Co-lab Shiny Workshop

Session 3, Integrating Shiny and `visNetwork`

April 21, 2021

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 relationships 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, while eliminating "noisy" relationships that are unrelated to findings of interest. 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 phenotypes (observed traits) in both studies that are correlated with genotypes of a common chromosomal position (these positions are known as single nucleotide ploymorphisms, or SNPs). The app will produce two styles of graph: one that generates two sets of phenotype vertices, one from each GWAS, and joins them by common SNP, and another that generates vertices from SNPs that are joined by phenotype. Implemented features include:

- Phenotype or SNP node filtering by strength of within-GWAS correlation (regression p-value)
- Node filtering by number of edges
- Selection of phenotype nodes with SNP edges or SNP nodes with phenotype edges
- `visnetWork` "physics" feature (places nodes using gravitation-like attraction based on mass, or node size)
- Edge transparency adjustment
- Subnetting of graph by node selection (retains all connected nodes)
- Node clustering (collapse of nodes having a specified number of edges into a single node)
- Display of a node centrality table

1 Overview

- Preliminaries
  - What can `Shiny and visNetwork` do for you?
  - What are your expectations of this workshop?
- Download Workshop Material and configure R
- Resources
- Anatomy of a Shiny App
- Reactivity
- Hello `visNetwork`
- `visnetwork` Pleiotropy App
  - Version 1, Basic Network Graph
  - Version 2, Enhanced Reactive Graph Controls
  - Version 3, Additional Graph Controls with Centrality Table

- Integrating `visNetwork` with a Graph Database for Visual Data Exploration
- Alternative Visualizations
- Debugging

2 Download Workshop Material and Configure R

- Copy course outline, scripts, and data from https://github.com/tbalmat/Duke-Co-lab/tree/master/Spring-2021/Session-4-visNetwork
  - App.zip
  - Co-lab-Session-4-visNetwork.pdf
  - Data.zip
- Expand zip files (one subdirectory per file)
- Launch RStudio
- Install packages:
  - `install.packages("shiny")`
  - `install.packages("shinythemes")`
  - `install.packages("visNetwork")`
  - `install.packages("DT")`

3 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: https://github.com/tbalmat/Duke-Co-lab/tree/master/Examples/GraphDeliveryMap
- Neo4j graph database: https://neo4j.com/developer/graph-database/

4 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/Spring-2021/Session-4-visNetwork

5 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

6 Reactivity

Reactivity is the single most important feature that Shiny offers. Variables are defined in your `ui()` function with an `input$` prefix and when these variables appear in `observe()` functions within in your `server()` function, execution events are triggered by on-screen changes to the corresponding `ui()` variables. In addition to referencing `input$` variables `observe()` functions include standard R commands, including those supported by any valid R package, so that the reactive variables become parameters to your R functions, enabling dynamic analysis of data. Output is rendered in the app by targeting `ui()` variables defined with an `output$` prefix. A simple example follows. It has a single, numeric input (`x`) and one plot output (`plot`). Changes in `x` cause the `observeEvent()` to be executed. The `observeEvent()` generates a histogram of `x` random, normal values. The histogram is a suitable input value to `renderPlot()`. Assignment of the `renderPlot()` result to `output$plot` causes the histogram to be displayed as defined in `ui()`. Notice how a Shiny input variable (`input$x`) is used as a parameter to an R function (`rnorm()`) and the result of an R function (`plot()`) is used as a parameter to a Shiny function (`renderPlot()`). Note that modifying `x` to its current value does not cause execution of the `observeEvent()` (try it).

```
library(shiny)

# Define UI
ui <- function(req) {
  fluidPage(
    numericInput(inputId="x", label="x", value=100),
    plotOutput(outputId="plot")
  )
}
```

```
# Define server function
server <- function(input, output, session) {
  observeEvent(input$x, {
    output$plot <- renderPlot(hist(rnorm(input$x)))
  })
}

# Execute
runApp(list("ui"=ui, "server"=server), launch.browser=T)
```

7 Hello `visNetwork`

The following instructions execute examples provided by the `visNetwork` package.

```
library(visNetwork)

# Generate three nodes
nodes <- data.frame(id = 1:3)
visNetwork(nodes)

# Create edges joining node 1 to 1 and node 2 to 3
edges <- data.frame(from = c(1,2), to = c(1,3))
visNetwork(nodes, edges)

# Introduction to visNetwork
vignette("Introduction-to-visNetwork") # with CRAN version

# Shiny visNetwork demo
shiny::runApp(system.file("shiny", package = "visNetwork"))

# java documentation
visDocumentation()
```

8 `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.

