a code chunk in Chapter 10, the reader probably needs to also run all previous code chunks in Chapter 10, but nothing more. This can be done easily with the 'igraphbook' R package, that can jump to a code chunk, and run the required other chunks automatically.

For much of the code in the package, we will use the *magrittr* R package. This package defines the %>% forward piping operator. This is similar to the classic Unix pipes, and simply applies a sequence of operators on a data set (most often a graph in this book), one after the other. The input of an operation is the output of the previous operation. *magrittr* pipes result much more readable program code, eliminating many temporary variables, and deeply nested function calls. An example will probably make all this clear. We create a small graph, add some labels to it, place its parts in a visually pleasing way, and finally plot it. Here is the tradidional syntax with temporary variables:

```
g <- make_graph('Diamond')
g2 <- set_vertex_attr(g, 'name', value = LETTERS[1:4])
g3 <- set_graph_attr(g2, 'layout', layout_with_fr)
plot(g3)
```

The same with nested function calls:

```
plot(
  g <- set_graph_attr(
    set_vertex_attr(
      make_graph('Diamond'),
      'name',
      value = LETTERS[1:4]),
    'layout',
    layout_with_fr)
  )
```

With the pipe operator this can be written as

```
g <- make_graph('Diamond') %>%
  set_vertex_attr('name', value = LETTERS[1:4]) %>%
  add_layout_(with_fr()) %>%
  plot()
```

The pipe operator eliminates the need of temporary variables that usually keep lying around, without making the code much harder to read with nested function calls.

1.1.2 Data sets

The data sets that appear in this book, are part of the *igraphdata* R package. (The *igraphbook* package automatically installs *igraphdata* as well.)

Here is the current list of data sets in the *igraphdata* package:

```
1.4.1) library(igraphdata)
   2) data(package="igraphdata")
```

```
Data sets in package 'igraphdata':

Koenigsberg       Bridges of Koenigsberg from Euler's
                  times
UKfaculty         Friendship network of a UK
                  university faculty
USairports        US airport network, 2010 December
enron             Enron Email Network
foodwebs          A collection of food webs
immuno            Immunoglobulin interaction network
karate            Zachary's karate club network
kite              Krackhardt's kite
macaque           Visuotactile brain areas and
                  connections
rfid              Hospital encounter network data
yeast             Yeast protein interaction network
```

The first graph we will use is the 'macaque' data set. It is a directed graph containing the anatomical connections between the brain areas of the macaque monkey. A graph is stored in an igraph object, i.e. `'macaque'` is an igraph object in *igraphdata*.

```
1.5.1) library(igraph)
   2) library(igraphdata)
   3) data(macaque)
```

1.2 Exploring graphs

In this section we introduce the most basic graph theoretical concepts, and illustrate them with igraph on two data sets.

1.2.1 Igraph graphs

In igraph graphs are special objects, with a given internal representation and a set of R functions (the igraph application programming interface, API), to perform various operations on them: calculate properties, manipulate the graph structure, visualize the graphs, etc. This book deals with the API, and not with the internal representation.

The *igraphdata* package contains graph data sets as igraph objects, the 'macaque' name refers to one of them. If the name of an igraph object is given at the R prompt, then a short summart of the graph is printed.

1.6.1) `macaque`

```
IGRAPH f7130f3 DN-- 45 463 --
+ attr: Citation (g/c), Author (g/c), shape (v/c), name
| (v/c)
+ edges from f7130f3 (vertex names):
 [1] V1->V2     V1->V3     V1->V3A    V1->V4     V1->V4t
 [6] V1->MT     V1->PO     V1->PIP    V2->V1     V2->V3
[11] V2->V3A    V2->V4     V2->V4t    V2->VOT    V2->VP
[16] V2->MT     V2->MSTd/p V2->MSTl   V2->PO     V2->PIP
[21] V2->VIP    V2->FST    V2->FEF    V3->V1     V3->V2
[26] V3->V3A    V3->V4     V3->V4t    V3->MT     V3->MSTd/p
[31] V3->PO     V3->LIP    V3->PIP    V3->VIP    V3->FST
+ ... omitted several edges
```

This is the standard way of showing (printing) an igraph graph object on the screen. The top line of the output declares that the object is an igraph graph, and also lists its most important properties. A four-character long code is printed first:

'D/U' The first character is either 'D' or 'U' and encodes whether the graph is directed or undireted.

'N' The second letter is 'N' for named graphs (see Section 1.2.5). A dash here means that the graph is not named.

'W' The third letter is 'W' if the graph is weighted (in other words, if the graph is a valued graph, Section 2.4). Unweighted graphs have a dash in this position.

'B' Finally, the fourth is 'B' if the graph is bipartite (two-mode, Section **??**). For unipartite (one-mode) graphs a dash is printed here.

This notation might seem quite dense at first, but it is easy to get used to and conveys much information in a small space. Then two numbers are printed, these are the number of vertices and the number of edges in the graph, 45 and 463 in our case. At the end of the line the name of the graph is printed, if there is any. The next line(s) list attributes, meta-data that belong to the vertices, edges or the graph itself. Finally, the edges of the graph are listed. Except for very small graphs, this list is truncated, so that it fits to the screen.

1.2.2 The igraph data model

The number of vertices in a graph is the *order* of the graph, see `gorder()`, the number of edges is the *size* of the graph, see `gsize()`. In the text of this book we will denote the order of the graph by $|V|$ and the size of the graph by $|E|$.

order
gorder()
size
gsize()

1.7.1)
```
gorder(macaque)
```
```
[1] 45
```
3)
```
gsize(macaque)
```
```
[1] 463
```

Although the *graph* seems to be a straightforward concept, different scientific fields have different definitions. It is important that we make sure what exactly a graph means throughout the rest of this book. Here are the rules.

A *directed graph* $G = (V, E)$ is a multiset of ordered pairs over a finite set of vertices V. For simplicity we usually denote a set of n vertices by the numbers $1, 2, \ldots n$. An *ordered pair* of vertices is $(i, j) \in E$, which is different from the ordered pair (j, i), unless $i = j$. In a *multiset*, every element has a multiplicity, so a directed graph may contain the same ordered pair more than once.

An *undirected graph* is a multiset of unordered pairs and singletons over a finite set of vertices. An unordered pair means the set $\{i, j\}$ if $i \neq j$, and a singleton is simply $\{i\}$. A singleton defines an undirected loop, an undirected edge from a vertex to itself.

Some important consequences of these definitions:

1. An igraph graph is either directed or undirected. Igraph does not support mixed graphs that have both directed and undirected edges, although they can be simulated with attributes. Note that a directed graph with only mutual edges is different from an undirected graph.
2. An igraph graph may have multiple edges between the same pair of vertices. (In social network terminology, it is a multi-graph.)
3. An igraph graph can have loops, i.e. (directed or undirected) edges that are incident to only one vertex.
4. An igraph graph is binary, an edge is a relationsship between *two* vertices (for loop edges these two vertices are the same). Hypergraphs are not supported.

1.2.3 Vertices and edges

As in standard graph theory notation, the set of vertices in a graph is queried with `V()`, and the set of edges is with `E()`. Just like for graphs, igraph will not

V()

print more vertices or edges than what fits on the screen, so for larger graphs, some vertices or edges will be omitted from the screen output. Of course `V()` and `E()` refer to all vertices and edges, including the ones not printed.

1.9.1) `V(macaque)`

```
+ 45/45 vertices, named, from f7130f3:
 [1] V1     V2     V3     V3A    V4     V4t    VOT    VP
 [9] MT     MSTd/p MSTl   PO     LIP    PIP    VIP    DP
[17] 7a     FST    PITd   PITv   CITd   CITv   AITd   AITv
[25] STPp   STPa   TF     TH     FEF    46     3a     3b
[33] 1      2      5      Ri     SII    7b     4      6
[41] SMA    Ig     Id     35     36
```

9) `E(macaque)`

```
+ 463/463 edges from f7130f3 (vertex names):
 [1] V1 ->V2     V1 ->V3     V1 ->V3A    V1 ->V4
 [5] V1 ->V4t    V1 ->MT     V1 ->PO     V1 ->PIP
 [9] V2 ->V1     V2 ->V3     V2 ->V3A    V2 ->V4
[13] V2 ->V4t    V2 ->VOT    V2 ->VP     V2 ->MT
[17] V2 ->MSTd/p V2 ->MSTl   V2 ->PO     V2 ->PIP
[21] V2 ->VIP    V2 ->FST    V2 ->FEF    V3 ->V1
[25] V3 ->V2     V3 ->V3A    V3 ->V4     V3 ->V4t
[29] V3 ->MT     V3 ->MSTd/p V3 ->PO     V3 ->LIP
[33] V3 ->PIP    V3 ->VIP    V3 ->FST    V3 ->TF
[37] V3 ->FEF    V3A->V1     V3A->V2     V3A->V3
+ ... omitted several edges
```

In graph theory, the vertices of a graph are often denoted by the natural numbers $1, 2, \ldots, |V|$, and igraph supports this notation, both for vertices and for edges: $1, 2, \ldots, |E|$.

However, graph data sets often have natural vertex identifiers, e.g. names of people, URLs of web pages, etc. In this case it is easier to refer to vertices using the natural identifiers, or as they are called in igraph, symbolic vertex names. In case of the macaque graph, the vertex identifiers are names of brain areas, and whenever you refer to a vertex in this graph, you can simply use the name of the brain area, in single or double quotes.

Similarly, edges can be referred to via the names of the vertices they connect, and a vertical bar between them: `'V1|V2'` means the directed edge from vertex `'V1'` to vertces `'V2'`. In undirected graphs, the order of the vertices are ignored and `'X|Y'` refers to the same edge as `'Y|X'`. Note that this notation might be ambiguous if the graph has multiple edges between the same pair of vertices.

incident

We say that an edge is *incident* to the vertices it connects, and that these vertices are also incident to the edge. The two incident vertices of an edge are called the *end*s (see `ends()`) of the edge. For a directed edge, the first vertex

end
ends()

of the ordered pair is the *tail* (see `tail_of()`), the second vertex is the *head* (see `head_of()`) of the edge. (For directed loops the head and the tail are the same.) Two vertices that are connected by an edge are *adjacent*, or *neighbors*. See `neighbors()`. In directed graphs, a vertex has a potentially different set of out-neighbors and in-neighbors, depending on whether the vertex is the tail or the head of the common edge:

tail
tail_of()
head
head_of()
adjacent
neighbors
neighbors()

```
1.11.1) macaque %>% ends('V1|V2')
```

```
     [,1] [,2]
[1,] "V1" "V2"
```

```
4) macaque %>% tail_of('V1|V2')
```

```
+ 1/45 vertex, named, from f7130f3:
[1] V1
```

```
7) macaque %>% head_of('V1|V2')
```

```
+ 1/45 vertex, named, from f7130f3:
[1] V2
```

```
10) macaque %>% neighbors('PIP', mode = "out")
```

```
+ 8/45 vertices, named, from f7130f3:
[1] V1 V3 V4 VP MT PO DP 7a
```

```
13) macaque %>% neighbors('PIP', mode = "in")
```

```
+ 8/45 vertices, named, from f7130f3:
[1] V1 V2 V3 V4 VP MT PO DP
```

A directed *path* is a sequence of zero or more edges, such that the tail of an edge is the head of the previous edge. If the graph does not have multiple edges, then a path can also be given by a sequence of vertices, starting with the tail of the first edge, and ending at the head of the last edge.

path

Igraph can select a sequence of edges or vertices that belong to a path. For vertices, simply list them in the index of `V()`:

```
1.16.1) V(macaque)[c('V1', 'V2', 'V3A', 'V4')]
```

```
+ 4/45 vertices, named, from f7130f3:
[1] V1  V2  V3A V4
```

To select the edges of a path, you can similarly list the edges, or use a shorter form. The shorter form has the additional benefit that igraph checks that the given vertices indeed form a path.

```
1.17.1) E(macaque)[c('V1|V2', 'V2|V3A', 'V3A|V4')]
```

```
+ 3/463 edges from f7130f3 (vertex names):
[1] V1 ->V2  V2 ->V3A V3A->V4
```

```
4) E(macaque, path = c('V1', 'V2', 'V3A', 'V4'))
```

```
+ 3/463 edges from f7130f3 (vertex names):
[1] V1 ->V2  V2 ->V3A V3A->V4
```

1.2.4 Subgraphs and components

subgraph

We say that a $G_2 = (V_2, E_2)$ graph is a *subgraph* of $G_1 = (V_1, E_1)$, if $V_2 \subseteq V_1$ and $E_2 \subseteq E_1$. G_2 is an induced subgraph of G_1 if it is a subgraph, and for every edge $(i, j) \in V_1$, it also holds that $(i, j) \in V_2$, and $i, j \in E_2$. In other words, on its restricted vertex set, G_2 contains exactly the edges that G_1 contains.

subgraph() *induced_subgraph()*

`subgraph()` creates a subgraph from a graph, and `induced_subgraph()` creates an induced subgraph. For example, an indiced subgraph consisting of `'V1'` and `V2`, and all their neighbor vertices can be created as:

```
1.19.1) V(macaque)['V1', 'V2', nei('V1'), nei('V2')] %>%
2)   induced_subgraph(graph = macaque) %>%
3)   summary()
```

```
Warning: 'nei' is deprecated.
Use '.nei' instead.
See help("Deprecated")
```

```
Warning: 'nei' is deprecated.
Use '.nei' instead.
See help("Deprecated")
```

```
IGRAPH 63f47db DN-- 16 156 --
+ attr: Citation (g/c), Author (g/c), shape (v/c), name
| (v/c)
```

connected

An undirected graph is called *connected* if there is a path from every vertex to every other vertex. A directed graph is *weakly* connected if its underlying undirected graph is connected. In other words, in contains an undirected path from each vertex to every other vertex. A directed graph is *strongly* connected if there is a directed path from each vertex to every other vertex. The `is_connected()` function decides if a graph is (weakly or strongly for directed graphs) connected.

is_connected()

```
1.20.1) is_connected(macaque, mode = "weak")
```

```
[1] TRUE
```

```
3) is_connected(macaque, mode = "strong")
```

```
[1] TRUE
```

components

A graph that is not (weakly or strongly) connected consists of multiple *components* (also called connected components). A component is a maximal connected subgraph. Maximal means that you cannot add other vertices to the subgraph and keep it connected. A directed graph has weakly and strongly connected components, corresponding to weak and strong connectedness.

1.2.5 Vertex and edge sequences

Internal, numeric ids

The graph we have been dealing with so far had natural vertex identifiers: the names of brain areas in tha macaque graph.

This is not always the case. If a graph does not have symbolic vertex ids, igraph uses natural numbers between one and $|V|$ to identify vertices. Note that even if a graph has symbolic ids, the user can still use the (internal) numeric ids. `'V(macaque)'` contains the vertices in the order of the numeric ids, so the following two lines are equivalent:

```
1.22.1) V(macaque)[1:4]
```

```
+ 4/45 vertices, named, from f7130f3:
[1] V1  V2  V3  V3A
```

```
   4) V(macaque)[c('V1', 'V2', 'V3', 'V3A')]
```

```
+ 4/45 vertices, named, from f7130f3:
[1] V1  V2  V3  V3A
```

Using the internal ids in inconvenient if the structure of the graph changes, and one intends to follow individual vertices or edges. This is because it is impossible to have a hole in the numbering of the vertices or the edges; if a graph has 4 vertices and 5 edges, then the vertex ids are always 1, 2, 3 and 4, and the edge ids are always 1, 2, 3, 4 and 5.

Because the numeric ids have to be consecutive, igraph reassigns them when a smaller graph (e.g. a subgraph) is created. This means that brain area *V4* does not necessarily have the same numeric ids in both graphs here:

```
1.24.1) sub_macaque <- induced_subgraph(macaque, c('V1', 'V2', 'V4', 'MT'))
   2) V(macaque)[1:3]
```

```
+ 3/45 vertices, named, from f7130f3:
[1] V1 V2 V3
```

```
   5) V(sub_macaque)[1:3]
```

```
+ 3/4 vertices, named, from 451915d:
[1] V1 V2 V4
```

We suggest that the user uses vertex names, these are kept across graphs. Vertex names can be character strings, and while arbitrary characters can be used, it is best if one restricts oneself to alphanumeric characters if portability is a concern. Symbolic vertex names seem so handy that it is a valid question why don't all igraph graphs have them. The reason is that they introduce an overhead in terms of storage space and processing time. This overhead is negligible if one works with graphs of moderate size, but important for graphs with millions of vertices.

Vertex and edge sequences

vertex sequence
edge sequence

Much of graph exploration and manipulation means performing operations on sequences and sets of vertices and edges. Igraph has data types and functions to do this. A *vertex sequence* is simply a vector of vertex ids and an *edge sequence* is a vector of edge ids. Whenever igraph functions expect a sequence or set of vertices, a vertex sequence should be given, similarly, for functions operating on edges, an edge sequence should be given. Similarly, many igraph functions return vertex and/or edge sequences as their results.

Vertex sequences are created using the `V()` function, which we have seen already. `'V(kite)'` means the sequence of all vertices in the `'kite'` graph, in the order of their internal numeric ids. Vertex sequences can be indexed like regular R vectors, and they also have some additional operations defined. Table 1.1 summarizes them.

Edge sequences are created with `E()`, and have similar operations, see Table 1.2.

Vertex and edge sequences interact nicely with vertex and edge attributes, see more about this in Sec. 1.2.7.

Reference to the graph.

A vertex sequence knows which graph it was created from, and it can be only used with this graph. This is important to keep this in mind when working with vertices from multiple graphs. In particular, to combine vertices from different graphs, you need to convert them to vertex names or ids with `as_ids()`. The same applies to edge sequences. This does not work:

1.26.1)
```
c(V(macaque)[11:20], V(sub_macaque))
```

```
Warning in parse_op_args(..., what = "a vertex", is_fun = is_igraph_vs, : Combining vertex/edg
This will not work in future igraph versions
```

```
Warning in parse_op_args(..., what = "a vertex", is_fun = is_igraph_vs, : Combining vertex/edg
This will not work in future igraph versions
```

`V(kite)`	Select all vertices in the order of their numeric id.
`V(kite)[1:3, 7:10]`	Select vertices in the given positions.
`V(kite)[degree(kite) < 2]`	Select vertices that satisfy a condition.
`V(kite)[nei('D')]`	Select vertices that are neighbors of a given vertex.
`V(macaque)[innei('V1')]`	Like `nei()`, but for incoming edges only.
`V(macaque)[outnei('V1')]`	Like `nei()`, but for outgoing edges only.
`V(kite)[inc('A|D')]`	Select vertices that are incident to a given edge.
`V(kite)['A', 'B', 'D']`	Select vertices with the given names.
`c(V(kite)['A'], V(kite)['D'])`	Concatenate two or more vertex sequences.
`rev(V(kite))`	Reverse a vertex sequence.
`unique(V(kite)['A', 'A'])`	Remove duplicates from a vertex sequence.
`union(V(kite)[1:5], V(kite)[6:10])`	Union of vertex sequences. (Set operation.)
`intersection(V(kite)[1:7], V(kite)[5:10])`	Intersection of vertex sequences. (Set operation.)
`difference(V(kite), V(kite)[1:5])`	Difference of vertex sequences. (Set operation.)

Table 1.1 Operations on vertex sequences.

```
+ 14/45 vertices, named, from f7130f3:
 [1] MSTl PO   LIP  PIP  VIP  DP   7a   FST  PITd PITv V1
[12] V2   V4   MT
```

But the following does:

1.27.1)
```
c(V(macaque)[11:20] %>% as_ids(), V(sub_macaque) %>% as_ids())
```

```
 [1] "MSTl" "PO"   "LIP"  "PIP"  "VIP"  "DP"   "7a"   "FST"
 [9] "PITd" "PITv" "V1"   "V2"   "V4"   "MT"
```

`as_ids()` creates a character vector of names for named graphs, and an integer vector of internal numeric ids for others.

`E(kite)`	Select all edges, in the order of their numeric id.
`E(kite, path = c('A', 'D', 'C'))`	Select edges along a path.
`E(kite)[ vs %--% vs2 ]`	Select all edges between the vertices of two vertex sequences.
`E(macaque)[ vs %->% vs2 ]`	The same, but consider edge directions.
`E(kite)[1:3, 7:10]`	Select edges in the given positions.
`E(kite)[seq_len(gsize(kite)) %% 2]`	Select edges that satisfy a condition.
`E(kite)[inc('D')]`	Select edges incident to a vertex.
`E(macaque)[from('V1')]`	Like `inc()`, but only if the vertex is the tail of the edge.
`E(macaque)[to('V1')]`	Like `inc()`, but only if the vertex is the head of the edge.
`E(kite)['A|B', 'B|C', 'D|A']`	Select edges with the given names.
`c(E(kite)['A|B'], E(kite)['D|A'])`	Concatenate two or more edge sequences.
`rev(E(kite))`	Reverse an edge sequence.
`unique(E(kite)[1:5, 1:10])`	Remove duplicates from an edge sequence.
`union(E(kite)[1:5], E(kite)[6:10])`	Union of edge sequences. (Set operation.)
`intersection(E(kite)[1:7], E(kite)[5:10])`	Intersection of edge sequences. (Set operation.)
`difference(E(kite), E(kite)[1:5])`	Difference of edge sequences. (Set operation.)

Table 1.2 Operations on edge sequences. `'vs'` and `'vs2'` are vertex sequences from the same graph.

1.2.6 Queries

Before we continue our tutorial on igraph, we need to take a small detour and learn about how graphs can be represented using adjacency matrices, or adjacency lists. While igraph does not use this representations, the various graph queries and manipulationsare easier to understand if one thinks about igraph graphs as adjacency matrices, or adjacency lists.

In the rest of this chapter we will also use a small graph known as Krackhardt's kite, see Fig. 1.1:

```
1.28.1) data(kite)
    2) kite
```

```
IGRAPH 6b7ddad UN-- 10 18 -- Krackhardt's kite
+ attr: name (g/c), layout (g/n), Citation (g/c),
| Author (g/c), URL (g/c), label (v/c), Firstname
```

```
| (v/c), name (v/c)
+ edges from 6b7ddad (vertex names):
 [1] A--B A--C A--D A--F B--D B--E B--G C--D C--F D--E D--F
[12] D--G E--G F--G F--H G--H H--I I--J
```

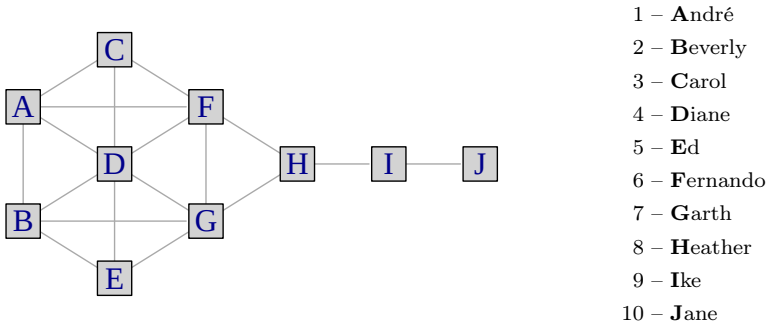


Fig. 1.1 Krackhardt's kite, a fictional social network of ten actors. The right hand side shows the mapping of the vertices to numeric vertex ids. The kite is an example of a graph in which the most central actor is different according to the three classic centrality measures. You will see more about centrality and this graph in Chapter **??**.

Adjacency matrices

adjacency matrix

A straightforward way of representing a graph is the adjacency matrix. This is a square matrix. The number of rows and the number of columns in the table equal the number of vertices in the graph. We assign integer number to vertices of the graph: $1, 2, \ldots$, and refer to the vertices using these numbers. The table element in row $\mathtt{i}$ and column $\mathtt{j}$ is equal to 1 if and only if there in an edge between vertices $\mathtt{i}$ and $\mathtt{j}$. Otherwise the table element is zero. You can see the adjacency matrix of the kite graph in Fig. **??**.

The adjacency matrix of an undirected graph is always symmetric, i.e. matrix element $(\mathtt{i}, \mathtt{j})$ (in row $\mathtt{i}$ and column $\mathtt{j}$) is the same as matrix element $(\mathtt{j}, \mathtt{i})$. This is because the existence of an edge between vertices $\mathtt{i}$ and $\mathtt{j}$ is exactly equivalent to the existence of an edge between vertices $\mathtt{j}$ and $\mathtt{i}$ in undirected graphs. This is, however, not necessarily true for directed graphs (unless all edges are reciprocated), so the adjacency matrix of a directed graph can be (and usually is) non-symmetric.

Adjacency lists

adjacency list

A third possible representation is called the adjacency list. An adjacency

list is a list consisting of smaller lists. There is one (smaller) list for each vertex, and the list for vertex `i` contains the successors of `i`. igraph sometimes prints graphs to the screen as adjacency lists, either using symbolic names or numeric ids.

As we mentioned earlier,igraph's internal data structure is none of the ones discussed in this book, but it is closest to the adjacency list representation. Luckily, igraph users do not need to know anything about this representation, they just need to know the igraph functions that work on igraph graphs.

The imaginary adjacency matrix

While igraph graph objects are not matrices, they behave as if they were, in that the square bracket indexing operator works for them the same way as it does for matrices. This adjacnecy matrix is imaginary, because it is never created internally. We can use the imaginary adjacency matrix to make queries and to manipulate the graph, as if it was an adjacency matrix.

Querying edges

Firstly, the '`[`' operator can be used to query whether an edge exists in a graph. It returns `1` for existing, and `0` for non-existing edges.

```
1.29.1) kite['A', 'B']
```

```
[1] 1
```

```
3) kite['A', 'J']
```

```
[1] 0
```

If multiple vertices are specified as indices, the '`[`' operator returns the whole submatrix that corresponds to the given vertices:

```
1.31.1) kite[c('A', 'B', 'C'), c('A', 'B', 'C', 'D')]
```

```
3 x 4 sparse Matrix of class "dgCMatrix"
  A B C D
A . 1 1 1
B 1 . . 1
C 1 . . 1
```

Your output might be slightly different, depending on whether your igraph version and setup uses sparse matrices from the *`Matrix`* R package or regular R matrices. In sparse matrices zeros are shown as dots when printed. The result is a (sparse or dense) submatrix of the imaginary adjacency matrix of the kite graph.

The usual R matrix indexing rules apply to indexing igraph graphs as well. If one of the indices are omitted, this is equivalent to giving all vertices. Querying the existence of edges between 'A' and all vertices is then easy:

```
1.32.1) kite['A',]
```

```
A B C D E F G H I J
0 1 1 1 0 1 0 0 0 0
```

If both indices are omitted, then the full adjacency matrix is given. Note that for large graphs, this might consume all memory if there is not sparse matrix support or it is turned off.

```
1.33.1) kite[]
```

```
10 x 10 sparse Matrix of class "dgCMatrix"

   [[ suppressing 10 column names 'A', 'B', 'C' ... ]]


A . 1 1 1 . 1 . . . .
B 1 . . 1 1 . 1 . . .
C 1 . . 1 . 1 . . . .
D 1 1 1 . 1 1 1 . . .
E . 1 . 1 . . 1 . . .
F 1 . 1 1 . . 1 1 . .
G . 1 . 1 1 1 . 1 . .
H . . . . . 1 1 . 1 .
I . . . . . . . 1 . 1
J . . . . . . . . 1 .
```

Negative indices specify vertices to be omitted:

```
1.34.1) kite[1,-1]
```

```
B C D E F G H I J
1 1 1 0 1 0 0 0 0
```

queries all edges between the first vertex, *other* vertices. You cannot mix negative indices and regular positive ones, this will give an error message.

Logical indices are also allowed:

```
1.35.1) degree(kite)
```

```
A B C D E F G H I J
4 4 3 6 3 5 5 3 2 1
```

```
4) kite[ degree(kite) >= 4, degree(kite) < 4 ]
```

```
5 x 5 sparse Matrix of class "dgCMatrix"
  C E H I J
A 1 . . . .
B . 1 . . .
D 1 1 . . .
F 1 . 1 . .
G . 1 1 . .
```

gives an adjacency submatrix for vertices with degree at least 4 against vertices with degree less than 4. As one might suspect, the `degree()` function calculates the vertex degree and returns it in a numeric vector, ordered according to vertex ids.

degree()

It often happens that one wants to test the existence of edges between several pairs of vertices. One solution to this is to query the submatrix of the adjacency matrix involving these vertices. This is, however quite inefficient. A much simpler solution is to use the `'from'` and `'to'` arguments of the indexing operator:

```
1.37.1) kite[ from = c('A', 'B', 'C'), to = c('C', 'D', 'E') ]
```

```
[1] 1 1 0
```

This form does exactly what we want, in this case, it queries that edges (A − C), (B − D) and (C − E), out of which 2 exist. Note that the `'from'` and `'to'` arguments must have the same length, otherwise it is not possible to make vertex pairs out of them and an error message will be given.

For directed graphs the '`[`' operator indexes the (usually) non-symmetric adjacency matrix, the first index specifies the tails, the second the heads of the queried edges.

So for the directed macaque brain graph, the order or the indices matters when querying edges:

```
1.38.1) macaque['V2', 'PIP']
```

```
[1] 1
```

```
   3) macaque['PIP', 'V2']
```

```
[1] 0
```

since there is a directed edge from 'V1' to 'PIP', but there is no directed edge from 'PIP' to 'V1'. Similarly, if the `'from'` and `'to'` arguments are used, then the former gives the tail vertices, the latter gives the heads.

The imaginary adjacency list

The imaginary adjacency list is similar to the imiaginary adjacency matrix. It is an adjacency list, however, and supports the '`[[`' double bracket indexing

operator. We can use it to query adjacent vertices, incident edges and edges between two sets of vertices.

Adjacent vertices

A common operation is querying the adjacent vertices of a vertex or a set of vertices. The `neighbors()` function does exactly this for a single vertex: *neighbors()*

```
1.40.1) neighbors(kite, "A")
```

```
+ 4/10 vertices, named, from 6b7ddad:
[1] B C D F
```

A more convenient way to do the same is to use the '`[[`' double bracket operator on the imaginary adjacency list. The advantage of '`[[`' is that is it more readable than `neighbors()`, and it works for multiple vertices. Of course it also works with symbolic vertex names:

```
1.41.1) kite[["A"]]
```

```
$A
+ 4/10 vertices, named, from 6b7ddad:
[1] B C D F
```

```
   5) kite[[c("A","B")]]
```

```
$A
+ 4/10 vertices, named, from 6b7ddad:
[1] B C D F

$B
+ 4/10 vertices, named, from 6b7ddad:
[1] A D E G
```

Note that for directed graphs, it does matter whether you put a comma before or after the index:

```
1.43.1) macaque[['V2', ]]
```

```
$V2
+ 15/45 vertices, named, from f7130f3:
 [1] V1     V3     V3A    V4     V4t    VOT    VP     MT
 [9] MSTd/p MSTl   PO     PIP    VIP    FST    FEF
```

```
   6) macaque[[, 'V2']]
```

```
$V2
+ 13/45 vertices, named, from f7130f3:
 [1] V1     V3     V3A    V4     V4t    VOT    VP     MT
 [9] MSTd/p MSTl   PO     FST    FEF
```

The first form gives the successors, the second form the predecessors of the specified vertices.

Also note that, unlike `neighbors()`, the '`[[`' operator always returns a list, even if only one vertex is queried. The list is named according to the vertex names if the graph itself is named.

Incident edges

The '`[[`' double bracket operator can just as well query the incident edges of some vertices, the only difference is that one needs to set the `'edges'` argument to `'TRUE'`:

1.45.1)
```
kite[['A', edges = TRUE]]
```

```
$A
+ 4/18 edges from 6b7ddad (vertex names):
[1] A--B A--C A--D A--F
```

gives the ids of the edges incident on vertex 'A'. Again, note, that the result is a list, always.

Edges between two sets of vertices

The '`[[`' can be also used to query all edges between two sets of vertices. For this both vertex sets must be given (their order only matters for directed graphs, in which case the first one contains the tail vertices), and also the `'edges = TRUE'` option.

1.46.1)
```
kite[[ c('A', 'B', 'C'), c('C', 'D', 'E', 'F'), edges = TRUE]]
```

```
[[1]]
+ 3/18 edges from 6b7ddad (vertex names):
[1] A--C A--D A--F

[[2]]
+ 2/18 edges from 6b7ddad (vertex names):
[1] B--D B--E

[[3]]
+ 2/18 edges from 6b7ddad (vertex names):
[1] C--D C--F
```

1.2.7 Storing meta-data as attributes

In this section we will use the `'UKfaculty'` data set. This is a social network of staff at a UK university faculty consisting of three schools (Nepusz et al, 2008). Since for some studies it is important which school a faculty member belongs to, the igraph object contains this information as well, as a *vertex attribute* named `'Group'`:

vertex attribute

```
1.47.1) data(UKfaculty)
```

Vertex attributes contain non-structural data associated with vertices. Vertex attributes always have a unique name, like `'Group'` above, and they are always defined for all vertices. In other words, it is not possible to define a vertex attribute for a subset of vertices only. (Of course it is always possible to set a vertex attribute to `'NA'`, the R 'not available' value.) The '`$`' operator, together with the `V()` function can be used to query and set vertex attributes.

```
1.48.1) V(UKfaculty)$Group
```

```
 [1] 3 1 3 3 2 2 2 1 3 2 1 2 2 1 1 2 3 1 1 1 1 2 2 1 1 1 2 2 1
[30] 2 1 1 2 1 1 3 1 3 1 2 1 2 1 3 3 1 2 1 2 4 1 1 3 1 1 1 1 1
[59] 3 3 3 3 2 1 2 2 2 2 2 4 2 2 3 3 3 2 2 3 1 1 3
```

In the `'UKfaculty'` graph, `'Group'` is a numeric vertex attribute, a single (integer) number for each vertex. This is not necessarily so in general. Vertex attributes can be of other types as well: strings, logical values, lists, etc.

Just like for vertices, it is also desirable to assign attributes to the edges of the graph. In our social network we have an *edge attribute* called `'weight'`, which is the strength of the friendship tie, calculated based on questionnaire response of both parties involved in the connection. The `E()` function can be used to query and set edge attributes. E.g. the weights of the first ten edges of our faculty network are:

edge attribute

```
1.49.1) E(UKfaculty)$weight[1:10]
```

```
 [1]  4 14  4  4 10  2  6  2  4  4
```

Finally, some meta-data belong to the graph itself. For each network in the *igraphdata* R package, there is a *graph attribute* called `'Citation'` that contains the reference to the paper in which it is discussed. Graph attributes can be queried by using the `'$'` operator on the graph object itself. (We nicely break the long string into several lines with the `strwrap()` function.)

graph attribute

```
1.50.1) cat(sep="\n", strwrap(UKfaculty$Citation))
```

```
Nepusz T., Petroczi A., Negyessy L., Bazso F.: Fuzzy
communities and the concept of bridgeness in complex
networks. Physical Review E, 77:016107, 2008.
```

So far we ignored the line starting with `'attr:'` in the output when printing or summarizing graphs to the screen:

```
1.51.1) summary(UKfaculty)
```

```
IGRAPH 6f42903 D-W- 81 817 --
+ attr: Type (g/c), Date (g/c), Citation (g/c), Author
| (g/c), Group (v/n), weight (e/n)
```

This line (and the following ones if one line is not enough) lists the various attributes of the graph. After each attribute name, it is given whether it is a graph (`'g'`), vertex (`'v'`), or edge (`'e'`) attribute, and also its R class, which can be numeric (`'n'`), character (`'c'`), logical (`'l'`) or something else which we call complex (`'x'`). Igraph does not distinguish between the complex attributes types currently.

Setting attributes is just as easy as querying them, `V()`, `E()` and the `'$'` operator can be used on the left hand side of an assignment, both to create new attributes and to assign new values to already existing ones. E.g. to create a copy of the faculty graph with randomly permuted groups, one would do

```
1.52.1) UKfaculty_perm <- UKfaculty
    2) V(UKfaculty_perm)$Group <- V(UKfaculty)$Group %>% sample()
```

If you specify a vector that is shorter than the number of vertices (or edges, in case of `E()`), then it will be recycled, according to the usual R recycling rules.

In addition to the `V()`/`E()` and `'$'` notation, igraph includes some functions that are perhaps more convenient when attributes are used programatically. Think of querying all vertex attributes and printing them individually to the screen or writing them to a file as an example. See Table **??** These functions, just like all graph manipulating igraph functions, do not modify the graph in place, but create new graph objects, so don't forget to assign the result to an R variable.

Special attributes

Igraph treats some attributes specially. Perhaps the most important of them is the `'name'` vertex attribute that contains the symbolic names of the graph vertices in named graphs. In fact, what we have seen as named graphs previously in Section 1.2.5 are nothing else but graphs that have a vertex attribute called `'name'`. Other such vertex attributes are `'color'`, `'size'`, etc., these are considered for visualization. The `'weight'` edge attribute is used by many igraph functions (such as `max_flow()`, `diameter()`, etc.) for invoking the weighted versions of graph algorithms. See Section 2.4 for more about weighted graphs. You can find the most commonly used specially treated attributes in Table 1.3.

Name	*Type*	*Meaning*
`'color'`	edge	Gives the color of the edges, when plotting.
`'color'`	vertex	Plotting functions use this for the color of the vertices.
`'layout'`	graph	The layout (a function or matrix) of the graph when plotting it.
`'name'`	vertex	Defines symbolic vertex names. Symbolic vertex names can be used to refer to vertices, in all igraph functions. Graphs that have this attribute are called named graphs.
`'shape'`	vertex	The shape of the vertices when plotting the graph.
`'type'`	vertex	For bipartite (two-mode) graphs it defines the vertex groups.
`'weight'`	edge	Edge weights. Used by several functions, e.g. shortest path related functions, community structure finding functions, maximal flow functions, etc.

Table 1.3 The most commonly used graph, vertex and edge attributes that are treated specially by igraph. For a complete list of plotting-related attributes see Chapter 3.

Be aware of the special treatment of some vertex/edge attributes, otherwise igraph functions might behave in a surprising way. The `'name'` vertex attribute is used to print the graph to the screen in a human readable way, but if one stores something else but character strings in it, the output might not make sense at all, or igraph might not be even able to interpret the attribute as vertex names. To make things worse, future igraph versions will probably treat even more attributes specially. So the best practice is to use attribute names starting with an uppercase letter to make sure that they have no influence on igraph's internal functions. Igraph special attributes are never capitalized.

Queries with attributes

Attributes interact nicely with vertex and edges sequences. It is easy to select edges and vertices based on attributes, and it is easy to query and set attributes for members in a vertex or edge set.

Attributed graphs as data frames

1.3 Creating and manipulating graphs

1.3.1 Creating graphs

There are several ways of representing graphs on the computer: adjacency matrices, edge lists, etc. Igraph needs that the data is in a special format, the igraph graph object. This is to ensure that operations on the data are

consistent and fast. There are several functions that create igraph graph objects, the most frequently used ones are listed in Table 1.4. They can be classified into four categories:

1. Predefined graph structures. These are useful to experiment with graph measures, or as building blocks of larger graphs.
2. Converters from other graph representations, they are most often used to convert graph data sets to igraph objects.
3. Random graph models.
4. The `read_graph()` function, that reads files in various graph data formats.

Once an igraph graph object is created, you can use igraph functions to explore its structure and visualize it.

Let us now create a graph from scratch. We want to work with the social network illustrated in Fig. 1.1; this graph has 10 vertices and 18 edges. We will build the graph step by step: first we create an empty graph, then add the vertices, and finally the edges to it. We first show the complete R command that creates the graph, and will explain each step afterwards.

1.53.1)
```
kite <- make_empty_graph(directed = FALSE) +
  vertices(LETTERS[1:10]) +
  edges('A','B',   'A','C',   'A','D',   'A','F',
        'B','D',   'B','E',   'B','G',
        'C','D',   'C','F',
        'D','E',   'D','F',   'D','G',
        'E','G',
        'F','G',   'F','H',
        'G','H',
        'H','I',
        'I','J')
```

make_empty_graph()

The simplest possible graph is an empty graph, a graph without vertices and edges, and the function `make_empty_graph()` creates such a graph. An empty graph seems useless, but we will only use it as a starting point anyway. By default `make_empty_graph()` creates directed graphs, so we also set the optional `'directed'` argument to `'FALSE'`, as our graph is undirected.

The next step will be to add vertices to the graph. To save ourselves from typing, we will use the first letters of the first names of the actors as ids. We simply list the vertex ids in `vertices()`.

vertices()

edges()

Next, we add the edges to the graph. Each pair of ids in `edges()` will correspond to an undirected edge. Note that since in an undirected graph `('A','B')` and `('B','A')` mean the same edge, we only need to list each edge once. The `'kite'` graph is ready to use:

1.54.1)
```
kite
```

1 Predefined structures

`make_directed_graph()`	Directed graph with given edges.
`make_undirected_graph()`	Undirected graph with given edges.
`make_empty_graph()`	Empty graph with given order and zero size.
`make_full_graph()`	Full graph of given order.
`make_full_bipartite_graph()`	Full bipartite graph.
`make_graph("Petersen")`	Predefined graph structures, see `?make_graph` for a list.
`make_ring()`	Ring graph.
`make_lattice()`	Regular lattice.
`make_star()`	Star graph.
`make_tree()`	Regular, almost complete trees.

2 Convert a graph representation to igraph

`graph_from_adjacency_matrix()`	From a dense or sparse adjacency matrix.
`graph_from_edgelist()`	From edge lists.
`graph_from_adj_list()`	From adjacency lists.
`graph_from_data_frame()`	From data frame(s).
`graph_from_incidence_matrix()`	From incidence matrices.
`graph_from_graphnel()`	From the `graphNEL` representation of the *`graph`* package.
`graph_from_literal()`	From a simple formula-like notation.

3 Sample from random graph models

`sample_gnp()`	$\mathsf{G}(\mathfrak{n}, \mathfrak{p})$ Erdős-Rényi random graphs, also called Bernoulli graphs.
`sample_gnm()`	$\mathsf{G}(\mathfrak{n}, \mathfrak{m})$ Erdős-Rényi random graphs.
`sample_degseq()`	Random graphs with given degree sequence.
`sample_grg()`	Geometric random graphs.
`sample_pa()`	Preferential attachment model.
`sample_sbm()`	Stochastic block-models.
`sample_smallworld()`	Small-world graphs.
`sample_hrg()`	Hierarchical random graphs.
`sample_bipartite()`	Bipartite random graphs.

4 From graph files

`read_graph()`	From various file formats, including GraphML, GML and Pajek.

Table 1.4 The most frequently used functions that create igraph graph objects.

```
IGRAPH 3c60b3a UN-- 10 18 --
+ attr: name (v/c)
+ edges from 3c60b3a (vertex names):
 [1] A--B A--C A--D A--F B--D B--E B--G C--D C--F D--E D--F
[12] D--G E--G F--G F--H G--H H--I I--J
```

+ operator

The vertices and the edges were added to the graph via the plus (+) operator. This operator is very versatile in igraph, and can add vertices, edges, paths, or other graphs to an existing graph.

There are by and large three ways to manipulate igraph graphs in R. We have already touched upon one of them, the use of the plus and minus operators to add and delete vertices and edges. Now we start our detailed discussion of graph manipulation with another method, the indexing operators.

'[' operator
'[[' operator

The single bracket ('[') and double bracket ('[[') operators are used in R to index vectors, matrices and lists. They can also be used to index the imaginary adjacency matrix of an igraph graph. We stress the word 'imaginary' here. The actual adjacency matrix of the graph is never created, this would be inefficient for large graphs. The igraph graph only pretends to be an adjacency matrix. Let us know discuss what this imaginary matrix is good for. We will use the kite graph that we created in the Section 1.2.5.

The bracket operators can be used with numeric vertex ids and symbolic vertex names as well. To illustrate this, we query the mapping of vertex names to vertex ids:

```
1.55.1) rbind(as.vector(V(kite)), V(kite)$name)
```

```
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
[2,] "A"  "B"  "C"  "D"  "E"  "F"  "G"  "H"  "I"  "J"
```

The idiom `'as.vector(V(kite))'` gives the numeric ids of the vertices, we have already seen that `'V(kite)$name'` lists the vertex names, in an order that matches our listed vertex ids. The `rbind()` function puts them together in a two-row matrix.

Adding edges

The '[' operator can also be used to add edges to the graph, the code

```
1.56.1) kite['F', 'I'] <- 1
    2) kite
```

```
IGRAPH c5c0873 UN-- 10 19 --
+ attr: name (v/c)
+ edges from c5c0873 (vertex names):
 [1] A--B A--C A--D A--F B--D B--E B--G C--D C--F D--E D--F
[12] D--G E--G F--G F--H G--H H--I I--J F--I
```

adds an undirected edge from Fernando to Ike. Quite logically, all the previously shown indexing variants can be used to specify the edges to be added. As another example, we create a star graph, where all other vertices point to the center vertex:

```
1.57.1) star <- make_empty_graph(10)
    2) star[-1,1] <- 1
    3) star
```

```
IGRAPH 5e4a5b1 D--- 10 9 --
+ edges from 5e4a5b1:
[1]  2->1  3->1  4->1  5->1  6->1  7->1  8->1  9->1 10->1
```

First, the `10` optional argument of `make_empty_graph()` specifies the number of isolate vertices in the graph. Then we add the edges using the matrix notation. The first -1 index defines the tails of the edges, and it means "all vertices, except vertex 1"; the second index defines the head vertex, which is now vertex 1. The whole expression reads "Add edges from all vertices, except 1, to vertex 1".

An interesting feature of the '`[`' based edge addition is that it does not create multiple edges, but just ignores the already existing edges:

```
1.58.1) star[-1,1] <- 1   #label{vrb:star2:add}
    2) star
```

```
IGRAPH 3fcdd29 D--- 10 9 --
+ edges from 3fcdd29:
[1]  2->1  3->1  4->1  5->1  6->1  7->1  8->1  9->1 10->1
```

Nothing happens here, since all the edges we are trying to add in line **??** are already present in the graph.

The '`from`' and '`to`' arguments are handy when one wants to add many edges and they are given by their endpoints. To illustrate this we create a graph of roman letters, where two vertices are connected by a directed edge if and only if they are adjacent in the expression "adjacency matrix" (ignoring the space character).

```
1.59.1) adjletters <- strsplit("adjacencymatrix", "")[[1]] #label{vrb:adj:split}
    2) adj <- make_empty_graph()
    3) adj <- adj + unique(adjletters)                    #label{vrb:adj:unique}
    4) adj[ from=adjletters[-length(adjletters)],         #label{vrb:adj:set1}
    5)      to  =adjletters[-1] ] <- 1
    6) adj
```

```
IGRAPH 545faf5 DN-- 12 14 --
+ attr: name (v/c)
+ edges from 545faf5 (vertex names):
 [1] a->d d->j j->a a->c c->e e->n n->c c->y y->m m->a a->t
[12] t->r r->i i->x
```

The `strsplit()` function in line **??** splits the given word into letters. Then we use these letters as vertex names in line **??**, where the `unique()` function makes sure that each letter appears only once. Line **??** add the edges between the letters. As usually in R, the negative indices specify the elements to omit from the vector, in this case we omit the last and then the first element.

Let's stop for a second to examine what exactly happens when we add edges to a graph using the matrix notation. The reader might remember, that we stressed earlier that igraph functions never modify igraph graph objects in place, but they always make copies of them; our latest examples, however, *seem* to modify the graph in place. The truth is, they only *seem* to do that, but in fact when one writes `'tree6[1,5] <- 1'`, R first creates a copy of the object named `'tree6'`, then performs the operation on this copy, and finally names the copy as `'tree6'`. Most of the time we don't need to know about this internal mechanism, it is only important if one works with large graphs, where copying must be avoided, if possible.

Deleting edges

Not very surprisingly, the matrix notation and the '`[`' operator can also be used to delete edges from the graph. To achieve this, one should use the value `0` (or `'FALSE'`) at the right hand side of the assignment:

```
1.60.1) kite['F', 'I'] <- 0
    2) kite
```

```
IGRAPH cb00454 UN-- 10 18 --
+ attr: name (v/c)
+ edges from cb00454 (vertex names):
 [1] A--B A--C A--D A--F B--D B--E B--G C--D C--F D--E D--F
[12] D--G E--G F--G F--H G--H H--I I--J
```

removes the recently added edge from the kite graph.

All the different methods (negative, logical indices, the `'from'` and `'to'` arguments, etc.) can be used to select the edges to be removed.

Simple graphs, loops, multiple edges

1.4 Exercises

▶ EXERCISE 1.1. Create a graph that is weakly connected, but not strongly connected. Which one is the smallest such graph, in terms of order and size?

▶ EXERCISE 1.2. Show at least three ways to delete the isolated vertices from an undirected graph. (Hint: (1) use the '`-`' operator, (2) use the `delete.vertices()` function, (3) use the `subgraph()` function.)

▶ EXERCISE 1.3. Show at least three ways to delete edges that are incident on vertex 'E' in the kite graph. (Hint: (1) use the '`-`' operator, (2) use the '`[`' operator, (3) use the `delete.edges()` function. You might need to use the '`[[`' operator as well, with the `'edges'` argument, to select the edges to be deleted.)

▶ EXERCISE 1.4. What is wrong with the following code? (`'letters'` is a builtin R character array, and contains the 26 letters of the Roman alphabet, in lowercase; `seq_len()` creates a sequence of integers; `runif()` generates uniformly distributed random numbers. See their manual pages for details.)

```
1.61.1) R <- graph.ring(10)
    2) V(R)$name <- letters[seq_len(vcount(R))]
    3) add.edges(R, c('a', 'e', 'b', 'h'))
    4) E(R)$weight <- runif(ecount(R))
    5) E(R)['a|e']$weight
```

Chapter 2
Paths

2.1 Introduction

Recall that a path in a graph is a sequence of (zero or more) edges. The path is the most important concept when dealing with network data, because everything else is based on it. When trying to find central actors in a network, we can look for vertices that fall on a lot of paths, or vertices that are connected to all others via short paths. When measuring the connectedness of a network, we look for many independent paths between vertices, so that if one is cut, the others still keep the network together. When modeling epidemics, the modeled disease speads on the paths.

Dealing with paths means that we need special tools, and table-based data analysis software is, however sophisticated, not enough. Table-based software tools, such as relational databases, or R packages that deal with data frames focus on individual relationships, and are typically not capable of handling a chain of relationships, a path.

In this Chapter we will use a sample the network of US airports. Each airport is a vertex, and each directed edge denotes a flight between two airports, by a specific carrier. If a route is served by multiple carriers, then multiple edges are added between the two airports.

2.1.1)
```
library(igraph)
library(igraphdata)
data(USairports)
summary(USairports)
```

```
IGRAPH bf6202d DN-- 755 23473 -- US airports
+ attr: name (g/c), name (v/c), City (v/c), Position
| (v/c), Carrier (e/c), Departures (e/n), Seats (e/n),
| Passengers (e/n), Aircraft (e/n), Distance (e/n)
```

The network has some metadata, for the vertices we know the corresponding `City` and `Position`. For edges we have the name of the `Carrier`, the

number of `Departures`, the total number of `Seats` on these, and number of `Passengers`, the `Aircraft` type, the the `Distance` in miles. It is usually worth peeking at the vertex and edge metadata, to get a better sense of what they are:

```
2.2.1) V(USairports)[[1:5]]
```

```
+ 5/755 vertices, named, from bf6202d:
  name          City         Position
1  BGR    Bangor, ME N444827 W0684941
2  BOS    Boston, MA N422152 W0710019
3  ANC Anchorage, AK N611028 W1495947
4  JFK  New York, NY N403823 W0734644
5  LAS Las Vegas, NV N360449 W1150908
```

```
9) E(USairports)[[1:5]]
```

```
+ 5/23473 edges from bf6202d (vertex names):
  tail head tid hid             Carrier Departures Seats
1  BGR  JFK   1   4 British Airways Plc          1   226
2  BGR  JFK   1   4 British Airways Plc          1   299
3  BOS  EWR   2   7 British Airways Plc          1   216
4  ANC  JFK   3   4 China Airlines Ltd.         13  5161
5  JFK  ANC   4   3 China Airlines Ltd.         13  5161
  Passengers Aircraft Distance
1        193      627      382
2        253      819      382
3        141      627      200
4       3135      819     3386
5       4097      819     3386
```

Even if a network is not supposed to have self-loops, i.e. flights that went back to the site of departure, it is a good idea to check for them, because the presence of self-loops can modify the behavior of many graph algorithms.

```
2.4.1) sum(which_loop(USairports))
```

```
[1] 53
```

Indeed, the network has 53 self-loops. These might be mistakes, or aircrafts that really returned to the departure airport because of some reasons. We don't want to consider them in the further analysis, and remove them:

```
2.5.1) USairports <- simplify(USairports, remove.loops = TRUE,
2)   remove.multiple = FALSE)
3) any(which_loop(USairports))
```

```
[1] FALSE
```

2.2 Paths in graphs and multigraphs

Because of the multiple edges corresponding to several carriers flying a route, the network is not a simple graph (in other words, it is a multi-graph). E.g. there are 14 edges from Boston to JFK, New York City. They belong to different carriers and different aircraft types. Here is the first five of them:

```
2.6.1) length(USairports[["BOS", "JFK", edges = TRUE]][[1]])
```

```
[1] 14
```

```
3) USairports[["BOS", "JFK", edges = TRUE]][[1]][[1:5]]
```

```
+ 5/23420 edges from d71bfa8 (vertex names):
      tail head tid hid                      Carrier
22558  BOS  JFK   2   4    Lufthansa German Airlines
21946  BOS  JFK   2   4     Chautauqua Airlines Inc.
19859  BOS  JFK   2   4 American Eagle Airlines Inc.
19858  BOS  JFK   2   4 American Eagle Airlines Inc.
18195  BOS  JFK   2   4  Atlantic Southeast Airlines
      Departures Seats Passengers Aircraft Distance
22558          1   221        202      696      187
21946          2   100         92      675      187
19859         65  2860       2441      676      187
19858         72  2664       2295      674      187
18195          1    65         35      631      187
```

The data on multiple carriers might or might not be useful, depending on the analysis we are to perform. In the following we don't need it, so we simplify the network and remove all multiple edges. Note that we still keep some edge attributes and we sum their values over the multiple edges between the same pair of vertices. The rest of the edge attributes we ignore.

```
2.8.1) air <- simplify(USairports, edge.attr.comb =
2)   list(Departures = "sum", Seats = "sum", Passengers = "sum", "ignore"))
3) is_simple(air)
```

```
[1] TRUE
```

```
5) summary(air)
```

```
IGRAPH ce25b00 DN-- 755 8228 -- US airports
+ attr: name (g/c), name (v/c), City (v/c), Position
| (v/c), Departures (e/n), Seats (e/n), Passengers
| (e/n)
```

Often, we want to select a path in a network, to visualize it, or to manipulate the attributes of its edges or vertices. In a simple directed graph this is easy. As there is at most one edge between each pair of vertices, we can

simply list the vertices along the path. The edges along the path can also easily selected and manipulated:

```
flight <- V(air)['BDL', 'FLL', 'BOS', 'PBI']
E(air)$width <- 0
E(air, path = flight)$width <- 1
```

In graphs with multiple edges, it is often not sufficient to list the vertices along a path, since these may not uniquely determinte the edges. In this case the edges must be listed, usually based on some edge attributes. E.g. to select the edges along all paths for the route from `BDL` to `PBI` we can write

```
BDL_PBI <- E(USairports)[ 'BDL' %->% 'FLL', 'FLL' %->% 'BOS',
                          'BOS' %->% 'PBI']
BDL_PBI[[]]
```

```
+ 8/23420 edges from d71bfa8 (vertex names):
      tail head tid hid                    Carrier Departures
5007   BDL  FLL  25 109            JetBlue Airways         59
9392   BDL  FLL  25 109       Delta Air Lines Inc.         43
11830  BDL  FLL  25 109     Southwest Airlines Co.         30
5090   FLL  BOS 109   2            JetBlue Airways        156
7589   FLL  BOS 109   2 Continental Air Lines Inc.          1
23142  FLL  BOS 109   2           Spirit Air Lines         84
5032   BOS  PBI   2  16            JetBlue Airways         29
5033   BOS  PBI   2  16            JetBlue Airways        117
      Seats Passengers Aircraft Distance
5007   8850       6990      694     1173
9392   6127       4530      655     1173
11830  4110       2668      612     1173
5090  23400      17608      694     1237
7589    160        110      614     1237
23142 12180       8775      698     1237
5032   2900       1926      678     1197
5033  17550      14707      694     1197
```

and then filter this for a given carrier:

```
BDL_PBI[ Carrier == 'JetBlue Airways' ][[]]
```

```
+ 4/23420 edges from d71bfa8 (vertex names):
     tail head tid hid         Carrier Departures Seats
5007  BDL  FLL  25 109 JetBlue Airways         59  8850
5090  FLL  BOS 109   2 JetBlue Airways        156 23400
5032  BOS  PBI   2  16 JetBlue Airways         29  2900
5033  BOS  PBI   2  16 JetBlue Airways        117 17550
     Passengers Aircraft Distance
5007       6990      694     1173
```

```
5090      17608      694     1237
5032       1926      678     1197
5033      14707      694     1197
```

2.3 Shortest paths

In the airport network, there are a lot of paths that get to `PBI` from `BDL`. One path corresponds to a single edge, a direct flight, others require multiple steps. This is true for most networks and vertices in general: usually there are many paths between vertices, and these probably contain different number of edges and vertices. In fact, if a directed network contains cycles, then the number of paths between a vertex pair can be infinite, as the path may go around a cycle any number of times. A path that has no repeated vertices is called a *simple path*.

The number of edges included in a path is called the *length* of the path. Of all the different paths from a vertex to another one, the one(s) that contain the fewest number of edges is (are) called the shortest paths (or geodesics). Shortest paths are important in many optimization problems. When searching for flights, one usually prefers the smallest number of transfers, i.e. the shortest paths in the network. The length of the shortest path between two vertices is also called their *distance*.

The `distances()` igraph function calculates the lengths of shortest paths between the specified vertices, and the `shortest_paths()` igraph function returns the paths themselves. *shortest_paths()*

```
2.13.1) distances(air, c('BOS', 'JFK', 'PBI', 'AZO'),
2)                 c('BOS', 'JFK', 'PBI', 'AZO'))
```

```
    BOS JFK PBI AZO
BOS   0   1   1   2
JFK   1   0   1   2
PBI   1   1   0   1
AZO   2   2   1   0
```

```
8) shortest_paths(air, from = 'BOS', to = 'AZO')$vpath
```

```
[[1]]
+ 3/755 vertices, named, from ce25b00:
[1] BOS PBI AZO
```

Often shortest paths are not unique, and `shortest_paths()` only computes a single (arbitrary) shortest path between each requestes pair of vertices. `all_shortest_paths()` lists all of them: *all_shortest_paths()*

```
2.15.1) all_shortest_paths(air, from = 'BOS', to = 'AZO')$res
```

```
[[1]]
+ 3/755 vertices, named, from ce25b00:
[1] BOS ORD AZO

[[2]]
+ 3/755 vertices, named, from ce25b00:
[1] BOS MSP AZO

[[3]]
+ 3/755 vertices, named, from ce25b00:
[1] BOS DTW AZO

[[4]]
+ 3/755 vertices, named, from ce25b00:
[1] BOS PBI AZO
```

2.4 Weighted graphs

A simple (binary) graph corresponds to a binary relation, edges are either present or absent between any pair of vertices. An *edge-weighted graph*, or simply *weighted graph* mathematically corresponds to a graph (as before), plus a mapping from the edges of the graph, to a set of numbers, typically (but not always) the $\mathbb{R}$ set of real numbers. These numbers are called the edge weights. Often an unweighted graph can be considered the special case of a weigthed graph, with all the edge weights being one.

Edge weights are often crucial for network analysis and modeling, and many data sets include natural edge weights. They often represent the strength of a connection, or distance, or some other quantity. Whereas they might have different meaning in different graphs, it is very important to be clear what they represent, when interpreting the results of graph algorithms on weighted graphs.

Edge weights are represented in igraph as the `'weight'` edge attribute. We will now create a simple weigthed graph from the original airport network, and set the distance between airports as edge weight.

2.16.1)
```
wair <- simplify(USairports, edge.attr.comb =
   list(Departures = "sum", Seats = "sum", Passangers = "sum",
        Distance = "first", "ignore"))
E(wair)$weight <- E(wair)$Distance
```

`Distance = "first"` specifies to take the Distance value of the (arbitrary) first edge from the multiple edges between each vertex pair. Since these are phyisical distances between airports they are independent of the actual flight.

We keep and sum over some other edge attributes and ignore the rest of them. `wair` is now a weighted graph, as it has a `weight` edge attribute:

```
2.17.1) summary(wair)
```

```
IGRAPH bcd7add DNW- 755 8228 -- US airports
+ attr: name (g/c), name (v/c), City (v/c), Position
| (v/c), Departures (e/n), Seats (e/n), Distance (e/n),
| weight (e/n)
```

In a weighted graph, it is often natural to consider paths that contain the smallest total edge weight as shortest paths. In the airport network, these are the paths that require the shortest distance of travel. By default `distances()`, `shortest_paths()` and `all_shortest_paths()` consider weighted paths if the graph is weighted:

```
2.18.1) distances(wair, c('BOS', 'JFK', 'PBI', 'AZO'),
    2)                  c('BOS', 'JFK', 'PBI', 'AZO'))
```

```
     BOS  JFK  PBI  AZO
BOS    0  187 1197  745
JFK  187    0 1028  621
PBI 1197 1028    0 1116
AZO  745  621 1116    0
```

```
    8) shortest_paths(wair, from = 'BOS', to = 'AZO')$vpath
```

```
[[1]]
+ 3/755 vertices, named, from bcd7add:
[1] BOS DTW AZO
```

```
   12) all_shortest_paths(wair, from = 'BOS', to = 'AZO')$res
```

```
[[1]]
+ 3/755 vertices, named, from bcd7add:
[1] BOS DTW AZO
```

Igraph finds shortest paths in unweighed graphs using the Breadth first search (BFS) algorithm. This is fast, and requires linear time to find shortest paths from one vertex to all others in the network. For weighted graphs, the algorithms are more difficult, especially if the edge weights can be zero or negative. For a graph with negative edge weights and cycles, the shortest path might not even exist, if the total edge weight of the cycle is negative, then there is always a shorter path between its vertices: just go over the cycle one more.

In weighted graphs with strictly positive edge weights `distances()`, `shortest_paths()` and `all_shortest_paths()` use Dijkstra's algorithm, requiring $\mathcal{O}(n \log n)$ time for a single source vertex in a graph with n vertices. This is fast even for large graphs.

If negative edge weights are present, then `disances()` uses either Johnson's algorithm or the Bellman-Ford algorithm is used, depending on the number of input vertices. These require at least $\mathcal{O}(nm)$ time for a single source vertex and a graph of n vertices and m edges. Note that negative cycles are still not allowed, and lead to error messages.

Currently, `shortest_paths()` and `all_shortest_paths()` do not work on graphs with negative edge weights.

2.4.1 The Pocket Cube graph

It is often natural to represent turn-based single- or multi-person games as graphs. Each state of the game is a vertex in the graph, and if it is possible to get to state w from the current state v, then the graph contains a v, w edge.

The Pocket Cube is a 2x2 Rubik's Cube, that consists of 8 smaller cubes (cubies). The goal is to get to the state where all faces of the large cube have a single color, by turning the sides (i.e. the cubies) of the large cube. In the graph representation of the cube, each state consist of the positions and orientation of the cubies, and two states are connected by an edge, if there is a turn that takes the first state to the second. It is clear that an "opposite" turn brings back to the previous state, so the graph is undirected.

Many interesting questions about the Pocket Cube can be formalized in terms paths and distances of its graph:

- How many states does the cube have?
- How many different turns can we make from a given state?
- At least how many turns do we need to solve the cube from a given state?
- What are these turns?
- Is the sequence of turns to the solution always unique?
- At most how many turns do we need from an arbitrary state?
- How many turns do we need on average?
- Can we solve the cube from any state?
- Can we reach any state from any other?

While some concepts that neatly describe these problems mathematically only come later in this book, we suggest the reader to take a couple minutes and think about what they mean in terms of the graph of the game.

2.5 Diameter and mean shortest path length

It is sometimes useful to describe an undirected graph with the distances between its vertices. In this description one extreme graph is the full graph,

containing has all possible edges. Another extreme is a graph that looks like a straight line, each vertex is connected only to the next one. For n vertices this graph has a distance of $n-1$, the largest possible, and also all smaller distances between 0 and $n-1$.

The largest distance of a graph is called its *diameter*. A graph with a small diameter is bound to have only short distances. It is an interesting observation that most real networks have a small diameter compared to their size.

In the special case when some vertices are not reachable via a path from some others, the diameter of the graph is infinite. Note that the igraph `diameter()` function works differently in this case, and it returns the longest *finite* distance. *diameter()*

The diameter of a directed graph is rarely used, because in network data sets if happens very often, that some vertex pairs are not connected by a directed path at all, and then the diameter would be infinite.

For the airport network, that largest finite distance corresponds to the most transfers we have to have when flying within the US:

```
2.21.1) diameter(air)
```

```
[1] 9
```

This is surprisingly high, so it is worth inspecting the vertices and edges along it.

```
2.22.1) dia_v <- get_diameter(air)
    2) dia_e <- E(air, path = dia_v)
    3) dia_v[[]]
```

```
+ 10/755 vertices, named, from ce25b00:
    name              City         Position
416  HYG      Hydaburg, AK N551223 W1324942
413  DOF      Dora Bay, AK N551400 W1321300
253  WFB     Ketchikan, AK N552040 W1313948
375  KTN     Ketchikan, AK            MIAMI
161  SEA       Seattle, WA N472656 W1221834
3    ANC     Anchorage, AK N611028 W1495947
232  ADQ        Kodiak, AK N574460 W1522938
236  KKB     Kitoi Bay, AK N581127 W1522214
246  SYB      Seal Bay, AK N581000 W1523000
239  KPR Port Williams, AK N582924 W1523456
```

```
   16) dia_e[[]]
```

```
+ 9/8228 edges from ce25b00 (vertex names):
     tail head tid hid Departures Seats Passengers width
7018  HYG  DOF 416 413          2    12          1     0
7014  DOF  WFB 413 253          2    12          1     0
```

```
5953  WFB  KTN 253 375         59   482         59     0
6829  KTN  SEA 375 161         88 11034       8664     0
4828  SEA  ANC 161   3        451 69591      59116     0
109   ANC  ADQ   3 232        181  9692       5064     0
5876  ADQ  KKB 232 236          6    24          2     0
5894  KKB  SYB 236 246          4    16          3     0
5916  SYB  KPR 246 239          4    16          5     0
```

It turns out that many of these routes had only 2-6 flights with 1-5 passengers, and maybe it is better to exclude these flight completely from this calculation:

```
2.24.1) air_filt <- delete_edges(air,
2)   E(air)[ Passengers <= 10 ])
3) summary(air_filt)
```

```
IGRAPH 6db6386 DN-- 755 7472 -- US airports
+ attr: name (g/c), name (v/c), City (v/c), Position
| (v/c), Departures (e/n), Seats (e/n), Passengers
| (e/n), width (e/n)
```

```
8) diaf_v <- get_diameter(air_filt)
9) diaf_e <- E(air_filt, path = diaf_v)
10) diaf_v[[]]
```

```
+ 9/755 vertices, named, from 6db6386:
    name              City         Position
181  TIQ        Tinian, TT N145949 E1453705
180  SPN        Saipan, TT N150708 E1454346
178  GUM          Guam, TT N132900 E1444746
196  HNL      Honolulu, HI N211907 W1575521
5    LAS     Las Vegas, NV N360449 W1150908
372  BLI    Bellingham, WA N484734 W1223215
517  FRD Friday Harbor, WA N483119 W1230128
657  LKE       Seattle, WA N473744 W1222019
656  KEH       Kenmore, WA N474526 W1221526
```

```
22) diaf_e[[]]
```

```
+ 8/7472 edges from 6db6386 (vertex names):
     tail head tid hid Departures Seats Passengers width
5256  TIQ  SPN 181 180        262  1572       1544     0
5253  SPN  GUM 180 178        164  7544       4684     0
5250  GUM  HNL 178 196         31  7831       6731     0
5354  HNL  LAS 196   5        101 26739      23094     0
279   LAS  BLI   5 372         89 13519      12243     0
6289  BLI  FRD 372 517        140   698         57     0
6909  FRD  LKE 517 657          9    58         20     0
7327  LKE  KEH 657 656         24   200         41     0
```

The diameter is not an accurate description of the distances in a network, it is only an upper bound. It is often better to consider the mean of all distances instead. Note that by default the `mean_distance()` igraph function only considers finite distances. The `distance_table()` creates a histogram of all distances

mean_distance()
distance_table()

```
2.27.1) mean_distance(air)
```

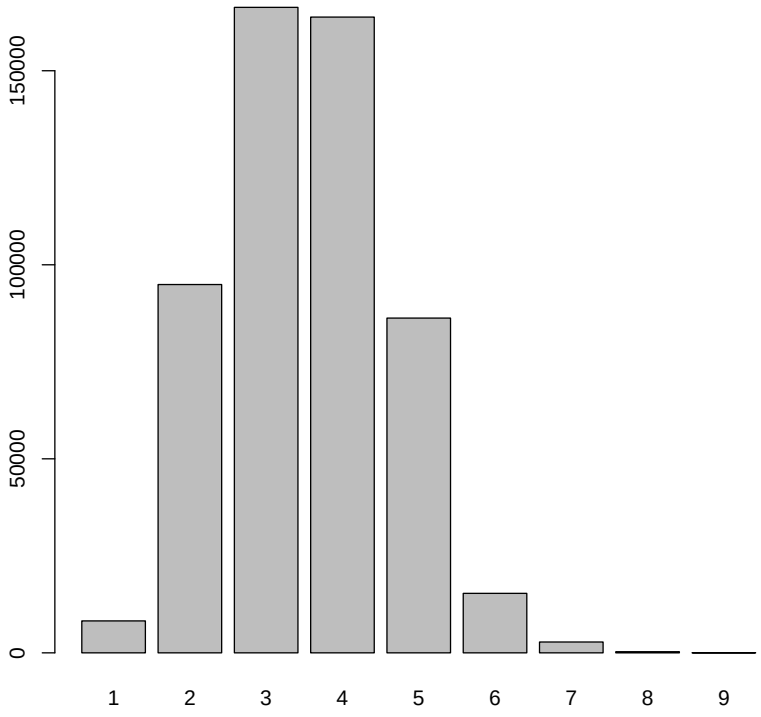
```
[1] 3.52743
```

```
3) air_dist_hist <- distance_table(air)
4) air_dist_hist
```

```
$res
[1]   8228  94912 166335 163830  86263  15328   2793    291
[9]     27

$unconnected
[1] 31263
```

```
11) barplot(air_dist_hist$res, names.arg = seq_along(air_dist_hist$res))
```



Most routes need three or four individual flights, in other words, two or three transfers. It is probably also true that most passengers travel on routes that only need at most one transfer.

2.6 Components

An undirected graph is *connected* if there is a path from any vertex to any other. A graph that is not connected is said to be *unconnected*. An unconnected graph has multiple *components*. A component of a graph is a maximal induced subgraph that is connected. It is maximal in the sense that is is not a subgraph of any other connected subraph.

Connectedness is a property of a graph that is often important in practice. If a network of internet routers is connected, then we can send packets from

any machine to any other. If an organizational network is connected then it has the potential to spread information to everybody.

It is an interesting question whether the Pocket Cube graph is connected. If it is connected, then we can start from any (valid) painting of the cubies, and get to the solved state with a finite number of legal turns.

```
2.30.1) is_connected(pc)
    2) count_components(pc)
    3) components(pc)$csize
```

`is_connected()` checks if a graph is connected. `count_components()` reports its number of components. The Pocket Cube graph is not connected and has three components. The components have the same size. This means that if we are to remove the stickers from the cubies and then put them back (uniformly) randomly, we have 1/3 chance to make a cube that is solvable.

is_connected()

count_components()

The `decompose()` function decomposes a graph into its connected components and returns a list of graphs. This is sometimes useful if we want to work with the components individually, e.g. to visualize them.

decompose()

2.6.1 Biconnected components

Connected networks can further classified according to the strength of their connectedness. The intuition is that if there tend to be more independent paths between vertices, then the graph is more strongly connected. An undirected graph that has at least two paths between each pairs of vertices is said to be biconnected. A graph that is not biconnected has biconnected components. Biconnected components are maximal induced subgraphs that are biconnected. The `biconnected_components()` function finds them.

biconnected_components()

Note that in contrary to components, biconnected components are not necessarily disjunct. For example, in the bowtie network the center node is clearly part of both biconnected components. This node is also special, because if removed from a graph, the remainder is not connected any more. These kind of vertices are called articulation points

```
2.31.1) bow_tie <- make_graph( ~ A - B - C - A - D - E - A)
    2) biconnected_components(bow_tie)$components
```

```
[[1]]
+ 3/5 vertices, named, from 6dea8b8:
[1] C B A

[[2]]
+ 3/5 vertices, named, from 6dea8b8:
[1] E D A
```

```
   10) articulation_points(bow_tie)
```

```
+ 1/5 vertex, named, from 6dea8b8:
[1] A
```

If the biconnected components of the Pocket Cube graph coincide with its connected components, that means that there are at least two genuinely different paths to get from any state to any other. Two paths that have no common vertices, except from the source and target vertices of course.

```
pc_comps <- decompose(pc)
vapply(pc_comps, function(g) biconnected_components(g)$no)
```

2.6.2 Strongly connected components

A directed network is called *weakly connected* if its corresponding undirected network that ignored edge directions, is connected. A directed network is *strongly connected*, if and only if it has a directed path from each vertex to all other vertices. The airport network is not weakly connected, so it cannot be strongly connected, either:

```
is_connected(air, mode = "weak")
```

```
[1] FALSE
```

```
is_connected(air, mode = "strong")
```

```
[1] FALSE
```

A directed graph has strongly connected components. These are its maximal induced subgraphs that are strongly connected. Within a strongly connected component, there is a directed path between each ordered pair of vertices. The airport network has 30 strongly connected components:

```
count_components(air, mode = "strong")
```

```
[1] 30
```

```
table(components(air, mode = "strong")$membership)
```

```
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
  1   1   1   1   1   1   1   1   1   1   2   1   2   1   1
 16  17  18  19  20  21  22  23  24  25  26  27  28  29  30
  2   1   1   1   1   1   1   1   1 723   1   1   1   1   1
```

The sizes of the connected components show that most airports are strongly connected, and a few vertices are only connected to the rest of the network via a single uni-directional route.

In terms of strongly connectedness, directed networks typically have a bow-tie like structure, with four parts:

1. The left side of the bow-tie contains the vertices from which all other bow-tie vertices are reachable.
2. The middle part is a large strongly connected component.
3. The right part of the bow-tie contains the vertices that are reachable from all other bow-tie vertices.
4. The rest of the vertices are in very small strongly connected components.

