But there are two packages that integrate network analysis much more nicely with the tidyverse: tidygraph and ggraph. These, I am going to show how to use for analyzing yet another network of characters from **A Song of Ice and Fire** / **Game of Thrones** (to be correct, this new network here is strictly based on the **A Song of Ice and Fire** books, NOT on the TV show).

**What can network analysis tell us?**

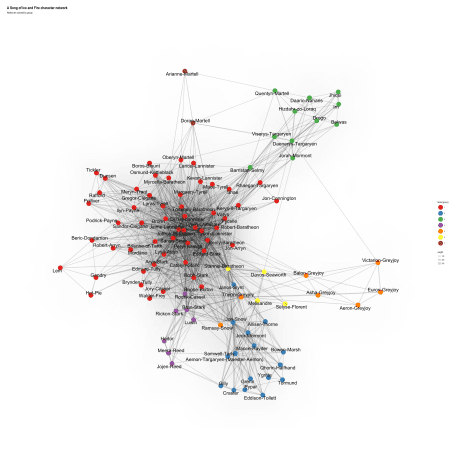
Network analysis can e.g. be used to explore relationships in social or professional networks. In such cases, we would typically ask questions like:

* How many connections does each person have?
* Who is the most connected (i.e. influential or “important”) person?
* Are there clusters of tightly connected people?
* Are there a few key players that connect clusters of people?
* etc.

These answers can give us a lot of information about the patterns of how people interact.

So, how do we find out who the most important characters are in this network? We consider a character “important” if he has connections to many other characters. There are a few network properties, that tell us more about this, like node centrality and which characters are key-players in the books.

**A word of caution before you read on: BEWARE of SPOILERS for all books!**



A Song of Ice and Fire character network across all five books; find out how I made it by following the code below…

library(readr) # fast reading of csv files

library(tidyverse) # tidy data analysis

library(tidygraph) # tidy graph analysis

library(ggraph) # for plotting

**The Data**

I obtained the following data and attached on this repository:

Character Interaction Networks for George R. R. Martin’s “A Song of Ice and Fire” saga These networks were created by connecting two characters whenever their names (or nicknames) appeared within 15 words of one another in one of the books in “A Song of Ice and Fire.” The edge weight corresponds to the number of interactions. You can use this data to explore the dynamics of the Seven Kingdoms using network science techniques. For example, community detection finds coherent plotlines. Centrality measures uncover the multiple ways in which characters play important roles in the saga.

Here, I don’t aim to replicate his analyses but I want to show how you could do this or similar analyses with tidygraph and ggraph. Thus, I am also not going to use all of his node and edge files.

path <- "/Users/ Github/binmishr/"

files <- list.files(path = path, full.names = TRUE)

**Characters across all books**

The first data set I am going to use are the character interactions in all five books. I am not using the node files here, because I find the edge names sufficient for this demonstration. If you wanted to have nice name labels, you could use the node files.

cooc\_all\_edges <- read\_csv(files[1])

Because there are so many characters in the books, many of them minor, I am subsetting the data to the 100 characters with the most interactions across all books.The edges are undirected, therefore there are no redundant Source-Target combination; because of this, I gathered Source and Target data before summing up the weights.

main\_ch <- cooc\_all\_edges %>%

select(-Type) %>%

gather(x, name, Source:Target) %>%

group\_by(name) %>%

summarise(sum\_weight = sum(weight)) %>%

ungroup()

main\_ch\_l <- main\_ch %>%

arrange(desc(sum\_weight)) %>%

top\_n(100, sum\_weight)

main\_ch\_l

## # A tibble: 100 x 2

## name sum\_weight

##

## 1 Tyrion-Lannister 2873

## 2 Jon-Snow 2757

## 3 Cersei-Lannister 2232

## 4 Joffrey-Baratheon 1762

## 5 Eddard-Stark 1649

## 6 Daenerys-Targaryen 1608

## 7 Jaime-Lannister 1569

## 8 Sansa-Stark 1547

## 9 Bran-Stark 1508

## 10 Robert-Baratheon 1488

## # ... with 90 more rows

cooc\_all\_f <- cooc\_all\_edges %>%

filter(Source %in% main\_ch\_l$name & Target %in% main\_ch\_l$name)

**tidygraph and ggraph**

With tidygraph I set out to make it easier to get your data into a graph and perform common transformations on it, but the aim has expanded since its inception. The goal of tidygraph is to empower the user to formulate complex questions regarding relational data as simple steps, thus enabling them to retrieve insights directly from the data itself. The central idea this all boils down to is this: you don’t have to plot a network to understand it. While I absolutely love the field of network visualisation, it is in many ways overused in data science — especially when it comes to extracting knowledge from a network. Just as you don’t need a plot to tell you which car in a dataset is the fastest, you don’t need a plot to tell you which pair of friends are the closest. What you do need, instead of a plot, is a tool that allow you to formulate your question into a logic sequence of operations. For many people in the world of rectangular data, this tool is increasingly dplyr (and friends), and I do hope that tidygraph can take on the same role in the world of relational data.

The first step is to convert our edge table into a tbl\_graph object structure. Here, we use the as\_tbl\_graph() function from tidygraph; it can take many different types of input data, like data.frame, matrix, dendrogram, igraph, etc.

Underneath the hood of tidygraph lies the well-oiled machinery of igraph, ensuring efficient graph manipulation. Rather than keeping the node and edge data in a list and creating igraph objects on the fly when needed, tidygraph subclasses igraph with the tbl\_graph class and simply exposes it in a tidy manner. This ensures that all your beloved algorithms that expects igraph objects still works with tbl\_graph objects. Further, tidygraph is very careful not to override any of igraphs exports so the two packages can coexist quite happily.

Code Chunks Of TidyGraph

Tidy network data?

There’s a discrepancy between relational data and the tidy data idea, in that relational data cannot in any meaningful way be encoded as a single tidy data frame. On the other hand, both node and edge data by itself fits very well within the tidy concept as each node and edge is, in a sense, a single observation. Thus, a close approximation of tidyness for relational data is two tidy data frames, one describing the node data and one describing the edge data.

The tbl\_graph object

Underneath the hood of **tidygraph** lies the well-oiled machinery of igraph, ensuring efficient graph manipulation. Rather than keeping the node and edge data in a list and creating **igraph** objects on the fly when needed, **tidygraph** subclasses **igraph** with the **tbl\_graph** class and simply exposes it in a tidy manner. This ensures that all your beloved algorithms that expects **igraph** objects still works with **tbl\_graph** objects. Further, **tidygraph** is very careful not to override any of **igraph**s exports so the two packages can coexist quite happily.

To underline the tidyness of the **tbl\_graph** class the print method shows the object as two tibbles along with additional network information.

library(tidygraph)

create\_ring(10)

*#> # A tbl\_graph: 10 nodes and 10 edges*

*#> #*

*#> # An undirected simple graph with 1 component*

*#> #*

*#> # Node Data: 10 x 0 (active)*

*#> #*

*#> # Edge Data: 10 x 2*

*#> from to*

*#> <int> <int>*

*#> 1 1 2*

*#> 2 2 3*

*#> 3 3 4*

*#> # ... with 7 more rows*

**tbl\_graph** objects can be created directly using the **tbl\_graph()** function that takes a node data.frame and an edge data.frame. On top of that, **tidygraph** also provides coercion from a huge amount of relational data structures. The following list gives the packages/classes that can currently be converted to **tbl\_graph**s, using the **as\_tbl\_graph** function:

* **data.frame**, **list**, **matrix** from **base**
* **igraph** from **igraph**
* **network** from **network**
* **dendrogram** and **hclust** from **stats**
* **Node** from **data.tree**
* **phylo** and **evonet** from **ape**
* **graphNEL**, **graphAM**, **graphBAM** from **graph** (in Bioconductor)

For all of the coercions you can expect that data on the nodes and edges are kept and available after conversion to **tbl\_graph**:

iris\_clust <- hclust(dist(iris[1:4]))

iris\_tree <- as\_tbl\_graph(iris\_clust)

iris\_tree

*#> # A tbl\_graph: 299 nodes and 298 edges*

*#> #*

*#> # A rooted tree*

*#> #*

*#> # Node Data: 299 x 4 (active)*

*#> height leaf label members*

*#> <dbl> <lgl> <fctr> <int>*

*#> 1 0.0000000 TRUE 108 1*

*#> 2 0.0000000 TRUE 131 1*

*#> 3 0.2645751 FALSE 2*

*#> 4 0.0000000 TRUE 103 1*

*#> 5 0.0000000 TRUE 126 1*

*#> 6 0.0000000 TRUE 130 1*

*#> # ... with 293 more rows*

*#> #*

*#> # Edge Data: 298 x 2*

*#> from to*

*#> <int> <int>*

*#> 1 3 1*

*#> 2 3 2*

*#> 3 7 5*

*#> # ... with 295 more rows*

Lastly, **tidygraph** also wraps the multitude of graph constructors available in **igraph** and exports them under the **create\_\*()** family of functions for deterministic constructors (e.g. the call to **create\_ring(10)** above) and the **play\_\*()** family for constructors that incorporate sampling (e.g. **play\_erdos\_renyi()** for creating graphs with a fixed edge probability). All of these functions provide a consistent argument naming scheme to make them easier to use and understand.

Meet a new verb…

There are many ways a multitable setup could fit into the tidyverse. There could be an added qualifier to the verbs such as **mutate\_nodes()** and **filter\_edges()** or each verb could take an additional argument specifying what is targeted e.g. **arrange(..., target = 'nodes')**. Both of these approachable are viable but would require a huge amount of typing as well as being taxing to support down the line.

The approach used by **tidygraph** is to let the data object itself carry around a pointer to the active data frame that should be the target of manipulation. This pointer is changed using the **activate()** verb, which, on top of changing which part of the data is being worked on, also changes the print output to show the currently active data on top:

iris\_tree %>% activate(edges)

*#> # A tbl\_graph: 299 nodes and 298 edges*

*#> #*

*#> # A rooted tree*

*#> #*

*#> # Edge Data: 298 x 2 (active)*

*#> from to*

*#> <int> <int>*

*#> 1 3 1*

*#> 2 3 2*

*#> 3 7 5*

*#> 4 7 6*

*#> 5 8 4*

*#> 6 8 7*

*#> # ... with 292 more rows*

*#> #*

*#> # Node Data: 299 x 4*

*#> height leaf label members*

*#> <dbl> <lgl> <fctr> <int>*

*#> 1 0.0000000 TRUE 108 1*

*#> 2 0.0000000 TRUE 131 1*

*#> 3 0.2645751 FALSE 2*

*#> # ... with 296 more rows*

As can be seen, **activate()** takes a single argument specifying the part of the data that should be targeted for subsequent operations as an unquoted symbol. **tidygraph** continues the naming conventions from **ggraph** using **nodes** and **edges** to denote the entities and their connections respectively, but **vertices** and **links** are allowed synonyms inside **activate()**.

