Network Analysis

Liudmyla Kotusenko (11258622), Yuchen Wang (11273123)

March 2020

1. Introduction

In the past century, networks have surged in popularity and have become universal in our society: from virtual networks like the World Wide Web (web-pages connected by hyperlinks) and the Internet (routers linked by various physical lines) to physical networks such as transportation networks, supply chain networks (brings groceries to your nearest supermarket) as well as networks of neurons in the brain(Franceschet 2020). The recent explosion in social media also gave rise to social networks which have become integral to everyone's daily lives.

Network analysis is a set of techniques that aim to learn and extract insights from networks. It is often employed to deal with large and complex networks that require special algorithms and techniques. The field of network analysis is vast and it could easily take an entire course to gain merely a basic understanding of network analytics (for example, Gilles Caporossi gives a whole course on Complex Networks Analysis at HEC Montreal).

In this paper, we will only scrape the surface of network analysis and focus on several elementary but nevertheless important applications of network analytics. We will first show how to generate networks from a simple dataset in R, how to create basic but powerful visualizations and calculate some basic graph properties. Given its importance and ubiquity, we will also briefly discuss community detection.

As the best way to learn anything is by doing it, we will explore the above-mentioned aspects of network analysis by analyzing a narrative network of relations between characters in fiction novels. The techniques explored are however applicable to all types of networks. The dataset that we'll be working with is generated based on the bestselling book series *A Song of Ice and Fire* by George R.R. Martin. The relationships between characters are generated based on the proximity of the position of character names in the text, an edge (connection) is created every time two character names appear less than 15 words apart.

2. Literature review

Network analysis is rooted in graph theory, a very old and well-established branch of mathematics. The field of graph theory began with Euler's solution to the problem of the Seven Bridges of Königsberg and has been developed extensively due to its usefulness in organic chemistry, electrical engineering and biology. As a result, resources for network analysis are abundant, ranging from textbooks to online open access lecture slides. We have selected a few that were particularly useful and relevant to our study.

In the online lecture slide, Ghoshdastidar (2019) presents an introductory course of network analysis and explains basic network properties and measures. The course also explains more complex models and theorems that are outside the scope of this paper and will not be discussed.

Hevey (2018) presents an overview of applied network analysis in psychology. The paper explores the uses of networks and provides simpler explanations of network measures as well as examples in psychology. This paper also provides an example of how to perform network analysis in R using data on the Theory of Planned Behaviour.

With recent developments in social network analysis, the topic of community detection has surged in popularity. Fortunato (2009) discusses recent advancements and evolution in the problem of community detection and presents some common approaches such as graph partitioning, hierarchical clustering and spectral clustering.

Despite recent developments, the problem of community detection remains vague and elusive. In their 2016 paper, Fortunato and Hric explain in detail why community detection remains difficult: from disagreements in its very definition to non-existent group structures naturally arising out of random graphs.

With regards to implementations of network analysis in R, Luke (2015) provides a comprehensive user's guide to network analytics in R. The book covers wrangling network data in R, its visualization, description and analysis along with network modeling and simulation. The examples of R use are abundant and cover two major R packages such as `statnet` and `igraph`, among others.

Samatova et al also give a useful guide with their work Nagiza F. Samatova (2014). The 2014 book, albeit more mathematical in nature, explains in detail the proximity measures as well as the algorithms implemented in the R libraries for community detection and clustering.

3. Brief description of the methods

Networks are commonly represented as graphs - mathematical structures describing pairwise relations between objects. As such, networks are defined as objects of graph theory (Wikipedia 2020). Each network contains **nodes** (e.g. people in a social network, cities in a flight network of a given air company), which are called **vertices** in the graph theory, and **connections** or **ties** between them, also known as **edges**, **arcs** or **links**.

A Network can also be represented as an **adjacency matrix** (aka a **sociomatrix** in social networks analysis). It is a square matrix with node names in both rows and columns with cells specifying whether the connection between two nodes exists (1) or not (0). Alternatively, instead of 0-1 values, adjacency matrix cells can contain edge weights. The adjacency matrix is symmetric/non-symmetric for undirected/directed networks, respectively.

Before describing the network analysis methods, it is crucial to first define some basic notations and explain some fundamental concepts used in graph theory:

A graph is defined as:

$$G = (V, E)$$

- where V is the set of *nodes* or *vertices* (e.g. people in a social network, cities in a flight network of a given air company)
- and E is the set of *edges* or *links*

Depending on the connections between the nodes, networks can be classified as either **directed** or **undirected**. An **undirected** network has reciprocal connections between nodes (like friends in a social network or two-way roads in the city), whilst a directed one has connections going in one direction only, from a source node to the target one (e.g., followers on social media, a coronavirus spread network, a water supply network or a network of Bixi trips).

Nodes or edges in the network can have labels. For example, for clients of a given company, node labels can contain not only the name of a client but also their sex, age, whether a client is a churner or not etc. (DataCamp 2020).

Networks can be weighted, i.e., each node and/or edge can have a weight (or several weights) assigned to it. For example, in a supply chain network, edges can have weights representing distances between nodes (warehouses), while node weights could describe each warehouse capacity (Caporossi and Camby 2017).

There also exist many special network structure definitions such as bipartite networks, where the nodes can be split into two disjoint and independent sets and every edge connects two nodes from the two sets. One example would be Amazon clients (one set of nodes) rating products (another set of nodes). These networks are however not the focus of this paper and will not be discussed in detail.

We will be focusing on analysis done on an undirected network for the sake of simplicity, but all of our methods can be generalised to directed networks with minimal alteration. Due to the limited scope of this paper, in the network analysis part, we will concentrate on:

- creating a network and exploring its basic properties and statistics, along with network centrality measures;
- network visualization;
- community detection.

Network generation and description

We'll begin by generating a graph from the dataset and explore some basic descriptive statistics for networks such as size, degree, centrality, etc.

- The **Size** of a network specifies the number of nodes, $|V|$, in the network, or occasionally the number of edges $|E|$.
- **Density** of a network is the ratio of the number of edges in the network over the total number of possible edges in the network given the number of nodes $|V|$. In practice, network density is often very low and the number of possible edges grows factorially.

$$Density = \frac{2|E|}{|V|(|V|-1)}$$

- A measure similar to *density*, **transitivity** is another important measure in social networks. It is defined as the ratio of transitive triads (sub-graphs of 3 nodes that are interconnected) over the total number of possible transitive triads that can be formed in a graph with V nodes.

- The **distance** or **graph geodesic**, $d(v_i, v_j)$ between two nodes v_i and v_j of the graph is the shortest path that connects the two nodes together.
- The **Diameter** of a network is defined as the longest geodesic between two nodes in the network. In more technical terms, given the set $S(d)$ of geodesics between all pairs of nodes $< v_i, v_j >$ it is defined as $max\{S\}$. The diameter represents the linear size of the network.
- **Node degree**: the degree of a node in the network is the number of edges connected to the node. In directed graphs we can have *in-degree* (number of inward edges) and *our-degree* (number of outward edges). Given the adjacency matrix A of an undirected graph, we have:

$$deg(v_i) = \sum_j a_{ij}$$

- **Centrality**, commonly denoted as C, is a measure of prominence (importance) of nodes in undirected graphs. In directed graphs it is known as **prestige**. Many centrality measures exist, we will discuss a few common ones in this paper.

  - *Degree centrality*: based on the simple notion that more connected nodes are more prominent than less connected ones, *degree centrality* is quite straightforwardly the degree of nodes.

$$C_d(v_i) = deg(v_i)$$

- *Closeness centrality*: a step further would be to include indirect connections in the analysis. *Closeness centrality* measures the average length of the shortest path between the current node and all other nodes. The prominent nodes are the ones closest to other nodes in the network.

$$C_c(v_i) = \frac{1}{\sum_{v_j} d(v_i, v_j)}$$

- *Harmonic centrality* is a variant of closeness centrality that deals with unconnected graphs.

$$C_h(v_i) = \sum_{v_j \neq v_i} \frac{1}{d(v_i, v_j)}$$

- *Betweenness centrality* measures the number of times a node acts as an intermediary for the shortest path between two other nodes. Given $d_{v_j v_k}$, the geodesic of two nodes v_j and v_k, and $d_{v_j v_k}(v_i)$, the geodesic of two nodes v_j and v_k passing through v_i, the betweenness centrality is defined as:

$$C_b(v_i) = \sum_{v_i \neq v_j \neq v_k} \frac{d_{v_j v_k}(v_i)}{d_{v_j v_k}}$$

- *Eigencentrality* measures the influence of a node in the network. It provides a measure of importance taking into account the importance of its adjacent nodes. Given the adjacency matrix A of an undirected

graph, we can calculate the eigencentrality by finding the eigenvector $\mathbf{x}$ of the adjacency matrix:

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

Community detection (clusters)

In networks, a commonly encountered structure is that some nodes are better connected to each other than to the rest of the network, i.e. the nodes are grouped together. This type of network is said to have community structure and the groups are called communities.

