Netzwerkanalyse

IV. Netzwerkdaten laden bzw. importieren

> library(igraph) # lädt das *igraph-*Paket

> library(igraphdata) # lädt das *igraphdata-*Paket

> data(package="igraphdata") # Inhalt des Pakets in der Übersicht

> data(karate) # lädt den Datensatz Karate

> ?karate # Infos über den Datensatz karate unter dem Reiter „help“

> karate # ruft das *igraph*-Datenset karate auf

Jedes *igraph*-Objekt ist genau gleich aufgebaut:

* **IGRAPH** zeigt an, dass es sich um ein Objekt der Klasse „igraph“ handelt und beschreibt die intern vergebene Nummer des *igraph*-Objekts. Diese unterscheidet sich jedes Mal beim Laden, da der Datensatz neu geladen wird.
* Die **vierstellige Buchstabenfolge** danach ist wie folgt codiert:  
  1) **Art des Netzwerks**: **U** = undirected (ungerichtet), **D** = directed (gerichtet)  
  2) **Bezeichnung der Knoten**: **N** = named, dazu muss ein Vertex-Attribut „name“ vorliegen  
  3) **Gewicht**: **W** = weighted network, d.h. die Edgelist ist mit einem numerischen Wert gewichtet  
  4) **Two-Mode-Netzwerk**: **B**= bi-partite (oder two-mode Netzwerk), wenn das Vertex-Attribut „type“ vorhanden ist.  
  Fehlt eine Codierung, wird dies einfach mit einem „–“ angezeigt.
* Danach folgt die Nennung der im Graphen angelegten **Attribute**. Diese entsprechen in der Regel der Bezeichnung der Spalten im Header und werden genau so übernommen. Hinter jedem Attribut steht eine Klammer, die die Art des Attributs genauer beschreibt, etwa (v/n):  
  Die **erste Nennung** vor dem Trennstrich bezieht sich auf die Ebene des Attributs:  
  **- g** meint, dass sich das Attribut auf den gesamten Graphen bezieht   
  **- v** meint, dass sich das Attribut auf die vertices oder Knoten bezieht  
  **- e** meint, dass sich das Attribut auf die edges oder Kanten bezieht
* Die **zweite Nennung** in der Klammer gibt an, wie die Werte des Attributs codiert sind, in der Regel sind dies **c** für characters, also Text, und **n** für numerische Werte, also Zahlen. Beispielsweise hat der Datensatz numerische Werte nur bei den Attributen „Faction“, „color“ und „weight“.

Netzwerkdaten laden und importieren

> install.packages("network", dependencies=TRUE) # installiert das Paket *network*

> library(network) # lädt das Paket *network*

> data(flo) # lädt das Netzwerk der florentinischen Heiraten nach Padgett

> flo # zeigt die Matrix-Daten des Netzwerks an: Jede Heirat/Beziehung ist mit 1 markiert

> iflo <- graph\_from\_adjacency\_matrix(flo, mode ="undirectet") # Matrix wird in ein *igraph-*Objekt umgewandelt

> iflo # zeigt das *igraph-*Objekt an

> plot(iflo) #erste Visualisierung

### **Vorhandene Daten in R verwenden**

#### **Paket igraphdata**

> library(igraphdata) # lädt das *igraphdata*-Paket

> data(package="igraphdata") # liefert einen Überblick über den Inhalt des Pakets

> data(enron) # lädt den Datensatz enron

> enron # zeigt das *igraph*-Objekt enron an

> ?enron # ruft die Hilfefunktion auf, die den Datensatz enron erklärt

### **Dateien und Datensätze einlesen**

#### **Daten einlesen**

﻿> edges <- read.csv("wohl\_edge.csv", header=T, as.is=T) # liest die Edgelist ein und ordnet sie der Variablen edges zu

> edges # zeigt die edges-Daten aus der csv-Datei in der Konsole an

> head(edges) # zeigt lediglich die ersten Zeilen des Datensatzes an

> nodes <- read.csv("wohl\_node.csv", header=T, as.is=T) # liest die Nodelist ein und ordnet sie der Variablen nodes zu