The current active data can always be extracted as a tibble using **as\_tibble()**

as\_tibble(iris\_tree)

*#> # A tibble: 299 x 4*

*#> height leaf label members*

*#> <dbl> <lgl> <fctr> <int>*

*#> 1 0.0000000 TRUE 108 1*

*#> 2 0.0000000 TRUE 131 1*

*#> 3 0.2645751 FALSE 2*

*#> 4 0.0000000 TRUE 103 1*

*#> 5 0.0000000 TRUE 126 1*

*#> 6 0.0000000 TRUE 130 1*

*#> 7 0.3464102 FALSE 2*

*#> 8 0.5196152 FALSE 3*

*#> 9 0.5567764 FALSE 5*

*#> 10 0.0000000 TRUE 119 1*

*#> # ... with 289 more rows*

The dplyr verbs

Using **activate()** it is possible to use the well known **dplyr** verbs as one would expect without much hassle:

iris\_tree <- iris\_tree %>%

activate(nodes) %>%

mutate(Species = ifelse(leaf, as.character(iris$Species)[label], NA)) %>%

activate(edges) %>%

mutate(to\_setose = .N()$Species[to] == 'setosa')

iris\_tree

*#> # A tbl\_graph: 299 nodes and 298 edges*

*#> #*

*#> # A rooted tree*

*#> #*

*#> # Edge Data: 298 x 3 (active)*

*#> from to to\_setose*

*#> <int> <int> <lgl>*

*#> 1 3 1 TRUE*

*#> 2 3 2 TRUE*

*#> 3 7 5 TRUE*

*#> 4 7 6 TRUE*

*#> 5 8 4 TRUE*

*#> 6 8 7 NA*

*#> # ... with 292 more rows*

*#> #*

*#> # Node Data: 299 x 5*

*#> height leaf label members Species*

*#> <dbl> <lgl> <fctr> <int> <chr>*

*#> 1 0.0000000 TRUE 108 1 setosa*

*#> 2 0.0000000 TRUE 131 1 setosa*

*#> 3 0.2645751 FALSE 2 <NA>*

*#> # ... with 296 more rows*

In the above the **.N()** function is used to gain access to the node data while manipulating the edge data. Similarly **.E()** will give you the edge data and **.G()** will give you the **tbl\_graph** object itself.

Some verbs have effects outside of the currently active data. **filter()**/**slice()** on node data will remove the edges terminating at the removed nodes and **arrange()** on nodes will change the indexes of the **to** and **from** column in the edge data.

While one might expect all of **dplyr**s verbs to be supported in that manner, there is a clear limitation in the relational data structure that requires rows to maintain their identity. Thus, **summarise()** and **do()** are not allowed as there is no clear interpretation of how alterations on the node and edge data with these verbs should be interpreted. If these operations are required I suggest applying them to a tibble representation and then joining the result back in.

Speaking of joining, all joins from **dplyr** are supported. Nodes and edges are added and removed as required by the join. New edge data to be joined in must have a **to** and **from** column referencing valid nodes in the existing graph.

library(dplyr)

iris\_sum <- iris %>%

group\_by(Species) %>%

summarise\_all(mean) %>%

ungroup()

iris\_tree <- iris\_tree %>%

activate(nodes) %>%

left\_join(iris\_sum)

iris\_tree

*#> # A tbl\_graph: 299 nodes and 298 edges*

*#> #*

*#> # A rooted tree*

*#> #*

*#> # Node Data: 299 x 9 (active)*

*#> height leaf label members Species Sepal.Length Sepal.Width*

*#> <dbl> <lgl> <fctr> <int> <chr> <dbl> <dbl>*

*#> 1 0.0000000 TRUE 108 1 setosa 5.006 3.428*

*#> 2 0.0000000 TRUE 131 1 setosa 5.006 3.428*

*#> 3 0.2645751 FALSE 2 <NA> NA NA*

*#> 4 0.0000000 TRUE 103 1 setosa 5.006 3.428*

*#> 5 0.0000000 TRUE 126 1 setosa 5.006 3.428*

*#> 6 0.0000000 TRUE 130 1 setosa 5.006 3.428*

*#> # ... with 293 more rows, and 2 more variables: Petal.Length <dbl>,*

*#> # Petal.Width <dbl>*

*#> #*

*#> # Edge Data: 298 x 3*

*#> from to to\_setose*

*#> <int> <int> <lgl>*

*#> 1 3 1 TRUE*

*#> 2 3 2 TRUE*

*#> 3 7 5 TRUE*

*#> # ... with 295 more rows*

Expanding the vocabulary

On top of what has been showed so far, **tidygraph** provides an assortment of graph specific verbs that can be used to power your analysis and manipulation. Analogous to **bind\_rows()**, **tidygraph** provides three functions to expand your data: **bind\_nodes()** and **bind\_edges()** append nodes and edges to the graph respectively. As with the join functions **bind\_edges()** must contain valid **from** and **to** columns. **bind\_graphs()** allows you to combine multiple graphs in the same graph structure resulting in each original graph to become a component in the returned graph.

While **bind\_graphs()** cannot be used to create edges between the merged graphs **graph\_join()** can do just that. It merges nodes using a **full\_join()** semantic and keeps the individual edges from both graphs:

library(ggraph)

gr1 <- create\_notable('bull') %>%

mutate(name = letters[1:5])

gr2 <- create\_ring(5) %>%

mutate(name = letters[4:8])

*# Plot*

gr1 %>% bind\_graphs(gr2) %>%

ggraph(layout = 'kk') +

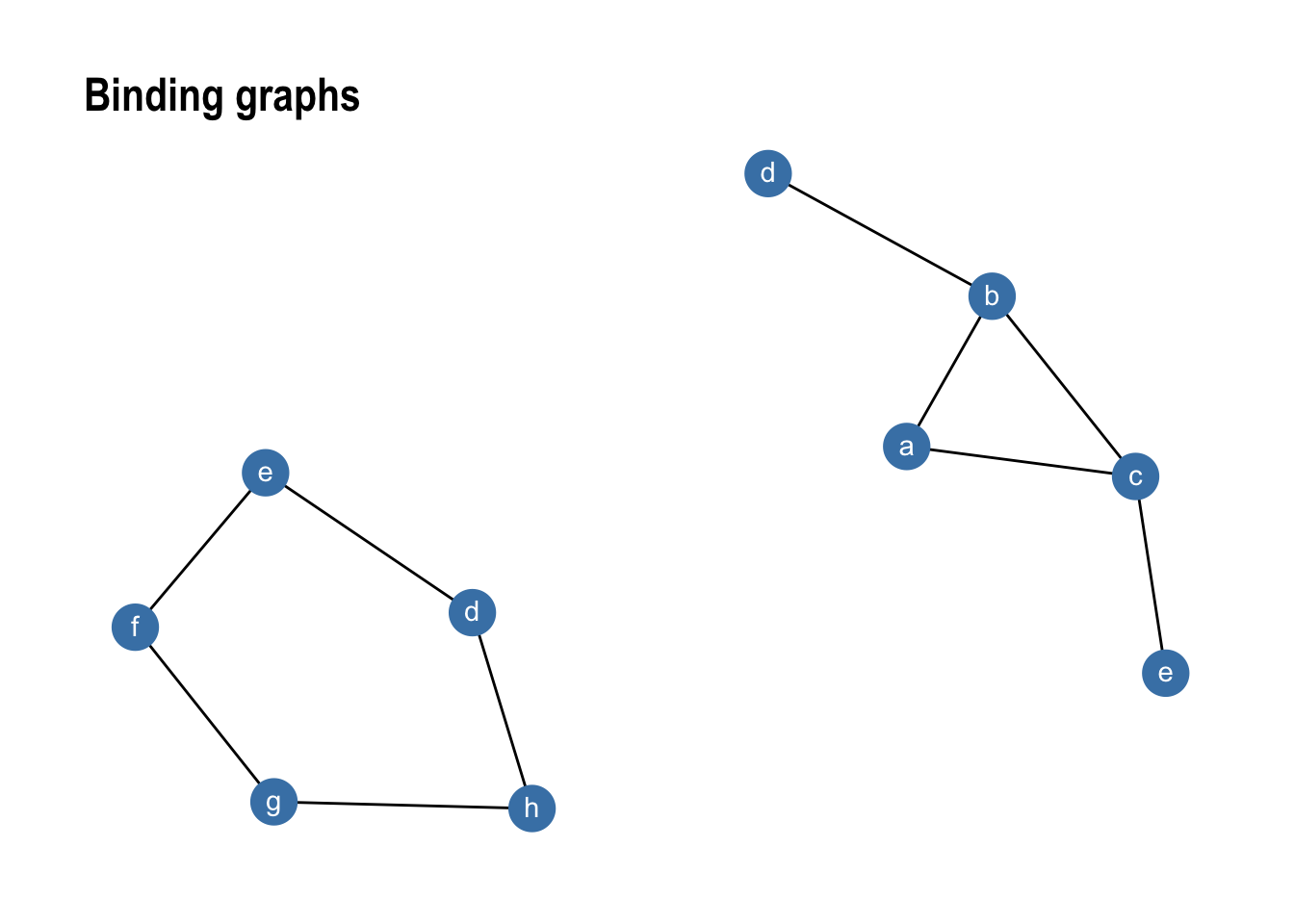
geom\_edge\_link() +

geom\_node\_point(size = 8, colour = 'steelblue') +

geom\_node\_text(aes(label = name), colour = 'white', vjust = 0.4) +

ggtitle('Binding graphs') +

theme\_graph()



gr1 %>% graph\_join(gr2) %>%

ggraph(layout = 'kk') +

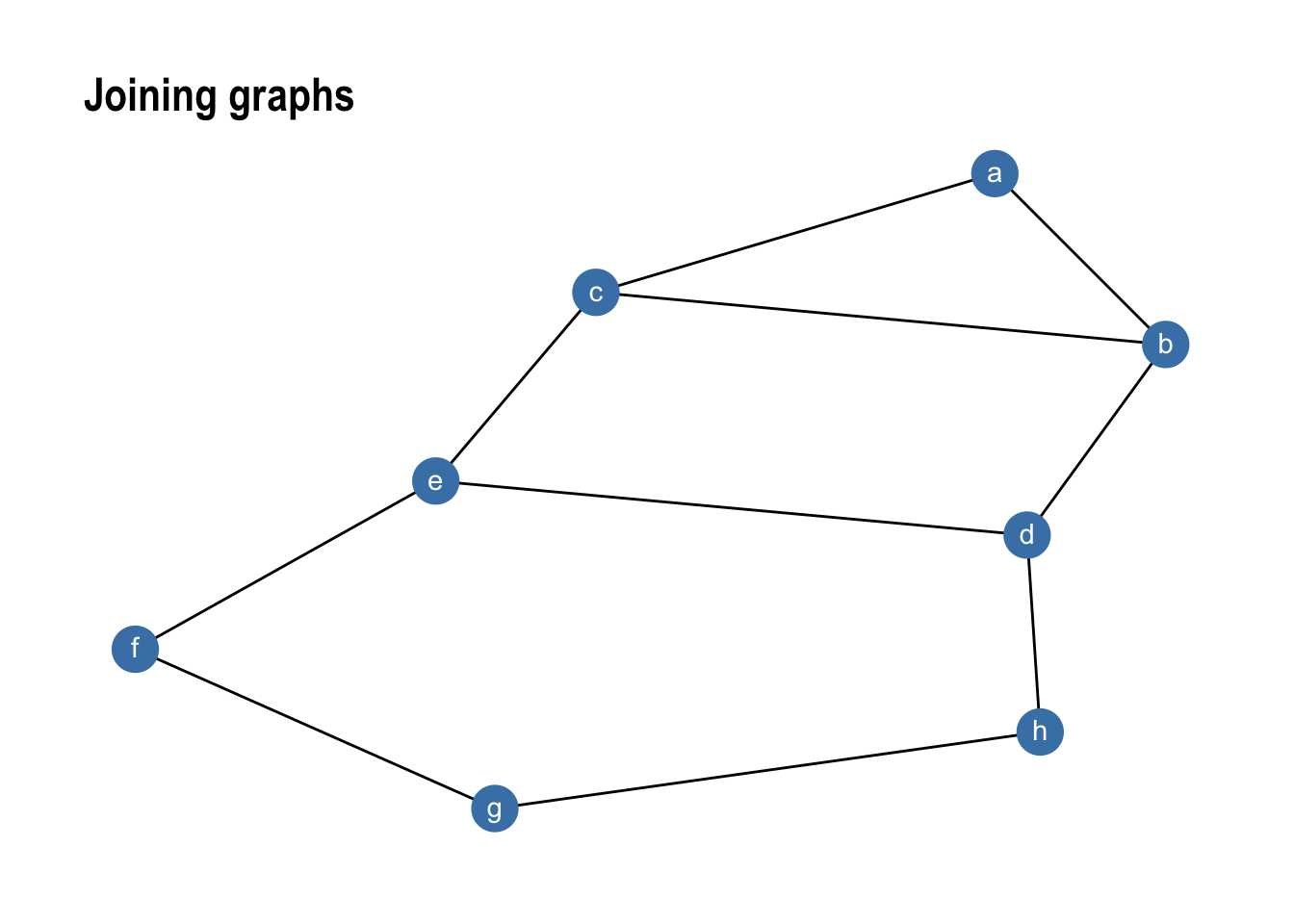
geom\_edge\_link() +

geom\_node\_point(size = 8, colour = 'steelblue') +

geom\_node\_text(aes(label = name), colour = 'white', vjust = 0.4) +

ggtitle('Joining graphs') +

theme\_graph()