8.1 `visNetwork` App Version 1, Basic Network Graph

The version 1 graph implements the following features:

- Phenotypes as nodes, SNPs as edges
- Within-GWAS strength of association (phenotype to SNP) filtering
- visnetWork physics feature
- Edge transparency adjustment

To execute the app locally:

- Download App.zip and Data.zip from
  https://github.com/tbalmat/Duke-Co-lab/tree/master/Spring-2021/Session-4-visNetwork

- Expand App.zip and Data.zip in a local project directory
- Set the current working directory to App/V1
- Modify the directory reference (`setwd()` command) in App/V1/InteractivePleiotropyNetwork.r to reference the App/V1 directory on your computer
- Modify the directory reference (`setwd()` command) in App/V1/ui.r to reference the App/V1 directory on your computer
- Launch RStudio then load and execute App/V1/InteractivePleiotropyNetwork.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 include (unnn indicates line number in ui.r, snnn indicates line number in server.r):

- (s22-32) Data set description
- (u58) Specification of `visNetworkOutput`() object
- (s80-150) `assembleNetComponents()` function. This function generates the vertices and edges to be used in graph construction.
- (s113-121) Vertex data frame contents
- (s127-137) Edge data frame contents
- (s156-174) `composeNet()` function. This function produces, from previously assembled vertices and edges, the graph object to be rendered.
- (s158-163) `visNetwork()` parameters (groups, legend, options)
- (s165-168) Enabling/disabling of "physics"
- (s186-189) Reactive observer function that responds to changes in on-screen elements
- (s194) Call to `assembleNetComponents()`
- (s195) Rendering of graph with call to `composeNet()`