> nodes # zeigt die nodes-Daten aus der csv-Datei in der Konsole an

> head(nodes) # zeigt lediglich die ersten Zeilen des Datensatzes an

**Erstellung eines *igraph* Objekts**

> ties <- as.matrix(edges) # wandelt die Edgelist in eine Matrix um; ordnet die Matrix der Variablen ties zu

﻿> wohled <- graph\_from\_edgelist(ties, directed=FALSE)

> wohled # zeigt das *igraph*-Objekt wohled an

> wohledno <- graph\_from\_data\_frame(d=ties, vertices=nodes, directed=F)

> plot(wohled, vertex.size=degree, vertex.label=NA, edge.arrow.width=0.4)

bzw.

> plot(wohledno, vertex.size=degree, vertex.label=NA, edge.arrow.width=0.4)  
> list.vertex.attributes(wohled) # zeigt die vorhandenen Knotenattribute an, wenn noch keine nodelist eingelesen ist

> list.vertex.attributes(wohledno) # zeigt die vorhandenen Knotenattribute an

> V(wohledno)[V(wohledno)$type == 1]$shape <- "square" # ordnet allen Typen, die den Wert 1 annehmen, Quadrate zu

> V(wohledno)[V(wohledno)$type == 0]$shape <- "circle" # ordnet allen Typen, die den Wert 0 annehmen, Kreise zu

> plot(wohledno, layout = layout\_with\_kk, vertex.size=degree, vertex.label=NA, edge.arrow.width=0.4)

## V. Netzwerke analysieren

﻿**V(karate)$name** # Mit dem $ Attribut wird ausgewählt, welches Attribut angezeigt werden soll

name <- V(karate)$name # Vertex-Attribute vereinfachen

label <- V(karate)$label # Vertex-Attribute vereinfachen  
color <- V(karate)$color # Vertex-Attribute vereinfachen

gewicht <- E(karate)$weight # Edge-Attribute vereinfachen

Datenatz in zwei Gruppen aufteilen

f1 <- V(karate)$Faction == "1" # Nur die Werte mit 1  
f2 <- V(karate)$Faction != "1" # alle Werte ungleich 1, alternativ wäre auch == "2" möglich gewesen

mrhi <- V(karate)[label == "H"] # sucht Knoten „H“ raus und weist ihm die Variable „mrhi“ aus

E(karate)[ adj(mrhi) # sucht nach allen Knoten, die um „H“ angesiedelt sind

hi <- incident(karate, V(karate)[name == "Mr Hi"], mode = "all") # berechnet Werte für einzelne Knoten 🡪 sucht nach allen Knoten, die um „H“ angesiedelt sind (schnellere Variante als oben)  
hi

Visualisieren eines Knotens mit seinen direkt verbundenen Kanten im Netzwerk:

ecol <- rep("grey80", ecount(karate)) # alle Kanten in Grauton 80% Deckung  
ecol[hi] <- "orange" # alle Kanten von „Mr Hi“ in orange  
vcol <- rep("grey40", vcount(karate)) # alle Knoten in Grauton 40% Deckung  
vcol[V(karate)$name == "Mr Hi"] <- "gold" # Knoten von Mr. Hi in gold  
V(karate)$label <- NA # optional blenden wir damit die labels der Knoten aus

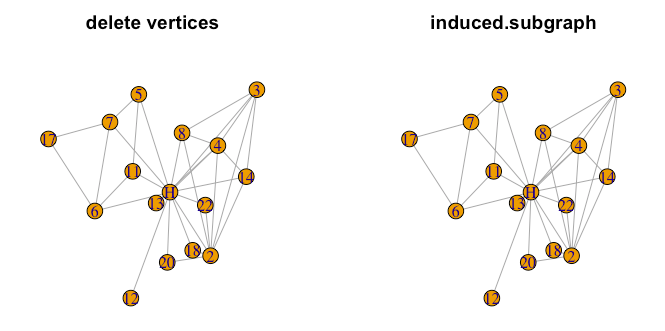
plot(karate, vertex.color=vcol, edge.color=ecol)  
# wichtig ist, dass sich die Visualisierung auf die zuvor angelegten Visualisierungsparameter beziehen