The standard **dplyr** verbs protects the **to** and **from** columns in the edge data in order to avoid accidental modification of the graph topology. If changing of the terminal nodes are necessary the **reroute()** verb will come in handy:

gr1 <- create\_star(6, directed = TRUE)

layout <- create\_layout(gr1, layout = 'fr')

gr1 <- gr1 %>%

mutate(x = layout$x, y = layout$y, graph = 'original')

gr2 <- gr1 %>%

mutate(graph = 'reverse') %>%

activate(edges) %>%

reroute(from = to, to = from)

gr3 <- gr1 %>%

mutate(graph = 'using subset') %>%

activate(edges) %>%

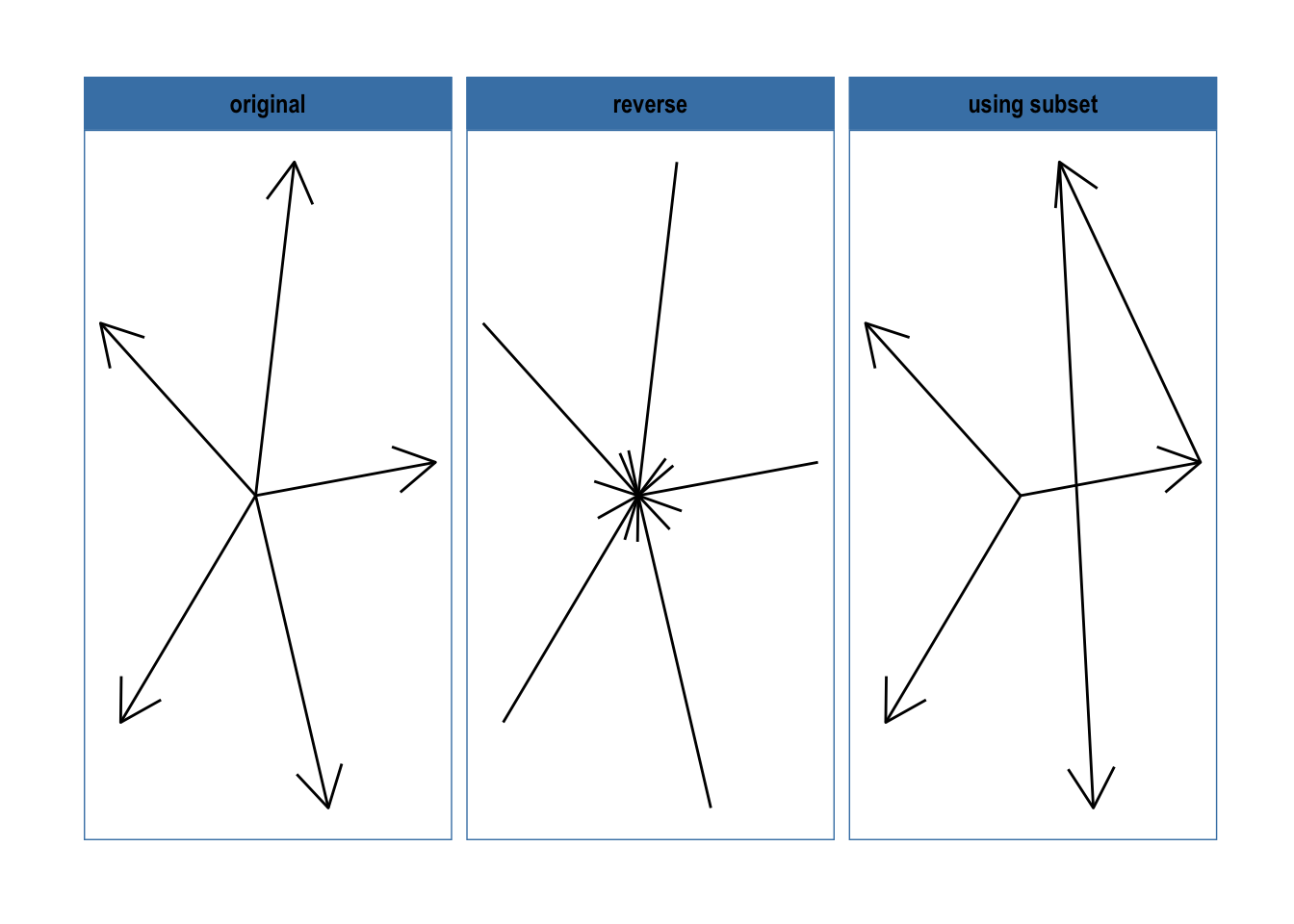
reroute(from = to + 1, subset = to < 4)

ggraph(bind\_graphs(gr1, gr2, gr3), layout = 'nicely') +

geom\_edge\_link(arrow = arrow()) +

facet\_nodes(~graph) +

theme\_graph(foreground = 'steelblue')



As can be seen, reroute works pretty much as a **to** and **from** specific **mutate()**, with the added benefit of incorporating a subset operator if only a few edges should be changed.

Making the most of graphs

While being able to use the **dplyr** verbs on relational data is nice and all, one of the reasons we are dealing with graph data in the first place is because we need some graph-based algorithms for solving our problem at hand. If we need to break out of the tidy workflow every time this was needed we wouldn’t have gained much. Because of this **tidygraph** has wrapped more or less all of **igraph**s algorithms in different ways, ensuring a consistent syntax as well as output that fits into the tidy workflow. In the following we’re going to take a look at these.

Central to all of these functions is that they know about which graph is being computed on (in the same way that **n()** knows about which tibble is currently in scope). Furthermore they always return results matching the node or edge position so they can be used directly in **mutate()** calls.

Node and edge types

On the top of our list of things we might be interested to know about is whether nodes or edges are of specific types such as *leaf*, *sink*, *loop*, etc. All of these functions return a logical vector indicating if the node or edge belong to the specified group. To easily find functions that queries types, all functions are prefixed with **node\_is\_\***/**edge\_is\_\***.

create\_tree(20, 3) %>%

mutate(leaf = node\_is\_leaf(), root = node\_is\_root()) %>%

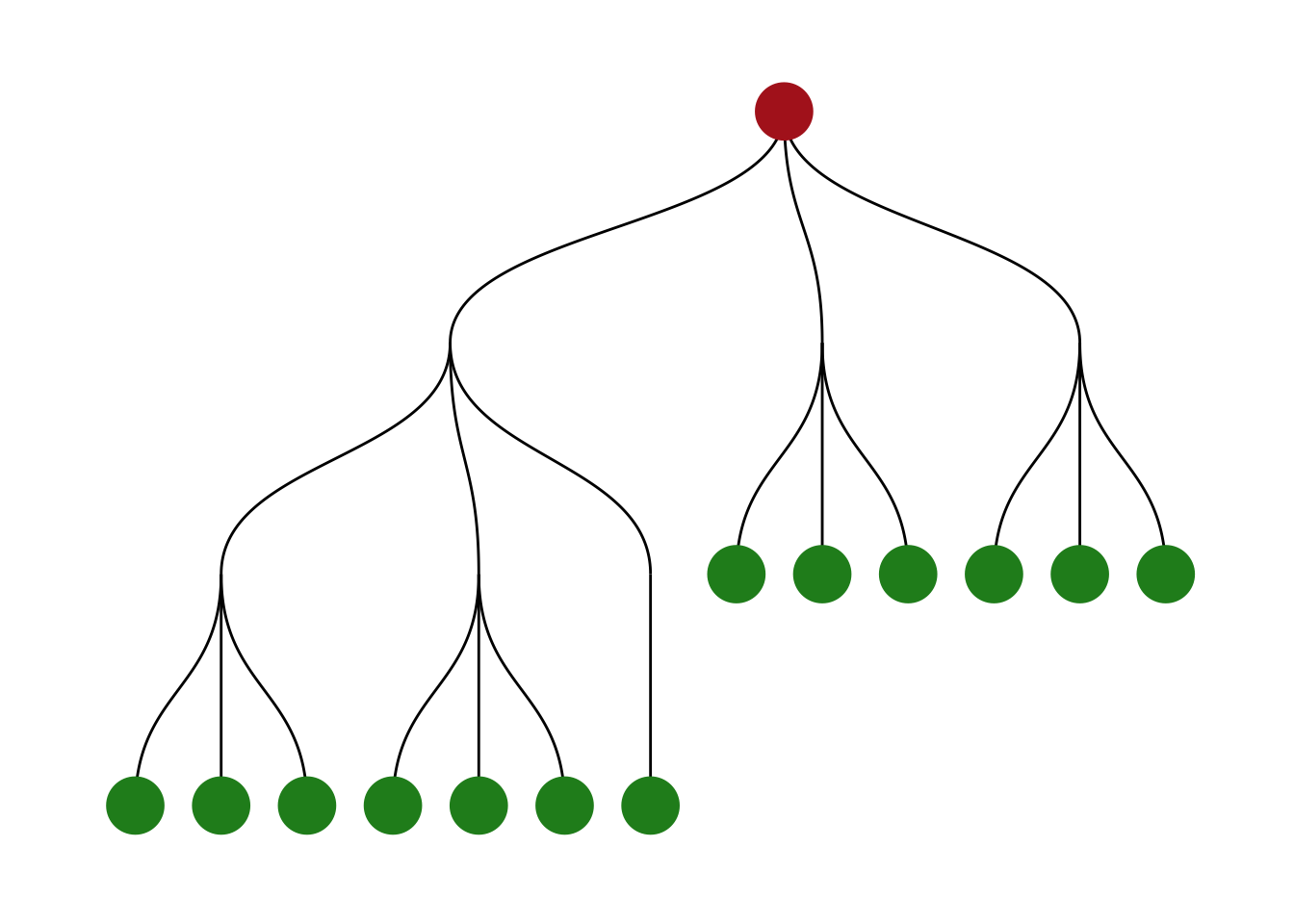
ggraph(layout = 'tree') +

geom\_edge\_diagonal() +

geom\_node\_point(aes(filter = leaf), colour = 'forestgreen', size = 10) +

geom\_node\_point(aes(filter = root), colour = 'firebrick', size = 10) +

theme\_graph()



Another example could be to remove loop edges using **filter(!edge\_is\_loop())**.