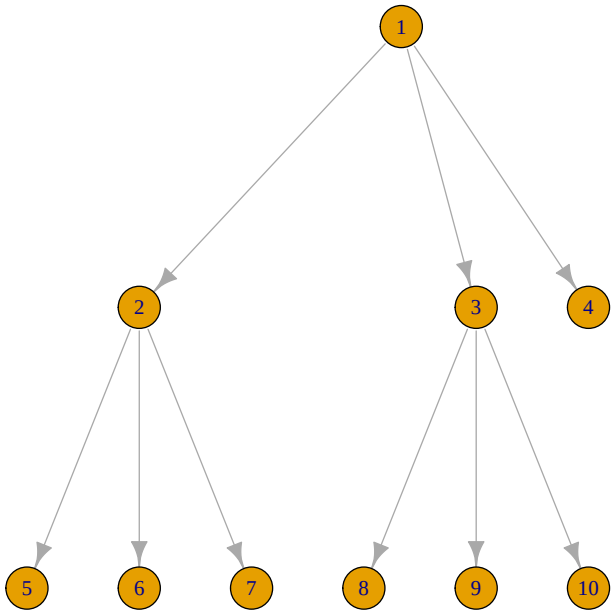
2.7 Trees

An undirected graph is called a tree if there is exactly one simple path between all pairs of its vertices. This also implies for each vertex pair, the shortest path is unique. Tree graphs are important for engineering algorithms and data structures. A forest is a graph if there is *at most* one simple graph between all pairs of its vertices.

Igraph can create regular trees with the `make_tree()` function. In a regular tree the number of incident edges is the same for all vertices, except for the one at the "lower right corner" of the tree:

make_tree()

```
2.38.1) tree3 <- make_tree(10, children = 3) %>%
    2)   add_layout_(as_tree())
    3) plot(tree3)
```



2.7.1 Minimum spanning trees

For each weighted graph G, we can define its minimum spanning tree MST(G). This is a tree that contains the same vertices as G, and a a subset of its edges (i.e. it is a *spanning tree*), such that the sum the edge weighs is less than in any other spanning tree.

Minimum spanning trees are useful in various algorithmic problems, and also in network visualization. Networks are often too dense to visualize, but a minimum spanning tree is always sparse, and often conveys information about the original network. Let's visualize the backbone of the US airport network. Plotting all edges would not lead to an attractive figure, but we can plot a minimum spanning tree (or forest) of the network, the one that corresponds to the most transported passangers. For this we need to set the

edge weights to the opposite of the carried passengers, and then calculate the minimum spanning tree:

```
2.39.1) tree_air <- air
    2) E(tree_air)$weight <- - E(tree_air)$Passengers
    3) air_mst <- mst(tree_air)
    4) # air_plot(air_mst)
```

2.7.2 DAGs

A tree cannot contain cycles, in other words it is an acyclic graph. The analog concept in directed graphs is a directed acyclic graph, a DAG. DAGs are an important class of directed graphs, because of their applications in algorithmic problems. A DAG is often weighted.

Consider the following example. The current version of the *igraph* R package uses some functions from the *Matrix* R package, which means that *Matrix* must be installed on a system before *igraph* is installed. *Matrix* on the other hand uses functions from the *lattice* R package, so lattice must be installed before *Matrix*. The dependency graph of *igraph* is this:

```
2.40.1) igraph_deps <- make_graph( ~ lattice -+ Matrix -+ igraph )
```

Because of technical limitations a package dependency graph must always be a DAG. In other words, it is not possible to have circular package dependencies. When R installs a package, it downloads and installs all its direct and indirect dependencies as well. First it has to order these dependencies such that each package only depends on others that come earlier in the ordering. This is called the *topological ordering* of the graph, and it exists if and only if the graph is a DAG. Once the topological order is determined, the packages can be installed one after the other, according to the ordering. Note that the topological ordering might not be unique.

The `is_dag()` function checks if a directed graph is a DAG. `topo_sort()` gives a topological ordering of the graph. *is_dag()*

```
2.41.1) is_dag(igraph_deps)
```

```
[1] TRUE
```

```
    3) topo_sort(igraph_deps)
```

```
+ 3/3 vertices, named, from 92e980e:
[1] lattice Matrix  igraph
```

Because of the acyclic property, some graph problems can be solved much faster on DAGs. An example is finding shortest and longest paths. First of all, it is true that v comes before w in some topological ordering of a DAG,

then it contains no directed paths from w to v. If just a path existed, that would mean that w must come before v in all topological orderings.

Additionally, the following recursive equation is also true for all vertices:

$$d(s, w_i) = \min_{(w_j, w_i) \in E} \{d(s, w_j) + c(w_j, x_i)\}, \tag{2.1}$$

where $c(w_j, w_i)$ is the weight of the (w_j, w_i) edge, and $d(s, w_i)$ is the distance from s to w_i. We can calculate the distances from a given s to all other vertices in the topological order. This is because when we calculate $d(s, w_i)$ the $d(s, w_j)$ distances were already calculated for all (w_j, w_i) edges, since w_j must come before w_i in the topological ordering.

A common application of DAGs and topological ordering comes from scheduling and planning. If a project is divided into subprojects that depend on each other, and each of which takes possibly different amount of time to complete. It is natural to represent these relationships with a directed graph, which has to be a DAG. A natural question is then the minimum time that is needed to complete the project, if the subprojects can be done by different people or teams, in parallel.

It is not hard to see that the minimum time needed for the project is the longest path in the graph. Finding the longest path in a DAG is closely related to finding the shortest path in it, and all we need to do is replacing $\min \cdot$ in Eq. 2.1 with $\max \cdot$.

2.8 Transitivity

Especially in social network analysis, parts of networks are often analyzed in isolation. An ego-network contains a distinguised vertex (ego), its neighbors and all edges among them, including edges between the neighbors. An ego network is a localized structural summary of the ego vertex. Ego networks are useful in various settings:

- Focused, detailed analysis on the most important vertices of a graph.
- Comparing and classifying roles of vertices, in a supervised or unsupervised way.
- Detecting local changes in network structure, by focusing on the structural properties of all ego networks in a graph.
- Visualization, if visualizing the complete graph is hard or infeasible.

make_ego_graph()

An ego network is embedded into a larger network, although the larger network is sometimes partially or fully unknown. igraph can extract ego networks from via a the `make_ego_graph()` function. It extracts one or more ego networks, and note that it always returns a list of graphs, for consistency:

```
2.43.1) make_ego_graph(air, nodes = c("BOS", "PBI")) %>%
    2)   lapply(summary) %>%
    3)   invisible()
```

```
IGRAPH 090008a DN-- 84 2446 -- US airports
+ attr: name (g/c), name (v/c), City (v/c), Position
| (v/c), Departures (e/n), Seats (e/n), Passengers
| (e/n), width (e/n)
IGRAPH 8b9b765 DN-- 41 757 -- US airports
+ attr: name (g/c), name (v/c), City (v/c), Position
| (v/c), Departures (e/n), Seats (e/n), Passengers
| (e/n), width (e/n)
```

Related functions `ego_size()` and `ego()` calculate the size and the vertices of the ego network(s), respectively. *ego_size()* *ego()*

A generalization of an ego network is called a neighborhood. A k-neighborhood of a vertex contains all vertices within k steps from it, and all edges among them. An ego-network is a special k = 1 neighborhood. k-neighborhoods of vertices can be creates by setting the `order` argument of the `make_ego_graph()` function.

The most basic property of an ego network is its order: the number vertices in it. This of course always equals to the degree of ego, minus one for the ego vertex itself.

Another basic property is the edge density of the ego networks. This is the ratio of the realized edges and the possible edges, and can be calculated with `edge_density()`. *edge_density()*

```
2.44.1) air_deg <- degree(air)
    2) air_ego_dens <- make_ego_graph(air) %>%
    3)   vapply(edge_density, numeric(1))
```

An extremely simple classification of US airports could use these two ego properties:

```
2.45.1) air_clusters0 <- data.frame(stringsAsFactors = FALSE,
    2)             deg = air_deg, dens = air_ego_dens) %>%
    3)   na.omit() %>%
    4)   kmeans(centers = 4)
    5) air_clusters0$size
```

```
[1] 626  31  12  85
```

In the third cluster, we clearly have the large regional hubs. The first cluster contains the opposite, the smallest airports, with typically just a few connections and because of the few connections, a highly variable ego network density. The second and fourth clusters contain the airports in between.

```
2.46.1) which(air_clusters0$cluster == 3)
```

```
LAS LAX CLT DTW MSP PHL MCO IAH ORD ATL DEN DFW
  5  10  37  44  64  71 112 124 131 148 151 152
```

```
4) air_deg[ air_clusters0$cluster == 2 ] %>% summary()
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   93.0   105.5   129.0   131.5   150.0   183.0
```

```
7) air_deg[ air_clusters0$cluster == 4 ] %>% summary()
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   2.00   34.00   45.00   48.69   60.00   85.00
```

Being in an ego-network, the edge density is somewhat constrained because the edges between ego and the alters (non-egos) are always present. The measure that only calculates the density between the alters is called transitivity. Two alters from the same ego network are always connected through a path of length two, through ego. Transitivity measures how often they are also directly connected and form a triangle. The `transitivity()` function calculates transitivity directly on the embedding graph, no need to create the ego networks first:

transitivity()

```
2.49.1) transitivity(air, vids = c("BOS", "LAS"), type = "local")
```

```
[1] 0.3529239 0.2238521
```

```
3) ego_size(air, nodes = c("BOS", "LAS"))
```

```
[1]  84 131
```

The transitivity value for Boston is higher than for Las Vegas, meaning that cities you can fly from to Boston (or the other way around) are more likely to be connected, compared to Las Vegas. Given that the ego network of Las Vegas is bigger, this is not very surprising.

There are various types of transitivity measures. The one we defined here is the *local* transitivity of the egos in a network: the ratio of closed and open triangles centered on ego. The global transitivity (or clustering coefficient) simply measures this for the whole network, i.e. it is the ratio of the number of triangles and the number of connected triples.

```
2.51.1) transitivity(air, type = "global")
```

```
[1] 0.3384609
```

The global transitivity is a crude summary of the structure of a network. A (relatively) high value means that vertices are connected well *locally*, in other words the network has dense subgraphs. Network data sets typically show high transitivity.

2.9 Maximum flows and minimum cuts

The `Seats` edge attribute in the `air` network shows the number of seats on all flights between a departure and a destination airport, per month:

```
2.52.1) E(air)[[ "BOS" %->% "JFK" ]]
```

```
+ 1/8228 edge from ce25b00 (vertex names):
   tail head tid hid Departures Seats Passengers width
12  BOS  JFK   2   4        491 39403      31426     0
```

Considering non-direct flights, this number is of course much higher. In this section we calculate how many passengers the US airport network can transport from a given airport to another one, without introducing new flights or changing existing ones.

Questions about moving goods or people from one vertex to another, through the network, naturally translate to maximum flow problems. A flow between a *source* and a *sink* vertex on a (potentially weighted) directed graph is a mapping of the edges to non-negative numbers, such that at every vertex the in-flow equals the out-flow, except for the source vertex, where the in-flow is zero, and the sink vertex, where the out-flow is zero. The *value of the flow* is the in-flow at the sink vertex. The maximum flow from source to sink is the flow with the largest value.

In the airports example, the edges of the network are labeled by the number of passengers traveling over them, during one month. The value of the flow is the number of passengers transported to the sink vertex corresponding to the destination airport. For maximum flow problems wdge weights are considered to be edge capacities, i.e. the number of passengers that can be transported along that edge. It is important to realize that with our maximum flow model we already made some implicit assumptions:

- We ignored the actual departure and arrival times. The existance of a flight from `A` to `B`, and another one from `B` to `C` does not imply that passengers can fly from `A` to `C` in practice. The second flight might leave before the arrival of the first, or the second flight might be a weekly flight and passengers will not wait for days to get on it. So our model will gives un an upper bound on the number of passengers that can be transported.
- We do not consider that some flights are missing from our data because they were canceled. Other flights were diverted, so their arrival airport is incorrect.
- When calculating a maximum flow from a departure to a destination, we ignore that passengers also travel between other airports and interfere with the flights along out maximum flow.

We assume that these assumptions are reasonable if we are only interested in the general throughput of the airport system, especially when comparing different networks or the same network over time.

The maximum flow problem is algorithmically not very difficult, but it is also not easy. There are many algorithms for it, one of the most popular ones is the push-relabel algorithm by Goldberg and Tarjan. Very superficially the algorithm tries to assign height labels to the vertices, so that the flow along the edges is always going down or at least never up. Then it pushes *down* flow from the source towards the sink. It iteratively relabels the vertex heights, as it discovers the structure of the network.

max_flow()

The push-relabel algorithm is implemented in the `max_flow()` function. It requires the `source` and `target` vertices. The edge attribute `capacity` is used automatically if present in the graph, otherwise all edges are assumed to have unit capacity:

```
2.53.1) E(air)$capacity <- E(air)$Seats
    2) max_flow(air, 'BOS', 'PBI')$value
```

```
[1] 337730
```

```
    4) max_flow(air, 'BOS', 'LAX')$value
```

```
[1] 1218036
```

The value of the maximum flow is returned in the `value` entry of the result list, and the `flow` entry gives the actual flow values for each edge.

Minimum network cuts are strongly related to maximum flows. First we motivate them with an example. In a transportation network it is often important to know how much the delivery of goods or passengers from one place to another relies on a single or a small number of roads or flights. Formally, we seek for the minimum number of edges, that disconnect a destination vertex from a departure vertex. This is called the minimum cut between the vertices. In a weighted network with edge capacities the minimum cut calculates the total capacity needed to disconnect the vertex pair.

It is easy to show that the minimum cut in a graph from a source vertex to a target vertex always equals the maximum flow between the same vertices. This is often called the *max-flow min-cut theorem*, and was proven in 1956, independently by two teams of researchers: Elias, Feinstein, and Shannon, and by Ford and Fulkerson.

min_cut()

The igraph function `min_cut()` calculates a minimum cut. Internally it uses the same machinery as `max_flow()`, not very surprisingly. It also uses the `capacity` edge attribute, if available. By default it only returns the value of the minimum cut:

```
2.55.1) min_cut(air, 'BOS', 'LAX')
```

```
[1] 1218036
```

We are in this case interested in the number of edges to be removed to disconnect `LAX` from `BOS`, instead of the total capacity of these edges, so we ignore edge capacities:

```
2.56.1) min_cut(air, 'BOS', 'LAX', capacity = rep(1, gsize(air)))
```

```
[1] 74
```

A minimum cut (or any cut, really) can be also viewed as the most natural separation of a connected graph into two partitions. It is most natural, because it gives the partitions that are the least connected. The `min_cut()` and `max_flow()` functions also calculate these partitions.

After learning about minimum cuts a natural question is whether there is a similar notion for vertices. We can define a (minimum) vertex cut, the smallest set of vertices that need to be removed to separate two other vertices in the network.

2.9.1 Cohesive blocks

2.10 Exercises

▶ EXERCISE 2.1. Write a function that calculates the diameter of a graph. (Without calling the `diameter()` function directly, of course.) Can your implementation avoid storing all n^2 (or $n(n-1)/2$ for an undirected graph) distances? Can you make your algorithm parallel? (Hint: look at the *`parallel`* package and the `parLapply()` function.)

▶ EXERCISE 2.2. A is a path that visits each vertex of a graph exactly once. Solve the following puzzle by writing a function that finds the Hamiltonian path in the graph: write out all integer numbers from 1 to 17 in a line, such that the sum of every consecutive pair is a square number. Show that you cannot do this for the numbers between 1 and 18. (Hints: 1) the `outer()` R function can help you create the graphs, 2) for this size of graphs, you can use the brute-force algorithm to search for a Hamiltonian path. I.e. query *all* simple paths between all pairs of vertices using `all_simple_paths()`.) *`http://stackoverflow.com/questions/10150161`*

Hamiltonian path

`all_simple_paths()`

▶ EXERCISE 2.3. Write a program that lists the steps of a Knight over a chess board, such that it visits each square of the (empty) board exactly once. Can you find a solution from each possible starting square?

▶ EXERCISE 2.4. Write a function that extracts the largest component from a graph. Thinks about the case when the graph has multiple largest components.

▶ EXERCISE 2.5. Find a graph that has a non-unique diameter.

▶ EXERCISE 2.6. Let us consider an undirected, connected graph. Write a program that shows that the greedy diameter finding algorithm, as shown below, is incorrect. This naive algorithm tries to find a pair of vertices v, and w, such that w is the farthest vertex from v, and v is the farthest vertex from w. This property, however, does not ensure that the $v-w$ path is a diameter of the graph.

Algorithm 2.1. An incorrect algorithm to calculate the diameter.

Require: $G = (V, E)$ is a connected, undirected graph.
(1) $v :\in V$ {Choose an arbitrary vertex.}
(2) **repeat**
(3) $\quad v' \leftarrow v$
(4) $\quad w \leftarrow \text{farthest\_from}(v')$
(5) $\quad v \leftarrow \text{farthest\_from}(w)$
(6) **until** $v = v'$
(7) **return** distance(v, w)

▶ EXERCISE 2.7. Write a function that decides if a graph is tree. Write another function that decides id a graph is a forest. .

▶ EXERCISE 2.8. Write a function that calculates the bow-tie structure of a directed graph. Hint: in addition to the `components()` function, you'll probably need the `neighborhood()` function as well.

▶ EXERCISE 2.9. Reproduce the figure from the Watts-Strogatz paper.

▶ EXERCISE 2.10. Implement an algorithm to find an arbitrary cycle base of an undirected, simple graph. A cycle base is a set of cycles such that any other cycle not in the cycle base can be obtained by taking the disjoint union of appropriately selected base cycles.

▶ EXERCISE 2.11. This is a problem from a German programming compatition for high school students. Twin towns all over Europe celebrate their partnership this year. The partnerships are given as an undirected graph, in the `'twintowns'` data set in the *`igraphbook`* package. Each town has a number of festivals it is allowed to organize (see the `'Budget'` vertex attribute). Is it possible to distribute the festivals among the twin-towns in a way, such that each pair of twin-towns organizes one festival in one the two towns and no town organizes more festivals than it is allowed to?

The problem can be solved by crafting an augmented network from the input, and calculating the maximum flow from a newly added *source* vertex to a newly added *sink* vertex. Create the augmented network and calculate the maximum flow that answers the question. (Hint: each vertex and each

edge in the input graph will be a vertex in the augmented graph.) *(30. Bundeswettbewerb Informatik, First round, Problem 5.)*

▶ EXERCISE 2.12. Find the connected graph of n vertices with the largest mean distance. (Hard.)

▶ EXERCISE 2.13. Find the graph on n vertices and with m edges, that has the smallest possible diameter. (Hard.)

Chapter 3
Visualization

3.1 Introduction

Graphs are abstract mathematical structures. Formally, a graph is nothing else but a set of objects where some pairs of the objects are connected by links, and such objects have no unique visual representation which one could simply draw or print to a piece of paper. The figures we have seen in this book do not show graphs; they show an arbitrarily chosen visual representation corresponding to these graphs. Graph visualization is the art of choosing an appropriate visual representation to an arbitrary graph that is aesthetically pleasing and makes the most important structural properties of the graph accessible to the viewer.

The most common visual representation of a graph, which we have used throughout the book and with which igraph works, is the so-called *node-edge diagram*. Node-edge diagrams assign the vertices to points in the two- or three-dimensional space, and connect adjacent nodes by straight lines or curves. For directed graphs, arrowheads on the lines may be used to indicate the directionality of connections. Vertices are then drawn on top the edges using simple geometric shapes, and the most important attributes of vertices and edges are assigned to visual properties of the shapes and lines; for instance, one can make the area of a circle representing a vertex proportional to the degree of the vertex in order to highlight hubs (i.e. highly connected nodes). Therefore, making a graph visualization usually consists of three distinct steps:

node-edge diagram

1. Finding an appropriate arrangement of the vertices in the 2D or 3D space. This is probably the most important one: a good layout often reveals interesting symmetries or densely connected regions among the vertices which would otherwise remain hidden. For instance, the three panels of Fig. 3.1 all show the famous Petersen graph (Holton and Sheehan, 1993), but the random layout on the left does not reveal anything about its inner structure.

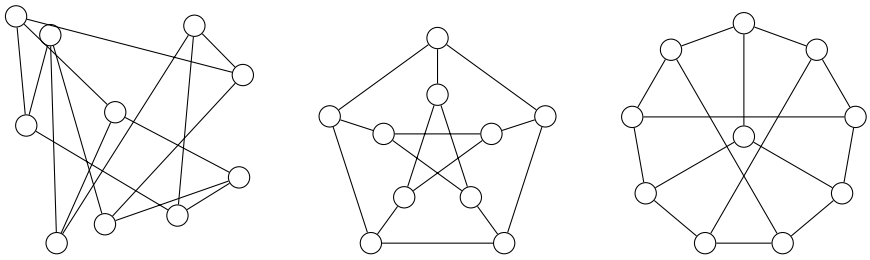


Fig. 3.1 Three layouts of the Petersen graph. The left panel shows a random layout that does not reveal anything about the inner structure of the graph. The middle and right panels show two different layouts that reveal some of the symmetries and highlight the importance of choosing a proper layout.

2. Mapping the important attributes of vertices and edges to visual properties of the corresponding shapes (for vertices) and lines (for edges).
3. Drawing the edges and vertices in an appropriate order.

igraph provides tools for all the above steps. These tools will be our primary interest in this chapter.

3.2 Preliminaries

3.3 Layout algorithms

Layout algorithms are responsible for finding a suitable arrangement of the vertices of a graph in the 2D or 3D space that is aesthetically pleasing and informative. Since neither of these requirements can be formalized easily, most graph layout algorithms use some kind of an indirect measure (such as the number of edge crossings) to assess the quality of a given arrangement, and apply a heuristic procedure that does not even guarantee to minimize the chosen quality or to work well in all the cases. Nevertheless, these algorithms usually work well with smaller graphs consisting of a few hundred vertices at most, and specialized techniques exist for handling larger graphs.

Layout algorithms in igraph are all implemented as methods of the *Graph* class. The names of these methods always start with *layout_* (e.g., *layout_circle()*, *layout_drl()* and so on). There is also a central entry method called *layout()* which takes the name of a layout algorithm as the first argument, then calls the appropriate layout method and forwards all the remaining positional and keyword argument to the called method.

layout_circle()
layout_drl()
layout()

Each layout method returns an instance of *Layout*, a list-like object that specifies the X and Y coordinates for each vertex in the graph. *Layout* instances also provide some convenience method to translate or rotate a layout

Layout

or to fit it in a given bounding box. Let us experiment a little bit with the *Layout* class using probably the least sophisticated layout algorithm, the random layout.

The random layout, as its name suggests, simply places the vertices in random positions in the unit square. It is implemented by the *layout_random()* method, but can also be invoked by passing "*random*" as the first argument of *layout()*. Throughout this chapter, we will use the second syntax, i.e. we will call the *layout()* method and specify the name of the layout algorithm as the first argument:

layout_random()

```
3.1.1) petersen = Graph.Famous("petersen")
   2) layout = petersen.layout("random")
```

Layout objects behave like lists; they have a length (the number of vertices), and they can be indexed, iterated and sliced just like a regular Python list. They even have an *append()* method to add new coordinates to the layout:

```
3.2.1) len(layout)
   2) layout[2]
   3) layout.append((5, 7))
   4) len(layout)
   5) del layout[10]
   6) for x, y in layout:
   7)     print "x=%7.4f, y=%7.4f" % (x, y)
```

The number of dimensions in the layout can be queried with the *dim* readonly property:

```
3.3.1) layout.dim
```

You can easily calculate the centroid (i.e. center of mass) of a layout using the *centroid()* method and move the centroid to a given point in the coordinate system by calling *center()*:

centroid()
center()

```
3.4.1) layout.centroid()
   2) layout.center(1, 2)
   3) layout.centroid()
```

The bounding box of a layout (i.e. the smallest box that contains all the points in the layout) is returned by the *bounding_box()* method. The result of this call is an instance of *BoundingBox*, another convenience class to represent rectangles and their most common operations:

bounding_box()
BoundingBox

```
3.5.1) bbox = layout.bounding_box()
   2) print bbox
   3) print bbox.width, bbox.height
   4) print bbox.left, bbox.right, bbox.top, bbox.bottom
   5) print bbox.midx, bbox.midy
```

Property	*Access*	*Meaning*
`bottom`	read/write	The Y coordinate of the bottom edge
`coords`	read/write	The coordinates of the upper left and lower right corners as a 4-tuple
`height`	read/write	The height of the box. When setting it, the bottom edge of the box is adjusted.
`left`	read/write	The X coordinate of the left edge
`midx`	read/write	The X coordinate of the center
`midy`	read/write	The Y coordinate of the center
`right`	read/write	The X coordinate of the right edge
`shape`	read/write	The shape of the bounding box, i.e. a tuple containing the width and the height. When setting it, the bottom and right edges are adjusted.
`top`	read/write	The Y coordinate of the top edge
`width`	read/write	The width of the box. When setting it, the right edge of the box is adjusted.

Table 3.1 Properties of the `BoundingBox` class

Name	*Meaning*
`contract()`	Contracts the box by the given margin (either a single number or a list of four numbers, referring to the left, top, right and bottom edges).
`expand()`	The opposite of `contract()`: expands the box by the given margin.
`isdisjoint()`	Returns whether two boxes are disjoint or not.
`intersection()`	Calculates the intersection of two boxes, and returns a new `BoundingBox`.
`union()`	Calculates the smallest bounding box that contains both boxes, and returns a new `BoundingBox`.

Table 3.2 Methods of the `BoundingBox` class

The members of `BoundingBox` are summarized in Tables 3.1 and 3.2. You can also ask the layout to fit the coordinates into a given bounding box, keeping the aspect ratio of the layout by default:

```
3.6.1) new_bbox = BoundingBox(-1, -1, 1, 1)
   2) layout.fit_into(new_bbox)
   3) layout.bounding_box()
```

If you do not want to keep the aspect ratio, just pass `keep_aspect_ratio=False` to `fit_into()`:

```
3.7.1) layout.fit_into(new_bbox, keep_aspect_ratio=False)
   2) layout.bounding_box()
```

`translate()` Finally, we also mention the methods of `layout` that perform simple coordinate transformations: `translate()` translates the layout with a given

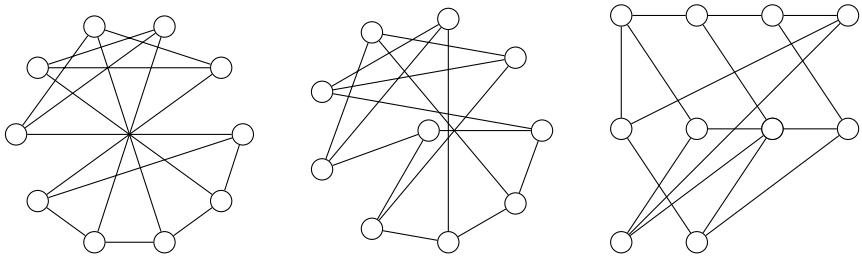
vector, *rotate()* rotates the layout by a given angle (specified in degrees), *scale()* scales the layout by given scaling factors along the axes, and *transform()* calls a user-defined function on each of the points in the layout. *rotate()* and *scale()* also allows the user to specify the origin of the transformation which will stay in place. In the following examples, we will track the effect of the transformations on the centroid of the layout:

rotate()
scale()
transform()

```
3.8.1) layout.center(1, 2)
   2) layout.centroid()
   3) layout.translate(2, -2)           ## shift 2 units left and 2 units down
   4) layout.centroid()
   5) layout.scale(2, origin=(2, 0))    ## 2x scaling around (2, 0)
   6) layout.centroid()
   7) layout.rotate(45, origin=(4, 0))  ## rotating by 45 degrees
   8) layout.centroid()
```

3.3.1 Simple deterministic layouts

The layout algorithms to be introduced in this section are *deterministic*, i.e. they always produce the same layout for the same graph. To be frank, they do not even look at the structure of the graph but produce a static layout based on the number of vertices in the graph and some general guidelines. On the other hand, they are fast and they may be useful for some graphs with a special structure that is known *a priori*.



(a) Circle layout (b) Star layout (c) Grid layout

3.3.1.1 Circle and sphere layout

The *circle and sphere layouts* place the vertices on a circle (in 2D) or on the surface of a sphere (in 3D) in a way that tries to equalize the distances

circle layout
sphere layout

between vertices. The center of the circle (or sphere) is always in the origin of the coordinate system, and the radius is equal to one:

```
3.9.1) layout = petersen.layout("circle")
    2) print layout
    3) layout.centroid()
    4) layout.bounding_box()
    5)
    6) layout = petersen.layout("sphere")
    7) print layout
    8) layout.centroid()
```

Figure 3.2(a) shows the Petersen graph when laid out in a circle. You can also create such a figure in igraph by using the *plot()* function:

plot()

```
3.10.1) plot(petersen, layout=layout)
```

The *plot()* function is the main function for plotting graphs in igraph. A more detailed description will be given later in Section 3.4.1.

3.3.1.2 Star layout

star layout

The *star layout* is similar to the circle layout, but one of the vertices is placed in the center of the circle. For star-like graphs, this layout will indeed draw the graph in a way that resembles a star, hence its name. The layout lets you specify the ID of the vertex that will be placed in the center and also the order of vertices on the circumference of the circle. This can sometimes influence the quality of the layout significantly. The default parameter settings of the star layout place vertex 0 in the center and the remaining vertices around it in increasing order of their vertex IDs, which does not reveal much about the structure of the Petersen graph (see also Figure 3.2(b)):

```
3.11.1) layout = petersen.layout("star")
    2) plot(petersen, layout=layout)
```

However, with some *a priori* information (namely by knowing that the Petersen graph contains a cycle of length 9), one can specify the order of the vertices in a way that produces a better layout, similarly to the one we have seen already on Figure 3.1(c).

```
3.12.1) layout = petersen.layout("star", center=0,                    order=[0,1,2,3,4,9,7,5,8,
    2) plot(petersen, layout=layout)
```

Note that the *order* parameter must include each vertex ID only once, including the ID of the central vertex.

3.3.1.3 Grid layout

grid layout

The *grid layout* places the vertices on an $m \times k$ regular grid, where m denotes the number of rows and k denotes the number of columns. The default values are $k = \lceil \sqrt{n} \rceil$ and $m = \lceil n/k \rceil$, where n is the number of vertices in the graph. If n is a square number, these settings will yield a perfect square layout, and even if n is not a square number, igraph will strive to make the layout close to a perfect square, but of course a few slots in the last row will be empty. This can be seen on Figure 3.2(c) where $k = 4$ and $m = 3$, and the two rightmost places in the last row are empty. The code which generates this layout is as follows:

```
3.13.1) layout = petersen.layout("grid")
    2) plot(petersen, layout=layout)
```

You can override the width of the layout (i.e. the number of vertices in a row) using the *width* parameter:

```
3.14.1) layout = petersen.layout("grid", width=5)
    2) for coords in layout:
    3)     print "x=%d, y=%d" % tuple(coords)
```

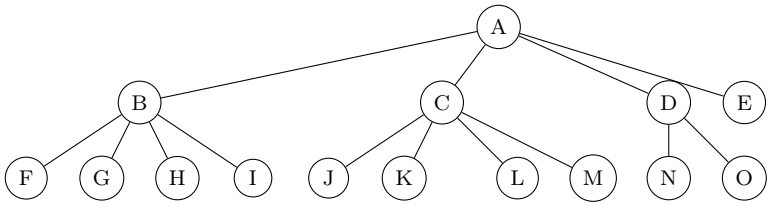
There is also a 3D variant of the grid layout, which can be invoked either by using "*grid_3d*" as the name of the layout, or by passing *dim=3* as a keyword argument to *layout()*. You may override both the width (the number of vertices in a row) and the height (the number of rows) of a 3D grid layout using the *width* and *height* keywords arguments, respectively, and igraph will calculate the depth of the layout (i.e. the number of layers along the Z axis) automatically.

```
3.15.1) layout = petersen.layout("grid", dim=3)
    2) for coords in layout:
    3)     print "x=%d, y=%d, z=%d" % tuple(coords)
```

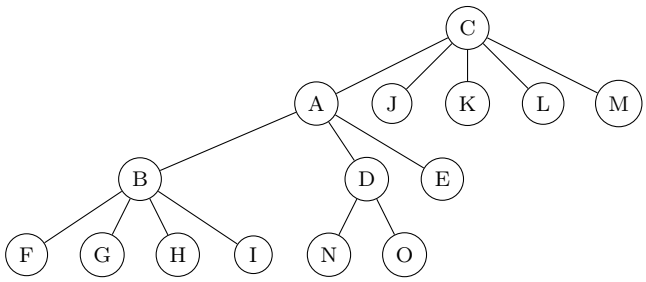
3.3.2 Layout for trees

Reingold–Tilford layout

The *Reingold–Tilford layout* (Reingold and Tilford, 1981) is a layout algorithm designed specially for drawing tree graphs – graphs that contain no cycles. It is selected by using *reingold_tilford*, *rt* or *tree* as the name of the layout algorithm in the call to the *layout()* method. The algorithm generates a layout where vertices are organized into layers based on their geodesic distance from a chosen *root vertex*. It also strives to minimize the number of edge crossings and to make the layout as narrow as possible. Even if the graph is not a tree (i.e. it contains cycles), the layout will select an



(d) Regular tree



(e) With different root vertex

Fig. 3.2 Reingold–Tilford layout of a regular tree with different root vertices.

arbitrary spanning tree from the root vertex and use the spanning tree to determine the layout. Needless to say, the more cycles you have in your graph, the worse layouts the algorithm will produce, therefore it is most useful for graphs that are trees or almost trees with a few extra edges. Since the Petersen graph we have used until now is far from being a tree, we will construct a regular tree graph first, with an incomplete lowermost layer, then calculate a Reingold–Tilford layout with vertex 0 as the root:

```
3.16.1) tree = Graph.Tree(15, 4)
    2) layout = tree.layout("reingold_tilford", root=0)
```

We can now plot the tree, which results in a plot similar to Figure 3.2(d).

```
3.17.1) plot(tree, layout=layout)
```

If you do not specify a root vertex when calling the layout method, igraph will try to select one based on some simple criteria. For undirected graphs, the root of the tree will be the vertex with the largest degree. For directed graphs, you may specify whether the tree is oriented downwards or upwards by using the *mode* argument (where "*out*" means a downward oriented tree and "*in*" means the opposite), and the root will be a vertex with no incoming edges for downward trees and a vertex with no outgoing edges for upward trees. If the graph consists of multiple components, igraph will select a root vertex for each component. This heuristic is not perfect; for instance, it will select a vertex from the second layer of our tree graph, as seen on Figure 3.2(e):

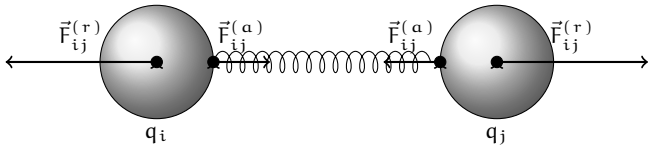


Fig. 3.3 Basic principles behind force-directed graph layout algorithms. q_i and q_j represent the charges of nodes i and j, $\vec{F}_{ij}^{(r)}$ is the repulsive force as given by Coulomb's law, $\vec{F}_{ij}^{(a)}$ is the attractive force the spring exerts on the nodes according to Hooke's law.

```
3.18.1) layout = tree.layout("reingold_tilford")
    2) plot(tree, layout=layout)
```

3.3.3 Force-directed layouts

Force-directed layouts are a large class of layout algorithms that are suitable for general, small to medium sized graphs. They are based on simple physical analogies and do not rely on any assumptions about the structure of the graph, but they can be computationally demanding and do not scale up to graphs with many thousands of vertices. The time complexity of these algorithms is usually $O(n^2h)$, where n is the number of vertices and h is the number of iterations the algorithm takes until convergence.

Force-directed techniques strive to satisfy the following, generally accepted aesthetic criteria (Fruchterman and Reingold, 1991):

1. Vertices are distributed roughly evenly on the plane.
2. The number of edge crossings is minimized.
3. The lengths of edges are approximately uniform.
4. Inherent symmetries in the graph are respected, i.e. subgraphs with similar internal structures are usually laid out in a similar manner.

igraph implements three force-directed techniques: the *Fruchterman–Reingold algorithm* (Fruchterman and Reingold, 1991), the *Kamada–Kawai algorithm* (Kamada and Kawai, 1989) and the *GraphOpt algorithm* (Schmuhl, 2003). The Fruchterman–Reingold and the Kamada–Kawai algorithms exist in 2D and 3D variants. We will discuss the 2D variants only, the 3D versions can be accessed by adding an extra `dim=3` keyword argument to the `layout()` call.

Fruchterman–Reingold algorithm
Kamada–Kawai algorithm
GraphOpt algorithm

The basic assumption behind force-directed algorithms is that nodes are tiny particles with a given electrical charge, and the edges connecting them can be modeled with springs (see Figure 3.3). All pairs of nodes repel each other, and the magnitude of the repulsive force $F_{ij}^{(r)}$ acting between nodes i and j is given by Coulomb's law:

$$\left|\left|F_{ij}^{(r)}\right|\right| = k_e \frac{q_i q_j}{\|\vec{x}_i - \vec{x}_j\|^2} \tag{3.1}$$

where q_i and q_j are the charges of node i and j, $\vec{x}_i$ and $\vec{x}_j$ are the positions of the nodes involved, $\|\vec{v}\|$ is the length of $\vec{v}$ and k_e is the Coulomb constant. Since the nodes all have the same charge, we can simply treat $k_e q_i q_j$ as a constant multiplicative factor for all vertex pairs, therefore the magnitude of the repulsive force simply depends inversely on the squared distance of the nodes involved. $k_e q_i q_j$ is considered as a parameter of the layout algorithm.

We would not get much of a meaningful layout with only repulsive forces. These forces are counterbalanced by the springs attached to pairs of nodes that are connected in the original graph. Attractive spring forces are modeled with Hooke's law:

$$\left|\left|F_{ij}^{(a)}\right|\right| = -k(\|\vec{x}_i - \vec{x}_j\| - l_{ij}) \tag{3.2}$$

where l_{ij} is the equilibrium length of the spring between vertices i and j and k is the spring constant. k can be considered as a parameter of the algorithm just like $k_e q_i q_j$ in Coulomb's law, and it is also assumed that all the springs between vertices have an equal equilibrium length l.

Although it is said that the force-directed algorithms are based on physical analogies and they simulate a physical system in order to determine the final layout, this is not entirely correct – the analogies end somewhere around where we are now. The most important difference between an actual, accurate physical simulation and the simulation employed by these techniques is that forces adjust the positions of the points directly, while in a real simulation, forces would influence velocities, which, in turn, would influence the positions. The reason for this difference is that the accurate simulation leads to dynamical equilibria (i.e. pendulums and orbits), while the layout algorithms seek a static equilibrium.

GraphOpt algorithm

The three techniques take a different turn here. The *GraphOpt algorithm* (igraph name: `"graphopt"`) simply simulates the system for a given number of time steps, and adds a simulated friction force to prevent nodes from moving "too far" in one step; this is motivated by the fact that the algorithm was designed for continuous graph visualizations where the layout unfolds gradually in front of the viewer's eyes.

Fruchterman–Reingold algorithm

The *Fruchterman–Reingold algorithm* (`"fr"` or `"fruchterman_reingold"`) also constraints the displacement of vertices in each time step, but the maximum allowed displacement starts from a large value and is gradually decreased in each time step, therefore the vertices are allowed to make large jumps in the early stages of the algorithm, but the final stages allow small refinements only. The desired spring length l is zero, and the algorithm naturally terminates when the displacement limit becomes smaller than a given threshold.

Kamada–Kawai algorithm

The *Kamada–Kawai algorithm* (`"kk"` or `"kamada_kawai"`) does not use node charges, but the desired spring length is non-zero to keep the vertices from collapsing into the same point. The algorithm then tries to minimize

the total energy of the graph, where the energy is defined as follows:

$$\sum_{(i,j)\in E(G)} (\|\vec{x}_i - \vec{x}_j\| - l)^2 \tag{3.3}$$

where $E(G)$ is the set of edges of a graph G. Instead of running the simulation, the Kamada–Kawai algorithm searches for configurations with low stress using a simulated annealing algorithm. For each vertex, a new position is proposed in each step. The proposed displacement of the vertex is drawn from a normal distribution with zero mean and $\sigma^2(t)$ variance (where t is the number of iterations so far). The proposal is then accepted or rejected; proposals that reduce the stress are always accepted, while proposals that increase the stress are accepted only with a probability that depends on t and the difference between the new and old stress values and the number of iterations so far. In the early stages of the procedure, the variance of the proposed displacements is larger, and proposals that lead to worse configurations are accepted with a relatively high probability, while later stages allow only small displacements and reject almost all the proposals that would increase the stress.

The parameters of the above mentioned layout algorithms are summarized in Tables 3.3, 3.4 and 3.5.

It is important to note that although these layouts seem to be deterministic (after all, each step involves calculating the displacement of nodes in a deterministic manner), the starting configuration is completely randomized, thus every invocation of these algorithms lead to a completely different layout. If you need exactly the same layout for multiple invocations, you have to set the seed value of the random number generator of Python to a given constant, which guarantees that the same starting configuration is used. Alternatively, you can use the *seed* keyword argument to specify the starting configuration explicitly. This also allows one to refine the result of a force-directed layout algorithm with another one. For instance, it is generally claimed that the best results are obtained by using the Kamada–Kawai layout algorithm to find an approximate placement of nodes, followed by the Fruchterman–Reingold layout algorithm to refine the positions.

geometric random graphs

Let us experiment a bit with these algorithms on *geometric random graphs*, which are generated by dropping n vertices into the unit square $[0;1]^2$, and connecting all pairs of vertices that are closer to each other than a predefined distance threshold r (see Section ?? for more details). We will use $n = 25$ and $r = 0.5$:

3.19.1)
```
graph = Graph.GRG(25, 0.5)
```

Note that the geometric random graphs have a "pre-defined" layout on the 2D plane, where the vertices are placed at the exact positions where they were when the graph was generated. The original X and Y coordinates

Parameter	Meaning
weights	Weights of each edge in the layout. It can be an iterable yielding floating point values or the name of an edge attribute. Springs corresponding to edges with larger weights are stronger, making the edges shorter. All edge have equal strength if this is omitted.
maxiter	The maximum number of iterations to perform; the default is 500.
maxdelta	The maximum distance a vertex is allowed to move in the first step. The default is n, the number of vertices. The displacement limit in step i is equal to $\delta(i/n)^{\alpha}$, where δ is the value of *maxdelta*, and α is the value of *coolexp*.
area	The area of the square in which the vertices are placed before the first step. The default is n^2.
coolexp	The cooling exponent of the simulated annealing process that decreases the maximum allowed vertex displacements. The default is 1.5; see *maxdelta* for more information.
repulserad	A constant specifying the distance where the repulsive forces of two nodes cancel out the attractive forces of a spring between them. This constant controls the relation between k_e (the Coulomb constant), q_i (the node charges) and k (the spring constant). The default is n^3.
miny, *maxy*	Specifies the minimum and maximum Y coordinates for each vertex in the layout. They must either be lists containing one Y coordinate for each vertex, or *None*, which means that no constraint is enforced. The default is of course *None*.
seed	The initial layout from which the simulation will start. The default is *None*, meaning a random layout.
dim	The number of dimensions in the space where the vertices will be laid out. It must be either 2 or 3, the default being 2.

Table 3.3 Parameters of the Fruchterman–Reingold layout algorithm ("*fr*" or "*fruchterman_reingold*")

are assigned to the *x* and *y* vertex attributes, respectively, so we can easily reconstruct this layout using the attribute values:

```
3.20.1) layout_orig = Layout(zip(graph.vs["x"], graph.vs["y"]))
```

We can also generate layouts for the same graph using the three different force-directed layout algorithms using their default parameters, and compare them with *layout_orig*:

```
3.21.1) layout_fr = graph.layout("fruchterman_reingold")
    2) layout_kk = graph.layout("kamada_kawai")
    3) layout_graphopt = graph.layout("graphopt")
    4) plot(graph, layout_orig)
    5) plot(graph, layout_fr)
    6) plot(graph, layout_kk)
    7) plot(graph, layout_graphopt)
```

Parameter	Meaning
maxiter	The maximum number of iterations to perform; the default is 1000.
sigma	The standard deviation of the position change proposals in the first step. The default is $n/4$, where n is the number of vertices.
initemp	Initial temperature of the simulated annealing; the default is 10.
coolexp	The cooling exponent of the simulated annealing process. The temperature is multiplied by this value in each step. The default is 0.99.
kkconst	The Kamada–Kawai vertex attraction constant. The default is n^2.
seed	The initial layout from which the simulation will start. The default is *None*, meaning a random layout.
dim	The number of dimensions in the space where the vertices will be laid out. It must be either 2 or 3, the default being 2.

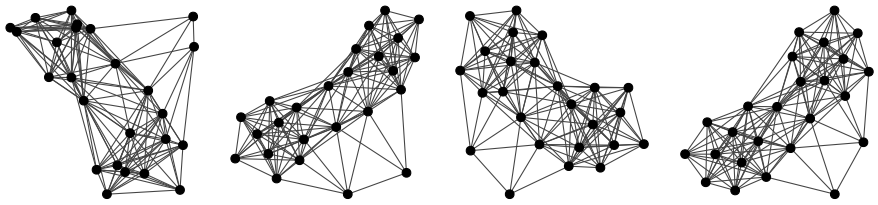
Table 3.4 Parameters of the Kamada–Kawai layout algorithm ("*kk*" or "*kamada_kawai*")

Parameter	Meaning
niter	The maximum number of iterations to perform; the default is 500.
node_charge	The charge of the vertices, used to calculate the repulsive forces. The default is 0.001.
node_mass	The mass of each vertex; the calculated forces are divided by this number in the layout algorithm to obtain the displacement. The default is 30.
spring_length	The equilibrium length l of the springs; the default is zero.
spring_constant	The spring constant k; the default is 1.
max_sa_movement	The maximum distance a vertex is allowed to move *along a single axis* in each step. The default is 5 units.
seed	The initial layout from which the simulation will start. The default is *None*, meaning a random layout.

Table 3.5 Parameters of the GraphOpt layout algorithm ("*graphopt*")

The results are shown on Figure 3.4; of course your results are likely to be different due to the randomized nature of both the graph generation method and the layout algorithms, but the general conclusions are the same: all the three algorithms managed to produce an aesthetically pleasing layout of the original graph while also recovering the most important structural properties: the two denser clusters of vertices and the presence of two extra nodes that are connected only loosely to the rest of the graph.

Finally, it should be noted that force-directed layout algorithms work well on *connected* graphs, but their behaviour may be strange if the graph is disconnected. The components of a disconnected graph repel each other in both the Fruchterman–Reingold and the GraphOpt layout. For the GraphOpt layout, the final distance between the individual components will depend on the number of iterations taken; the more steps you take, the farther the components will be. The Fruchterman–Reingold layout counteracts this effect by adding an extra, invisible node that is anchored to the origin of the coordi-



(a) Original layout (b) Fruchterman–Reingold (c) Kamada–Kawai (d) GraphOpt

Fig. 3.4 Comparison of force-directed layout algorithms on a geometric random graph. The left panel shows the positions of the vertices that were used to generate the graph. The remaining panels show layouts generated by different force-directed layout methods.

nate system, and is also connected to one randomly selected node from each of the components. The phantom edges between the invisible node and the components keep the layout together to some extent. For the Kamada–Kawai layout, there is no repulsive force between vertices in different components, therefore all the components will be laid out in the same area on top of each other. It is thus generally advised to use force-directed layout algorithms directly on connected graphs only; disconnected graphs should first be decomposed into connected components using the `clusters()` and `subgraph()` methods of the `Graph` class, the components should be laid out individually, and the layouts should be merged in a post-processing step.

clusters()
subgraph()

3.3.4 Multidimensional scaling

Multidimensional scaling (MDS) is a name for a set of related statistical techniques to explore dissimilarity relationships encoded in a dissimilarity matrix. More precisely, classical multidimensional scaling (also known as Torgerson–Gower scaling) takes a symmetric dissimilarity matrix of size $n \times n$ between n objects, and assigns a point in the k-dimensional space for each of the n objects such that the distances between points are as close to the distance matrix as possible (Borg and Groenen, 2005). This is done by minimizing a loss function called *strain*:

strain

$$E = \sum_{i=1}^{n} \sum_{j=i+1}^{n} (\|\vec{x}_i - \vec{x}_j\| - d_{ij})^2 \tag{3.4}$$

where d_{ij} is the desired distance of objects i and j according to the distance matrix $\mathbf{D}$.

The strain is surprisingly similar to the energy function used in the Kamada–Kawai layout (see Eq. 3.3), with two key differences: first, the dis-

tances are not uniform; second, the sum goes over all (i, j) pairs, not only those that are connected in the graph. As a consequence, the distance matrix must be specified for disconnected and connected vertex pairs alike if we want to use this technique for finding an appropriate layout for a graph. But how can we define the distance matrix for a graph in general? In specific applications, we may have such a distance *a priori*, but in the absence of any other information, there is also an easy way to specify a distance matrix: we simply use the square root of the geodesic distances of the vertices; i.e. d_{ij} will be the square root of the minimum number of hops one has to take on the graph in order to reach vertex j from vertex i.

It can be shown that for Euclidean distances, the strain is minimized by calculating the eigenvectors corresponding to the eigenvalues with the k largest magnitude of a doubly-centered squared distance matrix $\hat{\mathbf{D}}$ obtained from the original matrix $\mathbf{D}$, and using the eigenvectors to place the vertices. More precisely, the elements of $\hat{\mathbf{D}}$ are calculated as follows:

$$\hat{d}_{ij} = -\frac{1}{2}\left(d_{ij}^2 - \frac{1}{n}\sum_{k=1}^{n} d_{ik}^2 - \frac{1}{n}\sum_{k=1}^{n} d_{kj}^2 + \frac{1}{n^2}\sum_{k=1}^{n}\sum_{l=1}^{n} d_{kl}^2\right) \tag{3.5}$$

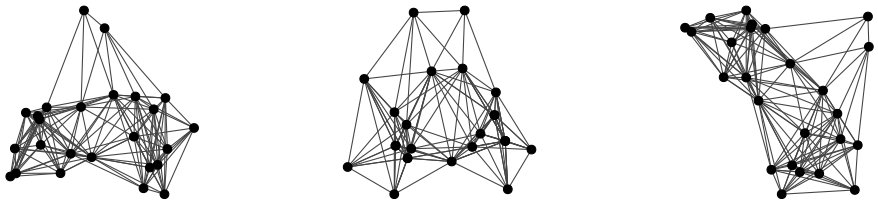
The kth coordinates of the vertices in the final layout are then given by the eigenvector corresponding to the kth eigenvalue when the eigenvalues are sorted in decreasing order of their magnitudes.

Both the distance matrix $\mathbf{D}$ and the doubly-centered squared distance matrix $\hat{\mathbf{D}}$ is usually dense, which means that this technique is also applicable only for small and mid-sized graphs; for instance, a graph with 10,000 vertices would require a distance matrix with 10^8 items. However, the algorithm works well for up to a few hundred or thousand vertices (depending on our patience), and it also guarantees that the distances are recovered exactly by the layout if it is possible to do so in the metric space we are embedding the graph in. In other words, if the distance values represent actual distances and not estimates, the layout will be equivalent to the original placement of vertices in the 2D (or 3D) space up to rotations, translations and reflections.

Since we already have a geometric graph which we used for testing force-directed layouts in the previous section, it is natural to use the same graph to test the performance of multidimensional scaling. We will generate two layouts; one with the exact distance matrix that we calculate from the original positions of the vertices, and one with the estimated distance matrix where d_{ij} is the square root of the geodesic distance between vertices i and j. First we calculate the exact distance matrix:

3.22.1)
```
points = [Point(v["x"], v["y"]) for v in graph.vs]
dist = [[p1.distance(p2) for p1 in points] for p2 in points]
```

The layouts can then be generated by calling *`layout()`* with `"mds"` as the name of the layout algorithm. The distance matrix is given in a keyword



(a) Using exact distances (b) Using estimated distances (c) Original layout

Fig. 3.5 Multidimensional layouts of a geometric random graph. Note that the layout obtained from the exact geodesic distances (left) is equivalent to the original layout (right) up to a rotation. The middle panel shows a layout with an estimated distance matrix.

argument called *dist*; if this is omitted, the estimated distance matrix will be used with the square roots of the geodesic distances:

```
3.23.1) layout_exact = graph.layout("mds", dist=dist)
    2) layout_est   = graph.layout("mds")
```

The two layouts can be plotted using the *plot()* function as before; the results should be similar to the ones seen on Fig. 3.5, which also shows the original arrangement of the vertices. Note that the original arrangement is equal to the one reconstructed by the MDS algorithm in *layout_exact* up to a rotation.

Finally, we note that the MDS algorithm can also be used to obtain a three-dimensional layout by passing *dim=3* as a keyword argument to *layout()*.

3.3.5 Handling large graphs

None of the layout algorithms mentioned so far are specifically recommended for large graphs. The simple deterministic layouts (circle, star, grid) do not look at the structure of the graph and hence they are unlikely to yield high-quality layouts for more than a handful of vertices; the force-directed algorithms have a computational complexity of at least $O(n^2h)$, and they are likely to get stuck in local minima for larger graphs; the multidimensional scaling requires a dense distance matrix on which we also have to calculate a few eigenvectors. What can we do then when we are confronted with large graphs of a few thousand vertices or more?

igraph provides three different layout algorithms that are specifically designed for large graphs: the grid-based variant of the Fruchterman–Reingold algorithm (Fruchterman and Reingold, 1991), the LGL (Large Graph Layout) algorithm (Adai et al, 2004) and the DrL algorithm (Martin et al, 2011)[1].

[1] The algorithm is now called OpenOrd, but igraph still uses its old name.

According to the authors, the DrL algorithm produced useable layouts for graphs up to 849,000 vertices, and the LGL algorithm has successfully been applied to a protein map containing more than 145,000 proteins from 50 genomes (see Adai et al (2004)), and also to draw a map of the Internet on the level of autonomous systems (see `http://www.opte.org/maps`). One of the authors of this book used the DrL algorithm to generate a global map of music artists from the similarity data provided by the website Last.fm (`http://www.last.fm`). This particular graph contained almost 75,000 artists and the DrL layout was calculated in less than 20 minutes; see `http://sixdegrees.hu/last.fm` for more details. However, what works for large graphs is not necessarily suitable for small ones: all of these layouts tend to produce worse layouts than a simple force-directed algorithm like the Fruchterman–Reingold or Kamada–Kawai algorithm if the graph has only a few hundred vertices.

Each of the three algorithms use a different approach to simplify the layout procedure in order to make it scalable up to large graphs. The *grid-based Fruchterman–Reingold layout* (identified by the names `"grid_fr"`, `"grid_fruchterman_reingold"` or `"gfr"`) is essentially equivalent to the Fruchterman–Reingold algorithm, but the repulsive forces between vertices are calculated only for some of the vertex pairs. This is motivated by the fact that the magnitude of repulsive forces inversely depend on the square of the distance between the vertices involved, thus these forces are negligible for vertices that are farther from each other than a given distance threshold. The grid-based variant therefore splits the space into cells using a regular grid, and calculates the repulsive forces only between vertex pairs that are in neighboring cells or in the same cell (where each cell in a 2D grid is assumed to have eight neighbors: two horizontally, two vertically and four diagonally). This idea decreases the time complexity of the entire algorithm to $O(n \log nh)$ from $O(n^2 h)$. The algorithm has essentially the same parameters in igraph as the original Fruchterman–Reingold algorithm (see Table 3.3); the only extra parameter is called `cellsize` and it defines the sizes of the cells to be used. The default cell size is the fourth root of the `area` parameter; in other words, the square root of the length of the side of the square in which the vertices are laid out in the first step.

grid-based Fruchterman–Reingold layout

The *LGL algorithm* (identified by `"lgl"` as the name of the layout) takes this idea one step further by not trying to lay out the entire graph at once. Instead of that, the algorithm constructs a breadth-first spanning tree of the original network, starting from a root vertex, and proceeds down the spanning tree layer by layer. In the first step, the root vertex and its direct neighbors are placed in a star-like configuration (with the root vertex in the center) and then laid out using a grid-based Fruchterman–Reingold algorithm as described above. When the vertices reached an equilibrium position, the next layer of the spanning tree (i.e. the set of nodes that are at distance two from the root) is added in a circle around the nodes that are laid out already, and the layout continues until a new equilibrium is reached. This procedure is repeated by

LGL algorithm

adding newer and newer layers of the spanning tree to the layout until the entire graph is laid out. The root vertex is picked randomly by igraph, but a specific root vertex may also be used. Most of the speedup compared to the grid-based Fruchterman–Reingold layout is achieved by decomposing the graph into layers. Due to its usage of a spanning tree as a guide for the layout, the LGL layout works the best if the underlying graph has an inherent tree-like structure, or if at least the most heavyweight edges form a tree. The parameters of the LGL algorithm are the same as the parameters of the Fruchterman–Reingold algorithm (see Table 3.3) with two additional keyword arguments: `cellsize` defines the sizes of the cells to be used in the grid-based Fruchterman–Reingold step, and `root`, which defines the root vertex to be used for the layout. `root` may be `None` (in fact, this is the default), meaning that a root vertex will be selected randomly.

DrL algorithm

The *DrL algorithm* (igraph name: `"drl"`) also uses graph decomposition as a way to speed up the layout process, but the decomposition is achieved by a simple clustering procedure instead of a spanning tree calculation. In the first step of the algorithm, the graph is *coarsened* by collapsing densely connected regions into single vertices. The second step lays out the graph using a grid-based force directed layout algorithm. The third step reverts the coarsening of the first step, and lays out the vertices by using the coarsened layout determined in the second step as a guide. The algorithm also employs a heuristic called *edge cutting*, which may removes a few edges from the graph if those edges are responsible for a significant amount of the total stress of the layout; the idea here is that a more aesthetically pleasing layout may be achieved if we remove those edges that connect distant regions of the graph as the stress on these edges cannot really be relieved anyway without compromising the overall quality of the layout. The strength of the DrL algorithm lies in the fact that the coarsening procedure may be applied multiple times if the coarsened graph is still too large to be laid out in a single step. This allows the algorithm to scale up to hundreds of thousands of vertices.

3.3.6 Selecting a layout automatically

In the previous subsections, we have learned about quite a few layout algorithms that are supported by igraph. There is one additional layout we should know about, which can be invoked by passing `"auto"` as the name of the layout algorithm to the `layout()` method. The automatic layout tries to select the most suitable layout algorithm for a graph based on a few simple structural properties according to the following procedure:

1. If the graph has an attribute called `layout`, its value will be used to determine the layout of the graph. If the `layout` attribute is an instance of `Layout` or a list of coordinate pairs, igraph will simply return these as the preferred layout. If the attribute is a string, it will be interpreted as

the name of a built-in layout algorithm; this algorithm will be invoked and the result will be returned. If the attribute is a callable function, it will be invoked with the graph as the first (and only) argument and its return value will be interpreted as the preferred layout.

2. If the graph has vertex attributes called *x* and *y*, these will be treated as the X and Y coordinates of the vertices in the preferred layout. Graphs generated by the geometric random graph generator (see the *GRG()*-method of *Graph*) will possess these vertex attributes automatically, therefore the vertices will always be laid out according to their original positions unless you delete the attributes explicitly. For three-dimensional layouts, an additional vertex attribute *z* is also needed. *GRG()*
3. If the graph is connected and has at most 100 vertices, the Kamada–Kawai layout will be used.
4. If the graph has at most 1000 vertices, the Fruchterman–Reingold layout will be used.
5. If everything else above has failed, the DrL layout algorithm will be used.

You can request the three-dimensional variant of these layouts by passing *dim=3* as a keyword argument when invoking the automatic layout function. The automatic layout is also the default layout in igraph, therefore you can simply invoke the *plot()* function on an arbitrary graph if you wish to draw it quickly without thinking too much about which layout algorithm would be the most appropriate. For instance, the Petersen graph can simply be plotted as follows, which would lay it out according to the Kamada–Kawai layout algorithm:

3.24.1)
```
plot(petersen)
```

3.4 Drawing graphs

3.4.1 The `plot()` *function*

Throughout the previous section, we have used the *plot()* function without knowing exactly what it does. It is not hard to guess that *plot()* plots graphs, but the whole truth is more complicated than that: this function can be used just as well to save the visualization in PNG, PDF, SVG or Postscript formats. *plot()*

This section will be devoted entirely to the usage of the *plot()* function, which draws a graph (or, as we will see later in Section 3.5, many other igraph objects as well) to a given canvas. The canvas itself means an image file on the disk, which can be in any format that is supported by the Cairo plotting backend. Even if you do not specify an output file name to indicate that you simply wish to see the graph on the screen, igraph will plot to a temporary

PNG file in the background and show this file to you using the default image viewer of your platform.

The *plot()* function has three positional arguments and three keyword arguments, most of which are optional. The positional arguments are as follows:

'obj' Specifies the object being plotted. Throughout this section, *obj* will always be an instance of the *Graph* class. This is the only argument of *plot()* that is mandatory.

'target' Specifies the canvas where Cairo will plot the given object. The *target* is usually *None* (the default value), which means that the graph has to be plotted to a temporary file and this graph has to be shown on the screen using the default image viewer of the operating system. You may also specify a string here, which will be treated as the name of a file where the plot should be saved. The supported formats are PNG, PDF, SVG and PostScript. Finally, you can also pass an arbitrary Cairo surface object here, which is useful if you are embedding an igraph plot in an application that draws to the screen using Cairo, as you can simply pass in the Cairo surface that the application itself draws to.

'bbox' Specifies the bounding box in which the plot will be drawn on the canvas. This is usually a tuple specifying the width and height of the plot. PNG files and on-screen plots assume that the width and height are given in pixels, while PDF, SVG and PostScript plots assume points, where 72 points are equal to 1 inch (2.54 cm). The default value is *(600, 600)*, which yields a plot of 600×600 pixels on the screen or in PNG format, or a plot of $8\ 1/3$ in $\times$ $8\ 1/3$ in ($21\ 1/6$ cm $\times$ $21\ 1/6$ cm) in PDF, SVG or PostScript formats. Alternatively, you can use an instance of *BoundingBox* here.

Besides these, there are three extra keyword arguments that *plot()* handles:

'opacity' Specifies the opacity of the object being plotted. Zero means that the object will not be plotted at all (it is totally transparent), while one means that the object is entirely opaque. An opacity of 0.75 thus corresponds to 75% opacity, or 25% transparency.

'margin' The top, right, bottom and left margins of the plot as a 4-tuple. This can be used to leave extra space on the sides of the plot if needed. If the tuple contains less than 4 elements, they will be re-used. For instance, if you specify a pair of numbers here, they will be interpreted as the vertical (top and bottom) and horizontal (right and left) margins, respectively. Specifying a single number is also permitted; this means that the same margin will be used on all sides. The default margin is 20.

'palette' The palette of the plot. We will talk about palettes later in Section 3.4.2.

The remaining positional and keyword arguments are passed on intact to the *__plot__()* internal method of the object being plotted (along with a few

__plot__()

extra arguments that we will describe later in Section 3.5). This means that *plot()* accepts many more arguments than the five described above, but most of these are interpreted by *obj* and not by *plot()* itself. We have already seen an example of such an argument: in all the code examples involving *plot*, we have added the *layout* keyword argument to specify the layout that should be used to determine the positions of the vertices of the graph.

The main purpose of the keyword arguments for *Graph* instances in *plot()* determine how the vertices and edges should be plotted. However, before we dive in to the full list of these arguments, let us first familiarize ourselves with the various data sources igraph collects information from to determine how a particular vertex or edge should look like on a graph plot.

Let us assume that the graph drawer in igraph is currently drawing vertex `i` and is trying to figure out what the color of the vertex should be. The drawer first turns to the keyword arguments of the *plot()* call and checks whether there is a keyword argument called *vertex_color* which contains a list of colors. If so, the drawer takes the `i`th element of the list (or, if the list contains only `k` elements, then the `i` mod `k`th element, taking into account that indices start from zero both for the vertices and the list items), and uses that as the color of the vertex. If there is no keyword argument called *vertex_color*, igraph turns to the vertex attributes of the graph and looks for a vertex attribute called *color*, whose value will be used to determine the color of the vertex. In the absence of such a vertex attribute, igraph takes a look at the default settings in the *config* variable of the *igraph* module and retrieves the value of the "*plotting.vertex_color*" key. If even this attempt fails (because there is no such key), the drawer uses a hard-wired default color, which is red.

As an illustration, let us construct a simple ring graph and play with the vertex colors a little bit. The following code snippet constructs the graph and plots it with the default settings (and, remember, the automatic layout, since we did not specify the *layout* keyword argument either):

```
3.25.1) ring = Graph.Ring(9)
    2) plot(ring)
```

You can also save the visualization in a file by adding the name of the file as the second argument of *plot*:

```
3.26.1) plot(ring, "ring.pdf")
```

In the obtained visualization, all the vertices are red, since there is no *vertex_color* keyword argument to *plot()*, no *color* vertex attribute in *ring* and no default color specification in the configuration. Let us first override the default vertex color in the configuration:

```
3.27.1) config["plotting.vertex_color"] = "lightblue"
```

Plotting the graph now would draw every vertex with a light blue color due to the presence of the "*plotting.vertex_color*" configuration key. To

return to the hard-wired default vertex color, just delete the configuration key from the *config* dictionary:

```
3.28.1) del config["plotting.vertex_color"]
```

The *config* variable is not just a plain dictionary, although it behaves like one. It also allows you to save the configuration to the disk using the *save()* method or load it from a file using the *load()* method. The default configuration of igraph is stored in a file named `.igraphrc` in your home directory; this is `/home/`*yourusername* on Linux, `/Users/`*yourusername* on Mac OS X and `C:\Documents and Settings\`*yourusername* on Windows.

If we plotted our ring graph now after having deleted the vertex color key from the configuration, all the vertices would be red again – unless we set a vertex attribute that specifies the color of each vertex. For instance, to turn all the vertices green, you should type:

```
3.29.1) ring.vs["color"] = "green"
```

This is a shorthand notation for the following to save some typing when the same attribute value is assigned to all the vertices:

```
3.30.1) ring.vs["color"] = ["green"] * ring.vcount()
```

You can also assign a list of vertex colors to the "*color*" vertex attribute; if the list is shorter than the number of vertices, it will be re-used:

```
3.31.1) ring.vs["color"] = ["red", "red", "green"]
    2) ring.vs["color"]
```

Plotting the graph now would draw every third vertex in green. However, the visual properties dictated by the vertex attributes can be overridden by the keyword arguments of *plot()*; for instance, the following code snippet would draw every third vertex in red, green or blue, respectively (note that the elements of the list are re-used again just like with attribute assignment):

```
3.32.1) plot(ring, vertex_color=["red", "green", "blue"])
```

The same principles work for *any* vertex or edge attribute, not just for the color. In summary, igraph follows the following procedure when determining the value of a visual property *property* of a vertex or edge with index `i`:

1. If a keyword argument named *vertex_property* (for vertices) or *edge_property* (for edges) is specified in the *plot()* invocation, the value of the keyword argument is used to determine the visual property. If the argument is a number or string, it is used for all the vertices (or edges). If the argument is a sequence of `k` elements, the (`i` mod `k`)th element is used.
2. If a vertex (or edge) attribute *property* exists in the graph being plotted, its value for vertex (or edge) `i` is used.

Name	Meaning	Default
color	The color of the vertex. See Section 3.4.2 for more details about specifying colors.	"*red*"
label	The label of the vertex.	none
label_angle	The angle of the label relative to the horizontal axis, in radians. See Section 3.4.3 for more details about the label placement algorithm.	$-\pi/2$
label_color	The color of the label. See Section 3.4.2 for more details about specifying colors.	"*black*"
label_dist	The distance of the label from the center of the vertex shape. See Section 3.4.3 for more details about the label placement algorithm.	0
label_size	The font size of the label.	14
shape	The shape of the vertex. igraph supports the following shapes at the moment: • "*circle*": a regular circle. "*circular*" is an alias. • "*rectangle*": a rectangle. "*rect*", "*square*" and "*box*" are aliases. • "*triangle*": an upright triangle. "*triangle-up*" and "*up-triangle*" are aliases. • "*down-triangle*": a downward pointing triangle. "*triangle-down*" is an alias. • "*diamond*": a diamond shape. "*rhombus*" is an alias.	"*circle*"
size	The size of the vertex, measured in pixels for PNG and on-screen plots, or points for PDF, SVG and PostScript files.	20

Table 3.6 Visual properties of vertices in igraph plots

3. If a configuration key called *plotting.vertex_property* exists in the *config* variable, its value will be used.
4. Otherwise, a hard-wired default value will be used.

Of course the *property* has to be replaced with the name of the visual property we are interested in (e.g., *label*, *size*, *width* and so on). Tables 3.6 and 3.7 summarize the visual properties of vertices and edges that you are allowed to control in the plots.

There are four additional keyword arguments of the *plot()* function that we have not talked about so far. The first one is the *layout* argument, which specifies the layout of the graph (i.e. the assignment between the vertices of the graph and the points of the plane). The value of this argument can be one of the following:

- A string describing the name of a layout algorithm. The graph drawer will then invoke the layout algorithm with the default parameters and use the resulting layout to plot the graph. This is a shortcut to calling *layout()*

Name	Meaning	Default
arrow_size	Multiplicative factor to the size of the arrowhead on directed edges. The actual size will be 15 units of the canvas times the factor given here.	1
arrow_width	The width of the arrowhead on directed edges. This controls the angle of the arrowhead: the angle will be 36° times the value of *arrow_width*.	1
color	The color of the edge. See Section 3.4.2 for more details about specifying colors.	"*#444*"
width	The width of the edge in the units of the canvas.	1

Table 3.7 Visual properties of edges in igraph plots

with the name of a layout algorithm and then using the result as the value of the *layout* keyword argument.
- An instance of the *Layout* class which we have typically calculated beforehand using an appropriate call to the *layout()* method. Of course it is perfectly acceptable to use any hand-crafted layout as well.
- A list of coordinate pairs, which are simply converted to a *Layout* instance on the fly.

The second keyword argument is called *mark_groups*, and it allows one to enclose a subset of vertices in the graph using a shaded background. This is typically used to show clusters in a graph, but it can also be used to point out any arbitrary region of interest as long as the vertices of the region are relatively close to each other. The usage of this argument will be described in depth later in Section 3.4.4.

The final two keyword arguments serve the same purpose: they allow the user to influence the order in which the vertices are drawn on the canvas. By default, igraph draws the vertices in increasing order of the numeric vertex IDs. If the layout is very dense, vertices drawn later may cover the vertices that were drawn earlier. The *vertex_order* keyword argument lets you specify the drawing order. Each element of this list must be a vertex ID, and igraph will simply draw the vertices in the order dictated by *vertex_order*. For instance:

3.33.1)
```
plot(ring, vertex_order=[0,8,1,7,2,6,3,5,4])
```

Alternatively, you can pass the name of a vertex attribute as the value of the *vertex_order_by* keyword argument to use the values of this attribute to control the order. The argument may also be a tuple where the first element describes the name of the attribute and the second is *True* or "*desc*" if the ordering should be reversed, and *False* or "*asc*" otherwise.

To demonstrate the usage of this keyword argument (and also the other ones), we are going to visualize a real-world dataset: a subset of the air transportation network of the United States, assembled for 500 cities with the most air traffic by Colizza et al (2007). The dataset is readily available

for download in Nexus (see Chapter **??**), therefore we will simply use the built-in Nexus interface to obtain the data:

```
3.34.1) air = Nexus.get("us-airport")
```

Let us first examine the summary of the dataset:

```
3.35.1) summary(air)
```

This dataset is a moderately small network with `air.vcount()` vertices and `air.ecount()` weighted edges.

The vertices represent the airports considered in the dataset, and edges correspond to air travel connections. Each edge is weighted by the total number of available passenger seats per year for a given connection. We can also see that the dataset is anonymized as the vertex names contain numeric IDs only:

```
3.36.1) air.vs["name"][:5]
```

This means that we do not know which node belongs to which airport, and we cannot lay the graph out according to a map projection. Luckily, we can still use the built-in layout algorithms to obtain a nice arrangement for the nodes. The automatic layout selection will use the Fruchterman–Reingold layout algorithm to find the positions of the vertices. For a graph of this size, calculating the layout takes a couple of seconds, and since we are going to experiment a bit with different plot parameters, it will save us some time to save the layout to a variable before we start plotting:

```
3.37.1) layout = air.layout()
```

Note that we did not use the edge weights in the layout algorithm; doing so would probably have pulled high-traffic network hubs too close to each other. Let us plot the graph with the calculated layout to see what we have so far:

```
3.38.1) plot(air, layout=layout)
```

Well, one thing is for sure: the layout of the airport network looks very different from the geographical layout, but of course this is expected since force-directed layout algorithms tend to pull connected vertices close to each other, while there is no point in a high-traffic air connection between cities that are close to each other in geographical space. We can also see that the majority of the vertices are situated in a tightly connected core, but we do not know how strong the edges are in or outside the core, neither can we see how many passengers a given airport can handle per year. We will add these features to the plot soon. We will also increase the size of the plot to 1200×1200 units (i.e. twice as large as before) to be able to see more details.

First, we are going to use the widths of the edges to show the number of passenger seats for a given connection. Thicker edges will correspond to

connections carrying more passengers, but how shall we calculate the width of the edge from the connection capacity? Let us first get a rough idea of the distribution of connection capacities by calculating the minimum and the maximum weight and the 25th, 50th and 75th percentiles of the weight distribution:

```
3.39.1) min(air.es["weight"])
    2) max(air.es["weight"])
    3) percentile(air.es["weight"])
```

percentile() *percentile()* is an igraph function that calculates sample percentiles of a vector of numbers given as an argument. The second parameter specifies the percentile(s) to calculate; the default value is *(25, 50, 75)* which returns exactly those percentiles we were looking for.

We can see that the weight distribution spans several orders of magnitude, therefore it seems reasonable to use the logarithms of the weights to calculate the edge widths. Quite arbitrarily, we will map a weight of 10^4 to a line width of 1 and a weight of 2×10^6 to a line width of 10 using a logarithmic scale. The *rescale()* *rescale()* function is designed exactly for such transformations. *rescale()* takes a list of numbers, an input range, an output range and an optional transformation function `f`, then transforms the numbers using `f` and re-scales them such that the endpoints of the input range (before applying `f`) are mapped to the endpoints of the output range. The complete argument list is as follows:

`'values'` The list of values to be re-scaled.

`'out_range'` The output range, i.e. the range in which the values within the input range will be mapped. The range must be given as a tuple with lower and upper limits, and it is allowed to use a lower limit that is larger than the upper limit. The default is *(0, 1)*.

`'in_range'` The input range, i.e. the range whose endpoints will be mapped to the lower and upper limit of the output range. The default is equivalent to the minimum and maximum values in *values*.

