Co-lab Shiny Workshop

Integrating Shiny and `visNetwork`

November 19, 2019

thomas.balmat@duke.edu
rescomputing@duke.edu

A network graph is effective in showing relationships between associated entities. Entities, or items that are related, are referred to as vertices and the relationships that connect them are referred to as edges. A graph with many vertices or edges can be very dense, making central features, such as a vertex with many edges or a particular type of edge that connects many pairs of vertices, difficult to identify. Iterated adjustment of rendering features, such as the number of vertices displayed, opacity of edges, etc., can aid in presenting a graph that reveals important features of the system being studied. In this session, we will use the reactive feature of Shiny to develop an app that renders and adjusts a network graph in a genome wide association study (GWAS) pleiotropy setting. Pleiotropy is a term used to describe the process of comparing significant results from two independent GWAS studies, with the objective of identifying associations between phenotypes (observed traits) from both studies. The app will produce two basic graphs: one that generates two sets of vertices, one from each GWAS, and joins them (edges) by common single nucleotide polymorphysm (SNP, a single locus, or position, on a specific chromosome) and the other generates vertices from SNPs that are joined by phenotype.

1 Overview

- Preliminaries
  - What can `Shiny and visNetwork` do for you?
  - What are your expectations of this workshop?
  - Interest in R Day?
- Examples
- Resources
- Anatomy of a Shiny app
- Workshop material
  - From github (execute locally)
  - From RStudio Cloud
- Review previous app, `plotly()` GWAS pleiotropy app
- `visnetwork()` pleiotropy app
  - Version 1, Basic
  - Version 2, Enhanced reactive graph controls
  - Version 3, Additional graph controls with centrality table
- Debugging

2 Examples

- `visNetwork()` features: https://datastorm-open.github.io/visNetwork/
- `visNetwork()` Shiny demo (from within R): `shiny::runApp(system.file("shiny", package="visNetwork"))`
- Delivery graph network app
- Neo4j graph database: https://neo4j.com/developer/graph-database/

3 Resources

- R
  - Books
    - Norm Matloff, *The Art of R Programming*, No Starch Press
    - Wickham and Grolemund, *R for Data Science*, O'Reilly
    - Andrews and Wainer, *The Great Migration: A Graphics Novel*, https://rss.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1740-9713.2017.01070.x
    - Friendly, *A Brief History of Data Visualization*, http://datavis.ca/papers/hbook.pdf
  - Reference cards
    - R reference card: https://cran.r-project.org/doc/contrib/Short-refcard.pdf
    - Base R: https://rstudio.com/wp-content/uploads/2016/10/r-cheat-sheet-3.pdf
    - Shiny, `ggplot`, `markdown`, `dplyr`, `tidy`: https://rstudio.com/resources/cheatsheets/
- Shiny
  - `?shiny` from the R command line
  - Click `shiny` in the `Packages` tab of RStudio
  - https://cran.r-project.org/web/packages/shiny/shiny.pdf
- `visNetwork`
  - `?visNetwork` from the R command line
  - Click `visNetwork` in the `Packages` tab of RStudio
  - https://cran.r-project.org/web/packages/visNetwork/visNetwork.pdf
- Workshop materials
  - https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-5

4 Anatomy of a Shiny App

A Shiny app is an R script executing in an active R environment that uses functions available in the Shiny package to interact with a web browser. The basic components of a Shiny script are

- `ui()` function
  - Contains your web page layout and screen objects for inputs (prompt fields) and outputs (graphs, tables, etc.)
  - Is specified in a combination of Shiny function calls and raw HTML
  - Defines variables that bind web objects to the execution portion of the app
- `server()` function

  - The execution portion of the app
  - Contains a combination of standard R statements and function calls, such as to `apply()`, `lm()`, `ggplot()`, etc., along with calls to functions from the Shiny package that enable reading of on-screen values and rendering of results

- `runApp()` function
  - Creates a process listening on a tcp port, launches a browser (optional), renders a screen by calling the specified `ui()` function, then executes the R commands in the specified server() function

5 Access Workshop Material

This document:

https://github.com/tbalmat/Duke-Co-lab/blob/master/Session-5/CourseOutline/Co-lab-Session-5-visNetwork.pdf

