Mutating and filtering

Your name here

We start with a “setup” block of code, that loads our libraries and initializes our settings. This block should work for basically all of our assignments this semester, so you can just copy and paste it.

```r
knitr::opts_chunk$set(echo = T)
knitr::opts_knit$set(root.dir = './')
library(igraph)
library(tidygraph)
library(ggraph)
library(tidyverse)
set_graph_style(family='sans') # This sets the default style to the graph style
```

R has a lot of great tools for working with tables (also called dataframes or tibbles). One really powerful and relatively intuitive set of tools is called the “tidyverse”. Tools in the tidyverse make assumptions about what data will look like—rows represent observations and columns represent variables that describe that observation.

The tidygraph package extends that paradigm to networks, by representing networks as two tables: 1) a table of nodes and node attributes and 2) a table of edges and edge attributes.

This lets you work with networks using many of the same tools that have been developed for working on other types of data.

Just to make things simple, we’ll use the Zachary Karate club data for most of this tutorial. Let’s load it, and save it as `G`

```r
G <- create_notable('Zachary')
```

If we look at `G` we can see that it’s already a `tbl_graph` object. It has 34 nodes and 78 edges.

```r
G
```

```
# A tbl_graph: 34 nodes and 78 edges
#
# An undirected simple graph with 1 component
#
# Node Data: 34 x 0 (active)
#
# Edge Data: 78 x 2
   from    to
  <int> <int>
1     1     2
2     1     3
3     1     4
# i 75 more rows
```

Getting to the data

Activating a table

Because a network object is really composed of two tables, we have to let R know which table we want to manipulate. This is done using `activate(nodes)` or `activate(edges)`.

For example, the code below activates the node table and then uses `mutate` (which we will talk more about further down) to create a variable called `degree` in the nodes table.

(Note that the code throughout this tutorial uses "pipes". Pipes (`|>`) let you express a sequence of operations, by taking the output of the previous operation and using it as the input of the next operation.)

```r
G |>
  activate(nodes) |>
  mutate(degree = centrality_degree())
```

```
# A tbl_graph: 34 nodes and 78 edges
#
# An undirected simple graph with 1 component
#
# Node Data: 34 x 1 (active)
   degree
    <dbl>
 1     16
 2      9
 3     10
 4      6
 5      3
 6      4
 7      4
 8      4
 9      5
10      2
# i 24 more rows
#
# Edge Data: 78 x 2
   from    to
  <int> <int>
1     1     2
2     1     3
3     1     4
# i 75 more rows
```

Mutate

`mutate` is a function that just creates a new column in a spreadsheet/dataframe (in our case, either the one for the nodes or the one for the edges). Often, we will want to calculate some network statistic (for example, the code above calculates the degree for each node).

However, you can also create other kinds of columns. For example, you could use `mutate` to give names to each of the nodes. The code below creates a small network and adds names to it. (The `c` is how we create a list of things in R)

```r
play_erdos_renyi(5, .5) |>
activate(nodes) |>
  mutate(names = c('Alfred','Bonnie', 'Clyde', 'Doug','Enola'))
```

```
Warning: `play_erdos_renyi()` was deprecated in tidygraph 1.3.0.
i Please use `play_gnp()` instead.
```

```
# A tbl_graph: 5 nodes and 11 edges
#
# A directed simple graph with 1 component
#
# Node Data: 5 x 1 (active)
  names
  <chr>
```

```
1 Alfred
2 Bonnie
3 Clyde
4 Doug
5 Enola
#
# Edge Data: 11 x 2
   from    to
  <int> <int>
1     2     1
2     3     1
3     5     1
# i 8 more rows
```

Excercise

Modify the code on lines 50-54 to calculate the betweenness centrality instead of the degree of `G`. Name the column `betweenness` (Hint: use the `centrality_betweenness()` function. (Here is the list of all centrality functions.)

```
# Your code here
```

Note that if we look at `G` again, the `degree` and `betweenness` columns that we created no longer appear.

```
G
```

```
# A tbl_graph: 34 nodes and 78 edges
#
# An undirected simple graph with 1 component
#
# Node Data: 34 x 0 (active)
#
# Edge Data: 78 x 2
   from    to
  <int> <int>
1     1     2
2     1     3
3     1     4
# i 75 more rows
```

This is because we didn't save them; by default mutate will just produce a temporary version of the new column. But don't worry—saving it is super easy; we can just re-save it as G like this.

```r
G <- G |>
  activate(nodes) |>
  mutate(degree = centrality_degree())
```

The `<-` means take whatever is on the right side, and save it to the variable name on the left side. So, this takes all of the output of the `mutate` operation and saves it. Now, when we look at `G`, we'll see the `degree` column.

```r
G
```

```
# A tbl_graph: 34 nodes and 78 edges
#
# An undirected simple graph with 1 component
#
# Node Data: 34 x 1 (active)
   degree
    <dbl>
 1     16
 2      9
 3     10
 4      6
 5      3
 6      4
 7      4
 8      4
 9      5
10      2
# i 24 more rows
#
# Edge Data: 78 x 2
   from    to
  <int> <int>
1     1     2
2     1     3
3     1     4
# i 75 more rows
```

Exercise

Do this yourself - add the `betweenness` column to `G`. When you're done, the nodes spreadsheet of `G` should have two new columns - `degree` and `betweenness`.

```
# Your code here
```

Filtering

The one other thing we're going to learn about today is filtering. Sometimes we may want to know about some subset of the nodes or edges in a network.