`'clamp'` If *True*, input values outside the input range will be mapped to the lower and upper limits, depending on which one is closer. This ensures that every number in the result is within the output range, even if the input range is smaller than the range of input values.

`'scale'` The function `f` that is used to transform the input values. Typical choices are *log()* or *log10()* from the *math* module for a logarithmic transformation, or *sqrt()* to use the square root of values, which is useful when we want the area of a vertex to be proportional to some value, since igraph controls the size of a vertex and not the area.

Putting it all together, we can use the following call to determine the edge widths:

```
from math import log
air.es["width"] = rescale(air.es["weight"], out_range=(1, 10),
                          in_range=(1e4, 2e6), scale=log)
```

We also want the sizes of the vertices to be proportional to the number of passengers the corresponding airport serves. This can be calculated by summing the weights of edges incident on a given vertex using the *strength()* method. We calculate the extrema and the percentiles as well:

strength()

```
importance = air.strength(weights="weight")
min(importance), max(importance)
percentile(importance)
```

As expected, the distribution of vertex strengths is just as heavily skewed as the edge weights, therefore it is also advised to use a log-transformation before rescaling. We will transform the strength into a size range between 1 and 50 units:

```
air.vs["size"] = rescale(importance, out_range=(1, 50),
                         scale=log)
```

Finally, we ask igraph to plot the vertices in increasing order of vertex sizes so that larger vertices (i.e. more important airports) are drawn on top of smaller ones. Since the visual properties of the vertices and edges are set up already in the appropriate attributes, we only have to specify the size of the plot (which is going to be 1200×1200 pixels), the pre-calculated layout and the desired vertex drawing order:

```
plot(air, bbox=(1200, 1200), layout=layout,
     vertex_order_by="size")
```

The final plot is depicted on Figure 3.6.

When experimenting with plots, it often requires several iterations until a satisfying visualization is obtained. A common trick to reduce the amount of typing involved is to prepare a dictionary that contains all the keyword arguments to be passed to *plot()*. This dictionary can be updated easily, and it takes much less typing to invoke *plot()* again. E.g.:

```
params = dict(layout=layout, bbox=(1200, 1200),
              vertex_order_by="size")
plot(air, **params)
```

Now if we realized that the plot would look much better with light blue square vertices, we only have to update 'params' and run the *plot()* command again, which you can easily do in most Python IDEs by pressing the up arrow on your keyboard until you get back to the appropriate *plot()* call in the command history:

```
params["vertex_shape"] = "square"
params["vertex_color"] = "lightblue"        #labelvrb:colorname
plot(air, **params)
```

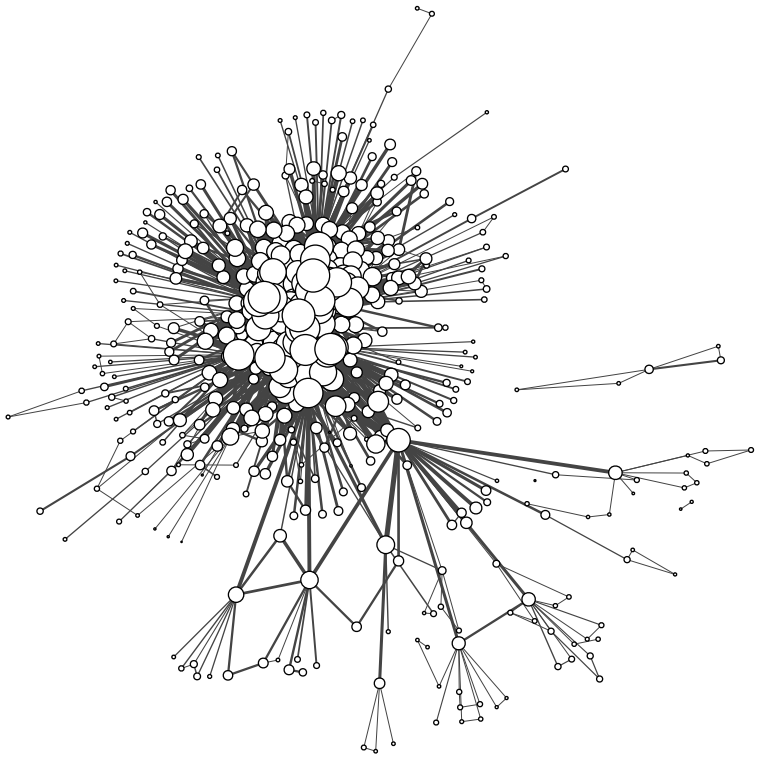


Fig. 3.6 The US airport network as plotted by igraph using the Fruchterman–Reingold layout.

3.4.2 Colors and palettes

Line **??** in the previous code snippet uses the name of a color (`lightblue`), but so far we have not talked about how igraph recognizes colors, and what other ways are there to define the colors of edges and vertices on a graph plot.

First of all, igraph recognizes all the standard X11 and HTML color names. The full list can be retrieved by printing the keys of the `known_colors` dictionary. We only print the first ten entries here:

3.46.1)
```
sorted(known_colors)[:10]
```

This is the easiest way to specify a color – just use its X11 name and igraph will know what to do. The other possible way is to specify the red (R), green (G) and blue (B) components of the color as floats in the range 0 to 1. For instance, red=1.0, green=0.5 and blue=0 gives you a tangerine shade:

RGB components

```
3.47.1) air.vs["color"] = [(1.0, 0.5, 0)] * air.vcount()
```

Note that we had to multiply the tuple by the number of vertices to assign the same color to all the vertices, otherwise the color of the first three vertices would have been assigned 1.0, 0.5 and 0, respectively, and the elements of the tuple would have been re-used for the remaining vertices. This is something that you have to watch out for when using tuples to specify colors for a set of vertices rather than a single vertex. An alternative syntax lists the components in a string, separated by commas, slashes or whitespace, therefore strings like `"1.0 0.5 0"`, `"1/0.5/0"` or `"1, 0.5, 0"` are all equivalent to `(1.0, 0.5, 0)`.

You can also add a fourth component to the color, which will describe the *alpha channel*, i.e. the "opaqueness" of the surface the color is painted to. Zero corresponds to a fully transparent surface, and 1 corresponds to a fully opaque one. Intermediate values describe partially translucent surfaces; for instance, painting a red circle with an alpha value of 0.5 on top of a white surface would yield a pink circle as red is evenly mixed with white.

alpha channel

Besides RGB components, you can also use CSS3 color specifications[2]. These are always given in a string that follows one of the following formats:

HTML-like format. The color is given as `#rrggbb`, `#rgb`, `#rrggbbaa` or `#rgba`, where `r`, `g`, `b` and `a` denote hexadecimal digits (0-9, A-F) corresponding to the red, green, blue and alpha components, respectively. Letter codes like `rr` mean that the corresponding component should be specified in the range 00-FF (in hexadecimal), where `r` means that the range is 0-F only. The orange color we have cited as an example above is then given as `#f80` or `#ff8000` without alpha channel, and `#f80f` or `#ff8000ff` with alpha channel.

RGB components. Here the format is `rgb(red, green, blue)` or `rgba(red, green, blue, alpha)` where `red`, `green`, `blue` and `alpha` specify the corresponding components of the color using either decimal numbers between 0 and 255, or using percentages between 0% and 100%. Therefore, `rgb(100%, 50%, 0%)` or `rgba(255, 128, 0, 255)` also correspond to our favourite tangerine color.

HSL components. HSL refers to the *HSL color space*, which also describes colors using three components: *hue*, *saturation* and *lightness*. Hue is given as degrees on the color circle between 0° and 360°, where 0° corresponds to red, 60° to yellow, 120° to green, 180° to cyan, 240° to blue and 300° to purple. Saturation and lightness is specified in percentages. Orange is situated halfway between red and yellow, giving us a hue of 30°; its corresponding saturation and lightness is 50% and 0%, respectively, hence its HSL specification: `hsl(30, 50%, 0%)`. Alpha channels are also supported if you use `hsla` instead of `hsl`; the generic format is `hsl(hue, saturation, lightness)` and `hsla(hue, saturation, lightness, alpha)`.

HSL color space
hue
saturation
lightness

HSV components. HSV refers to the *HSV color space* (hue-saturation-

HSV color space

[2] See `http://www.w3.org/TR/css3-color/`

value), which is similar to the HSL color space, but the lightness component is replaced by the *value* component. The value is also given as a percentage and it is equal to the maximum of the values of the red, green and blue components in the RGB space. Tangerine is `hsv(30, 50%, 100%)` in HSV space; the generic format is `hsv(hue, saturation, value)` or `hsva(hue, saturation, value, alpha)`.

value

The last option to specify colors in igraph is to supply a single integer whenever a color is expected; in such cases, the number is used as an index into the current *palette* of the plot. Roughly speaking, a palette is a mapping between integers from zero to $n-1$ (where n is the size of the palette) to colors. igraph provides a handful of default palettes that are useful for various types of visualizations; these are stored in the `palettes` dictionary:

palette

```
3.48.1) for name in sorted(palettes):
    2)     print "%-16s %r" % (name, palettes[name])
```

Whenever you draw a graph using the `plot()`, you may specify a palette explicitly using the `palette` keyword argument, e.g.:

```
3.49.1) pal = GradientPalette("red", "blue")
    2) plot(petersen, layout="fr", palette=pal)
```

The palette selection works similarly to other visual properties. If you specify a `palette` as a keyword argument, this palette will be used. If there is no `palette` argument, igraph falls back to the `plotting.palette` key of the configuration settings. If no such key exists in the configuration, a hardwired default will be used, which always refers to a grayscale palette from black to white.

The list of default palettes in igraph is summarized in Table 3.8. Each entry in `palettes` is an instance of some subclass of the `Palette` class; most frequently `GradientPalette` or `AdvancedGradientPalette`. Whenever you specify a palette in the `palette` keyword argument of `plot()`, or in the configuration settings, you may use a palette name to refer to one of the default palettes.

Palette
GradientPalette
AdvancedGradientPalette

You can also examine any of the `Palette` instances using the `plot()`-function:

plot()

```
3.50.1) plot(palettes["red-black-green"], bbox=(300, 100))
```

`GradientPalette`s represent a linear gradient between two given colors in the RGB space:

```
3.51.1) red_green = GradientPalette("red", "green")
    2) len(red_green)
    3) red_green[128]
```

Name	Description
gray	Grayscale gradient from black to white
heat	Heat-map style palette: a gradient from red to yellow to white, where yellow is reached at palette index 192.
rainbow	Rainbow palette that contains all the fully saturated colors around the edge of the color circle. This is achieved by varying the hue from 0° to 360° while keeping saturation and value at 100% in HSV space.
red-black-green	Gradient from red to black to green; typically used to show gene expression values in biological networks (red: down-regulated, black: neutral, green: up-regulated)
red-blue	Gradient from red to blue
red-green	Gradient from red to green
red-purple-blue	Gradient from red to purple to green
red-yellow-green	Gradient from red to yellow to green
terrain	Gradient from green (plains) to dark yellow (hills) to light gray (mountains), just like the colors used on a topographic map.

Table 3.8 The list of default palettes in igraph. Each palette contains 256 entries.

As seen above, palettes behave like lists, but of course you cannot assign to the entries in a palette. They can also resolve string color specifications to RGB components using the *get()* method, even if the colors being queried are not in the palette:

```
3.52.1) red_green.get("maroon")
    2) red_green.get_many(["rosybrown", "hsv(30, 50%, 100%)"])
```

The default length of a gradient palette is 256 colors, but you can override it at construction time:

```
3.53.1) red_green = GradientPalette(n=42)
    2) len(red_green)
    3) red_green[21]
    4) red_green[50]
```

The *AdvancedGradientPalette()* is similar to *GradientPalette()*, but it allows more than two stops in the gradient; for instance, a more saturated gradient between the red and green colors can be obtained if we add an extra yellow stop halfway between red and green:

AdvancedGradientPalette(

```
3.54.1) red_yellow_green = AdvancedGradientPalette(["red", "yellow",
    2) red_yellow_green[0]            ## this will be red
    3) red_yellow_green[2]            ## this will be orange
    4) red_yellow_green[4]            ## this will be yellow
    5) red_yellow_green[6]            ## this will be lime
    6) red_yellow_green[8]            ## and this is green
```

RainbowPalette() varies the hue between two extremes while keeping the saturation, the value and the alpha level in HSV space constant:

RainbowPalette()

```
3.55.1) rainbow_part = RainbowPalette(start=0.1, end=0.7, n=120)
```

Here, hue is specified as a floating-point value between 0 and 1 (0 corresponding to 0° and 1 corresponding to 360°), but you may also use values larger than 1 or smaller than zero. For instance, to go *counter-clockwise* from 60° (yellow) to 240° (blue):

```
3.56.1) rainbow_part = RainbowPalette(start=1.0/6, end=-2.0/6, n=120)
```

PrecalculatedPalette()

PrecalculatedPalette() simply treats a list of color identifiers as a palette; every color in the list specified at construction time will be resolved to RGBA components:

```
3.57.1) colors = PrecalculatedPalette(["red", "green", "blue",                    "yellow
    2) colors[0]      ## this will be red
    3) colors[3]      ## this will be yellow
    4) colors[6]      ## there is no such color in the palette
```

PrecalculatedPalette() can be used to plot a flat clustering of a graph easily as you can simply assign the membership vector of a clustering to the *color* attribute of the vertices or use it as the value of the *vertex_color* keyword argument of *plot()*. The following code snippet generates a geometric random graph, splits it up to four clusters, and then colors each of the clusters with a different color:

```
3.58.1) grg = Graph.GRG(100, 0.25)
    2) clusters = grg.community_fastgreedy().as_clustering(4)
    3) plot(grg, vertex_color=clusters.membership, palette=colors)
```

Actually, there is an even easier way to plot clusterings, which is described later in Section 3.5.1.

3.4.3 Placing vertex labels

The position of vertex labels on graph plots is controlled primarily by two vertex attributes: *label_angle* and *label_dist*. The former specifies the direction of the label from the center of the node, while the latter defines the distance. The distance is actually the *ratio* between the real distance and the size of the vertex, which allows us to speciy the same distance ratio for all the vertices while still taking into account that a label must be farther from a larger vertex than from the smaller one. We will denote the angle by α, the distance ratio by d and the size (radius) of the vertex by r. When igraph

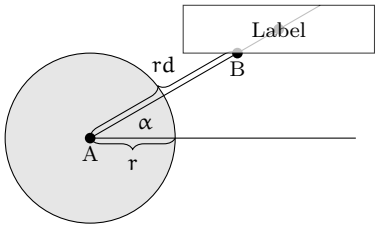


Fig. 3.7 Placement of a vertex label near the vertex centered at A. d is the value of the *label_dist* attribute, α is the angle given by *label_angle* in radians. B is the reference point that is at distance rd from A, and the label is placed such that B is on the bounding box of the label and A, B and the center of the bounding box lie on the same straight line.

places the labels of the vertices, it treats all the vertices as if they were circles with radius r even if they have a different shape.

The idea behind the vertex placement algorithm of igraph is illustrated on Figure 3.7. First, the algorithm finds the point B that is at distance rd from the center of the vertex A in the direction specified by α. The label is then placed such that its bounding box intersects the line connecting A with the center of the bounding box exactly at point B. Therefore, $\alpha = 0$ and $d = 1$ will place the label in a way that the middle of its left edge touches the rightmost point of the vertex; $\alpha = \pi/2$ places the label above the vertex such that the middle of its bottom edge touches the topmost point of the vertex and so on. When α is either 0, $\pi/2$, π or $-\pi/2$, it is thus ensured that the label does not overlap with the vertex. Small overlaps may occur for non-horizontal and non-vertical angles, especially close to the diagonals, but it is usually enough to set d to slightly larger than 1 to avoid such overlaps as well.

3.4.4 Grouping vertices

This section will describe the *mark_groups* keyword argument that you can pass to the *plot()* function when plotting graphs. As its name suggests, *mark_groups* allows you to specify groups of vertices that will be marked by a colored area that encloses them on the plot to a single group. A natural use-case for this feature is to show the result of a clustering algorithm (assuming that vertices in the same cluster are fairly close to each other on the plot), or to highlight a group of vertices with some specific structural properties.

mark_groups can take one of the following values:

- *None*, meaning that no groups will be highlighted. This is the default setting.
- A dictionary where the keys are collections (e.g., tuples) of vertex indices and the corresponding values are colors (see Section 3.4.2 for color speci-

fication formats). E.g., the following code would group vertices 1, 3 and 5 together in a red shaded area:

```
3.59.1) marks =
    2) marks[1,3,5] = "red"
    3) plot(grg,  mark_groups=marks)
```

- A sequence or iterable that contains pairs, the first element of the pair being a collection of vertex IDs and the second element being the corresponding color:

```
3.60.1) marks = [((1, 3, 5), "red")]
    2) plot(grg, mark_groups=marks)
```

Instead of collections of vertex indices, you may also use *VertexSeq* instances in this syntax. For instance, the above code would group all the vertices with betweenness larger than 200 in a red shaded area:

```
3.61.1) marks = [(grg.vs.select(_betweenness_gt=200), "red")]
    2) plot(grg, mark_groups=marks)
```

Note that marking a group of vertices makes sense only if these vertices are situated close to each other on the graph. igraph simply calculates the convex hull of the selected vertices (i.e. the smallest convex polygon that contains all the vertices), expands its boundary by a few units and creates a smooth Bézier curve around it. Therefore, it should be ensured that other vertices not belonging to the set being marked are not placed within the convex hull to avoid ambiguity.

3.5 Visualization support for other igraph objects

VertexClustering
VertexDendrogram
VertexCover
CohesiveBlocks
Matrix
__plot__()

Graphs are not the only objects that can be plotted in igraph using the *plot()* function. We have already seen one example in Section 3.4.2 where we used *plot()* to take a look at the colors of a palette. Many other igraph objects also support plotting: you can also plot a *VertexClustering*, a *VertexDendrogram*, a *VertexCover*, a *CohesiveBlocks* instance (representing the cohesive blocking structure of a graph) or a heatmap-style representation of a 2D *Matrix* (which is igraph's wrapper class to a list of lists of numbers). All these objects implement a *__plot__()* method that igraph uses to draw the objects on a Cairo context.

As we remember from Section 3.4.1, the *plot()* function handles only a handful of its positional and keyword arguments on its own, and the remaining ones are processed by the object being plotted. What this means is that most of the arguments are passed on to the *__plot__()* method of the object being plotted, and it is the *__plot__()* method that interprets these

and adjusts the plot accordingly. Section 3.4.1 already described the keyword arguments understood by `Graph.__plot__()`. In the following subsections, we will briefly describe the keyword arguments processed by other igraph objects.

3.5.1 Plotting clusterings

Instances of `VertexClustering`, `VertexCover` and `CohesiveBlocks` understand all the keyword arguments handled by `Graph` instances; after all, they also plot a graph in the end. Besides plotting the graph, `VertexClustering` also colors the vertices according to the clustering such that vertices in the same cluster are assigned the same color. `VertexCover` and `CohesiveBlocks` are overlapping structures, hence they do not color the vertices to avoid confusion with vertices belonging to multiple groups, but they can optionally mark groups of vertices belonging to the same cluster, just like we could have done manually using the `mark_groups` keyword argument as described in Section 3.4.4.

The following keyword arguments are handled differently by these classes:

- `mark_groups` accepts `True` as a possible value. This means that all clusters should be enclosed in shaded areas:

```
3.62.1) clusters = grg.community_spinglass()
    2) plot(clusters, mark_groups=True)
```

  For `VertexCover` and `CohesiveBlocks`, this setting is required to distinguish the plot from a standard graph plot as these classes do not color the vertices according to the clusters by default.
  Another difference is that the default value for `mark_groups` is not necessarily `None` (meaning not to mark the groups at all); igraph also looks at the value of the `plotting.mark_groups` configuration key. If the value of the key is `True`, igraph will mark the groups even if `mark_groups` is not specified explicitly.
- `mark_groups` also accepts cluster indices wherever a collection of vertex IDs is expected. For instance, `mark_groups = [((1,2,3), "red")]` would mark vertices 1, 2 and 3 with a red shaded area, but `mark_groups = [(1, "red")]` would mark *members* of the cluster with ID 1 with a similar area.
- `mark_groups` may also be a list of cluster indices that should be highlighted on the figure; colors will be assigned to the clusters automatically.
- `vertex_color` is not allowed for `VertexClustering` instances as it would override the coloring imposed on the graph by the cluster structure.
- The default value of `palette` is an instance of `ClusterColoringPalette`, a special palette class that contains colors that are easily distinguishable from each other. Use the following code snippet to take a look at the typical

`ClusterColoringPalette`

structure of a *ClusterColoringPalette* containing 35 different colors (and thus suitable for showing 35 clusters):

3.63.1)
```
plot(ClusterColoringPalette(35))
```

base colors

The first seven colors of a *ClusterColoringPalette* are the so-called *base colors*: red, green, blue, yellow, magenta, cyan and gray. These are used for the first seven clusters. If more colors are needed (because there are more clusters in the plot), igraph generates darker and lighter variants of the base colors; for instance, colors between indices 7 and 13 (inclusive) are dark red, dark green, dark blue and so on, while colors between indices 14 and 20 (inclusive) are light red light green, light blue etc. Even darker and lighter variants are created in blocks of seven if they are needed, potentially *ad infinitum*, but the colors become harder and harder to distinguish after about 25-30 elements. Of course you can still choose to provide your own palette if needed.

Another subtle difference between a *VertexClustering* plot and a *Graph* plot is that edges going between different clusters are dimmed in a *VertexClustering* plot by default to emphasize intra-cluster edges. An example of a plot that can be obtained by plotting a *VertexClustering* directly is to be seen on Figure 4.8(a).

3.5.2 Plotting dendrograms

igraph provides plotting support for the *Dendrogram* and *VertexDendrogram* classes as well, allowing the user to produce simple dendrograms like the one seen on Figure 4.8(b). igraph does not aim to be a full-fledged plotting package for dendrograms, though, therefore only a single keyword argument is supported for these classes: *orientation*, which describes the orientation of the dendrogram. Four orientations are possible: the "left to right" orientation plots the leaves of the dendrogram on the left and the root on the right; the "right to left" orientation plots leaves on the right and the root on the left; the "top-down" or "top to bottom" orientation plots leaves on the top and the root on the bottom, while the "bottom-up" or "bottom to top" orientation puts leaves on the bottom and the root on the top. The default orientation is "left to right". The orientations and their corresponding aliases are shown on Table 3.9.

format()

If you need more sophisticated dendrogram plots, you can always export the dendrogram in Newick format into a string using its *format()* method. The Newick format is read by the vast majority of dedicated dendrogram plotting packages; a comprehensive list of such packages can be found at `http://bioinfo.unice.fr/biodiv/Tree_editors.html`.

Orientation	Names
left to right	*left-right*, *lr*, *horizontal*, *horiz* or *h*
right to left	*right-left* or *rl*
top-down	*top-down*, *top-bottom*, *td* or *tb*
bottom-up	*bottom-up*, *bottom-top*, *bu* or *bt*

Table 3.9 Dendrogram orientations and their names in igraph.

3.5.3 Plotting matrices

Instances of the *Matrix* class in igraph can be used to represent matrices of numeric values; for instance, the *get_adjacency()* method of the *Graph* class returns a *Matrix* that contains the (possibly weighted) adjacency matrix of the graph. It is beyond the scope of this chapter to describe all the methods of *Matrix*, most of which are self-explanatory, therefore we will focus only on the keyword arguments accepted by *Matrix.__plot__()*:

Matrix
get_adjacency()

- *style* describes the style of the plot. When *style = "boolean"*, the matrix is assumed to contain Boolean values (zeros and ones) only, and a grid will be plotted where each cell corresponding to a true value is black; the remaining cells are white. *style = "palette"* means that the matrix cells contain indices into the current palette (which can be controlled by the *palette* keyword argument), and these colors are used to paint the backgrounds of the matrix cells. *style = "none"* means that the backgrounds of the matrix cells are left transparent. In both cases, *None* values in matrix cells are treated specially, such cells always remain empty.
- *square* describes whether the matrix cells should be squares or not. If *square* is *True*, the area where the matrix will be plotted may contain empty padding either at the top or the bottom, or at the left and right edges to ensure that the matrix cells are squares. If *square* is *False*, the entire area is used for plotting the matrix even if that means that the cells are not squares. The default value is *True*.
- *grid_width* controls the width of the lines used to separate neighboring matrix cells. Zero or negative grid width turns the grid off. It will also be turned off if the size of a cell is less than three times the grid width. The default value is 1. Fractional line widths are also allowed.
- *border_width* controls the thickness of the border drawn around the matrix. Zero or negative values turn the border off. The default value is 1.
- *row_names* may be a list that assigns a name to each of the matrix rows. These will be shown to the left of each row.
- *col_names* may be a list that assigns a name to each of the matrix columns. These will be shown above each column. If *col_names* is not given but *row_names* is, and the matrix is square, the same names will be used for both the rows and the columns.

- *values* may be *True* to ask igraph to print the numeric values of each cell in the plot, or another *Matrix* instance to override the values from another matrix. The default value is *False*, which means that nothing should be printed in the cells.
- *value_format* can be a format string or a callable function that specifies how the values should be printed. If it is a callable, it will be called for every cell with the cell value as the only argument, and it should return a string that contains the formatted cell value. For instance, passing *value_format="%#.2f"* will always print exactly two digits after the decimal point in each cell.

As an example, let us plot the adjacency matrix of our geometric random graph 'grg' on a canvas of 1200 × 1200 pixels:

3.64.1)
```
plot(grg.get_adjacency(), bbox=(1200, 1200))
```

Interestingly enough, most of the non-zero values in the adjacency matrix are near the diagonal; this is because vertices in geometric random graphs generated by igraph are ordered by their y coordinates, hence it is very unlikely that an edge is generated between two vertices in two distant rows.

3.5.4 Plotting palettes

Palettes are plotted in a way that is very similar to matrices: a row or column of cells where each cell is colored according to an entry from the palette. It is not too surprising that igraph actually re-uses the code that produces matrix plots when plotting a palette, therefore the list of accepted keyword arguments is very similar to a limited subset of the ones understood by *Matrix*:

- *border_width* controls the thickness of the border drawn around the palette. Zero or negative values turn the border off. The default value is 1.
- *grid_width* controls the width of the lines used to separate neighboring cells in the palette. Contrary to *Matrix* plots, the default value for *grid_width* is zero.
- *orientation* controls the orientation of the palette; *horizontal* plots the colors in a horizontal strip, while *vertical* plots a vertical strip instead. You may also use any of the orientation aliases seen on Table 3.9.

3.6 Compositing multiple objects on the same plot

Plot()

The *plot()* function in igraph is a convenient wrapper around a lower-level class called *Plot()*. This class allows you to show multiple objects on the

same plot; for instance, you can plot different layouts of the same graph next to each other. The invocation of the *plot()* function is roughly equivalent to the following:

```
def plot(obj, target=None, bbox=(600, 600), *args, **kwds):
    figure = Plot(target, bbox, background="white")
    figure.add(obj, bbox, *args, **kwds)
    if target is None:
        result.show()
    elif isinstance(target, basestring):
        result.save()
    return result
```

This means that *plot()* simply constructs an appropriate *Plot* instance, adds the object being drawn to the plot using a bounding box that covers the entire area, and then either shows it on the screen (if there was no target filename or Cairo surface specified), or saves it (if *target* was a string). Finally, the constructed *Plot* instance is returned so you can manipulate it further.

For more complicated plots, you can repeat what *plot()* does and add multiple objects at the appropriate places. For instance, the following code plots the Petersen graph on a canvas of 400pt $\times$ 400pt, adds its adjacency matrix in the top right corner with an opacity of 70%, and then saves the entire plot in a file called *petersen.pdf*.

```
figure = Plot("petersen.pdf", bbox=(400, 400))
figure.add(petersen, layout="fr")
figure.add(petersen.get_adjacency(), bbox=(300, 0, 400, 100),
           opacity=0.7)
figure.save()
```

The constructor of the *Plot* takes at most three arguments, all of which have default values. *target* is the target surface where the plot will be drawn, similarly to the *target* argument of the *plot()* function. *None* means that a temporary image will be created and this will be shown on the screen; strings mean that the plot should be saved in an appropriate PNG, PDF, SVG or PostScript file; instances of Cairo surfaces mean that the plot should be drawn on the given surface directly. *bbox*, the second argument specifies the bounding box in which the plot will be drawn, again similarly to the *bbox* argument of the *plot()* function. The third argument, *palette*, specifies the default palette that will be used in the plot unless it is overridden explicitly by one of the objects being plotted.

The most important methods of the *Plot* class are as follows.

add() Adds a new object to the plot. The first argument is the object to be added. The remaining ones are keyword arguments: *bbox* specifies where the object should draw itself, *palette* specifies the palette used to map

integers into colors when the object (typically a graph) is being drawn, and *opacity* specifies the opacity of the object (the default being 1.0). The remaining positional and keyword arguments are stored and passed on later to the *__plot__()* method of the object when the plot is being drawn. *bbox* may be a 2-tuple or 4-tuple or an instance of *BoundingBox*, just like the *bbox* argument of the *plot()* function.

remove() Removes an object from the plot. Returns *True* if the removal was successful or *False* when the object was not part of the plot.

redraw() Draws the plot to the Cairo surface that backs the plot, without saving the plot. This allows you to add extra decorations (such as labels or arrows) to the plot before saving it to a file. Before drawing, *redraw()* looks at the *background* property of the plot and if it is not *None*, fills the entire area of the plot with the given color.

save() Saves the plot to the file given at construction time. It also calls the *redraw()* method before saving the plot unless it was called already.

show() Saves the plot to a temporary file and shows it. igraph tries to make an educated guess about the default image viewer application on your platform, but if that fails or you wish to use a different image viewer, you can specify its full path in the *apps.image_viewer* configuration key.

Plot also has a few useful properties: *background* returns or sets the background color of the plot that *redraw* uses to prepare the canvas; *width* and *height* return the width and height of the area where the plot will be drawn, respectively; *bounding_box* returns a copy of the bounding box of the plot, and *surface* returns the Cairo surface on which the plot will be drawn. This is useful to add decorations to the figure: first you can set up the plot, then add the necessary igraph objects, call *redraw()* to draw it to the Cairo surface, and use the *surface* property and native Cairo methods to draw the decorations before finally calling *save()*.

3.7 Exercises

▶ EXERCISE 3.1. Implement a custom layout algorithm for general graphs that uses the Kamada–Kawai layout algorithm to find an approximate placement of the nodes and then refines the layout using the Fruchterman–Reingold algorithm.

▶ EXERCISE 3.2. Layouts produced by force-directed layout algorithms (which are used most frequently when nothing else is known about the structure of a graph) are invariant to rotations and translations. However, the human eye finds layouts with mostly horizontal and vertical edges aesthetically more pleasing. Implement an algorithm that takes an arbitrary igraph graph and its layout, and rotates the layout in a way that the edges are as horizontal or vertical as possible. (Hint: for each edge, you can calculate its

angle with the `x` axis using the *`atan2()`* function from the *`math`* module. Try to find a goal function which is low when most of the edges are parallel either to the `x` or the `y` axis, and try to minimize this function using its derivative).

Chapter 4
Community structure

4.1 Introduction

(Zachary, 1977; Fortunato, 2010; Porter et al, 2009) **TODO: Write this section**

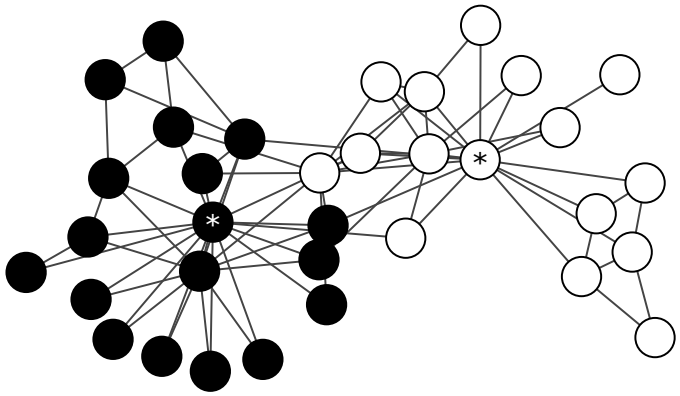
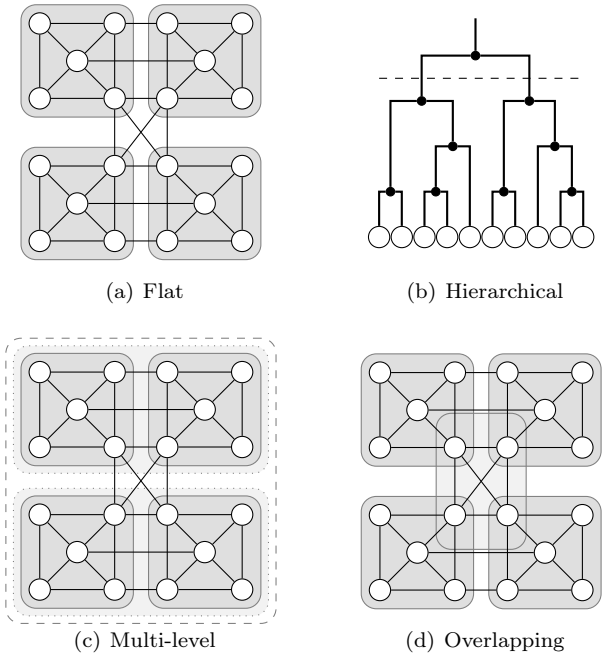


Fig. 4.1 The karate club network of Zachary (1977). The two communities observed by Zachary are shown in black and white, respectively. The community leaders (the administrator and the instructor of the club) are marked by a star.



(a) Flat (b) Hierarchical

(c) Multi-level (d) Overlapping

Fig. 4.2 Illustration of different community structure types on a schematic graph.

4.2 Types of community structure

4.2.1 Flat community structures

Flat community structures are probably the simplest and most widely studied representations of communities in real networks. In a flat community structure, every vertex belongs to one and only one of the communities; in other words, there are neither outlier vertices nor overlaps between the communities. Fig. 4.2(a) shows a possible flat community structure on a simple graph with 20 vertices.

membership vector

A flat community structure with n vertices and k communities can be described in two ways. The first option is the *membership vector*, which is simply an integer vector of length n. Communities are indexed by integers from 1 to k, and element i of the membership vector is j if vertex i belongs to community j. One could theoretically use any k distinct integers to denote the communities; igraph uses indexes from 1 to k by convention, but also understands any membership vector as long as the members are integers. Membership vectors using indices $1, 2, \ldots, k$ are called *canonical membership vector*s.

canonical membership vector

The other representation of a flat community structure is a list of `k` lists, where list `i` contains the indices of the vertices corresponding to community `i`. Note that it must be ensured that each vertex appears in one and only one of the lists.

igraph uses the membership vector representation internally, but the community detection methods wrap the membership vectors in a more convenient interface provided by the `communities` class. Whenever you are working with flat community structures in igraph, you are dealing with `communities` instances. As an example, let us construct the example graph on Fig. 4.3 and its corresponding `communities` representation:

`communities` class

```
4.1.1) graph <- make_graph( ~ A-B-C-D-A, E-A:B:C:D,
2)                            F-G-H-I-F, J-F:G:H:I,
3)                            K-L-M-N-K, O-K:L:M:N,
4)                            P-Q-R-S-P, T-P:Q:R:S,
5)                            B-F, E-J, C-I, L-T, O-T, M-S,
6)                            C-P, C-L, I-L, I-P)
7) flat_clustering <- make_clusters(
8)     graph,
9)     c(1,1,1,1,1,2,2,2,2,2,3,3,3,3,3,4,4,4,4,4))
10) flat_clustering
```

```
IGRAPH clustering unknown, groups: 4, mod: 0.51
+ groups:
  $‘1‘
  [1] 1 2 3 4 5

  $‘2‘
  [1]  6  7  8  9 10

  $‘3‘
  [1] 11 12 13 14 15

  $‘4‘
  + ... omitted several groups/vertices
```

As seen above, `communities` instances are constructed by specifying the membership vector. The clustering object can then be indexed like a regular R list to retrieve the members of the communities:

```
4.2.1) flat_clustering[[1]]
```

```
[1] 1 2 3 4 5
```

```
3) flat_clustering[[3]]
```

```
[1] 11 12 13 14 15
```

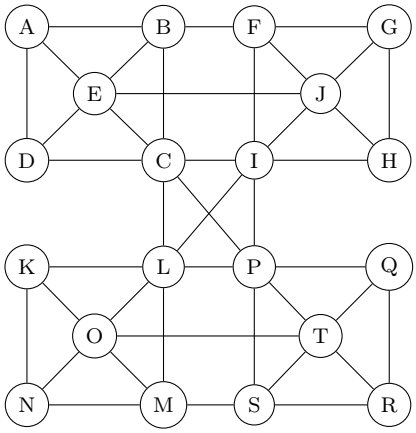


Fig. 4.3 Close-up view of the graph on the panels of Fig. 4.2.

You can also retrieve the number of clusters using the `length()` function or iterate over the clusters in a `'for'` loop:

```
4.4.1) length(flat_clustering)
```

```
[1] 4
```

```
3) for (cluster in groups(flat_clustering)) {
4)   print(V(graph)[cluster])
5) }
```

```
+ 5/20 vertices, named, from 06a2207:
[1] A B C D E
+ 5/20 vertices, named, from 06a2207:
[1] F G H I J
+ 5/20 vertices, named, from 06a2207:
[1] K L M N O
+ 5/20 vertices, named, from 06a2207:
[1] P Q R S T
```

`communities` also provides a convenience method to retrieve the sizes of the clusters, and creating a subgraph from a community is also easy:

```
4.6.1) sizes(flat_clustering)
```

```
Community sizes
1 2 3 4
5 5 5 5
```

```
5) induced_subgraph(graph, flat_clustering[[1]])
```

```
IGRAPH d09011f UN-- 5 8 --
+ attr: name (v/c)
+ edges from d09011f (vertex names):
[1] A--B A--D A--E B--C B--E C--D C--E D--E
```

4.2.2 Hierarchical community structures

Hierarchical communities are typically produced by top-down or bottom-up clustering algorithms. A *top-down clustering algorithm* initially considers the whole graph as a single cluster and iteratively divides each cluster containing at least two vertices into two nonempty subclusters. At the end of the process, each vertex will belong to a separate cluster. A *bottom-up clustering algorithm* proceeds the opposite way, agglomerating initial communities containing single vertices into larger ones by merging two clusters at each and every step until only one cluster remains.

top-down clustering algorithm

bottom-up clustering algorithm

The outcome of both algorithms can be represented as a *dendrogram*, i.e. a tree-like diagram that illustrates the order in which the clusters are merged (in the bottom-up case) or split (in the top-down case). Such a dendrogram can be seen on Fig. 4.2(b). The nodes in the bottom layer of the dendrogram are called *leaves*, and each leaf represents one of the original vertices of the graph. The remaining nodes in the dendrogram are the *branches*; each branch corresponds to a split or a merge of two leaves or other branches. The tip of the uppermost branch is called the *root* of the dendrogram. One can obtain flat clusterings from a dendrogram by cutting it at a specified level, essentially stopping the merging or splitting process after a given number of steps. The dashed line on Fig. 4.2(b) shows a possible cut of the dendrogram; this particular cut will yield two clusters with the leftmost and rightmost five vertices, respectively.

dendrogram

leaves

branches

root

igraph can represent dendrograms as instances of the `communities` class: it can store instance store the structure of the dendrogram in the form of a *merge matrix* that contains the whole merge history of the clustering process. Each row in the merge matrix corresponds to one merge, therefore the merge matrix of a complete dendrogram with n leaf vertices has $n-1$ rows and two columns. The numbers in row i of the merge matrix refer to the nodes of the dendrogram. Integers between 1 and n (inclusive) are the leaf nodes, and therefore the original nodes of the graph as well. Integers larger than n correspond to branches that were created in earlier merge steps. Branch $n+1$ is created after the first merge, branch $n+2$ is created after the second merge and so on. A concrete example dendrogram and its merge matrix is shown on Fig. 4.4. Top-down clusterings are represented in the same way as bottom-up ones; in other words, a dendrogram obtained from a top-down clustering is treated as if it were created by a bottom-up one.

merge matrix

Let us construct our dendrogram using the merge matrix seen on Fig. 4.4:

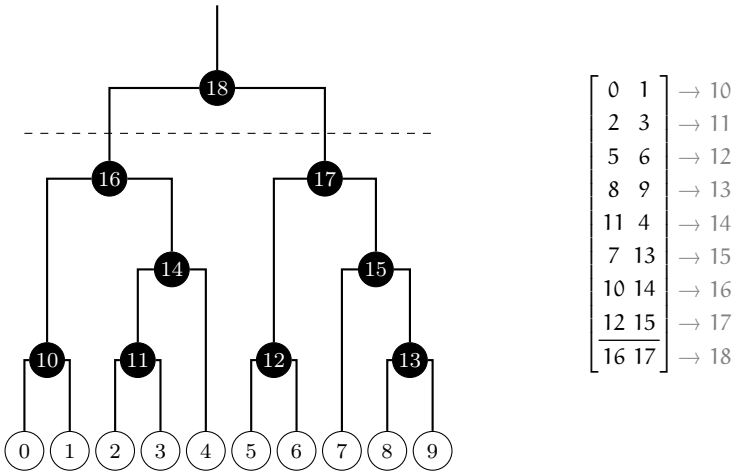


Fig. 4.4 Close-up view of the dendrogram on Fig. 4.2(b) along with its corresponding merge matrix. The indices in the vertices refer to the numbers used in the merge matrix. The gray numbers next to each row of the merge matrix show the index of the branch that was created by merging the two nodes referred by the row. A horizontal line in the merge matrix is drawn above the first merge which is *not* performed if the dendrogram is cut at the dashed line.

```
4.8.1) dummy_graph <- make_full_graph(5) * 2 + edge(1, 6)
   2) merge_matrix <- rbind(c(1, 2), c(3, 4), c(6, 7), c(9, 10),
   3)   c(12, 5), c(8, 14), c(11, 15), c(13, 16), c(17, 18))
   4) dendrogram <- make_clusters(dummy_graph, merges = merge_matrix)
   5) dendrogram
```

```
IGRAPH clustering unknown, groups: NA, mod: NA
 + groups not available
```

cut_at()

The bridge between dendrograms and flat clusterings is the `cut_at()`-method of the `communities` class. The method cuts the dendrogram appropriately to obtain a specified number of clusters:

```
4.9.1) cut_at(dendrogram, 2)
```

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

```
   3) cut_at(dendrogram, 5)
```

```
 [1] 4 4 1 1 1 3 3 5 2 2
```

4.2.3 Multi-level community structures

Multi-level communities represent another step forward towards modeling the whole complexity of community structures found in real networks. They are similar to the hierarchical community structures introduced in the previous section, but it is allowed to merge more than two communities into a larger super-community in a step. In contrast, hierarchical community structures always merge exactly two chosen communities at every given step.

Most multi-level community detection algorithms are built on top of a flat community detection algorithm as follows. The input network is first decomposed into disjoint communities. The communities are then used to construct a meta-network where each community of the *original* network is contracted into a single node. Edges between the communities are usually kept in the meta-network, but multiple edges between the same communities are collapsed into a single one whose weight is equal to the total weight of the collapsed edges. The meta-network is then re-clustered using the same algorithm, and the whole process is repeated until only one giant community remains. Fig. 4.2(c) shows an example of multi-level community structure in a network. The four communities at the lowest level are shown inside dark grey shaded areas. The two upper and the two lower communities are then aggregated into larger communities in the middle layer (see the light gray shaded areas). The dashed line represents the single higher level community that encloses the two mid-level communities.

Multi-level community structures in igraph are represented simply by list of `VertexClustering` instances, one item belonging to each level. Section 4.4.4.1 will present one particular multi-level community detection algorithm that is natively implemented in igraph.

4.2.4 Overlapping community structures

Up to now, all the representations we have seen classify a single vertex of the network into one and only one of the communities. Even with hierarchical and multi-level community structures, a vertex cannot be a member of two communities at the same "resolution level". Overlapping community structures lift this limitation by allowing a vertex to belong to one or more communities, or to none of them at all. Fig. 4.2(d) shows a possible overlapping community structure of the same example graph that we have been using throughout the last few sections. Note that the four nodes in the middle of the figure participate in two communities instead of one.

R igraph does not currently have a separate class that represents an overlapping community structure, but one easy way around this is to just store the overlapping groups of vertices in a list:

```
4.11.1) clusters <- list(
2)    V(graph)[1, 2, 3, 4, 5],
3)    V(graph)[5, 6, 7, 8, 9],
4)    V(graph)[10, 11, 12, 13, 14],
5)    V(graph)[15, 16, 17, 18, 19],
6)    V(graph)[2, 8, 11, 15]
7) )
```

This concludes our short overview of the possible representation of community structures, but before we move on to the actual community detection algorithms, we should first discuss how to distinguish good communities (i.e. the ones that are representative of the underlying structure of the network) from bad ones.

4.3 How good is a community structure?

Community detection is a heuristic process; different community detection methods usually provide different results on the same graph (unless the graph has a very clear-cut community structure), and it is usually up to the user to decide which one of the results represents the ground truth the most accurately. Researchers have nevertheless invented measures to quantify how good a given community structure is according to some internal or external criterion, and one can make use of such measures before coming to a decision. Such measures are usually defined for flat community structures, mostly because little is known yet about the "true" overlapping communities of a real network, while a flat clustering of real networks is known in advance in many cases, providing us a yardstick to measure detected communities with.

External quality measures

There are basically two types of quality functions for community structures. *External quality measures* compare the set of detected communities with another set that was given in advance. These measures are therefore useful in cases when the ground truth (i.e. the set of communities we are looking for) is given in advance; in other words, when we are benchmarking community detection algorithms against each other to find out which one is the best for some particular task. For instance, suppose you would like to find out which proteins are likely to form protein complexes, given a protein-protein interaction network where the nodes represent proteins and the edges represent connections between them. In this case, you can take *another* protein-protein interaction network from a different organism where the complexes are known already, compare the different community detection methods with each other using an external quality measure, and then choose the one which performed the best to find the complexes in the dataset you are actually interested in.

internal quality measures

Contrary to the external ones, *internal quality measures* do not require the ground truth to quantify how good the clusters are, but need an assumption

about how the "good" clusters should look like. For instance, a measure may assume that a good clustering has only a few edges that go between different clusters, or that clusters in a good clustering are as dense as possible. Later on, we will see what the problem is with these assumptions and how can they be resolved in order to obtain useful internal quality measures.

4.3.1 External quality measures

External quality measures compare a set of detected communities to a gold standard set, which is thought to represent the ground truth. igraph implements four external quality measures natively: the Rand index (Rand, 1971) and its adjusted variant (Hubert and Arabie, 1985), the split-join distance (van Dongen, 2000), the variation of information metric (Meilă, 2003) and the normalized mutual information (Danon et al, 2005). With the exception of the split-join distance and the variation of information, these measures are *similarity measures*, meaning that higher values represent a better agreement between the two sets being compared. The split-join distance and the variation of information are distance metrics, i.e. the farther the clusterings are, the higher the value of the measure is. Table 4.1 lists the supported measures and their basic properties.

Measure	Type	Range	igraph name	Citation
Rand index	similarity	0 to 1	`rand`	Rand (1971)
Adjusted Rand index	similarity	-0.5 to 1	`adjusted.rand`	Hubert and Arabie (1985)
Split-join distance	distance	0 to $2n$	`split.join`	van Dongen (2000)
Variation of information	distance	0 to $\log n$	`vi`	Meilă (2003)
Normalized mutual information	similarity	0 to 1	`nmi`	Danon et al (2005)

Table 4.1 External quality measures implemented in igraph and their basic properties. Similarity scores are higher for similar clusterings, while distance scores are higher for dissimilar ones.

Let us first load the Zachary karate club network (shown on Fig. 4.1) into igraph. The annotated version of this network is available in the *`igraphdata`* package:

```
4.12.1) data(karate)
```

Zachary's original publication also gives us which faction each of the members of the club belong to. This is available as the `'Faction'` vertex attribute.

This will be our ground truth with which we wish to compare other clusterings:

```
4.13.1) ground_truth <- make_clusters(karate, V(karate)$Faction)
    2) length(ground_truth)
```

```
[1] 2
```

```
    4) ground_truth
```

```
IGRAPH clustering unknown, groups: 2, mod: 0.37
+ groups:
  $‘1‘
   [1]  1  2  3  4  5  6  7  8 11 12 13 14 17 18 20 22

  $‘2‘
   [1]  9 10 15 16 19 21 23 24 25 26 27 28 29 30 31 32 33 34
```

For the sake of studying external quality measures, let us assume a very simple clustering method that takes two vertices of the graph as input, and classifies each vertex v to one of two communities depending on whether it is closer to the first or second in terms of the shortest paths in the graph. A natural choice for the two vertices would be the instructor and the administrator of the club; these are represented by vertices named ‘Mr Hi‘ and ‘John A’. We have already seen how to work with shortest paths in Chapter 2, therefore it is easy to decide for each vertex whether it is closer to Mr Hi or John A.

```
4.15.1) dist_memb <- karate %>%
    2)   distances(v = c("John A", "Mr Hi")) %>%  #label{brv:commdistance}
    3)   apply(2, which.min) %>%
    4)   make_clusters(graph = karate)
```

Line **??** calculates the distance of all the vertices from Mr Hi and from John A. The result is a matrix with two rows, one for each vertex. The next line creates the membership vector. The ith element of the vector is one if the distance to Mr Hi is smaller than the distance to John A and 2 otherwise. Finally, we convert the membership vector to a `communities`. Note that vertices that are equally far from Mr Hi and John A automatically get assigned to John A's community, due to `which.min()` returning the index on the first minimum.

Now that we have the ground truth in `‘ground_truth’` and our clustering in `‘dist_memb’`, we can try comparing them using any of the external quality measures igraph provides. All the external measures are provided by the `compare()` function, which accepts the two clusterings and the name of the measure to be calculated, and returns the value of the measure.

compare()

4.3.1.1 Rand index

Rand index

The *Rand index* (Rand, 1971) is one of the earliest measures for comparing communities. To calculate the Rand index, one has to consider all possible *pairs* of items and check whether the two clusterings being compared agree or disagree regarding the affiliations of the vertices in the pair. Two clusterings are said to agree about vertices v_1 and v_2 if v_1 and v_2 either belong to the *same* community in both clusterings, or belong to *different* communities in both clusterings. We call such pairs *congruent*, and it is obvious that a higher number of congruent pairs means that the two clusterings are closer to each other. The Rand index is then simply the number of congruent pairs divided by the number of possible vertex pairs.

The Rand index is a similarity metric and its maximal value is 1, which is being attained if the clusterings are identical. Note that `compare()` also works on simple numeric membership vectors:

```
4.16.1) compare(c(1,1,2,2,2,2), c(2,2,1,1,1,1), method = "rand")
```

```
[1] 1
```

Note that the measure is maximal even though cluster 0 in the first clustering matches with cluster 1 in the second clustering is vice versa. This is true for more general cases as well; the Rand index (and all the other measures we present here) are insensitive to the permutations of cluster indices.

The minimum value of zero can also be easily attained if one of the clusterings puts every vertex in the same cluster and the other one puts every one of them in a cluster of its own:

```
4.17.1) compare(c(1,1,1,1), c(1,2,3,4), method = "rand")
```

```
[1] 0
```

Now let us see how our simple algorithm performed with respect to the ground truth:

```
4.18.1) rand_index <- compare(ground_truth, dist_memb, method = "rand")
    2) rand_index
```

```
[1] 0.6292335
```

This is not bad, especially when taking into account how simple our algorithm is. Also note that the two clusterings are interchangeable; the result does not change if we provide the clusterings in the opposite order:

```
4.19.1) compare(dist_memb, ground_truth, method = "rand")
```

```
[1] 0.6292335
```

Note that we can also calculate the exact number of congruent pairs by multiplying the Rand index with the total number of pairs:

```
4.20.1) n <- gorder(karate)
    2) rand_index * n * (n-1) / 2
```

```
[1] 353
```

4.3.1.2 Adjusted Rand index

In the previous subsection, we have seen how can one attain the maximum value for the Rand index (by making the clustering equal to the ground truth it is compared with) and have also seen a case where the index attained its lower bound of zero. However, that case was quite special: a trivial clustering of one cluster was compared with another trivial clustering where each cluster had only one member. What happens if we compare clustering that are created totally randomly?

First, let us create a function that assigns n vertices randomly to one of k clusters and returns a membership vector of length n:

```
4.21.1) random_partition <- function(n, k = 2) { sample(k, n, replace = TRUE) }
    2) random_partition(10)
```

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

```
    4) random_partition(10)
```

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

Each invocation generates a different partition of n vertices into (by default) 2 clusters. Now, what happens if we generate 100 random community structures of 100 vertices each, pair them and calculate the average Rand index of each pair?

```
4.23.1) total <- numeric(100)
    2) for (i in seq_len(100)) {
    3)   c1 <- random_partition(100)
    4)   c2 <- random_partition(100)
    5)   total[i] <- compare(c1, c2, method = "rand")
    6) }
    7) mean(total)
```

```
[1] 0.4988929
```

This number is suspiciously close to $1/2$, and it is indeed the case: when we generate clusterings at random, there is a $1/2$ probability that a vertex pair becomes congruent by chance. Since the Rand index is the fraction of congruent vertex pairs, it will also be equal to $1/2$. Feel free to repeat our experiment for $k > 2$ – no matter how many clusters we use, the expected Rand index between random clusterings will never be zero, although its exact value will of course depend on the number of clusters and their expected sizes.

This means that we can start to worry about the value of the Rand index we obtained in the previous subsection for our simple clustering: what if its value is also a result of random chance?

The non-zero expectation of the Rand index was the motivation of Hubert and Arabie (1985) when they introduced the *adjusted Rand index*. The adjustment calculates the expected value of the Rand index under the assumption that the clusterings are generated randomly with the same cluster sizes, subtracts the expected value from the actual value of the index and then re-scales the result so that the maximum becomes 1 again, effectively making the expected value after adjustment equal to zero. *adjusted Rand index*

igraph can also calculate the adjusted Rand index – all you have to do is to specify `'method = "adjusted.rand"'` in `compare()`. Let's repeat our experiment with random clusterings to confirm that the expected value of the adjusted Rand index is indeed zero for random clusterings:

```
4.24.1) total <- numeric(100)
    2) for (i in seq_len(100)) {
    3)   c1 <- random_partition(100)
    4)   c2 <- random_partition(100)
    5)   total[i] <- compare(c1, c2, method = "adjusted.rand")
    6) }
    7) mean(total)
```

```
[1] 0.001395469
```

The adjusted Rand index between the ground truth of the Zachary karate club network and our simple clustering then confirms that our method does not perform too bad after all:

```
4.25.1) compare(ground_truth, dist_memb, method = "adjusted.rand")
```

```
[1] 0.2579365
```

The adjusted Rand index is therefore more suitable for comparing community structures than the original, unadjusted variant, as positive values mean that the agreement between the two clusterings is at least better than random chance.

4.3.1.3 Split-join distance

The Rand index gave us a similarity measure: the more similar the two clusterings being compared are, the higher the value of the index is. Some people find distance measures more intuitive; such measures are high for dissimilar clusterings, and gradually decrease as clusterings become more similar to each other. This and the following subsection will present two distance measures that are implemented in igraph.

Loosely speaking, the *split-join distance* (van Dongen, 2000) is propor- *split-join distance*

tional to the number of vertices that have to be moved between clusters in order to transform one clustering to the other. The whole truth is more complicated than that, and we will return to a more detailed definition shortly, but first let us focus on the major properties of this distance measure.

Let $d(C_1, C_2)$ denote the split-join distance between clusterings C_1 and C_2. The split-join distance then satisfies four very important conditions for all C_1, C_2 clustering pairs:

1. $d(C_1, C_2) \geq 0$, i.e. the distance of any two clustering is non-negative.
2. $d(C_1, C_2) = 0$ if and only if $C_1 = C_2$. Every clustering is at zero distance from itself, and different clusterings always have some non-zero distance to each other.
3. $d(C_1, C_2) = d(C_2, C_1)$; the measure is symmetric.
4. $d(C_1, C_3) \geq d(C_1, C_2) + d(C_2, C_3)$ for any C_3. This is the equivalent of the well-known triangle equality in the space of clusterings.

The above four conditions are the so-called *metric axioms*. Since the split-join distance satisfies these conditions, it is a metric; in other words, the split-join distance in the space of clustering works similarly to the ordinary Euclidean distance we know so well from geometry.

The split-join distance can be calculated in two ways in igraph. The first method is to call `compare()` with `'method = "split.join"'`. The other way is to call the `split_join_distance()` function directly.

split_join_distance()

```
4.26.1) c1 <- c(1,1,1,2,2,2)
    2) c2 <- c(1,1,2,2,2,2)
    3) compare(c1, c2, method = "split.join")
```

```
[1] 2
```

```
    5) split_join_distance(c1, c2)
```

```
distance12 distance21
         1          1
```

The curious reader may have two immediate questions here:

1. One can obtain `'c2'` from `'c1'` by moving the third vertex from community 1 to community 2, so why does the measure return 2 instead of 1? Does it always return twice the number of steps?
2. Why does `split_join_distance()` return two numbers instead of one, and why do they sum up exactly to the result of `compare()`?

To answer these questions, we have to introduce the concept of *subclustering* first. Clustering C is a subclustering of clustering A if a vertex pair that is in the same cluster in clustering C is also in the same cluster in clustering A; in other words, the clusters in C can be obtained by splitting the clusters in cluster A. This relationship is denoted by $C \subseteq A$. If clustering C is a subclustering of A, then it is also said that A is a *superclustering* of C. Of

subclustering

superclustering

course the trivial clustering where every vertex is in the same cluster is a superclustering of every other clustering, and the other trivial clustering where every vertex is in its own cluster is a subclustering of every other clustering. Note that this implies that every pair of clustering (A, B) has at least one common subclustering C; in the worst case, this is the trivial clustering consisting of singleton clusters, but its existence is guaranteed. These concepts are illustrated on Fig. 4.5.

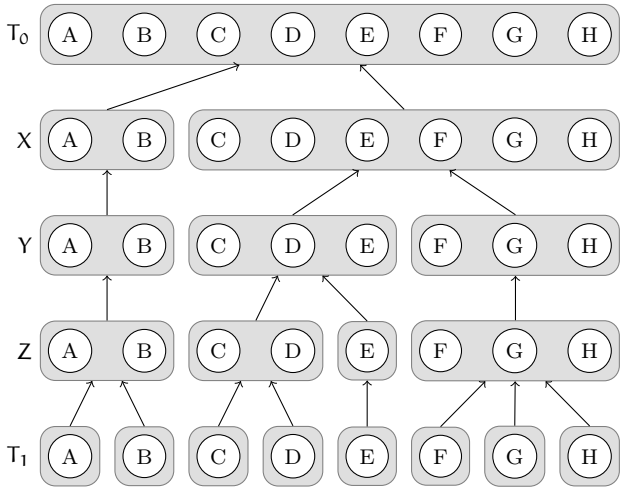


Fig. 4.5 Subclusters and superclusters. T_0 is the trivial clustering where every vertex belongs to the same cluster; T_1 is the trivial clustering where every vertex is in its own cluster. X, Y and Z are non-trivial clusterings such that $T_1 \subseteq Z \subseteq Y \subseteq X \subseteq T_0$.

This leads us to the formal definition of the split-join distance. For any clusterings A and B, one must first find the common subclustering C that satisfies the following requirements:

- $C \subseteq A$ and $C \subseteq B$ (C is a common subclustering)
- For any clustering D, $C \subseteq D$ implies that either $D \not\subseteq A$ or $D \not\subseteq B$. In other words, there exists no superclustering of C that is also a common subclustering of A and B.

The split-join distance is then the number of vertices that have to be splitted into different clusters to reach C from A plus the number of vertices that have to be joined with other clusters to reach B from C; hence the name of the measure. That's why moving a vertex to a different cluster counts as two steps: one step is required to split the vertex to its separate cluster in C, and one step is required to join the vertex with a different cluster to reach B.

The curious reader may have a hunch now: the two numbers in the output of `split_join_distance()` actually belong to the distance of C from A

and B, respectively, and since the split-join distance is the sum of these two distances, the two numbers returned by `split_join_distance()` always sum up to the output of `compare()`. Let us see how our simple clustering method performs when the performance is measured using the split-join distance:

4.28.1)
```
split_join_distance(ground_truth, dist_memb)
```

```
distance12 distance21
         8          8
```

Finally, note that if one clustering is almost a subclustering of the other, the corresponding number in the tuple returned by `split_join_distance()` would be close to zero.

4.3.1.4 Measures based on information theory

The split-join distance is very useful and intuitive, but it suffers from the so-called "problem of matching". In other words, the algorithm of the split-join distance constructs the common subclustering of a clustering pair A and B by finding the best one-to-one match between each cluster in A and B, and it is only the best matching part that counts in the value of the final index; the measure completely ignores the unmatched parts of the clusterings. Consider the example on Fig. 4.6.

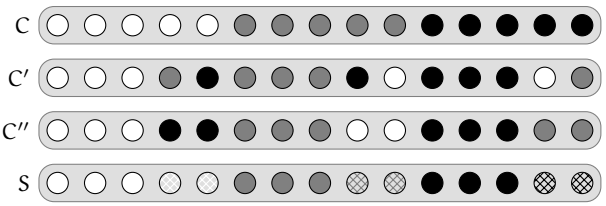


Fig. 4.6 A desired clustering C and two disrupted clusterings C′ and C″. Each row is a clustering, and vertices with the same shade and pattern are in the same cluster. Intuitively, C is closer to C′ı than to C′. However, C is exactly as far from C′ as from C″ according to the split-join distance. S is the common subclustering of C, C′ and C″.

Fig 4.6 shows three clusterings: C, C′ and C″. Each clustering has three clusters. The clusters in C′ were obtained from the clusters of C by redistributing 40% of the vertices in each cluster among the other two clusters evenly. The clusters in C″ were obtained by moving 40% of the vertices from cluster i to cluster $i \bmod 3+1$ (where $i \in 1,2,3$). Our intuition says that C″ is farther from C than C′ is because it seems more "disorganised" than C′. However, the split-join distance is exactly the same:

```
4.29.1) c  <- c(1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3)
    2) c1 <- c(1, 1, 1, 2, 3, 2, 2, 2, 3, 1, 3, 3, 3, 1, 2)
    3) c2 <- c(1, 1, 1, 3, 3, 2, 2, 2, 1, 1, 3, 3, 3, 2, 2)
    4) compare(c, c1, method = "split.join")
```

```
[1] 12
```

```
    6) compare(c, c2, method = "split.join")
```

```
[1] 12
```

To this end, Meilă (2003) proposed the *variation of information* (VI) measure from information theory to quantify the distance of two clusterings from one another. The variation of information is also a metric, but it makes a distinction between C' and C''. We will shortly see how.

variation of information

Imagine that a random vertex is drawn from a clustering A. The probability of drawing a vertex from cluster i is given by $p_i = n_i/n$, where n_i is the number of vertices in cluster i and n is the total number of vertices in the clustering. The set of probabilities p_i define a discrete random variable corresponding to clustering A. Similarly, we can define a set of probabilities q_i for another clustering B. We then define the *entropy* of the random variable corresponding to the clusterings as follows:

entropy

$$H(A) = -\sum_{i=1}^{k_A} p_i \log p_i \tag{4.1}$$

$$H(B) = -\sum_{i=1}^{k_B} q_i \log q_i \tag{4.2}$$

where k_A and k_B denote the number of clusters in A and B, respectively. The entropy of a random variable quantifies the uncertainty in the value of the random variable; in other words, higher entropy means that we are less certain about the cluster affiliation of a randomly drawn vertex from the given clustering. The lowest possible value of the entropy is zero when there is only one cluster. Note that the entropy does not depend on the number of vertices, only on the relative sizes of each cluster in the clustering.

The other concept we will need besides the entropy is the *mutual information* between two clusterings. Loosely speaking, the mutual information between clusterings A and B quantifies how much we would know about clustering B if we knew clustering A. Let p_{ij} denote the number of vertices that are in cluster i in clustering A and in cluster j in clustering B. The mutual information between A and B is then defined as:

mutual information

$$I(A, B) = \sum_{i=1}^{k_A} \sum_{j=1}^{k_B} p_{ij} \log \frac{p_{ij}}{p_i q_j} \tag{4.3}$$

Here we assume that $0 \log 0 = 0$. The mutual information is always non-negative and symmetric, and it can never exceed the entropy of any of the two clusterings involved. The variation of information is then defined as follows:

$$\mathrm{VI}(A, B) = [\mathrm{H}(A) - \mathrm{I}(A, B)] + [\mathrm{H}(B) - \mathrm{I}(A, B)] = \mathrm{H}(A) + \mathrm{H}(B) - \mathrm{I}(A, B) \quad (4.4)$$

The first term in square brackets in Eq. 4.4 is the amount of information in clustering A that is left "unexplained" even if we know clustering B; similarly, the second term is the amount of unexplained information in clustering B even if we know clustering A. The sum of the two terms gives us the variation of information metric. Since the sums in Eq. 4.3 take into account every possible pairing of clusters, the measure does not suffer from the problem of matching and gives a different distance for our clusters C, C′ and C″ on Fig. 4.6:

```
4.31.1) compare(c, c1, method = "vi")
```

```
[1] 1.900541
```

```
3) compare(c, c2, method = "vi")
```

```
[1] 1.346023
```

The variation of information metric is really in agreement with our intuition about the distances of C′ and C″ from C. Finally, let us see how our simple clustering algorithm introduced in Section 4.3.1 performs according to the VI measure:

```
4.33.1) vi <- compare(ground_truth, dist_memb, method = "vi")
    2) vi
```

```
[1] 1.090122
```

Since the theoretical maximum of the VI metric for a clustering of n vertices is $\log n$, the above value is relatively small, indicating that the clustering our algorithm has found is indeed close to the ground truth:

```
4.34.1) vi / log(gorder(karate))
```

```
[1] 0.3091352
```

normalized mutual information

An alternative measure called *normalized mutual information* (NMI) was also suggested by Fred and Jain (2003), and it was later applied to community detection in graphs by Danon et al (2005). Contrary to the variation of information metric, the NMI is a similarity measure, but it can also be calculated from the mutual information and the entropies as follows:

$$\mathrm{NMI}(A, B) = 2\frac{\mathrm{I}(A, B)}{\mathrm{H}(A) + \mathrm{H}(B)} \quad (4.5)$$

The normalized mutual information takes its maximum value of 1 when the clusterings are equivalent, and its minimum value of zero is attained when

they are independent from each other in a probabilistic sense. igraph can also calculate the normalized mutual information by using `'method = "nmi"'` in `compare()`:

```
4.35.1) compare(ground_truth, dist_memb, "nmi")
```

```
[1] 0.2116744
```

4.3.2 Internal quality measures

Internal quality measures are the counterparts of external quality measures presented in Section 4.3.1. They differ in one fundamental aspect: no gold standard clustering is required in order to calculate an internal quality measure on a clustering. This makes them very useful in scenarios where the ground truth is not known, which is the case in almost all research problems.

An internal quality measure is always built upon some kind of a prior assumption about how good clusters should look like. Intuitively, we expect clusters to be dense (at least when compared to the density of the network as a whole) and we also expect that there are only a small number of edges that are situated between different clusters. Given a good internal quality measure, one can devise a clustering algorithm that optimizes this measure directly or indirectly in order to obtain a good clustering of a network. However, we will see that naïve internal quality measures often attain their optima at trivial clusterings, necessitating the development of more sophisticated scores that are significantly harder to optimize.

4.3.2.1 Density and subgraph density

igraph can readily calculate the *density* of an entire graph, which is simply defined as the number of edges in the graph divided by the total number of possible edges (assuming that there is at most edge between vertex pairs and that there are no loop edges). Since there are $n(n-1)/2$ possibilities to create an edge in a simple undirected graph with n vertices and $n(n-1)$ possibilities to do the same in a simple directed graph, the density formula of a graph G with n vertices and m edges is straightforward:

density

$$\rho(G) = \begin{cases} \frac{2m}{n(n-1)} & \text{if } G \text{ is undirected} \\ \frac{m}{n(n-1)} & \text{if } G \text{ is directed} \end{cases} \tag{4.6}$$

Density calculation is performed by calling the `density()` function. We still use the Zachary karate club network here:

density()

```
4.36.1) density(karate)
```

```
Error in density.default(karate): argument 'x' must be numeric
```

To see how dense the two factions are in the karate club network, we will create a helper function to calculate the density of a subgraph:

subgraph density

```
4.37.1) subgraph_density <- function(graph, vertices) {
    2)   graph %>%
    3)     induced_subgraph(vertices) %>%              #label{vrb:subgraph}
    4)     density()
    5) }
```

Line **??** simply creates the subgraph consisting of the vertex indices given in `'vertices'` and then returns its density. It could not be simpler. Since we already have the ground truth of the Zachary karate club network in `'ground_truth'`, we can start with checking whether our assumptions about the intra-cluster edge densities hold:

```
4.38.1) subgraph_density(karate, ground_truth[[1]])
```

```
Error in density.default(.): argument 'x' must be numeric
```

```
    3) subgraph_density(karate, ground_truth[[2]])
```

```
Error in density.default(.): argument 'x' must be numeric
```

Indeed the two subgraphs are denser than the network as a whole. So, what is wrong with using the densities of the subgraphs as a quality score? Take a look at the graph shown on Fig. 4.7 and one of its many possible clusterings. Obviously, a ring graph has no clear-cut community structure; every vertex in the graph is connected to two neighbors, and each of them are essentially equal from a topological point of view. However, the density of each cluster in the clustering on the figure is 1, which is the maximal possible density for simple graphs. Clearly, the cluster densities alone are not enough, and there is a similar problem with the inter-cluster edge density as well: a trivial clustering that puts every vertex in the same cluster would obviously minimize the inter-cluster edge density to zero.

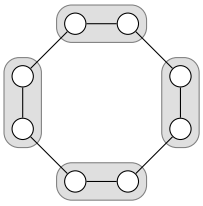


Fig. 4.7 A ring graph of 8 vertices and one of its possible clusterings. Each cluster has maximal density, but the whole clustering is still meaningless.

4.3.2.2 Modularity

Newman (2004) recognized that it is not the density or the number of edges within a cluster that is important when one judges the quality of a clustering. Good clusters are not only dense, but the edges within a cluster are also unlikely to have been placed there by random chance. To this end, he introduced the *modularity* measure, which since then became one of the most widely used internal quality measures for graph clustering.

modularity

Let p_{ij} be the probability of the event that the edge between vertices i and j in a graph exists purely by chance; in other words, it has nothing to do with the modular structure of the graph. The matrix of p_{ij} values is often called the *null model* of the graph, as it represents our prior assumption about how a randomly organized graph would look like. The popular choices for p_{ij} will be discussed later. The modularity of a clustering C of the graph G (having n vertices and m edges) is then defined as the difference between the *observed* number of edges within clusters and the number of edges that would have been *expected* if all the edges were placed randomly:

null model

$$Q(G) = \frac{1}{2m} \sum_{i=1}^{n} \sum_{j=1}^{n} (A_{ij} - p_{ij})\, \delta_{ij} \tag{4.7}$$

where A_{ij} is the element in row i and column j of the adjacency matrix (see Section 1.2.6) and δ_{ij} is 1 if and only if vertices i and j are in the same cluster and zero otherwise. The double sum effectively iterates over all intra-cluster vertex pairs in the graph, and calculates the difference between the observed edge count (A_{ij}) and the edge count we expect from the null model (p_{ij}) for each such pair. The initial $1/2m$ term is for normalization purposes only – it allows one to compare the modularity scores of graphs with different edge counts. A clustering with zero modularity has exactly as many edges within clusters as we would expect in a graph that was generated by the null model, positive modularity scores represent good clusterings and negative scores represent bad ones where there are *less* edges in clusters than what we would expect in a randomly generated graph. It is commonly said that a modularity score larger than 0.3 is usually considered to be an indicator of significant modular structure, but this is not always the case: graphs that are *designed* to be random may show significant fluctuations in their maximal modularity scores, and it is easy to find structureless graphs with high modularity. We will talk about the limitations of the modularity measure later in Section 4.4.2.4.

In the overwhelming majority of the cases, the null model used in the definition of modularity is the so-called configuration model of Molloy and Reed (1995). This model generates graphs where the expected degree of each vertex is prescribed. It can be shown that the probability of the existence of an edge between vertex i with degree k_i and vertex j with degree k_j is $k_i k_j / 2m$, yielding the most widely used form of the modularity measure:

$$Q(G) = \frac{1}{2m} \sum_{i=1}^{n} \sum_{j=1}^{n} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta_{ij} \tag{4.8}$$

In this null model, an edge is more likely to exist between vertices with higher degrees than between vertices with lower degrees. This is a plausible assumption in many cases: a connection between two people in a social network is less likely to be significant if those people have large social circles (since popular people are more likely to know others only because they are popular in general), and an observed interaction between proteins in a protein-protein interaction network may also be of small importance if one of the proteins is promiscuous and can interact with almost any other protein in the cell because of its general biochemical properties. The bottom line is that higher modularity scores correspond to partitions where the edges within the communities are not likely to have been intra-cluster edges in a randomly generated graph which has the same degree sequence as the graph being studied *and* the same vertex partition. Obviously, the trivial clustering where every vertex is in the same community has zero modularity as we would see exactly the same number of edges in the community if we rewired the graph while preserving the degrees of the vertices. (For a more formal proof, see Newman (2004)). The other trivial clustering where every vertex is in a separate community has a negative modularity. The optimum lies somewhere between the two extremities, and this optimum is the one that many community detection algorithms try to find. However, finding the partition that yields the optimal modularity is a hard problem; we will learn more about that in Section 4.4.2.1.

modularity()

igraph can readily calculate the modularity of any clustering of a given graph using the `modularity()` function. This can be called on a `communities` object, or directly on graph, supplying the community structure as a membership vector:

```
4.40.1) modularity(karate, membership(ground_truth))
        [1] 0.3714661
    3) modularity(ground_truth)
        [1] 0.3714661
```

Let us also check the modularity of the two trivial clusterings for the karate club network:

```
4.42.1) modularity(karate, rep(1, gorder(karate)))
        [1] 0
    3) modularity(karate, seq_len(gorder(karate)))
        [1] -0.04980276
```

The modularity score also has a weighted variant:

$$Q(G) = \frac{1}{w} \sum_{i=1}^{n} \sum_{j=1}^{n} \left(w_{ij} - \frac{w_i w_j}{w} \right) \delta_{ij} \tag{4.9}$$

where $\mathbf{W} = [w_{ij}]$ is the weighted adjacency matrix, w_i is the total weight of edges incident on node i, and $w = \sum_{i=1}^{n} w_i$. The weighted variant is similar to the unweighted case, but the elements of the adjacency matrix are replaced by the elements of the weighted adjacency matrix $\mathbf{W}$, and the *strength* of a vertex (w_i) is used instead of its degree. The strength of a vertex is the sum of the weights of the edges incident on that vertex, and it can be calculated using the `strength()`, just like regular degrees are calculated by the `degree()` method. The `strength()` method also takes the name of the edge attribute containing the edge weights (or a list of the weights themselves) as a parameter.

strength

4.3.2.3 The dynamical formulation of modularity

One can also obtain the modularity score as a quality function from an entirely different approach. Let us consider a set of random walkers that move on the nodes of the graph. At any given time step, each random walker is on exactly one of the nodes of the graph. At the end of the time step, the random walker chooses one of the edges incident on the node and moves along the edge to arrive at the other end when the next time step beings. Let the density of random walkers on node i at time step n denoted by $p_i^{(n)}$. The dynamics is then given by the following simple equation:

$$p_i^{(n+1)} = \sum_j \frac{A_{ij}}{k_j} p_j^{(n)} \tag{4.10}$$

Assuming that the graph is undirected, connected, non-bipartite and simple, it is easy to show that the densities $\vec{p}^{(n)}$ converge to a stationary distribution $\vec{p}^{\infty}$ as n approaches infinity. Furthermore, the stationary solution of the dynamics is generically given by $p_i^{\infty} = k_i/2m$.

