It is because I am not a graph analysis expert that I though it important to write this article. For someone who thinks in terms of single rectangular data sets, it is a bit of a mental leap to understand how to apply *tidy* principles to a more robust object, such as a graph table. Thankfully, there are two packages that make this work much easier:

* [tidygraph](https://github.com/thomasp85/tidygraph) – Provides a way for dplyr to interact with graphs
* [ggraph](https://github.com/thomasp85/ggraph) – Extension to ggplot2 for graph analysis

**Quick intro**

Simply put, graph theory studies relationships between objects in a group. Visually, we can think of a graph as a series of interconnected circles, each representing a member of a group, such as people in a Social Network. Lines drawn between the circles represent a relationship between the members, such as friendships in a Social Network. Graph analysis helps with figuring out things such as the influence of a certain member, or how many friends are in between two members. A more formal definition and detailed explanation of Graph Theory can be found in [Wikipedia here](https://en.wikipedia.org/wiki/Graph_theory).

**Example**

Using an example, this article will introduce concepts of graph analysis work, and how tidyverse and tidyverse-adjacent tools can be used for such analysis.

**Data source**

The [tidytuesday](https://github.com/rfordatascience/tidytuesday) weekly project encourages new and experienced users to use the tidyverse tools to analyze data sets that change every week. I have been using that opportunity to lean new tools and techniques. One of the most recent data sets relates to French trains; it contains aggregate daily total trips per connecting stations.

library(readr)

url <- "https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2019/2019-02-26/small\_trains.csv"

small\_trains <- read\_csv(url)

head(small\_trains)

## # A tibble: 6 x 13

## year month service departure\_stati… arrival\_station journey\_time\_avg

##

## 1 2017 9 Nation… PARIS EST METZ 85.1

## 2 2017 9 Nation… REIMS PARIS EST 47.1

## 3 2017 9 Nation… PARIS EST STRASBOURG 116.

## 4 2017 9 Nation… PARIS LYON AVIGNON TGV 161.

## 5 2017 9 Nation… PARIS LYON BELLEGARDE (AI… 164.

## 6 2017 9 Nation… PARIS LYON BESANCON FRANC… 129.

## # … with 7 more variables: total\_num\_trips ,

## # avg\_delay\_all\_departing , avg\_delay\_all\_arriving ,

## # num\_late\_at\_departure , num\_arriving\_late ,

## # delay\_cause , delayed\_number

**Data Preparation**

Even though it was meant to analyze delays, I thought it would be interesting to use the data to understand how stations connect with each other. A new summarized data set is created, called *routes*, which contains a single entry for each connected station. It also includes the average journey time it takes to go between stations.

library(dplyr)

routes <- small\_trains %>%

group\_by(departure\_station, arrival\_station) %>%

summarise(journey\_time = mean(journey\_time\_avg)) %>%

ungroup() %>%

mutate(from = departure\_station,

to = arrival\_station) %>%

select(from, to, journey\_time)

routes

## # A tibble: 130 x 3

## from to journey\_time

##

## 1 AIX EN PROVENCE TGV PARIS LYON 186.

## 2 ANGERS SAINT LAUD PARIS MONTPARNASSE 97.5

## 3 ANGOULEME PARIS MONTPARNASSE 146.

## 4 ANNECY PARIS LYON 225.

## 5 ARRAS PARIS NORD 52.8

## 6 AVIGNON TGV PARIS LYON 161.

## 7 BARCELONA PARIS LYON 358.

## 8 BELLEGARDE (AIN) PARIS LYON 163.

## 9 BESANCON FRANCHE COMTE TGV PARIS LYON 131.

## 10 BORDEAUX ST JEAN PARIS MONTPARNASSE 186.

## # … with 120 more rows

The next step is to transform the tidy data set, into a graph table. In order to prepare *routes* for this transformation, it has to contain two variables specifically named: *from* and *to*, which are the names that tidygraph expects to see. Those variables should contain the name of each member (e.g., “AIX EN PROVENCE TGV”), and the relationship (“AIX EN PROVENCE TGV” -> “PARIS LYON”) .

In graph terminology, a member of the group is called a **node** (or vertex) in the graph, and a relationship between nodes is called an **edge**.

library(tidygraph)

graph\_routes <- as\_tbl\_graph(routes)

graph\_routes

## # A tbl\_graph: 59 nodes and 130 edges

## #

## # A directed simple graph with 1 component

## #

## # Node Data: 59 x 1 (active)

## name

##

## 1 AIX EN PROVENCE TGV

## 2 ANGERS SAINT LAUD

## 3 ANGOULEME

## 4 ANNECY

## 5 ARRAS

## 6 AVIGNON TGV

## # … with 53 more rows

## #

## # Edge Data: 130 x 3

## from to journey\_time

##

## 1 1 39 186.

## 2 2 40 97.5

## 3 3 40 146.

## # … with 127 more rows

The as\_tbl\_graph() function splits the *routes* table into two:

1. Node Data – Contains all of the unique values found in the *from* and *to* variables. In this case, it is a table with a single column containing the names of all of the stations.
2. Edge Data – Is a table of all relationships between *from* and *to*. A peculiarity of tidygraph is that it uses the row position of the node as the identifier for *from* and *to*, instead of its original name.