Usually, we would do this based on some attribute of the nodes—for example, looking just at the network of the older (or younger) members of the group.

Let's load in a richer network from the `networkdata` library. (See R Lab 2 for instructions on how to install it if this code doesn't work.)

This code loads the `networkdata` library and loads the `ht_advice` network. It then changes it from an `igraph` network to a `tbl_graph` network object.

```
library(networkdata)
```

```


Attaching package: 'networkdata'


The following object is masked from 'package:dplyr':

    starwars
```

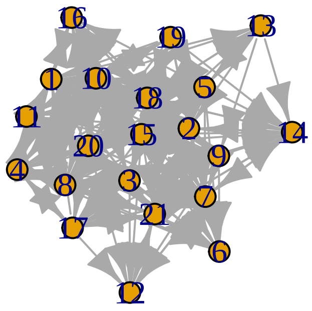
```
ht_advice <- ht_advice |> as_tbl_graph()

ht_advice
```

```
# A tbl_graph: 21 nodes and 190 edges
#
# A directed simple graph with 1 component
#
# Node Data: 21 x 4 (active)
     age tenure level  dept
   <dbl>  <dbl> <dbl> <dbl>
```

```
 1    33      9     3     4
 2    42     20     2     4
 3    40     13     3     2
 4    33      8     3     4
 5    32      3     3     2
 6    59     28     3     1
 7    55     30     1     0
 8    34     11     3     1
 9    62      5     3     2
10    37      9     3     3
# i 11 more rows
#
# Edge Data: 190 x 2
   from    to
  <int> <int>
1     1     2
2     1     4
3     1     8
# i 187 more rows
```

We can see that there are 21 nodes with four attributes: `age`, `tenure`, `level`, and `dept`. These nodes have 190 edges. Let's plot the network.

```r
plot(ht_advice)
```



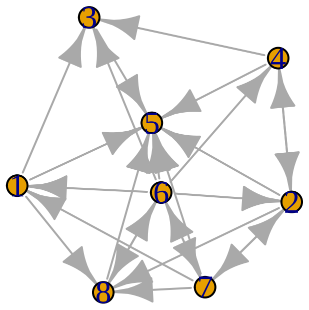
Now we're going to filter the graph. Maybe we just want to see the network from the Sales department. Let's pretend like that's department two, and filter to just those nodes.

Remember, we first `activate` the `nodes`. Then, we use `filter` to write a condition or set of conditions that will evaluate to True for each of the nodes that we want to keep. Usually, this will involve one node attribute (like `dept`) and a comparison, like `<`, `>`, or `==`. Note that when we are testing *if* things are equal, we use two equal signs in R. This is because one equals sign can be used to assign something to a variable.

```
ht_advice |>
  activate(nodes) |>
  filter(dept == 2) |> # Filter to just nodes in dept 2
  print() |> # We can throw in print() and it will print the output of the last step. In this
  plot()
```

```
# A tbl_graph: 8 nodes and 25 edges
#
# A directed simple graph with 1 component
#
# Node Data: 8 x 4 (active)
    age tenure level  dept
  <dbl>  <dbl> <dbl> <dbl>
1    40     13     3     2
```

```
2    32      3     3     2
3    62      5     3     2
4    48      0     3     2
5    43     10     2     2
6    40      8     3     2
7    32      5     3     2
8    38     12     3     2
#
# Edge Data: 25 x 2
   from    to
  <int> <int>
1     1     3
2     1     5
3     1     8
# i 22 more rows
```



Exercise

Filter the `ht_advice` network to just the people who are older than 35.

How many nodes are there?

How many edges?

(Hint: you can use `print` like I do above to print information about the graph before you plot it)

```
# Your code here
```

Challenge Exercise

Filter the `ht_advice` network to only those nodes whose `tenure` is less than the median value for `tenure`. Plot the graph.

```
## Your code here
```

Chaining Filtering and Mutating

Finally, we can use `mutate` and `filter` together in powerful ways.

For example, we may want to get the betweenness centrality of just the Sales department.

```
ht_advice |>
  activate(nodes) |>
  filter(dept == 2) |>
  mutate(betweenness = centrality_betweenness())
```

```
# A tbl_graph: 8 nodes and 25 edges
#
# A directed simple graph with 1 component
#
# Node Data: 8 x 5 (active)
    age tenure level  dept betweenness
  <dbl>  <dbl> <dbl> <dbl>       <dbl>
1    40     13     3     2         0.5
2    32      3     3     2         4.5
3    62      5     3     2         0
4    48      0     3     2         1
5    43     10     2     2         0
6    40      8     3     2         9
7    32      5     3     2         3
8    38     12     3     2         5
#
# Edge Data: 25 x 2
   from    to
  <int> <int>
1     1     3
2     1     5
3     1     8
# i 22 more rows
```

Remember that the order matters - if we ran `mutate` before `filter` then it would calculate the betweenness centrality *before* filtering.

Exercise

Sometimes, that's what we want. For the last exercise, calculate the degree centrality for `ht_advice`, save it in a column called `degree`, and then filter to just the nodes with `degree` of at least 2.

```
## Your code here
```