Let us now imagine a graph with densely connected modules and only a few links between them. A random walker that starts in one of the modules is likely to get stuck there for a long time before eventually finding an edge that leads out of the module and into another one. In other words, the probability of a random walker being in community C_l in time step n and staying there in time step $n+1$ is larger than the probability of finding two *independent* random walkers both in C_l as the community affiliations of consecutive locations of a random walker are correlated. In the stationary state, the probability of finding a random walker in C_l is equal to:

$$\sum_{i,j \in C_l} \frac{A_{ij}}{k_j} \frac{k_j}{2m} \tag{4.11}$$

which was obtained by simply replacing $p_j^{(n)}$ with p_j^{∞} in Eq. 4.10 and summing it for all i, j pairs where both of them are in C_l. At the same time, the probability of two independent walkers both being in C_l is simply the product of the stationary probabilities p_i^{∞} and p_j^{∞}:

$$\sum_{i,j \in C_l} \frac{k_i k_j}{4m^2} \tag{4.12}$$

Let us then take the differences between the two probabilities and add them together for all communities and all internal vertex pairs:

$$\sum_{C_l \in C} \sum_{i,j \in C_l} \left(\frac{A_{ij}}{2m} - \frac{k_i k_j}{4m^2} \right) \tag{4.13}$$

Is it getting familiar? By moving $\frac{1}{2m}$ outside the parentheses and introducing δ_{ij} to denote whether there exists an l such that $i \in C_l$ and $j \in C_l$, we obtain the now familiar modularity score in Eq. 4.8.

4.4 Finding communities in networks

In the previous section, we have introduced a few measures that can be used to quantify how good a given set of communities are, but nothing has been said about how to find an optimal or near-optimal community structure in a network. This section will describe a few methods implemented by igraph that can be used to solve this problem and also a few other popular methods that are not implemented natively but can be added in only a few lines of code.

4.4.1 The Girvan-Newman algorithm

Girvan-Newman algorithm

The *Girvan-Newman algorithm* (also called the edge betweenness method) was one of the earliest community detection methods published in the network science literature (Girvan and Newman, 2002). The algorithm builds on the idea that the number of shortest paths passing through an intra-community edge should be low (since a community is densely connected, hence there are many alternative paths between vertices of the same community), while

inter-community edges are likely to act as bottlenecks that participate in many shortest paths between vertices of different communities.

We already know from Section ?? that the number of shortest paths passing through a given edge is given by its *edge betweenness*. The Girvan-Newman algorithm proceeds by iteratively removing the edge with the largest edge betweenness from the graph until it falls apart into separate components. The output of the algorithm is therefore a hierarchical clustering, or a `communities()` instance in igraph terms:

edge betweenness

```
4.44.1) dendrogram <- cluster_edge_betweenness(karate)
    2) dendrogram
```

```
IGRAPH clustering edge betweenness, groups: 6, mod: 0.35
+ groups:
  $‘1‘
  [1] "Mr Hi"    "Actor 2"  "Actor 4"  "Actor 8"  "Actor 12"
  [6] "Actor 13" "Actor 18" "Actor 20" "Actor 22"

  $‘2‘
  [1] "Actor 3"  "Actor 10" "Actor 14" "Actor 29"

  $‘3‘
  [1] "Actor 5"  "Actor 6"  "Actor 7"  "Actor 11" "Actor 17"

  + ... omitted several groups/vertices
```

For a visual representation of the dendrogram, use the `plot_dendrogram()` command. The dendrogram should be similar to the one you see on Fig. 4.8(b).

plot_dendrogram()

Unfortunately the algorithm has two drawbacks. First, it is computationally intensive since the edge betweenness scores have to be re-calculated after every edge removal. Calculating the edge betweenness is an $O(nm)$ process on an unweighted graph with n vertices and m edges using Brandes' algorithm (Brandes, 2001), and this is repeated $m-1$ times (once for each edge removal), therefore the time complexity of the algorithm is $O(nm^2)$, which renders it unusable for large networks. The other disadvantage is that the algorithm is not able to select the level where the dendrogram should be cut in order to obtain a flat clustering. However, it is fairly easy to use the modularity score to flatten the clustering at the level where the modularity is maximal. igraph even calculates this level for you in advance, so `membership()` returns the community structure of the best cut in terms of modularity.

```
4.45.1) membership(dendrogram)
```

```
   Mr Hi  Actor 2  Actor 3  Actor 4  Actor 5  Actor 6  Actor 7
       1        1        2        1        3        3        3
 Actor 8  Actor 9 Actor 10 Actor 11 Actor 12 Actor 13 Actor 14
```

```
       1        4        2        3        1        1        2
Actor 15 Actor 16 Actor 17 Actor 18 Actor 19 Actor 20 Actor 21
       4        4        3        1        4        1        4
Actor 22 Actor 23 Actor 24 Actor 25 Actor 26 Actor 27 Actor 28
       1        4        5        5        5        6        5
Actor 29 Actor 30 Actor 31 Actor 32 Actor 33   John A
       2        6        4        4        4        4
```

Finally, since we know the ground truth for the Zachary karate club network, we can compare the clustering proposed by the edge betweenness algorithm with the ground truth:

```
compare_using_all_methods <- function(cl1, cl2) {
  ## List all possible values of compare 'methods'
  methods <- args(compare) %>% as.list()%>% .$method %>% eval()
  sapply(methods, compare, comm1 = cl1, comm2 = cl2)
}
compare_using_all_methods(dendrogram, ground_truth)
```

```
           vi           nmi    split.join          rand
    1.1355464     0.5178731    17.0000000     0.6844920
adjusted.rand
    0.3581858
```

These results are not very impressive; even our simple clustering algorithm that we experimented with in Section 4.3.1 produced similar scores to this. However, note that our algorithm was more informed than the edge betweenness algorithm: we knew in advance that we should look for two clusters and that 'Mr Hi' and 'John A' are in different clusters. Let us cut the dendrogram at a different place to obtain two clusters and see whether the results have improved!

```
clustering <- cut_at(dendrogram, no = 2)
clustering
```

```
 [1] 2 2 1 2 2 2 2 2 1 1 2 2 2 1 1 1 2 2 1 2 1 2 1 1 1 1 1 1 1 1
[30] 1 1 1 1 1
```

```
compare_using_all_methods(clustering, ground_truth)
```

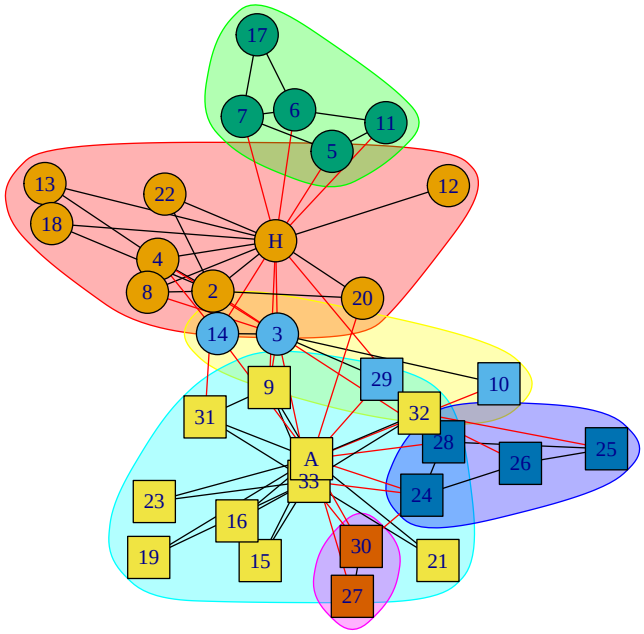
```
           vi           nmi    split.join          rand
    0.3685289     0.7307867     4.0000000     0.8859180
adjusted.rand
    0.7718469
```

This is much better; for instance, the split-join distance shows that it is enough to move two vertices from one cluster to the other in our clustering to recover the ground truth exactly, and the edge betweenness method was

able to detect the two clusters without knowing in advance that 'Mr Hi' and 'John A' should be in different clusters.

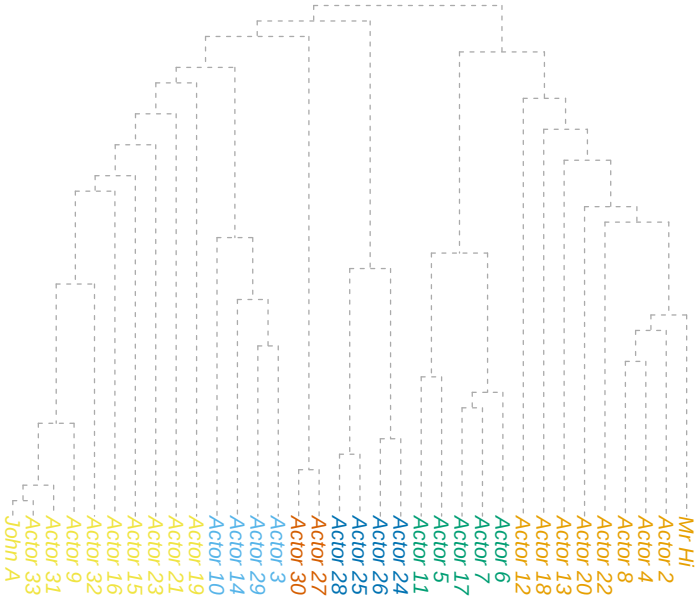
Now we briefly show how can one inspect the results of a clustering visually. Cluster plots will be discussed in detail in Section 3.5.1.

```
4.49.1) V(karate)[Faction == 1]$shape <- "circle"
    2) V(karate)[Faction == 2]$shape <- "square"
    3) plot(dendrogram, karate)
```



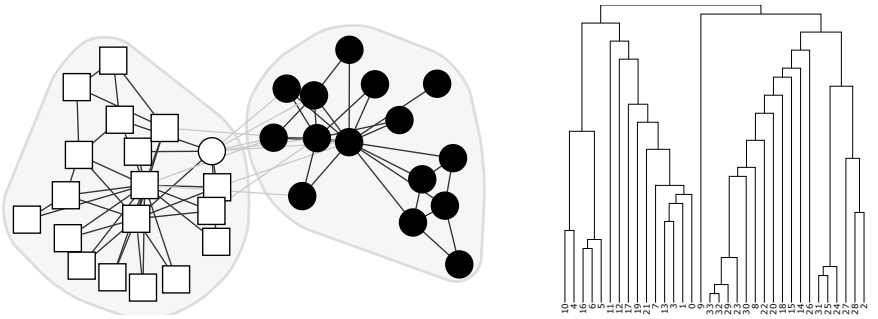
The first two lines assign shapes to the vertices based on the ground truth. Colors are be used to distinguish the members of the different clusters. Fig. 4.8(a) shows the result. The dendrogram on Fig. 4.8(b) can also be drawn with igraph:

```
4.50.1) plot_dendrogram(dendrogram, direction = "downwards")
```



4.4.2 Modularity-based methods

This subsection presents several clustering methods implemented in igraph that have at least one thing in common: they all strive to maximize the modularity of the obtained partition. The problem of modularity maximization was shown to be NP-hard by Brandes et al (2008) – informally speaking, this means that modularity maximization is at least as hard as the hardest problems in NP, and we cannot expect to find an algorithm that runs in polynomial time and provides a clustering with the highest modularity in an arbitrary graph G, unless it turns out that P = NP. (For more information about P, NP, NP-hardness and other concepts in computational complexity theory. Most of the algorithms in igraph trade off the exactness of the solution for a reasonable computational cost in large graphs, with one notable



(a) Clustering (b) Dendrogram

Fig. 4.8 Left: clusters detected in the Zachary karate club network using the Girvan-Newman algorithm (Girvan and Newman, 2002). Circles and squares represent the original division observed by Zachary (1977); shades of gray show the detected clusters when the dendrogram is cut to obtain two clusters. Right: the dendrogram from which the clustering was obtained.

exception which sacrifices performance but always provides a clustering with maximum modularity.

4.4.2.1 Exact modularity maximization

Brandes et al (2008) has shown that modularity maximization can be cast into the form of a linear integer optimization problem. Linear integer optimization problems are a special case of optimization problems in general. In these problems, the variables are integers which are subjects of linear equality and inequality constraints. The goal is to maximize the value of a goal function which is also a linear function of the variables.

The formulation of Brandes et al (2008) contains n^2 binary decision variables δ_{uv} for a graph with n vertices. δ_{uv} will be one if and only if vertices u and v are in the same cluster and zero otherwise. The constraints are as follows:

Reflexivity constraints. For all u, $\delta_{uu} = 1$; in other words, each vertex is in the same cluster with itself.

Symmetry constraints. For all u and v, $\delta_{uv} = \delta_{vu}$; in other words, if u and v are in the same (different) cluster, then v and u are also in the same (different) cluster.

Transitivity constraints. For all u, v and w, it holds that $\delta_{uv} + \delta_{vw} - 2\delta_{uw} \leq 1$. This can be true only if $\delta_{uv} = 1$ and $\delta_{vw} = 1$ implies $\delta_{uw} = 1$, therefore it cannot happen that u is in the same cluster with both v and w, but v and w are in different clusters.

The goal function to be optimized is the modularity function itself, as presented in Eq. 4.8:

$$Q(G) = \frac{1}{2m} \sum_{i=1}^{n} \sum_{j=1}^{n} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta_{ij} \tag{4.14}$$

where A_{ij} is 1 if and only if vertices i and j are connected (zero otherwise), k_i is the degree of vertex i and m is the number of edges in G.

Pruning redundant variables leaves us with $\binom{n}{2}$ variables and $\binom{n}{3}$ constraints, which is clearly a cause for concern in the case of large graphs: a relatively modest graph with 1000 vertices would require 499,500 variables and more than 166 million constraints. Even the state-of-the-art solvers for linear integer optimization problems can not cope with problems of this size, therefore this approach is mainly of theoretical interest. igraph nevertheless includes an algorithm that re-formulates modularity optimization as a linear integer problem and uses the GNU Linear Programming Kit (GLPK, see `http://www.gnu.org/software/glpk/`) to solve it. The corresponding igraph function is `cluster_optimal()`. Since Zachary's karate club network is well below the practical size limit of this approach, we can use igraph to find the partition corresponding to the maximum modularity and compare it with the modularity of the five-cluster partition we have obtained from the Girvan-Newman method:

cluster_optimal()

```
4.51.1) optimal <- cluster_optimal(karate)
    2) girvan_newman <- cluster_edge_betweenness(karate)
    3) rbind(girvan_newman %>% membership(),
    4)       optimal %>% membership(),
    5)       ground_truth %>% membership())
```

```
     Mr Hi Actor 2 Actor 3 Actor 4 Actor 5 Actor 6 Actor 7
[1,]     1       1       2       1       3       3       3
[2,]     1       1       1       1       2       2       2
[3,]     1       1       1       1       1       1       1
     Actor 8 Actor 9 Actor 10 Actor 11 Actor 12 Actor 13
[1,]       1       4        2        3        1        1
[2,]       1       3        3        2        1        1
[3,]       1       2        2        1        1        1
     Actor 14 Actor 15 Actor 16 Actor 17 Actor 18 Actor 19
[1,]        2        4        4        3        1        4
[2,]        1        3        3        2        1        3
[3,]        1        2        2        1        1        2
     Actor 20 Actor 21 Actor 22 Actor 23 Actor 24 Actor 25
[1,]        1        4        1        4        5        5
[2,]        1        3        1        3        4        4
[3,]        1        2        1        2        2        2
     Actor 26 Actor 27 Actor 28 Actor 29 Actor 30 Actor 31
```

```
[1,]        5        6        5        2        6        4
[2,]        4        3        4        4        3        3
[3,]        2        2        2        2        2        2
     Actor 32 Actor 33 John A
[1,]        4        4      4
[2,]        4        3      3
[3,]        2        2      2
```

An important lesson to learn here is that the partition with the maximal modularity (i.e. `'optimal'`) is not necessarily the partition which is the ground truth (i.e. `'ground_truth'`), at least when such an external ground truth is available. In fact, the result of the Girvan-Newman method is much closer to the optimal partition than the ground truth, both in terms of modularity and in terms of external quality measures. Here we calculate both the modularity scores of the three partitions and their distances using the variation of information metric of Meilă (2003):

```
4.52.1) modularity(girvan_newman)
[1] 0.345299
3) modularity(optimal)
[1] 0.4449036
5) modularity(ground_truth)
[1] 0.3714661
7) compare(girvan_newman, optimal, method = "vi")
[1] 0.7790644
9) compare(ground_truth, optimal, method = "vi")
[1] 0.629254
```

Since the variation of information is a distance metric, a lower VI score of `'girvan_newman'` also indicates that it is closer to the optimum than the ground truth is.

4.4.2.2 Heuristic optimization strategies

It was shown in the previous subsection that the exact optimization of modularity is not feasible in large graphs. To this end, many heuristics have been proposed in the literature that may not return a clustering with maximal modularity but are capable of returning a clustering whose modularity is close to the optimum. The first heuristic we will discuss here is the *fast greedy modularity optimization*, the method originally proposed by Newman

fast greedy modularity optimization

(2004) in the paper that describes modularity as a quality function.

The procedure starts from a configuration where every vertex is considered to be in a separate community. The modularity of such a partition can be calculated easily. Assuming no loop edges (which is usually the case):

$$Q(G) = \frac{1}{2m}\sum_{i=1}^{n}\left(a_{ii} - \frac{d_i^2}{2m}\right) = -\left(\frac{d_i}{2m}\right)^2 \tag{4.15}$$

The algorithm maintains a matrix of $'Q_{ij}$ values where $'Q_{ij}$ represents the change in modularity when the clusters corresponding to vertices `i` and `j` are merged. The matrix is sparse; most of its values will be zeros as no modularity gain could be achieved by merging two clusters that are not connected to each other at all.

In each step of the algorithm, the $'Q$ matrix is scanned and the element yielding the maximum increase in modularity is selected. The corresponding two clusters are merged, the values of the $'Q$ matrix are re-calculated and the process continues until only one cluster remains. It can be shown that only a small part of the $'Q$ matrix has to be updated after a merge, making the algorithm scalable up to hundreds of thousands of vertices. For more details see the paper of Clauset et al (2004). The result of the process is a dendrogram, similarly to the Girvan-Newman algorithm, but the algorithm readily provides the point where the dendrogram should be cut: exactly at the point where the maximal $'Q$ value becomes negative for the first time.

The algorithm is implemented by the `cluster_fast_greedy()` function, and it supports both weighted and unweighted graphs. We are going to use the unweighted version of the karate club network, and the weighted `UKfaculty` network from the *`igraphdata`* package as examples here.

```
4.57.1) karate <- delete_edge_attr(karate, "weight")
    2) data(UKfaculty)
```

This network describes the social interactions between the lecturers of three schools in a faculty of a UK university. The interactions are directed and weighted. Since the modularity maximization method works with undirected graphs only, we have to get rid of the directions of the edges first using the `as.undirected()` function:

`as.undirected()`

```
4.58.1) UKfaculty <- as.undirected(UKfaculty, edge.attr.comb = "sum")
    2) summary(UKfaculty)
```

```
IGRAPH 87a8668 U-W- 81 577 --
+ attr: Type (g/c), Date (g/c), Citation (g/c), Author
| (g/c), Group (v/n), weight (e/n)
```

The `'edge.attr.comb = "sum"'` argument specifies that the weights of the edges pointing in opposite directions (i.e. from A to B and from B to A) should be added up to obtain the weight of the corresponding undirected

edge. Other choices are also possible; e.g., we could have used `'"mean"'` to take the mean of the two weights, or `'"max"'` to keep the larger weight only, etc.

First, let us calculate a near-optimal clustering of the Zachary karate club network using the fast greedy modularity optimization, and compare its modularity score with the optimal one that we determined in the previous subsection!

```
4.59.1) dendrogram <- cluster_fast_greedy(karate)
    2) dendrogram
```

```
IGRAPH clustering fast greedy, groups: 3, mod: 0.38
+ groups:
  $`1`
  [1] "Mr Hi"    "Actor 5"  "Actor 6"  "Actor 7"  "Actor 11"
  [6] "Actor 12" "Actor 17" "Actor 20"

  $`2`
   [1] "Actor 9"  "Actor 15" "Actor 16" "Actor 19" "Actor 21"
   [6] "Actor 23" "Actor 24" "Actor 25" "Actor 26" "Actor 27"
  [11] "Actor 28" "Actor 29" "Actor 30" "Actor 31" "Actor 32"
  [16] "Actor 33" "John A"

  + ... omitted several groups/vertices
```

For weighted graphs, the weights are automatically detected and used:

```
4.60.1) dendrogram <- cluster_fast_greedy(UKfaculty)
    2) dendrogram
```

```
IGRAPH clustering fast greedy, groups: 5, mod: 0.56
+ groups:
  $`1`
   [1]  2  8 11 15 18 19 21 25 29 31 34 35 39 41 43 46 57 58 79

  $`2`
   [1] 24 32 37 38 48 50 52 54 55 64 70

  $`3`
   [1]  1  3  4  9 17 36 44 45 53 59 60 61 62 73 74 75 78 81

  $`4`
  + ... omitted several groups/vertices
```

Later on, Newman (2006) proposed an alternative optimization strategy for the modularity score. The so-called *leading eigenvector method* is a top-down method that starts from a single giant community that contains every vertex, and splits it iteratively into smaller fragments such that each split

leading eigenvector method

creates two new communities from one. This means that instead of solving the general modularity problem for any number of communities, one has to find only an optimal *split* within a community.

Given a single giant community, let us introduce a vector $\vec{s} = [s_1, s_2, \ldots, s_n]$ such that s_i is 1 if vertex i should be on one side of the split and -1 if the vertex should be on the other side. The modularity of the split can then be written as:

$$Q = \frac{1}{4m} \sum_{i=1}^{n} \sum_{j=1}^{n} b_{ij} s_i s_j \tag{4.16}$$

where b_{ij} is the well-known term from the original modularity formula in Eq. 4.8:

$$b_{ij} = A_{ij} - \frac{k_i k_j}{2m} \tag{4.17}$$

By denoting the matrix of b_{ij} values with $\mathbf{B}$, we can reduce the modularity formula to two matrix-vector multiplications:

$$Q = \frac{1}{4m} \vec{s}^T \mathbf{B} \vec{s} \tag{4.18}$$

Newman (2006) proposed to choose $\vec{s}$ by finding the leading eigenvector $\vec{s}^*$ (i.e. the eigenvector corresponding to the largest eigenvalue) of $\mathbf{B}$ and replacing positive coordinates in $\vec{s}^*$ with 1 and negative coordinates with -1 to obtain $\vec{s}$ and thus the split itself. Cases when all the coordinates of $\vec{s}^*$ are positive (or negative) correspond to situations when it is not possible to improve modularity by splitting a community further, serving as a natural stopping condition for the algorithm. Newman's leading eigenvector method therefore proceeds as follows:

1. Create an empty queue that will contain clusters, and an empty set that will contain the clusters of the final result.
2. Initialize the queue with a cluster containing all the vertices.
3. If the queue is empty, terminate the algorithm and return the result set.
4. Remove the first cluster from the queue and find the corresponding modularity matrix $\mathbf{B}$.[1]
5. Find the leading eigenvector $\vec{s}^*$ of $\mathbf{B}$.
6. Calculate $\vec{s} = \mathrm{sgn}(\vec{s}^*)$ where sgn denotes the signum function. Vertices corresponding to zeros in $\vec{s}^*$ may be grouped with either side of the clustering without any change in modularity.
7. If all the values in $\vec{s}$ are positive (or all the values are negative), append the cluster to the result set. Otherwise, split the cluster according to $\vec{s}$ and append the two new clusters to the end of the queue.

[1] Newman (2006) showed that one should not consider the cluster in isolation but also take into account the edges that lead out of the cluster to other clusters in the network, effectively yielding a matrix $\mathbf{B}$ that corresponds to a submatrix of the modularity matrix of the network as a whole.

8. Return to step 3.

The algorithm is implemented in igraph by the `cluster_leading_eigen()`-function:

cluster_leading_eigen()

```
4.61.1) clusters <- cluster_leading_eigen(karate)
    2) clusters
```

```
IGRAPH clustering leading eigenvector, groups: 4, mod: 0.39
+ groups:
  $‘1‘
  [1] "Mr Hi"    "Actor 5"  "Actor 6"  "Actor 7"  "Actor 11"
  [6] "Actor 12" "Actor 17"

  $‘2‘
   [1] "Actor 9"  "Actor 10" "Actor 15" "Actor 16" "Actor 19"
   [6] "Actor 21" "Actor 23" "Actor 27" "Actor 30" "Actor 31"
  [11] "Actor 33" "John A"

  $‘3‘
  + ... omitted several groups/vertices
```

The leading eigenvector method can also be asked to stop when a given number of clusters has been obtained. For instance, the following statement asks for at most two clusters (which means that the algorithm takes at most one step only):

```
4.62.1) clusters <- cluster_leading_eigen(karate, steps = 1)
    2) clusters
```

```
IGRAPH clustering leading eigenvector, groups: 2, mod: 0.37
+ groups:
  $‘1‘
   [1] "Mr Hi"    "Actor 2"  "Actor 3"  "Actor 4"  "Actor 5"
   [6] "Actor 6"  "Actor 7"  "Actor 8"  "Actor 11" "Actor 12"
  [11] "Actor 13" "Actor 14" "Actor 17" "Actor 18" "Actor 20"
  [16] "Actor 22"

  $‘2‘
   [1] "Actor 9"  "Actor 10" "Actor 15" "Actor 16" "Actor 19"
   [6] "Actor 21" "Actor 23" "Actor 24" "Actor 25" "Actor 26"
  [11] "Actor 27" "Actor 28" "Actor 29" "Actor 30" "Actor 31"
  + ... omitted several groups/vertices
```

It is noteworthy that the above clustering is exactly the ground truth, since their split-join distance is zero:

```
4.63.1) compare(clusters, ground_truth, method = "split.join")
```

```
[1] 0
```

4.4.2.3 Spinglass clustering

The clustering method of Reichardt and Bornholdt (2006) is motivated by spin glass models from statistical physics. Such models are used to describe and explain magnetism at the microscopic scale at finite temperatures. Reichardt and Bornholdt (2006) drew an analogy between spin glass models and the problem of community detection on graphs and proposed an algorithm based on the simulated annealing of the spin glass model to obtain well-defined communities in a graph.

A spin glass model consists of a set of particles called *spins* that are coupled by ferromagnetic or antiferromagnetic bonds. Each spin can be in one of k possible states. The well-known Potts model then defines the total energy of the spin glass with a given spin configuration as follows:

$$\mathcal{H}(\vec{\sigma}) = -\sum_{i,j} J_{ij}\delta_{ij} \tag{4.19}$$

where J_{ij} is the strength of the interaction between spins i and j and δ_{ij} is 1 if and only if $\sigma_i = \sigma_j$, zero otherwise. $J_{ij} > 0$ corresponds to ferromagnetic interactions where the spins seek to align to each other, while $J_{ij} < 0$ represents antiferromagnetic interactions where the spins involved prefer to be in different states. $J_{ij} = 0$ means a noninteracting spin pair. Spin flips then occur in the system randomly, and the probability of a given spin configuration is proportional to $e^{-\beta\mathcal{H}(\vec{\sigma})}$, where $\beta = 1/T$ is the *inverse temperature* of the system. At infinite temperature, all the configurations occur with equal probability. At low temperatures, spin configurations with a smaller total energy occur more frequently.

Spins and interactions in the Potts model are very similar to graphs: each spin in the model corresponds to a vertex, and each interaction corresponds to an edge. When all the interactions are ferromagnetic ($J_{ij} = A_{ij}$, i.e. all the spin pairs are either noninteracting or prefer to align with each other), the spin states can be thought about as community indices, and the spin glass model becomes equivalent to a community detection method driven by the total energy function and the temperature of the system.

Reichardt and Bornholdt (2006) proposed to extend the energy function of the Potts model as follows:

$$\mathcal{H}(\vec{\sigma}) = -\sum_{i,j} a_{ij}A_{ij}\delta_{ij} + \sum_{i,j} b_{ij}(1 - A_{ij})\delta_{ij} \tag{4.20}$$

where a_{ij} denotes the contribution of the interaction between vertices i and j to the total energy when the two vertices are interacting ($A_{ij} = 1$), and b_{ij} denotes the contribution when they are noninteracting ($A_{ij} = 0$). Choosing $a_{ij} = 1 - \gamma p_{ij}$ and $b_{ij} = \gamma p_{ij}$ (where p_{ij} is the probability of the interaction between vertices i and j in an appropriately chosen null model, similar to the case of the modularity function), the energy function is further simplified to:

$$\mathcal{H}(\vec{\sigma}) = -\sum_{i,j}(A_{ij} - \gamma p_{ij})\delta_{ij} \tag{4.21}$$

With $\gamma = 1$, this energy function becomes proportional to the modularity score with a multiplicative factor of $-2m$ (cf. Eq. 4.7). In other words, finding a spin configuration with low $\mathcal{H}(\vec{\sigma})$ is equivalent to finding a flat community structure with high modularity if $\gamma = 1$.

Reichardt and Bornholdt (2006) gave efficient update rules for the above energy function, making it possible to apply a simulated annealing procedure to find the ground state of the model that corresponds to a low energy configuration. Their algorithm starts from a random configuration of spins and tries to flip all the spins once in each time step. After each individual spin flip, the energy of the new configuration is evaluated. If the new spin configuration has a lower energy, the flip is accepted unconditionally; otherwise the flip is accepted with probability $e^{-\beta(\mathcal{H}(\vec{\sigma}_{new})-\mathcal{H}(\vec{\sigma}_{old}))}$. At high temperatures (when β is low), proposals that lead to states with a higher energy have a higher chance to get accepted, while low temperatures tend to reject spin flips that increase the total energy of the system. In order to avoid getting stuck in a local minimum in the first few steps, the algorithm starts from a high temperature T. T is then decreased according to a cooling schedule, and the algorithm terminates when T reaches a lower threshold T_0. The spin configuration at T_0 is then returned as the final clustering.

igraph implements the spinglass clustering algorithm in the `cluster_spinglass()` method. The method has the following parameters:

`cluster_spinglass()`

`'weights'` The name of the edge attribute storing the edge weights, or a list of weights, one item corresponding to each edge.

`'spins'` The number of possible spin states. This is equivalent to the maximal number of communities. The actual number of detected communities may be less as some spin states may become empty.

`'parupdate'` Whether the spins should be updated in parallel or one after another. The default is `'FALSE'`, which corresponds to the asynchronous update; i.e. spins that are considered later during a sweep use the new values of spins that have been considered already in this step when the energy function is calculated. During a synchronous (parallel) update, each spin uses the old values of other spins, irrespectively of the order they are considered.

`'start.temp'` The initial temperature T_1 of the system. The default value is 1, and there should be little reason to adjust it to anything else.

`'stop.temp'` The final temperature T_0 of the system. The default value is 0.01.

`'cool.fact'` The cooling factor α, a real number between 0 and 1. At each time step, the current temperature is multiplied by the cooling factor to obtain the new temperature, resulting in an exponentially decaying temperature curve. The total number of steps taken by the algorithm is thus given by $\lceil(\log T_0 - \log T_1)/\log\alpha\rceil$.

`'update.rule'` Specifies the null model used in the simulation (i.e. the values of p_{ij}). The default value is `'"config"'`, which corresponds to the configuration model and optimizes the modularity function if $\gamma = 1$. The other choice is `'"simple"'`, which corresponds to a simple null model where the edges are placed randomly with equal probabilities (all p_{ij} values are equal) and the expected number of edges is equal to the actual edge count of the graph.

`'gamma'` The γ argument of the algorithm that specifies the balance between the importance of present and missing edges within a community. $\gamma = 1$ makes the energy function proportional to the modularity of the partition and assigns equal importance to present and missing edges.

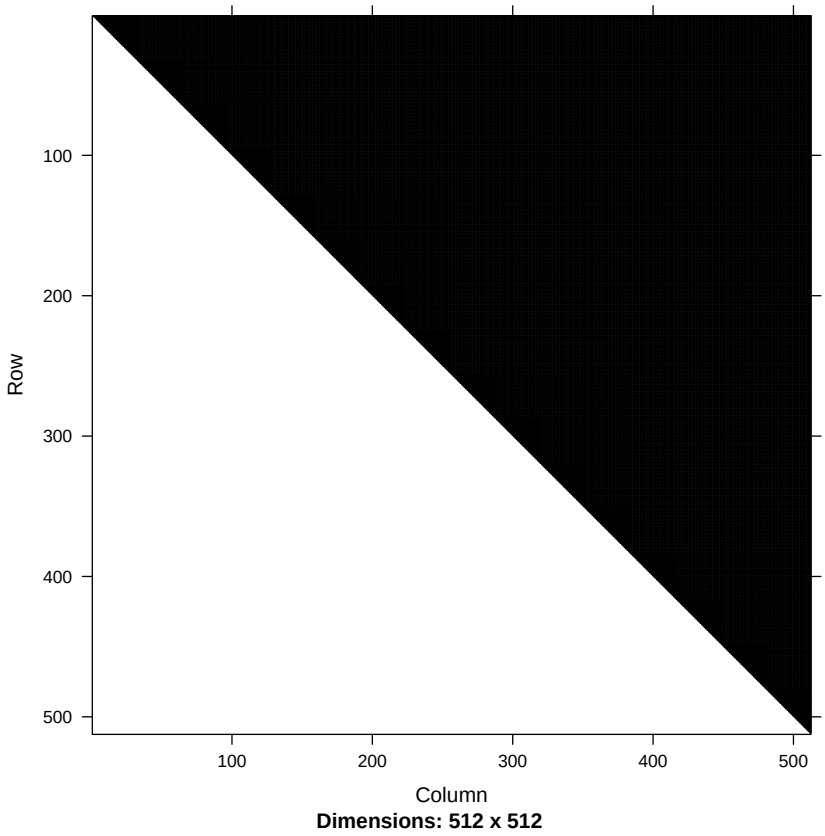
The real power of the spinglass clustering method lies in its γ parameter. As written above, γ adjusts the relative importance of existing edges within a community with respect to the missing edges. In other words, a larger γ parameter would prefer to create communities where the vast majority of edges are present, even if that means creating smaller communities. On the other hand, a smaller γ parameter creates larger and sparser communities. This allows one to explore the community structure of a graph at multiple scales by scanning over a range of γ parameters, possibly discovering different sets of communities in the presence of multi-level structures such as the one seen on Fig. 4.2(c). To illustrate this, we repeat the analysis of Reichardt and Bornholdt (2006) on an artificial graph with a multi-level community structure.

Our artificial graph will be generated as follows. First, we take a graph with 512 vertices. These vertices are grouped into four communities, each community having 128 vertices. For sake of simplicity, community i will contain vertices from index $128(i-1)+1$ to index $128i$. Each community will then have four sub-communities consisting of 32 nodes each. Each vertex is then a member of one community and one of its subcommunities. Vertices have an average of ten links towards other vertices within the same subcommunity, an average of 7.5 links towards other vertices within the same community but in a different subcommunity, and an average of five links towards other vertices not in the same community. In the following, we shall call the larger communities *outer communities* and the subcommunities *inner communities*.

First we create an $n \times n$ matrix that contains the edge probabilities between all pairs of vertices. Since we want an undirected network, we only consider each pair of vertices once, corresponding to the upper right triangle of the matrix. (There are actually much better way to create these kind of random graphs, especially if they are large and sparse, but we don't go into the details here.)

```
4.64.1) library(Matrix)
    2) comm_id <- rep(1:4, each = 128)
    3) sub_comm_id <- rep(1:16, each = 32)
    4) pref_matrix <- outer(1:512, 1:512, function(i, j) {
```

```
  ifelse(sub_comm_id[i] == sub_comm_id[j], 10 / (32 - 1),
         ifelse(comm_id[i] == comm_id[j], 7.5 / (128 - 32),
                10 / (512 - 128)))
})
pref_matrix[lower.tri(pref_matrix, diag = TRUE)] <- 0
image(Matrix(pref_matrix))
```



Now we are ready to generate the graph. We draw an edge between each pair with the probability specified in the matrix.

```
multilevel <- runif(length(pref_matrix)) %>%
  is_less_than(pref_matrix) %>%
  graph_from_adjacency_matrix(mode = "upper")
```

Finally, we run the clustering algorithm at two different resolution scales; the figures of Reichardt and Bornholdt (2006) were created with $\gamma = 1$ and $\gamma = 2.2$, but the implementation in igraph works better with $\gamma = 3$ for the latter case:

```
outer <- cluster_spinglass(multilevel, spins = 32, gamma = 1)
inner <- cluster_spinglass(multilevel, spins = 32, gamma = 3)
```

Since the membership vectors are fairly large, let us simply print the sizes of the clusters first:

```
sizes(outer)
```

```
Community sizes
  1   2   3   4   5   6   7
  2   1 128 127 127   1 126
```

```
sizes(inner)
```

```
Community sizes
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21
31 32  2 31 31 32 27 30  2  5 32 30 32  6 30 32 32 32 33  1 29
```

The result is not perfect, but it is very close to the expected result; the misclassified vertices seem to be put into their own clusters, which can probably be attributed to random noise in the benchmark graph. Your results are likely to be different from the one above because both the graph generation method and the community detection method is randomized. Let us compare the clusterings with the ground truth using the variation of information metric! «>= compare(outer, comm_id, method = "vi") compare(inner, sub_comm_id, method = "vi") @

The split-join distance also shows us that indeed `'inner'` is a subclustering of `'outer'`, since one of the two components of the score is nearly zero. This is not necessarily true for clusterings generated with different γ values, and indicates the presence of a strong hierarchical relationship between communities found at different resolution levels in this graph:

```
split_join_distance(inner, outer)
```

```
distance12 distance21
         7        380
```

4.4.2.4 Limitations of modularity-based methods

The popularity of modularity-based methods in network science is undeniable, but nevertheless the modularity metric itself has a few limitations that one should be aware of. This section will discuss such limitations and how to work around them.

The first misconception related to the concept of modularity is that there exists some kind of a threshold above which it can safely be said that the graph has a significant modular structure. Guimerà et al (2004) have shown that even the so-called Erdős-Rényi networks (where only a randomly selected fraction p of all the possible edges are present, see later on page 151)

have non-zero maximal modularity despite the fact that these networks do not exhibit any modular structure by definition. The maximal modularity of random Erdős-Rényi networks with n vertices and a connection probability of p (denoted by $\mathsf{ER}(n,p)$) agrees well with the following function:

$$Q_{max}(\mathsf{ER}(n,p)) \approx \left(1 - \frac{2}{\sqrt{n}}\right)\left(\frac{2}{np}\right)^{2/3} \tag{4.22}$$

The above formula yields a value very close to 0.5 for $n = 500$ and $p = 0.01$, and this is far above the threshold of 0.3 that some sources cite as a rule of thumb for deciding whether there is a significant modular structure in a network. In other words, modularity maximization algorithms may *overfit* the communities to the data in cases when no real modules are present.

A possible way to work around the problem is to evaluate the maximal modularity of random networks that have the same degree distribution as the original data. Let us assume that a community algorithm detection produced a community structure C for the original network G, and the modularity of C is denoted by $Q(C)$. In order to evaluate the significance of C, we generate a large number of random networks $G_1, G_2, \ldots, G_k$ such that the degree distribution of each network is equal to the degree distribution of G. Note that the definition of modularity implies that the expected modularity C on the ensemble $\{G_1, G_2, \ldots, G_k\}$ is zero by definition, but running our community detection algorithm on each G_i may uncover a *different* C_i community structure that maximizes the modularity of G_i (but not G). We are interested in whether the modularity of C (possibly overfitted to G by the algorithm) is significantly higher than the modularities of C_i's (which are possibly overfitted to the corresponding G_i's). This procedure will be illustrated on the Zachary karate club network:

```
4.70.1) degrees <- degree(karate)
    2) q <- karate %>%
    3)   cluster_fast_greedy() %>%
    4)   modularity()
    5) qr <- replicate(100, sample_degseq(degrees, method = "vl"),
    6)           simplify = FALSE) %>%
    7)   lapply(cluster_fast_greedy) %>%
    8)   sapply(modularity)
    9) sum(qr > q) / 100
```

```
[1] 0
```

The above procedure generates 100 randomized instances of the Zachary karate club network (with the same degree distribution) using the `sample_degseq()` function. We will learn more about this algorithm and its `'method'` argument later in Section **??**; for now, it is enough to know that `'method = "vl"'` ensures that there are no loop edges in the generated network. For each network, we check whether the modularity of the randomized instance

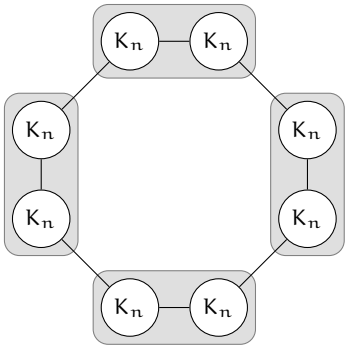
is larger than the modularity `'q'` which we have observed for the karate club network, and calculate the ratio of such networks, in the total 100.

Our result indicates that no randomized networks have community structure with a modularity score that is higher than the one obtained from the original karate club network. This means that this network has significant community structure.

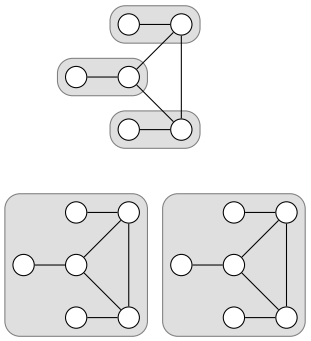
We should note that it is important to use the same community detection method for G_i's as we used for G because the whole procedure is meant to estimate the bias of the specific community detection method that was used to find C. The accuracy of the significance estimation depends on the number of random instances used. In cases when the estimated significance is close to the significance threshold, it is advised to use more random instances in order to come to a more definite conclusion.

resolution limit

The second caveat of modularity-based methods is known as the so-called *resolution limit* of the modularity measure. Fortunato and Barthélemy (2007) have pointed out that the modularity measure has a built-in, intrinsic scale $\sqrt{m}$ (where m is the number of links) such that real modules containing less than $\sqrt{m}$ edges are merged in the configuration that maximizes the modulaity. In case of weighted graphs, the resolution limit is at $\sqrt{w/2}$. A schematic example is shown on Fig. 4.9(a).



(a) Clique ring graph



(b) Resolution limit with replicates

Fig. 4.9 Left: a network containing k identical cliques of n vertices that are connected by single edges. Each clique is denoted by K_n, and the individual vertices are not shown. When k is larger than $n(n-1)+2$, modularity optimization joins adjacent cliques into groups of two or more. These groups are represented by gray shaded areas. Right: the optimal partition of the upper graph consists of three clusters with two vertices each. The lower graph contains two replicates of the upper graph, and the original partition of the upper graph now falls below the resolution limit.

In this example, k identical cliques (complete subgraphs) containing n vertices each are connected by single edges in the shape of a ring. The total number of edges m is $kn(n-1)/2+k$. Intuitively, the communities in this

network should correspond to the k single cliques. The modularity of this configuration is then:

$$Q_{\text{single}} = 1 - \frac{2}{n(n-1)+2} - \frac{1}{k} \tag{4.23}$$

Let us now assume that k is even and consider a community structure where adjacent cliques are paired to form $k/2$ communities, each community containing two cliques. The modularity is then:

$$Q_{\text{paired}} = 1 - \frac{1}{n(n-1)+2} - \frac{2}{k} \tag{4.24}$$

Modularity maximization would find the intuitively correct community structure if $Q_{\text{single}} > Q_{\text{paired}}$, which is true if and only if $n(n-1)+2 > k$. This can further be re-written as $k < m/k+1$, or, neglecting the additive constant, as $k < \sqrt{m}$. It is easy to confirm this on a specific example using igraph:

```
4.71.1) clique_ring <- function(n, k) {
    2)   g <- make_full_graph(n) * k
    3)   g + path(n * 1:k, n)
    4) }
    5) clique_ring(4, 13) %>% cluster_optimal() %>% sizes()
```

```
Community sizes
 1  2  3  4  5  6  7  8  9 10 11 12 13
 4  4  4  4  4  4  4  4  4  4  4  4  4
```

```
    9) clique_ring(4, 16) %>% cluster_optimal() %>% sizes()
```

```
Community sizes
1 2 3 4 5 6 7 8
8 8 8 8 8 8 8 8
```

An alternative example of the resolution limit was published by Brandes et al (2008). Consider the graph shown on the upper part of Fig. 4.9(b). The optimal modularity score is attained when the graph is partitioned into three communities (denoted by shaded areas). However, these communities are below the resolution limit in a graph that contains two exact replicas of the original graph (see the lower part of Fig. 4.9(b)). Kumpula et al (2007) have shown that this limitation is applicable not only to the modularity function Q but also to the goal function used by the spinglass clustering method of Reichardt and Bornholdt (2006) that was introduced in Section 4.4.2.3.

A possible solution to the problem of the resolution limit is the addition of loop edges with weight γ to each of the nodes (Arenas et al, 2008). Loop edges increase the total weight of edges within a community of size n by $n\gamma$, which is enough to push the total weight above the resolution limit if γ is large enough.

In case of the ring of cliques shown on Fig. 4.9(a), we have seen that the optimal modularity score merges adjacent cliques if the clique size is 4 and there are 16 cliques. Adding loop edges with $\gamma = 0.5$ increases the number of edges within a clique and makes them valid communities on their own. In fact, much smaller γ values also suffice because the difference between the 8-community and the 16-community case is so small:

```
4.73.1) clique_ring_loops <- clique_ring(4, 14)
    2) gamma <- 0.5
    3) E(clique_ring_loops)$weight <- 1
    4) clique_ring_loops[from = 1:52, to = 1:52] <- gamma
    5) clique_ring_loops %>% cluster_optimal() %>% sizes()
```

```
Community sizes
 1  2  3  4  5  6  7  8  9 10 11 12 13 14
 4  4  4  4  4  4  4  4  4  4  4  4  4  4
```

Finally, we should also mention the sensitivity of the modularity measure to the presence of "satellite nodes", i.e. nodes that have only one neighbor. Brandes et al (2008) have shown that satellite nodes never form a community on their own in the optimal partition: they are always grouped together with their single neighbor. A simple example graph where this causes problems is a clique of n vertices, extended by n satellite nodes such that each satellite connects to one of the vertices in the original clique and vice versa. Since satellite nodes are always grouped together with their neighbors, the optimum of the modularity will have n clusters, and each vertex in the original clique will be in a different cluster:

```
4.74.1) cliques_and_satellites <- function(n) {
    2)   make_full_graph(n) + n + edges(rbind(1:n, 1:n + n))
    3) }
    4) g <- cliques_and_satellites(6)
    5) g %>% cluster_optimal()
```

```
IGRAPH clustering optimal, groups: 6, mod: 0.12
+ groups:
  $‘1‘
  [1] 1 7

  $‘2‘
  [1] 2 8

  $‘3‘
  [1] 3 9

  $‘4‘
  + ... omitted several groups/vertices
```

A possible solution to the problem of satellites is to remove the satellite vertices before the community detection process and then post-process the membership vector in order to connect the satellite vertices to the communities of their neighbors. However, care must be taken when the satellite nodes are removed, as other nodes may become satellites after the original ones are removed.

For a more thorough review of the caveats of the modularity function, refer to Fortunato and Barthélemy (2007) and Good et al (2010).

4.4.3 Label propagation algorithm

The label propagation algorithm was proposed by Raghavan et al (2007), and it is one of the few community detection algorithms that run in nearly linear time (and the only such algorithm in igraph). The basic idea of the algorithm is as follows. Let us assume that each node carries a label that identifies the community it belongs to. Initially, each node has a unique label. The algorithm consists of rounds. In each round, each node updates its label and joins the community to which the maximum number of its neighbors belong. Ties between labels are broken randomly. Labels are updated asynchronously; in other words, nodes are considered one by one in a random order in each round, and the label of vertex `i` is updated before the algorithm moves on to the next vertex `j` in the ordering so that vertex `j` will already "see" the new label of vertex `i` instead of the old one. The algorithm terminates when it holds for each node that it belongs to a community to which a maximum number of its neighbors also belong. It is easy to see that the algorithm takes $O(m)$ steps per iteration where m is the number of edges in the network, hence the total time complexity is $O(mh)$, where h is the number of iterations. Since h is small compared to m, the algorithm runs in nearly linear time.

The implementation of the label propagation algorithm in igraph is provided by the `cluster_label_prop()` function. It is an extension of the original label propagation algorithm of Raghavan et al (2007) as it allows the user to specify an initial label configuration (and even let the user leave some of the nodes unlabeled) and can also handle nodes with fixed labels. However, it should be noted that the algorithm itself is randomized and may converge to different final clusterings even with identical starting conditions as ties are broken randomly. Sometimes it also happens that two or more disconnected groups of nodes have the same label. These should be taken into account when working with the label propagation algorithm.

cluster_label_prop()

The `cluster_label_prop()` fucntion has the following parameters:

`'weights'` The name of the edge attribute storing the edge weights, or a list of weights, one item corresponding to each edge.

`'initial'` The name of the vertex attribute storing the initial labels, or a list of labels, one item corresponding to each vertex. Labels are denoted

by integers from zero; negative integers represent vertices that should be unlabeled at the start of the algorithm. The default value is `'NULL'`, which means that all the vertices have unique labels.

`'fixed'` The name of the vertex attribute containing `'TRUE'` for vertices with fixed labels and `'FALSE'` for vertices whose label may change during the course of the algorithm. Unlabeled vertices may not be fixed of course. The default value is `'NULL'`, which means that none of the labels are fixed.

As an example, let us find the communities of the Zachary karate club network using the label propagation algorithm. Since the method is randomized, we will run the algorithm five times and calculate the pairwise distances between each pair of solutions:

```
4.75.1) num_tries <- 5
    2) cls <- replicate(num_tries, cluster_label_prop(karate))
    3) outer(cls, cls, function(cl1, cl2) {
    4)   mapply(compare, cl1, cl2, method = "split.join")
    5) })
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    0   20   11   11   11
[2,]   20    0   12   12   12
[3,]   11   12    0    0    0
[4,]   11   12    0    0    0
[5,]   11   12    0    0    0
```

It is easy to see that not all the distances are zero: the algorithm indeed converged to different solutions from different initial conditions. This is not necessarily a bad thing; with the presence of overlaps between modules possible hierarchical relationships between them, it is perfectly sensible to think that there does not exist one single division of a network into disjoint communities that captures the whole diversity of the modular structure. Good et al (2010) have shown that even the modularity function exhibits similar behavour: there are exponentially many local optima of the modularity function in the space of partitions, and these local optima are in general dissimilar to each other.

The `'initial'` and `'fixed'` arguments of the label propagation algorithm can also be used to incorporate *a priori* information about the communities into the detection process. For instance, if we know that some vertices belong to different groups, we may give them fixed labels while leaving the others unlabeled and let these labels propagate to the entire network. This approach is very similar to our simple community detection algorithm that we used in Section 4.3.1. In the Zachary karate club network, we know that 'Mr Hi' and 'John A' belong to different communities, and we can communicate that to the algorithm to find a good clustering:

```
4.76.1) initial <- rep(-1, gorder(karate))
    2) fixed <- rep(FALSE, gorder(karate))
```

```
names(initial) <- names(fixed) <- V(karate)$name
initial['Mr Hi'] <- 1
initial['John A'] <- 2
fixed['Mr Hi'] <- TRUE
fixed['John A'] <- TRUE
cluster_label_prop(karate, initial = initial, fixed = fixed)
```

```
IGRAPH clustering label propagation, groups: 2, mod: 0.35
+ groups:
  $`1`
   [1] "Mr Hi"    "Actor 2"  "Actor 3"  "Actor 4"  "Actor 5"
   [6] "Actor 6"  "Actor 7"  "Actor 8"  "Actor 9"  "Actor 11"
  [11] "Actor 12" "Actor 13" "Actor 14" "Actor 17" "Actor 18"
  [16] "Actor 20" "Actor 22" "Actor 31"

  $`2`
   [1] "Actor 10" "Actor 15" "Actor 16" "Actor 19" "Actor 21"
   [6] "Actor 23" "Actor 24" "Actor 25" "Actor 26" "Actor 27"
  [11] "Actor 28" "Actor 29" "Actor 30" "Actor 32" "Actor 33"
  + ... omitted several groups/vertices
```

4.4.4 Multi-level and overlapping community detection

Earlier in Section 4.1, we have learned that the community structure of real-world networks is exceedingly complex: communities may overlap with each other and a community may also consist of further sub-communities. Section 4.2 went into great details in describing how igraph represents such multi-level and overlapping community structures, but none of the algorithms presented so far was able to work at multiple resolution levels simultaneously or to detect communities that overlap with each other only partially. This section will present a multi-level method that is implemented natively in igraph and also show how two of the most popular overlapping community detection methods can be implemented in only a few lines of code.

4.4.4.1 The Louvain method

Louvain method

The *Louvain method*, named after the Catholic University of Louvain where it was conceived by Blondel et al (2008) is a multi-level community detection algorithm that produces not only a single clustering but a list of clusterings, each at a different meaningful resolution scale. It is built on a greedy optimization scheme of the modularity function (see Section 4.3.2.2), but in some cases it is able to discern communities below the resolution limit.

The algorithm starts from the trivial community structure where each vertex belongs to a separate community, and optimizes the modularity score by iteratively moving vertices between communities in a way that greedily optimizes the modularity. Some communities may become empty at this stage. After a while, it becomes impossible to improve the modularity score by moving single vertices – further improvements would require moving more than one vertex at the same time. The algorithm then stores the current community structure, merges the vertices of the same community into a single meta-node, and continues the optimization process on the graph consisting of the meta-nodes. When a community is collapsed into a single meta-node, the edges within the community are also collapsed into a loop edge incident on the meta-node, and the weight of the loop edge will be equal to twice the number of edges in the community. Edges between a pair of comunities (both of which are collapsed into a meta-node) are also merged into a single edge whose weight is equal to the number of edges between the two communities. The optimization process terminates when all the communities have been merged and only a single meta-node remains. The stored community structures before each aggregation step then correspond to meaningful hierarchical levels of flat communities embedded in each other, resulting in a multi-level community structure. Fig. **??** illustrates the optimization and the aggregation stages and shows how a network of 16 nodes is gradually collapsed while detecting communities at three meaningful scales.

Another advantage of the Louvain method is that it runs faster than most of the other community detection methods. Analyzing a real-world network dataset with over 2 million nodes can be done in less than two minutes on a desktop computer.

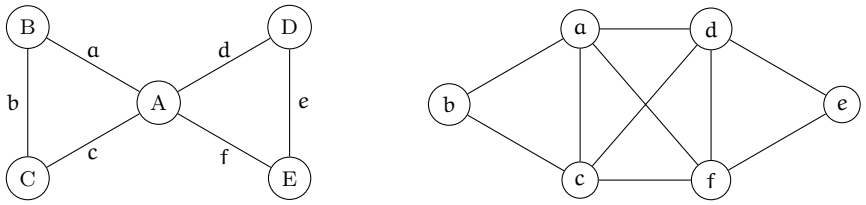
4.4.4.2 Other approaches

The hierarchical link clustering algorithm used a simple trick to obtain overlapping communities from a non-overlapping community detection algorithm: instead of grouping the vertices, the algorithm calculates pairwise similarities between *edges* and uses a hierarchical clustering algorithm to group the edges into edge communities. However, why should we limit ourselves to the admittedly simple hierarchical clustering when there are more sophisticated algorithms that we could use? This is the basic idea of the meta-algorithm of Evans and Lambiotte (2009), which can turn any non-overlapping clustering method into an overlapping one by clustering the edges instead.

The techinique of Evans and Lambiotte (2009) starts from the dynamical formulation of modularity that we have briefly discussed in Section 4.3.2.3. However, instead of considering a random walk on the nodes, they investigated two types of random walks where the walkers jump from edge to edge instead of from node to node, and have shown that optimizing the classical modularity of the so-called *line graph* of the original graph is equivalent to

line graph

finding clusters of edges in the original graph where a random edge walker is more likely to stay within. The line graph is constructed as follows. Each node in the line graph will correspond to one of the edges in the original graph, and two nodes u and v in the line graph will be connected if the corresponding edges share a vertex in the original graph. (For directed graphs, the line graph will also be directed and an edge $u \to v$ is drawn in the line graph if the target of the edge that u represents is the source of the edge that v represents in the original graph). Fig. 4.10 shows the so-called *bow-tie graph* and its corresponding line graph to illustrate the concept. *bow-tie graph*



(a) The bow-tie graph (b) Line graph of the bow-tie graph

Fig. 4.10 The bow-tie graph and its corresponding line graph. Lower-case letters associate the edges of the bow-tie graph to the vertices of its line graph.

igraph can readily calculate the line graph of an arbitrary directed or undirected graph using the `line_graph()` function, allowing us to provide a straightforward implementation of the method of Evans and Lambiotte (2009):

```
4.77.1) edge_communities <- function(graph, ...) {
    2)   to_vs <- function(es) V(graph)[inc(as.vector(es))]
    3)   graph %>%
    4)     make_line_graph() %>%
    5)     cluster_fast_greedy() %>%
    6)     groups() %>%
    7)     lapply(to_vs)
    8) }
```

We have chosen to use the greedy modularity optimization method, but the idea works with any other community detection method that tries to maximize the modularity. All the extra arguments of the `edge_communities()` function are passed to the actual community detection method. The disjoint communities of the edge graph are then converted back to overlapping vertex communities, making use of the fact that the vertex with id `i` in the line graph corresponds to the edge with id `i` in the original graph. However, note that the line graph conversion does not take into account the edge weights in the original network (if any). Readers interested in handling edge weights with this method are referred to the paper of Evans and Lambiotte (2009)

who propose a possible weighting scheme for the line graph that preserves the weight information of the original one.

Let us first find overlapping communities in the bow-tie graph of Fig. 4.10(a):

```
4.78.1) bow_tie <- make_graph(~ A-B-C-A, A-D-E-A)
    2) edge_communities(bow_tie)
```

```
Warning: 'inc' is deprecated.
Use '.inc' instead.
See help("Deprecated")
```

```
Warning: 'inc' is deprecated.
Use '.inc' instead.
See help("Deprecated")
```

```
$`1`
+ 3/5 vertices, named, from b2a2dbf:
[1] A D E

$`2`
+ 3/5 vertices, named, from b2a2dbf:
[1] A B C
```

The algorithm has correctly placed A in both communities, agreeing with our intuition.

Another possible extension of community detection methods was discussed in Traag and Bruggeman (2009). They have considered networks that contain positive and negative links as well. Assuming that positive links connect nodes that are thought to be similar in some sense and negative links connect nodes that are dissimilar, the problem of community detection becomes more complicated. In concordance with our intuition, communities in such a network should contain mostly positive links, while edges between communities should mostly be negative. Traag and Bruggeman (2009) have extended the spinglass clustering method of Reichardt and Bornholdt (2006) to be able to take into account the presence of negative links, and they have kindly contributed their code to igraph. Their method can be activated by adding the `'implementation = "neg"'` argument to the invocation of the `cluster_spinglass()` function. The importance of negatively weighted edges can be tuned by the `'gamma.minus'` keyword argument.

4.5 Interfacing with external community detection methods

There are many interesting community detection algorithms published in the scientific literature; a recent review in the Supplementary Material of

Kovács et al (2010) has cited 129 different methods for finding modules in network, and the list is by no means exhaustive. Most of these methods are not implemented in igraph directly; some of the noteworthy ones are Markov clustering algorithm of van Dongen (2008); the concept of community landscapes (Kovács et al, 2010) or the extensions of the Girvan-Newman method and the label propagation algorithm to overlapping communities (Gregory, 2007, 2010). These methods have one thing in common: the authors have made their implementations of the methods available, therefore we can make use of them from igraph by converting our graphs to a format suitable for the external implementation, asking the operating system to run the algorithm in an external process for us, and then converting the results back to one of the data structures in igraph. In this section, we will illustrate the process on the *Markov clustering* algorithm of van Dongen (2008).

Markov clustering

The implementation of the algorithm can be downloaded from `http://www.micans.org/mcl`. From now on, we will assume that MCL is installed properly and that it can be invoked from the shell of the operating system by typing its full path. According to the manual of the algoritm, the most basic example of usage is as follows:

```
4.78.1) $ mcl fname --abc -o fname-out
```

where $ represents the shell prompt, *`fname`* stands for the name of the input file and *`fname-out`* is the name of the file where the result will be written. *`--abc`* specifies that the input file is in a simple tab-separated format that looks like this:

```
4.78.1) node1   node2   weight
    2) node1   node3   weight
    3) node2   node4   weight
    4) ...
```

In this file format, each row encodes one edge of the graph. The first column contains an identifier for the source node, the second column contains an identifier for the target node, and the third column contains the weight of the edge. The result of the algorithm will also be in a tab-separated format where each line corresponds to one cluster, and the members of the cluster are separated by tabs as well.

Luckily, MCL accepts a single dash (`-`) for both the name of the input and the output file. In case of the input file, a single dash means that the input should be read from the standard input stream of the MCL process as if we typed it from the keyboard. In case of the output file, a single dash means that the output should be written to the standard output stream of the MCL process – this is usually the terminal if we launched MCL from the command line. The interface we are going to implement will connect the standard input of MCL to R so we can simply pass the input data to MCL without having to write it to a temporary file, and it will also connect the standard output of MCL back to R so we can read the result directly from MCL and not via

an intermediate file on the disk. If MCL were not so well-behaved, we could still do with temporary files if we clean them up properly after we are done.

inflation

The MCL algorithm also has an important input parameter called *inflation*, which controls the granularity of the clustering. The usual range of inflation is between 1.2 and 5.0, although the only strict limitation is that it must be larger than 1. Larger inflation values result in smaller clusters, while a small inflation value usually yields only a few large clusters. The optimal value of the inflation is usually found by experimentation or based on a comparison between the obtained clusters and a known gold standard in a dataset similar to the one being analyzed. The inflation parameter is passed on to MCL using the `-I` command line switch. We will also add `-V all` to suppress most of the messages printed by MCL as we are interested only in the final result. This makes the full command line we will use as follows:

```
4.78.1) $ mcl - --abc -V all -I inflation -o -
```

The whole complexity of this command line will nicely be hidden by the interface we are going to provide.

The bridge between igraph and MCL will be a single R function that takes the graph to be clustered and the preferred inflation value, launches MCL in the background with the appropriate command line arguments, reads the result and builds an appropriate `communities` object to return to the user. The heavy lifting will be handled by the `system()` R function that creates and handles a connection to an external process. For instance, to launch the `ls` command on a Unix-like operating system (such as Linux or Mac OS X) with the single argument `/Users`, one can simply do this:

```
2) system("ls /Users", intern = TRUE)
```

```
[1] "Guest"       "Shared"      "gaborcsardi"
```

'`intern = TRUE`' tells `system()` to connect the standard output of the `ls` subprocess back to R.

Now we know everything we need to drive MCL from igraph:

```
4.80.1) cluster_mcl <- function(graph, inflation = 1.2, verbose = FALSE) {
   2)   cmd <- paste("mcl", "-", "--abc")
   3)   if (verbose) cmd <- paste(cmd, "-V", "all")
   4)   cmd <- paste(cmd, "-I", inflation, "-o", "-")
   5)
   6)   input <- graph %>%
   7)     as_edgelist(names = FALSE)
   8)   if (is_weighted(graph)) input <- cbind(input, E(graph)$weight)
   9)   input <- input %>%
  10)     apply(1, paste, collapse = "\t")
  11)
  12)   output <- system(cmd, intern = TRUE, input = input) %>%
  13)     strsplit("\t") %>%
```

```
14)     lapply(as.numeric)
15)
16)   membership <- numeric(gorder(graph))
17)   for (line in seq_along(output)) {
18)     membership[output[[line]]] <- line
19)   }
20)   make_clusters(graph, membership = membership, algorithm = "MCL")
21) }
```

Make sure you adjust the path, so that you can call MCL. The `'input'` argument of `system()` takes a character vector and feeds it line by line to MCL. MCL then returns the communities, one per line, and we parse these into a proper `communities` object. The usage of the method is straightforward:

```
4.81.1) cluster_mcl(karate, inflation = 1.8)
```

```
IGRAPH clustering MCL, groups: 2, mod: 0.37
+ groups:
  $`1`
   [1]  9 10 15 16 19 21 23 24 25 26 27 28 29 30 31 32 33 34

  $`2`
   [1]  1  2  3  4  5  6  7  8 11 12 13 14 17 18 20 22
```

4.6 Exercises

▶ EXERCISE 4.1. Show that the expected value of the modularity of a given clustering is zero if the underlying graph is rewired randomly while keeping the degree distribution. (Hint: use the `rewire()` function to rewire a graph in-place or the `sample_degseq()` function to generate a new graph with a given degree sequence).

▶ EXERCISE 4.2. Find the graph with n vertices that has the partition with the largest possible modularity value.

▶ EXERCISE 4.3. Find the graph with $2n$ vertices that has the partition with the *smallest* possible modularity value. (Hint: it is $2n$ and not n for a reason).

▶ EXERCISE 4.4. Find a graph for which the leading eigenvector method may stop at different clusterings with unequal cluster counts, depending on the initial estimate of the eigenvector being used.

▶ EXERCISE 4.5. Investigate the modularity of Erdős-Rényi random networks and validate the formula of Guimerà et al (2004) for the expected maximal modularity of such networks (see Eq. 4.22) numerically.

▶ EXERCISE 4.6. Brandes et al (2008) have shown that satellite nodes are always grouped together with their neighbors, which may result in un-intuitive global maxima for the modularity score. Implement a method that removes the satellite nodes from a graph, runs an arbitrary community detection on the trimmed graph and post-processes the resulting membership vector to provide a clustering for the original graph. (Hint: the `coreness()` function helps to find satellite nodes).

▶ EXERCISE 4.7. Evans and Lambiotte (2009) have proposed a weighting scheme for line graphs that makes their meta-algorithm more accurate for graphs with high-degree nodes. Extend our implementation in Section 4.4.4.2 with their weighting scheme and compare the results obtained on the Zachary karate club network with and without weighting. Which one of the solutions seems closer to our intuition regarding the placement of 'Mr Hi'?

Chapter 5
Random graphs

5.1 Introduction

TODO: Write this section

5.2 Static random graphs

5.2.1 The Erdős–Rényi model

The Erdős–Rényi random graph model (Erdős and Rényi, 1959) is probably one of the earliest random graph models in the literature. It comes in two variants: the $G(n, p)$ and the $G(n, m)$ model, both of which start with an empty directed or undirected graph consisting of n vertices and no edges. The $G(n, p)$ model then considers all the possible vertex pairs (ordered pairs in the case of a directed graph, unordered pairs in the case of an undirected graph) and connects each pair with probability p. The $G(n, m)$ model chooses m out of all the possible vertex pairs in the graph and connects them with an edge. The two versions are related but not entirely equivalent: in the $G(n, m)$ model, the number of edges in the generated graph will always be m, while in the $G(n, p)$ model, only the *expected* number of edges m will be equal to $n(n-1)p$ in the case of a directed graph and $\frac{n(n-1)p}{2}$ in the case of an undirected graph[1], but the *actual* number of edges will fluctuate around the expected value, following a binomial distribution. In some sense, one can think about p as a tuning parameter; barring the cases when $p = 0$ or $p = 1$, the $G(n, p)$ model may generate *any* number of edges between n vertices, but small edge counts are more likely to occur when p is small and large edge

[1] These numbers assume that loop edges are disallowed. When we allow loops, the expected number of edges will be $n^2 p$ and $\frac{n(n+1)p}{2}$, respectively.

counts are more likely to occur when p is large. In particular, when $p = 1/2$, *every* graph with n vertices is equally likely to be generated by the model.

sample_gnm() *sample_gnp()*

The $G(n, m)$ and $G(n, p)$ Erdős–Rényi models are implemented in igraph by the `sample_gnm()` and `sample_gnp()` functions:

```
5.1.1) g <- sample_gnp(n = 100, p = 0.2)
   2) summary(g)
```

```
IGRAPH 4e08825 U--- 100 960 -- Erdos renyi (gnp) graph
+ attr: name (g/c), type (g/c), loops (g/l), p (g/n)
```

The above call generates an undirected Erdős–Rényi graph with no loop edges. We can easily see that the number of edges is not exactly equal to its expected value (which is $\frac{100 \times 99}{2} \times 0.2 = 990$), but it is fairly close to it. Switching to the $G(n, m)$ model allows us to set the number of edges exactly, while the generated graph will still be completely random:

```
5.2.1) g <- sample_gnm(n = 100, m = 990)
   2) summary(g)
```

```
IGRAPH bfcde57 U--- 100 990 -- Erdos renyi (gnm) graph
+ attr: name (g/c), type (g/c), loops (g/l), m (g/n)
```

To see that the generated graph is really random, we can try generating another with the same parameter settings and check whether the two graphs are isomorphic or not using the `isomorphic()` function. Not too surprisingly, the two graphs will be different:

```
5.3.1) g2 <- sample_gnm(n = 100, m = 990)
   2) summary(g2)
```

```
IGRAPH b2dca3c U--- 100 990 -- Erdos renyi (gnm) graph
+ attr: name (g/c), type (g/c), loops (g/l), m (g/n)
```

```
   5) g %>% isomorphic(g2)
```

```
[1] FALSE
```

Both `sample_gnp()` and `sample_gnm()` allow you to specify whether you would like a directed graph (using the `'directed'` argument) and whether you allow loop edges (using the `'loops'` argument). The default values are `'FALSE'` for both of them. To generate a directed Erdős–Rényi graph with loop edges:

```
5.5.1) g <- sample_gnp(n = 100, p = 0.2, directed = TRUE, loops = TRUE)
   2) summary(g)
```

```
IGRAPH 23ccee9 D--- 100 2051 -- Erdos renyi (gnp) graph
+ attr: name (g/c), type (g/c), loops (g/l), p (g/n)
```

The Erdős–Rényi random graph model is a very "democratic" one: all the vertices are born equal and treated equal; in other words, it is equally likely for any of the vertices to gain an edge in any of the steps, irrespectively of the number of edges they already have. Assuming that there are no loop edges and we are using the $G(n, p)$ model (which is usually easier to treat analytically), we can see that each vertex has $n - 1$ chances to form new connections, and each chance succeeds with probability p. It immediately follows that the degree distribution of Erdős–Rényi random graphs follows a binomial distribution $B(n-1, p)$, and the average vertex degree is $(n-1)p$. Assuming that n is very large (which we usually do when working with random graph models), we can approximate the binomial with a Poisson distribution of mean $\lambda = np$ since n versus $n - 1$ does not make a difference when $n \to \infty$.

Note that knowing the number of vertices n and the average degree allows us to derive the connection probability p easily. When one works with large and sparse graphs, it is usually easier to specify the average degree $\langle k \rangle$ instead of the connection probability p as the latter is usually very small.

Let us now inspect the degree distribution of a large Erdős–Rényi graph with 10^5 vertices where we set the average degree of each vertex to 3 – note how the appropriate probability value is obtained from the prescribed average degree:

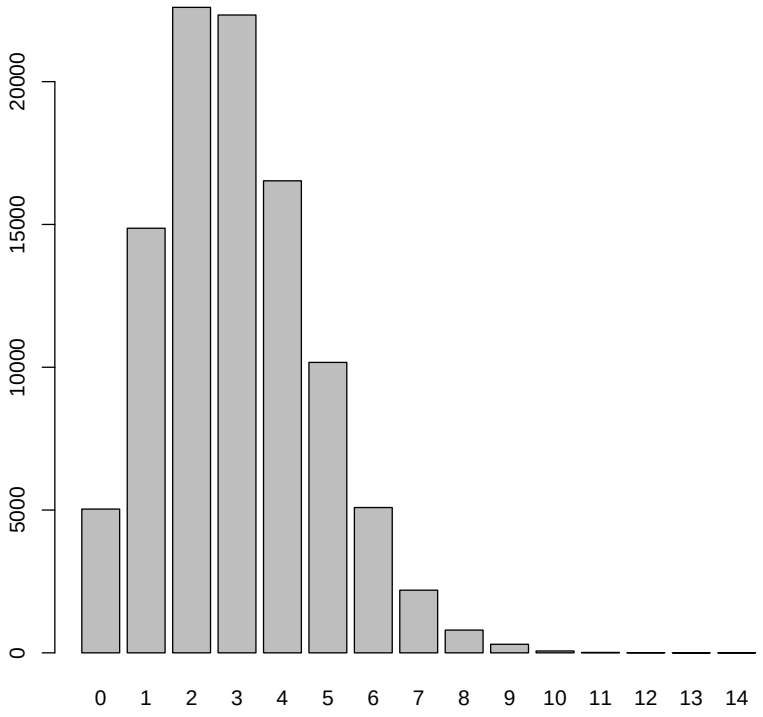
5.6.1)
```
n <- 100000
avg_degree <- 3
p <- avg_degree / n
g <- sample_gnp(n = n, p = p)
dd <- degree(g)
degree_distribution(g)
```

```
 [1] 0.05034 0.14867 0.22601 0.22333 0.16528 0.10169 0.05088
 [8] 0.02194 0.00797 0.00300 0.00066 0.00014 0.00006 0.00001
[15] 0.00002
```

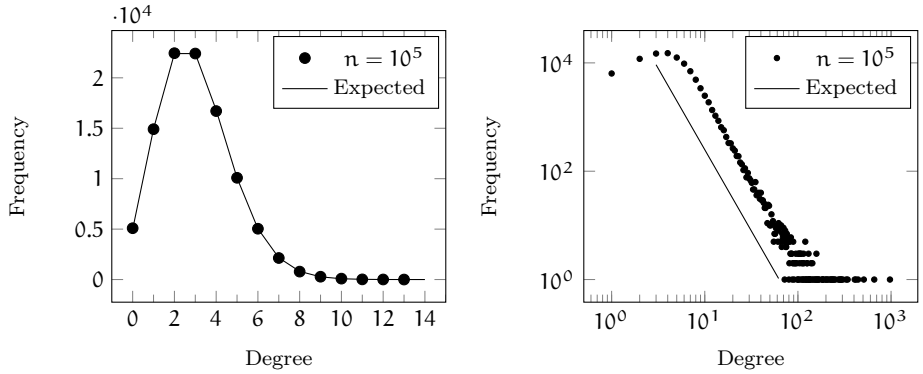
10)
```
max(dd)
```

```
[1] 14
```

12)
```
dd %>% table() %>% barplot()
```



As it can be seen on the above histogram (and also on Figure 5.1(a)), indeed the mean degree is very close to 3, and the distribution looks like a Poisson distribution. It can also be seen that the vertex with the maximum degree has only 14 incident edges. In other words, even in a large graph like this one, it is very unlikely that we see vertices with significantly more or significantly less edges than the average degree. If society were an Erdős–Rényi random graph, everyone would have approximately the same number of friends, and it would be very hard to find one who has thousands of friends, or who has not a single one of them. This is clearly a limitation of the Erdős–Rényi random graph model as we do not expect real networks to behave the same way. Despite this shortcoming, Erdős–Rényi random graphs still serve as an important "baseline" with which we can compare real networks and quantify how different they are from a truly random network. Later on, we will learn about other random graph models that are closer to reality, at least in their degree distribution.



