Name: Yash Rajiv Walke **Academic Year:** 2021-2022
Class: MSc CS Part I **Roll no.**: 15

Subject: Social Network Analysis

Index

8	Perform SVD analysis of a network.	
9	Identify ties within the network using two-mode core periphery analysis.	
10	Find “factions” in the network using two-mode faction analysis.	

Practical 1

Aim: Write a program to compute the following for a given a network:

- number of edges
- number of nodes
- degree of node
- node with lowest degree
- the adjacency list
- matrix of the graph.

External packages required: igraph

Description:

The igraph package

Description

igraph is a library and R package for network analysis.

Introduction

The main goals of the igraph library is to provide a set of data types and functions for 1) pain-free implementation of graph algorithms, 2) fast handling of large graphs, with millions of vertices and edges, 3) allowing rapid prototyping via high level languages like R.

Igraph graphs

Igraph graphs have a class 'igraph'. They are printed to the screen in a special format, here is an example, a ring graph created using make_ring:

```
IGRAPH U--- 10 10 -- Ring graph
+ attr: name (g/c), mutual (g/x), circular (g/x)
```

'IGRAPH' denotes that this is an igraph graph. Then come four bits that denote the kind of the graph: the first is 'U' for undirected and 'D' for directed graphs. The second is 'N' for named graph (i.e. if the graph has the 'name' vertex attribute set). The third is 'W' for weighted graphs (i.e. if the 'weight' edge attribute is set). The fourth is 'B' for bipartite graphs (i.e. if the 'type' vertex attribute is set).

Then come two numbers, the number of vertices and the number of edges in the graph, and after a double dash, the name of the graph (the 'name' graph attribute) is printed if present. The second line is optional and it contains all the attributes of the graph. This graph has a 'name' graph attribute, of type character, and two other graph attributes called 'mutual' and 'circular', of a complex type. A complex type is simply anything that is not numeric or character. See the documentation of print.igraph for details.

If you want to see the edges of the graph as well, then use the print_all function:

```
> print_all(g)
IGRAPH badcafe U--- 10 10 -- Ring graph
+ attr: name (g/c), mutual (g/x), circular (g/x)
+ edges:
 [1] 1-- 2 2-- 3 3-- 4 4-- 5 5-- 6 6-- 7 7-- 8 8-- 9 9--10 1--
10
```

Creating graphs

There are many functions in igraph for creating graphs, both deterministic and stochastic; stochastic graph constructors are called 'games'.

To create small graphs with a given structure probably the graph_from_literal function is easiest. It uses R's formula interface, its manual page contains many examples. Another option is graph, which takes numeric vertex ids directly. graph.atlas creates graph from the Graph Atlas, make_graph can create some special graphs.

To create graphs from field data, graph_from_edgelist, graph_from_data_frame and graph_from_adjacency_matrix are probably the best choices.

The igraph package includes some classic random graphs like the Erdos-Renyi GNP and GNM graphs (sample_gnp, sample_gnm) and some recent popular models, like preferential attachment (sample_pa) and the small-world model (sample_smallworld).

Vertex and edge IDs

Vertices and edges have numerical vertex ids in igraph. Vertex ids are always consecutive and they start with one. I.e. for a graph with *n* vertices the vertex ids are between *1* and *n*. If some operation changes the number of vertices in the graphs, e.g. a subgraph is created via induced_subgraph, then the vertices are renumbered to satisfy this criteria.

The same is true for the edges as well, edge ids are always between one and *m*, the total number of edges in the graph.

It is often desirable to follow vertices along a number of graph operations, and vertex ids don't allow this because of the renumbering. The solution is to assign attributes to the vertices. These are kept by all operations, if possible. See more about attributes in the next section.

Attributes

In igraph it is possible to assign attributes to the vertices or edges of a graph, or to the graph itself. igraph provides flexible constructs for selecting a set of vertices or edges based on their attribute values, see vertex_attr, V and E for details.