Another interesting thing about tidygraph is that it allows us to attach more information about the node or edge in an additional column. In this case, *journey\_time* is not really needed to create the graph table, but it may be needed for the analysis we plan to perform. The as\_tbl\_graph() function automatically created the column for us.

Thinking about *graph\_routes* as two tibbles inside a larger table graph, was one of the two major mental breakthroughs I had during this exercise. At that point, it became evident that dplyr needs a way to know which of the two tables (nodes or edges) to perform the transformations on. In tidygraph, this is done using the activate() function. To showcase this, the nodes table will be “activated” in order to add two new string variables derived from *name*.

library(stringr)

graph\_routes <- graph\_routes %>%

activate(nodes) %>%

mutate(

title = str\_to\_title(name),

label = str\_replace\_all(title, " ", "\n")

)

graph\_routes

## # A tbl\_graph: 59 nodes and 130 edges

## #

## # A directed simple graph with 1 component

## #

## # Node Data: 59 x 3 (active)

## name title label

##

## 1 AIX EN PROVENCE TGV Aix En Provence Tgv "Aix\nEn\nProvence\nTgv"

## 2 ANGERS SAINT LAUD Angers Saint Laud "Angers\nSaint\nLaud"

## 3 ANGOULEME Angouleme Angouleme

## 4 ANNECY Annecy Annecy

## 5 ARRAS Arras Arras

## 6 AVIGNON TGV Avignon Tgv "Avignon\nTgv"

## # … with 53 more rows

## #

## # Edge Data: 130 x 3

## from to journey\_time

##

## 1 1 39 186.

## 2 2 40 97.5

## 3 3 40 146.

## # … with 127 more rows

It was really impressive how easy it was to manipulate the graph table, because once one of the two tables are activated, all of the changes can be made using tidyverse tools. The same approach can be used to extract data from the graph table. In this case, a list of all the stations is pulled into a single character vector.

stations <- graph\_routes %>%

activate(nodes) %>%

pull(title)

stations

## [1] "Aix En Provence Tgv" "Angers Saint Laud"

## [3] "Angouleme" "Annecy"

## [5] "Arras" "Avignon Tgv"

## [7] "Barcelona" "Bellegarde (Ain)"

## [9] "Besancon Franche Comte Tgv" "Bordeaux St Jean"

## [11] "Brest" "Chambery Challes Les Eaux"

## [13] "Dijon Ville" "Douai"

## [15] "Dunkerque" "Francfort"

## [17] "Geneve" "Grenoble"

## [19] "Italie" "La Rochelle Ville"

## [21] "Lausanne" "Laval"

## [23] "Le Creusot Montceau Montchanin" "Le Mans"

## [25] "Lille" "Lyon Part Dieu"

## [27] "Macon Loche" "Madrid"

## [29] "Marne La Vallee" "Marseille St Charles"

## [31] "Metz" "Montpellier"

## [33] "Mulhouse Ville" "Nancy"

## [35] "Nantes" "Nice Ville"

## [37] "Nimes" "Paris Est"

## [39] "Paris Lyon" "Paris Montparnasse"

## [41] "Paris Nord" "Paris Vaugirard"

## [43] "Perpignan" "Poitiers"

## [45] "Quimper" "Reims"

## [47] "Rennes" "Saint Etienne Chateaucreux"

## [49] "St Malo" "St Pierre Des Corps"

## [51] "Strasbourg" "Stuttgart"

## [53] "Toulon" "Toulouse Matabiau"

## [55] "Tourcoing" "Tours"

## [57] "Valence Alixan Tgv" "Vannes"

## [59] "Zurich"

**Visualizing**

In graphs, the absolute position of the each node is not as relevant as it is with other kinds of visualizations. A very minimal ggplot2 theme is set to make it easier to view the plotted graph.

library(ggplot2)

thm <- theme\_minimal() +

theme(

legend.position = "none",

axis.title = element\_blank(),

axis.text = element\_blank(),

panel.grid = element\_blank(),

panel.grid.major = element\_blank(),

)

theme\_set(thm)

To create the plot, start with ggraph() instead of ggplot2(). The ggraph package contains geoms that are unique to graph analysis. The package contains geoms to specifically plot nodes, and other geoms for edges.

As a first basic test, the *point* geom will be used, but instead of callinggeom\_point(), we call geom\_node\_point(). The edges are plotted using geom\_edge\_diagonal().