(a) Erdős–Rényi network (b) Scale-free network with $\gamma = 3$

Fig. 5.1 Degree distribution of an Erdős–Rényi random network (left) and a scale-free network (right). Circles and crosses denote simulated results, solid lines denote the analytical ones. The expected degree is 3 in both cases. Note that the plot on the right has logarithmic axes.

Before we move on to more sophisticated random graph models, let us take a closer look at the degree histogram again. We can see that many of the vertices have degree zero, which means that these vertices are completely isolated in the network. It is reasonable to ask at this stage how the rest of the network looks like: do the vertices form disconnected small islands or are they organised into one giant component? To answer this question, we must first define what a *giant component* is.

giant component

Definition 5.1 (Giant component.). A giant component of a random graph with n vertices is a connected component containing n_0 vertices such that $n_0/n \to \rho > 0$ as $n \to \infty$.

(See Section 2.6 for the definition of a connected component).

Loosely speaking, a giant component is a connected component whose relative size (compared to the entire network) does not diminish as the size of the network grows to infinity. As the network grows, the giant component grows with it such that its relative size tends to a non-zero constant.

As an extreme example, consider an Erdős–Rényi network with $p = 0$, i.e. with no edges. We can easily see that the largest connected component of this graph is an isolated vertex, therefore its size is 1. Since $1/n \to 0$, there is clearly no giant component if $p = 0$. At the other extremity $p = 1$, all the edges are present in the network, thus the largest component always encapsulates all the vertices, and its relative size is $n/n = 1$, showing the presence of a giant component. Between the two extremes, there must be a tipping point where the giant component emerges. Physicists call this transition a *percolation transition*, and the point where it happens the *percolation threshold*. We will try to find the percolation threshold of Erdős–Rényi graphs with

percolation transition

percolation threshold

simulations.

We have seen before that specifying p is equivalent to specifying $\langle k \rangle$, but it is more convenient to work with the latter. Therefore, we are going to generate random graphs with $n = 10^5$ vertices and different $\langle k \rangle$ values to get a rough idea about how the fraction of the largest component size behaves as the function of $\langle k \rangle$. We will use the `components()()` function (which we have used earlier in Chapter 2 as well) to find the connected components of the graph, get the sizes of the components. The average degree of the graph will start at zero and will be increased by 0.2 in every iteration until we get a graph where at least 50% of the vertices are in the same connected component.

```
5.9.1) find_percolation_threshold <- function(n, step = 0.2) {
   2)   deg <- frac <- numeric()
   3)   avg_deg <- 0
   4)   while (TRUE) {
   5)     p <- avg_deg / n
   6)     g <- sample_gnp(n = n, p = p)
   7)     comps <- components(g)
   8)     frac_largest <- max(comps$csize) / n
   9)     cat("<k> = ", avg_deg, "\tfraction = ", frac_largest, "\n")
  10)     deg <- c(deg, avg_deg)
  11)     avg_deg <- avg_deg + step
  12)     if (frac_largest > 0.5) break
  13)   }
  14) }
```

```
5.10.1) find_percolation_threshold(1000)
```

```
<k> =  0         fraction =  0.001
<k> =  0.2         fraction =  0.006
<k> =  0.4         fraction =  0.01
<k> =  0.6         fraction =  0.015
<k> =  0.8         fraction =  0.048
<k> =  1         fraction =  0.102
<k> =  1.2         fraction =  0.2
<k> =  1.4         fraction =  0.55
```

Apparently, even the largest component is quite small when $\langle k \rangle < 1$, but something happens when $\langle k \rangle$ reaches 1 as the fraction of the largest component suddenly starts to increase. To check whether $\langle k \rangle = 1$ could indeed be the percolation threshold, we repeat the same analysis with larger graphs:

```
5.11.1) find_percolation_threshold(20000)
```

```
<k> =  0         fraction =  5e-05
<k> =  0.2         fraction =  0.00045
<k> =  0.4         fraction =  0.00075
```

```
<k> =  0.6         fraction =  0.00215
<k> =  0.8         fraction =  0.0051
<k> =  1         fraction =  0.02895
<k> =  1.2         fraction =  0.343
<k> =  1.4         fraction =  0.5129
```

```
find_percolation_threshold(100000)
```

```
<k> =  0         fraction =  1e-05
<k> =  0.2         fraction =  9e-05
<k> =  0.4         fraction =  0.00017
<k> =  0.6         fraction =  0.00042
<k> =  0.8         fraction =  0.00082
<k> =  1         fraction =  0.03046
<k> =  1.2         fraction =  0.32371
<k> =  1.4         fraction =  0.50628
```

We can clearly see that for $\langle k \rangle < 1$, the fraction of the largest component diminishes as n grows. For $\langle k \rangle > 1$, the fractions are definitely non-zero and seem to converge. The behaviour or $\langle k \rangle = 1$ is somewhat ambiguous, leading us to conjecture that the percolation threshold is there at $\langle k \rangle = 1$.

Of course these small experiments cannot be considered as rigorous scientific proofs, but the truth is that indeed the percolation threshold is exactly at $\langle k \rangle = 1$ (which is equivalent to $p = 1/n$). Erdős and Rényi have shown that when $np < 1$, the graph will almost surely[2] have no connected components of size larger than $O(\log n)$. When $np = 1$, the graph will almost surely have a largest component whose size is $O(n^{2/3})$. When $np > 1$, the graph will almost surely contain a component with size $O(n)$, and no other component will contain more than $O(\log n)$ vertices. Figure 5.2 shows an Erdős–Rényi graph with 100 vertices exactly at its percolation threshold; one can clearly see a few possible candidates for the giant component that is about to emerge, and also the set of small disconnected islands that surround them.

Note that when $p \approx 1/n$, the graph will have approximately one edge for each vertex. It is also known that one requires at least $n - 1$ edges to connect n vertices, therefore the number of edges required for the emergence of the giant component is surprisingly small.

[2] "Almost surely" in this context means that as $n \to \infty$, the probability of the described event tends to 1.

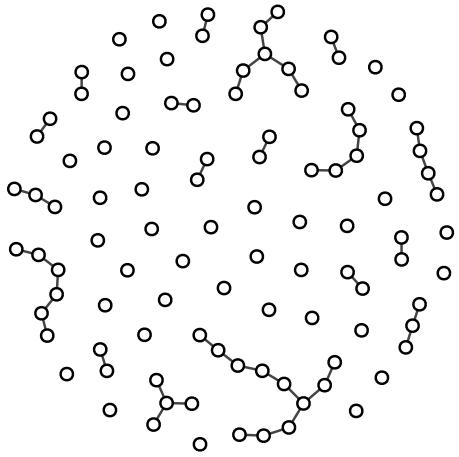


Fig. 5.2 An Erdős–Rényi random graph with $n = 100$ vertices at its percolation threshold ($p = 0.01, \langle k \rangle = 1$).

Chapter 6
Epidemics on networks

6.1 Introduction

TODO: Write this section (Kermack and McKendrick, 1927) (Anderson and May, 1992) (Keeling and Eames, 2005) (Daley and Gani, 2001)

6.2 Branching processes

Branching processes provide a fairly simple model for the spreading of infections in a population. Suppose that a randomly selected person in a population contracts a contagious disease. He or she then meets k other people while being contagious; these are the ones who can catch the disease via direct transmission from the source. We call these people the first wave of the epidemic. Each of the k people in the first wave then meet k different people again, forming the second wave. People in the second wave can potentially catch the disease from those in the first wave who are already infected. Subsequent waves are then formed in a similar manner; each person in the ith wave may spread the infection further to k people in the $(i+1)$th wave.

contact network

One can define the *contact network* for this simple branching process as follows. Each node in the network corresponds to one person, and a directed edge points from node u to node v if u contacted w in one of the waves. It is easy to see that the contact network will be an infinite, directed tree where the root of the tree is the person who contracted a contagious disease in the "zeroth" wave, and the people in the ith wave will be reachable from the root via exactly i hops. Fig. 6.1 shows the first three layers (i.e. the first two waves and the source of the infection) of the contact network of a branching process with $k = 3$.

Note that we have used at least three unrealistic assumptions here: first, that the population is infinite (as there is no upper limit on the number of

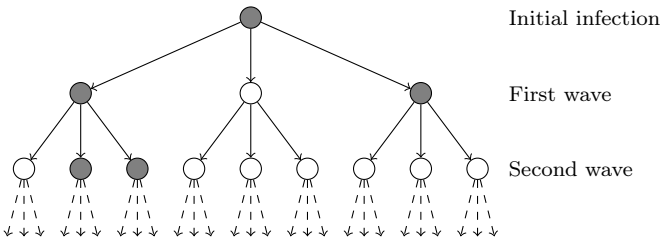


Fig. 6.1 The contact network of a branching process with $k = 3$. Gray shading corresponds to infected nodes. Thick lines denote connections through which the infection spread.

waves); second, that everyone in the ith wave spreads the disease to different people (there is no one in the $(i+1)$th wave who was in contact with more than one person from the ith wave); and third, that everyone has k contacts. These assumptions do not hold in real social or technological networks, but the model will nevertheless exhibit a peculiar behaviour that is prevalent in almost all of the other, more complicated epidemic models.

Let us now assume that a contagious person may infect a yet healthy person during a contact with probability p; in other words, people in the first wave get infected with probability p, and people in subsequent waves get infected with probability p *if* their parent in the first wave was infected. Will the virus stay alive forever in the population or will it become extinct eventually? How does the outcome depend on the value of p? Before turning to the (admittedly simple) analytical solution, let us run a few simulations in igraph.

Although igraph is capable of handling very large graphs, infinite contact networks are still beyond its limits, therefore we will restrict ourselves to finite contact networks with a given number of waves n and a constant branching number k at each layer. We will investigate the fraction of infected people in the nth wave, which should be a good indicator of the virulence of the infection as almost everyone would be infected in the nth wave in case of a highly contagious virus. Simulations will be run for different values of p, and the virulence will be evaluated 100 times for each studied value of p to suppress the effect of random fluctuations in the spreading process.

Let us first create a function that generates the contact network of the branching process with n waves and a constant branching degree k:

```
6.1.1) def create_tree(n, k):
   2)     num_vertices = (k**n - 1) / (k-1)
   3)     return Graph.Tree(num_vertices, k, mode="out")

6.2.1) print create_tree(4, 2)
```

The function simply makes use of the fact that the contact network has $(k^n - 1)/(k - 1)$ vertices and that the *Tree()* method of the *Graph* class

Tree()

creates directed trees with equal branching degrees.

Next, we will create an auxiliary function that we will use many times later on in this chapter. *sample()* will be a function that takes a list or iterable of items and selects each one of them with probability p:

```
6.3.1) import random
   2)
   3) def sample(items, p):
   4)     return [item for item in items if random.random() < p]
```

The next step is a function that simulates a single instance of the infection process. The graph will be generated outside the function by calling *create_tree()*, and we will use the "*infected*" vertex attribute to keep track of the states of the vertices. Note that the vertex IDs in an igraph tree increase as we progress down the tree, which means that it is enough to do a single pass on the vertices to propagate the infection. This will not be true in general but is very useful here.

```
6.4.1) def simulate_infection(tree, p):
   2)     ## Set all the vertices to healthy
   3)     tree.vs["infected"] = False
   4)
   5)     ## Infect the root vertex
   6)     tree.vs[0]["infected"] = True
   7)
   8)     ## Infected nodes propagate the infection downwards
   9)     for vertex in tree.vs:
  10)         if not vertex["infected"]:
  11)             continue
  12)
  13)         ## This is an infected node. Sample its successors
  14)         ## uniformly with probability p and infect them
  15)         neis = vertex.successors()
  16)         for vertex in sample(neis, p):
  17)             vertex["infected"] = True
```

The final ingredient is our main function that calls *simulate_infection()* many times on the same graph and counts the number of infected nodes in the last wave. The last wave is recognized by extracting the vertices that have zero outdegree; this is done using the *select()* method of the *VertexSeq* class. We also use the *RunningMean()* class of igraph to calculate the mean; this is a simple class that keeps track of the running mean and standard deviation of a sequence of numbers.

select()
VertexSeq
RunningMean()

```
6.5.1) def test_branching(n, k, p, num_trials=100):
   2)     ## Construct the tree
   3)     tree = create_tree(n, k)
```

```
4)
5)      ## Find the leaves (i.e. the last wave)
6)      leaves = tree.vs.select(_outdegree=0)
7)
8)      ## Run the trials and take the mean of the result
9)      result = RunningMean()
10)     for trial in xrange(num_trials):
11)         simulate_infection(tree, p)
12)         num_infected = len(leaves.select(infected=True))
13)         result.add(num_infected / float(len(leaves)))
14)
15)     return result.mean
```

Let us first run a quick test with 12 waves and a branching degree of 2:

```
6.6.1) for i in xrange(11):
2)      p = i / 10.0
3)      print "%.1f   %.3f" % (p, test_branching(12, 2, p))
```

We may see that there seems to be a phase transition around $p = 1/2$. When $p < 1/2$, almost none of the vertices get infected in the eighth wave: the infection dies out quickly. When $p > 1/2$, almost all the vertices are infected in the eighth wave and it seems likely that the infection will keep on spreading. $p = 1/2$ is the so-called *epidemic threshold* which separates cases of a relatively mild virus from a virulent one that is capable of infecting the vast majority of the population. But is it always $1/2$? Let us repeat the experiment with 8 waves and a branching degree of 3 (to keep the total number of vertices roughly the same):

epidemic threshold

```
6.7.1) for i in xrange(11):
2)      p = i / 10.0
3)      print "%.1f   %.3f" % (p, test_branching(8, 3, p))
```

This time the threshold seems to be somewhere between 0.3 and 0.4; in fact, it can be shown that it is exactly $1/3$. Further simulations with $k = 4$, $k = 5$ and so on reveal similar patterns; the plots on Fig. 6.2 show simulation results for $k = 2, 3, \ldots, 6$. We may begin to suspect a pattern here: it can be shown that the epidemic threshold in this simple model is $p = 1/k$. The proof is straightforward. An infected individual in a wave infects each of its successors in the next wave with probability p, thus the expected number of *additional* infections an infected individual creates is pk. This number is called the *basic reproductive rate* of the infection. Given $u^{(i)}$ infected individuals in the ith wave, the expected number of infections in the $(i+1)$th wave will be $u^{(i)}pk$. If $u^{(i)}pk$ is less than $u^{(i)}$, the prevalence of the virus will dwindle with every new wave and eventually reaches zero in the infinite limit, leading to the condition of an epidemic outbreak: $pk > 1$, i.e. $p > 1/k$.

basic reproductive rate

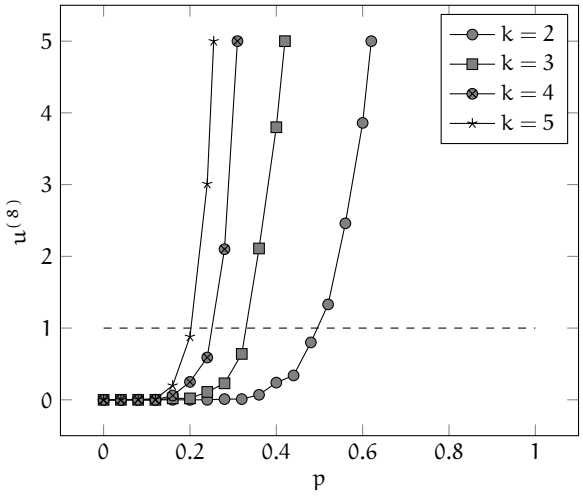


Fig. 6.2 Prevalence of the virus in the last wave of the branching process for $k = 2, 3, \ldots, 6$ and various values of p. The dashed line corresponds to $u(8) = 1$ (i.e. at least one infection on average in the eighth wave). Note that the epidemic threshold where $u(8) > 1$ is approximately $1/k$ for all k.

basic reproductive rate

The presence of an epidemic threshold has far-reaching implications. Assume that we are somehow able to measure the *basic reproductive rate* pk of a virus that is currently spreading in the population – this is simply the number of additional infections an infected individual may create. We know that a sufficient and necessary condition for an epidemic outbreak is $pk > 1$; in other words, every infected individual must produce at least one new infection. If the reproductive rate is only slightly larger than one, it may be enough to reduce p or k by only a small amount to prevent a potential outbreak. The question is then as follows: is the epidemic threshold only an artefact of the simplifications we have made in this simple branching model, or do similar thresholds exist for real-world epidemics? As we will see, the answer is yes, and most of the preventive measures in case of an epidemic are aimed at either reducing p or k: closing down schools and nurseries and advising people to avoid public places decreases k (the number of contacts between people), while vaccinations try to reduce p.

6.3 Compartmental models on homogeneous populations

We are already aware of the limitations of the branching processes that we studied in Section 6.2. First of all, the population is considered infinite; sec-

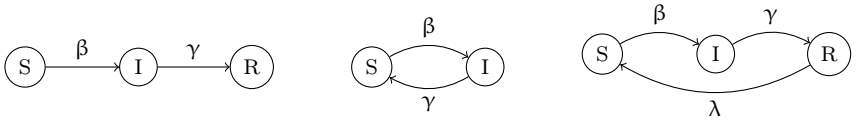
ond, infected individuals stay infected forever but they do not spread the infection after the wave they participate in has ended; Third, every infected individual is assumed to have the same number of contacts. The *compartmental models* (Kermack and McKendrick, 1927; **?**; Anderson and May, 1992) that we will briefly introduce in this section lift the second limitation.

compartmental models

The name of these models stem from the fact that they divide the population into discrete compartments; individuals in the same compartment are in the same state and behave identically in terms of spreading the infection. The population is assumed to be finite but very large, and it is still considered practically homogeneous. In the simplest case, the pool of individuals does not change over time, i.e. no new individuals are born into the pool and none of the individuals leave the population. The two variants of compartmental models we introduce here differ only in the number of compartments and the equations that define how the sizes of the compartments change over time.

6.3.1 The susceptible-infected-recovered (SIR) model

This model, first described by Kermack and McKendrick (1927), assumes three states (compartments) in which an individual can reside. The S compartment contains the susceptible individuals: those that are not infected yet but may contract the infection if they meet someone contagious. Infected individuals are in the I compartment, while the R compartment contains the individuals who have already contracted the disease and successfully recovered from it. (In the pessimistic description of the model, the latter group consists of individuals who have contracted the disease and have been removed from the population). Individuals are initially in the S pool, and they are moved to the I pool when they get in contact with a contagious person. People in the I pool eventually recover and are moved to the R pool. Recovered individuals are assumed to have developed a lifelong immunity to the disease; in other words, they stay in the R pool forever. Fig. 6.3(a) shows the state chart of the model with the possible transitions.



(a) SIR model (b) SIS model (c) SIRS model

Fig. 6.3 State charts of the SIR, SIS and SIRS models. The rate parameters are as follows: β = rate of infection, γ = rate of recovery, λ = rate of deimmunization.

The model has two parameters: β, the infection rate, and γ, the recovery rate. The larger the infection rate, the faster people move from the S pool to the I pool, and similarly, the larger the recovery rate, the faster people move from the I pool to the R pool. The flow is strictly unidirectional: there is no way to get susceptible again after being infected, and there is no way to get infected or susceptible after having recovered from an infection. This implies that the model is generally suitable to describe illnesses where the individual gains lifetime immunity after an infection (e.g., mumps or chickenpox). It can also be used to model terminal contagious illnesses such as AIDS.

Let us denote the number of individuals in each compartment at time step t by $S^{(t)}$, $I^{(t)}$ and $R^{(t)}$, and let us denote the *fraction* of such individuals by $s(t)$, $i(t)$ and $r(t)$. In a constant population consisting of N individuals, it easily follows that $S^{(t)} + I^{(t)} + R^{(t)} = N$ and $s(t) + i(t) = r(t) = 1$ for any t. The homogeneity assumption means that every individual has an equal chance to meet every other individual in the population, hence the number of contacts involving a susceptible and an infected person in a single time step will be proportional to the product of the sizes of the S and I pools. This leads to the following set of difference equations[1] that specify our model formally:

$$\frac{S^{(t+\Delta t)} - S^{(t)}}{\Delta t} = -\beta S^{(t)} I^{(t)} \tag{6.1}$$

$$\frac{I^{(t+\Delta t)} - I^{(t)}}{\Delta t} = \beta S^{(t)} I^{(t)} - \gamma I^{(t)} \tag{6.2}$$

$$\frac{R^{(t+\Delta t)} - R^{(t)}}{\Delta t} = \gamma I^{(t)} \tag{6.3}$$

In other words, the number of contacts involving an individual from the S-pool and another one from the I-pool is proportional to $S^{(t)}I^{(t)}$, and some of these contacts result in the actual transmission of the disease; the factor $\beta > 0$ accounts for both the probability of transmission and the frequency of S-I contacts. In each time step, the number of susceptible people $S^{(t)}$ decreases by $\beta S^{(t)} I^{(t)}$, and since these people are moved to the I-pool, the number of infected people $I^{(t)}$ increases with the same amount. The other part of the dynamics of the model is a steady flow from the I pool to the R pool, modeled by the $-\gamma I^{(t)}$ term in the difference equation for $I^{(t)}$ and the $\gamma I^{(t)}$ term in the difference equation for $R^{(t)}$ (where $\gamma > 0$).

We do not really need igraph to simulate the SIR model in a homogeneous population as there is no underlying network structure. The entire simulation can be performed in pure Python with the following code snippet which prints

[1] Many textbooks assume a continuous time scale instead of discrete time steps and thus formulate the SIR model with differential equations. The conclusions are essentially the same in both cases; here we decided to use the discrete time formulation because the simulations we are going to perform later require discretized time instances anyway.

the number of people in the S, I and R compartments (in this order) after each time step:

6.8.1)
```
def simulate_sir(beta=0.001, gamma=0.7, n=500, dt=0.1, steps=1000):
    s, i, r = n-1.0, 1.0, 0.0
    while steps > 0:
        print("%6.2f %8.2f %8.2f" % (s, i, r))
        s_new = s - beta * s * i * dt
        i_new = i + (beta * s - gamma) * i * dt
        r_new = r + gamma * i * dt
        s, i, r = s_new, i_new, r_new
        steps -= 1
```

6.9.1)
```
simulate_sir(beta=0.02, gamma=0.7, steps=10)
```

A word of warning to the above code snippet: the simulation does not ensure that *s_new* is positive in every time step; in fact, *s_new* can easily get negative if the expected number of new infections in the next Δt time units (which is $\beta S^{(t)} I^{(t)} \Delta t$) is larger than $S^{(t)}$. This usually happens after a few time steps for large β, but can easily be resolved by decreasing the time step *dt*.

There are three important properties of this model. First, the dynamics is mainly driven by $I^{(t)}$. This is easy to prove as all the terms in the difference equations include $I^{(t)}$, meaning that the model is in a steady state if $I^{(t)} = 0$, i.e. if everyone is either susceptible or recovered. Second, these are the only steady states if $\beta > 0$ and $\gamma > 0$ – this also follows trivially from the difference equations. The third property is less obvious but very important: we will show that there is an *epidemic threshold* in the SIR model as well.

epidemic threshold

Let us inspect how $I^{(t)}$ changes over time; in particular, let us focus on $I^{(t+1)} - I^{(t)}$ where $\Delta t = 1$:

$$I^{(t+1)} - I^{(t)} = \beta S^{(t)} - \gamma \tag{6.4}$$

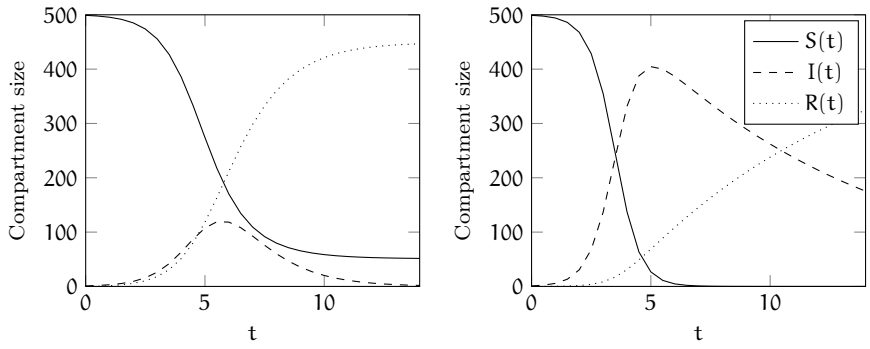
The number of infected people will thus increase if $\beta S^{(t)} > \gamma$, stay constant if $\beta S^{(t)} = \gamma$ and decrease if $\beta S^{(t)} < \gamma$. Assuming that β and γ are inherent and constant parameters depending on the virus that is infecting the population, $S(0) < \gamma/\beta$ implies that the infection can never turn into an epidemic as its prevalence starts to decrease immediately. This is clearly similar to the threshold-like behaviour that we have already seen with the branching processes. However, there is an important difference. The growth of $I^{(t)}$ implies the depletion of the S-pool, and since the growth rate of $I^{(t)}$ depends on $S^{(t)}$ with a positive multiplicative factor, the *rate* of growth decreases over time: even if the epidemic managed to gain momentum, it subsides eventually as the S-pool becomes empty.

To confirm the validity of our approach when determining the epidemic threshold, let us run two simulations for a given β and γ and a single infected individual at time step zero. One of the simulations will use a population size

N that makes the initial number of susceptible individuals $S(0) = N - 1$ just above the epidemic threshold, while the other one uses a population size slightly below the epidemic threshold. We will use $\beta = 0.2$ and $\gamma = 0.7$, leading to an epidemic threshold of $S(0) = N - 1 = 3.5$, thus the two simulations will be run with $N = 5$ and $N = 4$. As expected, the virus starts spreading for $N = 5$ but dwindles for $N = 4$:

```
6.10.1) simulate_sir(beta=0.2, gamma=0.7, n=5, steps=10)
    2) simulate_sir(beta=0.2, gamma=0.7, n=4, steps=10)
```

Another factor that influences the growth of $I^{(t)}$ is the recovery rate γ. One can also think about the recovery rate as the inverse of the expected duration of the infection as an infected individual experiences one recovery in $1/\gamma$ units of time. A low recovery rate means that infected individuals stay contagious longer, thus have more chance to cause additional infections. From the point of view of the virus itself, a low recovery rate is advantageous. However, note that the R in the SIR model can also stand for removed instead of recovered, and it does not change the model equations at all, leading to a somewhat paradoxical conclusion: aggressive terminal diseases that cause death in a short amount of time (i.e. that have a high removal rate γ) spread slower and cause smaller epidemics because infected individuals do not stay in the population for a long enough time to cause more infections. Fig. 6.4 illustrates this by showing two typical courses of a SIR-type epidemic over time, for two qualitatively different values of γ. Note that in case of $\gamma = 0.8$ on Fig. 6.4(a), some individuals are still susceptible when the infection vanishes.



(a) $\beta = 0.004, \gamma = 0.8, N = 500$ (b) $\beta = 0.004, \gamma = 0.1, N = 500$

Fig. 6.4 Typical courses of an epidemic in the SIR model with different parameters.

Similarly to the case of branching processes, we can also estimate the *basic reproductive rate* of the infection by calculating $I^{(t+1)}/I^{(t)} = 1 + \beta S^{(t)} - \gamma$. Note that it holds also in the case of the SIR model that a basic reproductive rate greater than one implies an epidemic outbreak: $1 + \beta S^{(t)} - \gamma > 1$

basic reproductive rate

if and only if $\beta S^{(t)} > \gamma$. This has an important consequence: if we want to prevent an epidemic outbreak, we may either try to decrease β (by forcing people to stay at home, thus reducing the contact rate), increase γ (by timely and efficient medical treatment of infected individuals), or decrease $S^{(t)}$. The latter is the motivation of vaccination which moves people directly from the S compartment to the R compartment without having to go through the infected state. Interestingly enough, it is not necessary to immunize everyone: vaccinating $\lceil (1-\gamma/\beta)S^{(t)} \rceil$ susceptible individuals already prevents the outbreak.

6.3.2 The susceptible-infected-susceptible (SIS) model

The susceptible-infected-susceptible (SIS) model (Bailey, 1975) is very similar to the SIR model but does not assume a lifelong immunity after recovery; individuals who have recovered from the disease become susceptible again (see Fig. 6.3(b). The equations that govern the dynamics of the model are as follows:

$$\frac{S^{(t+\Delta t)} - S^{(t)}}{\Delta t} = -\beta S^{(t)} I^{(t)} + \gamma I^{(t)} \tag{6.5}$$

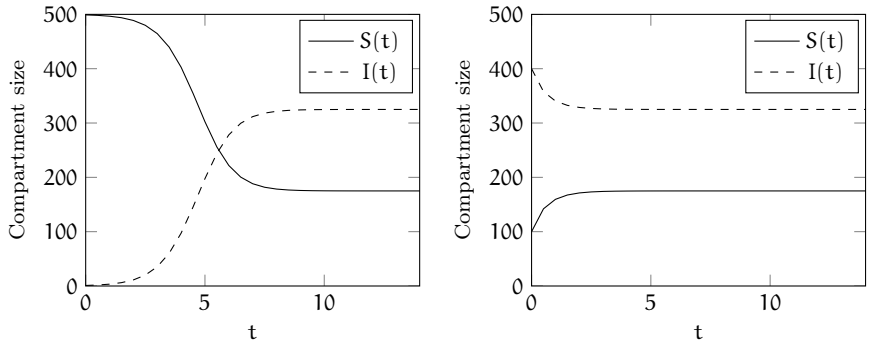
$$\frac{I^{(t+\Delta t)} - I^{(t)}}{\Delta t} = \beta S^{(t)} I^{(t)} - \gamma I^{(t)} \tag{6.6}$$

A key difference between the SIR and the SIS model is that the latter one has a non-trivial steady state when $\beta S^{(t)} = \gamma$, that is, when the number of additional infections caused by a single infected individual per time step is equal to the number of infected persons who recovered during the same time step. This can happen for a non-zero $I^{(t)}$, which means that the infection may never be eradicated from the population unless other preventive measures are taken (which can be modeled by raising γ or lowering β). This behaviour is characteristic of everyday illnesses like the common cold: in every moment, a non-zero fraction of the population is contagious, and although infected people recover all the time, this is counterbalanced by healthy people who have contracted the disease recently. The model is also used for modeling the spreading of computer viruses, assuming that each computer is equipped with an anti-virus software that is capable to returning the machine susceptible after an infection but is not able to make the system completely immune. Fig. 6.5(a) shows the emergence of a non-trivial steady state in the SIS model.

epidemic threshold

basic reproductive rate

It is probably not too surprising now that the SIS model also has an *epidemic threshold*, which is incidentally equal to the threshold of the SIR model. Following the same train of thought, one can work out that the *basic reproductive rate* of the infection is $I^{(t+1)}/I^{(t)} = 1+\beta S^{(t)}-\gamma$, similarly to the



(a) From $I^{(0)} = 1$ (b) From $I^{(0)} = 400$

Fig. 6.5 Non-trivial steady state in the SIS model for $\beta = 0.004$ and $\gamma = 0.7$.

SIR model. When $S^{(0)} > \gamma/\beta$, there are enough susceptible individuals in the population to turn the initial infections into an epidemic, while $S^{(0)} < \gamma/\beta$ implies that the infection will die out quickly. In a steady state, it is true that $S^{(t)} = \gamma/\beta$ and $I^{(t)} = N - \gamma/\beta$; see Fig. 6.5 where $S^{(14)} \approx 0.7/0.004 = 175$ and $I^{(14)} \approx N - 175 = 325$.

Note that the whole train of thought abve does not depend on the exact value of $S^{(0)}$, the only thing that matters whether it is above or below γ/β. The consequence is that the *same* steady state prevalence will be observed in case of an epidemic, no matter how many infections are there in the initial state – as long as the number of susceptible individuals (i.e. the total population minus the number of infected people) is still above γ/β.

6.3.3 The susceptible-infected-recovered-susceptible (SIRS) model

The final compartmental model we are going to study stands half-way between the SIR and the SIS model. In the SIR model, infected people recover eventually and obtain lifelong immunity. In the SIS model, infected people simply become susceptible again. In the SIRS model, infected people recover but stay immune to the infection only for a finite amount of time. This is modeled by the deimmunization rate λ: in every time step, a fraction λ of recovered (and thus immune) individuals become susceptible again (see Fig. 6.3(c). The dynamical equations are as follows:

$$\frac{S^{(t+\Delta t)} - S^{(t)}}{\Delta t} = -\beta S^{(t)} I^{(t)} + \lambda R^{(t)} \tag{6.7}$$

$$\frac{I^{(t+\Delta t)} - I^{(t)}}{\Delta t} = \beta S^{(t)} I^{(t)} - \gamma I^{(t)} \tag{6.8}$$

$$\frac{R^{(t+\Delta t)} - R^{(t)}}{\Delta t} = \gamma I^{(t)} - \lambda R^{(t)} \tag{6.9}$$

This model also has a non-trivial steady state, similarly to the SIS model, but the equilibrium is more delicate: the flow of individuals from the S to the I compartment and the flow from I to R is counterbalanced by the flow from R to S. The exact condition is as follows: $\beta S^{(t)} = \gamma = \lambda R^{(t)}/I^{(t)}$. Due to the additional constraint of $N = S^{(t)} + I^{(t)} + R^{(t)}$ for any t, this set of equations is uniquely solvable for a given β, γ and λ. Fig. 6.6 shows a possible course of the SIRS model with $\beta = 0.004$, $\gamma = 0.7$ and $\lambda = 0.3$, and the observed steady state compartment sizes agree well with the theoretical values: $S^{(t)} = \gamma/\beta = 175$, $I^{(t)} = 97.5$ and $R^{(t)} = 227.5$. The epidemic threshold is the same as for the SIR model since the dynamical equation describing $I^{(t)}$ is the same.

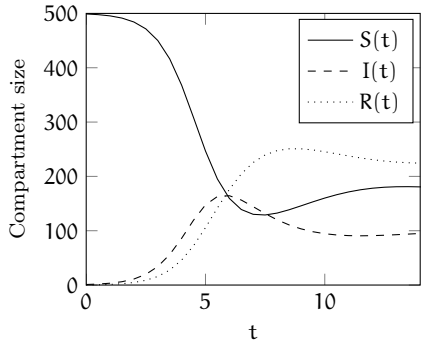


Fig. 6.6 Non-trivial steady state in the SIRS model for $\beta = 0.004$, $\gamma = 0.7$ and $\lambda = 0.3$.

6.4 Compartmental models on networks

The compartmental models introduced in Section 6.3 were designed with an implicit homogeneity assumption in mind; it was assumed that people in different compartments are mixed evenly in the population; in other words, there are no areas in the population that are more infected than others. Such inhomogeneities may slow down the spreading of the infection as individuals living in areas where everyone else is also likely to be infected can not spread the disease as easily as others living in less affected areas. On the other hand, an infected individual who has many contacts in different cities and commutes

regularly between them may easily become a new source of infection for others who would not become exposed to the infection otherwise. Network-based infection models take such inhomogeneities into account by providing a constraining background to the epidemic.

In this section, we will first create a general framework with which we can easily model epidemic processes on networks using igraph, and then use this framework to study the spread of infections in simulated and real-world networks with various topologies.

6.4.1 A general framework for compartmental models on networks

The first thing we will need is a data structure with the following capabilities:

1. Keep track of the states of the vertices in a graph; i.e. store which compartment a given vertex is in.
2. Allow the quick retrieval of the members of a given compartment. This will be one of the most frequently used operations; for instance, the simulation of the spreading process will require us to retrieve all the infected members of the population. It is easily achieved by using a Python dictionary of lists or sets that associates compartment letter codes to the members.
3. Allow the quick retrieval of the compartment of a single vertex. This will be another frequently used operation as we have to know whether a given vertex is susceptible or recovered in order to determine whether it is allowed to contract the infection from a contagious neighbour. The most suitable data structure for this purpose is a list of length N (the size of the population) where each item contains the compartment code of the corresponding vertex.
4. For displaying the results of the simulation, we should also be able to retrieve the relative size of a compartment compared to the whole population.

Since points 2 and 3 require different data structures for an efficient implementation, we will simply create a class that manages both representations:

```
6.11.1) class Compartments(object):
    2)     def __init__(self, graph, codes):
    3)         """Creates a set of compartments for the vertices of
    4)         the given ‘graph‘. The list of compartment codes is
    5)         provided by ‘codes‘.
    6)
    7)         Initially, all the vertices will be in the first
    8)         compartment."""
```

```
9)          self.codes = list(codes)
10)         self.n = graph.vcount()
11)
12)         first_comp = self.codes[0]
13)         self.states = [first_comp] * self.n
14)
15)         self.compartments = dict()
16)         for code in codes:
17)             self.compartments[code] = set()
18)         self.compartments[first_comp].update(xrange(self.n))
19)
20)     def __getitem__(self, code):
21)         """Returns the compartment corresponding to the given
22)         compartment ‘code‘."""
23)         return self.compartments[code]
24)
25)     def get_state(self, vertex):
26)         """Returns the state of the given ‘vertex‘ (i.e. the
27)         code of its compartment)."""
28)         return self.states[vertex]
29)
30)     def move_vertex(self, vertex, code):
31)         """Moves the vertex from its current compartment to
32)         another one."""
33)         self.compartments[self.states[vertex]].remove(vertex)
34)         self.states[vertex] = code
35)         self.compartments[code].add(vertex)
36)
37)     def move_vertices(self, vertices, code):
38)         """Moves multiple vertices from their current compartment
39)         to another one."""
40)         for vertex in vertices:
41)             self.move_vertex(vertex, code)
42)
43)     def relative_size(self, code):
44)         """Returns the relative size of the compartment with
45)         the given ‘code‘."""
46)         return len(self.compartments[code]) / float(self.n)
```

Compartments

`Compartments` uses two key instance variables: *`compartments`* provides the dict-of-sets representation that is required for the efficient retrieval of the members of a given compartment, while *`states`* contains the list of compartment codes for each vertex. Compartments for a SIR model on a given graph can then be created as:

```
graph = Graph.GRG(100, 0.25)
compartments = Compartments(graph, "SIR")
```

The *GRG()* method of the *Graph* class creates a *geometric random graph*, where n vertices are randomly dropped into the unit square $[0;1]^2$ and two vertices are connected if they are closer to each other than a predefined distance threshold r (see Section ?? for more details). In the above example, $n = 100$ and $r = 0.25$, and a possible realization of such a graph (with the vertices placed at the appropriate coordinates in 2D space) is shown on Fig. 6.7.

GRG()
geometric random graph

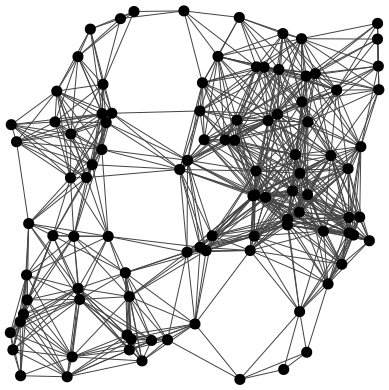


Fig. 6.7 A geometric random graph with $n = 100$ vertices and distance threshold $r = 0.25$.

The *move_vertex()* method of the *Compartments* class moves a vertex to a given compartment, and the implemented *__getitem__()* method allows us to use *compartments* as an ordinary Python dictionary:

```
compartments.move_vertex(2, "I")
compartments.move_vertex(5, "I")
compartments["I"]
```

We can also use *get_state()* to retrieve the state of a given vertex and *relative_size()* to get the relative size of a compartment:

```
compartments.get_state(2)
compartments.get_state(42)
print compartments.relative_size("S")
```

The concrete epidemic models will be implemented as subclasses of an abstract *CompartmentalModel* class. A *CompartmentalModel* holds a reference to the graph on which the epidemic spreads and an instance of *Compartments* to keep track of the state of the epidemic. It also has a *step()* method

CompartmentalModel

where the logic of a specific epidemic model will be implemented by subclasses, and a *reset()* method that takes back the model to its initial state. By convention, abstract methods such as *step()* and *reset()* will raise a *NotImplementedError* in the abstract superclass.

```
6.15.1) class CompartmentalModel(object):
2)      """Abstract base class for compartmental epidemic
3)      models."""
4)
5)      def __init__(self, graph, codes):
6)          """Creates a compartmental model associated to the given
7)          ‘graph‘. ‘codes‘ is a list that provides the compartment
8)          codes of the model."""
9)          self.graph = graph
10)         self.compartments = Compartments(graph, codes)
11)         self.reset()
12)
13)     def relative_compartment_sizes(self):
14)         """Returns the relative sizes of each compartment in the
15)         model."""
16)         return [self.compartments.relative_size(code)
17)                 for code in self.compartments.codes]
18)
19)     def reset(self):
20)         """Resets the compartments to an initial state. This
21)         method must be overridden in subclasses."""
22)         raise NotImplementedError
23)
24)     def step(self):
25)         """Implements the logic of the epidemic model. This
26)         method must be overridden by subclasses."""
27)         raise NotImplementedError
28)
29)     def step_many(self, n):
30)         """Runs ‘n‘ steps of the epidemic model at once by
31)         calling ‘step‘ multiple times."""
32)         for i in xrange(n):
33)             self.step()
```

SIRModel
SISModel
SIRSModel

We will implement three subclasses of the abstract *CompartmentalModel* class: *SIRModel*, *SISModel* and *SIRSModel*. Each of these subclasses will override the constructor of *CompartmentalModel* to allow the specification of the model parameters at construction time, and of course they will also implement the *step()* method that was left unimplemented in *CompartmentalModel* intentionally. Let us start with the simplest SIS model:

```
6.16.1) class SISModel(CompartmentalModel):
    2)     """SIS epidemic model for networks."""
    3)
    4)     def __init__(self, graph, beta=0.1, gamma=0.2):
    5)         """Constructs an SIS model on the given `graph` with
    6)         infection rate `beta` and recovery rate `gamma`."""
    7)         CompartmentalModel.__init__(self, graph, "SI")
    8)         self.beta = float(beta)
    9)         self.gamma = float(gamma)
   10)
   11)     def reset(self):
   12)         """Resets the model by making all the individuals
   13)         susceptible."""
   14)         vs = xrange(self.graph.vcount())
   15)         self.compartments.move_vertices(vs, "S")
   16)
   17)     def step(self):
   18)         """Runs a single step of the SIS model simulation."""
   19)         ## Contagious vertices spread the infection
   20)         s_to_i = set()
   21)         for vertex in self.compartments["I"]:
   22)             neis = self.graph.neighbors(vertex)
   23)             s_to_i.update(sample(neis, self.beta))
   24)         self.compartments.move_vertices(s_to_i, "I")
   25)
   26)         ## Recover some of the infected vertices
   27)         i_to_s = sample(self.compartments["I"], self.gamma)
   28)         self.compartments.move_vertices(i_to_s, "S")
```

When experimenting with epidemic modeling on networks, we are usually interested in how quickly the infection spreads throughout the network, whether it vanishes eventually or converges to a steady state, and the number of people who contracted the disease before it died out (if it did die out in the end). We could theoretically find answers for these questions by simply printing out the number of infected people after every time step, but the resulting stream of numbers is rather hard to interpret and also not very suitable for presentation in a book or on the screen. Therefore, from now on, we will turn to the *matplotlib* Python module to produce nice plots of various measures of interest (such as the size of the infected pool) versus time. If you are not familiar with *matplotlib* yet, refer to its homepage[2] which provides pre-compiled installers and installation instructions for Windows and Mac OS X. Most of the major Linux distributions also include *matplotlib*. If you are using Linux, the first port of call should be the package manager of your distribution to see if it provides a pre-compiled version of *matplotlib*

[2] http://matplotlib.sourceforge.net

before venturing out to compile it yourself. The plots we will show in the rest of this chapter are not produced by *matplotlib* but we will always provide an equivalent Python code that draws a similar plot. If you do not wish to install *matplotlib*, you may still read on as only trivial modifications are needed for the code presented here to print its result to a file which you can plot later with your favourite visualization toolkit.

Before we do any simulations, let us import the *pylab* submodule of *matplotlib* which contains commands that simulate a Matlab-like plotting environment:

```
6.17.1) from matplotlib import pylab
```

First, let us examine the spreading of 10 random epidemics on an Erdős-Rényi random network (see Section 5.2.1) with $N = 10^4$ individuals where each individual has 4 contacts on average; in other words, the number of edges in the network will be 2×10^4. We simulate the first 50 time steps with $\beta = 1/4$ and $\gamma = 1/4$. Note that Erdős-Rényi networks are not necessarily connected, therefore we take the largest connected component before setting up our model if the network is disconnected.

```
6.18.1) def generate_graph(n=10000):
    2)     graph = Graph.Erdos_Renyi(n, m=2*n)
    3)     if not graph.is_connected():
    4)         graph = graph.clusters().giant()
    5)     return graph
    6) 
    7) def simulate(model_class, runs=10, time=50, *args, **kwds):
    8)     results = []
    9)     for i in xrange(runs):
   10)         current_run = []
   11) 
   12)         ## Generate the graph and initialize the model
   13)         graph = generate_graph()
   14)         model = model_class(graph, *args, **kwds)
   15) 
   16)         ## Infect a randomly selected vertex
   17)         v = random.choice(graph.vs)
   18)         model.compartments.move_vertex(v.index, "I")
   19) 
   20)         ## Simulate the time steps
   21)         for t in xrange(time):
   22)             model.step()
   23)             frac_infected = model.compartments.relative_size("I")
   24)             current_run.append(frac_infected)
   25) 
   26)         results.append(current_run)
```

```
27)
28)     return results
```

simulate() was intentionally made generic with three parameters:

'model_class' The class of the model being evaluated. This will be *SISModel* in our case since we want to simulate SIS epidemics.
'runs' The number of epidemic instances to simulate.
'time' The number of time steps to simulate in each run.

The method also accepts an arbitrary number of extra positional and keyword arguments, which are simply passed on to the model constructor. This will allow us to re-use it later when we are testing other epidemic models. It is also very important to note that *simulate()* generates a new network instance for each iteration; this is necessary to ensure that the obtained plots capture the general properties of an epidemic on a specific *class* of networks instead of a specific *instance* of a network class.

The *simulate()* method can be used as follows:

```
6.19.1) results = simulate(SISModel, beta=0.25, gamma=0.25)
```

Note that we have passed in *SISModel*, the class itself, not an instance of it. The *results* variable now contains 10 lists, each of which corresponds to a time series describing the fraction of infected people after each time step:

```
6.20.1) len(results)
    2) print ", ".join("%.4f" % value for value in results[0][:10])
```

Since the spreading process is highly random, we are going to take the mean of the fraction of infected people at each time step `t` across all the simulations and create an *averaged* vector that should smooth out most of the random noise in the process:

```
6.21.1) from itertools import izip
    2) averaged = [mean(values) for values in izip(*results)]
    3) print ", ".join("%.4f" % value for value in averaged[:10])
```

mean() is a convenience function in igraph that allows one to take the mean of values from an arbitrary iterable, and *izip(*items)* is a common Python idiom to take the "transpose" of a list of lists. We can clearly see an increasing trend in the number of infections in the averaged results; the obvious next step is to plot them and see whether the trend continues, and for how long. We will plot all the ten time series with black lines, and plot the average on top of them with a red line using circle markers:

mean()

```
6.22.1) def plot_results(results):
    2)     ## Calculate the means for each time point
    3)     averaged = [mean(items) for items in izip(*results)]
    4)
```

```
    ## Create the argument list for pylab.plot
    args = []
    for row in results:
        args += [row, 'k-']
    args += [averaged, 'r-o']

    ## Create the plot
    pylab.plot(*args)
```

6.23.1)
```
plot_results(results)
```

You should see something similar to the plot on Fig. 6.8. It looks like in this particular instance, the SIS model has a non-trivial steady state, similarly to the case of the homogeneous compartmental model we have seen in Section 6.3.2. But does it happen for all sorts of networks or just Erdős-Rényi ones? What happens if we increase or decrease the rate of infection β? Is there an epidemic threshold below which there is no outbreak? Before we answer these questions, let us also implement the SIR and SIRS models in our framework. Both of these require only a few simple modifications in the *`step()`* method of *`SISModel`*.

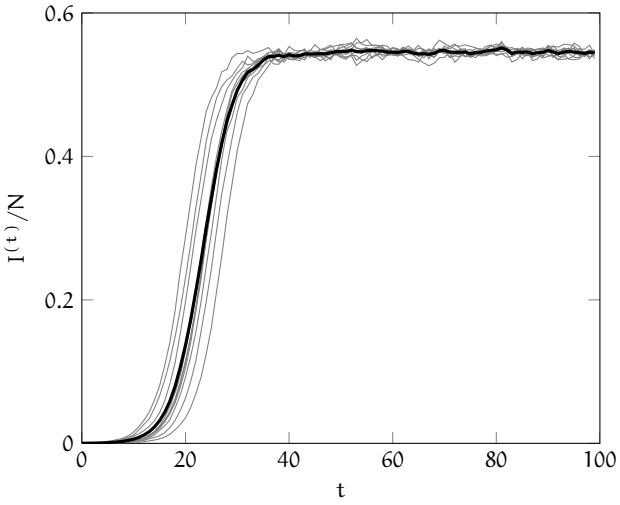


Fig. 6.8 Ten random instances of the spreading of a SIS epidemic on an Erdős-Rényi random network with 10^4 vertices and 2×10^4 connections (average degree = 4). The rate of infection β and the rate of recovery γ were both set to $1/4$.

We start with the SIR model:

6.24.1)
```
class SIRModel(CompartmentalModel):
    """SIR epidemic model for networks."""
```

```
3)
4)      def __init__(self, graph, beta=0.1, gamma=0.2):
5)          """Constructs an SIR model on the given ‘graph‘ with
6)          infection rate ‘beta‘ and recovery rate ‘gamma‘."""
7)          CompartmentalModel.__init__(self, graph, "SIR")
8)          self.beta = float(beta)
9)          self.gamma = float(gamma)
10)
11)     def reset(self):
12)         """Resets the model by making all the individuals
13)         susceptible."""
14)         vs = xrange(self.graph.vcount())
15)         self.compartments.move_vertices(vs, "S")
16)
17)     def step(self):
18)         """Runs a single step of the SIR model simulation."""
19)         ## Contagious vertices spread the infection
20)         for vertex in self.compartments["I"]:
21)             neis = self.graph.neighbors(vertex)
22)             ## We may infect susceptible neighbors only
23)             for nei in sample(neis, self.beta):
24)                 if self.compartments.get_state(nei) == "S":
25)                     self.compartments.move_vertex(nei, "I")
26)
27)         ## Recover some of the infected vertices
28)         i_to_r = sample(self.compartments["I"], self.gamma)
29)         self.compartments.move_vertices(i_to_r, "R")
```

The only substantial difference between *SIRModel* and *SISModel* is that we have to check the neighbors of an infected node one by one before spreading the infection further to ensure that recovered nodes do not get infected again. *SIRSModel* can then easily be derived from *SIRModel* by inheriting the dynamics and adding an extra step that moves back some of the recovered nodes to the susceptible state:

SIRModel

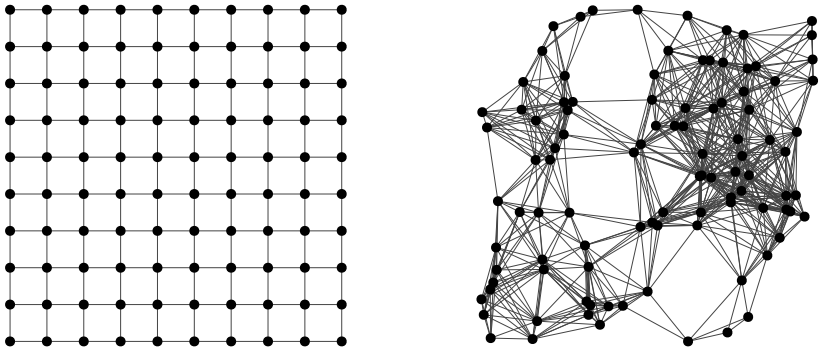
SIRSModel

```
6.25.1) class SIRSModel(SIRModel):
2)      def __init__(self, graph, beta=0.1, gamma=0.2, lambda_=0.4):
3)          SIRModel.__init__(self, graph, beta, gamma)
4)          self.lambda_ = lambda_
5)
6)      def step(self):
7)          ## Recovered vertices may become susceptible again
8)          r_to_s = sample(self.compartments["R"], self.lambda_)
9)          self.compartments.move_vertices(r_to_s, "S")
10)
```

```
11)         ## Re-use the dynamics of the SIR model
12)         SIRModel.step(self)
```

6.4.2 Epidemics on regular and geometric networks

Now that we have implementations for the three major model classes, let us turn our attention to the case of epidemic spreading in regular lattices and geometric random graphs. We have already seen a geometric random graph on Fig. 6.7, but we repeat it on Fig. 6.9 for clarity and contrast it with a regular lattice.



(a) Regular lattice (b) Geometric random graph

Fig. 6.9 A regular lattice of $n = 10 \times 10$ vertices and a geometric random graph with $n = 100$ and a distance threshold of $r = 0.25$.

Both networks are structured on the small scale; in other words, the neighbourhoods of vertices adjacent to each other are very similar. We will see that the correlations between local neighhbourhoods limit the spreading of the network, resulting in a far lower convergence in the case of the SIS model than what we would expect from a homogeneous network where the parameters of the virus are tuned to have a similar basic reproductive rate.

First, we extend our `generate_graph()` method to support different graph types; the current implementation will be able to generate Erdős-Rényi random graphs, regular lattices and geometric random graphs, all tuned to an average vertex degree of 4. With Erdős–Rényi graphs and regular lattices, this is easy to achieve: one can simply specify twice the number of vertices as the desired edge count for `Graph.Erdos_Renyi()`, and 2D regular and *circular* lattices satisfy the desired average vertex degree of 4 trivially. For

geometric random graphs, the calculation is a bit more involved. A vertex in a geometric random graph is connected to those other vertices which are closer to the vertex than r, the distance threshold. The area of a circle with radius r is $r^2\pi$. Since the area of the unit square in which the vertices are placed is 1, the number of *other* vertices that fall closer to a given vertex than r is approximately $(n-1)r^2\pi$, neglecting the fact that vertices closer to the edge than r have a smaller chance to gain neighbors as this bias diminishes as $n \to \infty$ and $r \to 0$. Therefore, the expected degree of a vertex is approximately $(n-1)r^2\pi$, thus the optimal r is $2/\sqrt{(n-1)\pi}$ if we would like to ensure an average degree of four[3].

Our extended *generate_graph()* method is then as follows:

```
6.26.1) def generate_graph(type, n=10000, only_connected=True):
    2)     if type == "er":
    3)         graph = Graph.Erdos_Renyi(n, m=2*n)
    4)     elif type == "lattice":
    5)         size = [int(n ** 0.5), int(n ** 0.5)]
    6)         graph = Graph.Lattice(size, circular=True)
    7)     elif type == "grg":
    8)         r = 2 / ((n-1) * pi) ** 0.5
    9)         graph = Graph.GRG(n, r)
   10)     else:
   11)         raise ValueError("unknown graph type: %r" % type)
   12)     if only_connected and not graph.is_connected():
   13)         graph = graph.clusters().giant()
   14)     return graph
```

```
6.27.1) lattice = generate_graph("lattice", 10000)
    2) mean(lattice.degree())
    3) geo = generate_graph("grg", 10000, only_connected=False)
    4) mean(geo.degree())
```

We also have to tweak our *simulate()* method to accept a keyword argument named *graph_type* which specifies the graph on which the epidemic is simulated. Remember, we cannot generate the graph outside *simulate()* and pass it in because the results would then be affected by the particular structural features of the network instance we provide to *simulate()*.

```
6.28.1) def simulate(model_class, graph_type, runs=10, time=50, *args, **kwds): #labelvrb:simulatediff
    2)     results = []
    3)     for i in xrange(runs):
    4)         current_run = []
    5)
    6)         ## Generate the graph and initialize the model
```

[3] In fact, $r = 2/\sqrt{(n-1)\pi}$ is exact when the geometric random graph is generated on a torus and not on the unit square.

```
7)         graph = generate_graph(graph_type)           #labelvrb:simulatediff2
8)         model = model_class(graph, *args, **kwds)
9) #rm...
```

This is very similar to the earlier version of the *simulate()* method on page 177, the only differences are in lines ?? and ??.

Let us now simulate an SIS epidemic on a 100×100 circular 2D lattice with $\beta = 0.25$ and $\gamma = 0.25$. $\gamma = 0.25$ means that each infected node recovers in each time step with probability 0.25; in other words, the expected length of an infection is 4 time steps. $\beta = 0.25$ seems to imply that each infected individual generates one new infection per time step, since each node has four neighbours in the lattice. With a simple intuitive argument, we may say that each infected node is then expected to generate four additional infections before it recovers, one per time step, therefore we will observe an epidemic outbreak since the basic reproductive number is greater than 1.

Looking at Fig. 6.8, we can see that a similar SIS epidemic on an Erdős-Rényi random network converged to a steady state in less than 50 time steps. Let us be generous and simulate the epidemic for 100 time steps since we have already posited that the spreading will be slowed down by the local structure of the network, and let us print the fraction of infected nodes at every 10th time step:

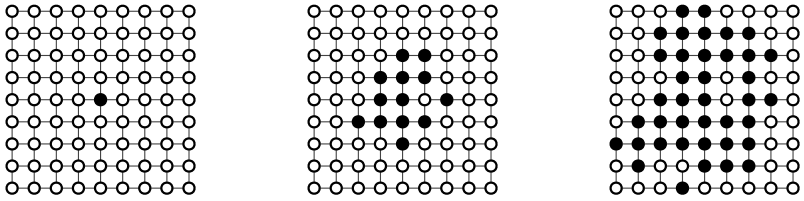
```
6.29.1) results = simulate(SISModel, "lattice", runs=1, time=100)
    2) print ", ".join("%.4f" % value for value in results[0][::10])
```

This is indeed a significant inhibition – there are barely any infected nodes after 100 time steps. Since we have 10,000 nodes in the network, the actual number of infected nodes is less than 30. Either we are dealing with an infection that will quickly die out on this network, or it did not have time to gain momentum yet. Fig. 6.10 explains why we see this: the infection spreads out slowly from a single source and it takes some time until the infection propagates to nodes far from the source of the epidemic.

With a bit of experimentation, one can find out that about 600 time steps are needed in this particular case for the network to reach the steady state; this is a dramatic increase compared to the case of Erdős-Rényi networks. Fig. 6.11 shows ten random instances of the SIS epidemic on this lattice; a figure similar to this can be plotted using *matplotlib* by invoking our *plot_results()* method again:

```
6.30.1) results = simulate(SISModel, "lattice", time=600)
    2) plot_results(results)
```

Note that despite the apparently large variance between the different curves (especially before $t = 400$), the process still converges approximately to the same fraction of infected individuals as we have seen for the Erdős-Rényi networks on Fig. 6.8.



(a) $t = 0$ (b) $t = 3$ (c) $t = 6$

Fig. 6.10 A SIS epidemic on a 9×9 lattice with $\beta = 0.5$ and $\gamma = 0.1$ at $t = 0$, 3 and 6. Note how the infection stays localized around the source of the infection (the central node).

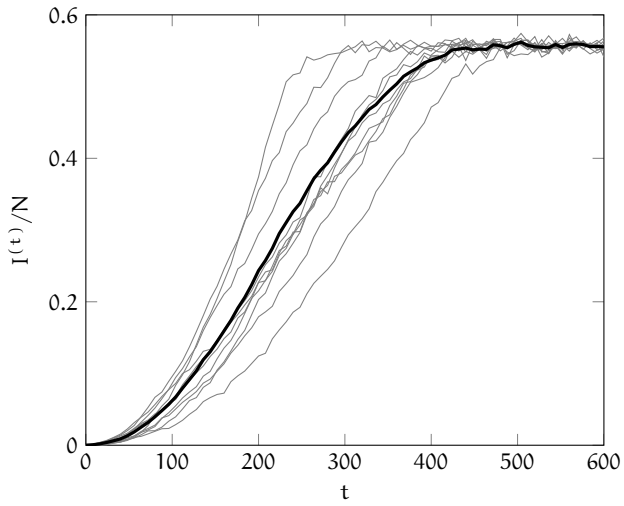


Fig. 6.11 Ten random instance of the spreading of a SIS epidemic on a circular lattice of size 100×100. The rate of infection β and the rate of recovery γ were both set to $1/4$.

Finally, we repeat our simulation with a geometric random graph of the same average connectivity. In this case, even $t = 600$ is not enough to reach the steady state, which emerges only around $t = 1500$.

```
6.31.1) results = simulate(SISModel, "grg", time=2000)
    2) plot_results(results)
```

The plot on Fig. 6.12 shows the aggregated results of ten simulations. Besides the late emergence of the steady state, we can also observe a very high variance between the results of individual simulations, which makes our estimate for the average fraction of infected individuals less reliable. The individual curves do not seem to stabilize around the estimated average as t

increases, which suggests that there aree local fluctuations in the prevalence of the infection. This can be explained by the clustered structure of geometric random graphs. Let us take a look at Fig. 6.9 again which compares regular lattices and geometric random graphs. It can clearly be seen that a geometric random graph can often be subdivided into dense regions which are connected only by a few links. When an infection reaches one of these dense regions, it can infect the whole cluster quickly, but it takes some time until it can spread further to the next cluster as most of the nodes inside the cluster just spend their time with re-infecting each other over and over again. The lower-most curve on Fig. 6.12 illustrates this clearly around $t = 900$ when the infection reached a large and previously uninfected cluster, and the relative prevalence quickly rose from around 0.25 to more than 0.5 in less than 300 time steps. Similarly, fluctuations may also arise from events when the infection is cleared from a cluster thanks to the natural recovery process; in this case, it also takes some time until the members of the cluster contract the contagion again from the outside world.

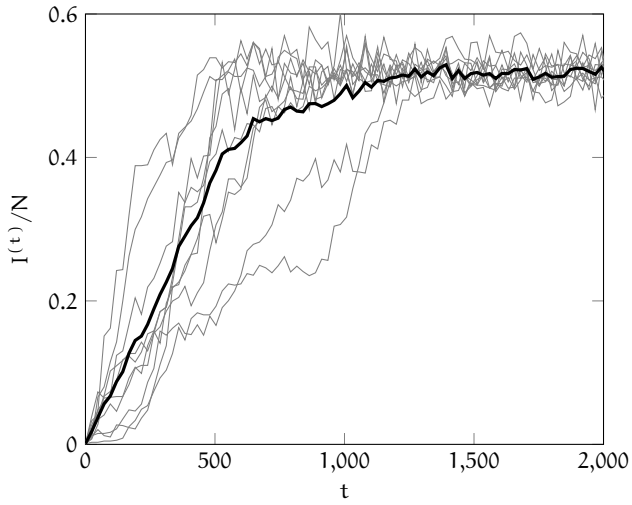


Fig. 6.12 Ten random instance of the spreading of a SIS epidemic on a geometric random graph with 10^4 vertices and an average degree of 4. The rate of infection β and the rate of recovery γ were both set to 1/4.

Let us now turn our attention to the problem of the epidemic threshold. For homogeneous models, we have seen that the epidemic threshold corresponds to the case where the basic reproductive rate of the virus is exactly 1, that is, where every single infection generates exactly one new infection during its lifetime. The lifetime of the infection is simply $1/\gamma$, therefore a naïve approximation of the basic reproductive rate of a virus spreading on a network with an average degree of $\langle k \rangle$ is $\beta \langle k \rangle / \gamma$, since a node may infect each of its

$\langle k \rangle$ neighbors with probability β in each time step. However, note that a node may not infect another node that is already infected, and usually there is at least one other node in the vicinity of a given node which is already infected: the one from which the node itself contracted the infection! A more precise approximation of the reproductive rate would therefore be $\beta(\langle k \rangle - 1)/\gamma$. In the case of $\gamma = 0.25$ and $\langle k \rangle = 4$ (which we have studied by simulations), the epidemic threshold is to be expected around $\beta = \gamma/(\langle k \rangle - 1) \approx 0.083$. We are going to try and confirm or refute this conjecture with numerical simulations in igraph.

There is one difficulty with simulations when β is small (compared to γ). When starting the simulation with a single infected node, it is often the case that the infected node recovers before it had the chance to start spreading the infection. However, we have seen that the initial number of infected individuals does not influence the steady state in case of a homogeneous population. If this property of the SIS model holds for networks as well, we can simply start the simulation with, say, half the nodes being infected, and it will still converge to the same steady state. To this end, we will add an extra argument to *simulate()* that controls the initial number of infected nodes and we will run our simulations with a higher number of initial infections; for instance, 100 vertices, which amounts for 1% of the population.

```
6.32.1) def simulate(model_class, graph_type, num_infected=1, runs=10,                time=50, *args, *
    2)     results = []
    3)     for i in xrange(runs):
    4)         current_run = []
    5)
    6)         ## Generate the graph and initialize the model
    7)         graph = generate_graph(graph_type)
    8)         model = model_class(graph, *args, **kwds)
    9)
   10)         ## Infect randomly selected vertices
   11)         vs = random.sample(range(graph.vcount()), num_infected)   #labelvrb:simulatediff3
   12)         model.compartments.move_vertices(vs, "I")    #labelvrb:simulatediff4
   13)
   14)         ## Simulate the time steps
   15)         for t in xrange(time):
   16)             model.step()
   17)             frac_infected = model.compartments.relative_size("I")
   18)             current_run.append(frac_infected)
   19)
   20)         results.append(current_run)
   21)
   22)     return results
```

This is again very similar to the previous version of the *simulate()* method, the only changes are in lines ?? and ??. Let us now run the SIS model

for Erdős–Rényi random networks with $\beta = 0.06, 0.07, \ldots, 0.1$, $\gamma = 0.25$ and an initial infection size of 100:

```
6.33.1) def find_epidemic_threshold(type, betas, *args, **kwds):
    2)     for beta in betas:
    3)         results = simulate(SISModel, type, 100, beta=beta,
    4)                            *args, **kwds)
    5)         avg = mean(run[-1] for run in results)
    6)         print("%.2f %.4f" % (beta, avg))
```

```
6.34.1) >>> betas = [0.06, 0.07, 0.08, 0.09, 0.10]
    2) >>> find_epidemic_threshold("er", betas, gamma=0.25, time = 100)
```

```
0.06 0.0000
0.07 0.0002
0.08 0.0081
0.09 0.0501
0.10 0.1294
```

Indeed there is a threshold-like behaviour here; for $\beta = 0.06, 0.07$ or 0.08, the fraction of infected individuals decreased from the initial 1% to 0%, 0.02% and 0.81%, respectively, while it increased to 5.01% for $\beta = 0.09$ and to 12.94% for $\beta = 0.10$. Our reasoning was proven to be surprisingly accurate here. But does it hold for lattices or geometric random graphs? Note that the simulation has to be run for a much longer time to allow the steady state to emerge, so be patient.

```
6.35.1) >>> find_epidemic_threshold("lattice", betas, gamma=0.25, time=1000)
```

```
0.06 0.0000
0.07 0.0000
0.08 0.0000
0.09 0.0000
0.10 0.0000
```

This result is surprising, but not quite unexpected. In an Erdős–Rényi network, connections are placed essentially randomly, thus the population can still be considered homogeneous, similarly to the traditional compartmental models. The epidemic threshold is therefore governed by the same principles. However, the structure of a lattice is much more correlated and gives rise to interesting phenomena like an increased epidemic threshold, which of course stems from a decreased basic reproductive rate. But where is the real epidemic threshold then? We can find it somewhere between 0.13 and 0.14:

```
6.36.1) >>> betas = [0.11, 0.12, 0.13, 0.14, 0.15]
    2) >>> find_epidemic_threshold("lattice", betas, gamma=0.25, time=1000)
```

```
0.11 0.0000
0.12 0.0000
```

```
0.13 0.0152
0.14 0.1761
0.15 0.2566
```

For geometric random graphs, we have to run the simulation for 5000 time steps and use 20 runs per β to obtain a reliable result due to the high variance in the course of the epidemic in individual runs. The epidemic threshold is somewhere between 0.1 and 0.11, although the transition seems to be less sharp than in the case of Erdős–Rényi networks or regular lattices:

```
6.37.1) >>> betas = [0.08, 0.09, 0.10, 0.11, 0.12]
    2) >>> find_epidemic_threshold("grg", betas, gamma=0.25,  ...                                time=5000,
```

```
0.08 0.0000
0.09 0.0000
0.10 0.0018
0.11 0.0117
0.12 0.0366
```

6.4.3 Epidemics on scale-free networks

In the previous section, we have investigated three classes of networks on which the SIS epidemic shows similar behaviour. For Erdős–Rényi random networks, regular lattices and geometric random graphs, a steady state of the SIS model emerges if β is large enough while γ and $\langle k \rangle$ is kept constant. In the steady state, a a fixed fraction of the population stays infected although the set of infected individuals changes all the time. Thus, the network-based models we have seen so far preserved the non-trivial steady state and the existence of an epidemic threshold from traditional compartmental models on homogeneous populations, but the relationship between β, γ and the epidemic threshold became more complicated as it now depends on the structure of the network as well.

A common property of all the random networks we have studied in the previous section is that they have a well-defined average connectivity value $\langle k \rangle$; in other words, node i is in contact with $k_i \approx \langle k \rangle$ other nodes. Of course a $\langle k \rangle$ value can be calculated for any finite network, but there is a special class of networks called scale-free networks where this value does not make much sense. The degree distribution of this type of networks resembles a power-law distribution; in other words, the probability of finding a node with degree x is proportional to $x^{-\alpha}$ for some exponent $\alpha > 1$. Unlike the Poisson distribution (the degree distribution of Erdős–Rényi random networks), the mean of the power-law distribution has no special meaning as there is no such thing as an "average" node in the network which has a characteristic degree $\langle k \rangle$; we can easily find nodes with much larger degrees than $\langle k \rangle$, but the vast

majority of nodes have degrees smaller than $\langle k \rangle$. Therefore, it does not hold that $k_i \approx \langle k \rangle$. You can read more about this phenomenon in Section **??**. In this section, we will see that this behaviour of scale-free networks gives rise to a surprising behaviour of the SIS model.

Scale-free networks are especially important since many real-world networks were shown to have degree distributions that resemble a power-law. One of the frequently cited examples is the network model of the Internet, where each vertex represents an autonomous system (AS) such as a router or a gateway machine to a local network, and each edge represents a physical connection between two autonomous systems. This network is used by computer viruses to spread from one computer to another. Pastor-Satorras and Vespignani (2001) have observed that computer virus incidents are very rare compared to the size of the network, and most surviving viruses live at a very low level of persistence; in other words, only a tiny fraction of all the computers connected to the Internet are affected by a given virus. If the Internet were randomly wired like an Erdős–Rényi network, or regularly wired like a lattice, we would know immediately what this means: the basic reproductive rate of computer viruses are just above the epidemic threshold, and an antiviral campaign would probably raise γ enough to eradicate these infections from the network. However, it is very unlikely that thousands of viruses that were developed (probably) independently are all tuned to just slightly above the epidemic threshold.

To understand the puzzling properties of computer viruses, Pastor-Satorras and Vespignani (2001) have conducted simulations of the SIS model on large scale-free networks and evaluated the persistence of viruses as a function of $1/\beta$ while keeping γ constant. We will now follow the trail of Pastor-Satorras and Vespignani (2001) and repeat some of their analyses using igraph.

A simple random graph model that generates scale-free networks with a power-law degree distribution is the well-known Barabási–Albert model (Barabási and Albert (1999), see Section **??** for a more detailed description), which is implemented by the *Barabasi()* method of the *Graph* class. We will use this method to generate scale-free networks. Pastor-Satorras and Vespignani (2001) used $\langle k \rangle = 6$, and performed simulations for various numbers of nodes ranging from $N = 10^3$ to $N = 8.5 \times 10^6$. For each combination of N and β, they have performed at least 1,000 simulations, and recorded the prevalence of the virus (i.e. the fraction of infected nodes) for the steady state of each simulation.

Barabasi()

The fundamental building block will then be a function that receives N and β, and simulates the SIS model until the steady state. Note that we do not know in advance how many steps are required to reach the steady state, therefore we adopt an adaptive strategy that tries to detect the steady state automatically. We will run the simulation in blocks of 100 time steps. In each block, we record the fraction of infected individuals after each time step and calculate the mean and variance of this vector using an instance of *RunningMean*; if the variance falls below a given threshold, we consider the

simulation to have reached the steady state and return the mean as the result. This approach is likely to work for larger networks but fail for smaller ones where the local fluctuations are too large. Each simulation will start from a state where half of the nodes are infected, just like the way Pastor-Satorras and Vespignani (2001) did:

```
6.38.1) def simulate_sf_single(n, beta, gamma=0.25, epsilon=0.01):
    2)     graph = Graph.Barabasi(n, 3)
    3)     model = SISModel(graph, beta=beta, gamma=gamma)
    4)     infected = random.sample(range(n), n // 2)
    5)     model.compartments.move_vertices(infected, "I")
    6)
    7)     rm = RunningMean()
    8)     while len(rm) == 0 or rm.sd > epsilon:
    9)         rm.clear()
   10)         for i in xrange(100):
   11)             model.step()
   12)             rm.add(model.compartments.relative_size("I"))
   13)
   14)     return rm.mean
```

A quick test first:

```
6.39.1) >>> simulate_sf_single(10000, 0.05)
        0.192701
```

To save some time, we will run 10 simulations for each combination of N and α, and calculate the average prevalence returned by *simulate_sf_single()*:

```
6.40.1) def simulate_sf(n, beta, gamma=0.25, epsilon=1, runs=10):
    2)     results = []
    3)     for i in xrange(runs):
    4)         frac = simulate_sf_single(n, beta, gamma, epsilon)
    5)         results.append(frac)
    6)     return mean(results)
```

A quick test again:

```
6.41.1) >>> simulate_sf(10000, 0.05)
        0.1920451
```

The paper of Pastor-Satorras and Vespignani (2001) showed the fraction of infected individuals in the steady state as a function of $\phi = \gamma/\beta$ for $N = 10^5$, $N = 5 \times 10^5$, $N = 10^6$, $N = 5 \times 10^6$ and $N = 8.5 \times 10^6$. We will therefore also invoke *simulate_sf()* for a representative set of (N, β) combinations which we set up now:

```
6.42.1) phi_for_n = dict()
    2) phi_for_n[100000]  = [7.5, 8, 10]
    3) phi_for_n[500000]  = [9, 9.5, 10, 10.5, 11, 11.5]
    4) phi_for_n[1000000] = [8.5, 9, 9.5, 10, 10.5, 11, 11.5]
    5) phi_for_n[5000000] = [12, 13, 14, 15, 16]
    6) phi_for_n[8500000] = [18, 20]
```

'phi_for_n' associates values of N to values of ϕ from which we can infer β if γ is given in advance. We will use $\gamma = 0.25$:

```
6.43.1) def sf_prevalence_plot(phi_for_n):
    2)     gamma = 0.25
    3)
    4)     symbols = ('b+', 'gs', 'rx', 'co', 'mD')
    5)     plots = []
    6)     for idx, n in enumerate(sorted(phi_for_n.keys())):
    7)         xs, ys = [], []
    8)         for phi in phi_for_n[n]:
    9)             beta = gamma / phi
   10)             frac = simulate_sf(n, beta, gamma)
   11)             xs.append(phi)
   12)             ys.append(frac)
   13)         args.extend(xs, ys, symbol[idx])
   14)
   15)     pylab.plot(*args)
```

Invoking *sf_prevalence_plot()* on 'phi_for_n' generates a plot similar to the one on Fig. 6.13. However, this may require several hours on a standard desktop computer, so make sure you run this only when you have plenty of time, or run it in the background and save the values of *xs* and *ys* in a file instead of plotting them directly.