Some vertex/edge/graph attributes are treated specially. One of them is the 'name' attribute. This is used for printing the graph instead of the numerical ids, if it exists. Vertex names can also be used to specify a vector or set of vertices, in all igraph functions. E.g. degree has a v argument that gives the vertices for which the degree is calculated. This argument can be given as a character vector of vertex names.

Edges can also have a 'name' attribute, and this is treated specially as well. Just like for vertices, edges can also be selected based on their names, e.g. in the delete_edges and other functions.

We note here, that vertex names can also be used to select edges. The form 'from|to', where 'from' and 'to' are vertex names, select a single, possibly directed, edge going from 'from' to 'to'. The two forms can also be mixed in the same edge selector.

Other attributes define visualization parameters, see igraph.plotting for details.

Attribute values can be set to any R object, but note that storing the graph in some file formats might result the loss of complex attribute values. All attribute values are preserved if you use save and load to store/retrieve your graphs.

Visualization

igraph provides three different ways for visualization. The first is the plot.igraph function. (Actually you don't need to write plot.igraph, plot is enough. This function uses regular R graphics and can be used with any R device.

The second function is tkplot, which uses a Tk GUI for basic interactive graph manipulation. (Tk is quite resource hungry, so don't try this for very large graphs.)

The third way requires the rgl package and uses OpenGL. See the rglplot function for the details.

Make sure you read igraph.plotting before you start plotting your graphs.

File formats

igraph can handle various graph file formats, usually both for reading and writing. We suggest that you use the GraphML file format for your graphs, except if the graphs are too big. For big graphs a simpler format is recommended. See read_graph and write_graph for details.

Further information

The igraph homepage is at https://igraph.org. See especially the documentation section. Join the discussion forum at https://igraph.discourse.group if you have questions or comments.

Creating (small) graphs via a simple interface

Description

This function is useful if you want to create a small (named) graph quickly, it works for both directed and undirected graphs.

Usage

```
graph_from_literal(..., simplify = TRUE)

from_literal(...)
```

Arguments

`...`	For `graph_from_literal` the formulae giving the structure of the graph, see details below. For `from_literal` all arguments are passed to `graph_from_literal`.
`simplify`	Logical scalar, whether to call `simplify` on the created graph. By default the graph is simplified, loop and multiple edges are removed.

Details

graph_from_literal is very handy for creating small graphs quickly. You need to supply one or more R expressions giving the structure of the graph. The expressions consist of vertex names and edge operators. An edge operator is a sequence of '-' and '+' characters, the former is for the edges and the latter is used for arrow heads. The edges can be arbitrarily long, ie. you may use as many '-' characters to "draw" them as you like.

If all edge operators consist of only '-' characters then the graph will be undirected, whereas a single '+' character implies a directed graph.

Let us see some simple examples. Without arguments the function creates an empty graph:

```
graph_from_literal()
```

A simple undirected graph with two vertices called 'A' and 'B' and one edge only:

```
graph_from_literal(A-B)
```

Remember that the length of the edges does not matter, so we could have written the following, this creates the same graph:

```
graph_from_literal( A-----B )
```

If you have many disconnected components in the graph, separate them with commas. You can also give isolate vertices.

```
graph_from_literal( A--B, C--D, E--F, G--H, I, J, K )
```

The ':' operator can be used to define vertex sets. If an edge operator connects two vertex sets then every vertex from the first set will be connected to every vertex in the second set. The following form creates a full graph, including loop edges:

```
graph_from_literal( A:B:C:D -- A:B:C:D )
```

In directed graphs, edges will be created only if the edge operator includes a arrow head ('+') at the end of the edge:

```
graph_from_literal( A -+ B -+ C )
graph_from_literal( A +- B -+ C )
graph_from_literal( A +- B -- C )
```

Thus in the third example no edge is created between vertices B and C.

Mutual edges can be also created with a simple edge operator:

```
graph_from_literal( A +-+ B +---+ C ++ D + E)
```

Note again that the length of the edge operators is arbitrary, '+', '++' and '+-----+' have exactly the same meaning.

If the vertex names include spaces or other special characters then you need to quote them:

```
graph_from_literal( "this is" +- "a silly" -+ "graph here" )
```

You can include any character in the vertex names this way, even '+' and '-' characters.