Centrality

One of the simplest concepts when computing graph based values is that of *centrality*, i.e. how central is a node or edge in the graph. As this definition is inherently vague, a lot of different centrality scores exists that all treat the concept of *central* a bit different. One of the famous ones is the pagerank algorithm that was powering Google Search in the beginning. **tidygraph** currently has 11 different centrality measures and all of these are prefixed with **centrality\_\*** for easy discoverability. All of them returns a numeric vector matching the nodes (or edges in the case of **centrality\_edge\_betweenness()**).

play\_smallworld(1, 100, 3, 0.05) %>%

mutate(centrality = centrality\_authority()) %>%

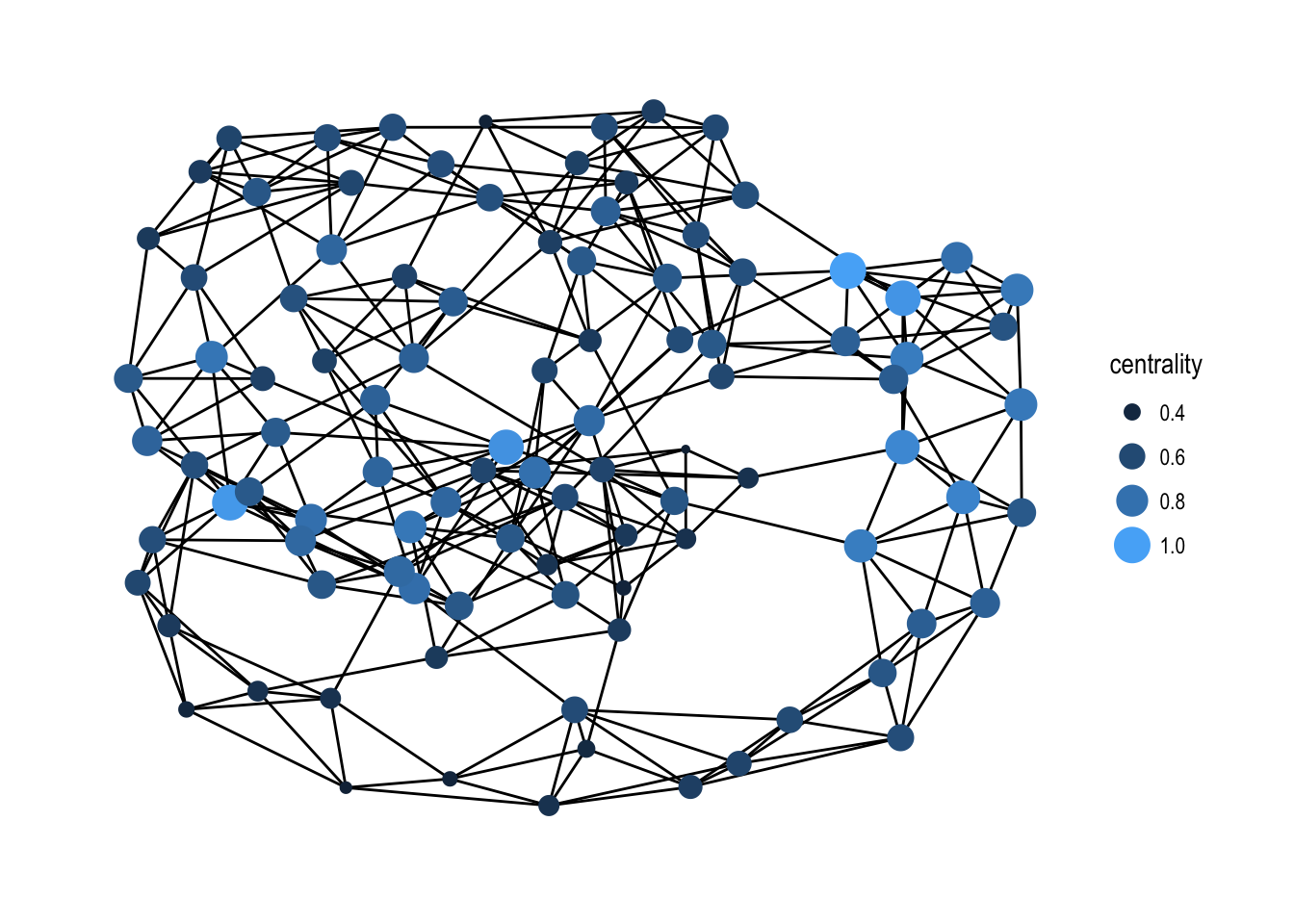
ggraph(layout = 'kk') +

geom\_edge\_link() +

geom\_node\_point(aes(size = centrality, colour = centrality)) +

scale\_color\_continuous(guide = 'legend') +

theme\_graph()



It is quite difficult to a priori decide which centrality measure makes most sense for a problem at hand so having easy access to a large range of them in a common syntax is a boon.

Clustering

Another common operation is to group nodes based on the graph topology, sometimes referred to as *community detection* based on its commonality in social network analysis. All clustering algorithms from **igraph** is available in **tidygraph** using the **group\_\*** prefix. All of these functions return an integer vector with nodes (or edges) sharing the same integer being grouped together.

play\_islands(5, 10, 0.8, 3) %>%

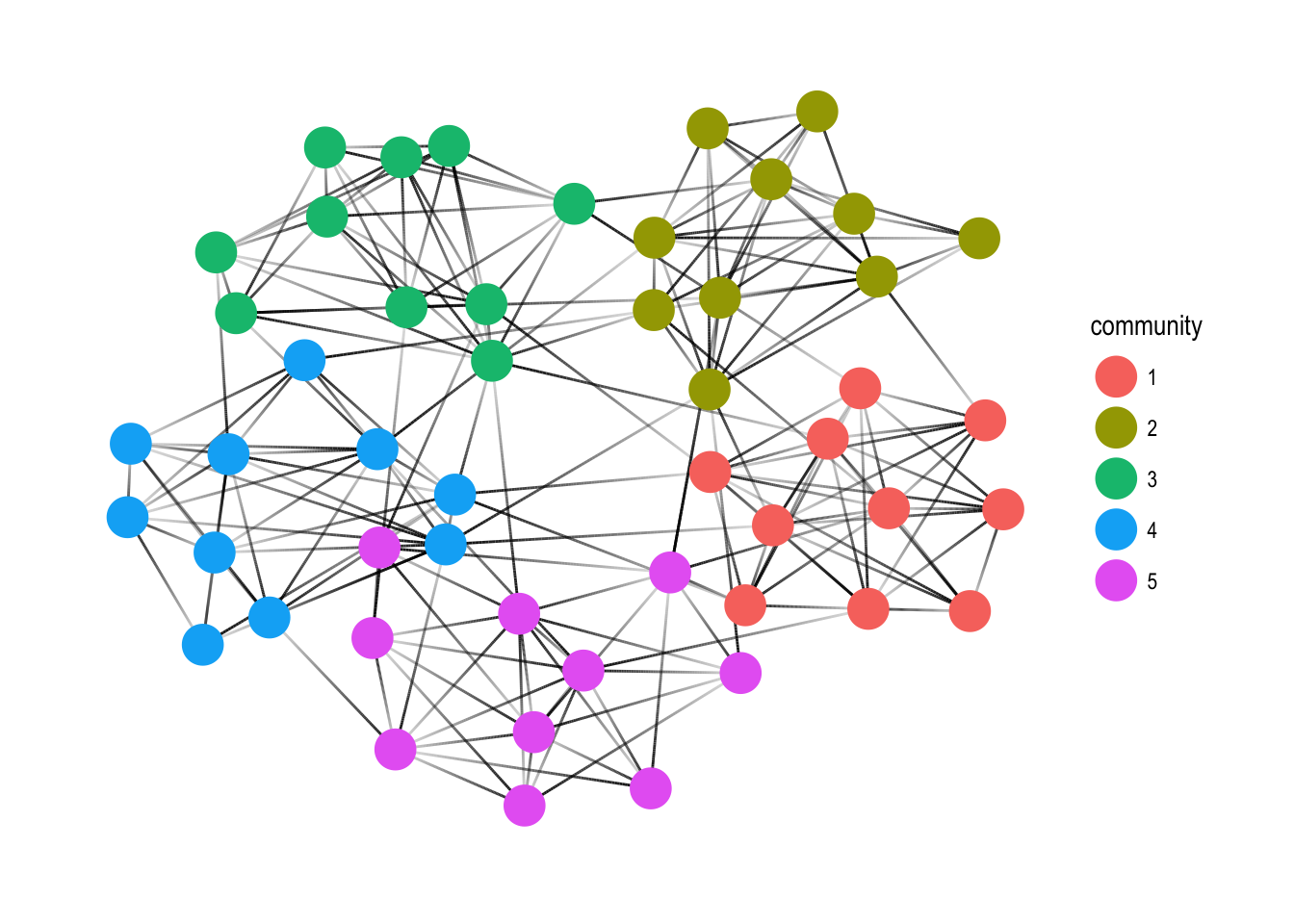
mutate(community = as.factor(group\_infomap())) %>%

ggraph(layout = 'kk') +

geom\_edge\_link(aes(alpha = ..index..), show.legend = FALSE) +

geom\_node\_point(aes(colour = community), size = 7) +

theme\_graph()



Node pairs

Some statistics are a measure between two nodes, such as distance or similarity between nodes. In a tidy context one of the ends must always be the node defined by the row, while the other can be any other node. All of the node pair functions are prefixed with **node\_\*** and ends with **\_from**/**\_to** if the measure is not symmetric and **\_with** if it is; e.g. there’s both a **node\_max\_flow\_to()** and **node\_max\_flow\_from()** function while only a single **node\_cocitation\_with()** function. The other part of the node pair can be specified as an integer vector that will get recycled if needed, or a logical vector which will get recycled and converted to indexes with **which()**. This means that output from node type functions can be used directly in the calls, e.g.