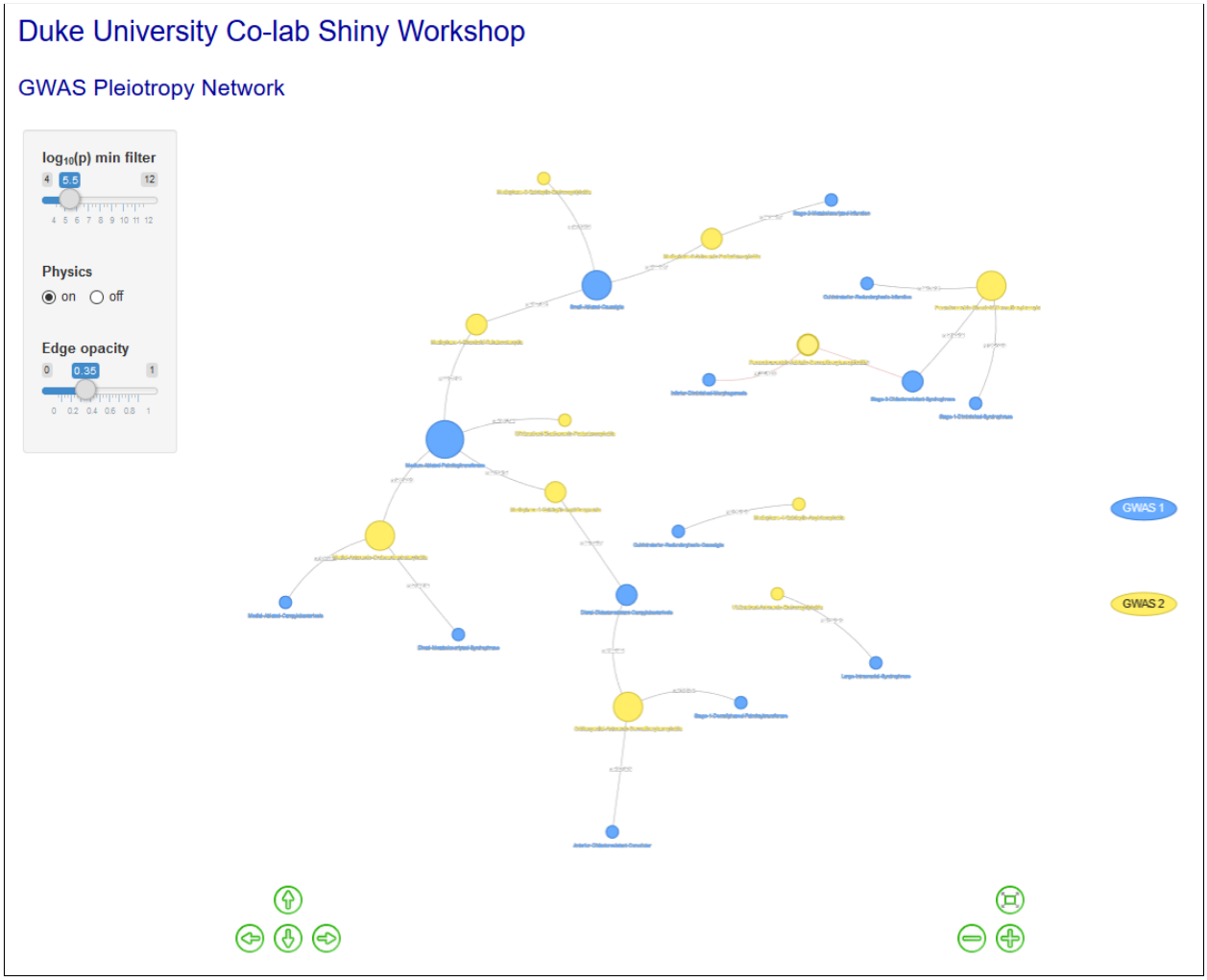

Figure 1: `visNetwork` pleiotropy app, version 1. Inter-GWAS phenotype as vertex, common SNP as edge. Basic Shiny features.

8.2 `visNetwork` App Version 2, Enhanced Graph Controls

Version 2 implements the following features:

- Phenotype or SNP vertex selection
- Node filtering by number of edges
- Subnetting graph to a selected node and all other nodes with shared edge

To execute the app locally:

- Download App.zip and Data.zip from https://github.com/tbalmat/Duke-Co-lab/tree/master/Spring-2021/Session-4-visNetwork
- Expand App.zip and Data.zip in a local project directory
- Set the current working directory to App/V2
- Modify the directory reference (`setwd()` command) in App/V2/InteractivePleiotropyNetwork.r to reference the App/V2 directory on your computer
- Modify the directory reference (`setwd()` command) in App/V2/ui.r to reference the App/V2 directory on your computer
- Launch RStudio then load and execute App/V2/InteractivePleiotropyNetwork.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 include (unnn indicates line number in ui.r, snnn indicates line number in server.r):

- (u44, s94) Vertex and edge reconfiguration from phenotype/SNP to SNP/phenotype
- (u58, s109-110) Vertex filtering by edge count
- (s114-115) Vertex hover label composition
- Use of global variables due to events and interrupt nature of execution
- (s292-295) Enabling of physics stabilization
- (s426-429) Physics stabilization event
- (s305-313) Enabling of shift-click event
- `ignoreInit`
- (s498-526) Graph subnetting with shift-click (on a vertex)
- (u66, s529-545) Restore after subnetting
- (u72-74) Use of a conditional panel to create hidden reactive variables for event and interrupt control
- (s340-368) Hidden render instruction event. This event coordinates graph construction and rendering based on the state of on screen controls and the current graph context.
- (s370-381) Hidden reactive instruction event. Used primarily to overcome the problem of multiple, sequential function calls being ignored, except the final one.
- (u60, s434-442) Physics on/off event