Unfortunately, there is no universally accepted definition of community. The broad definition is that there must be more *edges* within the community than with other communities, but different packages in R implement different definitions depending on their field of application. In addition, there exist many algorithms for community detection and each algorithm is suited to a different field of research. It is impossible to be exhaustive in our example and demonstrate all algorithms available, so we will explore only those from the `igraph` package and select that which yields useful insights in our example.

Network visualizations

We will present methods for visualising graphs to highlight certain graph characteristics. As plotting even a moderately sized network is not a trivial task, we will explore the main principles of network plotting and briefly discuss network plot layouts. Then we will plot our network highlighting different centrality measures, such as node degree and edge weight in one case, and node and edge betweenness in another. We will also plot communities detected by the best performing algorithm. On top of that, we will make several interactive network plots.

4. Resources available in R

There are many packages in R designed to implement network analysis and even more packages that include basic network description and analysis in their functionality. Despite the fact that many packages offer functions that are similar in nature, we have decided to discuss all relevant packages relating to network analysis in the interest of exhaustiveness. Unless otherwise specified, all packages are from the *CRAN* repository.

Two broad frameworks exist in R for the creation and manipulation of networks, the `statnet` package collection and `igraph`. Networks are stored and manipulated as **network** objects in the `statnet` framework and as **igraph** objects in `igraph`. Although it is technically possible to interchange between the two frameworks, it is discouraged as functions in both libraries have naming conflicts and would cause unnecessary complications.

`Statnet` is a collection of packages in network analysis commonly used by statisticians and social scientists such as `network`, `sna`(social network analysis) and `ergm`(exponential-family random graph models). This `statnet` package collection is more comprehensive than `igraph` in the variety of statistical tools offered, but its algorithms are less time-efficient and deal poorly with large datasets. Installing and loading the

`statnet` will automatically load all the packages mentioned below (this list is not exhaustive but mentions the main packages).

- The package `network` and `networkDynamic` (an extension of `network` that allows for the handling temporal data) are packages belonging to the `Statnet` project that includes many tools for the creation, description and manipulation of **network** objects. A **network** object can be created by applying the function *as.network()* to adjacency matrices or edge lists. Once a **network** object has been created, the package offers functions that can implement basic manipulations such as adding nodes and edges or transforming the structure to an adjacency matrix or edge list. Basic descriptive statistics such as *density* are also offered.
- `sna` and its temporal counterpart `tsna` expands the package by offering functions that implement more advanced statistics like the various centrality measures. The package also offers tools more specifically tailored towards social network analysis, such as structural equivalence detection and network regression. The package also offers network visualizations with `gplot()` function that allows for better customization than generic `plot()` function available in `statnet` suite.
- `ergm` and its derivatives (e.g: `ergm.count` and `ergm.ego`) are packages that offer tools to analyze, fit and generate a special type of graphs, the exponential-family random graph models. ERGMS are statistical models and once fitted, these models can be used to conduct statistical inference or prediction on graphical datasets. The popularity of these random network modes have surged recently and have had a substantial impact on social network analysis.

The package `igraph` is another package that offers network analysis and visualization tools in R. Its strength lies in its ability to handle large networks as it is more computationally efficient than the `statnet` suite and offers built-in algorithms for community detection. It has gained in popularity in recent years and its capabilities have been extended by other packages that can work with the **igraph** network object that it generates. Among the most useful are `tidygraph` and `ggraph`, packages built by Thomas Lin Pedersen that "leverage the power of igraph in a manner consistent with the tidyverse workflow". (Sadler 2017)

- The `igraph` package was developed in parallel with the statnet project and has established itself solidly in the R community due to its ability to efficiently handle very large graphs. In `igraph`, networks are stored as *igraph* objects and can be created from data frames of edges using the function *graph_from_data_frame()*. Once created, the *igraph* objects can be manipulated and described much like in the `statnet` packages.
- `igraphdata` is simply a collection of network datasets that has been formatted to work well with the `igraph` package. It includes many famous benchmark datasets like the Enron email network and the karate club network.

In addition to the two main frameworks, other less popular packages exist that offer similar functions. The package `graph` hosted on the Bioconductor repository provides tools for simple network handling. Compared to `statnet` and `igraph`, `graph` offers significantly fewer functionalities but is nevertheless a viable package that implements most of the basic tools used in network analysis.

Furthermore, a wide range of specialised packages also exist that serve specific purposes such as graph

visualisation and manipulation:

- Inspired by the `tidyverse` suite, `tidygraph` provides tools that implement functions capable of manipulating networks in a way that conforms to the "tidy" philosophy. While it is originally designed to work with *igraph* objects, `tidygraph` has since been updated to work on *tbl_graph* objects to which most network object structures can be converted.
- `ggraph` was motivated by the poor fit of the `ggplot2` package on graphs. It expands on `ggplot2` and offers implementations of graphical visualisation methods for networks. Like `ggraph`, `tidygraph` also works with *tbl_graph* objects. It is one of the most comprehensive packages for network graphics.
- `ggnet2` is a function for plotting networks in the `GGally` suite, a collection of packages that extends `ggplot2`. While the visualisation tools provided by `ggnet2` can only be applied on objects of type **network**, the package offers the `asNetwork` function that can coerce **igraph** objects into **network**.
- The package `ggnetwork` is yet another package that implements visualisation tools for networks. It supports equally well **network** and **igraph** objects.

Interactive visualisations of networks can also be implemented with the packages `visNetwork`, `networkD3`, `sigmajs` and `threejs` that make use of javascript. These packages make use of different javascript libraries and deliver slightly different results. For example, `visNetwork` and `networkD3` allow a user to zoom in/out and drag nodes around, while `threejs` allows rotations.

It is often difficult for novice data scientists to explore new packages via the CRAN manuals. The overwhelming number of functions inside each package manual is enough to discourage even those that are the most enthousiastic. As such, we provide below a collection of guides that give quick introductions to some of the aforementioned packages:

Sadler (2017) gives a gentle introduction to creating and plotting network objects first with packages **network** and **igraph** and later introduces network wrangling in a "tidy" way, using packages **tidyverse** and **tidygraph**. **ggraph**, **visNetwork** and **networkD3** package functionalities are used to visualize networks.

DataCamp offers several courses on network analytics in R and Python. A track "Network Analysis with R" includes the following courses (DataCamp 2020):

- *Network Analysis in R* covers creating networks from edge lists and adjacency matrices, plotting networks and their attributes, identifying important vertices and special relationships between vertices, and creating interactive network plots.
- *Predictive Analytics using Networked Data in R* uses network analytics "to predict to which class a network node belongs, such as churner or not, fraudster or not, defaulter or not, etc."
- *Network Analysis in the Tidyverse* shows how network analysis can be done in a tidy way, using R packages such as dplyr, ggplot2, igraph, ggraph and visNetwork.
- *Case Studies: Network Analysis in R* uses Amazon purchase data, Twitter mentions of R and bike-sharing data to demonstrate powers and challenges of network analysis on big real-world data sets.

In the subsequent part, we will focus mainly on the `igraph` package for network analysis and `ggraph` and

`visNetwork` packages for visualization. As interactive visualization is supported in *html* format only, we provide this document in *pdf* and *html* formats.

5. Data analysis example

In this chapter, we provide an example of network analysis. The dataset that we'll be working with is generated based on the bestselling book series by George R.R. Martin, *A Song of Ice and Fire*.This dataset can be found on Kaggle (https://www.kaggle.com/mmmarchetti/game-of-thrones-dataset), though it is believed that the original data source is here: https://github.com/mathbeveridge/asoiaf.

The dataset contains co-occurrences of pairs of characters in the books (connections). Two characters are considered to co-occur if their names appear in the vicinity of 15 words from one another. The number of times each pair of characters appears together in the text is captured in the *weight* field. There is a separate file for each book (1-5). To simplify our analysis, we will be using only the dataset based on the first book of the series, *A Game of Thrones*, to which we will refer as the *GoT dataset*.

5.1 Reading in and preprocessing data

First, we load all the necessary packages.

```
library(igraph)
library(ggplot2)
library(RColorBrewer)
library(ggraph)
library(visNetwork)
library(tidygraph)
library(tidyverse)
library(dplyr)
library(corrplot)
```

Next, we read in the data and quickly explore it.

```
book1 = read.csv("book1.csv")
nrow(book1) # show the number of rows in the dataset
```

```
## [1] 684
```

```
summary(book1)
```

```
##                Source                  Target            Type
##  Eddard-Stark     : 51   Robert-Baratheon: 41   Undirected:684
##  Catelyn-Stark    : 39   Tyrion-Lannister: 41
##  Bran-Stark       : 30   Sansa-Stark     : 31
##  Arya-Stark       : 27   Robb-Stark      : 22
##  Cersei-Lannister : 23   Jon-Snow        : 20
##  Joffrey-Baratheon: 19   Tywin-Lannister : 20
```

```
##  (Other)          :495   (Other)         :509
##      weight            book
##  Min.   :  3.00   Min.   :1
##  1st Qu.:  4.00   1st Qu.:1
##  Median :  5.00   Median :1
##  Mean   : 10.77   Mean   :1
##  3rd Qu.: 11.00   3rd Qu.:1
##  Max.   :291.00   Max.   :1
##
```

```
head(book1,3) # show top-3 lines from the dataset
```

```
##              Source             Target       Type weight book
## 1    Addam-Marbrand    Jaime-Lannister Undirected      3    1
## 2    Addam-Marbrand    Tywin-Lannister Undirected      6    1
## 3 Aegon-I-Targaryen Daenerys-Targaryen Undirected      5    1
```

Our dataframe contains 684 lines, or pairs of characters appearing together in the book 1, with the "weight" column specifying how many times a given pair co-occurred. Columns "Source" and "Target" show connected pairs of characters. Column "Type" mentions "undirected" everywhere: there is no direction in connections between characters. We can remove columns "Type" and "book" since they are of no use to us. Also, we replace hyphens in the character names with spaces, using the `gsub()` command for it.

```
book1 = book1[,c(1:2,4)]
book1$Source = gsub("-"," ", book1$Source)
book1$Target = gsub("-"," ", book1$Target)
head(book1,3)
```

```
##              Source             Target weight
## 1    Addam Marbrand    Jaime Lannister      3
## 2    Addam Marbrand    Tywin Lannister      6
## 3 Aegon I Targaryen Daenerys Targaryen      5
```

5.2 Creating a network and exploring its basic statistics

The next step would be to create a network from this data. There exist several ways to do it: one approach is to create a list of nodes and a list of edges which we can then combine in a network object; another is to represent the network as an **adjacency matrix**.