See more examples below.

Value

An igraph graph

See Also

Other determimistic constructors:

graph_from_atlas(), graph_from_edgelist(), make_chordal_ring(), make_empty_graph(), make_full_citation_graph(), make_full_graph(), make_graph(), make_lattice(), make_ring(), make_star(), make_tree()

Examples

```
# A simple undirected graph
g <- graph_from_literal( Alice-Bob-Cecil-Alice, Daniel-Cecil-Eugene,
                         Cecil-Gordon )
g

# Another undirected graph, ":" notation
g2 <- graph_from_literal( Alice-Bob:Cecil:Daniel, Cecil:Daniel-Eugene:Gordon
)
g2

# A directed graph
```

```
g3 <- graph_from_literal( Alice +-+ Bob --+ Cecil +-- Daniel,
                     Eugene --+ Gordon:Helen )
g3

# A graph with isolate vertices
g4 <- graph_from_literal( Alice -- Bob -- Daniel, Cecil:Gordon, Helen )
g4
V(g4)$name

# "Arrows" can be arbitrarily long
g5 <- graph_from_literal( Alice +---------+ Bob )
g5

# Special vertex names
g6 <- graph_from_literal( "+" -- "-", "*" -- "/", "%%" -- "%/%" )
g6
```

The size of the graph (number of edges)

Description

ecount of an alias of this function.

Usage

```
gsize(graph)
```

Arguments

graph	The graph.

Value

Numeric scalar, the number of edges.

See Also

Other structural queries: [.igraph(), [[.igraph(), adjacent_vertices(), are_adjacent(), ends(), get.edge.ids(), gorder(), head_of(), incident_edges(), incident(), is_directed(), neighbors(), tail_of()

Examples

```
g <- sample_gnp(100, 2/100)
gsize(g)

# Number of edges in a G(n,p) graph
replicate(100, sample_gnp(10, 1/2), simplify = FALSE) %>%
  vapply(gsize, 0) %>%
  hist()
```

Order (number of vertices) of a graph

Description

Order (number of vertices) of a graph

Usage

```
gorder(graph)
```

Arguments

graph	The graph

Value

Number of vertices, numeric scalar.

See Also

Other structural queries: [.igraph(), [[.igraph(), adjacent_vertices(), are_adjacent(), ends(), get.edge.ids(), gsize(), head_of(), incident_edges(), incident(), is_directed(), neighbors(), tail_of()

Examples

```
g <- make_ring(10)
gorder(g)
```

Edges of a graph

Description

An edge sequence is a vector containing numeric edge ids, with a special class attribute that allows custom operations: selecting subsets of edges based on attributes, or graph structure, creating the intersection, union of edges, etc.

Usage

```
E(graph, P = NULL, path = NULL, directed = TRUE)
```

Arguments

graph	The graph.
P	A list of vertices to select edges via pairs of vertices. The first and second vertices select the first edge, the third and fourth the second, etc.

`path`	A list of vertices, to select edges along a path. Note that this only works reliable for simple graphs. If the graph has multiple edges, one of them will be chosen arbitrarily to be included in the edge sequence.
`directed`	Whether to consider edge directions in the `P` argument, for directed graphs.

Details

Edge sequences are usually used as igraph function arguments that refer to edges of a graph.

An edge sequence is tied to the graph it refers to: it really denoted the specific edges of that graph, and cannot be used together with another graph.

An edge sequence is most often created by the E() function. The result includes edges in increasing edge id order by default (if. none of the P and path arguments are used). An edge sequence can be indexed by a numeric vector, just like a regular R vector. See links to other edge sequence operations below.

Value

An edge sequence of the graph.

Indexing edge sequences

Edge sequences mostly behave like regular vectors, but there are some additional indexing operations that are specific for them; e.g. selecting edges based on graph structure, or based on edge attributes. See [.igraph.es for details.

Querying or setting attributes

Edge sequences can be used to query or set attributes for the edges in the sequence. See $.igraph.es for details.

