Drawing a network with ggraph and tidygraph

Ken

2017-12-30

## Introduction

[Thomas Lin Pedersen](https://github.com/thomasp85) is the author of a lot of packages, including two that deal with graphs (in the sense of networks), tidygraph for storing and handling graphs, and ggraph for drawing them, ggplot-style. I have a feeling I will be spending a lot of time with tidygraph, but in this post, I get my feet wet reading in and storing a graph and then making a picture of it.

Packages:

library(tidyverse)

## ── Attaching packages ─────────────────────────────────────────────────────────────────────────── tidyverse 1.2.1 ──

## ✔ ggplot2 2.2.1.9000 ✔ purrr 0.2.4   
## ✔ tibble 1.4.2 ✔ dplyr 0.7.4   
## ✔ tidyr 0.8.0 ✔ stringr 1.3.0   
## ✔ readr 1.1.1 ✔ forcats 0.3.0

## ── Conflicts ────────────────────────────────────────────────────────────────────────────── tidyverse\_conflicts() ──  
## ✖ dplyr::filter() masks stats::filter()  
## ✖ dplyr::lag() masks stats::lag()

library(tidygraph)

##   
## Attaching package: 'tidygraph'

## The following object is masked from 'package:stats':  
##   
## filter

library(ggraph)

## Example: the West Yorkshire train network

[West Yorkshire](https://en.wikipedia.org/wiki/West_Yorkshire) is a metropolitan county of northern England, containing the city of Leeds. It has a considerable local rail network, as shown below:

My aim is to treat the stations as nodes in a graph and rail lines between them as edges, and see how I succeed and fail in reproducing this map.

## Reading in the data and storing as a graph

I had to think about how I wanted to represent the network. If these had been plane flights, I would have had a starting point and an ending point for each one. But trains, especially local trains, stop at a long list of places one after the other. For example, a train from Bradford (Forster Square) to Ilkley will travel between Bradford, Frizinghall, Shipley, Baildon, Guiseley, etc., to Ben Rhydding and Ilkley. I decided to list all the stations on one route in a long list, one per line, with an NA at the end to indicate the end of one line and the beginning of another. The input file (part) looks like this:

head --lines=20 yorkshire.txt

## station  
## Skipton  
## Cononley  
## Steeton & Silsden  
## Keighley  
## Crossflatts  
## Bingley  
## Saltaire  
## Shipley  
## Frizinghall  
## Bradford  
## NA  
## Shipley  
## Apperley Bridge  
## Kirkstall Forge  
## Leeds  
## NA  
## Ilkley  
## Ben Rhydding  
## Burley-in-Wharfedale

In actuality, Bradford and Wakefield have two stations and Pontefract has three, but I treated each place as having only one station to simplify things. (The rail network around Wakefield Kirkgate is complicated, with one line passing under another without being connected.)

To read this in, I treat it as a one-column .csv, since some of the stations have spaces inside their names:

yorks=read\_csv("yorkshire.txt")

## Parsed with column specification:  
## cols(  
## station = col\_character()  
## )

yorks %>% print(n=20)

## # A tibble: 145 x 1  
## station   
## <chr>   
## 1 Skipton   
## 2 Cononley   
## 3 Steeton & Silsden   
## 4 Keighley   
## 5 Crossflatts   
## 6 Bingley   
## 7 Saltaire   
## 8 Shipley   
## 9 Frizinghall   
## 10 Bradford   
## 11 <NA>   
## 12 Shipley   
## 13 Apperley Bridge   
## 14 Kirkstall Forge   
## 15 Leeds   
## 16 <NA>   
## 17 Ilkley   
## 18 Ben Rhydding   
## 19 Burley-in-Wharfedale  
## 20 Menston   
## # ... with 125 more rows

Now, how to connect each station with the next one on the list? I realized that the lag function would do exactly this:

yorks %>% mutate(station2=lag(station)) %>%   
 print(n=20)

## # A tibble: 145 x 2  
## station station2   
## <chr> <chr>   
## 1 Skipton <NA>   
## 2 Cononley Skipton   
## 3 Steeton & Silsden Cononley   
## 4 Keighley Steeton & Silsden   
## 5 Crossflatts Keighley   
## 6 Bingley Crossflatts   
## 7 Saltaire Bingley   
## 8 Shipley Saltaire   
## 9 Frizinghall Shipley   
## 10 Bradford Frizinghall   
## 11 <NA> Bradford   
## 12 Shipley <NA>   
## 13 Apperley Bridge Shipley   
## 14 Kirkstall Forge Apperley Bridge   
## 15 Leeds Kirkstall Forge   
## 16 <NA> Leeds   
## 17 Ilkley <NA>   
## 18 Ben Rhydding Ilkley   
## 19 Burley-in-Wharfedale Ben Rhydding   
## 20 Menston Burley-in-Wharfedale  
## # ... with 125 more rows

and now the rows of this data frame will be the edges of my graph. Separating the routes with NA was an accidentally-smart thing to do, since if I remove all the rows with NA in them, I will get only the genuine edges I want for my graph:

yorks2 = yorks %>%   
 mutate(station2=lag(station)) %>%   
 filter(!is.na(station),!is.na(station2)) %>%   
 print(n=20)