Teilnetzwerke bilden

karate1 <- delete\_vertices(karate, V(karate)[Faction != "1"]) # Übrig bleiben jetzt alle Knoten, deren Attribut Faction den Wert 1 annimmt, also das Netzwerk um Mr. Hi. 🡪Variante 1  
plot(karate1)

karate2 <- induced.subgraph(karate, V(karate)[Faction == "1"]) # Variante 2  
plot(karate2)

dev.off() # schaltet vorhergehende Befehle aus  
par(mfrow=c(1, 2)) # gibt vor, dass zwei Abbildungen in einer Zeile gezeigt werden sollen  
plot(karate1, layout = layout\_with\_kk, main = "delete vertices") 🡪Teilnetzwerk 1

plot(karate2, layout = layout\_with\_kk, main = "induced.subgraph") 🡪Teilnetzwerk 2  


**Teilnetzwerke nach Kantengewichten isolieren**

karate4 <- subgraph.edges(karate, E(karate)[weight > 4]) # zeigt nur Kanten mit einem Gewicht größer 4 an  
karate4 # nur Kanten mit einem Gewicht > 4 in dem Netzwerk behalten plot(karate4)

**Eingängige logische Operatoren**

= entspricht genau dem Wert  
> ist größer als bzw. < ist kleiner als  
! ist nicht (logischer Ausschluss, kann sich auf gleich bzw. größer oder kleiner beziehen und wird vorangestellt, z.B.  
!= ist nicht gleich, !> ist nicht größer oder !< ist nicht kleiner)

**Auflösung Faction 1 und Faction 2**

karate1 <- delete\_vertices(karate, V(karate)[Faction != "1"])  
karate2 <- delete\_vertices(karate, V(karate)[Faction == "1"])  
dev.off()  
par(mfrow=c(1,2))  
plot(karate1, layout = layout\_with\_kk, main = "Faction 1")  
plot(karate2, layout = layout\_with\_kk, main = "Faction 2")

### **Komponenten, Communities und Cluster**

* **Komponenten** (components) sind die Bestandteile eines Netzwerks. Sind alle Knoten miteinander verbunden, gibt es es nur eine Komponente. Zerfällt ein Netzwerk hingegen in unterschiedliche, unverbundene Unternetzwerke, dann werden diese als Komponenten eines Netzwerks bezeichnet.
* **Cluster** hingegen sind Ansammlungen von Knoten innerhalb eines Netzwerks, die aufgrund einer bestimmten Eigenschaft zusammen aggregiert sind.
* **Components können also Cluster enthalten, aber nicht umgekehrt.**

Einlesen eines Beispieldatensatzes

library(igraph) # nicht vergessen!  
# die Dateien werden direkt aus dem *GitHub*-Verzeichnis gelesen

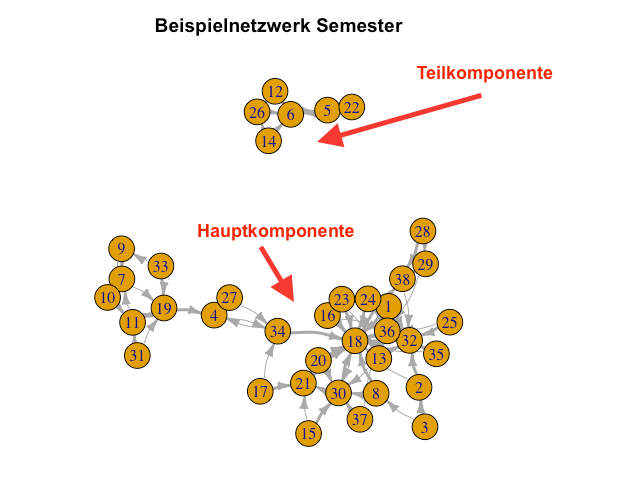
help <- read.csv("<https://raw.githubusercontent.com/hdm-crpr/226305/master/data/students/ehelp.csv>", header = T, as.is = T, sep = ",")

nodes <- read.csv("<https://raw.githubusercontent.com/hdm-crpr/226305/master/data/students/nodes.csv>", header = T, as.is = T, sep = ",")