play\_geometry(50, 0.25) %>%

mutate(dist\_to\_center = node\_distance\_to(node\_is\_center())) %>%

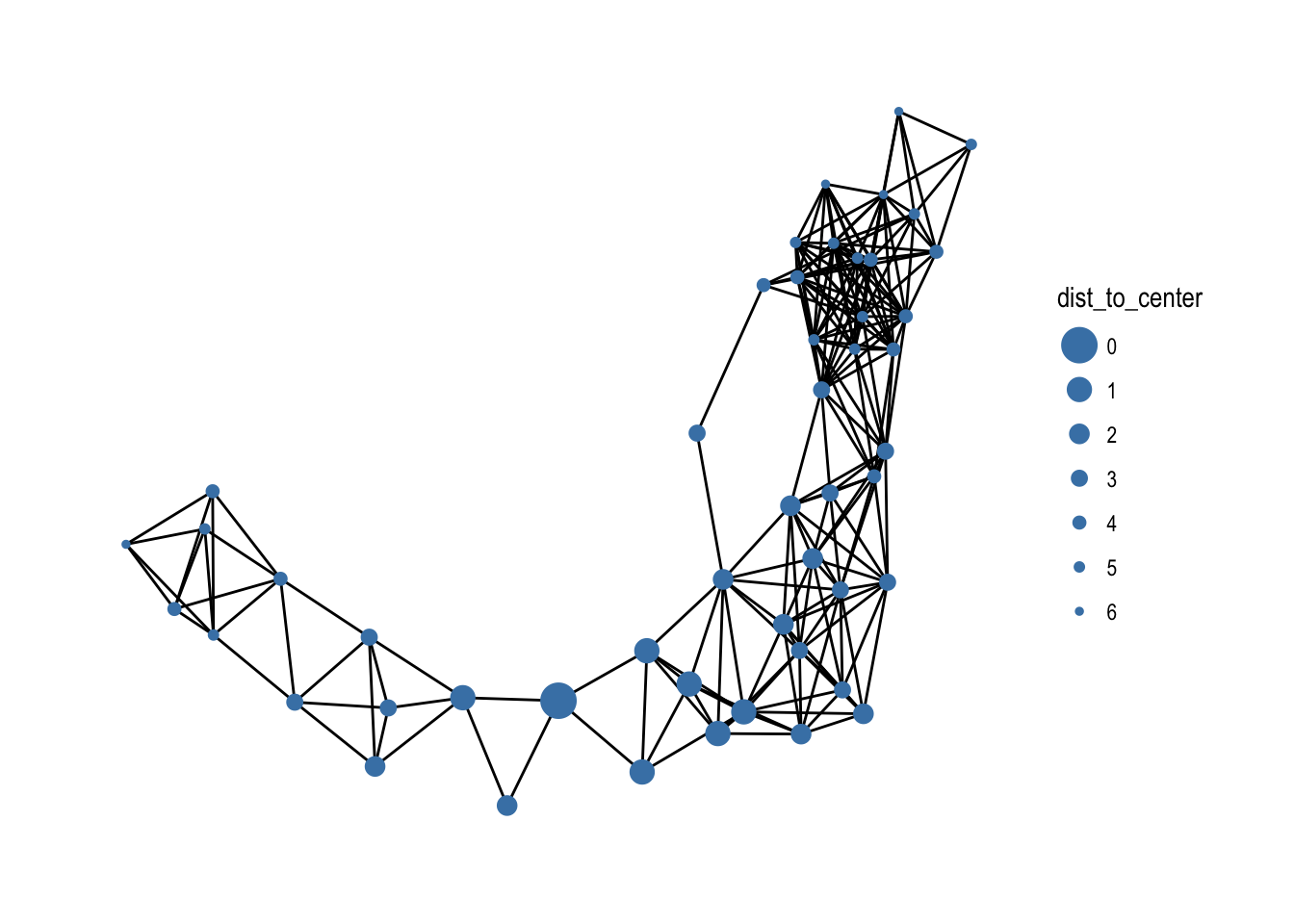
ggraph(layout = 'kk') +

geom\_edge\_link() +

geom\_node\_point(aes(size = dist\_to\_center), colour = 'steelblue') +

scale\_size\_continuous(range = c(6, 1)) +

theme\_graph()



Searches

An integral type of operation on graphs is to perform a search, that is, start from one node and then traverse the edges until all nodes has been visited. The most common approaches are either *breath first search* where all neighbors of a node is visited before moving on to the next node, or *depth first serch* where you move along to the next node immediately and only backtracks and visit other neighbors when you’ve hit a dead end. Different statistics from these searches are available in **tidygraph** through the **bfs\_\*()** and **dfs\_\*()** family of functions e.g. the distance to the start node along the search can be obtained with **bfs\_dist()**/**dfs\_dist()**. The root node can be specified in the same way as with node pairs. Sorting based on a search from the node with highest centrality can thus be done with:

play\_geometry(50, 0.25) %>%

mutate(order = bfs\_rank(which.max(centrality\_alpha())))

*#> # A tbl\_graph: 50 nodes and 227 edges*

*#> #*

*#> # An undirected simple graph with 1 component*

*#> #*

*#> # Node Data: 50 x 3 (active)*

*#> x y order*

*#> <dbl> <dbl> <int>*

*#> 1 0.01637861 0.11280499 49*

*#> 2 0.01721270 0.19181732 50*

*#> 3 0.02015524 0.74802650 46*

*#> 4 0.04361792 0.09377818 48*

*#> 5 0.04622344 0.95686281 47*

*#> 6 0.07677761 0.46011354 41*

*#> # ... with 44 more rows*

*#> #*

*#> # Edge Data: 227 x 2*

*#> from to*

*#> <int> <int>*

*#> 1 1 2*

*#> 2 1 4*

*#> 3 2 4*

*#> # ... with 224 more rows*

Local measures

Often we find ourselves interested in the local neighborhood of a node for various reasons. We might want to know the average degree around a node or the number of triangles each node participate in. The **local\_\*()** family of functions provide access to a range of node measures that are dependent on the local neighborhood of each node.

*# Weight the node degree by the average degree of its neighboors*

play\_smallworld(1, 100, 3, 0.05) %>%

mutate(weighted\_degree = centrality\_degree() / local\_ave\_degree())

*#> # A tbl\_graph: 100 nodes and 300 edges*

*#> #*

*#> # An undirected simple graph with 1 component*

*#> #*

*#> # Node Data: 100 x 1 (active)*

*#> weighted\_degree*

*#> <dbl>*

*#> 1 0.8064516*

*#> 2 1.0000000*

*#> 3 0.8064516*

*#> 4 1.2075472*

*#> 5 0.9729730*

*#> 6 0.6153846*

*#> # ... with 94 more rows*

*#> #*

*#> # Edge Data: 300 x 2*

*#> from to*

*#> <int> <int>*

*#> 1 1 2*

*#> 2 2 3*

*#> 3 4 97*

*#> # ... with 297 more rows*

All the rest

While an ontology of graph operations has been attempted in the different functions above, there are some that falls outside. These have been lumped together under the **node\_\*()** and **edge\_\*()** umbrellas and include things such as topological ordering and Burt’s constraint among others. All of these functions ensures a mutate-compatible output.

Graph measures

Along with computations on the individual nodes and edges it is sometimes necessary to get summary statistics on the graph itself. These can be simple measures such as the number of nodes and edges as well as more involved measures such as assortativity (the propensity of similar nodes to be connected). All of these measures can be calculated through the **graph\_\*()** function family and they will all return a scalar.

*# Normalise the node pair adhesion with the minimal adhesion of the graph*

play\_islands(5, 10, 0.7, 3) %>%

mutate(norm\_adhesion = node\_adhesion\_to(c(50, 1:49)) / graph\_adhesion())

*#> # A tbl\_graph: 50 nodes and 181 edges*

*#> #*

*#> # An undirected simple graph with 1 component*

*#> #*

*#> # Node Data: 50 x 1 (active)*

*#> norm\_adhesion*

*#> <dbl>*

*#> 1 1.75*

*#> 2 2.00*

*#> 3 2.25*

*#> 4 1.75*

*#> 5 1.50*

*#> 6 1.50*

*#> # ... with 44 more rows*

*#> #*

*#> # Edge Data: 181 x 2*

*#> from to*

*#> <int> <int>*

*#> 1 1 2*

*#> 2 1 3*

*#> 3 2 3*

*#> # ... with 178 more rows*

Mapping over nodes

Just to spice it all up a bit **tidygraph** pulls **purrr** into the mix and provides some additional graph-centric takes on the familiar **map\*()**. More specifically **tidygraph** provides functionality to apply a function over nodes as a breath or depth first search is carried out, while getting access to the result of the computations coming before, as well as mapping over the local neighborhood of each node. All of these function returns a list in their bare bone form, but as with **purrr** versions exists that ensures the output is of a certain type (e.g. **map\_bfs\_dbl()**).

Mapping over searches

The search maps comes in two flavors. Either the nodes are mapped in the order of the search, or they are mapped in the reverse order. In the first version, each call will have access to the statistics and map results of all the nodes that lies between itself and the root. In the second version each call will have access to the results and statistics of all its offspring. Furthermore the mapping function is passed the graph itself as well as all the search statistics of the node currently being mapped over. An example would be to propagate the *species* value in our iris clustering upwards as long as theirs agreement between the children. For this to work, we will need the reverse version of a breath first search to make sure that all children have been evaluated prior to mapping over a node:

iris\_tree <- iris\_tree %>%

activate(nodes) %>%

mutate(Species = map\_bfs\_back\_chr(node\_is\_root(), .f = function(node, path, ...) {

nodes <- .N()

if (nodes$leaf[node]) return(nodes$Species[node])

if (anyNA(unlist(path$result))) return(NA\_character\_)

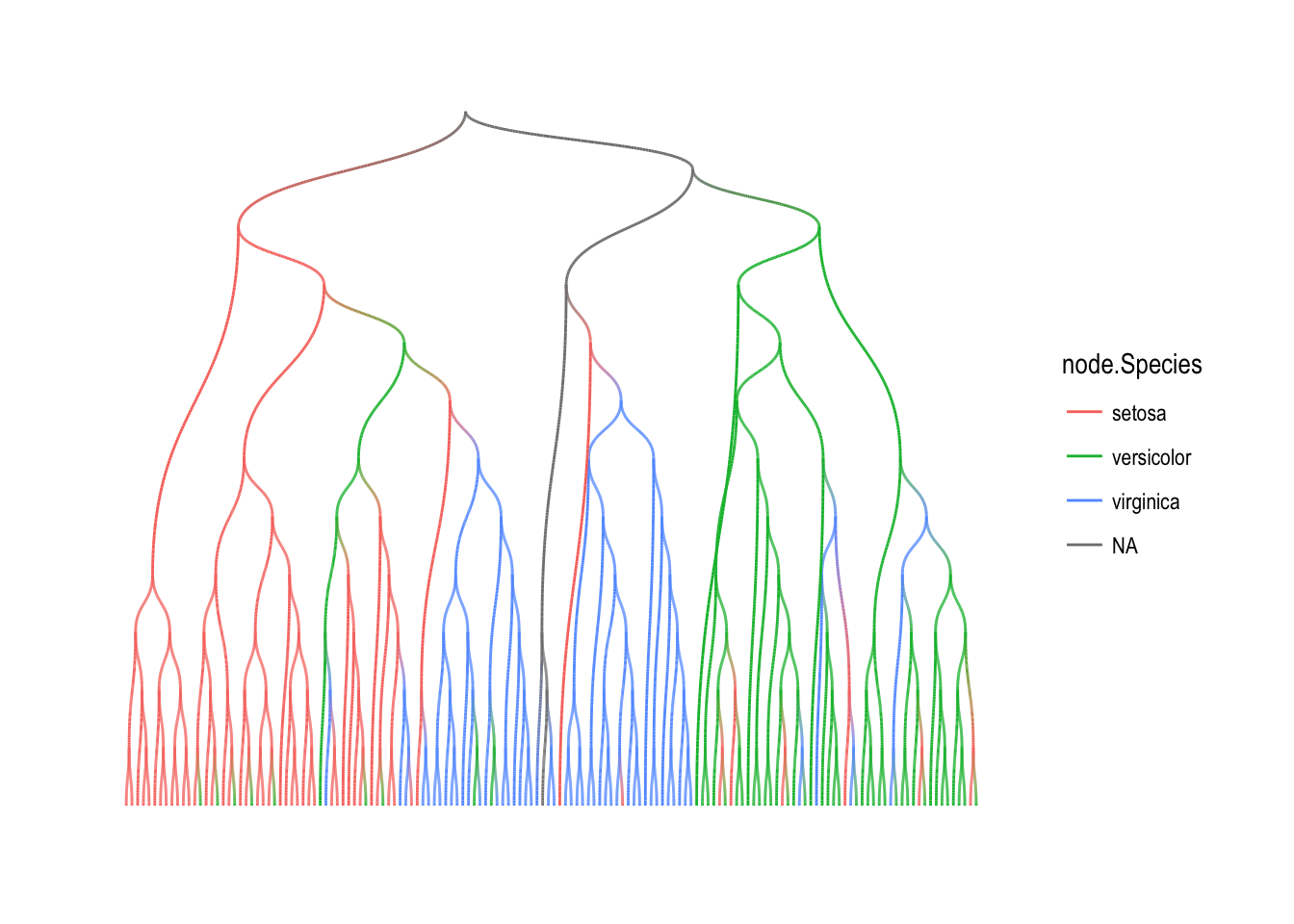
path$result[[1]]

}))

ggraph(iris\_tree, layout = 'dendrogram') +

geom\_edge\_diagonal2(aes(colour = node.Species)) +

theme\_graph()