Adjacency matrices can become overly large and sparse very quickly which complicates their handling. The former approach is thus a more practical one since all major network analysis packages in R allow creating of a network object from vertex and edge lists. The following commands, shown in a form of `package_name::function_name()` pairs, will do this for us:

- `network::network()` (from `statnet` suite of packages)
- `igraph::graph_from_data_frame()`

- `tidygraph::tbl_graph()`

In fact, `network::network()` and `igraph::graph_from_data_frame()` allow us to create a network object right from a dataframe of edges. The latter requires the first two columns to provide pairs of connected nodes. Any other columns are considered as edge attributes.

However, Sadler (2017) warns that *"The network object classes for network, igraph, and tidygraph are all closely related. It is possible to translate between a network object and an igraph object. However, it is best to keep the two packages and their objects separate. In fact, the capabilities of network and igraph overlap to such an extent that it is best practice to have only one of the packages loaded at a time"*.

Given the limitations of this paper, we will focus on the use of `igraph` package to carry out network analysis since it has implemented comprehensive community detection techniques and will also allow us to produce clear plots with the use of `ggraph` package.

As our dataframe already provides the pairs of connected nodes in the first two columns, we can simply use the `graph_from_data_frame()` command to create a network right away. Once we have created our network object, we can use the `summary()` function to explore it.

```
GoT = graph_from_data_frame(d = book1, directed = FALSE)
summary(GoT)
```

```
## IGRAPH d01a644 UNW- 187 684 --
## + attr: name (v/c), weight (e/n)
```

The first line of the summary contains the main information about our graph object. **UNW-** stands for undirected network (**U**) with a name attribute (**N**) and weighted edges (**W**) and which is not bipartite (**-**). This is followed by the number of nodes (187) and edges (684) in the graph (**size** of our network). The second line shows the attributes of the network: **name (v/c), weight (e/n)**. The first attribute refers to vertices and the second to edges which are shown by letters **v** and **e** in parentheses, respectively (Sadler 2017). **/c** and **/n** show the type of our attributes: **character** and **numeric**, respectively.

Functions `V(graph)` and `E(graph)` extract vertices and edges from an `igraph` network, respectively, and can be useful when we need to provide a function with either of them or add/extract a node/edge attribute.

```
head(V(GoT))
```

```
## + 6/187 vertices, named, from d01a644:
## [1] Addam Marbrand                  Aegon I Targaryen
## [3] Aemon Targaryen (Maester Aemon) Aerys II Targaryen
## [5] Aggo                            Albett
```

```
head(E(GoT),3)
```

```
## + 3/684 edges from d01a644 (vertex names):
## [1] Addam Marbrand   --Jaime Lannister    Addam Marbrand   --Tywin Lannister
## [3] Aegon I Targaryen--Daenerys Targaryen
```

We already saw the size of the GoT network while interpreting its summary. Another way to get **size** of a network is to count node and tie numbers with the following functions:

```
vcount(GoT) # get the number of nodes
```

```
## [1] 187
```

```
ecount(GoT) # get the number of ties
```

```
## [1] 684
```

Let's now look at main network statistics. While it's nice to know the total number of connections in our network, it does not tell us much about network interconnectedness. Instead, **network density** does, since it directly demonstrates the share of actual ties in a network among the maximum possible number of ties for this network. Network density takes values from 0 to 1. It appears that the density of the GoT network is quite low.

```
edge_density(GoT)
```

```
## [1] 0.03933069
```

People in a social network have a tendency to form closed triangles (triads): when two people have a friend in common, they often become friends as well. Recall that **transitivity** measures the proportion of closed triangles among existing closed and open triangles. It takes values from 0 to 1. The `igraph` package allows calculation of two types of transitivity: a *global* one, for the whole network, and a *local* one, for each node. In the context of local transitivity, we can calculate the number of triangles (closed and open) each node is a part of with the function `count_triangles`.

```
transitivity(GoT, type = "global") # removing type = "global" yields the same result
```

```
## [1] 0.3302343
```

```
head(transitivity(GoT, type = "local")) # transitivity for the first 6 people
```

```
## [1] 1.0000000 1.0000000 0.5714286 0.6666667 0.7333333 1.0000000
```

```
head(count_triangles(GoT)) # number of triangles for the first 6 people
```

```
## [1]  1  1 12 10 11  3
```

Another important measure of a network is **diameter**, the maximum shortest path between the nodes in a network (Ghoshdastidar 2019). In an unweighted network, it measures the maximum number of hops that separate one node from another and as such is a measure of the compactness of a network. According to Luke (2015), "the diameter reflects the 'worst-case scenario' for sending information (or any other resource) across a network." The diameter is undefined for networks that contain two or more components (parts that have no connections between them). In this case, it is usually best to examine the diameter of the largest component in the network.

If a network is weighted, by default, the call of `diameter()` will account for weights, but the result will be hard to interpret. We specify `weights = NA` in it to get a more intuitive result: we require 7 hops

to go between the farthest nodes in a network. We can also explore the longest shortest path by calling `get_diameter()` function. `farthest_vertices()` will output the pair of nodes with the highest distance between them as well as the distance itself.

```
diameter(GoT, weights = NA) # disable weights
```

```
## [1] 7
```

```
get_diameter(GoT, weights = NA) # explore the longest path between two nodes
```

```
## + 8/187 vertices, named, from d01a644:
## [1] Clement Piper    Karyl Vance      Edmure Tully     Catelyn Stark
## [5] Robert Baratheon Drogo            Ogo              Fogo
```

```
farthest_vertices(GoT, weights = NA)
```

```
## $vertices
## + 2/187 vertices, named, from d01a644:
## [1] Clement Piper Fogo
##
## $distance
## [1] 7
```

We can also calculate **degree** of each node, i.e. the number of nodes it is connected to. Since our network is non-directed, we'll have a single degree measure per node instead of two: **in-degree** and **out-degree**, as is the case in directed networks. The degree of each node in the network is calculated and the top-10 most connected characters are shown below :

```
deg = degree(GoT) # produces a named vector of degrees
head(sort(deg, decreasing = TRUE), 10) # top-10 characters by degree
```

```
##      Eddard Stark  Robert Baratheon  Tyrion Lannister     Catelyn Stark
##                66                50                46                43
##          Jon Snow        Robb Stark       Sansa Stark        Bran Stark
##                37                35                35                32
##  Cersei Lannister Joffrey Baratheon
##                30                30
```

It is possible to save *degree* as a vertex attribute for later use. To add/extract an attribute from a vertex/edge list, we use the command `V(graph)$attribute` or `E(graph)$attribute`. However, if we were given a network with attributes and needed to explore them, getting a list of top-10 most prominent characters would be more tricky than what we did above. We use the `order()` function to sort our degree vector in descending order, take top-10 values from it and apply this sequence to filter our vertices.

```
V(GoT)$degree = degree(GoT)
V(GoT)[head(order(V(GoT)$degree, decreasing = TRUE),10)] # top-10 characters by degree
```

```
## + 10/187 vertices, named, from d01a644:
```

```
##  [1] Eddard Stark      Robert Baratheon  Tyrion Lannister  Catelyn Stark
##  [5] Jon Snow          Robb Stark        Sansa Stark       Bran Stark
##  [9] Cersei Lannister  Joffrey Baratheon
```