# prüft, ob alle Variablen eingelesen wurden  
head(help)  
head(nodes)

# wandelt die edgelist in eine Matrix um und baut das *igraph*-Objekt in Verbindung mit der nodelist

hties <- as.matrix(help)  
help <- graph\_from\_data\_frame(d=hties, vertices=nodes, directed=T) # damit ist das vorherige help überschrieben  
help



# addiert edges auf, wenn sie auf der gleichen Beziehung sind

h <- simplify(help, edge.attr.comb = list(weight = "sum"))

# ruft das finale *igraph*-Objekt auf

h

# einfache Visualisierung

plot(h,  
edge.arrow.size=0.2,  
edge.curved=0.2,  
layout=layout\_with\_fr,  
edge.width=E(h)$weight,  
main="Beispielnetzwerk Semester")

##### Analyse von Komponenten

# Analyse der Components  
# Zunächst prüfen wir, ob das Netzwerk überhaupt zusammenhängend ist

is\_connected(h)

# Die Ausgabe des logischen Operators ist immer "true" oder "false". True bedeutet, dass es sich nur um eine Komponente handelt. False, dass es sich um unverbundene Netzwerke handelt.  
# Diese Prüfung ist besonders bei großen und unübersichtlichen Netzwerken hilfreich  
# Mit dem Befehl components können wir die Komponenten in Gruppen einteilen

components(h) # zeigt uns die Verteilung und die Größe der Komponenten an

# Dabei zeigt sich, dass das Netzwerk aus zwei Komponenten besteht

groups(co)

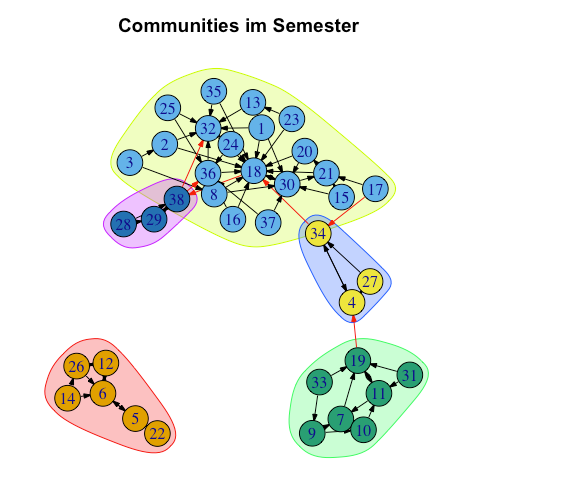
# Die Komponenten zeigen uns die natürliche Verteilung in einem Netzwerk an

**Community-Analyse**

k <- make\_graph("Zachary") # erstellt das Netzwerk  
cw <- cluster\_walktrap(k) # wendet die Clusterung auf das Netzwerk an  
modularity(cw) # berechnet den Modularitätswert  
membership(cw) # weist die Mitgliedschaft in die Communities zu  
plot(cw, k) # visualisiert die Communities

**Übung: Analysieren Sie die Modularität und den Walktrap-Algorithmus des Semesternetzwerks:**

cwh <- cluster\_walktrap(h)  
modularity(cwh)  
membership(cwh)  
plot(cwh, h, edge.arrow.size=0.1, main="Communities im Semester")



### **Dyaden und Triaden**

##### **Anzahl der Dyaden**

library(igraph)  
z <- make\_graph("Zachary")

dyad\_census(z)

**Mutual:** Die Ausgabe berechnet die Anzahl der dyadischen Beziehungen in einem Netzwerk. Da das Karate Netzwerk ungerichtet ist, entspricht die Anzahl der gegenseitigen Beziehungen (mutual) genau der Anzahl der Edges im Netzwerk (lässt sich aus dem *igraph*-Objekt z auslesen).