## # A tibble: 110 x 2  
## station station2   
## <chr> <chr>   
## 1 Cononley Skipton   
## 2 Steeton & Silsden Cononley   
## 3 Keighley Steeton & Silsden   
## 4 Crossflatts Keighley   
## 5 Bingley Crossflatts   
## 6 Saltaire Bingley   
## 7 Shipley Saltaire   
## 8 Frizinghall Shipley   
## 9 Bradford Frizinghall   
## 10 Apperley Bridge Shipley   
## 11 Kirkstall Forge Apperley Bridge   
## 12 Leeds Kirkstall Forge   
## 13 Ben Rhydding Ilkley   
## 14 Burley-in-Wharfedale Ben Rhydding   
## 15 Menston Burley-in-Wharfedale  
## 16 Guiseley Menston   
## 17 Baildon Guiseley   
## 18 Shipley Baildon   
## 19 Kirkstall Forge Guiseley   
## 20 Burley Park Leeds   
## # ... with 90 more rows

(something I discovered the other day: if you put a print on the end of a pipeline, it prints the result *even if you have saved it in a variable*.)

tidygraph has a function for turning a data frame structured like this, with edges as rows, into an actual tidy graph:

yorks\_g <- as\_tbl\_graph(yorks2)  
yorks\_g

## # A tbl\_graph: 98 nodes and 110 edges  
## #  
## # A directed acyclic simple graph with 1 component  
## #  
## # Node Data: 98 x 1 (active)  
## name   
## <chr>   
## 1 Cononley   
## 2 Steeton & Silsden  
## 3 Keighley   
## 4 Crossflatts   
## 5 Bingley   
## 6 Saltaire   
## # ... with 92 more rows  
## #  
## # Edge Data: 110 x 2  
## from to  
## <int> <int>  
## 1 1 97  
## 2 2 1  
## 3 3 2  
## # ... with 107 more rows

The nodes and edges are shown separately, with the nodes numbered (those numbers are used in the edges).

## Extracting the nodes and edges

tidygraph uses the same pipeline ideas as the tidyverse, but extra care is required because a tidy graph object has both nodes and edges in it. An extra verb activate lets you specify which one you would like to be “primary” (but you can also access the other one). Piping the output of activate into as\_tibble lets you save the nodes or edges as a data frame, separately:

nodes = yorks\_g %>%   
 activate(nodes) %>%   
 as\_tibble() %>%   
 mutate(node\_number=row\_number()) %>%  
 print(n=10)

## # A tibble: 98 x 2  
## name node\_number  
## <chr> <int>  
## 1 Cononley 1  
## 2 Steeton & Silsden 2  
## 3 Keighley 3  
## 4 Crossflatts 4  
## 5 Bingley 5  
## 6 Saltaire 6  
## 7 Shipley 7  
## 8 Frizinghall 8  
## 9 Bradford 9  
## 10 Apperley Bridge 10  
## # ... with 88 more rows

edges = yorks\_g %>%   
 activate(edges) %>%   
 as\_tibble() %>% print(n=10)

## # A tibble: 110 x 2  
## from to  
## <int> <int>  
## 1 1 97  
## 2 2 1  
## 3 3 2  
## 4 4 3  
## 5 5 4  
## 6 6 5  
## 7 7 6  
## 8 8 7  
## 9 9 8  
## 10 10 7  
## # ... with 100 more rows

I saved the node numbers as well as names into the nodes data frame. This lets me look up the *names* of the nodes referred to in the edges, to convince myself that I have something sensible:

edges %>%   
 left\_join(nodes,by=c("from"="node\_number")) %>%   
 left\_join(nodes,by=c("to"="node\_number")) %>%   
 print(n=20)

## # A tibble: 110 x 4  
## from to name.x name.y   
## <int> <int> <chr> <chr>   
## 1 1 97 Cononley Skipton   
## 2 2 1 Steeton & Silsden Cononley   
## 3 3 2 Keighley Steeton & Silsden   
## 4 4 3 Crossflatts Keighley   
## 5 5 4 Bingley Crossflatts   
## 6 6 5 Saltaire Bingley   
## 7 7 6 Shipley Saltaire   
## 8 8 7 Frizinghall Shipley   
## 9 9 8 Bradford Frizinghall   
## 10 10 7 Apperley Bridge Shipley   
## 11 11 10 Kirkstall Forge Apperley Bridge   
## 12 12 11 Leeds Kirkstall Forge   
## 13 13 98 Ben Rhydding Ilkley   
## 14 14 13 Burley-in-Wharfedale Ben Rhydding   
## 15 15 14 Menston Burley-in-Wharfedale  
## 16 16 15 Guiseley Menston   
## 17 17 16 Baildon Guiseley   
## 18 7 17 Shipley Baildon   
## 19 11 16 Kirkstall Forge Guiseley   
## 20 18 12 Burley Park Leeds   
## # ... with 90 more rows

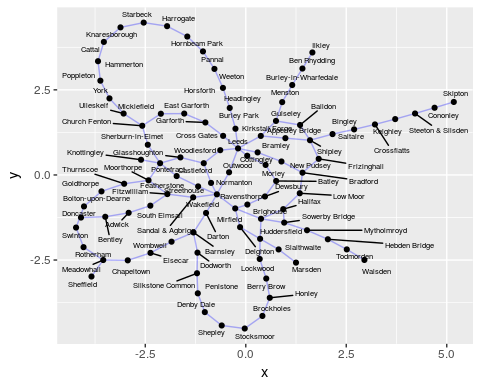
This is what I started with.