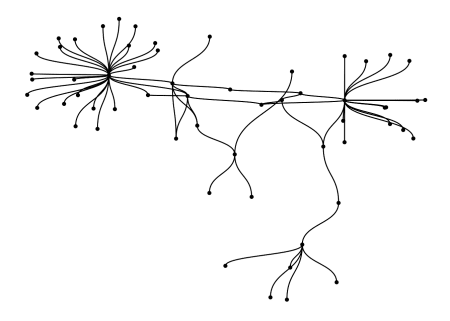
library(ggraph)

graph\_routes %>%

ggraph(layout = "kk") +

geom\_node\_point() +

geom\_edge\_diagonal()



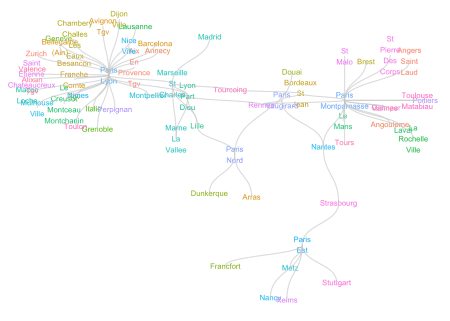
To make it easier to see where each station is placed in this plot, the geom\_node\_text() is used. Just as with regular geoms in ggplot2, other attributes such as size, color, and alpha can be modified.

graph\_routes %>%

ggraph(layout = "kk") +

geom\_node\_text(aes(label = label, color = name), size = 3) +

geom\_edge\_diagonal(color = "gray", alpha = 0.4)



**Morphing time!**

The second mental leap was understanding how a graph algorithm is applied. Typically, the output of a model function is a model object, not a data object. With tidygraph, the process begins and ends with a graph table. The steps are these:

1. Start with a graph table
2. Temporarily transform the graph to comply with the model that is requested (morph())
3. Add additional transformations to the morphed data using dplyr (optional)
4. Restore the original graph table, but modified to keep the changes made during the morph

The shortest path algorithm defines the “length” as the number of edges in between two nodes. There may be multiple routes to get from point A to point B, but the algorithm chooses the one with the fewest number of “hops”. The way to call the algorithm is inside the morph() function. Even though to\_shortest\_path() is a function in itself, and it is possible run it without morph(), it is not meant to be used that way. In the example, the *journey\_time* is used as weights to help the algorithm find an optimal route between the *Arras* and the *Nancy* stations. The print output of the morphed graph will not be like the original graph table.

from <- which(stations == "Arras")

to <- which(stations == "Nancy")

shortest <- graph\_routes %>%

morph(to\_shortest\_path, from, to, weights = journey\_time)

shortest

## # A tbl\_graph temporarily morphed to a shortest path representation

## #

## # Original graph is a directed simple graph with 1 component

## # consisting of 59 nodes and 130 edges

It is possible to make more transformations with the use of activate() and dplyr functions. The results can be previewed, or committed back to the original R variable using unmorph(). By default, nodes are active in a morphed graph, so there is no need to set that explicitly.

shortest %>%

mutate(selected\_node = TRUE) %>%

unmorph()

## # A tbl\_graph: 59 nodes and 130 edges

## #

## # A directed simple graph with 1 component

## #

## # Node Data: 59 x 4 (active)

## name title label selected\_node

##

## 1 AIX EN PROVENCE T… Aix En Provence T… "Aix\nEn\nProvence\n… NA

## 2 ANGERS SAINT LAUD Angers Saint Laud "Angers\nSaint\nLaud" NA

## 3 ANGOULEME Angouleme Angouleme NA

## 4 ANNECY Annecy Annecy NA

## 5 ARRAS Arras Arras TRUE

## 6 AVIGNON TGV Avignon Tgv "Avignon\nTgv" NA

## # … with 53 more rows

## #

## # Edge Data: 130 x 3

## from to journey\_time

##

## 1 1 39 186.

## 2 2 40 97.5

## 3 3 40 146.

## # … with 127 more rows

While it was morphed, only the few nodes that make up the connections between the Arras and Nancy stations were selected. A simple mutate() adds a new variable called *selected\_node*, which tags those nodes with TRUE. The new variable and value is retained once the rest of the nodes are restored via the unmorph() command.