**Asymmetrische**Beziehungen sind nur in einem gerichteten Netzwerk relevant

**Null** Beziehungen gibt an, wie viele Beziehungen nicht realisiert wurden. Auch wenn die Zahl auf den ersten Blick hoch erscheint, ist dies nicht weiter verwunderlich, weil das Netzwerk über eine geringe Dichte verfügt.

edge\_density(z, loops = FALSE)

##### **Triadenzensus**

**Triaden** sind die zentralen Bausteine von Netzwerken. Sie können 16 verschiedene Formen annehmen, die üblicherweise immer in der gleichen Reihenfolge benannt sind. Die Reihenfolge können Sie so nachschauen: Klicken Sie im Reiter „Packages“ auf das igraph-Paket. Dort gehen Sie auf den Buchstaben T und wählen „**triad\_census**“ aus. Hier sind dann die 16 Triaden-Typen aufgezählt, die in dieser Reihenfolge auch in der Konsole angezeigt werden. Sie können den folgenden Befehl nur bei gerichteten Netzwerken anwenden. Da das karate-Netzwerk ungerichtet ist, erstellen wir fix ein zufälliges Netzwerk.

﻿g <- sample\_gnp(24, 0.19, directed=T, loops=F)

Mit diesem Netzwerk können wir den Triadenzensus durchführen.

triad\_census(g) # führt den Triadenzensus durch.

Ausgegeben werden genau die 16 Typen in der erklärten Reihenfolge.  Da wir ein Zufallsnetzwerk erstellt haben, ist die Anzahl der jeweiligen Triadentypen bei jedem anders.

Bei ungerichteten Netzwerken kann man dennoch nach Cliquen suchen. Dies funktioniert durch den folgenden Befehl, den wir auf das karate-Netzwerk anwenden.

# Durchsuchen nach Cliquen  
clique\_num(z) # durchsucht nach maximaler Anzahl von Cliquen

## VI. Netzwerk- und Akteursmaße berechnen

Generell wird zwischen zwei Arten von Maßen unterschieden, den Netzwerk- und den Akteursmaßen. Bei einer effizienten Projektdurchführung sollten nicht sämtliche Maße berechnet werden, sondern nur diejenigen, die einen Stellenwert hinsichtlich der Forschungsfrage besitzen. Je nachdem, welche Strukturen man untersucht, sollte man die Aussagekraft und Interpretation der Maße hinterfragen.

### **Netzwerkmaße**

**Netzwerkmaße** beziehen sich auf das gesamte Netzwerk. Die Maßangaben eignen sich damit besonders, um mehrere Netzwerke miteinander zu vergleichen und Aussagen über die Gemeinsamkeiten und Unterschiede zu treffen. Netzwerkmaße geben ebenfalls Auskunft über die Beschaffenheit eines Netzwerks. Typische Maße sind die **Dichte, Cliquen, Pfaddistanz** und der **Durchmesser**, die in den folgenden Unterkapiteln näher beleuchtet werden.

#### **Dichte**

Die **Dichte** eines Netzwerks sagt aus, wie viele dieser maximal möglichen Beziehungen realisiert sind – man berechnet also den Quotienten der tatsächlichen und maximalen Kanten. Wenn zwischen den fünf Knoten wirklich 10 Kanten verlaufen, ist die Dichte=10/10=1, also 100%.

edge\_density() # berechnet die Dichte eines Netzwerks

Dabei sollte man aufpassen, dass das Netzwerk, das man als Objekt in der Klammer benennt, weder mehrfache Beziehungen zwischen den Knoten hat, noch zulässt, dass ein Knoten eine Beziehung zu sich selbst hat. Dies könnte dazu führen, dass Werte über 100% angezeigt werden und die Dichte damit verfälscht wäre. Falls das untersuchte Netzwerk jedoch mehrfache und selbstreferenzielle Beziehungen mit einschließt, kann man zur Untersuchung der Dichte einfach eine simplifizierte Version des Basisnetzwerks anfertigen:

> simplenetwork <- simplify(network, remove.multiple = TRUE, remove.loops = TRUE)

> edge\_density(simplenetwork)