Mapping over neighborhoods

The neighborhood map is exposed through **map\_local()** as well as its type safe versions. The mapping function has a much simpler format as it simply gets passed a subgraph representing the local neighborhood as well as the index of the node in the original graph being mapped over. E.g. to get the number of edges in the local neighborhood around each node, one would simply do:

play\_smallworld(1, 100, 3, 0.05) %>%

mutate(neighborhood\_edges = map\_local\_dbl(.f = function(neighborhood, ...) {

igraph::gsize(neighborhood)

}))

*#> # A tbl\_graph: 100 nodes and 300 edges*

*#> #*

*#> # An undirected simple graph with 1 component*

*#> #*

*#> # Node Data: 100 x 1 (active)*

*#> neighborhood\_edges*

*#> <dbl>*

*#> 1 14*

*#> 2 14*

*#> 3 13*

*#> 4 15*

*#> 5 12*

*#> 6 14*

*#> # ... with 94 more rows*

*#> #*

*#> # Edge Data: 300 x 2*

*#> from to*

*#> <int> <int>*

*#> 1 1 2*

*#> 2 2 3*

*#> 3 3 4*

*#> # ... with 297 more rows*

One last thing…

While the functions discussed above makes it easy to make slight changes to your network topology it is less straightforward to make radical changes. Even more so if the radical changes are only needed temporarily for the sake of a few computations. This is where the new **morph()** verb comes in handy (along with the accompanying **unmorph()** and **crystallise()** verbs). In essence, **morph()** lets you set up a temporary alternative version of your graph, make computations on it using the standard **dplyr** verbs, and then merge the changes back in using **unmorph()**. The types of alternative representations are varied and can be extended by the user. Nodes can be converted to edges and the other way around, both nodes and edges can be combined, and the alternate representation does not need to cover the full original graph. Instead of trying to describe it in words, let’s see how it plays out in use:

islands <- play\_islands(5, 10, 0.8, 3) %>%

mutate(group = group\_infomap())

*# Get the distance to the central node in each group*

islands <- islands %>%

morph(to\_split, group) %>%

mutate(dist\_to\_center = node\_distance\_to(node\_is\_center())) %>%

unmorph()

islands

*#> # A tbl\_graph: 50 nodes and 211 edges*

*#> #*

*#> # An undirected simple graph with 1 component*

*#> #*

*#> # Node Data: 50 x 2 (active)*

*#> group dist\_to\_center*

*#> <int> <dbl>*

*#> 1 5 0*

*#> 2 5 0*

*#> 3 5 0*

*#> 4 5 0*

*#> 5 5 0*

*#> 6 5 0*

*#> # ... with 44 more rows*

*#> #*

*#> # Edge Data: 211 x 2*

*#> from to*

*#> <int> <int>*

*#> 1 1 3*

*#> 2 2 3*

*#> 3 1 4*

*#> # ... with 208 more rows*

*# Get the number of edges exiting each group*

islands <- islands %>%

morph(to\_contracted, group, simplify = FALSE) %>%

activate(edges) %>%

filter(!edge\_is\_loop()) %>%

activate(nodes) %>%

mutate(exiting\_group = centrality\_degree(mode = 'out')) %>%

unmorph()

islands

*#> # A tbl\_graph: 50 nodes and 211 edges*

*#> #*

*#> # An undirected simple graph with 1 component*

*#> #*

*#> # Node Data: 50 x 3 (active)*

*#> group dist\_to\_center exiting\_group*

*#> <int> <dbl> <dbl>*

*#> 1 5 0 12*

*#> 2 5 0 12*

*#> 3 5 0 12*

*#> 4 5 0 12*

*#> 5 5 0 12*

*#> 6 5 0 12*

*#> # ... with 44 more rows*

*#> #*

*#> # Edge Data: 211 x 2*

*#> from to*

*#> <int> <int>*

*#> 1 1 3*

*#> 2 2 3*

*#> 3 1 4*

*#> # ... with 208 more rows*

*# Calculate an edge centrality score by converting to the linegraph*

islands <- islands %>%

morph(to\_linegraph) %>%

activate(nodes) %>%

mutate(edge\_centrality = centrality\_pagerank()) %>%

unmorph()

islands

*#> # A tbl\_graph: 50 nodes and 211 edges*

*#> #*

*#> # An undirected simple graph with 1 component*

*#> #*

*#> # Node Data: 50 x 3 (active)*

*#> group dist\_to\_center exiting\_group*

*#> <int> <dbl> <dbl>*

*#> 1 5 0 12*

*#> 2 5 0 12*

*#> 3 5 0 12*

*#> 4 5 0 12*

*#> 5 5 0 12*

*#> 6 5 0 12*

*#> # ... with 44 more rows*

*#> #*

*#> # Edge Data: 211 x 3*

*#> from to edge\_centrality*

*#> <int> <int> <dbl>*

*#> 1 1 3 0.005117026*

*#> 2 2 3 0.005079217*

*#> 3 1 4 0.004657749*

*#> # ... with 208 more rows*

As can be seen, the morph syntax both handles multiple graphs, collapsed nodes and changing edges to nodes, without any change in the mental model of the operations. All morphing functions are prefixed with **to\_\*** for easy discovery and includes minimum spanning trees, complement graph, dominator tree etc. In the case where you are interested to continue working with the morphed representation as a proper **tbl\_graph** you can use the **cystallise()** verbs that removes any link to the original graph and returns a tibble with a row per graph in the morphed representation (as a morph can result in multiple graphs):

islands %>%

morph(to\_split, group) %>%

crystallise()

*#> # A tibble: 5 x 2*

*#> name graph*

*#> <chr> <list>*

*#> 1 group: 1 <S3: tbl\_graph>*

*#> 2 group: 2 <S3: tbl\_graph>*

*#> 3 group: 3 <S3: tbl\_graph>*

*#> 4 group: 4 <S3: tbl\_graph>*

*#> 5 group: 5 <S3: tbl\_graph>*

Wrapping it all up

I hope I have given you a small glimpse of what **tidygraph** is all about. If working with network data in the past has felt intimidating and strange **tidygraph** might feel more at home, but even if you’re a seasoned pro within network analysis the package should provide a powerful but streamlined interface to many operations.

A central aspect of tidygraph is that you can directly manipulate node and edge data from this tbl\_graph object by **activating** nodes or edges. When we first create a tbl\_graph object, the nodes will be activated. We can then directly calculate node or edge metrics, like centrality, using tidyverse functions.

as\_tbl\_graph(cooc\_all\_f, directed = FALSE)

## # A tbl\_graph: 100 nodes and 798 edges

## #

## # An undirected simple graph with 1 component

## #

## # Node Data: 100 x 1 (active)

## name

##

## 1 Aemon-Targaryen-(Maester-Aemon)

## 2 Aeron-Greyjoy

## 3 Aerys-II-Targaryen

## 4 Alliser-Thorne

## 5 Arianne-Martell

## 6 Arya-Stark

## # ... with 94 more rows

## #

## # Edge Data: 798 x 5

## from to Type id weight

##

## 1 1 4 Undirected 43 7

## 2 1 13 Undirected 44 4

## 3 1 28 Undirected 52 3

## # ... with 795 more rows

We can change that with the activate() function. We can now, for example, remove multiple edges. When you are using RStudio, start typing ?edge\_is\_ and wait for the autocomplete function to show you what else is possible (or go to the tidygraph manual).

as\_tbl\_graph(cooc\_all\_f, directed = FALSE) %>%

activate(edges) %>%

filter(!edge\_is\_multiple())

## # A tbl\_graph: 100 nodes and 798 edges

## #

## # An undirected simple graph with 1 component

## #

## # Edge Data: 798 x 5 (active)

## from to Type id weight

##

## 1 1 4 Undirected 43 7

## 2 1 13 Undirected 44 4

## 3 1 28 Undirected 52 3

## 4 1 32 Undirected 53 20

## 5 1 34 Undirected 54 5

## 6 1 41 Undirected 56 5

## # ... with 792 more rows

## #

## # Node Data: 100 x 1

## name

##

## 1 Aemon-Targaryen-(Maester-Aemon)

## 2 Aeron-Greyjoy

## 3 Aerys-II-Targaryen

## # ... with 97 more rows

**Node ranking**

Often, especially when visualising networks with certain layouts, the order in which the nodes appear will have a huge influence on the insight you can get out (e.g. matrix plots and arc diagrams). The node\_rank\_\*() family of algorithms have been introduced to provide different ways of sorting nodes so that closely related nodes are positionally close. As there is often not a single correct answer to this endeavor, there’s a lot of different algorithms that may provide different insights into your network. Many of them are based on the seriation package, and the vignette provided therein serves as a nice introduction to the different algorithms.

There are many options for node ranking (go to ?node\_rank for a full list); let’s try out **Minimize hamiltonian path length using a travelling salesperson solver**.

as\_tbl\_graph(cooc\_all\_f, directed = FALSE) %>%

activate(nodes) %>%

mutate(n\_rank\_trv = node\_rank\_traveller()) %>%

arrange(n\_rank\_trv)

## # A tbl\_graph: 100 nodes and 798 edges

## #

## # An undirected simple graph with 1 component

## #

## # Node Data: 100 x 2 (active)

## name n\_rank\_trv

##

## 1 Janos-Slynt 1

## 2 Aemon-Targaryen-(Maester-Aemon) 2

## 3 Jeor-Mormont 3

## 4 Samwell-Tarly 4

## 5 Qhorin-Halfhand 5

## 6 Ygritte 6

## # ... with 94 more rows

## #

## # Edge Data: 798 x 5

## from to Type id weight

##

## 1 2 75 Undirected 43 7

## 2 2 76 Undirected 44 4

## 3 2 73 Undirected 52 3

## # ... with 795 more rows

**Centrality**

Centrality describes the number of edges that are in- or outgoing to/from nodes. High centrality networks have few nodes with many connections, low centrality networks have many nodes with similar numbers of edges. The centrality of a node measures the importance of it in the network.

This version adds 19(!) new ways to define the notion of centrality along with a manual version where you can mix and match different distance measures and summation strategies opening up the world to even more centrality scores. All of this wealth of centrality comes from the netrankr package that provides a framework for defining and calculating centrality scores. If you use centrality measures somewhere in your analysis I cannot recommend the vignettes provided by netrankr enough as they provide a fundamental intuition about the nature of such measures and how they can/should be used.

Again, type ?centrality for an overview about all possible centrality measures you can use. Let’s try out centrality\_degree().

as\_tbl\_graph(cooc\_all\_f, directed = FALSE) %>%

activate(nodes) %>%

mutate(neighbors = centrality\_degree()) %>%

arrange(-neighbors)

## # A tbl\_graph: 100 nodes and 798 edges

## #

## # An undirected simple graph with 1 component

## #

## # Node Data: 100 x 2 (active)

## name neighbors

##

## 1 Tyrion-Lannister 54.

## 2 Cersei-Lannister 49.

## 3 Joffrey-Baratheon 49.

## 4 Robert-Baratheon 47.

## 5 Jaime-Lannister 45.

## 6 Sansa-Stark 44.

## # ... with 94 more rows

## #

## # Edge Data: 798 x 5

## from to Type id weight

##

## 1 41 42 Undirected 43 7

## 2 41 60 Undirected 44 4

## 3 41 63 Undirected 52 3

## # ... with 795 more rows

**Grouping and clustering**

Another common operation is to group nodes based on the graph topology, sometimes referred to as community detection based on its commonality in social network analysis. All clustering algorithms from igraph is available in tidygraph using the group\_\* prefix. All of these functions return an integer vector with nodes (or edges) sharing the same integer being grouped together

We can use ?group\_graph for an overview about all possible ways to cluster and group nodes. Here I am using group\_infomap(): **Group nodes by minimizing description length using**.

as\_tbl\_graph(cooc\_all\_f, directed = FALSE) %>%

activate(nodes) %>%

mutate(group = group\_infomap()) %>%

arrange(-group)

## # A tbl\_graph: 100 nodes and 798 edges

## #

## # An undirected simple graph with 1 component

## #

## # Node Data: 100 x 2 (active)

## name group

##

## 1 Arianne-Martell 7

## 2 Doran-Martell 7

## 3 Davos-Seaworth 6

## 4 Melisandre 6

## 5 Selyse-Florent 6

## 6 Stannis-Baratheon 6

## # ... with 94 more rows

## #

## # Edge Data: 798 x 5

## from to Type id weight

##

## 1 32 33 Undirected 43 7

## 2 32 34 Undirected 44 4

## 3 32 36 Undirected 52 3

## # ... with 795 more rows

**Querying node types**

We can also query different node types (?node\_types gives us a list of options):

These functions all lets the user query whether each node is of a certain type. All of the functions returns a logical vector indicating whether the node is of the type in question. Do note that the types are not mutually exclusive and that nodes can thus be of multiple types.