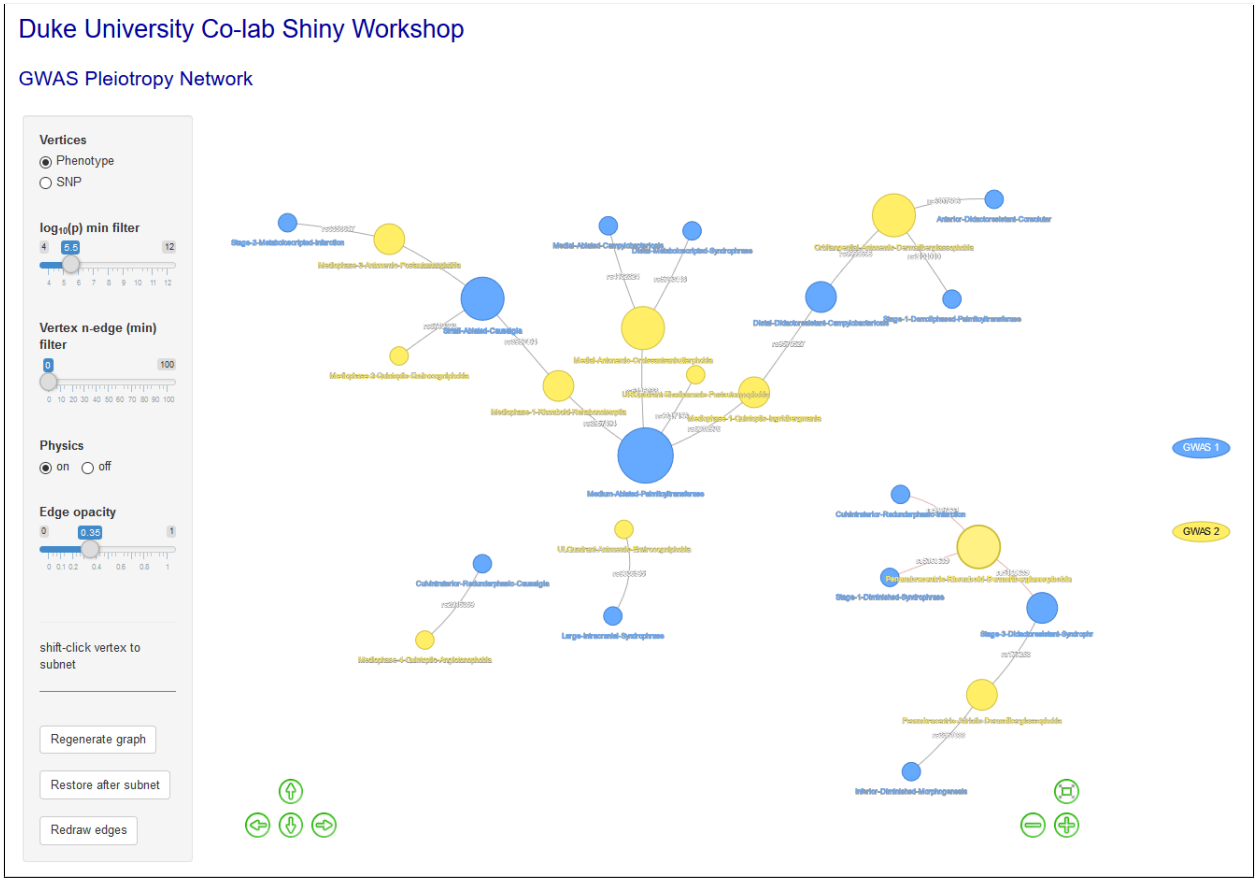

Figure 2: `visNetwork` pleiotropy app, version 2. Inter-GWAS phenotype as vertex, common SNP as edge. Additional Shiny control to filter vertices by number of edges. Subnetting available by clicking a vertex.