See Also

Other vertex and edge sequences: V(), igraph-es-attributes, igraph-es-indexing2, igraph-es-indexing, igraph-vs-attributes, igraph-vs-indexing2, igraph-vs-indexing, print.igraph.es(), print.igraph.vs()

Examples

```
# Edges of an unnamed graph
g <- make_ring(10)
E(g)

# Edges of a named graph
g2 <- make_ring(10) %>%
  set_vertex_attr("name", value = letters[1:10])
E(g2)
```

Vertices of a graph

Description

Create a vertex sequence (vs) containing all vertices of a graph.

Usage

```
V(graph)
```

Arguments

`graph`	The graph

Details

A vertex sequence is just what the name says it is: a sequence of vertices. Vertex sequences are usually used as igraph function arguments that refer to vertices of a graph.

A vertex sequence is tied to the graph it refers to: it really denoted the specific vertices of that graph, and cannot be used together with another graph.

At the implementation level, a vertex sequence is simply a vector containing numeric vertex ids, but it has a special class attribute which makes it possible to perform graph specific operations on it, like selecting a subset of the vertices based on graph structure, or vertex attributes.

A vertex sequence is most often created by the V() function. The result of this includes all vertices in increasing vertex id order. A vertex sequence can be indexed by a numeric vector, just like a regular R vector. See [.igraph.vs and additional links to other vertex sequence operations below.

Value

A vertex sequence containing all vertices, in the order of their numeric vertex ids.

Indexing vertex sequences

Vertex sequences mostly behave like regular vectors, but there are some additional indexing operations that are specific for them; e.g. selecting vertices based on graph structure, or based on vertex attributes. See [.igraph.vs for details.

Querying or setting attributes

Vertex sequences can be used to query or set attributes for the vertices in the sequence. See $.igraph.vs for details.

See Also

Other vertex and edge sequences: E(), igraph-es-attributes, igraph-es-indexing2, igraph-es-indexing, igraph-vs-attributes, igraph-vs-indexing2, igraph-vs-indexing, print.igraph.es(), print.igraph.vs()

Examples

```
# Vertex ids of an unnamed graph
```

```
g <- make_ring(10)
V(g)

# Vertex ids of a named graph
g2 <- make_ring(10) %>%
  set_vertex_attr("name", value = letters[1:10])
V(g2)
```

Degree and degree distribution of the vertices

Description

The degree of a vertex is its most basic structural property, the number of its adjacent edges.

Usage

```
degree(
  graph,
  v = V(graph),
  mode = c("all", "out", "in", "total"),
  loops = TRUE,
  normalized = FALSE
)

degree_distribution(graph, cumulative = FALSE, ...)
```

Arguments

`graph`	The graph to analyze.
`v`	The ids of vertices of which the degree will be calculated.
`mode`	Character string, “out” for out-degree, “in” for in-degree or “total” for the sum of the two. For undirected graphs this argument is ignored. “all” is a synonym of “total”.
`loops`	Logical; whether the loop edges are also counted.
`normalized`	Logical scalar, whether to normalize the degree. If `TRUE` then the result is divided by *n-1*, where *n* is the number of vertices in the graph.
`cumulative`	Logical; whether the cumulative degree distribution is to be calculated.
`...`	Additional arguments to pass to `degree`, eg. `mode` is useful but also `v` and `loops` make sense.

Value

For degree a numeric vector of the same length as argument v.

For degree_distribution a numeric vector of the same length as the maximum degree plus one. The first element is the relative frequency zero degree vertices, the second vertices with degree one, etc.

Author(s)

Gabor Csardi csardi.gabor@gmail.com

Examples

```
g <- make_ring(10)
degree(g)
g2 <- sample_gnp(1000, 10/1000)
degree_distribution(g2)
```

Convert a graph to an adjacency matrix

Description

Sometimes it is useful to work with a standard representation of a graph, like an adjacency matrix.