Here, I am trying out node\_is\_center() (does the node have the minimal eccentricity in the graph) and node\_is\_keyplayer() to identify the top 10 key-players in the network. You can read more about the node\_is\_keyplayer() function in the manual for the influenceR package:

The “Key Player” family of node importance algorithms (Borgatti 2006) involves the selection of a metric of node importance and a combinatorial optimization strategy to choose the set S of vertices of size k that maximize that metric. This function implements KPP-Pos, a metric intended to identify k nodes which optimize resource diffusion through the net …

Code Chunks of Graph\_Metrics.R

|  |
| --- |
| # graph\_metrics.R: R code for vertex importance metrics. |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | csv.to.igraph <- function(fname) { |
|  | x <- utils::read.csv(fname) # this may be dangerous because of users' settings. |
|  | # See: http://r-pkgs.had.co.nz/r.html |
|  | el <- as.matrix(x[c(1,2)]) |
|  | if(!is.character(el)) |
|  | el <- apply(el, 2, as.character) |
|  |  |
|  | igraph::graph.edgelist(el, directed=F) |
|  | } |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | betweenness <- function(g, snap=T) { |
|  | if (!igraph::is\_igraph(g)) { |
|  | stop("Not a graph object") |
|  | } |
|  | # 1/2 the values of our betweenness code, which is because this is UNDIRECTED for real |
|  | if (!snap) |
|  | return(igraph::betweenness(g)) |
|  |  |
|  | el <- igraph::get.edgelist(g, names=F) |
|  | el\_i <- as.integer(t(el)) |
|  | n <- as.integer(max(el)) |
|  | m <- as.integer(length(el)/2) # TODO: for directed too? |
|  |  |
|  | vals <- .Call("snap\_betweenness\_R", el\_i, n, m, PACKAGE="influenceR") |
|  | vals[vals<2^-128] <- 0 |
|  | vals[is.nan(vals)] <- 0 |
|  | names(vals) <- igraph::V(g)$name |
|  | vals |
|  | } |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | keyplayer <- function(g, k, prob = 0.0, tol = 0.0001, maxsec = 120, roundsec = 30) { |
|  | if (!igraph::is\_igraph(g)) { |
|  | stop("Not a graph object") |
|  | } |
|  |  |
|  | if (roundsec > maxsec) |
|  | roundsec <- maxsec |
|  |  |
|  | el <- igraph::get.edgelist(g, names=F) |
|  | el\_i <- as.integer(t(el)) |
|  | n <- as.integer(max(el)) |
|  | m <- as.integer(length(el)/2) |
|  |  |
|  | converges <- vector("integer", 1) # just allocate space for an integer |
|  |  |
|  | s <- .Call("snap\_keyplayer\_R", el\_i, n, m, as.integer(k), prob, tol, as.integer(maxsec), as.integer(roundsec), converges, PACKAGE="influenceR") |
|  |  |
|  | if (converges == 1) |
|  | warning("Maximum computation time (arg 'maxsec') exceeded!") |
|  |  |
|  | igraph::V(g)[which(s>0)] |
|  | } |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | bridging <- function(g) { |
|  | if (!igraph::is\_igraph(g)) { |
|  | stop("Not a graph object") |
|  | } |
|  | el <- igraph::get.edgelist(g, names = F) |
|  | el\_i <- as.integer(t(el)) |
|  | n <- as.integer(max(el\_i)) |
|  | m <- as.integer(length(el\_i) / 2) |
|  |  |
|  | x <- .Call("snap\_bridging\_R", el\_i, n, m, as.integer(FALSE), as.integer(0), PACKAGE = "influenceR") |
|  | names(x) <- igraph::V(g)$name |
|  | x |
|  | } |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | ens <- function(g) { |
|  | if (!igraph::is\_igraph(g)) { |
|  | stop("Not a graph object") |
|  | } |
|  | A <- igraph::get.adjacency(g) # This will be sparse, which is great. |
|  | S <- Matrix::crossprod(A) # S[i,j] = # of shared neighbors between i,j |
|  | Q <- A \* S # Q[i,j] = # of shared neighbors if i and j are neighbors, 0 else |
|  | qsum <- Matrix::rowSums(Q) |
|  | deg <- Matrix::rowSums(A) |
|  | ens <- deg - (qsum / deg) |
|  | ens[is.nan(ens)] <- 0 # If a vertex has no neighbors, make its ENS 0 |
|  | names(ens) <- igraph::V(g)$name |
|  | ens |
|  | } |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | constraint <- function(g, v=igraph::V(g)) { |
|  | if (!igraph::is\_igraph(g)) { |
|  | stop("Not a graph object") |
|  | } |
|  |  |
|  | process\_sparse <- function(A, Ai, deg) { |
|  | M <- methods::as(A, 'TsparseMatrix') |
|  | x <- .Call("process\_sparse\_R", M@i, M@j, M@x, Ai, deg, Matrix::nnzero(M), PACKAGE = "influenceR") |
|  | M@x <- x |
|  | M |
|  | } |
|  |  |
|  | A <- igraph::get.adjacency(g, sparse=T) |
|  | n <- dim(A)[1] |
|  | deg <- Matrix::rowSums(A) |
|  |  |
|  | constraint\_i <- function(i) { |
|  | # process sparse does this: jq <- drop0(t(A\*A[,i]) \* A[,i]); jqd <- drop0(jq \* deg) |
|  | jqd <- process\_sparse(A, A[i, ], deg) |
|  |  |
|  | jqd <- Matrix::drop0(jqd) |
|  | jqd@x <- (1 / jqd@x) \* (1 / deg[i]) |
|  |  |
|  | Sj <- Matrix::colSums(jqd) |
|  |  |
|  | idx <- as.numeric(igraph::neighbors(g, i)) |
|  | Sj[idx] <- Sj[idx] + (1 / deg[i]) |
|  |  |
|  | Sj2 <- Sj \* Sj |
|  | sum(Sj2) |
|  | } |
|  |  |
|  | vals <- sapply(v, constraint\_i) |
|  | names(vals) <- v$name |
|  | vals |
|  | } |
|  |  |

as\_tbl\_graph(cooc\_all\_f, directed = FALSE) %>%

activate(nodes) %>%

mutate(center = node\_is\_center(),

keyplayer = node\_is\_keyplayer(k = 10))

## # A tbl\_graph: 100 nodes and 798 edges

## #

## # An undirected simple graph with 1 component

## #

## # Node Data: 100 x 3 (active)

## name center keyplayer

##

## 1 Aemon-Targaryen-(Maester-Aemon) FALSE FALSE

## 2 Aeron-Greyjoy FALSE FALSE

## 3 Aerys-II-Targaryen FALSE FALSE

## 4 Alliser-Thorne FALSE FALSE

## 5 Arianne-Martell FALSE FALSE

## 6 Arya-Stark FALSE FALSE

## # ... with 94 more rows

## #

## # Edge Data: 798 x 5

## from to Type id weight

##

## 1 1 4 Undirected 43 7

## 2 1 13 Undirected 44 4

## 3 1 28 Undirected 52 3

## # ... with 795 more rows

**Node pairs**

Some statistics are a measure between two nodes, such as distance or similarity between nodes. In a tidy context one of the ends must always be the node defined by the row, while the other can be any other node. All of the node pair functions are prefixed with node\_\* and ends with \_from/\_to if the measure is not symmetric and \_with if it is; e.g. there’s both a node\_max\_flow\_to() and node\_max\_flow\_from() function while only a single node\_cocitation\_with() function. The other part of the node pair can be specified as an integer vector that will get recycled if needed, or a logical vector which will get recycled and converted to indexes with which(). This means that output from node type functions can be used directly in the calls.

as\_tbl\_graph(cooc\_all\_f, directed = FALSE) %>%

activate(nodes) %>%

mutate(dist\_to\_center = node\_distance\_to(node\_is\_center()))

## # A tbl\_graph: 100 nodes and 798 edges

## #

## # An undirected simple graph with 1 component

## #

## # Node Data: 100 x 2 (active)

## name dist\_to\_center

##

## 1 Aemon-Targaryen-(Maester-Aemon) 1.

## 2 Aeron-Greyjoy 2.

## 3 Aerys-II-Targaryen 1.

## 4 Alliser-Thorne 1.

## 5 Arianne-Martell 2.

## 6 Arya-Stark 1.

## # ... with 94 more rows

## #

## # Edge Data: 798 x 5

## from to Type id weight

##

## 1 1 4 Undirected 43 7

## 2 1 13 Undirected 44 4

## 3 1 28 Undirected 52 3

## # ... with 795 more rows

**Edge betweenness**

Similarly to node metrics, we can calculate all kinds of edge metrics. Betweenness, for example, describes the shortest paths between nodes. More about what you can do with edges can be found with ?edge\_types

as\_tbl\_graph(cooc\_all\_f, directed = FALSE) %>%

activate(edges) %>%

mutate(centrality\_e = centrality\_edge\_betweenness())

## # A tbl\_graph: 100 nodes and 798 edges

## #

## # An undirected simple graph with 1 component

## #

## # Edge Data: 798 x 6 (active)

## from to Type id weight centrality\_e

##

## 1 1 4 Undirected 43 7 1.00

## 2 1 13 Undirected 44 4 30.2

## 3 1 28 Undirected 52 3 42.1

## 4 1 32 Undirected 53 20 0.

## 5 1 34 Undirected 54 5 35.2

## 6 1 41 Undirected 56 5 18.9

## # ... with 792 more rows

## #

## # Node Data: 100 x 1

## name

##

## 1 Aemon-Targaryen-(Maester-Aemon)

## 2 Aeron-Greyjoy

## 3 Aerys-II-Targaryen

## # ... with 97 more rows

**The complete code**

Now let’s combine what we’ve done above in true tidyverse fashion:

cooc\_all\_f\_graph <- as\_tbl\_graph(cooc\_all\_f, directed = FALSE) %>%

mutate(n\_rank\_trv = node\_rank\_traveller(),

neighbors = centrality\_degree(),

group = group\_infomap(),

center = node\_is\_center(),

dist\_to\_center = node\_distance\_to(node\_is\_center()),

keyplayer = node\_is\_keyplayer(k = 10)) %>%

activate(edges) %>%

filter(!edge\_is\_multiple()) %>%

mutate(centrality\_e = centrality\_edge\_betweenness())

We can also convert our active node or edge table back to a tibble:

cooc\_all\_f\_graph %>%

activate(nodes) %>% # %N>%

as.tibble()