Es gibt die Möglichkeit, remove.multiple auf FALSE zu lassen, also nur die selbstreferenziellen Beziehungen durch das Argument remove.loops=TRUE auszublenden. Das kann je nach Forschungsfrage und beobachteter Beziehung bei gerichteten Netzwerken Sinn machen, etwa, wenn man den Austausch von Geld betrachtet. Hier können zwischen jedem Knoten zwei Beziehungen liegen, also gibt es maximal x(x-1) mögliche Kanten. R nimmt als 100%-Wert jedoch immer die maximal möglichen Kanten eines ungerichteten Netzwerks als Referenz. Die Dichte, die man bei einem gerichteten Netzwerk mit der Möglichkeit zweier Kanten zwischen Knoten anzeigen lassen kann, muss demnach auf 200% referenziert werden.

triad\_census() # welche Triadenkonstellation tritt wie oft im Netzwerk auf

cliques(network, min="1", max="4") # Zeigt alle Cliquen an, die mindestens einen und maximal vier Knoten beinhalten

largest\_cliques(network) # nach den größten Cliquen suchen

max\_cliques(network) # nach den maximalen Cliquen suchen

Maximale Cliquen sind Cliquen, die nicht mehr zu einer größeren Clique erweitert werden können. Größte Cliquen sind diejenigen, welche die meisten Knoten beinhalten.

#### **Pfaddistanz**

distances(network, v = V(network), to = V(network)) # Pfaddistanz zwischen zwei Knoten eines Netzwerks

farthest\_vertices(network) # gibt an, welche Knoten am weitesten voneinander entfernt sind

mean\_distance(network) # Drurchschnittsdistanz zwischen zwei Knoten eines Netzwerks

> mean\_distance(network, directed = FALSE)  
# Nutzt man diesen Befehl bei gerichteten Netzwerken, werden die Kanten wie ungerichtete Kanten behandelt.

> mean\_distance(network, directed = TRUE)  
# Nutzt man diesen Befehl bei gerichteten Netzwerken, werden die Kanten wie gerichtete Kanten behandelt.

#### **Durchmesser**

Der Durchmesser ist die Pfaddistanz der Knoten, die man mithilfe des Befehls farthest\_vertices ermittelt hat, also die Pfaddistanz der am weitesten voneinander entfernten Knoten. Man kann den Durchmesser mit dem folgenden Befehl errechnen lassen:

> get\_diameter(network) # Berechnet den Durchmesser eines Netzwerks

### **Akteursmaße**

**Akteursmaße** beziehen sich auf einzelne Akteure und deren Position im Netzwerk. Die Maße geben meist Auskunft darüber, welchen Stellenwert ein Knoten im Netzwerk einnimmt und wie stark er vernetzt ist. Im Mittelpunkt der Betrachtung steht immer die Frage, ob ein Knoten eine zentrale oder periphere Rolle im Netzwerk spielt. Bei den Akteursmaßen geht es dementsprechend um die Zentralität.

##### **Zentralität**

#### **Degree-Zentralität**

Bei der Degree-Zentralität werden „die Verbindungen eines Knoten gezählt – Akteure mit vielen Sozialbeziehungen sind wichtiger im Netzwerk als solche mit wenigen“ (Fuhse, 2016, S. 59). Dabei unterscheidet man „zwischen der Anzahl der empfangenen Beziehungen (**Indegree**) und der der von einem Akteur an andere geschickten Beziehungen (**Outdegree**)“.

degree(network) # Berechnet die Degree-Zentralität eines Netzwerks

degree(network, mode="in") # Zeigt nur die Anzahl der Indegrees an

degree(network, mode="out") # Zeigt nur die Anzahl der Outdegrees an

Die absoluten Zahlen sagen uns aber noch nicht viel. Deshalb können die Degrees normalisiert werden. Dabei werden sie durch alle möglichen Degrees des Graphs geteilt. Gerade bei großen Netzwerken ist eine Normalisierung/Indexierung wichtig, damit die Degrees vergleichbar werden.

degree(network, normalized = TRUE)

Auch bei der Spezifizierung auf In- und Outdegrees können die Werte normalisiert werden:

degree(network, mode="in", normalized=TRUE)

degree(network, mode="out", normalized=TRUE)

plot(network, vertex.size.degree(network)^2)  
# Visualisiert den Datensatz network basierend auf der Größe der Degrees von network. Das Ergebnis wird mit 2 potenziert, damit der Graph einfacher zu lesen ist. Alternativ kann die Größe auch über eine Skalierungsfunktion ausgegeben werden. Ideal für die Darstellung sind Werte zwischen 5 und 15.