It might be a bit disappointing to wait half a day for only a few points on the 2D plane, but the consequences of this result make up for it. First of all , we can conclude that the results are independent from our choice of N as several points for different values of N are almost exactly at the same position for the same value of ϕ (e.g., for $\phi = 11$). Second, but more importantly, note that the markers line up almost perfectly when they are plotted using a logarithmic y axis. This means that there exists a constant c such that $I/N \propto \exp(-c\gamma/\beta)$, thus the fraction of infected individuals will never reach zero, no matter how small β is. In other words, *there is no epidemic threshold*, and the virus can never be eradicated completely from the network.[4]

[4] To be precise, we note that the absence of an epidemic threshold applies only for infinite scale-free networks with $2 < \alpha < 3$, and finite ones do possess an epidemic threshold (May and Lloyd, 2001; Pastor-Satorras and Vespignani, 2002a). Infinite scale-free networks with $\alpha \geq 4$ were shown to behave like Erdős–Rényi random networks (Pastor-Satorras

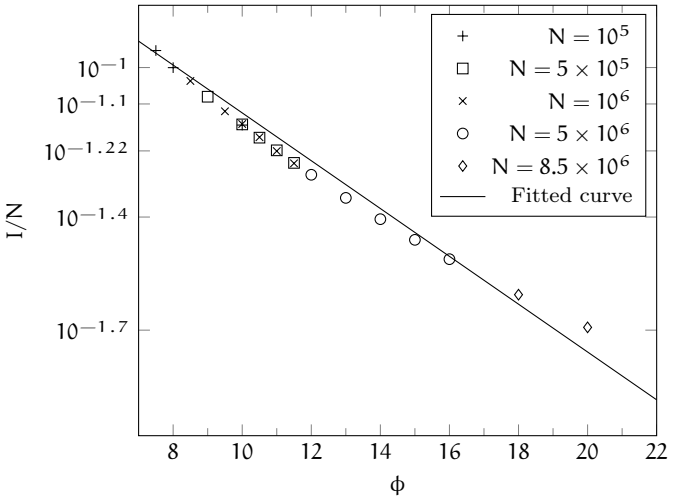


Fig. 6.13 The prevalence of the virus in the steady state of an SIS epidemic on a scale-free network for different network sizes. The full line is a fit to the form $I/N \sim \exp(-c\phi)$ where c is an appropriate constant.

Although this is quite baffling at first glance, there is a logical explanation for the phenomenon. In scale-free networks, most of the nodes are low-degree, but it is very easy to find a high-degree node in the vicinity of any low-degree node. A node stays infected for approximately the same amount of time $1/\gamma$, no matter how many neighbours it has, but high-degree nodes can spread the infection much more efficiently since the number of infections a node can create in a time step is proportional to both β and the degree of the node itself. Therefore, an infection of a high-degree node will cause an avalanche of further infections in the next few time steps, while an infection of a low-degree node can reach a high-degree node easily, which, in turn, will initiate a similar avalanche. High-degree nodes are therefore often called *super-spreaders* as they are responsible for a significant amount of the infections in a scale-free network. A more formal treatment of the problem is given in Pastor-Satorras and Vespignani (2001) and Pastor-Satorras and Vespignani (2002a). A recent article of Castellano and Pastor-Satorras (2010) have shown that the phenomenon is not even limited to scale-free networks; the same behaviour is observed for networks where the maximum degree k_{max} diverges with the size of the network.

super-spreaders

and Vespignani, 2001). For finite networks, Pastor-Satorras and Vespignani (2002a) have demonstrated that the epidemic threshold is $\beta/\gamma = \langle k \rangle / \langle k^2 \rangle$, which is in concordance with the absence of an epidemic threshold in infinite scale-free networks since in those networks $\langle k^2 \rangle \to \infty$.

6.5 Vaccination strategies

Another key problem in epidemiology is the prevention of epidemics. Typically, one wishes to prevent the outbreak of an epidemics in a population of N with the lowest possible quantity of vaccines to be used. The question is then usually as follows: how many people shall we vaccinate to prevent the outbreak of an epidemics? There is also an alternative formulation of the same question: if our budget allows the distribution of m vaccines in the population, who should we give them to in order to prevent the outbreak with high probability or at least slow it down as much as possible?

The simplest vaccination strategy one can consider is to immunize m individuals from the population randomly. This strategy leads to success in homogeneous populations and in networks where the average connectivity $\langle k \rangle$ is representative of the degrees of the vertices (i.e. $k_i \approx \langle k \rangle$ for vertex i), but not in scale-free networks. To show this, let us extend our *`simulate()`* method again, this time with an extra argument that specifies an *immunization function* which receives the graph as a parameter and returns a list of vertices that should be immunized. We will then implement a simple function that selects m vertices randomly.

```
6.44.1) def simulate(model_class, graph_type, num_infected=1, runs=10,                time=50, immunize
    2)     results = []
    3)     for i in xrange(runs):
    4)         current_run = []
    5)
    6)         ## Generate the graph and initialize the model
    7)         graph = generate_graph(graph_type)
    8)         model = model_class(graph, *args, **kwds)
    9)
   10)         ## Infect randomly selected vertices
   11)         vs = random.sample(range(graph.vcount()), num_infected)
   12)         model.compartments.move_vertices(vs, "I")
   13)
   14)         ## Immunize the vertices selected by the immunizer
   15)         if immunizer is not None:
   16)             vs = immunizer(graph)   #labelvrb:simulatediff5
   17)             model.compartments.move_vertices(vs, "R")   #labelvrb:simulatediff6
   18)
   19)         ## Simulate the time steps
   20)         for t in xrange(time):
   21)             model.step()
   22)             frac_infected = model.compartments.relative_size("I")
   23)             current_run.append(frac_infected)
   24)
   25)         results.append(current_run)
```

```
26)
27)     return results
```

The key differences are in lines **??** and **??**; in line **??**, we ask the immunizer function to select which vertices are to be immunized, and we move these vertices to the R (recovered) state in line **??**. This implies that we cannot use our present implementation of *SISModel* as this model does not have an R compartment; however, *SIRModel* is not suitable either as vertices should be moved back to the S compartment after recovery in the scenario we wish to study. Therefore we are going to implement a variant of the SIS model which we will call *SISModelWithImmunity*: *SISModelWithImmunity*

```
6.45.1) class SISModelWithImmunity(CompartmentalModel):
2)      """SIS epidemic model for networks where a separate compartment
3)      holds immunized vertices."""
4)
5)      def __init__(self, graph, beta=0.1, gamma=0.2):
6)          """Constructs an SIS model on the given ‘graph‘ with
7)          infection rate ‘beta‘ and recovery rate ‘gamma‘, and creates
8)          an R compartment for immunized nodes."""
9)          CompartmentalModel.__init__(self, graph, "SIR")
10)         self.beta = float(beta)
11)         self.gamma = float(gamma)
12)
13)     def reset(self):
14)         """Resets the model by making all the individuals
15)         susceptible."""
16)         vs = xrange(self.graph.vcount())
17)         self.compartments.move_vertices(vs, "S")
18)
19)     def step(self):
20)         """Runs a single step of the SIS model simulation."""
21)         ## Contagious vertices spread the infection
22)         s_to_i = set()
23)         for vertex in self.compartments["I"]:
24)             neis = self.graph.neighbors(vertex)
25)             for nei in sample(neis, self.beta):
26)                 if self.compartments.get_state(nei) == "S":
27)                     s_to_i.add(nei)
28)         self.compartments.move_vertices(s_to_i, "I")
29)
30)         ## Recover some of the infected vertices
31)         i_to_s = sample(self.compartments["I"], self.gamma)
32)         self.compartments.move_vertices(i_to_s, "S")
```

Now we can implement the function responsible for random immunization:

```
6.46.1) def random_immunizer(graph, m):
    2)     return random.sample(range(graph.vcount()), m)
```

The curious reader may have noticed that there is a discrepancy between the number of parameters of *random_immunizer()* as defined above and the number of parameters we pass to it in *simulate()*. *random_immunizer()* requires two parameters: the graph to be immunized and the number of vaccines that we can use, while *simulate()* specifies only the graph itself. This is intentional; we will use the *partial()* function of the *functools* module in the standard Python library to bind the second argument of *random_immunizer()* to a specific value before passing it in to *simulate()*. The following code snippet illustrates the usage of *partial()*:

```
6.47.1) g = Graph.GRG(100, 0.2)
    2) random_immunizer(g, 5)    ## selects 5 random vertices
    3) from functools import partial
    4) func = partial(random_immunizer, m=5)
    5) func(g)    ## the value of m was bound to 5 by partial()
```

In Section 6.4.2, we have conjectured that the epidemic threshold on Erdős–Rényi networks is approximately given by $\beta(\langle k\rangle - 1)/\gamma = 1$. Vaccination effectively removes the immunized nodes from the network, at least from the point of view of the virus, since immunized nodes can not be infected, they do not spread the infection themselves and they never return to the susceptible state. Therefore, we can simply consider the immunization of a fraction ρ of the nodes as decreasing the average connectivity of the network to $(1-\rho)\langle k\rangle$ as each node that is not yet immune will have $\rho\langle k\rangle$ immune neighbors. The epidemic threshold is then given by the following equation:

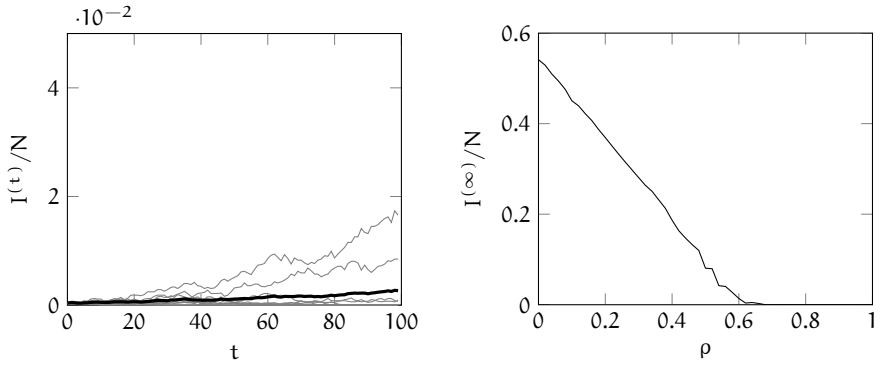
$$\frac{\beta(1-\rho)(\langle k\rangle - 1)}{\gamma} = 1 \tag{6.10}$$

Let us now consider an Erdős–Rényi network with $\beta = 0.25, \gamma = 0.25$ and $\langle k\rangle = 4$. The formula tells us that it is enough to vaccinate two third of the population to lower the basic reproductive rate of the virus to 1. Let us confirm this with simulations on an SIS model, starting from an initial outbreak affecting 10 out of the $10,000$ nodes that are generated by *generate_graph()*:

```
6.48.1) immunizer = partial(random_immunizer, m=6666)
    2) results = simulate(SISModelWithImmunity, "er", time=100,         num_infected=10, beta=0.25, g
    3) plot_results(results)
```

Indeed, immunizing two third of the population is enough to keep the infection at bay; see the plots on Fig. 6.14(a) where the vast majority of simulated infections did not turn into an epidemic. Even if our budget allows only the immunization of 20% of the population, we still manage to lower the fraction of infected individuals in the steady state from about 0.55 (Fig. 6.8) to 0.36. Fig. 6.14(b) shows how the prevalence in the steady state depends on

ρ, the fraction of immunized individuals, confirming the linear dependence prescribed by Eq. 6.10.



(a) $\beta = 0.25, \gamma = 0.25, \rho = 2/3$ (b) $\beta = 0.25, \gamma = 0.25$

Fig. 6.14 Immunization may be used to push the basic reproductive rate under 1 and prevent the outbreak of an epidemic. Left: the course of an SIS epidemic on an Erdős–Rényi network near the epidemic threshold after immunization; right: the prevalence of the virus in the steady state on an Erdős–Rényi network at different levels of immunization.

However, the random immunization strategy breaks down completely when it is applied to a scale-free network. Immunizing a randomly selected fraction of the population slows down the spreading of the epidemic, but it is not able to eradicate it completely. The reason is that most of the vertices in a scale-free network have a relatively low degree, while the nodes that are mostly responsible for the rapid spreading of the epidemic are the super-spreaders, i.e. those with a high degree. Pastor-Satorras and Vespignani (2002b) have shown that a better strategy for scale-free networks is to start immunizing the nodes in decreasing order of their degrees. This is the motivation behind the vaccination of public workers such as staff nurses, teachers and policemen as they are more likely to get in contact with others and therefore are more likely to spread the contagion should they get infected.

The degree-based immunization strategy can be implemented very easily:

```
6.49.1) def degree_immunizer(graph, m):
    2)     degrees = graph.degree()
    3)     order = sorted(range(graph.vcount()),
    4)                    key=degrees.__getitem__, reverse=True)
    5)     return order[:m]
```

We also have to extend *generate_graph()* for the last time to handle scale-free networks as well.

```
6.50.1) def generate_graph(type, n=10000, only_connected=True):
    2)     if type == "sf":
```

```
        graph = Graph.Barabasi(n, 2)
    elif type == "er":
        graph = Graph.Erdos_Renyi(n, m=2*n)
#rm...
```

We can now compare the random and the degree-based strategies on Erdős–Rényi and scale-free networks for a given configuration of β and γ. We will use $\beta = 0.25$ and $\gamma = 0.25$ like before, and print the fraction of infected individuals after 100 steps of a single run of a SIS epidemic while immunizing 20% of the nodes, selected according to the two competing strategies. For the random strategy, the following result is observed:

```
def test_immunizer(immunizer):
    for model in ("er", "sf"):
        results = simulate(SISModelWithImmunity, model,
            time=100, num_infected=10, immunizer=immunizer,
            beta=0.25, gamma=0.25)
        print("%s: %.4f" % (model, results[0][-1]))
```

```
>>> immunizer = partial(random_immunizer, m=2000)
>>> test_immunizer(immunizer)
```

```
er: 0.3673
sf: 0.3641
```

There is practically no difference in the effectiveness of the two strategies, but remember: since there is no epidemic threshold in scale-free networks, we cannot lower the number of susceptible individuals enough to prevent an outbreak, at least not with a uniform immunization strategy (see Pastor-Satorras and Vespignani (2002b) for more details). Let us check what happens in the targeted case where the individuals with the largest number of connections are immunized!

```
>>> immunizer = partial(random_immunizer, m=2000)
>>> test_immunizer(immunizer)
```

```
er: 0.1392
sf: 0.0000
```

This is a very impressive result; the strategy managed to exterminate the infection completely in the scale-free network, and it was also shown to be more effective in an Erdős–Rényi network than the random strategy.

There is one problem with degree-based immunization: in a real-world scenario, it is often not known who are the most connected people in the contact network. Cohen et al (2003) have proposed a strategy for such cases. Let us assume that we would like to immunize $\mathfrak{m}$ people from the contact network. First, we select $\mathfrak{m}$ people randomly, then we ask each of these people to nominate one of his acquaintances, and we immunize those who have been nominated:

```
6.54.1) def neighbor_immunizer(graph, m):
    2)     vs = random.sample(graph.vs, m)
    3)     result = []
    4)     for v in vs:
    5)         result.append(random.choice(v.neighbors()).index)
    6)     return result
```

```
6.55.1) >>> immunizer = partial(random_immunizer, m=2000)
    2) >>> test_immunizer(immunizer)
```

```
er: 0.3457
sf: 0.1154
```

This strategy is not as effective as the degree-based one, but it does not require any *a priori* information about the vertex degrees. We leave a more detailed investigation of these strategies as an exercise for the Reader.

6.6 Exercises

▶ EXERCISE 6.1. Implement discrete simulations for the SIS and SIRS compartmental models. Confirm the presence of the epidemic threshold.

▶ EXERCISE 6.2. Investigate the extension of the SIR model by population dynamics; i.e. assume that babies are born at a rate of κ and enter the susceptible pool, while people die from natural causes with the same rate and leave whichever pool they are already in. How does the dynamics of the model change? Is there a non-trivial steady state in the SIR model with births and deaths? Is there an epidemic threshold?

▶ EXERCISE 6.3. In Section 6.4.2, we have seen that the behaviour of the SIS model is slightly different for regular lattice and geometric random graphs as the variance between the curves of the individual simulations are much higher for geometric random graphs. Quantify the variance of the two models and compare the variances of the initial stage and the steady state.

▶ EXERCISE 6.4. In Section 6.5, we have studied three simple vaccination strategies that aim to prevent the outbreak of an epidemic or at least to decrease the number of affected individuals. Fig. 6.14(b) shows how the fraction of infected individuals in the steady state depends on the fraction of the immunized population for Erdős–Rényi networks when the random strategy is used. Implement a function that plots a figure similar to this, and extend it to support different graph types and any of the three vaccination strategies we have learned about.

▶ EXERCISE 6.5. All the network-based models that we have studied in this chapter assume that the underlying contact network is static and any link may spread the infection in any time step. This is reasonable for diseases that spread rapidly as the time scale of the epidemic is smaller than the typical life span of a relationship between individuals. Modify the general framework introduced in Section 6.4.1 to take into account that edges can be active or inactive and thus may not be used by the spreading process in all steps.

Chapter 7
Spectral Embeddings

Frequently in graph theory and related fields of mathematics it can be convenient to work with matrices associated with a graph. The adjacency matrix provides the most direct representation and manipulations of this matrix can be used for a variety of tasks. Various forms of Laplacian matrices can be used in the study of random walks on graphs and the determination of minimum cuts. One of the most intriguing aspects of working with these matrix representations is the connection between properties of the graphs and spectral properties of the matrices (Chung, 1997).

In this chapter we will introduce a technique using the spectral decomposition of a matrix that can be used for various tasks including vertex clustering, vertex classification, vertex nomination or ranking, as well as for exploratory data analysis. The spectral decomposition of a matrix associated with a graph provides a way to embed the vertices of a graph as points in finite dimensional Euclidean space. This embedding allows for standard statistical and machine learning methodology for multivariate Euclidean data to be used for graph inference.

We will give an overview of the mathematics behind these embeddings while discussing how to use these embeddings in igraph for a variety of tasks. While discussing the various optional arguments we will investigate three different real data examples to illustrate how various options impact the embeddings and subsequent analysis. For the purpose of visualization and to keep the ideas general we will focus on using the embedding as an exploratory data analysis tool to investigate the graphs in two dimensions. We will also consider tasks such as clustering the vertices and finding interesting vertices. We will demonstrate the embeddings with simulated examples and show how these examples relate to theoretical results from the literature.

7.1 Overview

A spectral embedding is defined in terms of either the singular value decomposition or the eigenvalue decomposition of a matrix associated with the graph. For undirected graphs, these decompositions have a special relationship, described shortly, while for directed graphs the eigendecomposition will result in complex numbers so we will work with the singular value decomposition. A variety of matrices can be considered and igraph supports the adjacency matrix, the combinatorial Laplacian, and the normalized Laplacian. We will focus on the adjacency matrix and introduce the Laplacian later.

We will assume the graph has n vertices so that the adjacency matrix A is of dimension $n \times n$ with vertex set $V = \{1, 2, \ldots, n\}$. We will also assume that we seek an embedding into d dimensions where typically $d \ll n$. Methods to select the embedding dimension will be discussed later.

We have seen in Sec. 1.2.6 that igraph graphs can be treated as adjacency matrices, for queries and manipulation of the graph structure. Sometimes we explicitly want to create the adjacency matrix, because we want to use it in R functions or other software that explicitly deal with matrices. This is simply done via omitting the indices of the '`[`' operator, which is simply a shortcut for the `as_adjacency_matrix()` function:

as_adjacency_matrix()

```
7.1.1) library(igraph)
   2) make_star(5)[]
```

By default `'A_mat'` is a sparse matrix, as defined in the *Matrix* R package.

The singular value decomposition of a matrix can be written as $A = [U|\tilde{U}](S \oplus \tilde{S})[V|\tilde{V}]^T$ (Horn and Johnson, 2013), see Eq. 7.1. Here, the matrices $[U|\tilde{U}]$ and $[V|\tilde{V}]$ have orthonormal columns and the matrices U and V have dimensions $n \times d$. The matrix $S \oplus \tilde{S}$ is diagonal with decreasing nonnegative entries and the matrix S has dimensions $d \times d$. The first step in a spectral embedding is to compute the matrices U, S and V. Note that these matrices are dense, even if the graph and its corresponding adjacency matrices are sparse. Luckily, for most applications and data sets, we do not need to compute the full singular value decomposition and modern linear algebra packages operate very quickly to compute the first few largest singular values and corresponding singular vectors, especially for sparse matrices. The matrix USV^T is the best rank d approximation to A.

$$\left[\begin{array}{c|c} S & 0 \\ \hline 0 & \tilde{S} \end{array}\right] \left[\begin{array}{c} V^T \\ \hline \tilde{V}^T \end{array}\right] = USV^T + \tilde{U}\tilde{S}\tilde{V}^T \tag{7.1}$$

We can retrieve the adjacency spectral embedding for a graph with the function `embed_adjacency_matrix()`. This function requires at least two inputs, the graph and the second is the dimension of the embedding corresponding to d in our notation above. The function returns a list with three elements with names `'X'`, `'Y'`, `'D'` and `'options'`.

embed_adjacency_matrix()

Depending on the properties of the graph and the subsequent data analysis task of interest, different embeddings options may be desirable. First, we will discuss the undirected case before noting a few differences for the directed case. Finally, we will discuss working with a graph Laplacian instead of the adjacency matrix.

7.1.1 Undirected case

For an undirected graph, the matrices U and V of the singular value decomposition are equal except for sign changes in the columns corresponding to any negative eigenvalues; so we only need to consider U. Hence, we can work between the singular value and eigen decompositions by making the appropriate sign changes. Due to this symmetry, the `'Y'` output will be `'NULL'` and only the `'X'` output will be populated.

The default output for `'X'` is $\mathfrak{n} \times \mathfrak{d}$ dimensional matrix $\mathsf{US}^{1/2}$. For the undirected case the vector `'D'` is set to the corresponding eigenvalues, which can be either positive or negative (whereas the singular values are) We view this matrix as $\mathfrak{n}$ points in $\mathfrak{R}^{\mathfrak{d}}$ where each point represents a vertex in the graph. Hence through dimensionality reduction and spectral decomposition, we have transformed the graph, a combinatorial object requiring special algorithms ot perform various tasks, into a point cloud where the tools of standard multivariate analysis can be used. Before going into more details we will examine the embedding on a random graph.

7.1.1.1 2-Block Stochastic Blockmodel

In this example we will sample a graph from a 2-block stochastic blockmodel Holland et al (1983), (see also Sec. **??**) with parameters

$$\mathsf{B} = \begin{bmatrix} 0.4 & 0.2 \\ 0.2 & 0.4 \end{bmatrix} \quad \text{and} \quad \vec{\mathfrak{n}}_{vec} = [100, 100].$$

Here, there are 100 vertices in each block and the probability that two vertices are adjacent is 0.4 for vertices in the same block and 0.2 for vertices in different blocks.

After generating the graph, we embed the graph into two dimensions with the adjacency spectral embedding. Fig. 7.1 shows the resulting scatter plot. This figure illustrates how vertices in the same block cluster together in the embedding.

```
7.2.1) set.seed(42)
   2) sbm_2 <- sample_(sbm(
   3)   n = 200,
```

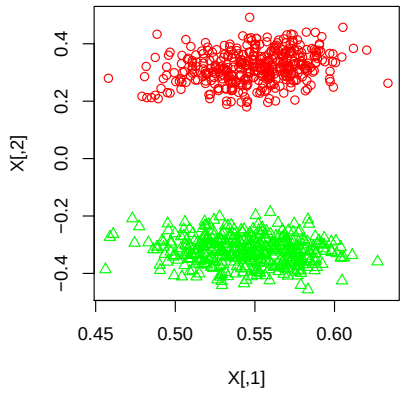


Fig. 7.1 The adjacency spectral embedding to two dimensions of a 2-block stochastic blockmodel graph. The embedding effectively seperates vertices in the two blocks into distinct clusters of points. The shapes indictate the true block labels and the colors indicate the the 2-means cluster labels.

```
4)   pref.matrix = diag(0.2, 2) + 0.2,
5)   block.sizes = c(100, 100)),
6)   vertex_attr(true_block = rep(1:2, each = 100))
7) )
8) X <- embed_adjacency_matrix(sbm_2, 2)$X
```

Simple clustering methods such as k-means, hierarchical clustering or Gaussian mixture modelling would all easily cluster the vertices into the correct clusters based on the model parameters. Indeed, running k-means and comparing to the true labels, we exactly recover the correct partition.

```
7.3.1) X_km <- kmeans(X, 2)$cluster
2) V(sbm_2)$true_block %>%
3)   as_membership() %>%
4)   compare(as_membership(X_km), method = "split.join")
5)
6) plot(X, col = c("green", "red")[X_km], pch = block)
```

By default the calculated embedding is scaled by the singular values, the `'scaled'` argument controls whether this is desired. When using subsequent techniques that are scale invariant such as certain scatter plots and Gaussian mixture modeling, it will not matter greatly whether we use the scaled or unscaled version. For other methods, such as k-means clustering, the scaling can have an impact and we will briefly compare these in the example below.

Note, that since the singular values can be retrieved from `'D'`, one can easily convert between scaled and unscaled versions.

If you examine the source code of the `embed_adjacency_matrix()` function, then you will notice, that it does not actually calculate the singular vectors and values of the adjacency matrix, but it modifies the diagonal.

The diagonal of the adjacency matrix can be viewed as missing data, and this *diagonal augmentation* is an imputation scheme that may improve performance of the embedding. By default igraph puts $d_v/(n-1)$ in the diagonal, where d_v is the degree of vertex v. The heuristic behind this default is that the probability that a vertex is adjacent to itself is assumed to be the same as the probability it is adjacent to any other vertex. Though this default has only a heuristic justification, in practice we have observed that it frequently improves performance for subsequent tasks such as clustering. To work with the original adjacency matrix, one would set `'cvec = rep(0, gorder(g))'`.

7.1.2 Interpreting the embedding

Naively, the embedding transforms the graph into a point cloud in $\mathfrak{R}^d$ that can then be treated as standard data set. Indeed, the general notion that nearby embedded points correspond to vertices with similar connectivity properties holds. Unlike some graph layout algorithms, it does not necessarily hold that nearby embedded points are likely to correspond to adjacent vertices.

To interpret the geometry of the point cloud it is valuable to consider the inner product of the embedded points. Before continuing, we will introduce the *random dot product graph* model for which the adjacency spectral embedding is particularly well suited.

random dot product graph

Definition 7.1 (RDPG, Scheinerman and Tucker (2010); Young and Scheinerman (2007)). Let $X_1, \ldots, X_n \in \mathfrak{R}^d$ be the latent positions and suppose that $0 \leq \langle X_i, X_j \rangle \leq 1$ for all i, j. Let $X = [X_1, \ldots, X_n]^\intercal \in \mathfrak{R}^{n \times d}$. We say $A \sim \text{RDPG}(X)$ if for all $i < j$, the A_{ij} are independent and $A_{ij} \sim \text{Bern}(\langle X_i, X_j \rangle)$ and $A_{ij} = A_{ji}$.

One useful interpretation of this model is found by taking a social network perspective where each vertex is a person and edges represent friendship or communication In this view, the d coordinates of each latent position may represent an individual's interest level in different topics. The magnitude of the latent position represents how talkative or friendly an individual is overall. People who are talkative and interested in the same topics are very likely to be friends while people who are not talkative or who are interested in a disjoint set of topics are unlikely to be friends.

In Sec. 7.2 we will show that if a graph has an RDPG distribution then the adjacency spectral embedding serves to estimate the latent positions. More generally, the geometry of the embedded points can be interepreted in a similar way as the interpretation of the latent positions in the embedded space. Points with high magnitude more likely correspond to vertices with a high degree and embedded points that are nearly orthogonal correspond to vertices that are unlikely to form edges with each other. The result of this is that points near the orgin correspond to low degree vertices that are unlikely to form edges with each other even though they are nearby.

Finally, it may occur that some of the dimensions of the embedded space correspond to negative eigenvalues. In this case, points that both have high magnitude in those dimensions will correspond to vertices that are actually less likely to be adjacent. In order to avoid errant interpretations, it is important to check the sign of the eigenvalues in the output `'D'`. As we go through some of the real data examples we will delve deeper into interpreting the spectral embeddings.

7.1.3 Laplacian

The Laplacian spectral embedding proceeds in exactly the same manner as the adjacency spectral embedding but operating on one of the graph Laplacian matrices. In igraph this is implemented with the function `embed_laplacian_matrix()`. First, we define the matrix D to be the diagonal matrix where D_{ii} is the degree of the i^{th} vertex.

embed_laplacian_matrix()

Three versions of the graph Laplacian are implemented and can be chosen by specifying the parameter `'type'`. The default is the combinatorial Laplacian $D - A$ which can be chosen with `'type="D-A"'`. This positive semi-definite matrix has the property that the number of eigenvalues equal to zero of $D - A$ is equal to the number of connected components in the matrix. Hence, there is always one eigenvalue equal to zero and it is easy to see that the vector of all ones is an eigenvector for the zero eigenvalue.

A well known use of the combinatorial Laplacian is to solve a convex relaxation of the ratio-cut problem which seeks a partition of the graph into two parts so that the parts have roughly the same number of vertices and there are few edges between the parts. One can find the solution to the convex relaxation of this problem by computing the eigenvector corresponding to the second smallest eigenvalue of $D - A$ and the parts are determined by the sign of components of that eigenvector Chung (1997). Using the embedding functionality, we could find the solution to this problem with

```
7.4.1) embed_laplacian_matrix(sbm_2, 2, which = "sa") %>%
   2)   extract(,2) %>%
   3)   sign()
```

The algument `'which'` specifies the eigenvalues and eigenvectors to compute. The default is `''lm''`, the eigenvalues with the largest magnitude. With the combinatorial Laplacian you should usually set `'which'` to `''sa''`, to select the smallest eigenvalues, and ignore the first dimension since it is always proportional to the vector of all ones and hence has no discriminatory signal.

The second option for the graph Laplacian is the matrix $I - D^{-1/2}AD^{-1/2}$ which is known as the normalized Laplacian and can be selected with `'type = "I-DAD"'`. Here the power in $D^{-1/2}$ is taken entrywise with $0^{-1/2}$ taken to be 0. Like the combinatorial Laplacian, the normalized Laplacian can be used to solve a relaxation of another graph cut problem called the normalized min-cut. Again, it is the eigenvector corresponding to the second smallest eigenvalue that is used to find the solution. This matrix is also positive semidefinite and always has an eigenvector with entries proportional to the square root of the degree of the associated vertices. Again, for this matrix it makes most sense to use the eigenvectors corresponding to the smallest eigenvalues with `'which = "sa"'`.

```
7.5.1) embed_laplacian_matrix(sbm_2, 2, which = "sa", type = "I-DAD") %>%
   2)   extract(,2) %>%
   3)   sign()
```

The final option is `'type = "DAD"'` which corresponds to the matrix $D^{-1/2}AD^{-1/2}$, which is sometimes called the normalized adjacency. This matrix has the same eigenvectors as the normalized Laplacian and the eigenvalues are equal to one minus the eigenvalues of the normalized Laplacian so it makes sense to work with `'which = "la"'` or `'which = "lm"'`, the default.

```
7.6.1) embed_laplacian_matrix(sbm_2, 2, type = "I-DAD") %>%
   2)   extract(,2) %>%
   3)   sign()
```

We will now investigate a real data example adapted from Sussman et al (2012a) to study how different results can be achieved with the different matrices.

7.1.3.1 Wikipedia graph

This graph consists of 1382 vertices and 18857 edges where each vertex corresponds to an article from the english language Wikiedia and edges indicate a hyperlink exists from one article to another. The original graph was collected with directed edges but we will work with the collapsed undirected version for this example. Each vertex is also given one of 5 class labels, either Category, Person, Location, Date and Math, which indicate the topic or category of the document. This graph was constructed by collecting all articles within two clicks of the article "Algebraic Geometry" and the hyperlinks between those articles.

```
7.7.1) library(igraphdata)
   2) data(algebraic)
   3) algebraic %>% summary()
```

For this example we will compare the adjacency, normalized adjacency, and combinatorial Laplacian spectral embeddings. Recall that since the normalized adjacency and normalized Laplacian have the same eigenvectors we don't need to compare them. For the adjacency and normalized adjacency we will embed to two dimensions corresponding to the largest magnitude eigenvalues. For the combinatorial Laplacian we will embed to three dimensions correspond to the smallest eigenvalues but ignore the first dimension, which is constant. Finally, for each embedding we perform a dominant sign correction as a simple way to try to align the embeddings (later we will discuss procrustes analyis).

```
7.8.1) algebraic <- algebraic %>%
   2)   as.undirected(mode = "collapse") %>%
   3)   simplify()
   4)
   5) correct_sign <- function(x) {
   6)   factor <- x %>%
   7)     sign() %>%
   8)     colSums() %>%
   9)     add(.5) %>%
  10)     sign() %>%
  11)     diag
  12)   x %*% factor
  13) }
  14)
  15) palette(c("red", "green", "black", "orange", "blue"))
  16) par(mfrow= c(1,3))
  17)
  18) ## Adjacency
  19) Xa <- algebraic %>%
  20)   embed_adjacency_matrix(2) %>%
  21)   use_series("X") %>%
  22)   correct_sign()
  23) plot(Xa, col = V(algebraic)$label + 1, pch = 4, cex = .5)
  24)
  25) ## Normalized Adjacency
  26) Xnl <- algebraic %>%
  27)   embed_laplacian_matrix(2, type = "dad") %>%
  28)   use_series("X") %>%
  29)   correct_sign()
  30) plot(Xnl, col = V(g)$label + 1,pch = 4,cex = .5)
  31)
```

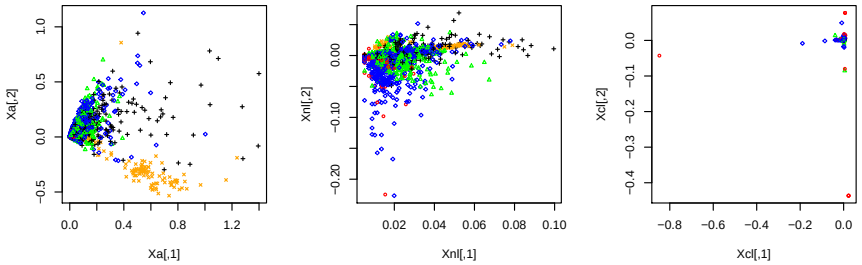


Fig. 7.2 A comparision of the adjacency (left), normalized adjacency (center) and combinatorial Laplacian (right) spectral embeddings. Each point is colored according to the class label of the vertex: red circle is Category, green triangle is Person, black cross is Location, orange x is Date and blue diamond is Math.

```
## Combinatorial Laplacian
Xcl <- algebraic %>%
  embed_laplacian_matrix(3, type = "D-A", which="sa") %>%
  use_series("X") %>%
  extract(,2:3) %>%
  correct_sign()
plot(Xcl, col = V(g)$label + 1,pch = 4, cex  =.5)
```

After each embedding, a dominant sign correction is performed as a very simple alignment step.

Fig. 7.2 shows the three 2-dimensional embeddings. One can see that the three embeddings are each quite different. First, the combinatorial Laplacian embedding is perhaps the hardest to work with because the scatter plot is dominated by a few large outliers. These outliers are red cirlces indicating they are in the Category class, which consists of Wikipedia pages linking to many different articles about the same category. It is thought by some that the combinatorial Laplacian is the least stable of the three embedding methodologies (von Luxburg, 2007). The adjacency and normalized adjacency both have somewhat more clear structure. For the adjacency, we see that one class, the orange crosses corresponding to Date articles, is well seperated from the other classes. Comparing this to the normalized adjacency we see that this is no longer the case. Sussman et al (2012b) suggest, the authors compare the clustering performance of k-means on the two embeddings.

A more thorough analysis may find that if we embed to higher dimensions than one of the embeddings would be more clearly preferred. However, this simple two dimensional example shows how the different embeddings may capture different components of the graph structure and could be useful for different purposes.

Currently, there are no strong theoretical or empirical results that suggest that one of these embeddings should necessarily be the default option. As with many statistical techniques, none of the embedding procedures are known to be always better for certain tasks. Sarkar and Bickel (2013) provides an asymptotic analysis of the role of normalization for the two-block stochastic blockmodel. When followed by k-means, both the adjacency and normalized adjacency embeddings are known to consistently estimate the true partition of the vertices under the general stochastic blockmodel. Other versions of the graph Laplacian have been proposed that regularize the embedding for the purpose of dealing with sparsity of the graph (Amini et al, 2013; Chaudhuri et al, 2012).

7.1.4 Directed case

For the directed case there are a few key differences to be aware of. Unlike in the undirected case, the adjacency matrix need not be symetric so there is no explicit relationship between the left singular vectors U and the right singular vectors V. Hence, for directed graphs the spectral embedding functions will populate both X and Y. For directed graphs the A_{ij} entry of the adjacency matrix is unity if there is an edge from vertex i to vertex j. In this case, the left singular vectors, U are representative of the edge-sending behavior of the vertices while the right singular vectors V and representative of the edge-receiving behavior of the vertices. As in the direced case, the option `'scale'` determines the values of X and Y, with $X = US^{1/2}$ and $Y = VS^{1/2}$ if `'TRUE'` (default) and with $X = U$ and $Y = V$ if `'FALSE'`.

For the Laplacian spectral embedding with a directed graph, we must construct an appropriate directed version of the Laplacian. Since each vertex has both an in-degree and an out-degree we cannot work easily with the combinatorial Laplacian but we can construct a version of the normalized Laplacian. This is given by $O^{-1/2}AP^{-1/2}$ where O is the diagonal matrix of out degrees and P is the diagonal matrix of in degrees (Rohe and Yu, 2012).

Since both X and Y are populated we have more flexibility with how to assess and analyze the embedding. For example, to cluster vertices based only on their "sending" behavior alone we can cluster just X whereas to cluster them based on their "receiving" behavior we can clsuter using just Y. Rohe and Yu (2012) suggest clustering each separately and then using the two clusterings to obtain a joint description of vertex behavior as a directed stochastic blockmodel. Sussman et al (2012b) suggest concatentating X and Y as $Z = [X|Y]$ and clustering the vertices in $\mathfrak{R}^{2d}$. Alternatively, we could concatenate the X and Y vertically and cluster all $2n$ points in $\mathfrak{R}^{d}$ as suggested for bipartite graphs in Dhillon (2001).

These are just a few of the options when working with directed graphs. We will now investigate spectral embeddings for the directed case for a political blogs network.

7.1.4.1 Political Blogs

The political blogosphere network data (Adamic and Glance, 2005) was collected during 2004 election United States presidential election. Each vertex in the graph represents a blog focused on politics and the directed edges represent links from one blog to another blog. Each vertex has also been assigned a political leaning, either left, for the US Democrat party, or right, for the US Republican party.

```
7.9.1) data(polblogs)
   2) polblogs %>% summary()
```

For this example we will use only the largest strongly connected component.

```
7.10.1) largest_strong_component <- function(graph) {
   2)   co <- components(graph, mode = "strong")
   3)   which.max(co$csize) %>%
   4)     equals(co$membership) %>%
   5)     induced_subgraph(graph = graph)
   6) }
   7) blog_core <- largest_strong_component(polblogs)
   8) blog_core %>% summary()
```

It consists of 793 vertices and 15841 edges. This graph has a particularly nice structure and before the embedding, we discuss the special aspects of the directed nature of this graph. First, we will investigate dimension selection, the degree corrected SBM and projecting onto the sphere.

To investigate what an appropriate dimension might be to perform the embedding we will first embed to dimension 100 and plot the singular values in Fig. 7.3.

```
7.11.1) blog_100 <- embed_adjacency_matrix(blog_core, 100)
   2) plot(blog_100$D, xlab = "Index", ylab = "Singular value")
   3) dim_select(ase$D)
```

As with principal component analysis, one way to select the dimension is to find a large gap between the singular value for that dimension and the next dimension. From Fig. 7.3 we see that there is certainly a large gap between the second and third singular values and a smaller gap between fifth and sixth singular values, with all the other gaps between relatively small. Furthermore, the automatic dimension selection function `dim_select()` also returns two as the embedding dimenions.

dim_select()

Since this graph is directed we can plot both X and Y. The left panel of Fig. 7.4 shows the embedding of the left singular vectors and the right panel

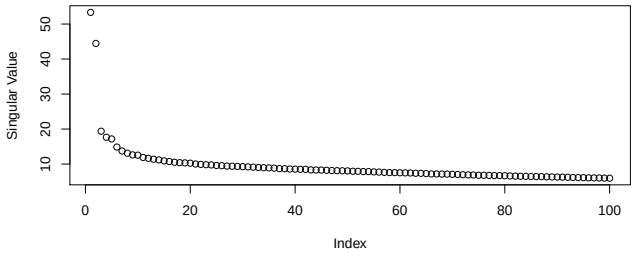


Fig. 7.3 The largest 100 singular values for the adjacency matrix of the Political Blogs Network. There is a large gap between the second and third singular values indicating this would be a good choice for an embedding dimension.

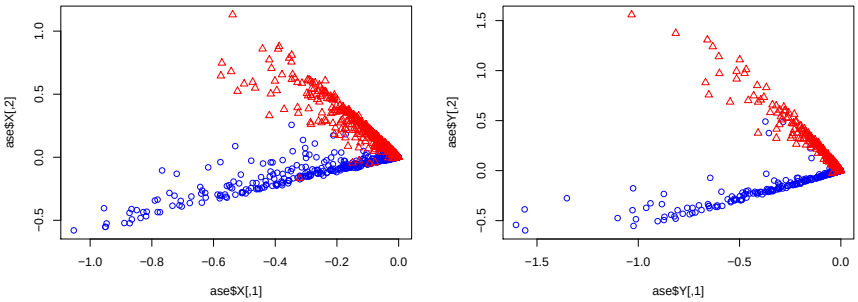


Fig. 7.4 The embedding to dimension two of the Political Blogs Network. The left panel corresponds to X, the left singular vectors representing "sending" behavior of the vertices, and the right panel corresponds to Y, the right singular vectors representing "receiving" behavior of the vertices. Each point is colored according to the political leaning of the blog, red triangles for Republican and blue circles for Democrat.

shows the right singular vectors. The structure of the left and right singular vectors are very similar with the embedded points being concentrated around two distinct rays.

```
7.12.1) par(mfrow=c(1, 2))
    2) palette(c("blue", "red"))
    3) plot(ase$X, col = V(blog_core)$LR)
    4) plot(ase$Y, col = V(blog_core)$LR)
```

This graph was studied in Karrer and Newman (2011) as part of their exposition of the degree corrected stochastic block model, a model where in addition to block membership each vertex has a scalar degree correction fac-

tor which scales the probability that a vertex forms edges with other vertices. Just as the embedding of a stochastic blockmodel graph will be concentrated around distinct points for each block, for the degree correctd stochastic blockmodel the embedding will be concentrated around distinct rays for each block. This means that the overall magnitude of the embedded points is not as important for determining block membership as the direction of the points.

So, to cluster the vertices in a degree-corrected stochastic blockmodel one should first project the embedded points onto the sphere (Qin and Rohe, 2013; Lyzinski et al, 2013). In the case of a two-dimensional embedding this is equivalent to calculating the angle of the embedded points. To illustrate the advantage of first projecting onto the sphere, we first compute the angles for the embedded points in X and Y.

```
7.13.1) theta <- cbind(atan((blog_100$X[,2]) / (blog_100$X[,1])),
    2)                 atan((blog_100$Y[,2]) / (blog_100$Y[,1])))
    3) plot(theta,col=V(g)$LR,pch=V(g)$LR)
```

Clustering the angles from X alone yields an error of 4.3% as compared to clustering directly on X which yields an error of 30.9%. This uses only the "sending" portion of the embeddings and if we use both sending and receiving we achieve an error of 2.9% by concatentating the angles into $\mathfrak{R}^2$ while concatenating the embedded points into $\mathfrak{R}^4$ yields an error 38.6%.

```
7.14.1) error <- function(x) {
    2)   k <- x %>%
    3)     kmeans(2) %>%
    4)     use_series("cluster")
    5)   min(mean(k == V(g)$LR), mean(k != V(g)$LR))
    6) }
    7) theta[,1] %>% error()
    8) theta %>% error()
    9) cbind(blog_100$X, blog_100$Y) %>% error()
```

We can conclude that the angular components of the embedding are leading to the most accurate clusterings, at least using k-means and the magnitude is confounding the true clustering. Additionally, the improved performance using both embedding angles suggests that it may be best to work in the joint space over working only with one embedding (the error rate clustering the angles on Y alone is 3.3%). The idea of projecting onto the sphere before clustering is similar to ideas from subspace clustering (Vidal, 2010).

Fig. 7.5 shows a scatter plot of the two angles for each pair of embedded points. We see that the two classes cluster closely around two points and that the angles are fairly well correlated (correlation$= 0.938$) suggesting that most vertices have similar sending a receiving tendencies, at least after correcting for degree.

Rohe and Yu (2012) suggests that sometimes we will be interested in vertices that behave differently as "senders" than as "receivers". To do this we

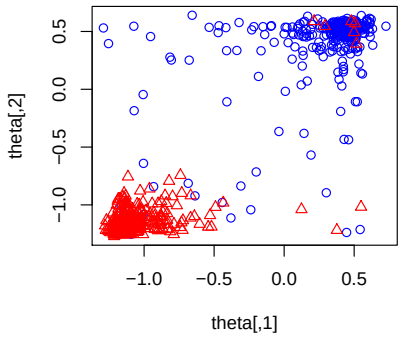


Fig. 7.5 This shows the angles of the embedded points for the left singular vectors X, representing "sending" behavior, on the horizontal axis and for the right singular vectors Y, representing "receiving" behavior, on the vertical axis. Each point is colored according to the political leaning of the blog, red triangles for Republican and blue circles for Democrat.

looked for vertices with a large positve angle for X, indicative of Democratic "sending" behavior, and a large negative angle for Y indicative of Republican "receiving" behaviors, by ordering the vertices according to the difference between the angles. We list the names of the first six vertices according to this ordering.

```
7.15.1) lr_order <- order( -theta[,1] -theta[,2])
    2) lr_top <- V(blog_core)[lr_order][1:6]
    3) degree(blog_core, v = lr_top)
```

We find that most of these are low degree vertices, which is logical because just a small change in the edges of low degree vertices can greatly change their angle in the embedded space, but one of the vertices was drudgereport.com, a new aggregator website that is often thought to be quite conservative. This indicates that even though this vertex receives links mostly from conservative blogs, many of the links it sends are to democratic blogs.

Finally, much of this analysis could have been done working with the degrees or the edge counts to Democratic versus Republican blogs. However, if the class labels were not known, then the spectral embedding provides a quick visualization to verify that there are two clearly distinct groups and a way to analyze the sending and receiving behavior seperately or together.

7.1.5 Weighted case

As usual in igraph, if the graph has the edge attribute 'weight' then the above methods operate on the weight matrix W where $W_{ij} = 0$ if there is no edge from vertex `i` to vertex `j` and otherwise W_{ij} equals the weight of that edge. The weighted Laplacian matrices are constructed in the same way except the vertex degree is replaced by the graph strength, the sum of the weights of edges incident to the vertex. Whether a graph has the weight edge attribute or not, you can set the optional argument `'weight'` to a vector equal to the size of the graph.

The weighted version of spectral embeddings can be interpretted in basically the same way, however the weights add a confounding factor. If the distribution of weights is heavy tailed then the large weights can heavily skew the embedding. Embedded points may have a large magnitude because of a few large weights rather than a high degree and visa versa a point may have a small magnitude if it is very weakly adjacent to many vertices rather than having a low degree. In the next example we briefly investigate the effects of weightings on a graph derived from brain imaging data.

7.1.5.1 Diffusion Tensor MRI

Diffusion tensor MRI is a technique in magnetic resonance imaging that captures the large-scale structural connectivity of the brain (Gray et al, 2012). The raw image data can be processed so that each voxel is associated with a direction that can then be traced through the volume. These traces can then be used to construct a graph. The weighted graph is then formed by contracting the voxel-level graph to 70 disitinct rergions which are given by the Desikan brain atlas.

The end result is an undirected graph with 70 vertices and 990 edges with a total graph strength of over five million.

```
7.16.1) data(mri)
    2) mri %>% summary()
    3) mri %>% strength() %>% sum()
    4) par(mfrow = c(1, 2))
    5) E(g)$weight %>% density() %>% plot()
    6) E(g)$weight %>% log() %>% density() %>% plot()
```

In the left panel of Fig. 7.6, we show a kernel density estimate of the edge weights for all the vertices. Clearly this distribution is very heavy tailed with weights ranging from one to 30469. We also plot the distribution of the log of the weights in the right panel of Fig. 7.6 which does not exhibit the heavy tailed nature of original weights.

```
7.17.1) X <- mri %>%
    2)   embed_adjacency_matrix(2) %>%
```

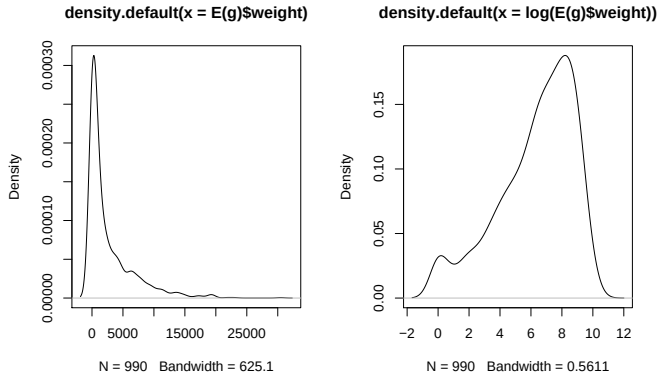


Fig. 7.6 A kernel density estimate of the edge weights for the 990 edges in the DTRMI graph.

```
  use_series("X") %>%
  correct_sign()
Xlog <- mri %>%
  embed_adjacency_matrix(2, weights = log(E(g)$weight)) %>%
  use_series("X") %>%
  correct_sign()
Xuw <- mri %>%
  embed_adjacency_matrix(2, weights = NA) %>%
  use_series("X") %>%
  correct_sign()
```

We investigate the impact of using weights in this context by comparing the two dimensional adjacency spectral embeddings using the original weights, the logarithm of the weights as well as an unweighted version by setting the weights to one. The vertices in the graph are attributed as belonging to either the left or right hemisphere and within each hemishphere they belong to one of 35 regions which each are matched across hemispheres. A simple task is to recover the left-right division based on the graph alone. The left hemisphere vertices are shown as triangles and the right hemisphere vertices are shown as circles.

Before plotting the embeddings we try to align them by doing a simple sign correction. Fig. 7.7 shows the three embeddings and we see that for the log-weights (center) and unweighted embedding (right) the two hemispheres are linearly separable and can be relatively easily clustered. For the original weights (left), the two hemisphers are somewhat more mixed up and exhibit more of a degree-corrected behavior like the political blogs in Example 7.1.4.1. Overall, the logarithmic transformation is strong enough to make it largely similar to the unweighted version.

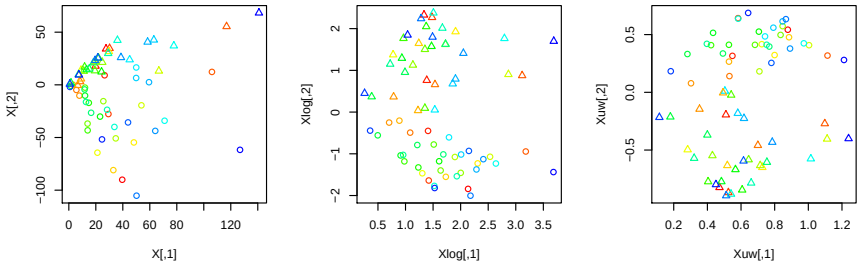


Fig. 7.7 The adjacency spectral embedding for the DTMRI graph of Example 7.1.5.1 based on three different weighting. The left panel uses the original weights, the center panel uses the log of the original weights and the right panel uses weights all equal to one. The points are colored according to the region names which are matched across hemispheres and the shapes represent the two hemispheres, circle for right hemisphere and triangle for right hemisphere.

```
7.18.1) par(mfrow=c(1,3))
    2) plot(X, col = V(g)$region.num, pch = (V(g)$lr == "r") + 1)
    3) plot(Xlog, col = V(g)$region.num, pch = (V(g)$lr == "r") + 1)
    4) plot(Xuw, col = V(g)$region.num, pch = (V(g)$lr == "r") + 1)
```

As we have seen, depending on how reliable or noisy the weights are, different weighting schemes will be advantagous. As with other options it will also depend on the subsequent analysis that will be performed on the embeddings. Overall, for heavy tailed weight distributions simple transformations or unweighting will often improve the spectral embedding properties.

7.2 Theory

Strong thoeretical results have been shown for these embeddings, which can be viewed as the graph version of principal component analysis. For various random graph models an assortment of probabilistic guarantees and asymptotic distributional results for spectral embeddings have been demonstrated. To keep things simple we will focus on the adjacency spectral embedding and the undirected random dot product graph model. This pair of embedding and model are particularly suited for each other because of the simple analysis of the singular value decomposition for a matrix of pairwise dot products, known as a gram matrix. We leave out results for the Laplacian matrices but we will refer the reader to the relevant literature for these results. In general, which particular matrix is best for the problem at hand will depend

on various factors. Currently there is limited research in comparing various matrices though this is an area of continued interest.

When working with the adjacency spectral embedding from a theoretical perspective, a highly tractable model is the random dot product graph (RDPG) model. We introduced the RDPG model in § 7.1.2 in the case of fixed latent positions. Presently we change things slightly to introduce the case with iid latent positions.

Definition 7.2 (iid RDPG, Scheinerman and Tucker (2010); Young and Scheinerman (2007)). We will write $A \sim \text{RDPG}(n, F)$ if $X_1, \dots, X_n \overset{iid}{\sim} F$ for some distribution F and conditioned on the X_is, $A \sim \text{RDPG}(X)$ where $X = [X_1, \dots, X_n]^T$. Note that we must enforce that F is supported on a set $\mathcal{X}$ satisfying $0 \leq \langle x, y \rangle \leq 1$ for all $x, y \in \mathcal{X}$.

Certain other random graph models can be reparametrized as an RDPG and it will be convenient to consider this reparameterization. We now define stochastic blockmodel in terms of the RDPG.

Definition 7.3 (SBM, Holland et al (1983)). We say an RDPG is a K-block SBM if there are only K distinct latent positions, or if the support of the distribution F has cardinality K. In either case, suppose $\nu_1, \dots, \nu_K$ are the distinct latent positions. The $K \times K$ matrix B where $B_{ij} = \langle \nu_i, \nu_j \rangle$ provides the matrix probabilities for edges between each block in the more traditional parameterization. The block membership vector $\tau \in [K]^n$ is defined so that $\tau_i = k$ if $X_i = \nu_k$. The block sizes are given by $\vec{n}$ where $\vec{n}_k$ is the number of vertices in block k.

Note that not all stochastic block models can be parametrized in this way. If an SBM is a d-dimensional RDPG, then we must have that B is positive semidefinite and has rank d. If B is not positive semidefinite then there is still a representation where the blocks are represented by K points in $\mathfrak{R}^{\text{rank}B}$ but rather than edge probabilities being given by the dot product of these positions, the probabilities will be given by a bilinear form. The theory herein focuses on the case the proper RDPG case but can be extended to the bilinear form setting.

7.2.1 Latent position estimation

One issue that must be confronted when studying the RDPG model is the *non-identifiability* of the latent positions or their distribution. This non-identifiability is due to the fact that the inner product that determines the edge presence probabilities is invariant to orthogonal transformations of its arguments. Practically, this non-identifiability is of minimal impact as

many statistical procedures such as Gaussian mixture modelling, k-nearest-neighbors, etc., will be invariant to orthogonal transformtions of the embedding. However, when considering the estimation of the latent positions we must contend with this issue.

Our assumptions on the distribution F in Theorems 7.1 and 7.2 below ensure that the distribution F is identifiable at least up to sign changes.

Theorem 7.1 (Sussman et al (2013); Lyzinski et al (2013); Tang et al (2014); Sussman (2014)). *Suppose* $A \sim \mathrm{RDPG}(n, F)$ *with iid latent positions given by* $X = [X_1, \ldots, X_n]^T \in \mathfrak{R}^{n \times d}$. *Suppose the distribution* F *is such that* $\mathbb{E}[X_i X_i^T]$ *is diagonal with distinct entries along the diagonal. Let* $\hat{X} \in \mathfrak{R}^{n \times d}$ *be the adjacency spectral embedding of* A *with the sign of each column chosen as to best match* X *in terms of square error.*

It holds that

$$\|\hat{X} - X\|_{2 \to \infty} = \max_i \|\hat{X}_i = X_i\|_2 = O_P(\sqrt{\log(n)/n}) \tag{7.2}$$

and

$$\|\hat{X} - X\|_F = O_P(1). \tag{7.3}$$

where O_P *denotes that there are constants, depending only on the eigenvalues of the second moment matrix for* F*, such that the bounds hold up to the constants with probability tending to one as* n *goes to* ∞.

The main impact of this theorem are that it is possible to consistently estimate the latent positions. Eq. 7.2 in particular ensures that all of the latent positions can be estimated simultaneously at a rate not dramatically worse than the typical parameter estimation accuracy of $1/\sqrt{n}$.

Theorem 7.2 (Athreya et al (2013)). *For each* $n \in \mathbb{N}$*, let* $A^{(n)} \sim \mathrm{RDPG}(\mathcal{X}, F)$ *with iid latent positions* $X_1^{(n)}, \ldots, X_n^{(n)} \overset{iid}{\sim} F$.[1] *Suppose the distribution* F *satisfies the same assumptions as in Theorem 7.1 and denote the second moment matrix by* $\mu_2 \in \mathfrak{R}^{d \times d}$. *Let* $\hat{X}^{(n)} = [\hat{X}_1^{(n)}, \ldots, \hat{X}_n^{(n)}]^T \in \mathfrak{R}^{n \times d}$ *be the adjacency spectral embedding of* $A^{(n)}$ *with the sign of each column chosen so that* $\max_i \hat{X}_{ij} > -\min_{i'} \hat{X}_{i'j}$ *for each* $j \in [d]$.

Letting $X^* \sim F$ *and suppressing the superscript* (n) *notation we have*

$$\sqrt{n}\left(\hat{X}_{i\cdot} - X_{i\cdot}\right) \xrightarrow{\mathcal{L}} \int_{\mathcal{X}} \mathcal{N}(0, \mu_2^{-1}\mathbb{E}[x^T X^*(1 - x^T X^*)X^* X^{*T}]\mu_2^{-1}) F(dx).$$

The integral denotes a F*-weighted mixture over normal distributions each with mean zero and with variance depending on the mixture component.*

[1] We assume the latent positions are i.i.d. for fixed n but we make no assumptions on the relationship across n.

Chapter 8
Change-point detection in temporal graphs

8.1 Introduction

In this chapter we deal with temporal graphs, graphs that change their structure over time by gaining and/or losing new edges and/or vertices. It is clear that most real graphs belong to this category, and once we are able to collect data for them from multiple time steps, we can make use of the fact that the graphs are related and they (partially) share the same vertex set.

A very natural problem in temporal graphs is the detection of change points, a point of time, when the structure of the graph changes qualitatively. In this chapter we show a method, called scan statistics, for detecting unusually dense regions in a temporal graph. These regions often signify anomalous activity in the modeled system.

Scan statistics (Wang et al (2014)) are commonly used in signal processing to detect a local signal in an instantiation of some random field. The idea is to scan over a small time or spatial window of the data and calculate some locality statistic for each window. The maximum of these locality statistics is known as the scan statistic. Large values of the scan statistic suggest existence of nonhomogeneity, a local region in the graph with excessive communications.

This chapter is structured as follows. First we generate a time series of random graphs, generated from stochastic block models, and use this to motivate the scan statistics change point detection method. Then we show two applications, the Enron email corpus, and the Bitcoin transaction network.

Some notation that we will use extensively in this chapter. For any $u, v \in V(G)$, we write $u \sim v$ if there exists an edge between u and v in G. For $v \in V$, we denote by $N_k[v; G]$ the set of vertices u at distance at most k from v, i.e., $N_k[v; G] = \{u \in V \colon d(u, v) \leq k\}$. For $V' \subset V$, $\Omega(V', G)$ is the subgraph of G induced by V'. Thus, $\Omega(N_k[v; G], G)$ is the subgraph of G induced by vertices at distance at most k from v.

8.2 A toy example: changing SBM

We create an artificial data set, a time series of 20 graphs, each sampled from a stochastic block model (see Sec. **??**. The block membership of the vertices is fixed over time while the connectivity probability matrix $\mathbf{P} = \mathbf{P}^t$ changes in the last time step, i.e. $\mathbf{P}^t = P^0$ for $t = 1, \dots, 19$, and $\mathbf{P}^{20} = P^A$, where $P^0 \neq P^A$:

$$\mathbf{P}^0 = \begin{bmatrix} p & p & p \\ p & h & p \\ p & p & p \end{bmatrix}, \qquad \mathbf{P}^A = \begin{bmatrix} p & p & p \\ p & h & p \\ p & p & q \end{bmatrix}, \tag{8.1}$$

the blocks contain $[n_1, n_2, n_3] = [16, 5, 5]$ vertics and $[p, h, q] = [0.2, 0.5, 0.8]$.

8.1.1)
```
library(igraph)
```

```
Loading required package: methods

Attaching package: 'igraph'

The following objects are masked from 'package:stats':

    decompose, spectrum

The following object is masked from 'package:base':

    union
```

13)
```
library(magrittr)
```

```

Attaching package: 'magrittr'

The following object is masked from 'package:igraph':

    %>%
```

20)
```
num_t <- 20
block_sizes <- c(10, 5, 5)
p_ij <- list(p = 0.1, h = 0.9, q = 0.9)

P0 <- matrix(p_ij$p, 3, 3)
P0[2, 2] <- p_ij$h
PA <- P0
PA[3, 3] <- p_ij$q
num_v <- sum(block_sizes)

tsg <- replicate(num_t - 1, P0, simplify = FALSE) %>% append(list(PA)) %>%
```

```
  lapply(sample_sbm, n = num_v, block.sizes = block_sizes,
         directed = TRUE) %>%
  lapply(set_vertex_attr, "name", value = LETTERS[seq_len(num_t)]) %>%
  lapply(set_vertex_attr, "label", value = LETTERS[seq_len(num_t)]) %>%
  lapply(set_vertex_attr, "color", value = rep(1:3, block_sizes)) %>%
  lapply(set_edge_attr, "arrow.size", value = 0.3)

tsg <- lapply(tsg, function(x) {
  E(x)$color <- "grey"
  E(x)[16:20 %--% 16:20]$color <- "red"
  x
})

tsg <- lapply(tsg, set_graph_attr, name = "layout",
              value = layout_with_fr(tsg[[num_t]]))
```

Let's plot in Fig. 8.1 the graphs at most recent four timestamps of `tsg` and peek the existence of anomolous subgroup at change-point, that is, $t = 20$ whose excessive communication is new at the change-point.

```
par(mfrow = c(2,2))
tsg %>%
  tail(n = 4) %>%
  lapply(plot)
```

8.3 Two locality statistics

In this section we define two different but related locality statistics on the temporal $\{G_t\}$ graph.

Firstly, for a given t, let $\Psi_{t;k}(v)$ be defined for all $k \geq 1$ and $v \in V$ by

$$\Psi_{t;k}(v) = |E(\Omega(N_k(v; G_t); G_t))|. \tag{8.2}$$

$\Psi_{t;k}(v)$ counts the number of edges in the subgraph of G_t induced by $N_k(v; G_t)$, the set of vertices u at a distance at most k from v in G_t. In a slight abuse of notation, we let $\Psi_{t;0}(v)$ denote the degree of v in G_t. A simple toy example to illustrate calculations of $\Psi_{t;k}(a)$ with varing $k = 0, 1, 2, 3$, on an undirected graph is presented in Fig. 8.2. $\Psi_{t;k}(v)$ is nicknamed the *us* statistics.

To calculate $\{\Psi_{t;k}(v)\}_{v=1}^{|V|}$ over all vertices in our last graph in the time series, which is a necessary intermediate step of computing scan statistics introduced below, you can use the `local_scan()` igraph function: *local_scan()*

```
local_scan(tsg[[num_t]], k = 1, mode = 'all')
```

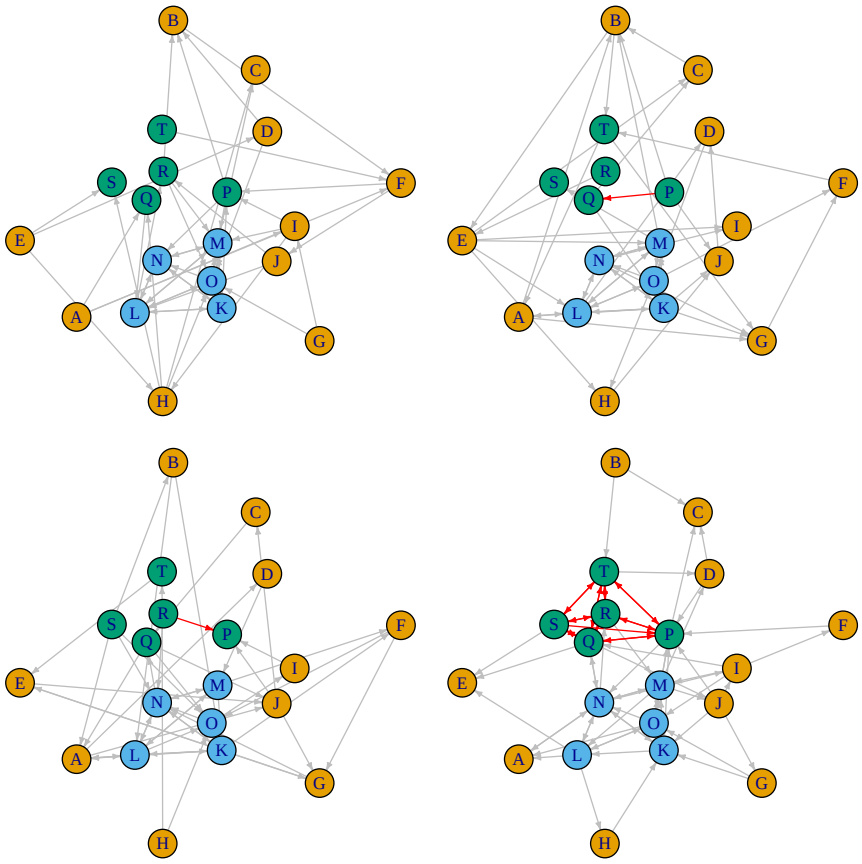


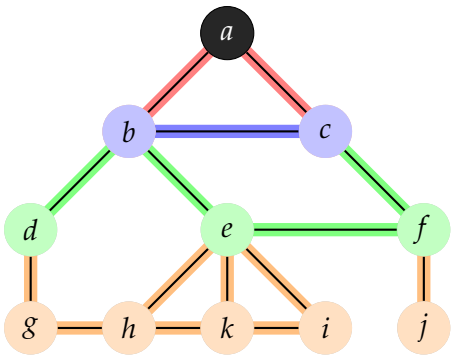
Fig. 8.1 Last four graphs from the time series of 20 SBM graphs, with a change point in the last one at $t = 20$. V is partitioned into three blocks with 10, 5 and 5 vertices each, vertices are colored according to their block membership. One block is a lot more dense in the last time step than before.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
9	2	4	7	5	2	6	3	14	12	29	24	26	39	30	39	27	26	23	22

Secondly, we introduce the other locality statistics $\Phi_{t,t';k}(v)$ (nicknamed the *them* statistics) based on two graphs G_t and G'_t. Let t and t' be given, with $t' \leq t$. Now define $\Phi_{t,t';k}(v)$ for all $k \geq 1$ and $v \in V$ by

$$\Phi_{t,t';k}(v) = |E(\Omega(N_k(v; G_t); G_{t'}))|. \tag{8.3}$$

The statistic $\Phi_{t,t';k}(v)$ counts the number of edges in the subgraph of $G_{t'}$ induced by $N_k(v; G_t)$. Once again, with a slight abuse of notation, we let



k	$E(\Omega(N_k[\mathbf{a}; G]))$	$\Psi_k(\mathbf{a})$
0	⊖	2
1	⊖ + ⊖	3
2	⊖ + ⊖ + ⊖	7
3	⊖ + ⊖ + ⊖ + ⊖	15

	d	Ψ_1	Ψ_2	Ψ_3
a	2	3	7	15
b	4	5	14	15
c	3	4	8	15
d	2	2	8	14
e	5	7	15	15
f	3	3	12	15
g	2	2	7	14
h	3	4	8	15
i	2	3	7	15
j	1	1	3	12
k	3	5	10	15

Fig. 8.2 A toy example to illustrate calculations of $\Psi_{t;k}(a)$ with various $k = 0, 1, 2, 3$, on an undirected graph. For simplicity, in this example, t is removed from all notations.

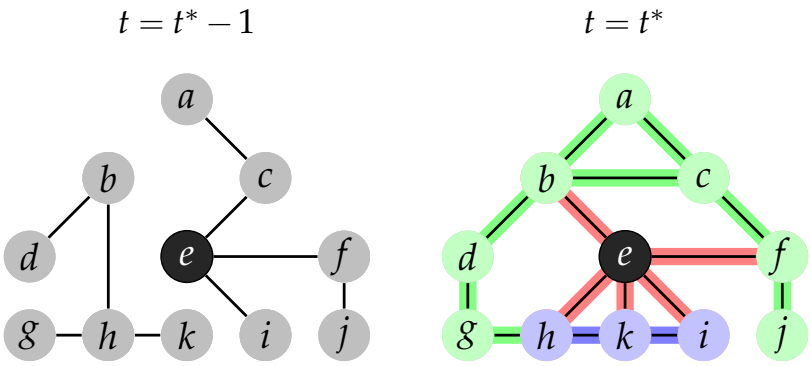
$\Phi_{t,t';0}(v)$ denote the degree of v in $G_t \cap G_{t'}$, where $G \cap G'$ denotes the graph $(V(G), E(G) \cap E(G'))$. The statistic $\Phi_{t,t';k}(v)$ uses the graph structure at time t in its computation of the locality statistic at time $t' \leq t$. Through this measure, a local density shift of v can be captured even when the connectivity level of v remains unchanged across time, i.e., when the Ψ_t stays mostly constant as t changes in some interval: the density around the vertex does not change, but the vertices that participate in it, do. With the purpose of determining whether t is a change-point, two kinds of normalizations based on past Ψ and Φ locality statistics and their corresponding normalized scan statistics are introduced in the next section. Fig. 8.3 illustrates the differences between our two locality statistics.

Given t, t', k, if we want to calculate $\{\Phi_{t,t';k}\}_{v=1}^{|V|}$ over all vertices in a graph, which is a necessary intermediate step of computing scan statistics introduced below in § 8.4, `local_scan()` function in `igraph` also implements these calculations and outputs an $|V|$-dimensional vector representing $\{\Phi_{t,t';k}\}(v)$ on each vertex v. In our previous setting $t = 20$, $t' = 19$, $k = 1$, the all 20 locality statistics $\{\Phi_{t,t';k}(v)\}_{v='A'}^{'T'}$ are

8.6.1)
```
local_scan(tsg[[20]], tsg[[19]], k = 1, mode = 'all')
```

```
 A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T
 6  0  0  1  0  1  2  2  9  4 18 18 17 24 20 16  6  6  5  2
```

Note that this time we specified two graphs, G_t and $G_{t'}$.



	$\Psi_{t^*-1;k}(e)$	$\Psi_{t^*;k}$ (e)	$\Phi_{t^*,t^*-1;k}(e)$
$k = 0$	3	5	2
$k = 1$	3	7	4
$k = 2$	5	5	9

Fig. 8.3 An example to differentiate the calculation of statistics ($\Psi_{t;k}$, $\Psi_{t';k}$ and $\Phi_{t,t';k}$) with varing orders ($k = 0, k = 1$ or $k = 2$). In the right graph G_{t^*}, note that the red edges are $E(\Omega(N_{k=0}[e; G_{t^*}], G_{t^*}))$; the red and blue edges are $E(\Omega(N_{k=1}[e; G_{t^*}], G_{t^*}))$; the red, blue and green edges are $E(\Omega(N_{k=2}[e; G_{t^*}], G_{t^*}))$. For instance, the magenta-marked number 3 is $\Psi_{t^*-1;0}$ where $\Psi_{t^*-1;0}(e) = |E(\Omega(N_0(e; G_{t^*-1}); G_{t^*-1}))|$ and $E(\Omega(N_0(e; G_{t^*-1}); G_{t^*-1})) = \{e \sim c, e \sim f, e \sim i\}$ in G_{t^*-1}; the orange-marked number 4 is $\Phi_{t^*,t^*-1;1}(e)$ where $\Phi_{t^*,t^*-1;1}(e) = |E(\Omega(N_1(e; G_{t^*}); G_{t^*-1}))|$ and $E(\Omega(N_1(e; G_{t^*}); G_{t^*-1})) = \{h \sim k, b \sim h, e \sim i, e \sim f\}$ in G_{t^*-1}.