To keep the change, the *shortest* variable is updated with the changes made to both edges and nodes.

shortest <- shortest %>%

mutate(selected\_node = TRUE) %>%

activate(edges) %>%

mutate(selected\_edge = TRUE) %>%

unmorph()

The next step is to coerce each NA into a 1, and the shortest route into a 2. This will allow us to easily re-arrange the order that the edges are drawn in the plot, ensuring that the route will be drawn at the top.

shortest <- shortest %>%

activate(nodes) %>%

mutate(selected\_node = ifelse(is.na(selected\_node), 1, 2)) %>%

activate(edges) %>%

mutate(selected\_edge = ifelse(is.na(selected\_edge), 1, 2)) %>%

arrange(selected\_edge)

shortest

## # A tbl\_graph: 59 nodes and 130 edges

## #

## # A directed simple graph with 1 component

## #

## # Edge Data: 130 x 4 (active)

## from to journey\_time selected\_edge

##

## 1 1 39 186. 1

## 2 2 40 97.5 1

## 3 3 40 146. 1

## 4 4 39 225. 1

## 5 6 39 161. 1

## 6 7 39 358. 1

## # … with 124 more rows

## #

## # Node Data: 59 x 4

## name title label selected\_node

##

## 1 AIX EN PROVENCE T… Aix En Provence T… "Aix\nEn\nProvence\n… 1

## 2 ANGERS SAINT LAUD Angers Saint Laud "Angers\nSaint\nLaud" 1

## 3 ANGOULEME Angouleme Angouleme 1

## # … with 56 more rows

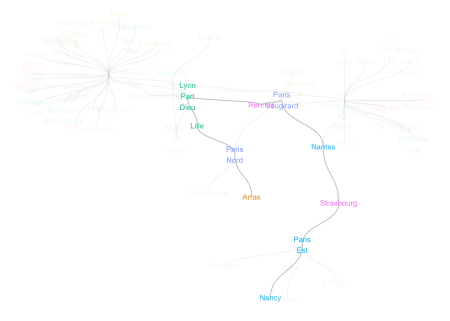
A simple way to plot the route is to use the *selected\_* variables to modify the alpha. This will highlight the shortest path, without completely removing the other stations. This is a personal design choice, so experimenting with different ways of highlighting the results is always recommended.

shortest %>%

ggraph(layout = "kk") +

geom\_edge\_diagonal(aes(alpha = selected\_edge), color = "gray") +

geom\_node\_text(aes(label = label, color =name, alpha = selected\_node ), size = 3)



The *selected\_* fields can also be used in other dplyr functions to analyze the results. For example, to know the aggregate information about the trip, *selected\_edge* is used to filter the edges, and then the totals can be calculated. There is no summarise() function for graph tables; this make sense because the graph table would become a summarized table with such a function. Since the end result we seek is a total rather than another graph table, a simple as\_tibble() command will coerce the edges, which will then allows us to finish the calculation.

shortest %>%

activate(edges) %>%

filter(selected\_edge == 2) %>%

as\_tibble() %>%

summarise(

total\_stops = n() - 1,

total\_time = round(sum(journey\_time) / 60)

)

## # A tibble: 1 x 2

## total\_stops total\_time

##

## 1 8 23

**Re-using the code**

To compile most of the code in a single chunk, here is an example of how to re-run the shortest path for a different set of stations: the Laval and Montpellier stations.

from <- which(stations == "Montpellier")

to <- which(stations == "Laval")

shortest <- graph\_routes %>%

morph(to\_shortest\_path, from, to, weights = journey\_time) %>%

mutate(selected\_node = TRUE) %>%

activate(edges) %>%

mutate(selected\_edge = TRUE) %>%

unmorph() %>%

activate(nodes) %>%

mutate(selected\_node = ifelse(is.na(selected\_node), 1, 2)) %>%

activate(edges) %>%

mutate(selected\_edge = ifelse(is.na(selected\_edge), 1, 2)) %>%

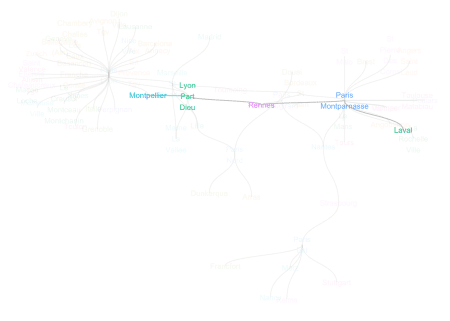
arrange(selected\_edge)

shortest %>%

ggraph(layout = "kk") +

geom\_edge\_diagonal(aes(alpha = selected\_edge), color = "gray") +

geom\_node\_text(aes(label = label, color =name, alpha = selected\_node ), size = 3)



Additional, the same code can be recycled to obtain the trip summarized data.

shortest %>%

activate(edges) %>%

filter(selected\_edge == 2) %>%

as\_tibble() %>%

summarise(

total\_stops = n() - 1,

total\_time = round(sum(journey\_time) / 60)

)

## # A tibble: 1 x 2

## total\_stops total\_time

##

## 1 3 10

**Shiny app**

To see how to use this kind of analysis inside Shiny, please refer to [this application](https://beta.rstudioconnect.com/content/4606/). It lets the user select two stations, and it returns the route, plus the summarized data. The source code is embedded in the app.