5.1 Execute Locally (copy workshop material from github repo)

- Copy scripts and data from https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-4
- In a separate directory, copy scripts and data from https://github.com/tbalmat/Duke-Co-lab/tree/master/Session-5
- Install Shiny package: R command `install.packages("shiny")`
- Install data tables package: R command `install.packages("ggrepel")`
- Install data tables package: R command `install.packages("plotly")`
- Install data tables package: R command `install.packages("visNetwork")`
- All workshop scripts should function locally

5.2 RStudio Cloud

- What is RStudio Cloud?
  - *We* [RStudio] *created RStudio Cloud to make it easy for professionals, hobbyists, trainers, teachers and students to do, share, teach and learn data science using R.*
- With RStudio Cloud
  - You do not need RStudio installed locally
  - Packages and data are available without installation and transfer
- Access workshop material
  - Create an Account: https://rstudio.cloud
  - Workshop project link: https://rstudio.cloud/project/580472
  - Files are in the directories `Duke-Co-lab/Shiny/Session-4-plotly` and `Duke-Co-lab/Shiny/Session-5-visNetwork`
- All workshop scripts should function on RStudio Cloud, except OS shells that are used to automate execution

6 Review previous app, `plotly()` Pleiotropy App

Overview: (git or RStudio Cloud directory) Session-4/CourseOutline/Co-lab-Session-4-plotly.pdf

Tab 1, two panel point selection graph. Discussion points (numbers are lines in `Pleiotropy.r` source file):

- (69) `geom_jitter()`
- (85) Axis text angle
- (69) User defined variable in `aes()` for appearance in hover labels
- (151) Use of `plotly()` %>% to string function calls
- (380, 425) Use of `ggplotly()` to convert a `ggplot()` object to `plotly()`
- Hover label configuration (position and ID of aesthetics)
- (380, 388) Point selection and control of events
- (396) Subsetting observations for selected point
- (102) Coloring selected points using two geoms and data subsets in `t1ComposePlot2()`
- (411) Adding custom labels with `add_annotations`, since `ggplotly()` does not convert them
- (420-425) Nested rendering of plots

Tab 2, bipartite phenotype selection graph. Discussion points (specifically, effect of Shiny features):

- Subsetting plotted points by p-filter
- Phenotype ordering by number of associated SNPs
- Display of number of SNPs per phenotype (vertex) or edge connecting phenotypes (`geom_text()`)
- Construction of phenotype point hover labels
- Coloring vertices (phenotype) and edges (lines) of selected point (left and right)
- Display of green intersection point
- SNP intersection of lines, hover labels

7 `visNetwork()` Pleiotropy App

The purpose of the pleiotropy network graph is to visually identify phenotypes from two GWAS sets that are associated with a common set of SNPs or SNPs that are associated with multiple phenotypes from both GWAS sets. Each of the following apps uses a p-significance threshold (from the GWAS regression results) to limit phenotype/SNP observations to a specified level of significance within GWAS, colors to differentiate vertices and edges by GWAS, vertex size proportional to number of edges, and transparency to reduce interference of edges.

7.1 `visNetwork()` App Version 1, Basic

Workshop files `App/V1/ui.r` and `App/V1/server.r`

Figure 1 is an example screen-shot of this app. Phenotypes forms vertices (blue from GWAS 1, yellow from GWAS 2) and SNPs common to both GWAS sets form edges. App and script features for discussion (unnn indicates line number in ui.r, snnn indicates line number in server.r):

- (s15-25) Data set
- (u51) `visNetworkOutput`() object
- (s187-189) Reactive elements
- (s192) Call to `assembleNetComponents()`
- (s74-148) `assembleNetComponents()` function. This function generates the vertices and edges to be used in graph construction.
- (s111-119) Vertex data frame contents
- (s125-135) Edge data frame contents
- (s195) Rendering of graph with call to `composeNet()`
- (s150-172) `composeNet()` function. This function generates the graph using composed vertices and edges.
- (s156-161) `visNetwork()` parameters (groups, legend, options)
- (s162-169) Enabling/disabling of "physics"