8.4 Temporally normalized scan statistics

We define now our final test statistics, based on Ψ (or Φ), $S_{\tau,\ell,k}(t;\Psi)$ (or $S_{\tau,\ell,k}(t;\Phi)$). In practice, the final test statistic measures the presence of a dense community. To detect changes in the time series of graphs, one needs to calculate this measure in each time step. An unusually large value of the measure implies the emergence of an anomalous community in the graph.

Let $J_{t,t';k}$ be either the locality statistic $\Psi_{t,t';k}$ or $\Phi_{t,t';k}$. For a given integer $\tau \geq 0$ and $v \in V$, we define the vertex-dependent normalization $\widetilde{J}_{t,\tau;k}(v)$ of $J_{t,t';k}(v)$ by

$$\widetilde{J}_{t;\tau,k}(v) = \begin{cases} J_{t,t;k}(v) & \tau = 0 \\ J_{t,t;k}(v) - \widehat{\mu}_{t;\tau,k}(v) & \tau = 1 \\ (J_{t,t;k}(v) - \widehat{\mu}_{t;\tau,k}(v))/\widehat{\sigma}_{t;\tau,k} & \tau > 1 \end{cases}, \tag{8.4}$$

where $\widehat{\mu}_{t;\tau,k}$ is the mean activity around vertex v over the previous τ time steps, and $\widehat{\sigma}_{t;\tau,k}$ is the sample standard deviation:

$$\widehat{\mu}_{t;\tau,k}(v) = \frac{1}{\tau}\sum_{s=1}^{\tau} J_{t,t-s;k}(v), \tag{8.5}$$

$$\widehat{\sigma}_{t;\tau,k}(v) = \sqrt{\frac{1}{\tau-1}\sum_{s=1}^{\tau}(J_{t,t-s;k}(v) - \widehat{\mu}_{t;\tau,k}(v))^2}. \tag{8.6}$$

We then consider the maximum of these vertex-dependent normalizations for all $v \in V$, i.e., we define a $M_{\tau,k}(t)$ by

$$M_{\tau,k}(t) = \max_{v} \widetilde{J}_{t,\tau;k}(v). \tag{8.7}$$

We shall refer to $M_{\tau,0}(t)$ as the standardized max-degree and to $M_{\tau,1}$ as the standardized scan statistics. From Eq. (8.4), we see that the motivation behind vertex-dependent normalization is to standardize the scales of the raw locality statistics $J_{t,t';k}(v)$. Otherwise, in Eq. (8.7), a noiseless vertex in the past who has dramatically increasing communications at the current time would be inconspicuous because there might exist a talkative vertex who keeps an even higher but unchanged communication level throughout time. Finally, for a given integer $l \geq 0$, we define the temporal normalization of $M_{\tau,k}(t)$ by

$$S_{\tau,\ell,k}(t) = \begin{cases} M_{\tau,k}(t) & \ell = 0 \\ M_{\tau,k}(t) - \tilde{\mu}_{\tau,\ell,k}(t) & \ell = 1 \\ (M_{\tau,k}(t) - \tilde{\mu}_{\tau,\ell,k}(t))/\tilde{\sigma}_{\tau,\ell,k}(t) & \ell > 1 \end{cases}, \tag{8.8}$$

where $\tilde{\mu}_{\tau,\ell,k}$ and $\tilde{\sigma}_{\tau,\ell,k}$ are defined as

$$\tilde{\mu}_{\tau,\ell,k}(t) = \frac{1}{\ell}\sum_{s=1}^{\ell} M_{\tau,k}(t-s), \tag{8.9}$$

$$\tilde{\sigma}_{\tau,\ell,k}(t) = \sqrt{\frac{1}{\ell-1}\sum_{s=1}^{\ell}(M_{\tau,k}(t-s) - \tilde{\mu}_{\tau,\ell,k}(t))^2}. \tag{8.10}$$

The motivation behind temporal normalization, based on recent ℓ time steps, is to perform smoothing for the statistics $M_{\tau,k}$. This is similar to how smoothing is performed in time series analysis. We denote by $S_{\tau,\ell,k}(t;\Psi)$ and $S_{\tau,\ell,k}(t;\Phi)$ the $S_{\tau,\ell,k}(t)$ when the underlying statistic $J_{t,t';k}$ is $\Psi_{t';k}$ and $\Phi_{t,t';k}$, respectively.

In the following, we return to our toy time series, and try to find the artificial anomaly in time step 20. The function `scan_stat()` uses `local_` *scan_stat()*

`scan()` to calculate the chosen locality statistics at each time step of the time series, and the vertex and temporal normalizations as well.

```
8.7.1) scan_stat(graphs = tsg, k = 1, tau = 3, ell = 3)
```

```
$stat
 [1]          NA          NA          NA          NA
 [5]          NA          NA -0.60000000  1.52841698
 [9] -1.67757505  0.61039045 -1.32542478 -0.06757374
[13]  1.52752523  0.64257593  0.13748270 -4.03561201
[17] -0.68889305  0.23448415  2.00000000 12.74315781

$arg_max_v
 [1] NA NA NA NA NA NA  5 20 17  2  8 19  6  1 16  3  9  5 19
[20] 20
```

The output contains the normalized scan statistics for each time steps. By default it is `'NA'` for the first $\tau + \ell$ time steps, because these are not properly normalized (because of the lack of enough previous time steps). It also contains the central vertices for which the largest (normalized) locality statistics was observed in each time step.

It is clear that the scan statistics is by far the largest in the last time step. The result also correctly points to vertex number 20, which is part of the block with increased activity.

We can also use Φ, the *them* statistics:

```
8.8.1) scan_stat(graph = tsg, locality = "them", k = 1, tau = 3, ell = 3)
```

```
$stat
 [1]          NA          NA          NA          NA
 [5]          NA          NA  0.72108555  0.08544336
 [9] -1.62258927 -1.42362758  1.32299184  1.14064686
[13]  2.02072594 -0.22222222 -0.33333333 -2.38939398
[17] -1.30457724 -0.55109123  0.68606606 17.14886066

$arg_max_v
 [1] NA NA NA NA NA NA 10 20 18  3  8 19 19  1 16 15  5 11  5
[20] 20
```

This gives similar results, with vertex number 20 being the center of the increased activity in time step 20.

A more comprehensive analysis would consider larger neighborhoods of the vertices.

```
8.9.1) library(ggplot2)
   2) tsg_scan <- function(k, locality) {
   3)   tau <- 4
   4)   ell <- 3
```

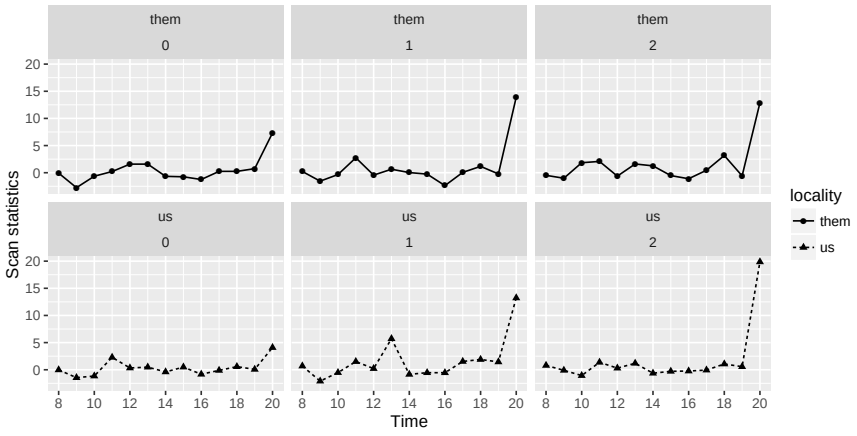


Fig. 8.4 $S_{\tau,\ell,k}(t;\Psi)$ and $S_{\tau,\ell,k}(t;\Phi)$, the temporally-normalized standardized scan statistics using $\tau = 4, \ell = 3$, in time series of graphs for $k = 0, 1, 2$.

```
  scan_stat(tsg, tau = tau, ell = ell, k = k, locality = locality,
            mode = "all") %>%
    .$stat %>%
    na.omit() %>%
    data.frame() %>%
    cbind(seq(tau + ell + 1, 20)) %>%
    set_names(c("Scan statistics", "Time"))
}

pars <- expand.grid(k = 0:2, locality = c("them", "us"))
tsg_scan_results <- plyr::mdply(pars, tsg_scan)

sd3 <- function(x) sd(x) * 3
ggplot(tsg_scan_results, aes(x = Time, y = `Scan statistics`)) +
 facet_wrap(~ locality + k, nrow = 2) +
 geom_line(aes(linetype = locality)) +
 geom_point(aes(shape = locality)) +
 scale_x_continuous(breaks = scales::pretty_breaks()) +
 ylim(range(tsg_scan_results$`Scan statistics`))
```

Figure 8.4 depicts $S_{\tau,\ell,k}(t;\Psi)$ and $S_{\tau,\ell,k}(t;\Phi)$, the temporally-normalized standardized scan statistics for various $k = \{0, 1, 2\}$ using $\tau = 4, \ell = 3$. The horizontal dashed line indicates 3 standard deviation. According to different scenarios, the criterion of anomaly alert is determined by the observer. The higer deviation in terms of number of standard deviations is selected, the lower false postive rate can be achieved; the lower deviation is selected, the higher true positive rate can be achieved. In this example, both $S_{\tau,\ell,k}(t;\Psi)$

and $S_{\tau,\ell,k}(t;\Phi)$ detect the anomaly ($t = 20$) with $k = 1, 2$, only $S_{\tau,\ell,k}(t;\Phi)$ detects it with $k = 0$.

8.5 The Enron email corpus

We use the Enron email data used in Priebe et al (2005) for this experiment. It consists of time series of graphs $\{G_t\}$ with $|V| = 184$ vertices for 189 weeks. An (v, w) edge of a graph means that vertex v sent at least one email to vertex w during that one week period. We remove multiple and loop edges from the graphs.

```
8.10.1) data(enron)
    2) enron <- lapply(enron, simplify)
```

We calculate $S_{\tau,\ell,k}(t;\Psi)$ and $S_{\tau,\ell,k}(t;\Phi)$. We choose both $\tau = \ell = 20$, suggested in Priebe et al (2005).

```
8.11.1) scan_enron <- scan_stat(enron, k = 2, tau = 20, ell = 20,
    2)                          locality = 'Phi', mode = 'out')
```

Figure **??** depicts $S_{\tau,\ell,k}(t;\Psi)$ and $S_{\tau,\ell,k}(t;\Phi)$ in the remaining 149 weeks from August 1999 to June 2002. As indicated in Priebe et al (2005), detections are defined as weeks t such that $S_{\tau,\ell,k} > 5$. Figure **??** we have following observations.

- Both the *them* (Φ) statistics (for $k = 0, 1, 2$) and the *us* (Ψ) statistics (for $k = 0, 1$) indicate a clear anomaly at $t^* = 58$, in December 1999. This coincides with Enron's tentative sham energy deal with Merrill Lynch to meet profit expectations and to boost the stock price Galasyn (2010). The center of suspicious community, employee v_{154} is identified by all five statistics.
- Both *them* ($k = 0$) and *us* ($k = 0, 1, 2$) capture an anomaly at $t^* = 146$, in mid-August 2001. This is the period that Enron CEO Skilling made a resignation announcement when the company was surrounded by public criticisms shown in Galasyn (2010). The center of this activity is v_{95}.
- The *us* statistics detects an anomaly at $t^* = 132$ in late April 2001, undetected by *them*: employee v_{90}'s second-order neighborhood contains 116 emails at $t^* = 132$ but no emails in the previous 20 weeks. In Galasyn (2010), this event appears after the Enron quaterly conference call in which Wall street analyst Richard Grubman questioned Skilling on the company's refusal on releasing a balance sheet, and then got insulted by Skilling.
- The *them* statistics shows a detection at employee v_{135} at $t^* = 136$, June 2001. According to Galasyn (2010), this corresponds to the formal notice of closure and termination of Enron's single largest foreign investment, the Dabhol Power Company in India.

Note that while in some cases both statistics capture the same events, this is not always the case.

Chapter 9
Clustering Multiple Graphs – an NMF approach

9.1 The Graph Clustering Problem

A collection of weighted graphs on n vertices often arises as a "histogram" of interaction events between n actors in terms of their interaction frequencies observed over periods of time. In particular, for each $t = 1, \ldots, T$, the (i, j)th entry $G_{ij,t}$ of G_t encodes the number of times that person i and person j interacted during tth period. This section considers a problem of grouping multiple graphs into few clusters. To give a more precise description of a "graph clustering" problem, consider $(\kappa(1), G_1), \ldots, (\kappa(T), G_T)$ be an (independent) sequence of pairs of a class label $\kappa(t)$ and a (potentially weighted) graph G_t on n vertices. We assume that the class label $\kappa(t)$ takes values in $\{1, \ldots, K\}$ and also that given $\kappa(t) = k$, each G_t is a random graph on n vertices whose distribution depends only on the value of k.

9.2 Theory : NMF, ARI, AIC – what is this?

In non-negative matrix factorization, we begin with a matrix, $A \in \mathbb{R}^{n \times T}$ and seek to approximate A by the product of two non-negative matrices $W \in \mathbb{R}^{n \times r}$ and $H \in \mathbb{R}^{r \times T}$, where r is the rank of the approximation. There are several choices of loss-function that determine the strength of the approximation, and each gives rise to a different factorization. Common choices are divergence:

$$D(A \parallel WH) = \sum_{i,j} A_{i,j} \log \left(\frac{A_{i,j}}{WH_{i,j}} \right) - A_{i,j} + WH_{i,j} \tag{9.1}$$

and squared loss:

$$\|A - WH\|_F^2, \tag{9.2}$$

where

$$\|B\|_F^2 = \sum_{i,j} B_{i,j}^2. \tag{9.3}$$

Note that the divergence equals the Kullback-Liebler divergence when A and WH are doubly stochastic.

Actual minimization of the loss functions can be difficult for various reasons. In the squared loss, the objective is non-convex and has non-unique minimizers. Further, since the problems have constraints which are generally active at the solution, the constraints must be considered while performing the optimization. Lee and Seung propose multiplicative update rules based on the steepest descent direction with a step-size chosen to allow for multiplicative updates that are guaranteed to result in a non-increasing loss function value. This ensures that if a non-negative factorization was chosen initially, the non-negativity constraints would automatically be met **?**. Another popular update rule is "Alternating Least Squares," where the squared loss problem is solved in W and H in alternating steps. These solutions are easy to obtain by standard least-squares procedures **?**.

The attractiveness of non-negative factorization can be seen in the applications, where we are able to give an interpretation of W (basis) and H (weights). Here, we present a simple example, where we form a matrix by taking its columns to be the weighted sum of basis vectors.

Here we use the R package "NMF" **?** to showcase how to use non-negative factorization on a toy problem. First, we build a matrix by drawing its columns from one of two distributions:

```
9.1.1) basis <- matrix( c(1,2,3,4, 4,3,2,1),4,2)
   2) A <- matrix(0,4,3)
   3) A[,1] <- .9*basis[,1]+.1*basis[,2]
   4) A[,2] <- .5*basis[,1]+.5*basis[,2]
   5) A[,3] <- .1*basis[,1]+.9*basis[,2]
```

Next, we apply NMF to it and note the near-zero error:

```
9.2.1) library('NMF')
```

```
Loading required package: methods
Loading required package: pkgmaker
Loading required package: registry

Attaching package: 'pkgmaker'

The following object is masked from 'package:base':

    isNamespaceLoaded

Loading required package: rngtools
Loading required package: cluster
```

```
Warning: replacing previous import by 'ggplot2::unit' when loading 'NMF'
```

```
Warning: replacing previous import by 'ggplot2::arrow' when loading 'NMF'
```

```
NMF - BioConductor layer [OK] | Shared memory capabilities [NO: bigmemory] | Cores 3/4
  To enable shared memory capabilities, try: install.extras('
NMF
')
```

```
theNmf <- nmf(A, rank=2,method = 'lee')
sum(abs(basis(theNmf) %*% coef(theNmf) - A))
```

```
[1] 1.168296e-08
```

Here we introduce a measure for similarity between clusterings, the adjusted rand index (ARI) **?**. Basically, the ARI is calculated by summing up the number of agreements between two clusterings (when pairs of objects are co-clustered in both or are not co-clustered in both) and subtracting off the expected number of agreements if the two clusterings were formed using a generalized hypergeometric distribution and are drawn with fixed number of clusters and cluster sizes. Finally, we divide by the total number of pairs minus the same expected number. The ARI is at most 1, indicating a perfect matching between clusterings. When a clustering is compared to the truth, an ARI value of 0 indicates that the clustering is no better than chance.

Using our confusion matrix above, we calculate the ARI between the truth and the baseline:

$$\text{ARI} = \frac{\binom{9}{2} + \binom{10}{2} - \left(2\binom{10}{2}(\binom{9}{2} + \binom{11}{2})\right)/\binom{20}{2}}{\frac{1}{2}\left(2\binom{10}{2} + \binom{9}{2} + \binom{11}{2}\right) - \left(2\binom{10}{2}(\binom{9}{2} + \binom{11}{2})\right)/\binom{20}{2}} \approx 0.80$$

Our model selection information criteria is defined as follows:

$$\text{AICc} = \text{Negative Log Likelihood} + \text{Penalty Term},$$

where the likelihood is specified by Poisson densities and the penalty term is specified by quantities that grows with the number of parameters. More specifically,

$$\texttt{-loglike} = \sum_{t=1}^{T} \sum_{ij} (X_{ij,t}/N_t) \log(X_{ij,t}/Nt) \tag{9.4}$$

$$\texttt{penalty} = \sum_{k=1}^{r} \frac{1}{\widehat{N}_k} \left(\sum_{ij} \mathbf{1}\{\widehat{W}_{ij,k} > 0\} - 1 \right). \tag{9.5}$$

We make some observation of the penalty term. First, for a graph that is "under-sampled" (i.e., N_t is small), it is difficult for it to be a "stand-alone

motif" graph. Next, a set of motifs such that $\widehat{W}_{\cdot,k_1}$ and $\widehat{W}_{\cdot,k_2}$ share "many" common support is not favored.

9.3 Examples

9.3.1 Wikipedia in English and French

A tensor is a mathematical term for a multi-way array. For instance, a 3-way array, i.e. $\mathbf{X} = (X_{ijk})$, is a tensor, and in particular, a sequence of adjacency matrices has a natural tensor representation. More explicitly, for a pair of graphs, say, $F = (F_{ij})$ and $E = (E_{ij})$ on n vertices, one can associate it with the tensor X, where for $k = 1$, $X_{ij1} = F_{ij}$ and for $k = 2$, $X_{ij2} = E_{ij}$. Then, the mode ℓ matrization of the tensor is a matrix version M of the tensor, where the ℓth column of M is a vectorization of the matrix obtained by fixing the ℓth index to be a particular value.

For example, let us consider a problem of deciding if two matrices F and E are similar or not, where n vertices are a page in the Wikipedia and F_{ij} and E_{ij} indicate whether or not page i and page j are linked. In the listing below, `FrWiki` and `EnWiki` corresponds, respectively, to F and E in our text, and henceforth, we take F and E to be as such.

```
9.4.1) data(wiki_en)
   2) data(wiki_fr)
   3) library(abind)
   4) X <- abind(wiki_fr,wiki_en,along=3)
   5) M <- cbind(as.vector(wiki_fr),as.vector(wiki_en))
```

Wikipedia is an open-source Encyclopedia that is written by a large community of users (everyone who wants to, basically). There are versions in over 200 languages, with various amounts of content. The full data for Wikipedia are freely available for download. A Wikipedia document has one or more of: title, unique ID number, text – the content of the document, images, internal links – links to other Wikipedia documents, external links – links to other content elsewhere on the web, and language links – links to 'the same' document in other languages. The multilingual Wikipedia provide a good testbed for developing methods for analysis of text, translation, and fusion of text and graph information.

Naturally, there are plenty of similarities between E and F since the connectivity between a pair of pages is driven by the relationship between topics on the pages. Nevertheless, E and F are different, i.e., $\|E - F\|_F^2 > 0$, since the pages in Wikipedia are grown "organically", i.e., there is no explicit coordination between English Wikipedia community and French Wikipedia community that try to enforce the similarity between E and F. To answer the

question, we fit the model using the inner dimension to be 1 and then 2 using `gclust.rsvt`, and then compute their information criteria using `getAICc`:

```
9.5.1) gfit1 <- gclust.rsvt(M,1)
   2) gfit2 <- gclust.rsvt(M,2)
   3) gic1 <- getAICc(gfit1)
   4) gic2 <- getAICc(gfit2)
```

The numerical results relevant to answering our decision problem are reported in Table 9.1. In particular, because the AIC value `gic1` for $\mathtt{r} = 1$ is lower than the AIC value `gic2` for $\mathtt{r} = 2$, our analysis suggests that $\mathtt{E}$ and $\mathtt{F}$ have the same connectivity structure.

Now, it is fair to ask the meaning of the claim that $\mathtt{E}$ and $\mathtt{F}$ have the same connectivity structure. This is very much connected with the formulation of `getAICc`, and without going into the full details, we will say that $\mathtt{E}$ and $\mathtt{F}$ are the same provided that for each $\mathtt{i}$ and $\mathtt{j}$,

$$\frac{\mathbf{E}[\mathtt{E}_{\mathtt{ij}}|\mathtt{s}(\mathtt{E})]}{\mathtt{s}(\mathtt{E})} = \frac{\mathbf{E}[\mathtt{F}_{\mathtt{ij}}|\mathtt{s}(\mathtt{F})]}{\mathtt{s}(\mathtt{F})}, \tag{9.6}$$

where $\mathtt{s}(\mathtt{E})$ and $\mathtt{s}(\mathtt{F})$ denotes the total edge weight of $\mathtt{E}$ and $\mathtt{F}$ respectively, i.e., $\mathtt{s}(\mathtt{E}) = \mathbf{1}^\top \mathtt{E} \mathbf{1}$ and $\mathtt{s}(\mathtt{F}) = \mathbf{1}^\top \mathtt{F} \mathbf{1}$. Alternatively, the equality in (9.6) can be restated as a hypothetical question about the probability, conditioning that a new link is formed, of the link being from page $\mathtt{i}$ to page $\mathtt{j}$, and about whether or not the probability being the same for English Wikipedia and French Wikipedia.

A notable consequence of our framework for comparing two graphs is that even for the case where $\mathtt{s}(\mathtt{E})$ is much bigger that $\mathtt{s}(\mathtt{F})$, our analysis framework still permit us to conclude that $\mathtt{E}$ and $\mathtt{F}$ are the same. In words, if we have concluded that $\mathtt{E}$ and $\mathtt{F}$ are different, then one could say that not only that the cross-reference *intensity* in two language is different but also that the connectivity *structure* is different in two languages. In our earlier example, since $\mathtt{r} = 1$ is more favorable, one can say that (despite the fact that $\mathtt{s}(\mathtt{E}) > \mathtt{s}(\mathtt{F})$), $\mathtt{E}$ and $\mathtt{F}$ have the same connectivity structure.

Table 9.1 Are French and English Wikigraphs similar? The answer using `gclust.rsvt` provides an evidence supporting the conclusion *Yes*.

nclust	negloglik	penalty	AIC
1	43.05	1.525	44.58
2	41.65	4	45.65

9.3.2 Wearable RFID Sensors

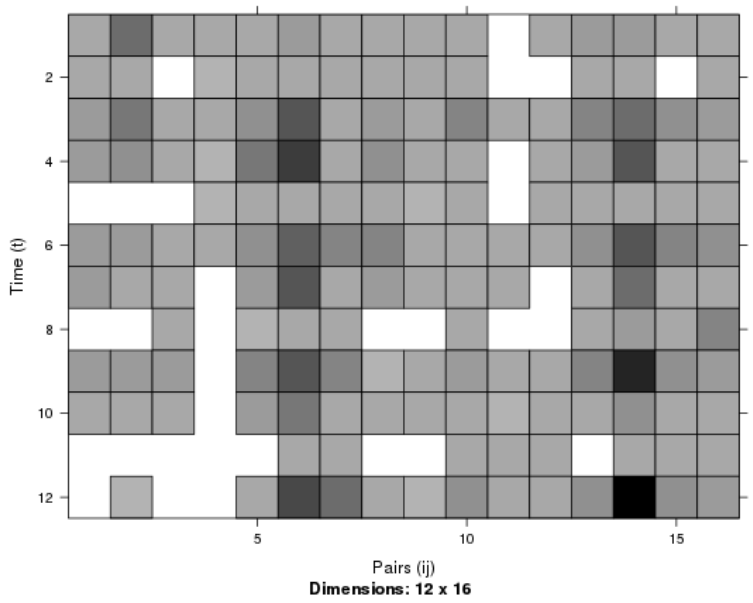


Fig. 9.1 Wearable Sensor Data as a 12×16 matrix, i.e., $\mathtt{M12}^\top$. Each row of $\mathtt{M12}^\top$ for each of the twelve periods, and each column of $\mathtt{M12}^\top$ for each entry of 4×4 matrix.

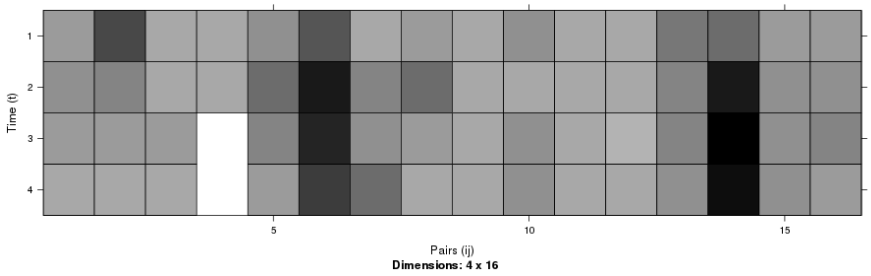


Fig. 9.2 Wearable Sensor Data as a 4×16 matrix, i.e., $\mathtt{M4}^\top$. The $\mathtt{t}$th row of $\mathtt{M4}^\top$ corresponds to the aggregation of the rows from the $(1 + 3 \times (\mathtt{t} - 1))$th row to $3 \times \mathtt{t}$th row of $\mathtt{M4}^\top$. For example, the first row of $\mathtt{M4}^\top$ corresponds to the first three rows of $\mathtt{M12}^\top$

Wearable RFID sensors are used to detect close-range interactions between individuals in the geriatric unit of a university hospital. The study involves 46 health care workers and 29 patients over the span of 4 days and 4 nights (from

Monday through Thursday). It is reported there that "the contact patterns were qualitatively similar from one day to the next", and in this example, we reexamine this statement through our procedure.

For simplicity, individuals were grouped in four classes according to their role in the ward: patients (PAT), medical doctors (physicians and interns, MED), paramedical staff (nurses and nurses' aides, NUR) and administrative staff (ADM). As such, the data naturally lends itself to a block structured graph.

Using the vertex contraction that groups the actors by their role (i.e., PAT, ADM, MED, NUR), we construct a collection of 4×4 weighted adjacency matrices. Dividing the entire duration to four intervals (i.e., by day), we arrive at a collection of four weighted adjacency matrices, each accounting for a 24-hour period.

From Figure 9.2, we make some observations that help us clustering of four graphs. First, only during the first day, NUR and MED interaction rate is high. Only during the third and fourth days, there is zero incident of PAT detecting MED. All four days, NUR->NUR interaction remains intense. Except on the first day, NUR-PAT interaction remains intense. These observations suggest that there are three clusters, where each of the first and the second graph constitute its own cluster, and the third and four graphs form the third cluster.

9.6.1)
```
data(RFIDsensors)
M4 <- sapply(RFIDsensors[[1]], as.vector)
gic1 <- getAICc(gclust.rsvt(M4,1))
gic2 <- getAICc(gclust.rsvt(M4,2))
gic3 <- getAICc(gclust.rsvt(M4,3))
gic4 <- getAICc(gclust.rsvt(M4,4))
```

As shown in Table 9.2, the value of `gic3` is the smallest among the four possible choices, and four and twelve respectively (c.f. and Table 9.3). The general conclusion that one can draw from this observations is that there does not seem to exist any recurring pattern.

Table 9.2 Does the contact pattern differ from day to day?

$\widehat{\mathrm{d}}$	NegLogLik	Penalty	AICc
1	18.1841	0.0010	18.1851
2	17.7843	0.0042	17.7885
3	**17.6532**	**0.0095**	**17.6627**
4	17.6720	0.0168	17.6887

9.7.1)
```
M12 <- sapply(RFIDsensors[[2]],as.vector)
gfit.M4 <- gclust(M4,3)
gfit.M12 <- gclust(M12,3)
```

Table 9.3 Does the contact pattern changes every eight hours? (Top four choices)

$\widehat{d}$	NegLogLik	Penalty	AICc
9	48.46202	0.1018	48.56385
10	48.51386	0.1386	48.65249
11	48.23954	0.1526	48.39221
12	**48.10388**	**0.1876**	**48.29151**

```
4) matplot(t(gfit.M4$H), type='b',cex=c('A','B','C'))
5) matplot(t(gfit.M12$H),type='b',cex=c('a','b','c','d'))
```

One can ask how the 12 period analysis compares with the 4 period version. For this, one can force on a 12 period the three motif clustering, and search therein for a pattern consistent with the result from 4 period analysis. As reported in Table 9.4, the two clustering is consistent with each other. In summary, the suggested conclusion is that the only pair that has the similar interaction pattern is the Wednesday-Thursday and Thursday-Friday pair, and this is indicated by 'C' appearing twice in Table 9.4. While the 12 period analysis yields the best clustering is each graph being its own cluster, the best 3 clustering of the 12 graphs shows that C is represented by the sequence (a,b,a).

Table 9.4 How do fitting a three-cluster model on the daily version and on the 8-hour version correspond to each other? The letters A, B and C code three clusters for the daily version and the letters a, b and c code three clusters for the 8-hour version.

Num. of Graphs	Day 1	Day 2	Day 3	Day 4
4	A	B	C	C
12	(c,c,a)	(a,a,a)	(a,b,a)	(a,b,a)

9.3.3 C. elegans' chemical and electric pathways

Consider graphs based on n neurons, where each edge weight is associated with the functional connectivity between neurons. The area of studying such a graph for further expanding our knowledge of biology is called "connectome". In this section, we introduce such graphs, first being the chemical path way connectivity, and the second being the functional path way connectivity, and consider how we can answer a simple connectom question using `gclust` and `getAICc`.

```
9.8.1) load('celegan')
   2) AeAc <- sapply(list(Ac,Ae) ,as.vector)
```

```
AeAc.gic1 <- getAICc(gclust.rsvt(AeAc,1,method='lee'))
AeAc.gic2 <- getAICc(gclust.rsvt(AeAc,2,method='lee'))

vcmat <- t(apply(diag(9),2,rep,times=c(rep(30,8),39)))
Ae.vc <- vcmat %*% Ae %*% t(vcmat)
Ac.vc <- vcmat %*% Ac %*% t(vcmat)
AeAc.vc <- sapply(list(Ae.vc,Ac.vc),as.vector)

AeAc.vc.gic1 <- getAICc(gclust.rsvt(AeAc,1,method='lee'))
AeAc.vc.gic2 <- getAICc(gclust.rsvt(AeAc,2,method='lee'))
```

The AIC values for $\hat{d} = 1$ and $\hat{d} = 2$ are respectively 30.12 and 33.31, whence this leads us to a suggestion that the connectivity structure of `Ae` and `Ag` could be the same (up to some random noise). A more careful statement of this can be stated as follows:

$$\mathbf{E}[(\mathtt{Ae})_{ij}] = \mathbf{E}[(\mathtt{Ac})_{ij}] \qquad \text{for each pair } ij. \tag{9.7}$$

A way to cross-check the claim that $\hat{d} = 1$ is to utilize the concept of *vertex contraction* that we have seen earlier. The main idea is simple, and we motivate the main idea by way of a overly simplified example. Consider two vectors p and q of positive integers. Then, if p and q are identical, then $\mathbf{1}^\top p = \mathbf{1}^\top q$. On the other hand, even if p and q are different, it is possible that $\mathbf{1}^\top p = \mathbf{1}^\top q$, e.g., consider

$$p = \begin{pmatrix} 1 \\ 2 \end{pmatrix} \text{ and } q = \begin{pmatrix} 2 \\ 1 \end{pmatrix}. \tag{9.8}$$

In other words, if $\boldsymbol{\mu}_e = \mathbf{E}[\mathtt{Ae}]$ and $\boldsymbol{\mu}_c := \mathbf{E}[\mathtt{Ac}]$ are the same, then it follows that for any matrix Q,

$$Q\boldsymbol{\mu}_c Q^\top = Q\boldsymbol{\mu}_e Q^\top \tag{9.9}$$

```
Q <- t(apply(diag(9),2,rep,times=c(rep(30,8),39)))
Ae.vc <- Q %*% Ae %*% t(Q)
Ac.vc <- Q %*% Ac %*% t(Q)
AeAc.vc <- sapply(list(Ae.vc,Ac.vc),as.vector)
```

In Listing **??**, the same vertex contraction is performed on both `Ae` and `Ac`, where the vertex contraction matrix Q is such that each of the first eight groups of thirty vertices is aggregated (collapsed) to a single vertex, and for the last thirty-nine vertices is aggregated to a single vertex. Performing our procedure to the collapsed graphs yields that the AIC values for $\hat{d} = 1$ and $\hat{d} = 2$ are 15.84 and 15.61, suggesting that there is two patterns. Performing a Monte Carlo experiment using 100 random choices for such partitions, we obtain Table

Table 9.5 Monte Carlo experiments involving 100 random vertex contraction

$\widehat{\mathrm{d}}$	1	2
Frequency	3	97

Next, let us consider two alternative vertex contraction "policies" coded in Listing **??** and Listing **??**. In Listing **??**, 279 neurons are collapsed according to their types, yielding 3 vertices and our procedure in this case suggests $\widehat{\mathrm{d}} = 2$. This is consistent with the random vertex contraction results' suggestion that the chemical pathways and electric pathways are sufficiently different with respect to their connectivity structures.

On the other hand, in Listing **??**, 279 neurons are aggregated/collapsed according to their types, yielding 52 vertices, and our procedure in this case suggests $\widehat{\mathrm{d}} = 1$. However, our "more-intense" vertex contraction suggests that the "hypothesis" that $\widehat{\mathrm{d}} = 1$ can not hold true. Note that this does not says that the information criteria mistakenly has chosen $\widehat{\mathrm{d}} = 1$. Rather, the choice made by `getAICc` says that given the amount of data that we have, the most frugal way to represent the data is to use a single mean in the spirit of the principle known in a machine learning community as the *bias-variance trade-off*. Roughly speaking, in our particular case, for $\mathrm{n} = 279$, it amounts to a saying that even if $\boldsymbol{\mu}_e$ and $\boldsymbol{\mu}_a$ are different, with the amount of data given, it is not frugal to model both with the same mean matrix. On the other hand, upon vertex contraction to a much smaller matrices, i.e., each entry in the matrices has a bigger number, such is no longer a hindrance to making a precise decision.

```
9.10.1) n.types <- data.Celegans$Types
    2) n.types.unique <- unique(n.types)
    3) Q <- matrix(0,52,278)
    4) for(itr in 1:3)     Q[itr,which(n.types==n.types.unique[itr])] <- 1
    5) Ae.Q <- Q %*% Ae %*% t(Q)
    6) Ac.Q <- Q %*% Ac %*% t(Q)
    7) AeAc.Q <- sapply(list(Ae.Q,Ac.Q),as.vector)
    8) AeAc.Q.gic <- foreach(itr=1:2,.combine='rbind') %do%     getAICc(gclust.rsvt(AeAc.Q,itr,metho
```

```
9.11.1) n.Vcols = data.Celegans$Vcols
    2) n.Vcols.unique = unique(n.Vcols)
    3) Q = matrix(0,3,278)
    4) for(itr in 1:3)     Q[itr,which(n.Vcols==n.Vcols.unique[itr])] = 1
    5) Ae.Q = Q %*% Ae %*% t(Q)
    6) Ac.Q = Q %*% Ac %*% t(Q)
    7) AeAc.Q = sapply(list(Ae.Q,Ac.Q),as.vector)
    8) AeAc.Q.gic = foreach(itr=1:2,.combine='rbind') %do%     getAICc(gclust.rsvt(AeAc.Q,itr,method=
```

9.3.4 Simulation experiment motivated by real data

We now consider a biologically motivated simulation example, with model parameters extracted from Izhikevich & Edelman (2008). This example is significantly simplified from the (necessarily incompletely understood) biology; nonetheless, it is (loosely) based on biological understanding and serves as a challenging illustrative test case.

We consider a sequence of (random) graphs from a stochastic block model with two motifs specified by $\overline{\mathrm{B}}^{(1)}$ and $\overline{\mathrm{B}}^{(2)}$, where

$$\overline{\mathrm{B}}^{(1)} := \begin{pmatrix} 0.1 & 0.045 & 0.015 & 0.19 & 0.001 \\ 0.045 & 0.05 & 0.035 & 0.14 & 0.03 \\ 0.015 & 0.035 & 0.08 & 0.105 & 0.04 \\ 0.19 & 0.14 & 0.105 & 0.29 & 0.13 \\ 0.001 & 0.03 & 0.04 & 0.13 & 0.09 \end{pmatrix}, \tag{9.10}$$

$$\overline{\mathrm{B}}^{(2)} := \begin{pmatrix} 0.19 & 0.14 & 0.29 & 0.105 & 0.13 \\ 0.001 & 0.03 & 0.13 & 0.04 & 0.09 \\ 0.015 & 0.035 & 0.105 & 0.080 & 0.04 \\ 0.045 & 0.05 & 0.14 & 0.035 & 0.03 \\ 0.1 & 0.045 & 0.19 & 0.015 & 0.001 \end{pmatrix}. \tag{9.11}$$

Note in particular that $\overline{\mathrm{B}}^{(2)}$ is obtained by $\overline{\mathrm{B}}^{(1)}$ by permuting the rows of $\overline{\mathrm{B}}^{(1)}$ and then permuting the columns of $\overline{\mathrm{B}}^{(1)}$.

```
9.12.1) B1 <- rbind(
     2)  c(.1,   .045, .015, .19,  .001),
     3)  c(.045, .05,  .035, .14,  .03),
     4)  c(.015, .035, .08,  .105, .04),
     5)  c(.19,  .14,  .105, .29,  .13),
     6)  c(.001, .03,  .04,  .13,  .09))
     7) P1<- t(matrix(c(
     8)  0,0,0,1,0,
     9)  0,0,0,0,1,
    10)  0,0,1,0,0,
    11)  0,1,0,0,0,
    12)  1,0,0,0,0),5,5))
    13) P2<- t(matrix(c(
    14)  1,0,0,0,0,
    15)  0,1,0,0,0,
    16)  0,0,0,1,0,
    17)  0,0,1,0,0,
```

```
 0,0,0,0,1),5,5))
B2 <- P1 %*% B1 %*% P2
```

Then, for each $t = 1, \ldots, 10$, we take $G(t) \sim SBM(B^{(\kappa(t))}, \nu)$ where for each ij, if $\kappa(t) = r$, then $G_{ij}(t)$ is a Poisson random variable with its success probability $B^{(r)}_{\nu(i),\nu(j)}$, where $\nu : \{1, \ldots, 100\} \to \{1, \ldots, 5\}$ and the cardinality of $\{k : \nu(t) = k\}$ is 20 for each $k = 1, \ldots, 5$.

In short, we have $T = 10$ graphs, each of them has $n = 100$ vertices and follows a 5-block model. There are $m = 20$ vertices in each block. We are using $\overline{B}^{(1)}$ to generate the first 5 graphs while we are using $\overline{B}^{(2)}$ to generate the rest 5 graphs. Moreover, note that the sequence $\{N(t)\}_{t=1}^{T}$ form an i.i.d. sequence of random variables. where $N(t) := \mathbf{1}^\top G(t)\mathbf{1}$. In other words, there are no class-differentiating signal in the edge weights. As such, the only class-differentiating signal present is from the structural difference between $\overline{B}^{(1)}$ and $\overline{B}^{(2)}$.

```
W <- cbind(as.vector(B1),as.vector(B2))
H <- rbind(
rep((c(0,1)),times=c(5,5)),
rep((c(1,0)),times=c(5,5)))
X0 <- apply(W %*% H,2,function(x) Matrix(x,5,5))
Mvec <- lapply(X0,
function(x)   retval <- apply(x,2,rep,times=rep(20,5))  retval <- apply(retval,1,rep,times=rep
Mvec <- sapply(Mvec,as.vector)
Xvec <- apply(Mvec,c(1,2),function(x) rpois(1,x))
```

In Table 9.6. we report the AICc values for performing clustering on a sequence of graphs whose expected values are

$$(\overline{B}^{(1)}, \ldots, \overline{B}^{(1)}, \overline{B}^{(2)}, \ldots, \overline{B}^{(2)}).$$

From the result, we can see that with repeated SVT steps (i.e., `gclust.rsvt`), we obtain the correct number of clusters, namely, 2. But without any SVT step (i.e., `gclust.app`), the algorithm's choice rank 6 is far away from the true rank. So we can conclude that the SVT step improves the model selection performance by getAICc.

Table 9.6 AIC for unperturbed $\overline{B}^{(k)}$

	Rank 1	Rank 2	Rank 3	Rank 4	Rank 5	Rank 6
`gclust.rsvt`	169.34	**167.16**	171.72	176.77	184.00	191.99
`gclust.app`	169.36	163.25	161.23	158.60	156.59	**155.21**

By perturbing the $\overline{B}^{(k)}$ to become sparser or denser. we compare the performance of the algorithm with SVT and without SVT. For each ε, we write

$$B^{(k)}(\varepsilon) = (\varepsilon \overline{B}^{(k)}) \wedge \mathbf{1}\mathbf{1}^{\top},$$

where $\wedge$ denote the operation that takes component-wise minimum. Note that it remains that $B^{(2)}(\varepsilon)$ can be derived from $B^{(1)}(\varepsilon)$ by permutations. We use the parameter ε to control the sparsity of the matrix while keeping the overall structure of B . When ε approaches 0, $B^{(1)}(\varepsilon)$ is close to a zero matrix; when ε is large enough, $B^{(1)}(\varepsilon)$ approaches J. In both cases, the difference between $B^{(1)}(\varepsilon)$ and $B^{(2)}(\varepsilon)$ are decreasing, which means the matrix is transforming from rank-2 to rank-1.

From Figure 9.3, we can see that no matter the matrix becomes sparser or denser, the algorithm with SVT always gives us the true rank except for the extreme case. The algorithm with SVT successfully captures this transformation. As to the algorithm without SVT, we can see that it performs as good as the one with SVT. But when the matrix becomes sparser, it performs consistently bad and do not capture the transformation of the rank.

For another perturbation scheme using

$$B^{(k)}(\varepsilon) = (\overline{B}^{(k)} + \varepsilon \mathbf{1}\mathbf{1}^{\top}) \wedge \mathbf{1}\mathbf{1}^{\top},$$

one can also see, from Figure 9.4, that a similar pattern persists.

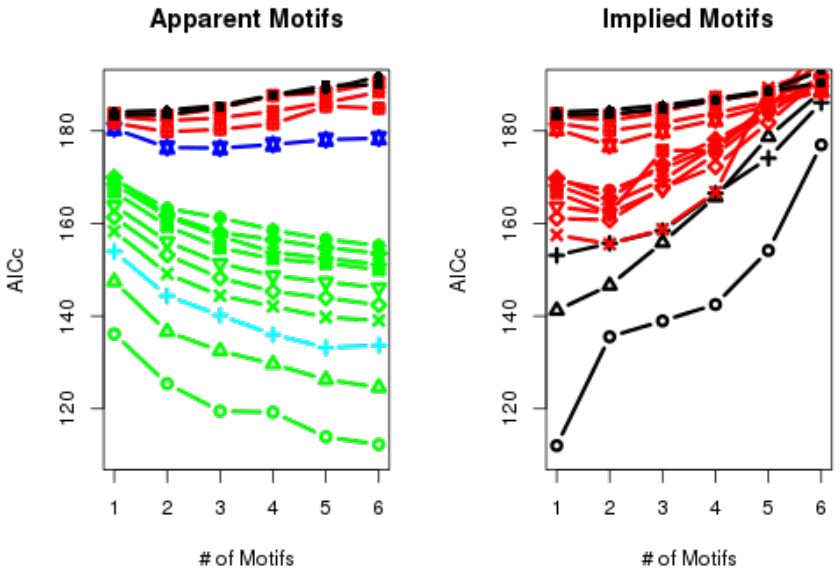


Fig. 9.3 AIC for $(\varepsilon B) \wedge (\mathbf{1}\mathbf{1}^{\top})$ as ε changes

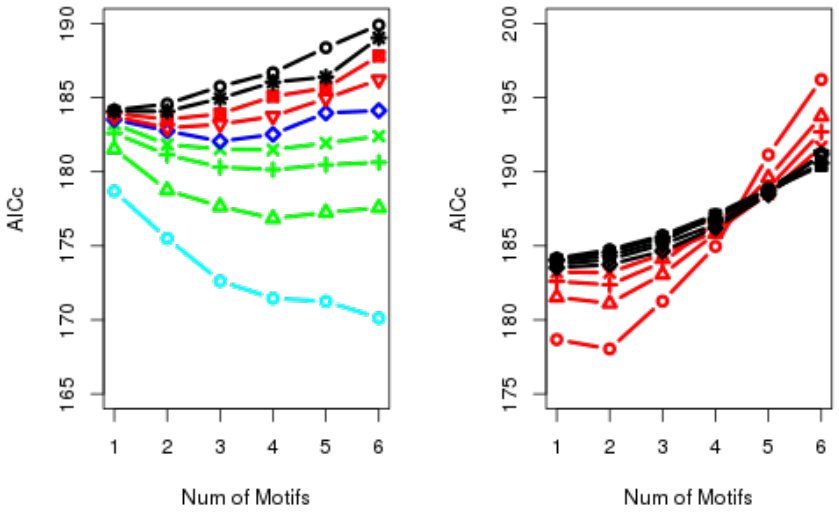


Fig. 9.4 AIC for $(\varepsilon B + \varepsilon \mathbf{1}\mathbf{1}^\top) \wedge (\mathbf{1}\mathbf{1}^\top)$ as ε changes

We will consider the following clustering procedure as the baseline: Form the matrix Λ with $\Lambda_{ij} = \|G(i) - G(j)\|_F$. Use function `pamk` from the `fpc` package on Λ and allow it to consider between 2 and $\min(\texttt{maxRank}, m - 1)$ clusters. With our SBM parameters $B^{\kappa(t)}$ specified in 9.10 and 9.11, we generate $T = 20$ graphs, with $\kappa(t) = 1$ for $t = 1, 2, \ldots, 9$, and $\kappa(t) = 2$ for $t = 10, 11, 12, \ldots, 20$. That is, the first nine graphs are from cluster 1 and the rest are from cluster 2. We let $m = 10$, so that $n = 50$.

```
9.14.1) mcout.pamk
    2)   <- foreach(mcitr=1:100,.combine='rbind') %dopar%         Xvec.r = Xvec100.r.tmp[[mcitr]]
    3) mcout.pamk.meanari <- mean(apply(mcout.pamk,1,
    4)         function(x) adjustedRandIndex(x,c(rep(1,5),rep(2,5)))))

9.15.1) mcout.gclust <-
    2)   foreach(mcitr=1:100,
    3)   .combine='rbind',
    4)   .errorhandling='remove',
    5)   .inorder=FALSE) %dopar%         Xvec.r = Matrix(Xvec100.r.tmp[[mcitr]])        aic.rsvt =
    6) mcout.gclust.meanari <- mean(apply(mcout.gclust,1,
    7)         function(x) adjustedRandIndex(x,c(rep(1,5),rep(2,5)))))
```

For one realization of the data, we see that gclust outperforms the baseline, as gclust correctly clusters all graphs and the baseline clusters one graph incorrectly. Both methods correctly identified 2 clusters.

Table 9.7 Confusion matrices for `gclust` and `baseline`

	True Cluster 1	True Cluster 2
`gclust` Cluster 1	9	0
`gclust` Cluster 2	0	11

	True Cluster 1	True Cluster 2
`pamk` Cluster 1	9	1
`pamk` Cluster 2	0	10

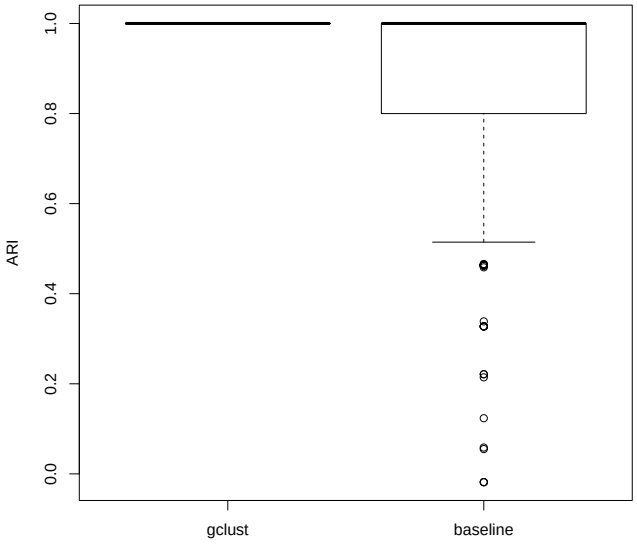


Fig. 9.5 Performance Comparison Between Two Clustering Approaches

After repeating the above experiment 1000 times, we obtain a range of ARI values for each method. We've summarized them using a boxplot as shown below. Basically, gclust never has a sub-perfect ARI, while the baseline method sometimes does. After repeating the above experiment 1000 times, we obtain a range of ARI values for each method.

9.3.5 Simulated experiment with an configuration antagonistic to "lee" and "brunet" options

In this example, we illustrate a typical situation under which (getAICc o gclust) does not perform well, and provide a heuristic that sometime remedy the situation.

```
9.16.1) nvertex <- 10
    2) maxT <- 10
    3) distr <- c(rep(10,maxT/2),rep(10,maxT/2))
    4) A <- foreach(itr=1:maxT,.combine='cbind') %do%    rpois(nvertexˆ2,lambda=distr[itr])
    5) gic <- foreach(itr=1:10,.combine='rbind') %do%    getAICc(gclust.rsvt(A,itr,method='lee'))
```

```
9.17.1) betagrid <- seq(0,1,by=0.25)
    2) gic <- foreach(beta=betagrid) %do%         foreach(itr=1:10),.combine='rbind') %do%
    3) sapply(gic,function(x) which.min(x[,4]))
```

The reason that Listing **??** pose a significant challenge to (getAICc o gclust) is that although $G(t)/s(G(t))$ yields a consistent estimate of its expected value $\mathbf{E}[G(t)|s(G(t))]/s(G(t))$, the first half has a significantly larger variance than the last half.

We now return to our first example using Wikipedia data.

```
9.18.1) vcpol <- c(rep(33,10),34,
    2)         rep(25,10),11,rep(18,10), 11,
    3)         rep(19,10), 12,
    4)         rep(19,10),19,
    5)         rep(9,10),6)
    6)
    7) vcmat <- t(apply(diag(length(mybrks)),2,rep,times=vcpol));
    8) rWikiFr <- vcmat %*% WikiFr %*% t(vcmat)
    9) rWikiEn <- vcmat %*% WikiEn %*% t(vcmat)
   10)
   11) tWikiEn<- vector('list',6)
   12) indx <- 1
   13) for(itr in seq(1,66,by=11))     LHS <- itr;RHS = itr + 10;    tWikiEn[[indx]] <- rWikiEn[LHS:R
   14)
   15) tWikiFr <- vector('list',6)
   16) indx <- 1
   17) for(itr in seq(1,66,by=11))     LHS <- itr;RHS = itr + 10;    tWikiFr[[indx]] <- tWikiFr[LHS:R
   18) tWiki <- c(tWikiEn,tWikiFr)
   19)
   20) aic.rsvt <- t(sapply(1:12,
   21)     function(x) getAICc(gdclust.rsvt(tWikiFr,x))));
```

Table 9.8 Cluster assignment based on taking $\widehat{d} = 6$ for clustering 12 Wikigraphs shows the perfect agreement – based on 6×6 graphs.

–	People	Places	Dates	Things	Math	Categories
English Wikipages	$*$	$+$	$\times$	$\bullet$	$\diamond$	$\circ$
French Wikipages	$*$	$+$	$\times$	$\bullet$	$\diamond$	$\circ$

Table 9.9 Are French and English Wikigraphs similar? The answer using `gclust.app` is *No*.

nclust	negloglik	penalty	AIC
1	43.11	1.525	44.63
2	41.65	4	45.65

9.3.6 Shooting patterns of NBA players

To find out 9 NBA player's shooting patterns using statistical learning methods(especially Non-Negative Matrix Factorization) with R.

Paper "Factorized Point Process Intensities: A Spatial Analysis of Professional Basketball"(Harvard University) Data Explanation: Our simulation data is based on raw data about made and missed field goal attempt locations from roughly half of the games in the 2012-2013 NBA regular seasons. These data were collected by optical sensors as part of a program to introduce spatio-temporal information to basketball analytics. We discretize the basketball court into V tiles and compute X such that $X_{n,\nu} = \{x_{n,i} : x_{n,i} \in \nu\}$, the number of shots by player n in tile ν. The following table (See Table 9.10) is the data used in this paper:

Table 9.10 Multinomial probability for 9 NBA players' shooting location

Player	Tile 1	Tile 2	Tile 3	Tile 4	Tile 5	Tile 6	Tile 7	Tile 8	Tile 9	Tile 10
LeBron James	0.21	0.16	0.12	0.09	0.04	0.07	0.00	0.07	0.08	0.17
Brook Lopez	0.06	0.27	0.43	0.09	0.01	0.03	0.08	0.03	0.00	0.01
Tyson Chandler	0.26	0.65	0.03	0.00	0.01	0.02	0.01	0.01	0.02	0.01
Marc Gasol	0.19	0.02	0.17	0.01	0.33	0.25	0.00	0.01	0.00	0.03
Tony Parker	0.12	0.22	0.17	0.07	0.21	0.07	0.08	0.06	0.00	0.00
Kyrie Irving	0.13	0.10	0.09	0.13	0.16	0.02	0.13	0.00	0.10	0.14
Stephen Curry	0.08	0.03	0.07	0.01	0.10	0.08	0.22	0.05	0.10	0.24
James Harden	0.34	0.00	0.11	0.00	0.03	0.02	0.13	0.00	0.11	0.26
Steve Novak	0.00	0.01	0.00	0.02	0.00	0.00	0.01	0.27	0.35	0.34

There are 9 players and 10 patterns in the data set. Each cell x_{ij} in this table represents the probability of `ith` player shooting in tile `j`.For instance,

see row “Tony Parker" column “Tile 3", the number 0.17 means that there is a 0.17 possibility that Tony Parker shoots in Tile 3.

Procedures After introducing background knowledge, we discuss how we can use the techniques above to research NBA players' shooting patterns.

9.3.6.1 How to get “mydata"

Notation: Let x_{ij} represents the number of shoots for ith player in jth tile. Assumptions: Assuming that the number of shoots follows a multinomial distribution. We have a probability vector of each player and the total number of shoots each player tried, we can get the following data set (See Table 9.11) by parametric bootstrap method.

Table 9.11 Number of Shoots

Pattern	LeBron	Brook	Tyson	Marc	Tony	Kyrie	Stephen	James	Steve
1	147	27	165	138	55	108	52	306	0
2	10	146	373	19	89	79	15	0	8
3	92	221	16	109	69	69	31	110	0
4	51	47	0	4	30	104	14	0	17
5	22	3	4	268	76	134	71	24	0
6	62	9	16	198	27	21	38	25	0
7	0	32	5	0	33	98	130	136	6
8	46	12	0	10	21	0	16	0	200
9	45	0	13	0	0	69	60	100	232
10	125	3	8	24	0	118	121	199	217

Chapter 10
Cliques and Graphlets

10.1 Cliques

A clique is a subset of vertices, such that its induced subgraph is the complete graph. In other words, a clique is a set of vertices in which each pair of vertices is connected. Cliques are very important in graph theory and also in applications, as they constitute the basis of many more complex notions.

Often, Somewhat imprecisely the induced subgraph of the clique vertices is also called a clique.

10.1.1 Maximal cliques

A clique is called maximal if it is not part of a larger clique. Note, that a graph may have several maximal cliques of different sizes.

The origanization of cliques in a graph tells us about its structure. Take the US airport network for example. This network dissortative with respect to vertex degree: most edges connect high degree vertices to low degree vertices. In other words, the majority of flights happen between big airline hubs and small regional airports, and the rest of the flights between the hubs, but there are not too many flights between small airports. We expect that this structure is also reflected in the cliques, and the large cliques contain the big airport hubs. The `clique_num()` function calculates the size of the largest clique in the graph.

clique_num()

```
10.1.1) library(igraphdata)
    2) data(USairports)
    3) u_air <- as.undirected(USairports, mode = "collapse") %>%
    4)   simplify()
    5) clique_num(u_air)
```

```
[1] 27
```

```
lc_air <- largest_cliques(u_air)
length(lc_air)
```

```
[1] 40
```

Note that we convert the graph to to undirected. Most research on cliques deals with undirected graphs, although directed cliques can be defined as well. The airport network contains 40 cliques of 27 vertices each. Lets see if these are indeed the airline hubs. First we calculate the airports that are part of all largest clique. Then we compare the degree of these vertices to the rest of the network.

```
lc_air_common <- do.call(intersection, lc_air)
lc_air_common
```

```
+ 21/755 vertices, named, from 5d3f7f9:
 [1] ATL ORD DTW CLT MCO PHL MSP DEN BWI IAH DFW CLE EWR CVG
[15] LAS DCA FLL TPA PHX MEM STL
```

```
lc_air_common %>%
  degree(graph = u_air) %>%
  summary()
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   64.0    76.0   103.0   107.4   134.0   166.0
```

```
difference(V(u_air), lc_air_common) %>%
  degree(graph = u_air) %>%
  summary()
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  0.000   2.000   4.000   9.523  10.000 109.000
```

Indeed, 21 airports are common in all largest cliques, and their degrees are much higher in general than for the rest of the network.

Finding cliques in a graph is in general a hard problem, most variations of its are NP-complete or require exponential running time, in non-constrained graphs:

- Finding whether there is a clique of a given size in the graph.
- Consequently, finding all cliques of a graph.
- Listing all maximal cliques of a graph has exponential running time, as there might be exponential number of cliques.

Despite this, there is an abundance of clique finding algorithms, and many of them in real networks, simply because real networks tend not to have very large cliques.

10.2 Clique percolation

The *clique percolation method* (Palla et al, 2005) was one of the first overlapping community detection methods published in the literature. The method is able to detect overlapping communities where a set of vertices is shared between two and more communities. The algorithm may also find *outlier vertices*, i.e. vertices that are not likely to belong to any of the detected communities.

clique percolation method

outlier vertices

Unlike the methods discussed so far, the clique percolation method operates with a very clear community definition. Each community is built of k-cliques: subgraphs of size k where each vertex is connected to all the other vertices within the subgraph. Two k-cliques are said to be *adjacent* if they share $k-1$ vertices, and a k-clique is *reachable* from another k-clique if one of the cliques can be transformed to the other via a path of pairwise adjacent cliques. A community is then a maximal set of k-cliques that are mutually reachable from each other. Overlaps may then naturally arise between the communities since a vertex may be a member of two or more cliques that are not reachable from each other. One can also think about the process as follows. Imagine that a k-clique "template" is placed over one of the k-cliques of the original network. The template is then rolled over the network. In each step, the only allowed move is to replace exactly one of the vertices in the template with another one such that the new subgraph still remains a k-clique. The set of vertices reachable from each other via this rolling process is considered a community. The procedure has to be repeated for all the k-cliques in order to find the whole set of communities. Fig. **??** shows an example graph and the communities found by the clique percolation method on this graph with $k=4$.

The method has only one parameter: k, the size of the clique template that is rolled over the network. k acts as a resolution parameter as lower values of k result in many communities with a smaller number of internal links, while higher values of k find only a small set of densely knit subgraphs. The special case of $k=2$ finds the connected components of the graph, just like we did with breadth first search using the `clusters()` method in Section 2.6.

The implementation of the method works with the k-clique reachability graph. In this graph, each vertex represents a clique of the original graph, and two vertices are connected by an edge if the corresponding two k-cliques are adjacent. This graph can be very large even for small (but relatively dense) input graphs. However, it is easy to recognize that it is enough to work with the maximal cliques of the graph instead of finding all the k-cliques. A *maximal clique* is a clique that can not be extended any further by adding more vertices. Each subgraph of k vertices in a maximal clique is a k-clique on its own. Furthermore, since each k-clique in a maximal clique is reachable from any other k-clique in the same maximal clique, and two maximal cliques are reachable via a path of k-cliques if they share at least $k-1$ vertices, the reachability graph over the maximal cliques provides exactly the same in-

maximal clique

formation for us as the k-clique reachability graph while being significantly smaller: a maximal clique of n vertices contains $\binom{n}{k}$ k-cliques, thus we managed to replace $\binom{n}{k}$ vertices of the k-clique reachability graph by a single vertex in the maximal clique reachability graph. The connected components of the maximal clique reachability graph then give us the communities of the original graph.

igraph does not implement the clique percolation method natively, but a simple but efficient implementation can be provided in only a few lines of code:

```
10.6.1) cluster_clique_percolation <- function(graph, k = 3)   clq <- max_cliques(graph, min = k)  if
```

The code is mostly self-explanatory. First we find the maximal cliques of the original graph using the `maximal_cliques()` function and filter them to the ones containing at least `k` vertices (where `k` is the parameter of the function). Then we iterate over all pairs of cliques and record the adjacent pairs in an edge list. This list is then passed on to the `make_undirected_graph()` constructor to create the maximal clique reachability graph. Then this graph is decomposed into connected components. The final communities are then determined by taking the union of all the vertices in the cliques of the corresponding cluster in the maximal clique reachability graph. The method returns a list of the communities, each community is a vertex sequence.

To test our implementation, we will try to reproduce the scientific communities of Giorgio Parisi, a well-known theoretical physicist who is known to have contributed to different areas of physics. Such a versatile scientist is well expected to appear in multiple scientific communities. The data we will use is a scientific co-authorship network derived from the Los Alamos Condensed Matter e-print archive by Newman (2004), last updated in 2005. This is almost the same as the dataset used by Palla et al (2005) in the publication describing the clique percolation method, with only minor differences. The dataset is available in the Nexus network dataset repository, therefore we will simply ask igraph to retrieve the data from Nexus:

```
10.7.1) cond_mat <- nexus.get("condmatcollab2005")
```

Following Palla et al (2005), we first remove edges with unknown weight, and edges with weight not larger than 0.75 and then run the clique percolation method with $k = 4$. This will take some time, so be patient:

```
10.8.1) cond_mat <- delete_edges(cond_mat, E(cond_mat)[is.na(weight)])
    2) cond_mat <- delete_edges(cond_mat, E(cond_mat)[weight <= 0.75])
    3) cover <- cluster_clique_percolation(cond_mat, k = 4)
```

Finally, we list the names of the vertices in those communities that contain Giorgio Parisi:

```
10.9.1) drop_null <- function(x) x [! vapply(x, is.null, TRUE) ]
    2) lapply(cover, function(x)
```

```
  if (V(cond_mat)['PARISI, G'] %in% x) x else NULL) %>%
    drop_null()
```

```
[[1]]
+ 5/40421 vertices, named, from 82f1983:
[1] LEUZZI, L   PARISI, G   CRISANTI, A RITORT, F   RIZZO, T

[[2]]
+ 13/40421 vertices, named, from 82f1983:
 [1] TARANCON, A        MARTIN-MAYOR, V    RUIZ-LORENZO, JJ
 [4] FERNANDEZ, LA      FRANZ, S           PARISI, G
 [7] RITORT, F          MARINARI, E        RICCI-TERSENGHI, F
[10] PAGNANI, A         PICCO, M           SUDUPE, AM
[13] BALLESTEROS, HG

[[3]]
+ 4/40421 vertices, named, from 82f1983:
[1] LANCASTER, D PARISI, G    RITORT, F    DEAN, DS

[[4]]
+ 4/40421 vertices, named, from 82f1983:
[1] VERROCCHIO, P   PARISI, G       GRIGERA, TS
[4] MARTIN-MAYOR, V

[[5]]
+ 6/40421 vertices, named, from 82f1983:
[1] ANGELANI, L  LEONARDO, RD SCIORTINO, F RUOCCO, G
[5] SCALA, A     PARISI, G

[[6]]
+ 4/40421 vertices, named, from 82f1983:
[1] GIARDINA, I CAVAGNA, A  GRIGERA, TS PARISI, G
```

This is almost identical to the ones obtained by Palla et al (2005), with the exceptions of Jorge Kurchan and Leticia F. Cugliandolo who are missing from our results. We can easily confirm that this is due to the absence of edges between them and Giorgio Parisi in our data:

```
cond_mat["KURCHAN, J", "PARISI, G"]
```

```
[1] 0
```

```
cond_mat["CUGLIANDOLO, LF", "PARISI, G"]
```

```
[1] 0
```

10.3 Graphlets

In this Section we briefly discuss "graphlet decomposition", a method that expresses a graph as the sum of its overlapping subgraphs. Graphlet decomposition can be considered as a community detection method, we discuss it here, because its natural implementation relies on finding the maximal cliques of the graph.

As an introductory example, consider a graph of actors, where connections are formed based on common memberships in groups. Take all students at a university. Each class can be considered as a group, the attending students its members. But several other groups exist: the students who live at a certain dormitory, or the ones that are members of the same sports team, etc. In the graphlet model two students (nodes) that are members of at least one common group are socially connected. The strengths of the connections depend on the groups: some groups are small and tight, with strong conenctions (the squash team with only 10 members), others are bigger and tipically less strongly connected (all 400 students that took the Statistics 101 course this year). It is also true, that group size and connection size is not necessarily correlated, a larger group might be more tight than a smaller one, and the graphlet model allows for this.

In this model, each group is a clique in the graph, and it is assigned a weight, the stregth of the ties between the members of the group. A pair of actors might be connecteed via several groups; the weight of a single edge is the sum of the weights of all common groups of the two actors it connects. E.g. if Alice and Bob are both members of the squash team, and both took the introductory statistics course, but have no other common groups, the weight of their edge is $\mu_{\text{squash}} + \mu_{\text{stats101}}$.

The graphlet decomposition of a matrix Λ with non-negative entries λ_{ij} is defined as $\Lambda = BWB'$, where B is an $N \times K$ binary matrix, W is a $K \times K$ diagonal matrix. Explicitly, we have

$$\begin{pmatrix} 0 & \lambda_{12} & \dots & \lambda_{1N} \\ \lambda_{21} & 0 & \dots & \lambda_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{N1} & \lambda_{N2} & \dots & 0 \end{pmatrix} = B \begin{pmatrix} \mu_1 & 0 & \dots & 0 \\ 0 & \mu_2 & \dots & 0 \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \dots & \mu_K \end{pmatrix} B', \tag{10.1}$$

where $\mu_i > 0$ is required for each i.

The matrix B defines the groups of the network: each $b_{\cdot i}$ column is a binary group membership vector, especially shown if we rewrite the matrix equation in the form

$$\Lambda = \sum_{i=1}^{K} \mu_i P_i = \sum_{i=1}^{K} \mu_i (b_{\cdot i} b'_{\cdot i})^*. \tag{10.2}$$

P_i is a binary matrix, with zero diagonal. The $()^*$ operator sets the diagonal of the matrix to zero.

The graphlet decomposition method takes an observed (integer) weighted network, and infers the latent groups of the network, together with the group weights. We model the observed network Y as

$$Y \sim \text{Poission}_+(\sum_{i=1}^{K} \mu_i P_i). \tag{10.3}$$

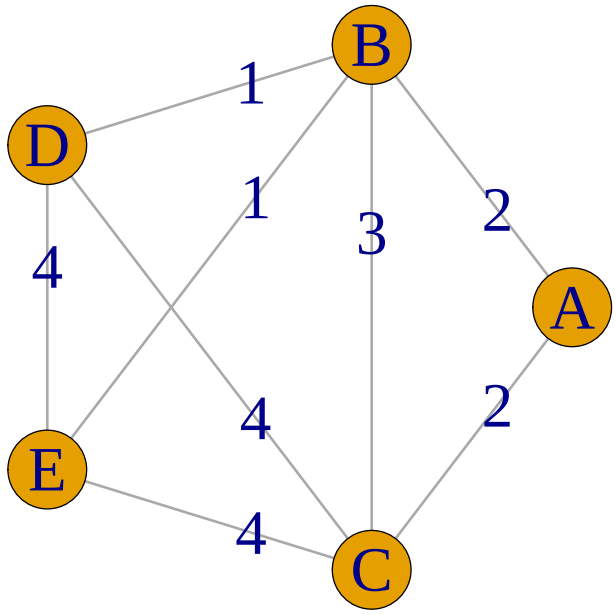
Note that the $P_i = (b_{\cdot i} b'_{\cdot i})^*$ basis elements correspond to cliques of the network. In other words, all connections within a group share the same group weight, as we already mentioned above.

10.3.1 The algorithm

The graphlet decomposition algorithm (as implemented in igraph), works for graphs with integer edge weights. It first creates a candidate set of basis elements (i.e. a candidate set of groups), by a simple iterative maximal clique finder algorithm. Then selects a subset of this basis, via an Expectation-Maximization (EM) algorithm, and also finds the μ_i group coefficients. The weights of several canditate basis elements are set to zero.

In igraph the two steps can be performed indiviadually, via the `graphlet_basis()` and `graphlet_proj()` functions. It is also possible to run both steps in one go, with the `graphlets()` function. We show the method on a small toy graph first.

```
10.12.1) toyg <- make_graph(~ A:B:E [2], B:C:D:E [1], C:D:E [3]) %>%
     2)   add_layout_(in_circle(c('A','B','C','D','E'))) %>%
     3)   set_edge_(label = E(.)$weight)
     4) plot(toyg)
```



This artificial graph is the union of three cliques. Each clique has a different edge weight. The first clique includes vertices `A`, `B` and `C` and has edge weights 2, the second clique has vertices `B`, `V`, `D`, `E` and has edge weights 1, the third clique has vertices `C`, `D` and `E`, and has edge weights 3. The union of them has edge weights between 1 and 4.

```
10.13.1) toy_basis <- graphlet_basis(toyg)
      2) toy_basis
```

```
$cliques
$cliques[[1]]
[1] 1 2 3

$cliques[[2]]
[1] 2 3 4 5

$cliques[[3]]
[1] 2 3
```

```
$cliques[[4]]
[1] 3 4 5


$thresholds
[1] 2 1 3 4
```

In the first stage of the graphlet algorithm, we go over all different edge weight values, and find maximum cliques where each edge has at least the target edge weight. We only consider each clique only once. Here are the cliques that we can find:

- For weight 4, we find `C-D-E`.
- For weight 3, we find `B-E` (`C-D-E` is not considered again).
- For weight 2, we find `A-B-E`.
- For weight 1, we find `B-C-D-D`.

In this artificial example, all three real groups of the graph are part of the candidate basis set and there is also an extra clique.

In the projection step, the EM algorithm aims to find the group weights that are most probable for the original graph, assuming the Poisson model.

```
10.14.1) toy_glet <- graphlet_proj(toyg, cliques = toy_basis$cliques)
     2) toy_glet
```

```
[1]  9.255426e-01  8.614780e-01 2.908813e-253  1.138422e+00
```

10.3.1.1 Consistency of $\hat{B}^c$ as a Generalized Method of Moments Estimator

In this section we propose an continuous almost everywhere function which imitates the process in algorithm (**??**) and illustrate that the estimator $\hat{B}^c$ is a GMM estimator for B^c. Then we will show that considering regimes that guaranties consistency of overall estimator for Poisson rates $\hat{\Lambda} = Y$ we will have the consistency of $\hat{B}^c$.

Generalized Method of Moments(GMM): Generalized method of moments has been define as following:

Definition 1 *suppose we have data generated from a model as* $P(Y|\Theta)$ *and there exists a vector valued function* $g(Y, \Theta)$ *such that:*

$$E\{g(Y, \Theta)\} = 0 \tag{10.4}$$

Then solving equation $g(D, \theta) = 0$ *provides a GMM estimator for the parameter* θ.

Algorithm (??) as a GMM: We can consider the algorithm (**??**) as GMM method using the following definition.

Definition 2 *Given a matrix* X *with positive elements, the thresholding function* $\mathsf{f}(\mathsf{t},\mathsf{X}) : (\mathcal{R},\mathcal{M}) \rightarrow \mathbb{B}$ *is:*

$$\mathsf{f}(\mathsf{t},\mathsf{X}) = \mathtt{maximalcliques}(\boldsymbol{1}(\mathsf{X} \geq \mathsf{t})) \tag{10.5}$$

Where the cliques are represented as an $\mathsf{N} \times \mathsf{K}$ *binary matrix and sorted with a unique order (e.g. a Lexicographical order).*

Lemma 1 *Function* $\mathsf{f}(\mathsf{t},\mathsf{X})$ *is almost continuous on* (t,X) *for a lebesgue measure on* t.

Definition 3 *Given a matrix* Y *with positive elements, the thresholding function* $\mathsf{g}(\mathsf{t},\mathsf{Y},\Lambda) : (\mathcal{R},\mathcal{R},\mathcal{M},\Lambda) \rightarrow \mathbb{B}$ *is defined as:*

$$\mathsf{g}(\hat{\mathsf{t}},\mathsf{Y},\mathsf{t},\Lambda) = \mathsf{f}(\hat{\mathsf{t}},\bar{\mathsf{Y}}) - \mathsf{f}(\mathsf{t},\bar{\Lambda}) \tag{10.6}$$

Where $\bar{\mathsf{Y}} = \frac{\mathsf{Y}}{\sum_{\mathsf{ij}} \mathsf{Y}_{\mathsf{ij}}}$ *and* $\bar{\Lambda} = \frac{\Lambda}{\sum_{\mathsf{ij}} \Lambda_{\mathsf{ij}}}$.

Lemma 2 *we consider a set of thresholds* $\{\mathsf{t}_1,\mathsf{t}_2,..,\mathsf{t}_\mathsf{T}\}$ *and* $\{\hat{\mathsf{t}}_1,\hat{\mathsf{t}}_2,..,\hat{\mathsf{t}}_\mathsf{T}\}$ *which are all distinct elements in matrix* $\bar{\Lambda}$ *and* $\bar{\mathsf{Y}}$ *respectively, and* $\mathsf{T} < \frac{\mathsf{N}^2}{2}$. *For the* $\mathsf{g}(.,.,.,.)$ *in the above definition we have:*

$$\mathsf{E}\{\mathsf{g}(\hat{\mathsf{t}}_\mathsf{i},\mathsf{Y},\mathsf{t}_\mathsf{i},\Lambda)\} = 0 \tag{10.7}$$

$$\mathsf{E}\{\cup_\mathsf{i}\ \mathsf{g}(\hat{\mathsf{t}}_\mathsf{i},\mathsf{Y},\mathsf{t}_\mathsf{i},\Lambda)\} = 0 \tag{10.8}$$

Considering generation of B^c from algorithm (**??**), for non expandable set of basis, we have:

$$\cup_\mathsf{i} \mathsf{f}(\mathsf{t}_\mathsf{i},\bar{\Lambda}) = \mathsf{B}^\mathsf{c}$$

Using the above lemma we can define a GMM estimator for B^c as following:

$$\hat{\mathsf{B}}^\mathsf{c} = \cup_\mathsf{i} \mathsf{f}(\hat{\mathsf{t}}_\mathsf{i},\bar{\mathsf{Y}}) \tag{10.9}$$

This estimator is the output of the algorithm (**??**).