Usage

```
as_adjacency_matrix(
  graph,
  type = c("both", "upper", "lower"),
  attr = NULL,
  edges = FALSE,
  names = TRUE,
  sparse = igraph_opt("sparsematrices")
)

as_adj(
  graph,
  type = c("both", "upper", "lower"),
  attr = NULL,
  edges = FALSE,
  names = TRUE,
  sparse = igraph_opt("sparsematrices")
)
```

Arguments

graph	The graph to convert.

`type` Gives how to create the adjacency matrix for undirected graphs. It is ignored for directed graphs. Possible values: `upper`: the upper right triangle of the matrix is used, `lower`: the lower left triangle of the matrix is used. `both`: the whole matrix is used, a symmetric matrix is returned.

`attr` Either `NULL` or a character string giving an edge attribute name. If `NULL` a traditional adjacency matrix is returned. If not `NULL` then the values of the given edge attribute are included in the adjacency matrix. If the graph has multiple edges, the edge attribute of an arbitrarily chosen edge (for the multiple edges) is included. This argument is ignored if `edges` is `TRUE`.

Note that this works only for certain attribute types. If the `sparse` argumen is `TRUE`, then the attribute must be either logical or numeric. If the `sparse` argument is `FALSE`, then character is also allowed. The reason for the difference is that the `Matrix` package does not support character sparse matrices yet.

`edges` Logical scalar, whether to return the edge ids in the matrix. For non-existant edges zero is returned.

`names` Logical constant, whether to assign row and column names to the matrix. These are only assigned if the `name` vertex attribute is present in the graph.

`sparse` Logical scalar, whether to create a sparse matrix. The '`Matrix`' package must be installed for creating sparse matrices.

Details

as_adjacency_matrix returns the adjacency matrix of a graph, a regular matrix if sparse is FALSE, or a sparse matrix, as defined in the 'Matrix' package, if sparse if TRUE.

Value

A vcount(graph) by vcount(graph) (usually) numeric matrix.

See Also

graph_from_adjacency_matrix, read_graph

Examples

```
g <- sample_gnp(10, 2/10)
as_adjacency_matrix(g)
V(g)$name <- letters[1:vcount(g)]
as_adjacency_matrix(g)
E(g)$weight <- runif(ecount(g))
as_adjacency_matrix(g, attr="weight")
```

Adjacency lists

Description

Create adjacency lists from a graph, either for adjacent edges or for neighboring vertices

Usage

```
as_adj_list(graph, mode = c("all", "out", "in", "total"))

as_adj_edge_list(graph, mode = c("all", "out", "in", "total"))
```

Arguments

`graph`	The input graph.
`mode`	Character scalar, it gives what kind of adjacent edges/vertices to include in the lists. '`out`' is for outgoing edges/vertices, '`in`' is for incoming edges/vertices, '`all`' is for both. This argument is ignored for undirected graphs.

Details

as_adj_list returns a list of numeric vectors, which include the ids of neighbor vertices (according to the mode argument) of all vertices.

as_adj_edge_list returns a list of numeric vectors, which include the ids of adjacent edgs (according to the mode argument) of all vertices.

Value

A list of numeric vectors.

Author(s)

Gabor Csardi csardi.gabor@gmail.com

See Also

as_edgelist, as_adj

Examples

```
g <- make_ring(10)
as_adj_list(g)
as_adj_edge_list(g)
```

Procedure:

1. Load the `igraph` external library.

```
> library(igraph)

Attaching package: 'igraph'

The following objects are masked from 'package:stats':

    decompose, spectrum

The following object is masked from 'package:base':

    union
```

2. Create a directed and undirected graph using the `graph.formula` function.

```
> u_graph <- graph.formula(A - B, A - C, A - D, B - C, B - F, C - D, C - E, C - F, D - E, E - F, F - G, G - H)
> d_graph <- graph.formula(A ++ B, A ++ D, A -+ C, B -+ C, B -+ E, B -+ F, C -+ D, C -+ F, D -+ E)
```

3. Find the number of edges in both graphs using the `ecount()` function.

```
> ecount(u_graph)
[1] 12
> ecount(d_graph)
[1] 11
```

4. Find the number of nodes in both graphs using the `vcount()` function.

```
> vcount(u_graph)
[1] 8
> vcount(d_graph)
[1] 6
```