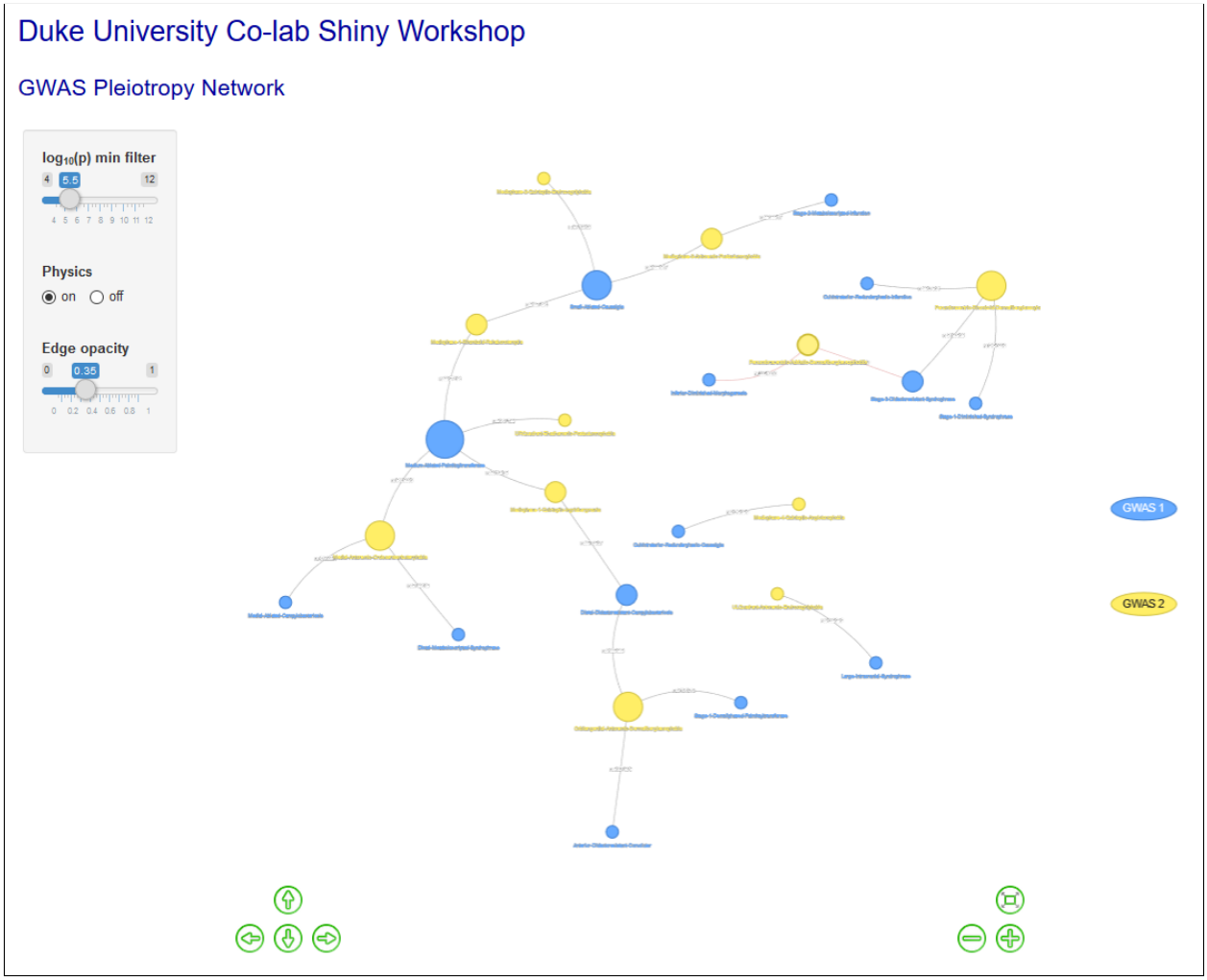

Figure 1: `visNetwork` pleiotropy app, version 1. Inter-GWAS phenotype as vertex, common SNP as edge. Basic Shiny features.