Asymptotic behavior of GMM estimator for $\hat{\mathsf{B}}^\mathsf{c}$:

Theorem 1 *For a sequence of matrices* $\mathsf{Y}_1,...,\mathsf{Y}_\mathsf{n},...$ *converges in probability to* Λ*, which is parameterized by a nonexpandable set* B*, and threshold levels* $\hat{\mathsf{t}}_\mathsf{i}^1,..,\hat{\mathsf{t}}_\mathsf{i}^\mathsf{n},...$ *with an existing limit, for the almost continuous function* $\mathsf{f}(.,.)$ *we have:*

- $\lim_{\mathsf{n}\rightarrow\infty} \mathsf{f}(\hat{\mathsf{t}}_\mathsf{i}^\mathsf{n},\bar{\mathsf{Y}}_\mathsf{n}) \rightarrow^\mathsf{P} \mathsf{f}(\lim_{\mathsf{n}\rightarrow\infty} \hat{\mathsf{t}}_\mathsf{i}^\mathsf{n}, \lim_{\mathsf{n}\rightarrow\infty} \bar{\mathsf{Y}}_\mathsf{n}) = \mathsf{f}(\mathsf{t}_\mathsf{i},\bar{\Lambda})$.
- $\lim_{\mathsf{n}\rightarrow\infty} \hat{\mathsf{B}}_\mathsf{n}^\mathsf{c} = \cup_\mathsf{i} \mathsf{f}(\lim_{\mathsf{n}\rightarrow\infty} \hat{\mathsf{t}}_\mathsf{i}^\mathsf{n}, \lim_{\mathsf{n}\rightarrow\infty} \bar{\mathsf{Y}}_\mathsf{n}) = \cup_\mathsf{i} \mathsf{f}(\mathsf{t}_\mathsf{i},\bar{\Lambda}) = \mathsf{B}^\mathsf{c}$

If we have an asymptotic regime in which $\bar{\mathsf{Y}_\mathsf{n}} \rightarrow^\mathsf{P} \bar{\Lambda}$ and $\mathsf{t}_\mathsf{i}^\mathsf{n} \rightarrow^\mathsf{P} \mathsf{t}_\mathsf{i}$, then we have consistency of the model for B^cs under the same regime.

For the case of truncated Poisson the following asymptotic regime will provide the convergence of $\bar{Y}_n \to^P \bar{\Lambda}$:

- to put the final regime.

10.3.1.2 Consistency of Maximum Likelihood Estimator for $\vec{\mu}$ given true B^c

The proposed model for data is truncated Poisson, however, by conditioning the data on B^c we ignore the zero weights and hence we have Poisson likelihood for the weights on edges corresponding to B^cs. This is a direct result of the following lemma.

Lemma 3 *We have* $B \subseteq B^c$ *and The edges in* B^c *are equal to the edges in* B *meaning* $\mathtt{Edges}(B^c) = \mathtt{Edgest}(B)$.

μ_ls can be found using the ML approach as follows:

$$\overrightarrow{\mu}_{ml} = \max_{\overrightarrow{\mu}} l(\overrightarrow{\mu}|Y, B^c) \tag{10.10}$$

Lemma 4 *The two following optimizations are equivalent:*

- *Maximizing likelihood function:*

$$\max_{\overrightarrow{\mu}} l(\overrightarrow{\mu}|Y, B^c)$$

- *Minimizing Kl divergence with a constraint:*

$$\min_{\mu} D(Y/M \| \sum_l \mu_l P_l/M)$$

with the constraint $\sum_l \mu_l a_l = M$.

We are interested in the behavior of the likelihood function for the observed data in comparison to the likelihood function for the perfect data (expected value of the data points).

$$l(\overrightarrow{\mu}|Y, B^c) = -\sum_l \mu_l a_l + \sum_{ij} Y_{ij} \ln \sum_l \mu_l P_{l,i,j}$$
$$\bar{l}(\overrightarrow{\mu}|B^c) = -\sum_l \mu_l a_l + \sum_{ij} \lambda_{ij} \ln \sum_l \mu_l P_{l,i,j}$$

And considering the maximum likelihood estimation for μ we have:

$$l(\hat{\mu}|Y, B^c) = -\sum_l \hat{\mu}_l a_l + \sum_{ij} Y_{ij} \ln \sum_l \hat{\mu}_l P_{l,i,j} = -\sum_l \hat{\mu}_l a_l - MD(Y/M\| \sum_l \hat{\mu}_l P_l) + \sum_{ij} Y_{ij} \ln Y_{ij}/M$$

$$\bar{l}(\bar{\mu}|B^c) = -\sum_l \bar{\mu}_l a_l + \sum_{ij} \lambda_{ij} \ln \sum_l \bar{\mu}_l P_{l,i,j}$$

Hence, the difference of the two likelihoods will be:

$$\begin{aligned} l(\hat{\mu}|Y, B^c) - \bar{l}(\bar{\mu}|B^c) &= -\sum_l (\hat{\mu}_l - \bar{\mu}_l) a_l - MD(Y/M\| \sum_l \hat{\mu}_l P_l) + \sum_{ij} Y_{ij} \ln Y_{ij}/M - \sum_{ij} \lambda_{ij} \ln \sum_l \bar{\mu}_l P_{l,i,j} \\ &= -[M - E\{M\}] - MD(Y/M\| \sum_l \hat{\mu}_l P_l) + MD(Y/M\|\bar{\mu}_l P_l) + X - E\{X\} \\ &= -[M - \bar{M}] + X - E\{X\} - MD(Y/M\| \sum_l \hat{\mu}_l P_l) + MD(Y/M\|\bar{\mu}_l P_l) \end{aligned}$$

Where $X = \sum_{ij} Y_{ij} \ln \sum_l \bar{\mu}_l P_{l,i,j}$ and $M = \sum_l \hat{\mu}_l a_l$.

Lemma 5 *For the estimator $\hat{\mu}$ defined in lemma 4. we have:*

$$|l(\hat{\mu}|Y, B^c) - \bar{l}(\bar{\mu}|B^c)| \leq |M - \bar{M}| + |X - E\{X\}| + M\ D(Y/M\|\bar{\mu}_l P_l)$$

Proof. Because $\hat{\mu}$ is the I-projection of Y/M on the space of μ, then we have the following inequality.

$$D(Y/M\| \sum_l \hat{\mu}_l P_l) \leq D(Y/M\|\bar{\mu}_l P_l)$$

Which gives us:

$$0 \leq D(Y/M\|\bar{\mu}_l P_l) - D(Y/M\| \sum_l \hat{\mu}_l P_l) \leq D(Y/M\|\bar{\mu}_l P_l)$$

Then:

$$|l(\hat{\mu}|Y, B^c) - \bar{l}(\bar{\mu}|B^c)| \leq |M - \bar{M}| + |X - E\{X\}| + M\ D(Y/M\|\bar{\mu}_l P_l)$$

Here we will bound each of the components in the right hand side of above inequality.

Lemma 6 *Using the generalized Chernoff bound we have:*

$$\Pr(\max_B [M - \bar{M}] > \bar{M}\delta) \leq e^{NK \ln 2} (\frac{e^\delta}{(1+\delta)^{(1+\delta)}})^{\bar{M}}$$

$$\Pr(\max_B [M - \bar{M}] < \bar{M}\delta) \leq e^{NK \ln 2} e^{-\bar{M}\delta^2/2}$$

And if $NK = o(M)$ *then* $M - \bar{M} = o(\bar{M})$, *meaning* $\bar{M} = O(M)$ *and* $M - \bar{M} = o(M)$.

Lemma 7 *We have* $c_1\bar{M} < E\{X\} < c_2\bar{M}$ *and using the generalized Chernoff bound we have:*

$$\Pr(\max_B[X - E\{X\}] > c_1\bar{M}\delta) \leq e^{NK\ln 2}(\frac{e^\delta}{(1+\delta)^{(1+\delta)}})^{c_1\bar{M}}$$

$$\Pr(\max_B[X - E\{X\}] < c_2\bar{M}\delta) \leq e^{NK\ln 2}e^{-c_1\bar{M}\delta^2/2}$$

And if $NK = o(M)$ *then* $X - E\{X\} = o(\bar{M}) = o(M)$.

Lemma 8 *Size of the space of possible* $\vec{\hat{\mu}}s$ *is* $\binom{M+k'-1}{k'-1}$. *However this can be bounded by considering the structure of the sets by simply counting the number of solutions to the equation* $z_1 + .. + z_K \geq M$ *where* z_i *is the size of set* i *and* $z_i \leq M$. *Hence,*

$$|Z| \leq M^K - \binom{M+K-1}{K}$$

Lemma 9 *Using method of types and Lemma 8 we have:*

$$\begin{aligned}\Pr(\max_B[M\ D(Y/M\|\bar{\mu}_l P_l)] > M\delta) &\leq e^{NK\ln 2}\binom{M+k'-1}{k'-1}e^{-M\delta} \\ &\leq e^{NK\ln 2}[M^K - \binom{M+K-1}{K}]e^{-M\delta} \leq e^{NK\ln 2}e^{K\ln(M)-M\delta}\end{aligned}$$

And if $K\ln M + NK = o(M)$ *then* $D(Y/M\|\bar{\mu}_l P_l) = o(1)$.

Theorem 2 *Combining the three lemmas 7, 6 and 9 above for the regime* $M = NK^{1+\epsilon}$ *and* $\epsilon > 0$ *is fixed we have* $NK = o(M)$ *and* $M = o(2^N)$ *we have:*

$$\max_B |l(\hat{\mu}|Y, B^c) - \bar{l}(\bar{\mu}|B^c)| = o(M)$$

Which provides consistency of maximum likelihood estimator for $\vec{\mu}$ *given* B^c.

Theorem 3 *For the regimes in which Theorem 2 holds(e.g.* $M = (NK)^{1+\epsilon}$*), we have:*

- $\hat{B}^c \to B^c$ *is consistent using the theorem 1.*
- $(\mu_M|B^c) \to (\mu|B^c)$ *is consistent for every choice of true* B.

Which can result in consistency of $(\hat{B}, \hat{\mu})$.

10.3.2 Accuracy with less than K *basis elements*

The main estimation Algorithm **??** recovers the correct number of basis elements K and the corresponding coefficients μ and basis matrix B, whenever the true weighted network Y is generated from a non expandable basis matrix. Here we quantify the expected loss in reconstruction accuracy if we were to use $\tilde{\mathsf{K}} < \mathsf{K}$ basis elements to reconstruct Y. To this end we introduce a norm for a network Y, and related metrics.

Definition 4 (τ-norm) *Let* $\mathsf{Y} \sim \mathtt{Poisson}(\mathsf{BWB}')$*, where* $\mathsf{W} = \mathtt{diag}(\mu_1 \dots \mu_{\mathsf{K}})$*. Define the statistics* $a_k \equiv \sum_{i=1}^{\mathsf{N}} \mathsf{B}_{ik}$ *for* $k = 1 \dots \mathsf{K}$*. The* τ*-norm of* Y *is defined as* $\tau(\mathsf{Y}) \equiv |\sum_{k=1}^{\mathsf{K}} \mu_k a_k|$*.*

Consider an approximation $\tilde{\mathsf{Y}} = \mathsf{B}\tilde{\mathsf{W}}\mathsf{B}'$ characterized by an index set $\mathsf{E} \subset \{1 \dots \mathsf{K}\}$, which specifies the basis elements to be excluded by setting the corresponding set of coefficients μ_{E} to zero. Its reconstruction error is $\tau(\mathsf{Y} - \tilde{\mathsf{Y}}) = |\sum_{k \notin \mathsf{E}} \mu_k a_k|$, and its reconstruction accuracy is $\tau(\tilde{\mathsf{Y}})/\tau(\mathsf{Y})$. Thus, given a network matrix Y representable exactly with K basis elements, the best approximation with $\tilde{\mathsf{K}}$ basis elements is obtained by zeroing out the lowest $\mathsf{K} - \tilde{\mathsf{K}}$ coefficients μ.

We posit the following theoretical model,

$$\mu_k \sim \mathtt{Gamma}(\alpha + \beta, 1) \tag{10.11}$$

$$a_k/\mathsf{N} \sim \mathtt{Beta}(\alpha, \beta) \tag{10.12}$$

$$\mu_k \cdot a_k/\mathsf{N} \sim \mathtt{Gamma}(\alpha, 1), \tag{10.13}$$

for $k = 1 \dots \tilde{\mathsf{K}}$. This model may be used to compute the expected accuracy of an approximate reconstruction based on $\tilde{\mathsf{K}} < \mathsf{K}$ basis elements, since the magnitude of the ordered μ_k coefficients that are zeroed out are order statistics of a sample of Gamma variates (**?**). Given K and α, we can compute the expected coefficient magnitudes and the overall expected accuracy τ_0.

Theorem 4 *The theoretical accuracy of the best approximate Graphlet decomposition with* $\tilde{\mathsf{K}}$ *out of* K *basis elements is:*

$$\tau_0(\tilde{\mathsf{K}}, \mathsf{K}, \alpha) = \sum_{j=1}^{\tilde{\mathsf{K}}} \frac{f(j, \mathsf{K}, \alpha)}{\alpha \mathsf{K}}, \tag{10.14}$$

where

$$f(j, \mathsf{K}, \alpha) = \binom{\mathsf{K}}{j} \sum_{q=0}^{j-1} (-1)^q \binom{j-1}{q} \frac{f(1, \mathsf{K} - j + q + 1, \alpha)}{\mathsf{K} - j + q + 1} \tag{10.15}$$

$$f(1, \mathsf{K}, \alpha) = \frac{\mathsf{K}}{\Gamma(\alpha)} \sum_{m=0}^{(\alpha-1)(\mathsf{K}-1)} c_m(\alpha, \mathsf{K} - 1) \frac{\Gamma(\alpha + m)}{\mathsf{K}^{\alpha+m}}, \tag{10.16}$$

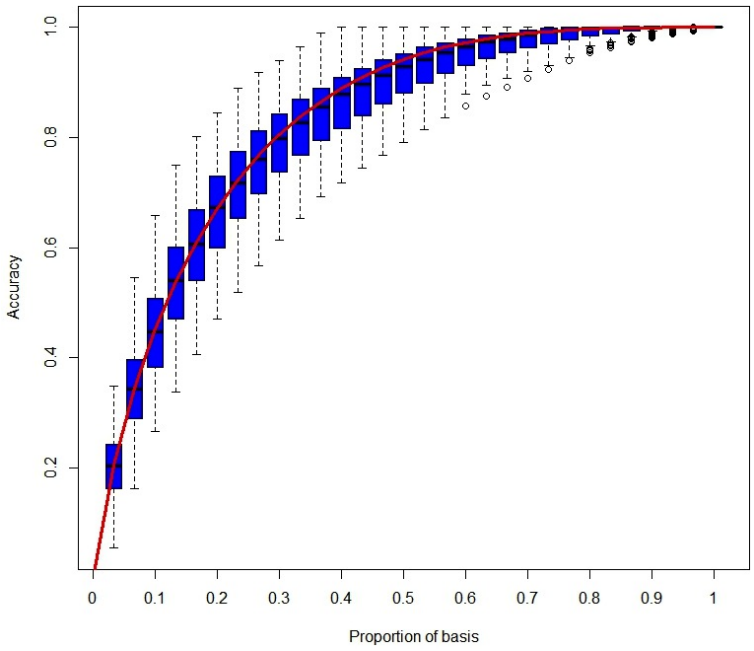


Fig. 10.1 Theoretical and empirical accuracy for different fractions of basis elements $\tilde{K}/K$ with $\alpha = 0.1$. The ratio $\tilde{K}/K$ also provides a measure of sparsity.

in which the coefficients c_m *are defined by the recursion*

$$c_m(\alpha, q) = \sum_{i=0}^{i=\alpha-1} \frac{1}{i!} c_{m-i}(\alpha, q-1) \tag{10.17}$$

with boundary conditions $c_m(\alpha, 1) = \frac{1}{i!}$ *for* $i = 1 \ldots \alpha$.

Figure 10.1 illustrates this result on simulated networks. The solid (red) line is the theoretical accuracy computed for $K = 30$ and $\alpha = 1$, the relevant parameters used to to simulate the sample of 100 weighted networks. The (blue) boxplots summarize the empirical accuracy at a number of distinct values of $\tilde{K}/K$.

10.3.3 A new notion of social information

Graphlet quantifies social information in terms of community structure (i.e., maximal cliques) at multiple scales and possibly overlapping. To illustrate this notion of social information, we simulated 200 networks: 100 Erdös-Rényi-Gilbert random graphs with Poisson weights and 100 networks from

the model described in Section **??**. While, arguably, the Poisson random graphs do not contain any such information, the data generating process underlying graphlets was devised to translate such information into edge weights.

As a baseline for comparison, we consider the singular value decomposition (**?**), applied to the symmetric adjacency matrices with integer entries encoding the weighted networks. The SVD summarizes edge weights using orthogonal eigenvectors ν_i as basis elements and the associated eigenvalues λ_i as weights, $Y_{N\times N} = \sum_{i=1}^{N} \lambda_i \nu_i \nu_i'$. However, SVD basis elements are not interpretable in terms of community structure, thus SVD should not be able to capture the notion of social information we are interested in quantifying.

We applied graphlets and SVD to the two sets of networks we simulated. Figure 10.2 provides an overview of the results. Panels in the top row report results for on the networks simulated from the model underlying graphlets. Panels in the bottom row report results on the Poisson random graphs. In each row, the left panel shows the box plots of the coefficients associated with each of the basis elements, for graphlets in blue and for SVD in red.

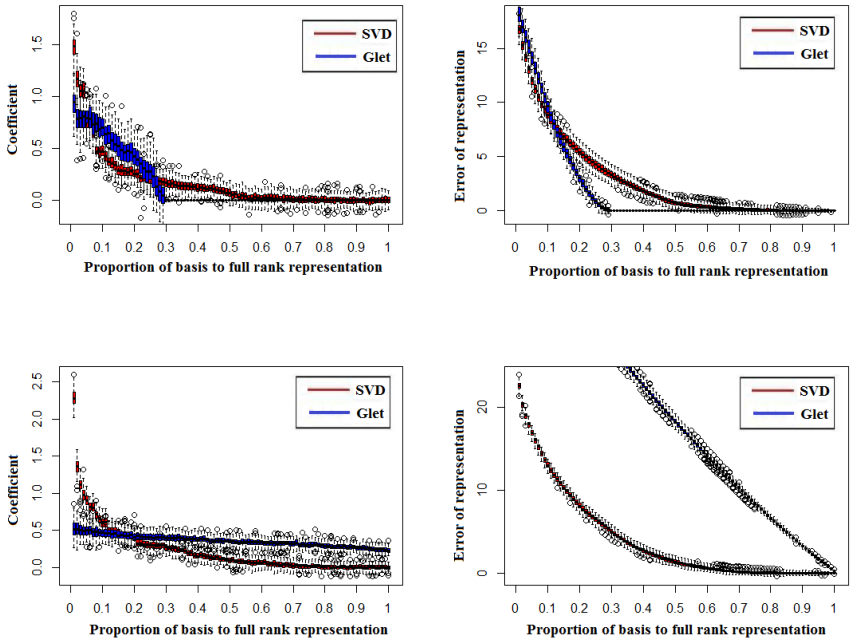


Fig. 10.2 Comparison between graphlet and SVD decompositions. Top panels report results for on the networks simulated from the model underlying graphlets. Bottom panels report results on the Poisson random graphs.

The right panel shows the box plots of the cumulative representation error as a function of the number of basis elements utilized. Graphlets coefficients decay more slowly than SVD coefficients on Poisson graphs (bottom left). Because of this, the error in reconstructing Poisson graphs achievable with graphlets is consistently worse then the error achievable with SVD (bottom right). In contrast, graphlets coefficients decay rapidly to zero on networks with social structure, much sharply then the SVD coefficients (top left). Thus, the reconstruction error achievable with graphlets is consistently better then the error achievable with SVD (top right).

These results support our claim that graphlets is able to distill and quantify a notion of social information in terms of social structure.

10.3.4 Analysis of messaging on Facebook

Here we illustrate graphlet with an application to messaging patterns on Facebook. We analyzed the number of public wall-post on Facebook, over a three month period, among students of a number of US colleges. While our data is new, the US colleges we selected have been previously analyzed (**?**).

Table 10.1 provides a summary of the weighted network data and of the results of the graphlet decomposition. Salient statistics for each college include the number of nodes and edges. The table reports the number of estimated basis elements $\hat{K}$ for each network and the runtime, in seconds. The $\tau(\tilde{Y})$ error incurred by using a graphlet decomposition is reported for different fractions of the estimated optimal number of basis elements $\hat{K}$, ranging from 10% to 100%—no error.

Overall, these results suggest that the compression of wall-posts achievable on collegiate network is substantial. A graphlet decomposition with about 10% of the optimal number of basis elements already leads to a reconstruction error of 10% or less, with a few exceptions. Using 25% of the basis elements further reduces the reconstruction error to below 5%.

Table 10.1 $\tau(\tilde{Y})$ error for different fractions of the estimated optimal number of basis elements $\hat{K}$.

college	nodes	edges	$\hat{K}$	sec	10%	25%	50%	75%	90%	100%
American	6386	435323	11426	151	13.5	6.5	1.80	.70	.30	0
Amherst	2235	181907	10151	124	6.5	2.3	.84	.33	.14	0
Bowdoin	2252	168773	9299	113	9.0	3.2	1.17	.41	.17	0
Brandeis	3898	275133	10340	116	6.8	2.9	1.21	.53	.25	0
Bucknell	3826	317727	13397	193	7.8	2.9	1.15	.45	.20	0
Caltech	769	33311	3735	51	5.7	1.8	.65	.27	.11	0
CMU	6637	499933	11828	169	14.9	5.2	1.89	.73	.34	0
Colgate	3482	310085	12564	151	8.0	3.3	1.26	.45	.19	0
Hamilton	2314	192787	11666	200	6.9	2.5	.84	.33	.15	0
Haverford76	1446	119177	9021	128	4.8	2.2	.74	.25	.10	0
Howard	4047	409699	12773	170	8.6	3.7	1.55	.60	.28	0
Johns Hopkins	5180	373171	11674	150	10.8	3.7	1.40	.58	.29	0
Lehigh	5075	396693	14076	206	9.5	3.2	1.14	.49	.23	0
Michigan	3748	163805	5561	54	11.4	4.6	1.92	.76	.35	0
Middlebury	3075	249219	9971	109	9.7	3.5	1.35	.49	.22	0
MIT	6440	502503	13145	191	11.5	4.7	1.68	.65	.30	0
Oberlin	2920	179823	7862	84	9.8	3.8	1.48	.57	.26	0
Reed	962	37623	3911	46	6.0	2.3	.99	.40	.17	0
Rice	4087	369655	12848	155	8.7	3.2	1.25	.51	.22	0
Rochester	4563	322807	10824	124	11.0	3.7	1.43	.56	.26	0
Santa	3578	303493	11203	127	10.3	3.5	1.30	.51	.24	0
Simmons	1518	65975	5517	60	6.4	2.5	1.04	.44	.19	0
Smith	2970	194265	8591	102	5.5	2.5	1.15	.49	.23	0
Swarthmore	1659	122099	7856	96	6.3	2.6	1.06	.44	.20	0
Trinty	2613	223991	10832	131	8.6	3.0	1.07	.40	.18	0
Tufts	6682	499455	14641	212	13.9	4.8	1.68	.65	.30	0
UC Berkeley	6833	310663	7715	105	16.3	6.2	2.28	.90	.42	0
U Chicago	6591	416205	12326	176	14.2	4.7	1.74	.66	.31	0
Vassar	3068	238321	11344	134	9.3	3.2	1.18	.47	.21	0
Vermont	7324	382441	10030	145	17.5	5.4	2.05	.82	.36	0
USFCA	2682	130503	6735	67	10.5	3.8	1.50	.60	.26	0
Wake Forest	5372	558381	15580	211	11.5	4.2	1.46	.58	.25	0
Wellesley	2970	189797	9768	107	8.9	3.3	1.24	.48	.22	0
Wesleyan	3593	276069	10506	118	9.9	3.6	1.40	.54	.25	0

Chapter 11
Graphons

Exchangeable graph models - DGP - graphon

Identifiability of graphons - Equivalence classes

Approaches to graphon estimation - Empirical performance of multi-stage estimation procedures (Yang, Han, Airoldi) - Consistency ..

Stochastic blockmodel approximation of a graphon

Estimating smooth graphons using total variation

Chapter 12
Graph matching

12.1 Introduction

In practice, it often happens that we need a version of (sub)graph isomorphism that allows for errors. Think about two graphs sampled at different time points, from the same social networks, over the same actors, without knowing the labels of the vertices. We do not expect the two graphs to be isomorphic. Most likely the more recent graph will have extra edges, and maybe also some edges removed. But we certainly expect the graphs to be correlated: the existence of an edge in the old graph is probably a good predictor for the existence of the edge in the second graph.

To analyze both graphs together, however, we need to know the mapping between their vertices, and for this we can rely on the correlated graph structure, which means solving an *inexact* version of the graph isomorphism problem, most often called graph matching. Given two graphs, the graph matching problem seeks to find a correspondence (i.e. "matching") between the vertex sets that best preserves structure across the graphs. The graph matching problem has a rich and active place in the literature with applications in such diverse fields as neuroscience (connectomics), document and image processing, and manifold learning to name a few (see Conte et al (2004)).

Given two graphs G_1 and G_2, with respective adjacency matrices A and B, the simplest version of the graph matching problem is

$$\min_{\mathsf{P}\in\Pi(\mathfrak{n})} \|\mathsf{AP}-\mathsf{PB}\|_{\mathsf{F}}, \tag{12.1}$$

where $\Pi(\mathfrak{n})$ is the set of $\mathfrak{n}\times\mathfrak{n}$ permutation matrices and $\|\cdot\|_{\mathsf{F}}$ is the matrix Frobenius norm. Note that

$$\operatorname{argmin}_{\mathsf{P}\in\Pi(\mathfrak{n})}\|\mathsf{AP}-\mathsf{PB}\|_{\mathsf{F}} = \operatorname{argmin}_{\mathsf{P}\in\Pi(\mathfrak{n})} -\operatorname{trace}(\mathsf{A}^{\mathsf{T}}\mathsf{PBP}^{\mathsf{T}}), \tag{12.2}$$

and the graph matching problem is equivalent to

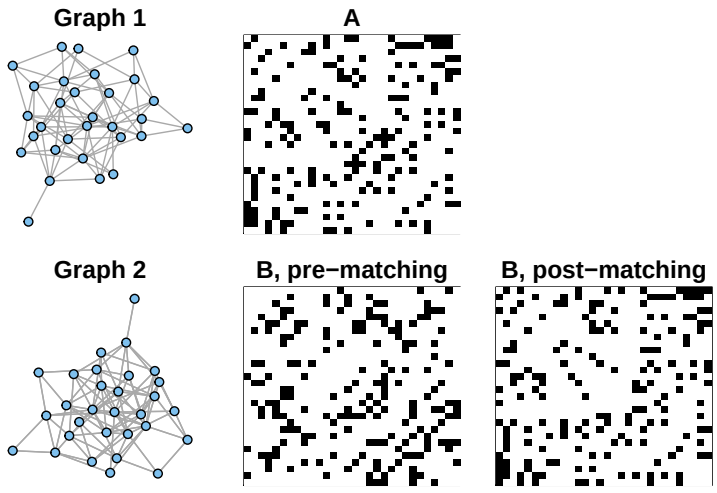


Fig. 12.1 Example of a matching of two non-isomorphic simple 30 vertex graphs G_1 and G_2, with respective adjacency matrices A and B. Each adjacency matrix (A and B pre/post-matched) is visualized as a 30×30 pixel image with a black pixel denoting an edge and a white pixel denoting no edge.

$$\min_{P \in \Pi(n)} -\mathrm{trace}(A^T P B P^T). \tag{12.3}$$

Even though $\|AP - PB\|_F^2$ is a convex quadratic function of P (when the integrality of P is relaxed), we shall see shortly the advantages of considering the nonconvex quadratic alternate formulation $-\mathrm{trace}(A^T P B P^T)$.

The combinatorial nature of this optimization (indeed, we are optimizing over the set of permutation matrices) makes the graph matching problem (NP) hard in its most general form. Many state-of-the-art graph matching algorithms (see, for example, Fiori et al (2013), Vogelstein et al (2012), Zaslavskiy et al (2009)) circumvent the difficulty inherent to optimizing over $\Pi(n)$ by first relaxing the constrained set of permutation matrices to its convex hull $\mathcal{D}(n)$, the set of doubly stochastic matrices, and optimizing instead over $D \in \mathcal{D}(n)$. Once a solution D^* is thus obtained, the algorithms then project it back onto $\Pi(n)$ yielding an approximate solution to the graph matching problem.

The relaxation of (12.1), namely $\min_{D \in \mathcal{D}(n)} \|AD - DB\|_F$, is a convex quadratic program with affine constraints and can therefore be efficiently exactly solved. Combined with a fast projection step, this gives an efficient approximate solution to (12.1). On the other hand, the relaxation of (12.3) is a nonconvex quadratic program with affine constraints, and nonconvex quadratic optimization is NP-hard in general. Indeed, it is often the case that the objective function $-\mathrm{trace}(A^T D B D^T)$ has numerous local minima in $\mathcal{D}(n)$. We will see shortly that the nonconvex relaxation is, theoretically, the *right*

relaxation, and moreover, there is an efficient algorithm for approximating its solution, the FAQ algorithm of Vogelstein et al (2012).

12.2 A model for matching

A convenient theoretical model for exploring the graph matching problem in the correlated Bernoulli graph model of Lyzinski et al (2014a). This model allows us to generate two graphs with a natural vertex alignment against which we can measure the performance of our graph matching algorithms.

Definition 12.1. Given parameters $n \in \mathbb{Z}^+$, a real number $\rho \in [0,1]$, and a symmetric hollow matrix $L \in [0,1]^{n \times n}$, we say that two random graphs, with respective adjacency matrices A and B, are ρ-*correlated Bernoulli*(L) distributed if for all $i = 1, 2, \ldots, n-1$, and $j = i+1, i+2, \ldots, n$, the random variables (matrix entries) $A_{i,j}, B_{i,j}$ are Bernoulli($L_{i,j}$) distributed, and all of these random variables are collectively independent except that, for each pair $\{i, j\}$, the Pearson product-moment correlation coefficient for $A_{i,j}, B_{i,j}$ is ρ.

It is straightforward to show that two ρ-correlated Bernoulli(L) graphs may be realized by first, for all ordered pairs $\{i, j\}$, having $B_{ij} \sim$ Bernoulli($L_{i,j}$) independently drawn and then, conditioning on B, have

$$A_{i,j} \sim \text{Bernoulli}\left((1-\rho)L_{i,j} + \rho B_{i,j}\right) \tag{12.4}$$

independently drawn.

The `sample_bernoulli()` igraph function creates a single Bernoulli(L) graph. `sample_correlated_bernoulli()` then takes this graph and creates its correlated pair, with the desired correlation level. Assuming $n = 100$ vertices and a connection probability of $L_{i,j} = ij/10000$ (except that $L_{i,i} = 0$), we write

sample_bernoulli()
sample_correlated_bernoulli()

```
12.1.1) L <- outer(1:100, 1:100, function(i, j) ifelse(i == j, 0, i * j / 10000))
    2) g1 <- sample_bernoulli(L, directed = FALSE)
    3) g2 <- sample_correlated_bernoulli(g1, corr = 0.9, L = L)
```

12.2.1 It doesn't always pay to relax

Two graphs drawn from the ρ-correlated Bernoulli(L) model have a natural alignment of their vertices given by the identity function. How well can the graph matching problem (and its associated relaxations) estimate this alignment? Let A and B be the respective adjacency matrices of two ρ-correlated Bernoulli(L) graphs. Under mild conditions on ρ and L, with high probability it holds that

$$\arg\min_{D\in\mathcal{D}} -\text{trace}(A^T DBD^T) = \arg\min_{P\in\Pi(n)} \|AP - PB\|_F = \{I_n\}, \qquad (12.5)$$

and yet $I_n \notin \arg\min_{D\in\mathcal{D}} \|AD - DB\|_F$ (see (Lyzinski et al, 2014a, Theorem 1)). On the one hand, the computationally difficult nonconvex relaxation yields the true alignment (with high probability), and on the other hand the true alignment is not the solution of the easily solved convex relaxation (with high probability). Complicating things further, often the doubly stochastic solution of the convex relaxation is far from the true alignment, and the true alignment is then not recovered via the projection step.

As the nonconvex objective function is quadratic subject to affine constraints, Frank-Wolfe methodology (Frank and Wolfe (1956)) provides a fast algorithm for approximately solving the relaxation. This is implemented in the FAQ algorithm of Vogelstein et al (2012), a fast and efficient approximate graph matching procedure.

12.3 The FAQ algorithm

We begin with two n-vertex graphs G_1 and G_2, and let their respective adjacency matrices be A and B. The FAQ algorithm proceeds as follows:

1. Relax $\min_{P\in\Pi(n)} -\text{trace}(A^T PBP^T)$ to $\min_{D\in\mathcal{D}(n)} -\text{trace}(A^T DBD^T)$;

2. Run the Frank-Wolfe algorithm on $f(D) := -\text{trace}(A^T DBD^T)$:

- **i.** Initialize $D_0 = \vec{1}_n \cdot \vec{1}_n^T/n$;
- **ii.** If at D_i, calculate $\nabla f(D_i) = -AD_iB^T - A^T D_i B$;
- **iii.** Set $Q_i := \min_{D\in\mathcal{D}(n)} \text{trace}(\nabla f(D_i)^T D)$;
- **iv.** Set $\alpha_i := \min_{\alpha\in[0,1]} f(\alpha D_i + (1-\alpha)Q_i)$;
- **v.** Update $D_{i+1} = \alpha_i D_i + (1-\alpha_i)Q_i$;
- **vi.** Repeat **ii.-v.** until some stopping criterion is met;

3. Set $P^* := \min_{P\in\Pi(n)} - \text{trace}(D_{\text{final}}^T P)$; Output P^*.

Each of steps **iii.** and **3.** in the FAQ algorithm amounts to solving a linear assignment problem, which can be done in $O(n^3)$ steps using the Hungarian algorithm of Kuhn (1955). With a bounded number of Frank-Wolfe iterates (in practice, excellent performance is usually achieved with ≤ 30 iterates), the FAQ algorithm then has running time $O(n^3)$.

Given two n vertex graphs we can run the FAQ algorithm in igraph with the `match_vertices()` function:

match_vertices()

```
12.2.1) g1_g2 <- match_vertices(g1, g2, num_iter = 30)
    2) g1_g2$match
```

For each vertex in `g1`, the result gives the corresponding vertex in `g2`.

12.4 Seeding

The nonconvex objective function $\operatorname{trace}(A^T DBD^T)$ can have many local minima in $\mathcal{D}(n)$, making the optimization problem (12.3) exceedingly challenging. One way to ameliorate this difficulty is to introduce seeded vertices, i.e. vertices whose latent alignment function is known across graphs a priori. The seeded graph matching problem leverages the information contained in these seeded vertices to efficiently recover the latent alignment among the unseeded vertices. One of the advantages of the FAQ algorithm is its amenability to seeding, and in Fishkind et al (2012) and Lyzinski et al (2014b), the authors demonstrate the potential for significantly improved performance in FAQ by incorporating even a modest number of seeded vertices.

Given two graphs with respective adjacency matrices A and B, we define a seeding function $\phi : S_1 \mapsto S_2$ to be a bijective function between two seed sets, $S_1 \subset V(G_1)$ and $S_2 \subset V(G_2)$ (without loss of generality, we will assume $S_1 = S_2 = \{1, 2, \ldots, s\}$. Partition A and B via

$$A = \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{21}^T \\ B_{21} & B_{22} \end{bmatrix}$$

where $A_{11}, B_{11} \in \{0,1\}^{s\times s}$, $A_{22}, B_{22} \in \{0,1\}^{m\times m}$, and $A_{21}, B_{21} \in \{0,1\}^{m\times s}$. The *seeded graph matching problem* is then

$$\min_{P\in\Pi(m)} \|A(I_s \oplus P) - (I_s \oplus P)B\|_F^2. \tag{12.6}$$

This is equivalent to minimizing the nonconvex quadratic function

$$-\operatorname{trace}\left(A_{21}^T PB_{21} + A_{12}^T B_{12} P^T + A_{22}^T PB_{22} P^T\right) \tag{12.7}$$

over all $m \times m$ permutation matrices P.

The seeded FAQ algorithm, the SGM algorithm of Fishkind et al (2012) and Lyzinski et al (2014b), proceeds in much the same way as the unseeded version:

1. Relax the permutation constraint to

$$\min_{D\in\mathcal{D}(m)} -\operatorname{trace}\left(A_{21}^T DB_{21} + A_{12}^T B_{12} D^T + A_{22}^T DB_{22} D^T\right), \tag{12.8}$$

calling this objective function $g(D)$

2. Run the Frank-Wolfe algorithm on $g(D)$:

i. Initialize $D_0 = \vec{1}_m \cdot \vec{1}_m^T / m$;

ii. If at D_i, calculate

$$\nabla f(D_i) = -A_{21}B_{21}^T - A_{12}^T B_{12} - A_{22}PB_{22}^T - A_{22}^T PB_{22}; \tag{12.9}$$

iii. Set $Q_i := \min_{D \in \mathcal{D}(m)} \operatorname{trace}(\nabla f(D_i)^T D)$;
iv. Set $\alpha_i := \min_{\alpha \in [0,1]} g(\alpha D_i + (1-\alpha) Q_i)$;
v. Update $D_{i+1} = \alpha_i D_i + (1-\alpha_i) Q_i$;
vi. Repeat **ii.-v.** until some stopping criterion is met;
3. Set $P^* := \min_{P \in \Pi(m)} -\operatorname{trace}(D_{\text{final}}^T P)$; Output $I_s \oplus P^*$, an approximate solution to (12.6).

As with the FAQ algorithm, steps **iii.** and **3.** are equivalent to linear assignment problems and can be solved in $O(n^3)$ time. Given a bounded number of Frank-Wolfe steps, the SGM algorithm then has $O(n^3)$ running time. In igraph, the seeds can be given using the optional `'seeds1'` and `'seeds2'` arguments to `match_vertices()`. If the seeds are the first n_s vertices in both graphs, then the simpler `'num_seeds'` argument can also be used.

```
12.3.1) g1_g2_5s <- match_vertices(g1, g2, num_seeds = 5, num_iter = 30)
    2) g1_g2_5s$match
```

12.5 The effect of seeding

The incorporation of only a few seeds into the FAQ algorithm has a profound effect on the algorithm's performance. Decomposing the SGM objective function, we have

$$-\operatorname{trace}\left(A_{21}^T P B_{21} + A_{12}^T B_{12} P^T + A_{22}^T P B_{22} P^T\right), \tag{12.10}$$

where the first two terms are linear in P, and contain all the adjacency information between the seeded and nonseeded vertices. The last term is nothing more than the standard graph matching problem for the unseeded vertices. Minimizing

$$-\operatorname{trace}(A_{21}^T P B_{21} + A_{12}^T B_{12} P^T) \tag{12.11}$$

over $P \in \Pi(m)$ is a simple linear assignment problem, hence can be solved in $O(n^3)$ time, and a logarithmic number of seeded vertices are sufficient (and necessary) for this linear subproblem to yield the true alignment between the unseeded vertices with high probability (Lyzinski et al (2014b)). This seeded term has the effect of "steering" the graph matching problem in the right direction!

We demonstrate this in the next example. We match two 0.9-correlated $G_{np}(100, 0.7)$ graphs, with $n_r = 100$ replicates. We match the two adjacency matrices, using $s = 0, 1, 2, 3$ seeds.

```
12.4.1) num_seeds <- 0:3
    2) num_reps <- 100
    3) num_runs <- length(num_seeds) * num_reps
    4) correct <- data.frame(Seeds = rep(NA_integer_, num_runs),
    5)                       Correct = rep(NA_integer_, num_runs))
```

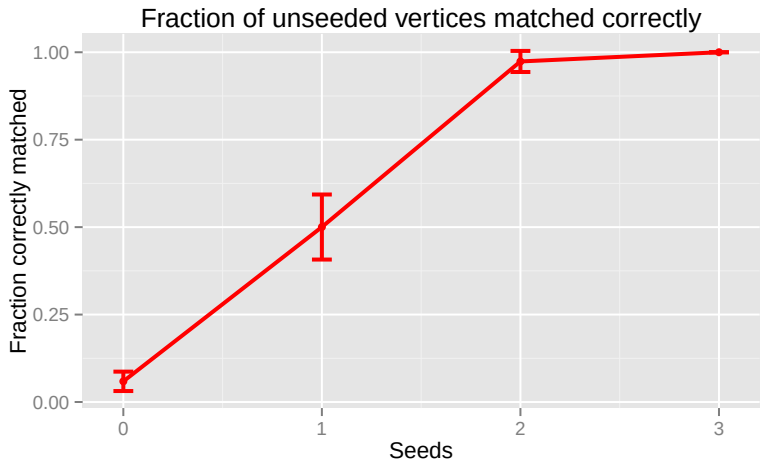


Fig. 12.2 Matching 0.9-correlated $G_{np}(100, 0.7)$ graphs. For each of 100 replicates, we calculate the number of unseeded vertices correctly matched when utilizing $s = 0, 1, 2, 3$ seeds. We plot mean accuracy $\pm$ 2 standard error versus the seed values.

```
idx <- 1
for (i in seq_len(num_reps)) {
  gnp_pair <- sample_correlated_gnp_pair(100, p = 0.7, corr = 0.9)
  for (j in num_seeds) {
    correct$Seeds[idx] <- j
    correct$Correct[idx] <-
      match_vertices(gnp_pair[[1]], gnp_pair[[2]], num_seeds = j)$match %>%
      equals(V(gnp_pair)[[2]]) %>%
      sum() %>%
      subtract(j) %>%
      divide_by(gorder(gnp_pair[[1]]) - j)
    idx <- idx + 1
  }
}
correct %>%
  dplyr::group_by(Seeds) %>%
  dplyr::summarize(mean(Correct), sd(Correct))
```

Note that the true alignment of G_1 and G_2 is given by the identity map. We see a dramatic performance increase by incorporating even a few seeds. The experiment is summarized in Figure 12.2, where we dramatically increased performance when utilizing even a single seed!

As the correlation between the graphs decreases, the impact of each seed is lessened. This is illustrated beautifully in the next example. For each of $n_r = 100$ replicates, we match two ρ-correlated Bernoulli(L) graphs using

$n_s = 0, 1, 2, 5, 10, 15, 20$ seeds for $\rho = 0, 0.3, 0.6, 0.9$. For each replicate and each value of ρ, we sample L uniformly from the space of symmetric hollow matrices in $[0, 1]^{100 \times 100}$.

```
num_seeds <- c(0, 1, 2, 5, 10, 15, 20)
num_reps <- 100
corr <- c(0, 0.3, 0.6, 0.9)
num_runs <- length(num_seeds) * length(corr) * num_reps
correct2 <- data.frame(Corr = rep(NA_real_, num_runs),
                       Seeds = rep(NA_integer_, num_runs),
                       Correct = rep(NA_integer_, num_runs))
idx <- 1
for (i in seq_len(num_reps))  {
  for (c in corr) {
    gnp_pair <- sample_correlated_gnp_pair(100, p = 0.7, corr = c)
    for (j in num_seeds) {
      correct2$Corr[idx] <- c
      correct2$Seeds[idx] <- j
      correct2$Correct[idx] <-
        match_vertices(gnp_pair[[1]], gnp_pair[[2]], num_seeds = j) %>%
          equals(V(gnp_pair)[[2]]) %>%
          sum() %>%
          subtract(j) %>%
          divide_by(gorder(gnp_pair[[1]]) - j)
      idx <- idx + 1
    }
  }
}
```

The example is summarized in Figure 12.3. From the figure, we see that SGM performs little better than chance if there are no seeds. In general, it takes more seeds to achieve commensurate performance when the correlation is decreased. This is quite sensible, as the information contained in each seed is lessened as ρ decreases. We generate the figure via

```
ggplot(correct2, aes(x = Seeds, y = Correct, colour = Corr)) +
  geom_errorbar(aes(ymin = ymin, ymax = ymax)) +
  geom_line() +
  geom_point() +
  labs(x = "Seeds", y = "Fraction correctly matched",
       title = "Effect of seeding vs correlation strength")
```

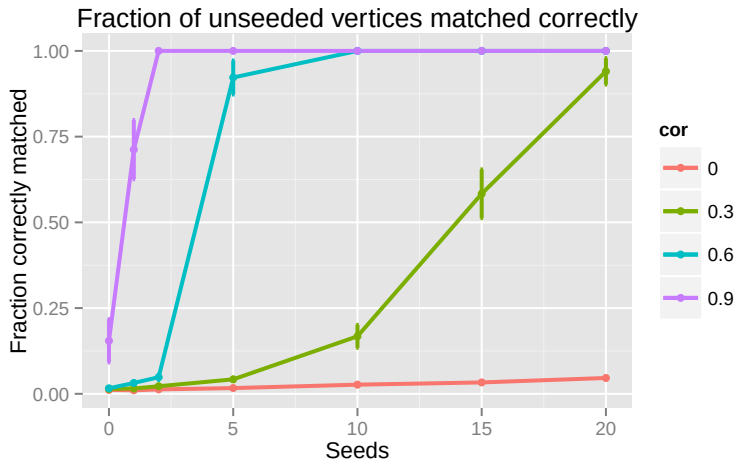


Fig. 12.3 Matching ρ-correlated Bernoulli(L) graphs using $s = 0, 1, 2, 5, 10, 15, 20$ seeds for $\rho = 0, 0.3, 0.6, 0.9$. For each of $nmc = 100$ replicates, and for each value of ρ, we sample L uniformly from the space of symmetric hollow matrices in $[0, 1]^{100 \times 100}$. We calculate the number of unseeded vertices correctly matched, and plot mean accuracy $\pm$ 2 s.e. versus the seed values.

12.6 Worms with brains: better than brains with worms

To demonstrate the effectiveness of the SGM procedure, we will walk through a real data application. Biologists currently believe that the nervous system of the Caenorhabditis elegans (abbreviated C. elegans) roundworm contains 302 labeled neurons, the same 302 for each organism. Twenty-three of the neurons are isolates, not making synapses with any other neurons, leaving 279 neurons with synaptic connections to other neurons (see Varshney et al (2011) for detail on how these nervous systems were mapped). Within the C. elegans connectome, there are two types of synaptic connections: chemical (chemical synapses) and electrical (junction potentials). How much of the synaptic structure of the nervous system is preserved across graphs? To understand this, we run our SGM algorithm to match the two connectomes (the chemical G_c and the electrical G_e). We will drop the isolates from the graphs, as these cannot be matched based on the graph structure.

```
12.7.1) data(celegans)
    2) celegans %>% summary()
    3) g_c <- subgraph(celegans, E(celegans)[type == "chemical"], FALSE)
    4) g_e <- subgraph(celegans, E(celegans)[type == "electric"], FALSE)
```

We match the two graphs with $s = 0, 30, 60, 90, 120, 150, 180, 210$ seeds. For each of $n_r = 50$ replicates, we choose the seeds uniformly at random

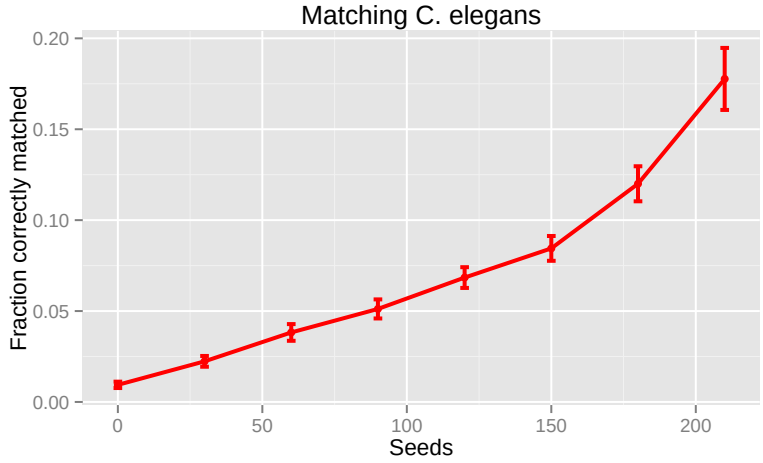


Fig. 12.4 Matching the electrical and chemical C. elegans connectomes unsing $s = 0, 30, 60, 90, 120, 150, 180, 210$ seeds. For each of 50 replicates, we choose the seeds uniformly at random from the 253 vertices.

from the 253 vertices (using nested seed sets for each trial). We summarize the results below in Figure 12.4.

```
12.8.1) num_seeds <- seq(0, 210, by = 30)
    2) num_reps <- 50
    3) num_runx <- length(num_seeds) * num_reps
    4) ce_match <- data.frame(Replicate = rep(NA_integer_, num_runs),
    5)                        Seeds = rep(NA_integer_, num_runs),
    6)                        Correct = rep(NA_integer_, num_runs))
    7) idx <- 1
    8) for (i in seq_len(num_reps)) {
    9)   all_seeds <- sample(gorder(g_c), max(num_seeds))
   10)   for (j in num_seeds) {
   11)     seeds <- all_seeds[seq_len(j)]
   12)     ce_match$Replicate <- i
   13)     ce_match$Seeds[idx] <- j
   14)     ce_match$Correct[idx] <-
   15)       match_vertices(g_c, g_e, seeds = seeds) %>%
   16)       equals(V(g_e)) %>%
   17)       sum() %>%
   18)       subtract(j) %>%
   19)       divide_by(gorder(g_e) - j)
   20)     idx <- idx + 1
   21)   }
   22) }
```

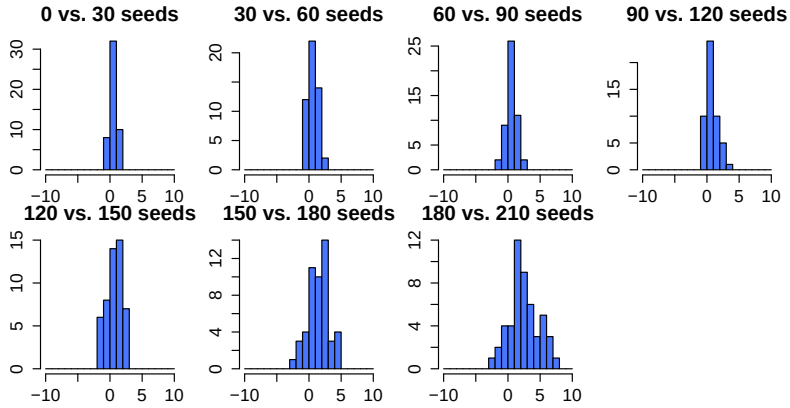


Fig. 12.5 The performance increase due to the inclusion of additional seeds when matching the C. elegans connectomes. For each of the 50 replicates, we plot a bar graph showing how many extra vertices are correclt matched by including 30 more seeds for each of $s = 0, 30, 60, 90, 120, 150, 180$.

```
ggplot(ce_match, aes(x = Seeds, y = Correct)) +
  geom_errorbar(aes(ymin = ymin, ymax = ymax)) +
  geom_line() +
  geom_point() +
  labs(x = "Seeds", y = "Fraction correctly matched",
       title = "Effect of seeding vs correlation strength")
```

The results have a profound impact of subsequent graph inference. Had the matching been perfect across graphs, even with moderate seed levels, then one need only to consider one (it doesn't matter which) of the graphs to pursue inference on. Had the matching been chance across seed levels, then joint inference should proceed by investigating each of the two graphs separately. As the matching is significantly better than chance, and significantly less than perfect, it follows that joint inference on the connectomes need proceed in the *joint* space considering both graphs simultaneously.

How much does the inclusion of additional seeds impact matching performance? In the simulated data examples, incorporating seeds induced a dramatic jump, with a single seed causing a poorly performing matching to be perfect. Here, the result is much less dramatic, as we see a very gradual performance increase when incorporating more seeds. Curiously, we also see that the performance can actually *decrease* when incorporating more seeds! This points to the primacy of intelligently selecting seeded vertices in real data applications. Poorly chosen seeds can actually hurt subsequent matching performance, while well-chosen seeds can significantly improve performance.

12.7 SGM extensions and modifications

In the remainder of this chapter, we will provide some examples of extensions and modifications of the SGM algorithm.

12.7.1 Matching graphs of different orders

In presenting the SGM algorithm so far, we have focused on graphs of the same order. What if we have two graphs of similar but not quite equal order, can we also match them? The answer is yes, with some ingenuity. Consider matching two stochastic block model random graphs from the same model with differing orders (i.e. on different sized vertex sets):

```
12.9.1) B <- matrix(c(0.7, 0.3, 0.4,
    2)              0.3, 0.7, 0.3,
    3)              0.4, 0.3, 0.7), 3, byrow = TRUE)
    4) g1 <- sample_sbm(300, B, c(100, 100, 100))
    5) g2 <- sample_sbm(225, B, c( 75,  75,  75))
```

One, perhaps naive, solution is to augment G_2 with dummy isolate vertices to make G_1 and G_2 commensurate orders, i.e. pad the adjacency matrix of G_2 with zeros until it is the same size as the adjacency matrix of G_1. We call this augmented graph G_2^a.

```
12.10.1) g2_a <- g2 + (gorder(g1) - gorder(g2))
    2) aug_match <- match_vertices(g1, g2_a)$match
```

We see from Figure 12.6 that the matching bewteen G_1 and G_2^a is very poor! By padding G_2 with isolate vertices, and then using FAQ to match G_1 and G_2^a, we are matching G_2 to the best fitting subgraph (*not* the best fitting induced subgraph) of G_1. There is no penalty for non-edges that are matched to edges or edges that are matched to non-edges. There is only a reward for edges matched to edges.

To rectify this, we can use weighted graphs. We first make G_1 and G_2 a complete graph by adding the edges of their complementers with weight -1. Then we pad G_2 with isolate vertices as before and proceed with the matching. This will have the effect of rewarding true non-edges of G_2 being matched to non-edges in G_1, rewarding edges of G_2 being matched to edges in G_1, and penalizing true non-edges of G_2 being matched to edges in G_1 and edges of G_2 being matched to non-edges in G_1. In essence, we are matching G_2 to the best fitting induced subgraph of G_1. Note, from Figure 12.7, the dramatically improved matching performance!

```
12.11.1) E(g1)$weight <- 1
    2) g1_comp <- g1 %>%
```

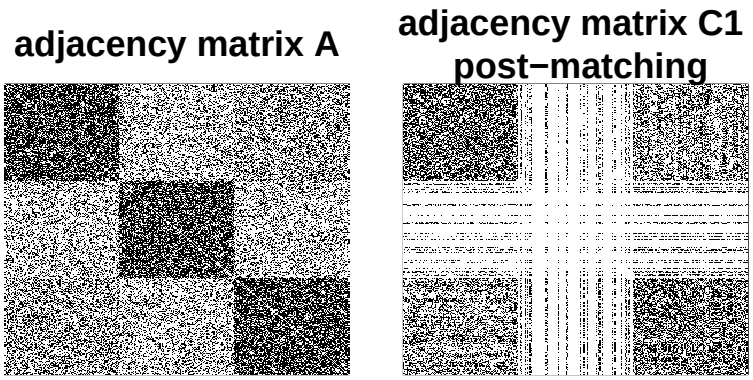


Fig. 12.6 Matching two SBM random graphs of different order from the same model. We pad the smaller adjacency graph (G_2) with isolate vertices to make it commensurate in size with G_1, yielding G_2^a. We plot the adjacency matrix of G_1 and post-matching G_2^a as 300×300 heatmaps. A black/white pixel in position i, j denotes an edge/no edge between vertices i and j.

```
  complementer() %>%
  set_edge_attr("weight", value = -1)
E(g2)$weight <- 1
g2_comp <- g2 %>%
  complementer() %>%
  set_edge_attr("weight", value = -1)
g1_new <- g1 + g1_comp
g2_new <- g2 + g2_comp
g2_new_a <- g2_new + (gorder(g1_new) - gorder(gw_new))
smart_aug_match <- match_vertices(g1_new, g2_new_a)$match
```

12.7.2 Matching more than two graphs

In this section, we show how SGM can be modified to match multiple graphs simultaneously. The modified algorithm, takes as its input a list of graphs to be matched and seed vertices, which are assumed to be the same in all graphs. Suppose we have 3 $G_{np}(200, 0.6)$ graphs. Suppose G_1 and G_2 have pairwise correlation $\rho = 0.8$ and G_2 and G_3 have pairwise correlation $\rho = 0.8$ (think time series!). We run the matching with seeds $s = 0, 1, 2, 3$, and show the results below.

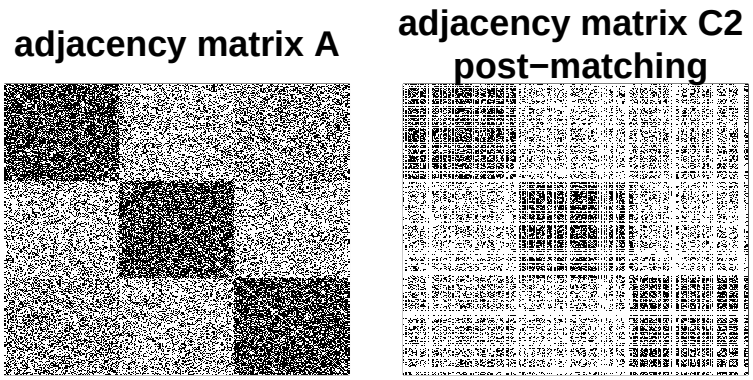


Fig. 12.7 Matching two SBM random graphs of different order from the same model. We change all none-edges in the two graph to edges with weight -1 and then pad the smaller graph (G_2^*) with isolate vertices to make it commensurate in size with G_1^*. We plot the adjacency matrices of G_2^* and post-matching G_2^* as 300×300 heatmaps. A black/white pixel in position i, j denotes an edge/no edge between vertices i and j.

```
12.12.1) g1 <- sample_gnp(100, 0.7)
     2) g2 <- sample_correlated_gnp(g1, 0.8)
     3) g3 <- sample_correlated_gnp(g2, 0.8)
```

We next match these three graphs simultaneously using $s = 0, 1, 2, 3$ seeds.

```
12.13.1) multi <- lapply(0:3, function(num_seeds) {
     2)   match_vertices(g1, g2, g3, num_seeds = num_seeds)
     3) })
```

The output of `match_vertices()` is a list of vertex sequences, with the mappings from the vertices of the first graph, to the vertices of the second, third, etc. graph.

Note that with $s = 3$ seeds, the graphs are perfectly matched and for $s = 2$ (and hence $s = 0$) seeds they are not.

```
12.14.1) multi[[3]][[1]] %>% equals(V(g2)) %>% sum()
     2) multi[[3]][[2]] %>% equals(V(g3)) %>% sum()
     3) multi[[4]][[1]] %>% equals(V(g2)) %>% sum()
     4) multi[[4]][[2]] %>% equals(V(g3)) %>% sum()
```

SGM, even with only 3 seeds across graphs, was able to correctly align the graphs!

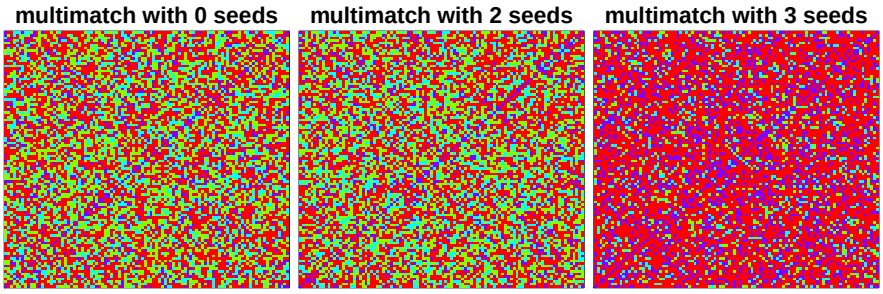


Fig. 12.8 Multimatching the three graphs G_1, G_2 and G_3. Each figure represents a 100×100 heatmap with i, j entry counting the number of times edge i, j appeared as an edge in each graph (after matching): red=3, green=2, light blue=1, blue=0.

12.8 Exercises

► EXERCISE 12.1. A graph is a common induced subgraph of two graphs, if it is an induced subgraph of both of them. It is the maximal common induced subgraph, if it has the most vertices among all common induced subgraphs. At the time of writing, igraph does not have a function to calculate the maximum common subraph(s) of a pair of graphs. Write a function in R that finds the maximum common subgraphs. (Hint: you can use the modular product of the two input graphs, and the fact that the largest cliques in the modular product correspond to the maximum common induced subgraphs of the original input graphs.)

References

Adai AT, Date SV, Wieland S, Marcotte EM (2004) LGL: Creating a map of protein function with an algorithm for visualizing very large biological networks. Journal of Molecular Biology 340:179–190

Adamic LA, Glance N (2005) The political blogosphere and the 2004 us election: divided they blog. In: Proceedings of the 3rd international workshop on Link discovery, ACM, pp 36–43

Amini AA, Chen A, Bickel PJ, Levina E (2013) Pseudo-likelihood methods for community detection in large sparse networks. Ann Statist 41(4):2097–2122, DOI 10.1214/13-AOS1138, URL http://dx.doi.org/10.1214/13-AOS1138

Anderson RM, May RM (1992) Infectious Diseases of Humans: Dynamics and Control. Oxford University Press

Arenas A, Fernández A, Gómez S (2008) Analysis of the structure of complex networks at different resolution levels. New Journal of Physics 10:053,039

Athreya A, Lyzinski V, Marchette DJ, Priebe CE, Sussman DL, Tang M (2013) A limit theorem for scaled eigenvectors of random dot product graphs. arXiv preprint arXiv:13057388

Bailey NTJ (1975) The Mathematical Theory of Infectious Diseases. Griffin

Barabási AL, Albert R (1999) Emergence of scaling in random networks. Science 286:509–512

Blondel V, Guillaume JL, Lambiotte R, Lefebvre E (2008) Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment p P10008

Borg I, Groenen P (2005) Modern Multidimensional Scaling: Theory and Applications, 2nd edn. Springer–Verlag New York

Brandes U (2001) A faster algorithm for betweenness centrality. Journal of Mathematical Sociology 25(2):163–177

Brandes U, Delling D, Gaertler M, Görke R, Hoefer M, Nikoloski Z, Wagner D (2008) On modularity clustering. IEEE Transactions on Knowledge and Data Engineering 20(2):172–188

Castellano C, Pastor-Satorras R (2010) Thresholds for epidemic spreading in networks. Physical Review Letters 105:218,701

Chaudhuri K, Chung F, Tsiatas A (2012) Spectral clustering of graphs with general degrees in the extended planted partition model. Journal of Machine Learning Research 2012:1–23

Chung FRK (1997) Spectral graph theory, CBMS Regional Conference Series in Mathematics, vol 92. Published for the Conference Board of the Mathematical Sciences, Washington, DC; by the American Mathematical Society, Providence, RI

Clauset A, Newman MEJ, Moore C (2004) Finding community structure in very large networks. Physical Review E 70:066,111

Cohen R, Havlin S, ben Avraham D (2003) Efficient immunization strategies for computer networks and populations. Physical Review Letters 91:247,901

Colizza V, Pastor-Satorras R, Vespignani A (2007) Reaction-diffusion processes and metapopulation models in heterogeneous networks. Nature Physics 3:276–282

Conte D, Foggia P, Sansone C, Vento M (2004) Thirty years of graph matching in pattern recognition. International Journal of Pattern Recognition and Artificial Intelligence 18(03):265–298

Daley DJ, Gani J (2001) Epidemic Modeling: An Introduction. Cambridge University Press

Danon L, Díaz-Guilera A, Duch J, Arenas A (2005) Comparing community structure identification. Journal of Statistical Mechanics: Theory and Experiment p P09008

Dhillon IS (2001) Co-clustering documents and words using bipartite spectral graph partitioning. In: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp 269–274

van Dongen S (2000) Performance criteria for graph clustering and markov cluster experiments. Tech. Rep. INS-R0012, National Research Institute for Mathematics and Computer Science in the Netherlands

van Dongen S (2008) Graph clustering via a discrete uncoupling process. SIAM J Matrix Anal Appl 30:121–141

Erdős P, Rényi A (1959) On random graphs. Publicationes Mathematicae 6:290

Evans T, Lambiotte R (2009) Line graphs, link partitions and overlapping communities. Physical Review E 80:016,105

Fiori M, Sprechmann P, Vogelstein J, Musé P, Sapiro G (2013) Robust multimodal graph matching: Sparse coding meets graph matching. Advances in Neural Information Processing Systems 26 pp 127–135

Fishkind D, Adali S, Priebe C (2012) Seeded graph matching. arXiv preprint arXiv:12090367

Fortunato S (2010) Community detection in graphs. Physics Reports 486:75–174

Fortunato S, Barthélemy M (2007) Resolution limit in community detection. Proceedings of the National Academy of Sciences of the United States of America 104(1):36–41

Frank M, Wolfe P (1956) An algorithm for quadratic programming. Naval Research Logistics Quarterly 3(1-2):95–110

Fred ALN, Jain AK (2003) Robust data clustering. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, IEEE, vol 2, pp 128–133, DOI 10.1109/CVPR.2003.1211462

Fruchterman E, Reingold E (1991) Graph drawing by force-directed placement. Software – Practice and Experience 21(11):1129–1164

Galasyn J (2010) Enron chronology. http://www.desdemonadespair.net/2010/09/bushenron-chronology.html, online source

Girvan M, Newman M (2002) Community structure in social and biological networks. Proceedings of the National Academy of Sciences of the United States of America 99(12):7821–6

Good B, de Montjoye Y, Clauset A (2010) Performance of modularity maximization in practical contexts. Physical Review E 81:046,106

Gray WR, Bogovic JA, Vogelstein JT, Landman BA, Prince JL, Vogelstein R (2012) Magnetic resonance connectome automated pipeline: an overview. Pulse, IEEE 3(2):42–48

Gregory S (2007) An algorithm to find overlapping community structure in networks. In: Proceedings of the 11th European Conference on Principles and Practice of Knowledge Discovery in Databases, Lecture Notes in Computer Science, vol 4702, pp 91–102

Gregory S (2010) Finding overlapping communities in networks by label propagation. New Journal of Physics 12:103,018

Guimerà R, Sales-Pardo M, Amaral LAN (2004) Modularity from fluctuations in random graphs and complex networks. Physical Review E 70:025,101

Holland PW, Laskey K, Leinhardt S (1983) Stochastic blockmodels: First steps. Social Networks 5(2):109–137

Holton DA, Sheehan J (1993) The Petersen graph. In: Australian Mathematical Society Lecture Notes, vol 7, Cambridge University Press

Horn RA, Johnson CR (2013) Matrix analysis, 2nd edn. Cambridge University Press, Cambridge

Hubert L, Arabie P (1985) Comparing partitions. Journal of Classification 2:193–218

Kamada T, Kawai S (1989) An algorithm for drawing general undirected graphs. Information Processing Letters 31(1):7–15

Karrer B, Newman MEJ (2011) Stochastic blockmodels and community structure in networks. Phys Rev E (3) 83(1):016,107, 10, DOI 10.1103/PhysRevE.83.016107, URL http://dx.doi.org/10.1103/PhysRevE.83.016107

Keeling MJ, Eames KTD (2005) Networks and epidemic models. Journal of the Royal Society Interface 2(4):295–307

Kermack WO, McKendrick AG (1927) A contribution to the mathematical theory of epidemics. Proceedings of the Royal Society A 115(772):700–721

Kovács IA, Palotai R, Szalay MS, Csermely P (2010) Community landscapes: a novel, integrative approach for the determination of overlapping network modules. PLoS ONE 7:e12,528

Kuhn HW (1955) The Hungarian method for the assignment problem. Naval Research Logistic Quarterly 2:83–97

Kumpula J, Saramäki J, Kaski K, Kertész J (2007) Limited resolution in complex network community detection with potts model approach. European Physics Journal B 56:41–45

von Luxburg U (2007) A tutorial on spectral clustering. Stat Comput 17(4):395–416, DOI 10.1007/s11222-007-9033-z, URL http://dx.doi.org/10.1007/s11222-007-9033-z

Lyzinski V, Sussman D, Tang M, Athreya A, Priebe C (2013) Perfect clustering for stochastic blockmodel graphs via adjacency spectral embedding. arXiv preprint arXiv:13100532

Lyzinski V, Fishkind D, Fiori M, Vogelstein J, Priebe C, Sapiro G (2014a) Graph matching: Relax at your own risk. arXiv preprint arXiv:14053133

Lyzinski V, Fishkind D, Priebe C (2014b) Seeded graph matching for correlated Erdos-Renyi graphs. Journal of Machine Learning Research

Martin S, Brown WM, Klavans R, Boyack K (2011) OpenOrd: an open-source toolbox for large graph layout. In: Visualization and Data Analysis, vol 7868, p 786806, DOI 10.1117/12.871402

May RM, Lloyd AL (2001) Infection dynamics on scale-free networks. Physical Review E 64(66):066,112

Meilă M (2003) Comparing clusterings by the variation of information. In: Learning Theory and Kernel Machines, Lecture Notes in Computer Science, vol 2777, Springer, pp 173–187

Molloy M, Reed B (1995) A critical point for random graphs with a given degree sequence. Random Structures and Algorithms 6:161–179

Nepusz T, Petróczi A, Négyessy L, Bazsó F (2008) Fuzzy communities and the concept of bridgeness in complex networks. Physical Review E 77:016,107

Newman M (2004) Fast algorithm for detecting community structure in networks. Physical Review E 69:066,133

Newman M (2006) Modularity and community structure in networks. Proceedings of the National Academy of Sciences of the United States of America 103(23):8577–82

Palla G, Derényi I, Farkas I, Vicsek T (2005) Uncovering the overlapping community structure of complex networks in nature and society. Nature 435(7043):814–8

Pastor-Satorras R, Vespignani A (2001) Epidemic spreading in scale-free networks. Physical Review Letters 86(14):3200–3203

Pastor-Satorras R, Vespignani A (2002a) Epidemic dynamics in finite size scale-free networks. Physical Review E 65:035,108(R)

Pastor-Satorras R, Vespignani A (2002b) Immunization of complex networks. Physical Review E 65:036,104

Porter M, Onnela JP, Mucha P (2009) Communities in networks. Notices of the American Mathematical Society 56(9):1082–1097, 1164–1166

Priebe CE, Conroy JM, Marchette DJ, Park Y (2005) Scan statistics on Enron graphs. Computational and Mathematical Organization Theory 11:229–247

Qin T, Rohe K (2013) Regularized spectral clustering under the degree-corrected stochastic blockmodel. In: Advances in Neural Information Processing Systems, pp 3120–3128

Raghavan U, Albert R, Kumara S (2007) Near linear time algorithm to detect community structures in large-scale networks. Physical Review E 76:036,106

Rand WM (1971) Objective criteria for the evaluation of clustering methods. Journal of the American Statistical Association 66(336):846–850

Reichardt J, Bornholdt S (2006) Statistical mechanics of community detection. Physical Review E 74:016,110

Reingold E, Tilford J (1981) Tidier drawing of trees. IEEE Transactions on Software Engineering SE-7(2):223–228

Rohe K, Yu B (2012) Co-clustering for directed graphs; the stochastic co-blockmodel and a spectral algorithm. arXiv preprint arXiv:12042296

Sarkar P, Bickel PJ (2013) Role of normalization in spectral clustering for stochastic blockmodels. arXiv preprint arXiv:13101495

Scheinerman ER, Tucker K (2010) Modeling graphs using dot product representations. Comput Statist 25(1):1–6, DOI 10.1007/s00180-009-0158-8, URL http://dx.doi.org/10.1007/s00180-009-0158-8

Schmuhl M (2003) The graphopt algorithm. URL http://www.schmuhl.org/graphopt/, retrieved on 12 March 2011

Sussman DL (2014) Foundations of adjacency spectral embedding. PhD thesis, The Johns Hopkins University

Sussman DL, Tang M, Fishkind DE, Priebe CE (2012a) A consistent adjacency spectral embedding for stochastic blockmodel graphs. J Amer Statist Assoc 107(499):1119–1128, DOI 10.1080/01621459.2012.699795, URL http://dx.doi.org/10.1080/01621459.2012.699795

Sussman DL, Tang M, Fishkind DE, Priebe CE (2012b) A consistent adjacency spectral embedding for stochastic blockmodel graphs. J Amer Statist Assoc 107(499):1119–1128, DOI 10.1080/01621459.2012.699795, URL http://dx.doi.org/10.1080/01621459.2012.699795

Sussman DL, Tang M, Priebe CE (2013) Consistent latent position estimation and vertex classification for random dot product graphs. IEEE Transactions on Pattern Analysis and Machine Intelligence (Accepted)

Tang M, Athreya A, Sussman DL, Lyzinski V, Priebe CE (2014) Two-sample hypothesis testing for random dot product graphs via adjacency spectral embedding. arXiv preprint arXiv:14037249

Traag VA, Bruggeman J (2009) Community detection in networks with positive and negative links. Physical Review E 80:036,115

Varshney LR, Chen BL, Paniagua E, Hall DH, Chklovskii DB (2011) Structural properties of the caenorhabditis elegans neuronal network. PLoS computational biology 7(2):e1001,066

Vidal R (2010) A tutorial on subspace clustering. IEEE Signal Processing Magazine 28(2):52–68

Vogelstein J, Conroy J, Podrazik L, Kratzer S, Harley E, Fishkind D, Vogelstein R, Priebe C (2012) Fast approximate quadratic programming for large (brain) graph matching. arXiv:11125507

Wang H, Tang M, Park Y, Priebe CE (2014) Locality statistics for anomaly detection in time series of graphs. IEEE Transactions on Signal Processing 62(3):703–717

Young SJ, Scheinerman ER (2007) Random dot product graph models for social networks. In: Algorithms and models for the web-graph, Lecture Notes in Comput. Sci., vol 4863, Springer, Berlin, pp 138–149, DOI 10.1007/978-3-540-77004-6_11, URL http://dx.doi.org/10.1007/978-3-540-77004-6_11

Zachary WW (1977) An information flow model for conflict and fission in small groups. Journal of Anthropological Research 33:452–473

Zaslavskiy M, Bach F, Vert J (2009) A path following algorithm for the graph matching problem. Pattern Analysis and Machine Intelligence, IEEE Transactions on 31(12):2227–2242