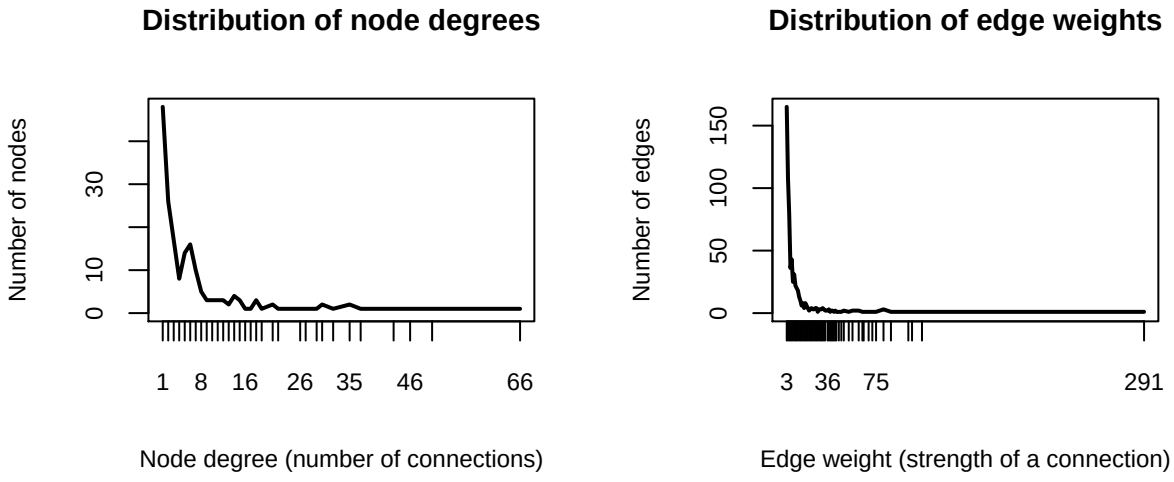
We can use the same commands to see what pairs of characters have the strongest connections (the largest weights associated with their ties).

```
E(GoT)[head(order(E(GoT)$weight, decreasing = TRUE),5)] # top-5 pairs of characters
```

```
## + 5/684 edges from d01a644 (vertex names):
## [1] Eddard Stark      --Robert Baratheon Bran Stark        --Robb Stark
## [3] Arya Stark        --Sansa Stark      Daenerys Targaryen--Drogo
## [5] Joffrey Baratheon --Sansa Stark
```

We can also explore how a number of connections per character and their strengths are distributed. According to Caporossi and Camby (2017), in most complex networks, a few nodes will have a high number of connections while the majority of nodes - a small number.

```
par(mfrow=c(1,2))
plot(table(V(GoT)$degree), type="l", main = "Distribution of node degrees",
     cex.main = 0.9, xlab = "Node degree (number of connections)",
     ylab = "Number of nodes", cex.lab = 0.75, cex.axis = 0.75)
plot(table(E(GoT)$weight), type="l", main = "Distribution of edge weights",
     cex.main = 0.9, xlab = "Edge weight (strength of a connection)",
     ylab = "Number of edges", cex.lab = 0.75, cex.axis = 0.75)
```



Node degrees follow a power law distribution, which we can see on the plot above. Edge weight distribution performs the same on our data set.

If we subset a little part of our network object with square brackets, we can have a glimpse into our adjacency matrix showing weights of our edges (the number of times characters were co-mentioned):

```
GoT[21:25,21:25]
```

```
## 5 x 5 sparse Matrix of class "dgCMatrix"
##                  Bronn Brynden Tully Catelyn Stark Cayn Cersei Lannister
## Bronn                .             .             6    .                .
## Brynden Tully        .             .            16    .                .
## Catelyn Stark        6            16             .    .               12
## Cayn                 .             .             .    .                .
## Cersei Lannister     .             .            12    .                .
```

5.3 Plotting a network

5.3.1 Static plots

Even though our network is not overly large, it's larger than most toy networks used as examples in books and tutorials. Plotting a network even with around 200 nodes is not a trivial task. Edges and nodes may overlay each other, making a plot illegible and useless. Also, there is no single correct way to make a network plot, since the same data can be represented in a huge number of ways, and finding a layout that would make our plot an informative one is another challenge.

"Although there are not in fact an infinite number of ways to display a network on a screen, the number of possibilities might as well be. (For example, consider a moderate sized network of 50 nodes, and a display grid of 10 by 10. In actuality, the display grid would be much larger than this. The first node in the network could be placed in any one of 100 positions, the 2nd node in 99 positions, and so on. In this example, there are 3.1×1093 different possible network layouts.) Most of the possibilities will produce ugly or confusing layouts, therefore there must be some way to pick a layout that has a better than average chance of being visually acceptable." (Luke (2015), p.47).

The author also specifies the guidelines of plotting network data:

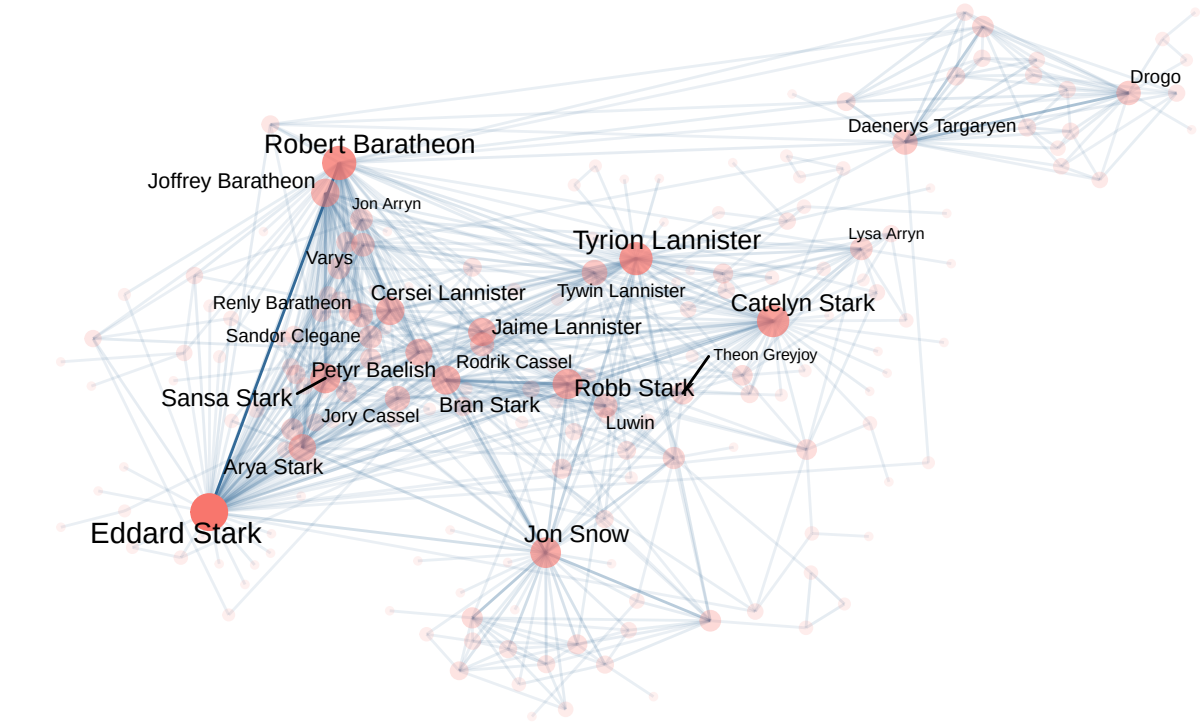
- "*Minimize edge crossings.*
- *Maximize the symmetry of the layout of nodes.*
- *Minimize the variability of the edge lengths.*
- *Maximize the angle between edges when they cross or join nodes.*
- *Minimize the total space used for the network display.*"

Both `statnet` and `igraph` packages have a `plot()` function that can produce network plots, and the latter also has `plot.igraph()` function (equivalent to generic `plot()` function). Both packages have many capabilities to customize plots. We, however, won't cover them here, as our network is too big for these generic functions. Plotting with the use of `statnet` library is covered in detail in Luke (2015) and examples of plotting with `igraph` package are given in Sadler (2017) and in the DataCamp course *Network Analysis in R*.

At the same time, the customization capabilities of the `ggraph` package are more substantial. It extends the power of `ggplot2`'s "grammar of graphics" to networks and allows the building of highly customized plots adding one element at a time. Knowledge of the `ggplot2` package comes in handy here, but if you

never worked with this package, it might be tricky to build it from scratch. We'll make our first plot of GoT network and then go through the code line by line.

```
set.seed(55555) # For reproducibility
ggraph(GoT, layout = "with_dh") +
  geom_edge_link(aes(alpha = weight, edge_colour = 1), show.legend = FALSE) +
  geom_node_point(aes(alpha = degree, size = degree, colour = "red"),
                  show.legend = FALSE) +
  geom_node_text(aes(label = name, filter = degree>=15, size=degree/3),
                 repel=TRUE, show.legend = FALSE) +
  labs(title = "GoT network plot (1): using node degree and tie weight",
       subtitle = "More connected characters and stronger connections are shown bigger/darker") +
  theme_void() # produces a plot similar to theme_graph()
```



By using `ggraph()` command, we signal our intention to build a plot and provide data (a graph object) for it. Within this command, we can also specify a graph layout with an argument `layout`. There exist many possible layouts for a network plot, and choosing one is rather an art than science. To explore the list of possible layouts, we can use the following command.

```
ggraph:::igraphlayouts
```

```
##  [1] "as_bipartite"  "as_star"       "as_tree"       "in_circle"
##  [5] "nicely"        "with_dh"       "with_drl"      "with_gem"
```

```
##  [9] "with_graphopt" "on_grid"       "with_mds"      "with_sugiyama"
## [13] "on_sphere"     "randomly"      "with_fr"       "with_kk"
## [17] "with_lgl"
```

It is possible to read about each layout, for example,with `?layout_with_kk` in the console. All these layouts are described in the `igraph` package documentation, for example, in Csardi (2019).