## Drawing the graph

Thomas Lin Pedersen also wrote the package ggraph which allows ggplot-style plotting of graphs. For our example, this was my ultimate aim: I wanted to see how well the connection information enabled us to reproduce the map above.

Graphs can be plotted in many different ways. For this graph, I know there is a representation in two dimensions, since the stations are scattered across the 2D-space of West Yorkshire. In ggraph, this is controlled by the layout parameter of the ggraph command. The Kamada-Kawai method tries to plot neighbouring nodes a constant distance apart, and chooses their locations as if they are connected by unit-length springs. I find this works well here. In addition, I want to draw the edges (as lines), draw the nodes (as points), and label the nodes by which station they are. These are accomplished by geom\_edge\_link, geom\_node\_point and geom\_node\_text. Additional notes: the alpha=0.3 draws the edges partly transparent, so that the station names stand out; the use of repel=T “repels” the station names away from the points they represent so that you can see them (in the manner of geom\_text\_repel):

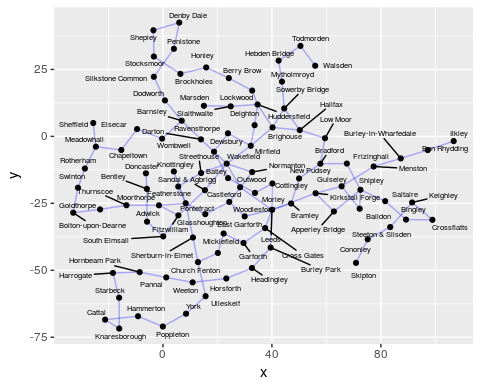
ggraph(yorks\_g,layout="kk")+  
 geom\_edge\_link(colour="blue",alpha=0.3)+  
 geom\_node\_point()+  
 geom\_node\_text(aes(label=name),size=2,repel=T)



This seems to do a nice job of reproducing the topology in a clear fashion with no overlaps where there shouldn’t be any. (I tried this using the separate stations, Westgate and Kirkgate, in Wakefield, and the display was a lot more confusing.) It doesn’t get compass directions right (it seems to have west and east reversed), but we had no right to expect that it would.

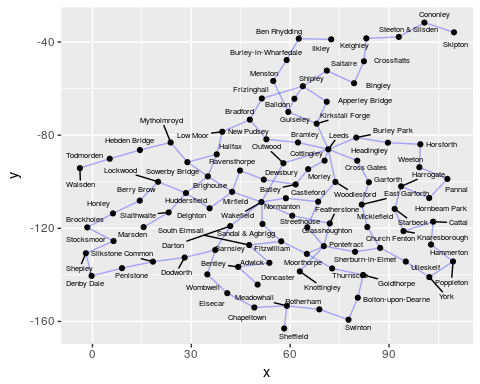
Another plausible layout is the one called lgl. How does that work here? We leave everything else the same:

ggraph(yorks\_g,layout="lgl")+  
 geom\_edge\_link(colour="blue",alpha=0.3)+  
 geom\_node\_point()+  
 geom\_node\_text(aes(label=name),size=2,repel=T)



This is much less pleasing, especially in the middle. Can I make it try more iterations than the default 150?

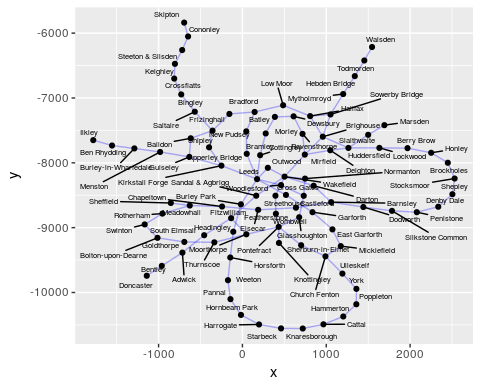
ggraph(yorks\_g,layout="lgl",maxiter=400)+  
 geom\_edge\_link(colour="blue",alpha=0.3)+  
 geom\_node\_point()+  
 geom\_node\_text(aes(label=name),size=2,repel=T)



This is a lot better, but still has some unnecessary crossings.

One more try: the GEM “force-directed” layout:

ggraph(yorks\_g,layout="gem")+  
 geom\_edge\_link(colour="blue",alpha=0.3)+  
 geom\_node\_point()+  
 geom\_node\_text(aes(label=name),size=2,repel=T)



This is better, but I like the kk one the best.