7.2 `visNetwork()` App Version 2, Enhanced Graph Controls

Workshop files `App/V2/ui.r` and `App/V2/server.r`

Figure 2 is an example screen-shot of this app with phenotype forming vertices and SNP forming edges. App and script features for discussion (unnn indicates line number in ui.r, snnn indicates line number in server.r):

- (u42, s83) Vertex and edge reconfiguration from phenotype/SNP to SNP/phenotype
- (u46, s100-104) Vertex filtering by edge count
- (s107-108, 120-121) Vertex hover label composition
- Use of global variables due to events and interrupt nature of execution
- (s285-296) Enabling of physics stabilization
- (s415-422) Physics stabilization event
- (s298-306) Enabling of shift-click event
- (s361, 380, 386, 392) `ignoreInit`
- (s488-519) Graph subnetting with shift-click (on a vertex)
- (u54, s521-538) Restore after subnetting
- (u57-62) Use of a conditional panel to create hidden reactive variables for event and interrupt control
- (s331-361) Hidden render instruction event. This event coordinates graph construction and rendering based on the state of on screen controls and the current graph context.
- (s363-374) Hidden reactive instruction event. Used primarily to overcome the problem of multiple, sequential function calls being ignored, except the final one.
- (u48, s424-428) Physics on/off event