The *force-directed* class of algorithms aims to place connected nodes closer on the graph while non-connected - apart from one another (Luke 2015). Examples of force-directed layout algorithms include:

- the Fruchterman-Reingold algorithm (`layout = "with_fr"`)
- the Kamada-Kawai algorithm (`layout = "with_kk"`)
- the Davidson-Harel algorithm (`layout = "with_dh"`)

We've experimented with several layouts and found that `layout = "with_dh"` provided the most readable and intuitive plot for the GoT network. It uses the simulated annealing algorithm by Davidson and Harel and produces interesting visualizations but is a slow one, so it should be applied with care on larger networks, similar to other force-directed algorithms. Also, it has a random component to it, so we need to set a random seed to make our plot reproducible. So our first two lines of code become the following:

```
set.seed(55555) # to make a plot reproducible
ggraph(GoT, layout = "with_dh") +
```

Next, we add **edge geometry** with `geom_edge_link()`. This command specifies to a plotting function that we would like our edges to appear on the plot and allows us to customize their look. If we specify nothing in parentheses, all edges will be of the same default format - solid black lines. If we stick to it, our plot will get over-plotted with lines and will be unreadable since we have 684 edges to plot. We'd do better if our edges represented, for example, the strength of the connection between nodes. Recall that our network is weighted and that most connections are weak. We'll check what attributes for nodes and edges the GoT network has:

```
vertex_attr_names(GoT)
```

```
## [1] "name"   "degree"
```

```
edge_attr_names(GoT)
```

```
## [1] "weight"
```

We have **weight** as an attribute of our edges and can use it to format the ties on the plot, for example, to make weak connections more transparent. This should increase the readability of our plot given that the majority of the ties are weak.

To do this, we add aesthetics `aes()` to our edges. Adding the argument `alpha` within `aes()` call allows us to specify transparency of edges. We set `alpha = weight` (note that edge attribute name is provided without quotation marks), so that stronger connections show up darker on a plot.

To make edges colored, we also specify the argument `edge_colour` (`col`, `color` or `colour` would do the same). Unfortunately, there seems to be a bug in the function implementation: even if we specify colors

such as "red", "blue", "green" (in different runs), the edges show up as red all the time. Specifying `edge_colour=1` allows us to get blue edges, but hopefully, this bug will be fixed in the future.

For every element within `aes()` command, `ggraph` will provide a legend. We set `show.legend = FALSE` as the legend for edge transparency and color does not increase the readability of this plot (rather vice versa). Our next line of code becomes as follows:

```
  geom_edge_link(aes(alpha = weight, edge_colour = 1), show.legend = FALSE) +
```

Next, we add **node geometry**. Specifying `geom_node_point()` will show our nodes as points. We would like our points to differ for nodes with higher/lower degree, so we set their transparency and size equal to `degree`. We color them with "red" (the only color that appears correctly when specified manually) and skip the legend again. The third line of code looks as follows:

```
  geom_node_point(aes(alpha = degree, size=degree, colour = "red"), show.legend = FALSE)
```

In this case, nodes with more connections will be bigger and darker. For anonymized networks, this might be enough for a first plot. However, since we know characters well, we would like to add to the plot names of the most prominent ones. We do this with `geom_node_text()` command. Again, we specify some aesthetics for it:

```
  geom_node_text(aes(label = name, filter = degree>=18, size=degree/3), repel=TRUE,
                 show.legend = FALSE)
```

- we set `label` to `name` of a character (recall that we have `name` among vertex attributes);
- we show names only for people who are connected to at least 18 other people (an experimental number that makes our plot readable). We do this by setting `filter = degree>=18`;
- we set character name size equal to the `degree` and scale it with /3 to make it smaller;
- setting `repel=TRUE` will ensure that character names don't overlay each other on the plot;
- once more, don't show the legend.

In the last several lines, we provide a title and subtitle to our plot and specify the theme that will produce our plot on a simple white background. Using `theme_graph()` developed specifically for graphs refuses to compile to pdf (but does compile to html), so we replace it with a similar theme - `theme_void()` in this document. However, in a simple R code, we keep `theme_graph()` since it works better than `theme_void()`.

```
  labs(title = "GoT network plot (1): using node degree and tie weight",
       subtitle = "More connected characters and stronger connections are shown bigger/darker") +
  theme_void() # produces a plot similar to theme_graph()
```

This plot may not represent GoT network the best, but it nevertheless allows us to gain some insights from our network. For example, we see that the strongest connection in the first book of GoT is between Eddard Stark and Robert Baratheon, and that the group of Daenerys and Drogo sets apart from others. Jon Snow seems to be connecting two worlds (which he really does). We also see that the main characters presented here coincide well with those in season one of the movie.

To read more on how to customize ggraph plots, you can check the corresponding vignettes (clickable):

- on layouts
- on nodes
- on edges

5.3.2 Interactive plots

There are two packages in R that allow creating interactive network plots: `visNetwork` and `networkD3`. They require as inputs data frames (or lists) with nodes and edges. We will let a curious reader look at an example of a plot created with `networkD3` in Sadler (2017) and proceed to the creation of an interactive network plot with `visNetwork`. There are several ways to do that:

- Option 1: use its core function `visNetwork()`. It takes as arguments data frames with nodes and edges. We did not create them from our initial data, but we can convert the `igraph` network object to a `visNetwork` object with the function `toVisNetworkData()` and then extract nodes and edges from it.
- Option 2: use `visIgraph()` command passing it the `igraph` object. The function will do the conversion behind the scenes and make a plot.

Here, again, we can specify a network layout. It's not easy to find available layouts in visNetwork documentation. However, using `visIgraphLayout()` allows us to select a layout from the list available in `igraph` package (DataStorm 2017) that we already explored above. To retrieve exact layout names, we can type in the console: `ls("package:igraph", pattern = "^layout_.")` (DataCamp 2020, *Network Analysis in the Tidyverse*).

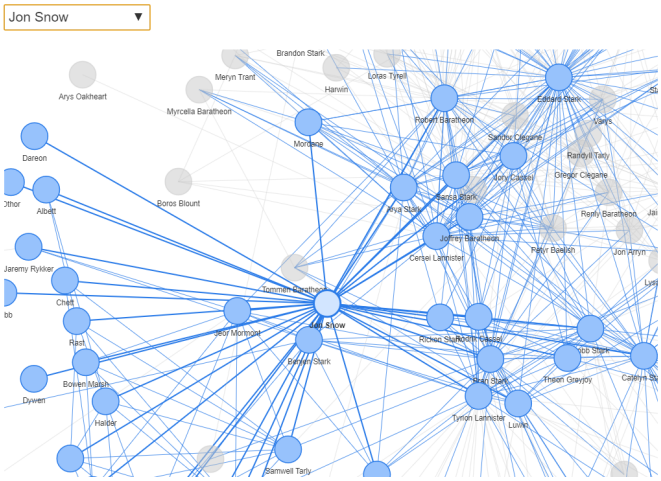
Let's select Kamada-Kawai layout this time by specifying `layout = "layout_with_kk"`. Also, let's create a filter to select nodes by character names and allow our network to highlight a node's nearest neighbours.

Both the Option 1 and Option 2 below will produce identical interactive plots: we can zoom in/out, click on a node and see its neighbours highlighted, move a node to another place, or pick up a character name from the filter. However, we can enjoy this only if we compile our file as an **html** document.

```
# Option 1
g1 = toVisNetworkData(GoT)
visNetwork(nodes = g1$nodes, edges = g1$edges) %>%
  visIgraphLayout(layout = "layout_with_kk") %>% # Select Kamada-Kawai layout
  # Create a filter to select nodes by character names
  # and highlight neighbours of the selected node
  visOptions(nodesIdSelection = TRUE, highlightNearest = TRUE)
```

```
# Option 2
visIgraph(GoT) %>%
  visIgraphLayout(layout = "layout_with_kk") %>%
  visOptions(nodesIdSelection = TRUE, highlightNearest = TRUE)
```

To give some idea of how a plot looks like in the **pdf** document, we show its screenshot. Here, we selected Jon Snow from the filer and zoomed in to investigate a bit more his network.



5.4 Network centrality measures

When exploring a network, we might be interested in finding the most prominent/influential nodes. We already used one measure of vertex centrality - degree centrality - to show more connected characters with bigger nodes and names in the previous plot. However, there are some other centrality measures we can explore, such as **strength**, **closeness**, **betweenness**, and **eigen-centrality**. Please see part 3 for their detailed definitions.