#### **Betweenness**

„Die Betweenness-Zentralität eines Knotens steht für die Anzahl der kürzesten Pfade im Netzwerk, die über diesen Knoten laufen“ (Fuhse, 2016, S. 63). Das bedeutet umgangssprachlich, dass ein Knoten, über den man zwangsweise laufen muss, um seine Ziele zu erreichen, einen höheren Betweenness-Wert hat als andere. Stellen Sie sich beispielsweise ein Ego-Netzwerk vor, bei dem mehrere Knoten an das Ego geknüpft, diese jedoch untereinander nicht verbunden sind. Wollen Sie auf kürzestem Weg von einem Knoten zu einem anderen, **müssen** Sie über das Ego laufen. Die Berechnung der Betweenness-Zentralität ist damit – wie fast jedes Zentralitätsmaß – bei einem Ego-Netzwerk unsinnig, da das Ego allein durch die Struktur des Netzwerks die höchste Zentralität hat.

Die Betweenness-Zentralität berechnet man für alle Knoten des Netzwerks mit:

betweenness(network)

#### **Closeness**

„Die **Closeness-Zentralität**misst die Anzahl der kürzesten Pfade (Beziehungen) von einem Akteur zu allen anderen Akteuren. Je größer die Closeness-Zentralität eines Akteurs ist, umso schneller kann er sämtliche direkten und indirekten Kontakte erreichen“ (Sakschewski, o.J., Abschnitt 6, Herv. i. Org.). Die Closeness berechnen sie mithilfe des folgenden Befehls für alle Knoten des Netzwerks:

> closeness(network) # Berechnet die Closeness-Zentralität eines Netzwerks

#### **Eigenvektor**

„Mit dem Zentralitätsmaß "Eigenvektor" wird ermittelt, wie gut eine Entität verbunden ist und wie stark der direkte Einfluss ist, den sie auf andere Entitäten im Netzwerk ausüben kann“ (IBM Corporation, 2017, Abschnitt 7). Knoten mit einem hohen Eigenvektor-Maß sind mit wichtigen Knoten verbunden.

Das Eigenvektor-Maß wird wie folgt für alle Knoten berechnet:

eigen\_centrality(network)

#### **K-Core**

„Coreness ist ein Maß, das bei der Identifizierung von eng miteinander verbundenen Gruppen innerhalb eines Netzwerks hilfreich ist. Der K-Core bezeichnet die größte der Gruppen, deren Entitäten mit mindestens *k* anderen Entitäten innerhalb der Gruppe verbunden sind“ (IBM Corporation, 2017, Abschnitt 10).

Der K-Core kann mithilfe dieses Befehls berechnet werden:

coreness(network)

## VII. Netzwerke visualisieren

### **Einfache Visualisierung**

plot(network)

plot(network, edge.arrow.size=0.4) # Natürlich kann die Zahl je nach Belieben angepasst werden.

plot(network, vertex.label=V(network)$name) # Visualisierung mit Namen Haben Sie das Attribut anders genannt, müssen Sie natürlich Ihre Attributs-Benennung angeben.

plot(network, vertex.label=NA) # Schaltet die Labels ganz aus

colours() # Zeigt alle verfügbaren Farben an

plot(network, vertex.color="pink2") # Färbt die Knoten mit der Farbe pink2

plot(network, vertex.frame.color="red3") # Färbt den Knotenrahmen in red3.  
plot(network, vertex.frame.color="transparent") # Setzt den Knotenrahmen transparent

plot(network, edge.color="yellow") # Färbt die Kanten in yellow

plot(network, vertex.label.family = "Helvetica") # Labels in seriöser Schrift  
plot(network, vertex.label.color = "magenta3") # Färbt die Labels in magenta3.

plot(network, vertex.label.dist=2) # Labels weiter weg von den Knoten

plot(network, main="Das ist der Titel") # gibt Netzwerk Überschrift -> darf keine Umlaute enthalten

plot(network, edge.width=E(network)$weight) # Kantenstärke abhängig von der Gewichtung visualisieren