5. Find the edge set in both graphs using the `E()` function.

```
> E(u_graph)
+ 12/12 edges from 505314f (vertex names):
 [1] A--B A--C A--D B--C B--F C--D C--F C--E D--E F--E F--G G--H
> E(d_graph)
+ 11/11 edges from 9fbbde8 (vertex names):
 [1] A->B A->D A->C B->A B->C B->E B->F D->A D->E C->D C->F
```

6. Find the vertex set in both graphs using the `V()` function.

```
> V(u_graph)
+ 8/8 vertices, named, from 505314f:
[1] A B C D F E G H
> V(d_graph)
+ 6/6 vertices, named, from 9fbbde8:
[1] A B D C E F
```

7. Find the degree of all vertices in both graphs using the `degree()` function. Also find the in-degree and out-degree using the `mode` parameter of the `degree()` function.

```
> degree(u_graph)
A B C D F E G H
3 3 5 3 4 3 2 1
> degree(u_graph, mode = "in")
A B C D F E G H
3 3 5 3 4 3 2 1
> degree(u_graph, mode = "out")
A B C D F E G H
3 3 5 3 4 3 2 1
```

```
> degree(d_graph)
A B D C E F
5 5 4 4 2 2
> degree(d_graph, mode = "in")
A B D C E F
2 1 2 2 2 2
> degree(d_graph, mode = "out")
A B D C E F
3 4 2 2 0 0
```

8. Find the node with the smallest degree in the undirected graph. Also find out the nodes with the smallest in-degree and out-degree in the directed graph.

```
> V(u_graph)$name[degree(u_graph) == min(degree(u_graph))]
[1] "H"
```

```
> V(d_graph)$name[degree(d_graph, mode = "in") == min(degree(d_graph, mode = "in"))]
[1] "B"
> V(d_graph)$name[degree(d_graph, mode = "out") == min(degree(d_graph, mode = "out"))]
[1] "E" "F"
```

9. Print the adjacency matrix for both the graphs using the `get.adjacency()` function.

```
> get.adjacency(u_graph)
8 x 8 sparse Matrix of class "dgCMatrix"
  A B C D F E G H
A . 1 1 1 . . . .
B 1 . 1 . 1 . . .
C 1 1 . 1 1 1 . .
D 1 . 1 . . 1 . .
F . 1 1 . . 1 1 .
E . . 1 1 1 . . .
G . . . . 1 . . 1
H . . . . . . 1 .
```

```
> get.adjacency(d_graph)
6 x 6 sparse Matrix of class "dgCMatrix"
  A B D C E F
A . 1 1 1 . .
B 1 . . 1 1 1
D 1 . . . 1 .
C . . 1 . . 1
E . . . . . .
F . . . . . .
```

10. Print the adjacency list for both the graphs using the `get.adjlist()` function.

```
> get.adjlist(u_graph)
$A
+ 3/8 vertices, named, from 505314f:
[1] B C D

$B
+ 3/8 vertices, named, from 505314f:
[1] A C F

$C
+ 5/8 vertices, named, from 505314f:
[1] A B D F E

$D
+ 3/8 vertices, named, from 505314f:
[1] A C E

$F
+ 4/8 vertices, named, from 505314f:
[1] B C E G

$E
+ 3/8 vertices, named, from 505314f:
[1] C D F
```

```
$G
+ 2/8 vertices, named, from 505314f:
[1] F H

$H
+ 1/8 vertex, named, from 505314f:
[1] G
```

```
> get.adjlist(d_graph)
$A
+ 5/6 vertices, named, from 9fbbde8:
[1] B B D D C

$B
+ 5/6 vertices, named, from 9fbbde8:
[1] A A C E F

$D
+ 4/6 vertices, named, from 9fbbde8:
[1] A A C E

$C
+ 4/6 vertices, named, from 9fbbde8:
[1] A B D F

$E
+ 2/6 vertices, named, from 9fbbde8:
[1] B D

$F
+ 2/6 vertices, named, from 9fbbde8:
[1] B C
```