Strength of a node in a weighted network is the sum of weights on all edges adjacent to this node. It measures the extent of engagement of a given node with its neighbourhood. The top-10 nodes by strength overlap to a high extent with top-10 by degree.

```
str = strength(GoT)
head(sort(str, decreasing = TRUE), 10) # top-10 characters by strength
```

```
##       Eddard Stark   Robert Baratheon           Jon Snow   Tyrion Lannister
##               1284                941                784                650
##        Sansa Stark         Bran Stark      Catelyn Stark         Robb Stark
##                545                531                520                516
## Daenerys Targaryen         Arya Stark
##                443                430
```

```
V(GoT)$strength = strength(GoT) # save as a node attribute
```

Closeness centrality measures how many steps, on average, separate one node from any other node. This time, the top-10 list is more different.

```
cls = closeness(GoT)
head(sort(cls, decreasing = TRUE), 10) # top-10 characters by centrality
```

```
## Robert Baratheon Tyrion Lannister  Jaime Lannister     Eddard Stark
##     0.0005446623     0.0005321980     0.0005313496     0.0005197505
##     Loras Tyrell      Sansa Stark      Janos Slynt    Theon Greyjoy
```

```
##     0.0005178664     0.0005165289     0.0005159959     0.0005138746
##    Rodrik Cassel       Arya Stark
##     0.0005128205     0.0005099439
```

```
V(GoT)$closeness = closeness(GoT) # save as a node attribute
```

Betweenness of a node shows how often it lies on the shortest paths between other pairs of nodes. The higher the betweenness, the more crucial the node is in terms of getting or controlling information flow in the network (Luke 2015).

```
btw = betweenness(GoT)
head(sort(btw, decreasing = TRUE), 10) # top-10 characters by betweenness
```

```
## Robert Baratheon     Eddard Stark Tyrion Lannister       Robb Stark
##         4015.971         3217.925         2634.296         1761.825
##    Catelyn Stark         Jon Snow  Jaime Lannister    Rodrik Cassel
##         1749.579         1553.213         1332.546         1319.278
##            Drogo    Jorah Mormont
##         1186.174         1078.900
```

```
V(GoT)$betweenness = betweenness(GoT) # save as a node attribute
```

Eigen-centrality measures the importance of a node by taking into account the importance of its neighbours. Notice how Varys and Petyr Baelish appear among the top-10 characters.

```
# the output of eigen_centrality is a list, need to add $vector
eig = eigen_centrality(GoT)$vector
head(sort(eig, decreasing = TRUE), 10) # top-10 characters by eigen-centrality
```

```
##      Eddard Stark  Robert Baratheon  Cersei Lannister       Sansa Stark
##         1.0000000         0.9158179         0.4234363         0.3314489
##     Petyr Baelish     Catelyn Stark Joffrey Baratheon          Jon Snow
##         0.3256606         0.3250923         0.3025768         0.2975685
##             Varys  Tyrion Lannister
##         0.2758104         0.2706707
```

```
V(GoT)$eigen = eigen_centrality(GoT)$vector
```

Degree and strength are based on the neighbourhood of a vertex and thus are **local** centrality measures. On the other hand, betweenness, closeness, harmonic centrality and eigen-centrality are **global** ones since they account for distances to all other nodes in a network.

Let's combine all the centrality measures in a data frame and explore correlations between them. Correlations are quite high, except for closeness, where they are moderate. Overlap in centrality measures often happens in many real-world networks (Luke 2015).

```r
options(digits = 3) # decrease the number of digits in the output
centrality = as.data.frame(cbind(deg, cls, btw, str, eig))
cor(centrality)
```

```
##       deg   cls   btw   str   eig
## deg 1.000 0.616 0.871 0.955 0.887
## cls 0.616 1.000 0.550 0.483 0.491
## btw 0.871 0.550 1.000 0.857 0.805
## str 0.955 0.483 0.857 1.000 0.923
## eig 0.887 0.491 0.805 0.923 1.000
```

In addition to betweenness of vertices, we can calculate **betweenness of ties**. Similarly to nodes, edges with high betweenness may have considerable power to control information passing through the network. Related to the edge betweenness are the notions of **cutpoints** and **bridges**. In some cases, only a single edge (a bridge) may exist between several sub-groups in a network, and removing it (making a cutpoint there) will disrupt communication between nodes.

Using `edge_betweenness()` is a bit tricky. If edges have weights, the function will use them by default, however, it will treat them as **distances** though they represent the **strength of a connection**. To adjust for this, we need to specify `weights` argument and pass to it a reciprocal of edge `weight` attribute (DataCamp 2020, *Network Analysis in the Tidyverse*). Otherwise, we will get strange results which we won't be able to interpret.

```r
E(GoT)$btw.e.w = edge_betweenness(GoT, weights = 1/E(GoT)$weight) # weighted
E(GoT)[head(order(E(GoT)$btw.e.w, decreasing = TRUE),10)] # top-10 connections
```

```
## + 10/684 edges from d01a644 (vertex names):
##  [1] Eddard Stark      --Robert Baratheon Catelyn Stark     --Eddard Stark
##  [3] Daenerys Targaryen--Robert Baratheon Catelyn Stark     --Tyrion Lannister
##  [5] Eddard Stark      --Jon Snow         Jon Snow          --Tyrion Lannister
##  [7] Bran Stark        --Jon Snow         Daenerys Targaryen--Drogo
##  [9] Jeor Mormont      --Jon Snow         Catelyn Stark     --Robb Stark
```

```r
head(sort(E(GoT)$btw.e.w, decreasing = TRUE),10) # their tie betweenness values
```

```
##  [1] 5257 4519 3465 2865 2773 1900 1149 1084 1084 1058
```

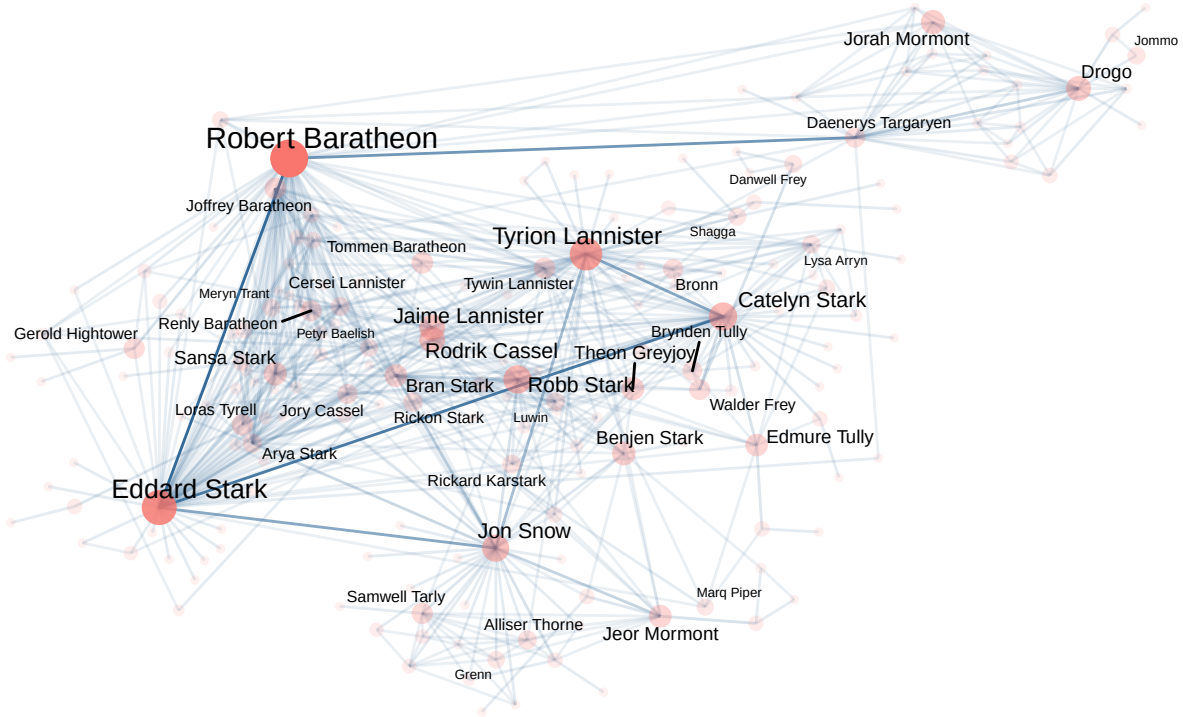
We can also calculate non-weighted edge betweenness - it has weak correlation with weighted betweenness. The latter is a more appropriate measure for us as it accounts for the connection strength.

```r
E(GoT)$btw.e.nw = edge_betweenness(GoT, weights = NA) # use NA to allow non-weighted
cor(E(GoT)$btw.e.w, E(GoT)$btw.e.nw)
```

```
## [1] 0.332
```

We'll plot our network once more, this time highlighting nodes and edges with high betweenness.

```
set.seed(55555)
ggraph(GoT, layout = "with_dh") +
  geom_edge_link(aes(alpha = btw.e.w, edge_colour = 1), show.legend = FALSE) +
  geom_node_point(aes(alpha = betweenness, size=betweenness, colour = "red"),
                  show.legend = FALSE) +
  geom_node_text(aes(label = name, filter = betweenness>=300,size=betweenness/3),
                 repel=TRUE, show.legend = FALSE) +
  labs(title = "GoT network plot (2): using node and edge betweenness",
       subtitle = "Characters and ties with higher betweenness are shown darker") +
  theme_void()
```



Now five more edges are highlighted as important ones in our network. Apart from main characters, several other characters such as Jorah Mormont, Drogo, Benjen Stark appear as important ones that join various parts of the network together. The only darker connection is questionable here: the one going from Robert Baratheon right down, to Daenerys. This is because characters never interact (at least in the movie), but only talk about each other.