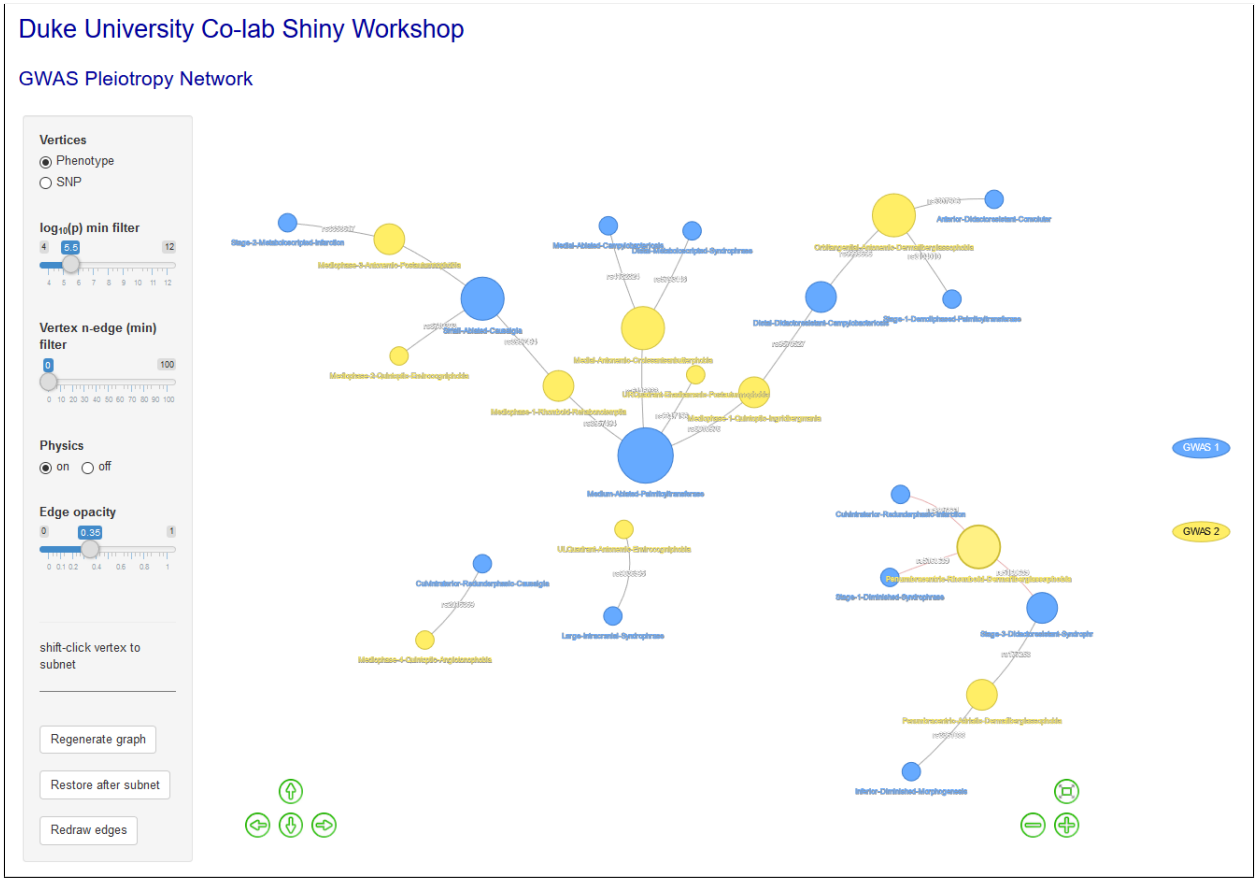

Figure 2: `visNetwork` pleiotropy app, version 2. Inter-GWAS phenotype as vertex, common SNP as edge. Additional Shiny control to filter vertices by number of edges. Subnetting available by clicking a vertex.

Figure 3 is an example screen-shot with SNP forming vertices and phenotype forming edges. The purpose of this view is to identify associations between SNPs that appear in both GWAS sets and the phenotypes that are associated with those SNPs within GWAS sets. Vertex hover labels indicate phenotype grouping and vertex color ranges from green-blue (high proportion of GWAS 1 phenotypes in group) to green (equal proportions form both GWAS sets) to green-yellow (high proportion of GWAS 2 phenotypes). Vertex size is proportional to the total number of phenotypes in the group. Vertex color (continuous from blue to yellow) indicates edge SNP proportion by GWAS set. App and script features for discussion (unnn indicates line number in ui.r, snnn indicates line number in server.r):

- (s140-158) Vertex construction
- (s148-158) Hover label construction
- (s162-185) Edge construction
- (s212-221) Vertex color definition