## # A tibble: 100 x 7

## name n\_rank\_trv neighbors group center dist\_to\_center keyplayer

##

## 1 Aemon-Targa… 45 13. 2 FALSE 1. FALSE

## 2 Aeron-Greyj… 21 5. 5 FALSE 2. FALSE

## 3 Aerys-II-Ta… 11 12. 1 FALSE 1. FALSE

## 4 Alliser-Tho… 48 13. 2 FALSE 1. FALSE

## 5 Arianne-Mar… 29 4. 7 FALSE 2. FALSE

## 6 Arya-Stark 79 37. 1 FALSE 1. FALSE

## 7 Asha-Greyjoy 20 7. 5 FALSE 1. FALSE

## 8 Balon-Greyj… 18 11. 5 FALSE 2. FALSE

## 9 Barristan-S… 54 23. 3 FALSE 1. FALSE

## 10 Belwas 52 6. 3 FALSE 2. FALSE

## # ... with 90 more rows

cooc\_all\_f\_graph %>%

activate(edges) %>% # %E>%

as.tibble()

## # A tibble: 798 x 6

## from to Type id weight centrality\_e

##

## 1 1 4 Undirected 43 7 1.00

## 2 1 13 Undirected 44 4 30.2

## 3 1 28 Undirected 52 3 42.1

## 4 1 32 Undirected 53 20 0.

## 5 1 34 Undirected 54 5 35.2

## 6 1 41 Undirected 56 5 18.9

## 7 1 42 Undirected 57 25 0.

## 8 1 48 Undirected 58 110 0.

## 9 1 58 Undirected 60 5 24.5

## 10 1 71 Undirected 62 5 17.0

## # ... with 788 more rows

**Plotting with ggraph**

For plotting our graph object, we can make good use of the ggraph package:

ggraph is an extension of ggplot2 aimed at supporting relational data structures such as networks, graphs, and trees. While it builds upon the foundation of ggplot2 and its API it comes with its own self-contained set of geoms, facets, etc., as well as adding the concept of layouts to the grammar.

First, I am going to define a layout. There are lots of options for layouts, here I am using a Fruchterman-Reingold algorithm.

layout <- create\_layout(cooc\_all\_f\_graph,

layout = "fr")

The rest works like any ggplot2 function call, just that we use special geoms for our network, like geom\_edge\_density() to draw a shadow where the edge density is higher, geom\_edge\_link() to connect edges with a straight line, geom\_node\_point() to draw node points and geom\_node\_text() to draw the labels.

Here are three options of plotting the network with the metrics we just calculated:

ggraph(layout) +

geom\_edge\_density(aes(fill = weight)) +

geom\_edge\_link(aes(width = weight), alpha = 0.2) +

geom\_node\_point(aes(color = factor(group)), size = 10) +

geom\_node\_text(aes(label = name), size = 8, repel = TRUE) +

scale\_color\_brewer(palette = "Set1") +

theme\_graph() +

labs(title = "A Song of Ice and Fire character network",

subtitle = "Nodes are colored by group")

**Characters devided by books**

The second data set I am going to use is a comparison of character interactions in the five books.

**A little node on the side:** My original plan was to loop over the separate edge files for each book, concatenate them together with the information from which book they are and then plot them via faceting. This turned out to be a bad solution because I wanted to show the different key-players in each of the five books. So, instead of using one joined graph, I created separate graphs for every book and used the bind\_graphs() and facet\_nodes() functions to plot them together.

for (i in 1:5) {

cooc <- read\_csv(paste0("/Users/shiringlander/Documents/Github/Data/asoiaf/data//asoiaf-book", i, "-edges.csv")) %>%

mutate(book = paste0("book\_", i)) %>%

filter(Source %in% main\_ch\_l$name & Target %in% main\_ch\_l$name)

assign(paste0("coocs\_book\_", i), cooc)

}

The concepts are the same as above, here I want to know the key-players in each book:

cooc\_books\_1\_graph <- as\_tbl\_graph(coocs\_book\_1, directed = FALSE) %>%

mutate(book = "Book 1: A Game of Thrones",

keyplayer = node\_is\_keyplayer(k = 10))

cooc\_books\_2\_graph <- as\_tbl\_graph(coocs\_book\_2, directed = FALSE) %>%

mutate(book = "Book 2: A Clash of Kings",

keyplayer = node\_is\_keyplayer(k = 10))

cooc\_books\_3\_graph <- as\_tbl\_graph(coocs\_book\_3, directed = FALSE) %>%

mutate(book = "Book 3: A Storm of Swords",

keyplayer = node\_is\_keyplayer(k = 10))

cooc\_books\_4\_graph <- as\_tbl\_graph(coocs\_book\_4, directed = FALSE) %>%

mutate(book = "Book 4: A Feast for Crows",

keyplayer = node\_is\_keyplayer(k = 10))

cooc\_books\_5\_graph <- as\_tbl\_graph(coocs\_book\_5, directed = FALSE) %>%

mutate(book = "Book 5: A Dance with Dragons",

keyplayer = node\_is\_keyplayer(k = 10))

And let’s combine and plot the key-players:

cooc\_books\_1\_graph %>%

bind\_graphs(cooc\_books\_2\_graph) %>%

bind\_graphs(cooc\_books\_3\_graph) %>%

bind\_graphs(cooc\_books\_4\_graph) %>%

bind\_graphs(cooc\_books\_5\_graph) %>%

ggraph(layout = "fr") +

facet\_nodes( ~ book, scales = "free", ncol = 1) +

geom\_edge\_density(aes(fill = weight)) +

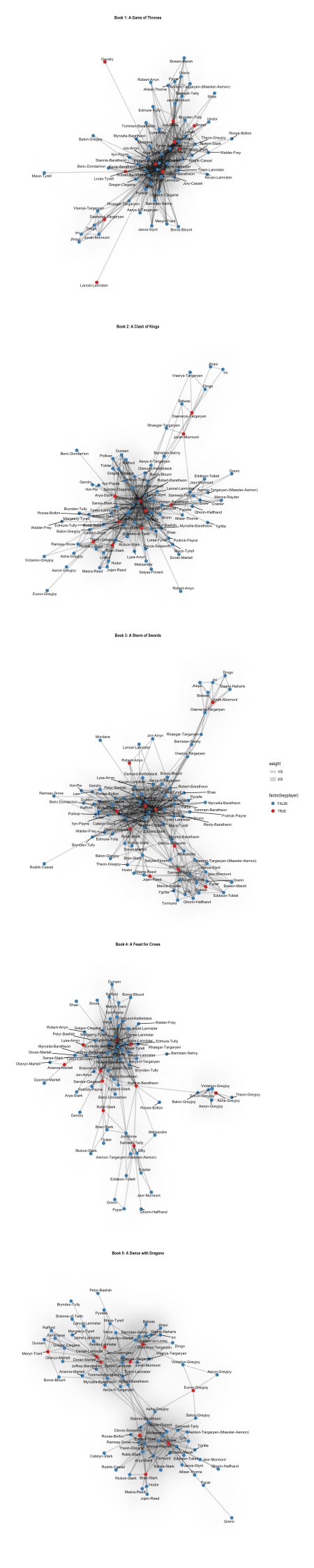
geom\_edge\_link(aes(edge\_width = weight), alpha = 0.2) +

geom\_node\_point(aes(color = factor(keyplayer)), size = 3) +

geom\_node\_text(aes(label = name), color = "black", size = 3, repel = TRUE) +

theme\_graph() +

scale\_colour\_manual(values = c(cols[2], cols[1]))



The networks and key-players of the five different books also offer a few surprises but also a lot that reflects the narrative quite well. I’m not going to go into details here as that would go a bit too far for an R-related blog – but if you are interested in in-depth discussions about the books, email me… 

sessionInfo()

## R version 3.4.3 (2017-11-30)

## Platform: x86\_64-apple-darwin15.6.0 (64-bit)

## Running under: macOS High Sierra 10.13.3

##

## Matrix products: default

## BLAS: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRblas.0.dylib

## LAPACK: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRlapack.dylib

##

## locale:

## [1] de\_DE.UTF-8/de\_DE.UTF-8/de\_DE.UTF-8/C/de\_DE.UTF-8/de\_DE.UTF-8

##

## attached base packages:

## [1] methods stats graphics grDevices utils datasets base

##

## other attached packages:

## [1] bindrcpp\_0.2 ggraph\_1.0.1 tidygraph\_1.1.0

## [4] forcats\_0.3.0 stringr\_1.3.0 dplyr\_0.7.4

## [7] purrr\_0.2.4 tidyr\_0.8.0 tibble\_1.4.2

## [10] ggplot2\_2.2.1.9000 tidyverse\_1.2.1 readr\_1.1.1

##

## loaded via a namespace (and not attached):

## [1] nlme\_3.1-131.1 bitops\_1.0-6 lubridate\_1.7.3

## [4] RColorBrewer\_1.1-2 httr\_1.3.1 prabclus\_2.2-6

## [7] rprojroot\_1.3-2 tools\_3.4.3 backports\_1.1.2

## [10] utf8\_1.1.3 R6\_2.2.2 KernSmooth\_2.23-15

## [13] lazyeval\_0.2.1 colorspace\_1.3-2 trimcluster\_0.1-2

## [16] nnet\_7.3-12 withr\_2.1.1.9000 tidyselect\_0.2.4

## [19] gridExtra\_2.3 mnormt\_1.5-5 compiler\_3.4.3

## [22] cli\_1.0.0 rvest\_0.3.2 TSP\_1.1-5

## [25] influenceR\_0.1.0 xml2\_1.2.0 labeling\_0.3

## [28] bookdown\_0.7 diptest\_0.75-7 caTools\_1.17.1

## [31] scales\_0.5.0.9000 DEoptimR\_1.0-8 robustbase\_0.92-8

## [34] mvtnorm\_1.0-7 psych\_1.7.8 digest\_0.6.15

## [37] foreign\_0.8-69 rmarkdown\_1.8 pkgconfig\_2.0.1

## [40] htmltools\_0.3.6 rlang\_0.2.0.9000 readxl\_1.0.0

## [43] rstudioapi\_0.7 bindr\_0.1 jsonlite\_1.5

## [46] mclust\_5.4 gtools\_3.5.0 dendextend\_1.7.0

## [49] magrittr\_1.5 modeltools\_0.2-21 Rcpp\_0.12.15

## [52] munsell\_0.4.3 viridis\_0.5.0 stringi\_1.1.6

## [55] whisker\_0.3-2 yaml\_2.1.17 MASS\_7.3-49

## [58] flexmix\_2.3-14 gplots\_3.0.1 plyr\_1.8.4

## [61] grid\_3.4.3 parallel\_3.4.3 gdata\_2.18.0

## [64] ggrepel\_0.7.0 crayon\_1.3.4 udunits2\_0.13

## [67] lattice\_0.20-35 haven\_1.1.1 hms\_0.4.1

## [70] knitr\_1.20 pillar\_1.2.1 igraph\_1.1.2

## [73] fpc\_2.1-11 stats4\_3.4.3 reshape2\_1.4.3

## [76] codetools\_0.2-15 glue\_1.2.0 gclus\_1.3.1

## [79] evaluate\_0.10.1 blogdown\_0.5 modelr\_0.1.1

## [82] tweenr\_0.1.5 foreach\_1.4.4 cellranger\_1.1.0

## [85] gtable\_0.2.0 kernlab\_0.9-25 assertthat\_0.2.0

## [88] xfun\_0.1 ggforce\_0.1.1 broom\_0.4.3

## [91] class\_7.3-14 viridisLite\_0.3.0 seriation\_1.2-3

## [94] iterators\_1.0.9 registry\_0.5 units\_0.5-1

## [97] cluster\_2.0.6