5.5 Community detection

Community detection is an important topic in network analysis since it helps to identify subgroups in a network. Depending on an application, communities may be called *clusters*, *groups*, or *modules*. Similar to clustering, community detection is rather an art than science.

Community detection is be very computationally demanding, especially in large networks. This along with other problems like the absence of an universal definition of community, led to the appearance of a large number of community detection algorithms. The package `igraph` is the most comprehensive one with regard to community detection. It implements various algorithms in the following functions:

- `cluster_label_prop`
- `cluster_edge_betweenness`
- `cluster_walktrap`
- `cluster_fast_greedy`
- `cluster_leading_eigen`
- `cluster_spinglass`
- `cluster_infomap`
- `cluster_louvain`
- `cluster_optimal`

Due to the limitations of this paper, we will not provide descriptions of these algorithms. To read more about each function (algorithm), type `?function_name` in the console.

All these algorithms take a graph as an input and produce a **community object** as an output. By default, the 'weight' edge attribute is used as weights if it's available in the data. To switch it off if necessary, we have to specify `weight = NA`. Similarly to `edge_betweenness`, `cluster_edge_betweenness` interprets weights as distances. To use weights in this clustering algorithm, we need to take their inverses.

To dig into the community object, several functions exist:

- `length()`: to get the number of communities detected by an algorithm
- `sizes()`: to get sizes of each community
- `membership()`: to get community labels per each node.

We will apply all algorithms except `cluster_optimal()` to our data set. The algorithm represents an NP-complete problem and runs in exponential time. According to the documentation, *"Graphs with up to fifty vertices should be fine, graphs with a couple of hundred vertices might be possible"*.

```
ca.eb = cluster_edge_betweenness(GoT, weights = 1 / E(GoT)$weight); sizes(ca.eb)
```

```
## Community sizes
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
## 12 23 22  3 19  3 40  2 19 17 12  6  3  4  2
```

```
ca.fg = cluster_fast_greedy(GoT); sizes(ca.fg)
```

```
## Community sizes
##  1  2  3  4  5  6  7
## 26 65 42 26  3 22  3
```

```
ca.lp = cluster_label_prop(GoT); sizes(ca.lp)
```

```
## Community sizes
```

```
##   1   2   3   4   5   6   7   8   9
## 107  22  25  14   6   3   5   3   2
```

```
ca.le <- cluster_leading_eigen(GoT); sizes(ca.le)
```

```
## Community sizes
##   1   2
## 124  63
```

```
ca.lou <- cluster_louvain(GoT); sizes(ca.lou)
```

```
## Community sizes
##  1  2  3  4  5  6  7  8
## 22 26 37 30  3 42  3 24
```

```
ca.wt <- cluster_walktrap(GoT); sizes(ca.wt)
```

```
## Community sizes
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
## 60  3 35  5  3 19 23 19  2  3  2  1  1  1  1  1  1  1  1  1  1  1  1  1
```

```
ca.sg <- cluster_spinglass(GoT); sizes(ca.sg)
```

```
## Community sizes
##  1  2  3  4  5  6  7  8
## 22 43 22 19 26  3  3 49
```

```
ca.info <- cluster_infomap(GoT); sizes(ca.info)
```

```
## Community sizes
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
## 60 31 26 22 15  6  5  5  3  3  3  2  2  2  2
```

Algorithms such as *edge betweenness*, *walktrap* and *infoMAP* detected several very small communities (consisting of just a few characters), inflating the total number of communities. At the same time, *louvain*, *fast_greedy*, *spinglass* and *label_prop* found a reasonable number of communities (7-8). The following code chunk will calculate `length` and `modularity` for all tried algorithms and join results in a data frame.

```
communities = list(ca.lou, ca.fg, ca.sg, ca.wt, ca.info, ca.eb, ca.lp, ca.le)
alg = c("Louvain","Fast-greedy","Spinglass","Walktrap","InfoMAP","Edge betw.",
"Label.prop.","Lead.eigen.")
len = c(); modul = c()
# loop over communities to calculate their length and modularity
for (i in communities) {
  len = c(len, length(i)) # calculate the number of communities
  modul = c(modul, round(modularity(i),3)) # calculate modularity
}
# create a neat data frame to compare different algorithms
```

```r
as.data.frame(cbind(alg,len,modul)) %>%
  arrange(desc(modul)) # sort by modularity in descending order
```

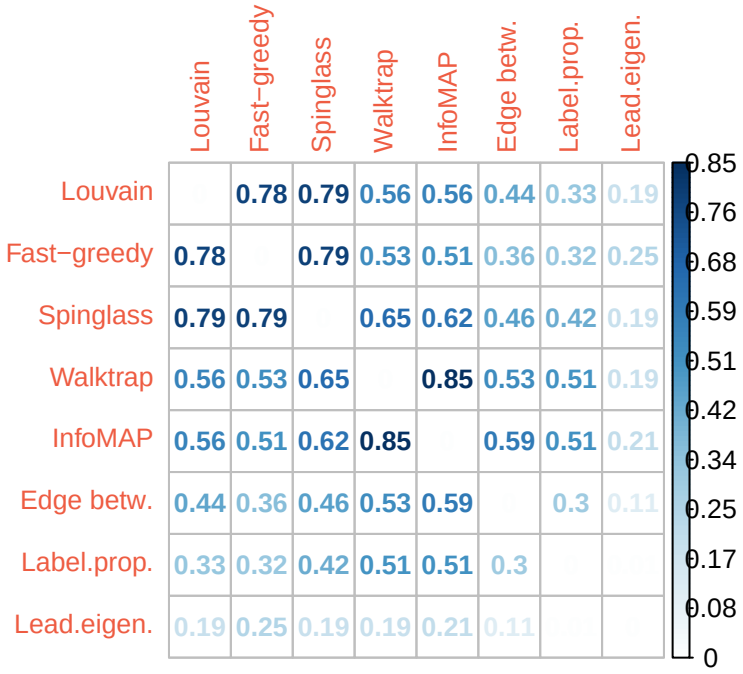
```
##           alg len modul
## 1     Louvain   8 0.512
## 2 Fast-greedy   7  0.51
## 3    Walktrap  24 0.488
## 4     InfoMAP  15 0.487
## 5  Edge betw.  15 0.477
## 6 Label.prop.   9 0.413
## 7 Lead.eigen.   2 0.316
## 8   Spinglass   8 0.058
```

Now that we have generated 8 community (cluster) solutions, the proceeding community will be selected based onthe following: first of all, we can look at modularity: the higher, the better. The *modularity* of a graph measures how good the division of nodes into groups is, or how separated are the different node types from each other, and varies between -0.5 and 1. Communities detected by the majority of algorithms have modularity 0.47 and higher, with Louvain and fast-greedy being at the top of the list and leading eigenvector and spinglass - at the bottom.

Secondly, we can explore how similar the communities detected by different algorithms are. We can compare them using the Rand index. The Rand index is a measure of similarity between two groupings (two cluster solutions) that ranges from 0 to 1. For detail, see https://en.wikipedia.org/wiki/Rand_index.

The function `compare` from the `igraph` package compares two communities at a time. `method = "rand"` will calculate the Rand index, while `method = "adjusted.rand"` will adjust Rand index for the chance grouping of elements. The former produces very high similarity of solutions on our data for 6 out of 8 algorithms (Rand index >= 0.82, not shown here). The latter shows more heterogeneity in communities generated by various algorithms. To get comparisons for all pairs of algorithms, we will run a for loop, save the *adjusted Rand* index in a matrix and visualize it with the `corrplot()` function.

```r
RI = matrix(nrow=8,ncol=8) # initialize a matrix to store the Rand index
for (i in 1:8) {
  for (j in 1:8) {
  RI[i,j] = compare(communities[[i]], communities[[j]], method = "adjusted.rand")
  }
}
rownames(RI) = alg; colnames(RI) = alg; diag(RI) = 0 # set row/col names and diagonal to 0
corrplot(RI, method = "number", is.corr = FALSE, tl.col = "tomato2", tl.cex = 0.8,
         number.cex = 0.8)
```



Surprisingly, *spinglass*, which generated communities with the lowest modularity, produces solutions very similar to those by *louvain* and *fast-greedy* algorithms which rank high by modularity. Also, *walktrap* and *infoMAP* produce very similar solutions, despite having a different number of communities.