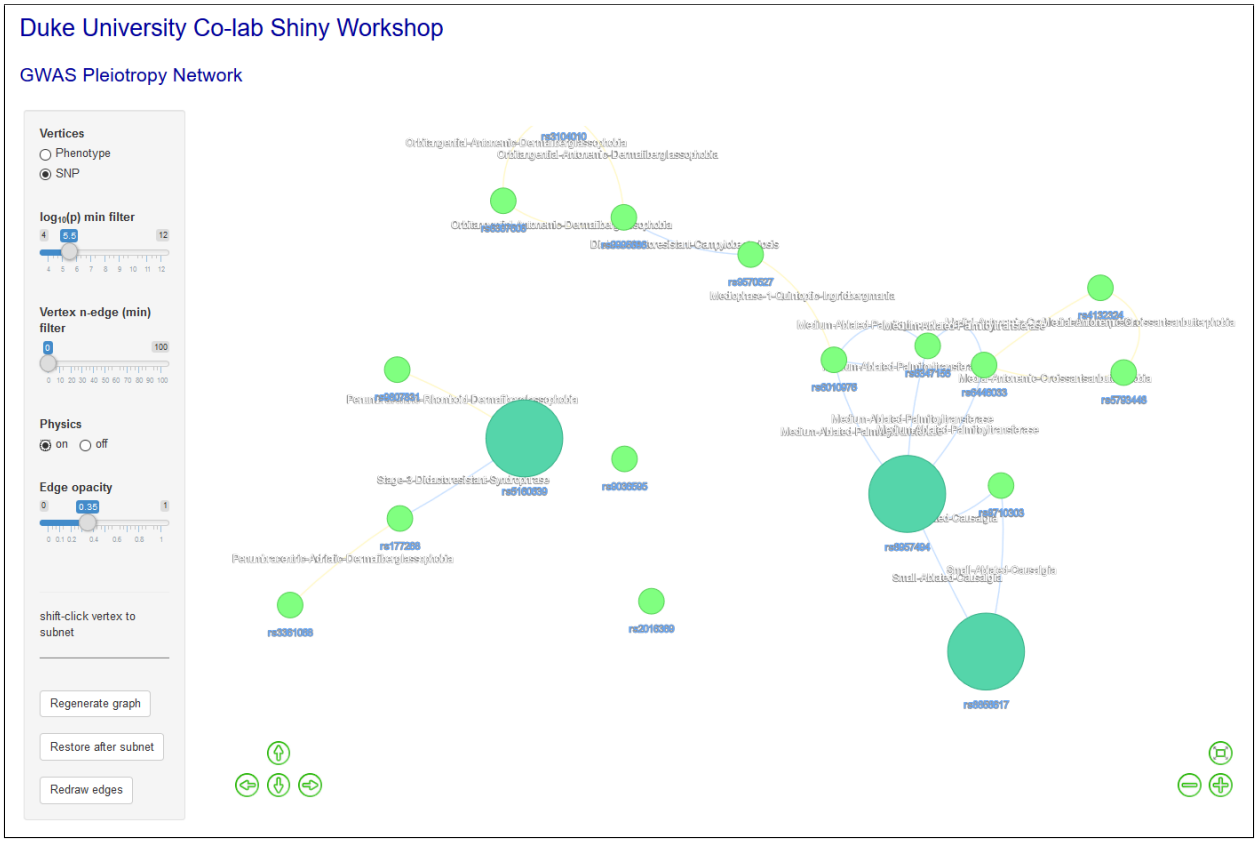

Figure 3: `visNetwork` pleiotropy app, version 2. SNP as vertex, individual edge for each phenotype. Additional Shiny control to filter vertices by number of edges. Subnetting available by clicking a vertex.

7.3 `visNetwork()` App Version 3, Additional Controls, Centrality Table

Workshop files `App/V3/ui.r` and `App/V3/server.r`

Figures 4 and 5 are example screen-shots of this app. The primary additional feature is the centrality table. Construction of it is accomplished in `server.r` lines 318-403.

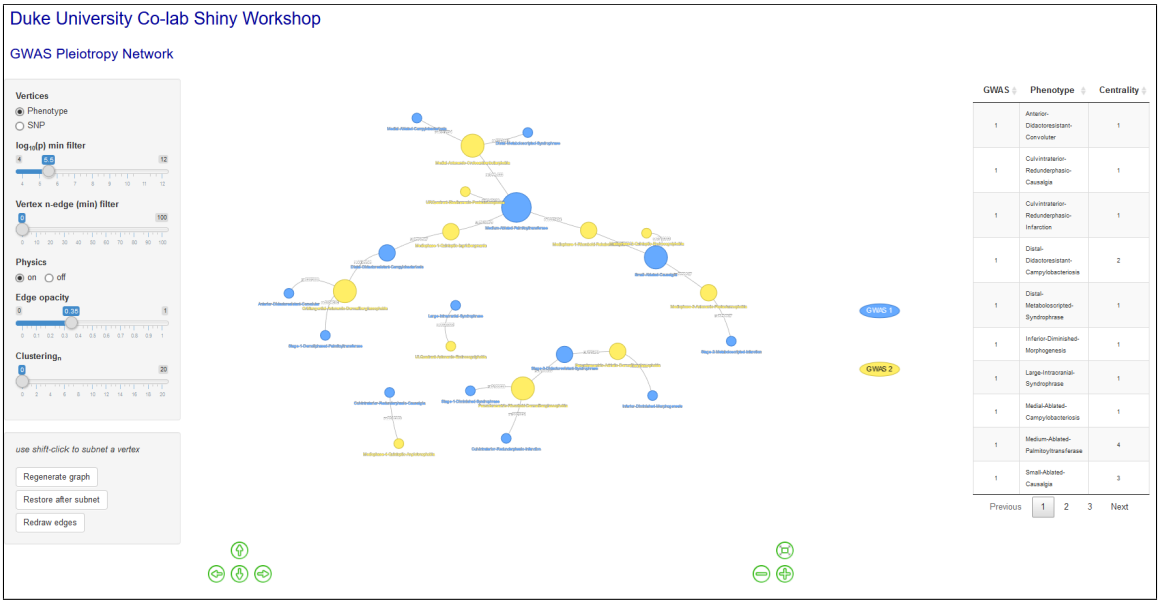

Figure 4: `visNetwork` pleiotropy app, version 3. Inter-GWAS phenotype as vertex, common SNP as edge. Additional Shiny control for clustering vertices. Centrality table included. Subnetting available by clicking a vertex.

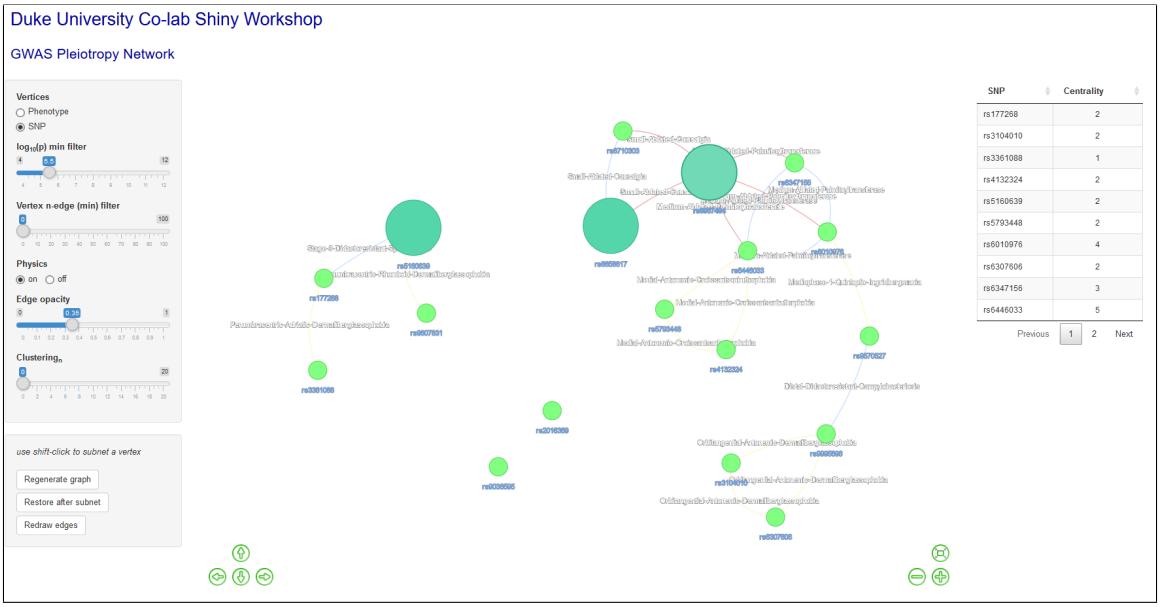

Figure 5: `visNetwork` pleiotropy app, version 3. SNP as vertex, individual edge for each phenotype. Additional Shiny control for clustering vertices. Centrality table included. Subnetting available by clicking a vertex.

8 Debugging

It is important that you have a means of communicating with your app during execution. Unlike a typical R script, that can be executed one line at a time, with interactive review of variables, once a Shiny script launches, it executes without the console prompt. Upon termination, some global variables may be available for examination, but you may not have reliable information on when they were last updated. Error and warning messages are displayed in the console (and the terminal session when executed in a shell) and, fortunately, so are the results of `print()` and `cat()`. When executed in RStudio, Shiny offers sophisticated debugger features (more info at https://shiny.rstudio.com/articles/debugging.html). However, one of the simplest methods of communicating with your app during execution is to use `print()` (for a formatted or multi-element object, such as a data frame) or `cat(, file=stderr())` for "small" objects. The `file=stderr()` causes displayed items to appear in red. Output may also be written to an error log, depending on your OS. Considerations include

- Shiny reports line numbers in error messages relative to the related function (`ui()` or `server()`) and, although not always exact, reported lines are usually in the proximity of the one which was executed at the time of error
- `cat("your message here")` displays in RStudio console (generally, consider Shiny Server)
- `cat("your message here", file=stderr())` is treated as an error (red in console, logged by OS)
- Messages appear in RStudio console when Shiny app launched from within RStudio
- Messages appear in terminal window when Shiny app launched with the `rscript` command in shell
- There exists a "showcase" mode (`runApp(display.mode="showcase")`) that is intended to highlight each line of your script as it is executing
- The reactivity log may be helpful in debugging reactive sequencing issues (`options(shiny.reactlog=T)`, https://shiny.rstudio.com/reference/shiny/0.14/showReactLog.html It may be helpful to initially format an apps appearance with an empty `server()` function, then include executable statements once the screen objects are available and configured
- Although not strictly related to debugging, the use of `gc()` to clear defunct memory (from R's recycling) may reduce total memory in use at a given time