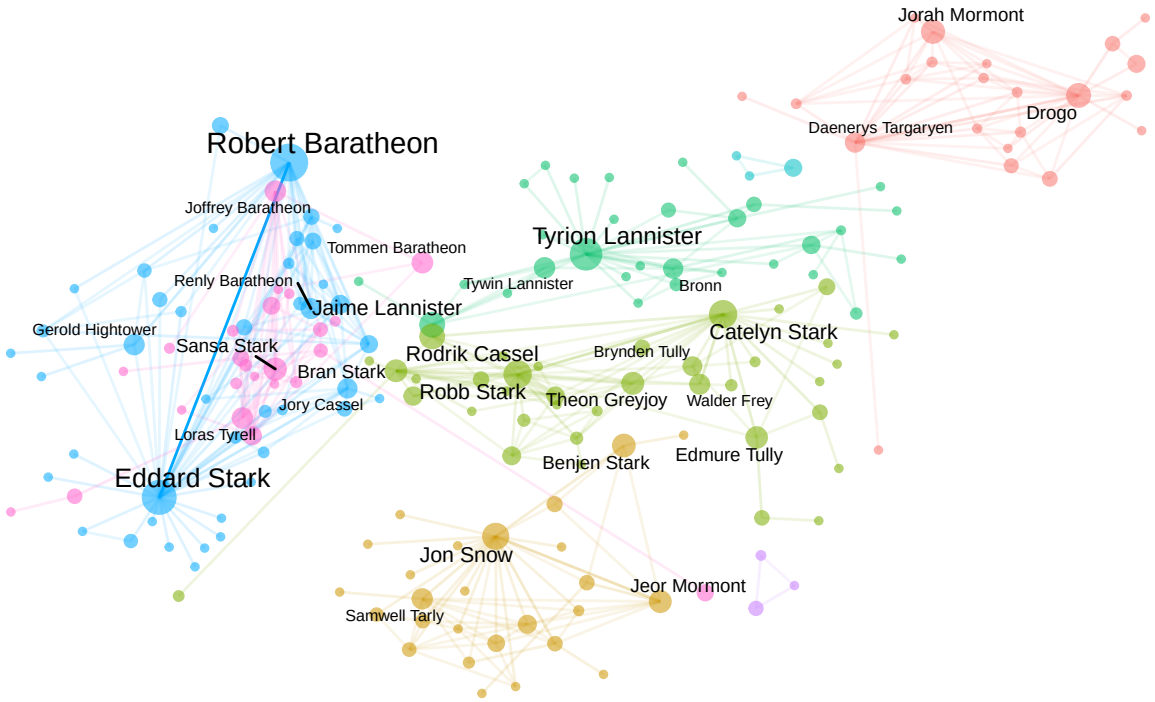
Let's save and visualize communities detected by *Louvain* algorithm. This solution has the highest modularity, produces a reasonable number of communities and is also similar to several other solutions. We will plot communities and edges between them with different colors. However, to color edges with the same colors as nodes, we need to make some data transformations to create edge color attributes.

```
# TO COLOR EDGES WITH CLUSTER COLORS
# create node data frame with numeric node IDs useful for matching nodes and edges
from = book1 %>% distinct(Source)
to = book1 %>% distinct(Target)
nodes = full_join(from, to, by=c("Source"="Target"))
names(nodes) = "label" # rename the column
nodes = rowid_to_column(nodes, var = "id") # create a column with numeric node IDs
# create a data frame with edges
edges = book1 %>%
  left_join(nodes, by = c("Source" = "label")) %>%  # join "id" for "Source"
  rename(from = id) %>%  # rename "id" that we joined into "from"
  left_join(nodes, by = c("Target" = "label")) %>%  # join "id" for "Target"
  rename(to = id) %>% # rename "id" that we joined into "to"
  select(from, to, weight)
V(GoT)$community = membership(ca.lou) # save louvain community membership
# create edge color attribute: edges within the same community colored as nodes,
# edges between communities having color 9999
# this function requires numeric IDs, does not work with character names
```

```
E(GoT)$color <- apply(edges, 1,
                function(x) ifelse(V(GoT)$community[x[1]] == V(GoT)$community[x[2]],
                                   V(GoT)$community[x[1]], 9999))
```

After these long preparations, we can make a plot with communities detected.

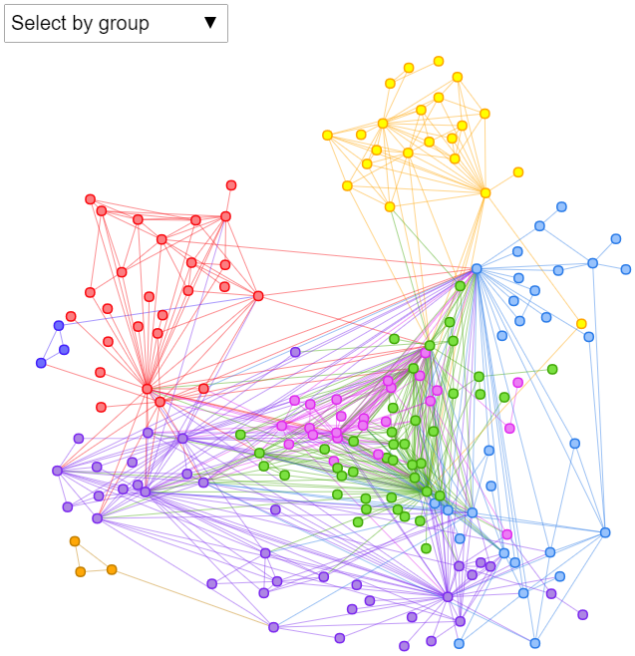
```
V(GoT)$community = membership(ca.lou)
set.seed(55555)
ggraph(GoT, layout = "with_dh") +
  # we use colour, fill and scale_fill_identity() to color communities in different colors
  # we don't show links between communities with filter=color!=9999
  geom_edge_link(aes(alpha = btw.e.w, color = factor(color), filter=color!=9999),
                 show.legend = FALSE) +
  geom_node_point(aes(alpha = 0.8, size=betweenness, colour = factor(community),
                      fill = factor(community)), show.legend = FALSE) +
  scale_fill_identity() +
  geom_node_text(aes(label = name, filter = betweenness>=500, size=betweenness/3),
                 repel=TRUE, show.legend = FALSE) +
  labs(title = "Eight GoT communities found by louvain algorithm",
       subtitle = "Characters and ties with higher betweenness are shown bigger/darker;
       only ties within communities are shown") +
  theme_void()
```



Cluster analysis results should be interpretable. This refers to community detection as well. Exploring the communities detected shows that they make sense: there is one big community of "the King and his retinues" (Robert Baratheon, Eddard Stark, Cercei, Petyr Baelish etc.) and five smaller ones: "Jon Snow and the North", "Daenerys and the East", "Lannisters and their friends", "Catelyn Stark and her sons", "Youngsters in King's Landing". There are also two triads which we won't analyse.

Finally, to dig into communities further, we can make an interactive plot. You can play around with it only in **html** version.

```
V(GoT)$color = V(GoT)$community # create color attribute equal to a cluster number
set.seed(333)
visIgraph(GoT) %>%
  visIgraphLayout(layout = "layout_with_dh") %>%
  # selectedBy = "group" will create a filter with cluster numbers
  visOptions(highlightNearest = TRUE, selectedBy = "group")
```



References

Caporossi, Gilles, and Eglantine Camby. 2017. "Complex Networks Analysis." Course notes. HEC Montreal.

Csardi, Gabor. 2019. "Package 'Igraph'." Web Page. https://cran.r-project.org/web/packages/igraph/igraph.pdf.

DataCamp. 2020. "Network Analysis with R." Online skill track. https://learn.datacamp.com/skill-tracks/analyzing-networks-with-r.

DataStorm. 2017. "VisNetwork, an R Package for Interactive Network Visualization." https://datastorm-

open.github.io/visNetwork.

Fortunato, Santo. 2009. “Community Detection in Graphs.” Journal Article. *Physics Reports* 486 (2). https://doi.org/10.1016/j.physrep.2009.11.002.

Fortunato, Santo, and Darko Hric. 2016. “Community Detection in Networks: A User Guide.” Journal Article. *Physics Reports* 659 (3). https://doi.org/10.1016/j.physrep.2016.09.002.

Franceschet, Massimo. 2020. “Network Examples.” Web Page. University of Udine. https://users.dimi.uniud.it/~massimo.franceschet/networks/nexus/examples.html.

Ghoshdastidar, Debarghya. 2019. “Statistical Network Analysis.” Web Page. University of Tuebingen. https://bit.ly/3aHtA69.

Hevey, David. 2018. “Network Analysis: A Brief Overview and Tutorial.” Journal Article. *Health Psychology and Behavioral Medicine* 6 (1). https://doi.org/10.1080/21642850.2018.1521283.

Luke, Douglas A. 2015. *A User’s Guide to Network Analysis in R*. Book. New York: Springer-Verlag. http://www.springer.com/9783319238821.

Nagiza F. Samatova, John Jenkins, William Hendrix. 2014. *Practical Graph Mining with R*. Book. United Kingdom: Taylor & Francis. https://www.crcpress.com/Practical-Graph-Mining-with-R/Samatova-Hendrix-Jenkins-Padmanabhan-Chakraborty/p/book/9781439860847.

Sadler, Jesse. 2017. “Introduction to Network Analysis with R. Creating Static and Interactive Network Graphs.” Web Page. https://www.jessesadler.com/post/network-analysis-with-r/.

Wikipedia. 2020. “Graph Theory.” Web Page. https://en.wikipedia.org/wiki/Graph_theory.