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

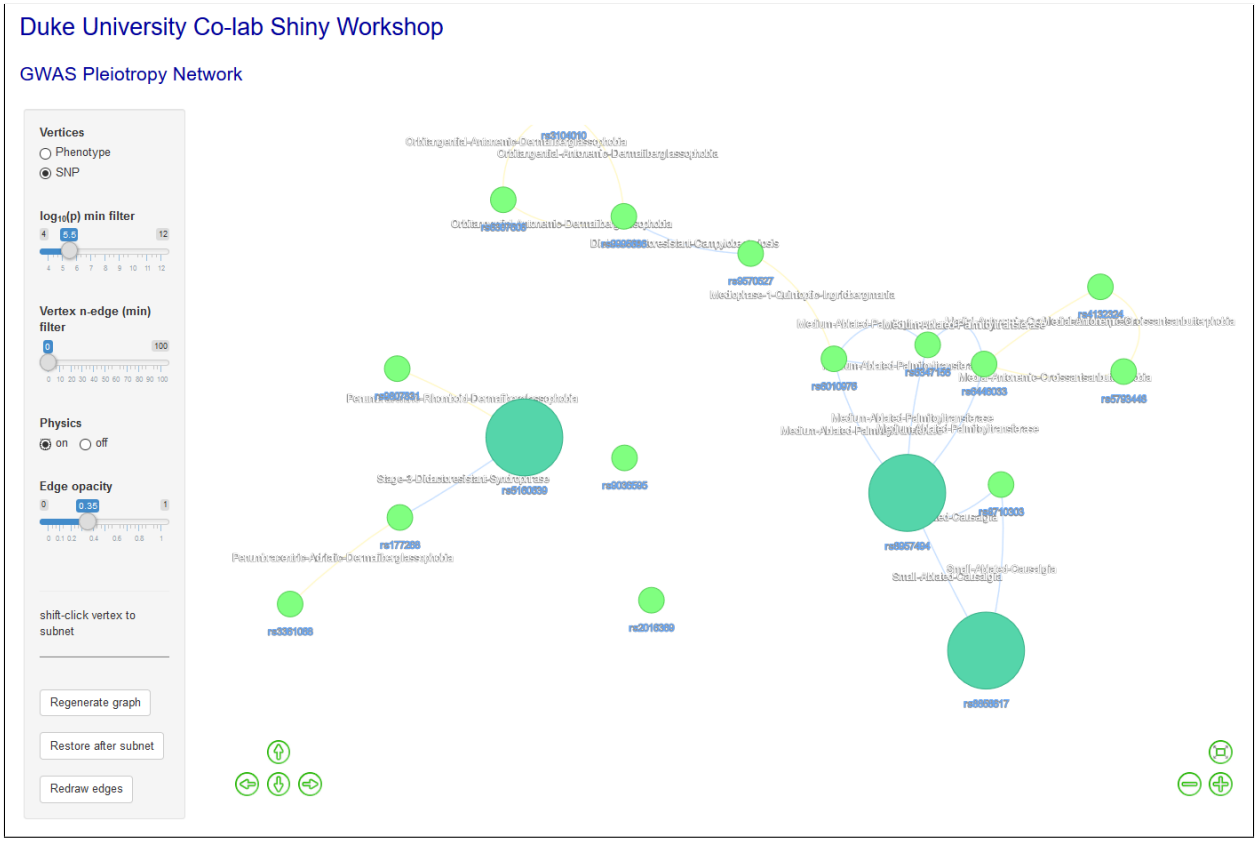

Figure 3: `visNetwork` pleiotropy app, version 2. SNP as vertex, individual edge for each phenotype. Additional Shiny control to filter vertices by number of edges. Subnetting available by clicking a vertex.

8.3 `visNetwork` App Version 3, Additional Controls, Centrality Table

Version 3 implements the following features:

- Node clustering (collapse of nodes having a specified number of edges into a single node)
- Display of a centrality table

To execute the app locally:

- Download App.zip and Data.zip from https://github.com/tbalmat/Duke-Co-lab/tree/master/Spring-2021/Session-4-visNetwork
- Expand App.zip and Data.zip in a local project directory
- Set the current working directory to App/V3
- Modify the directory reference (`setwd()` command) in App/V3/InteractivePleiotropyNetwork.r to reference the App/V3 directory on your computer
- Modify the directory reference (`setwd()` command) in App/V3/ui.r to reference the App/V3 directory on your computer
- Launch RStudio then load and execute App/V1/InteractivePleiotropyNetwork.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.

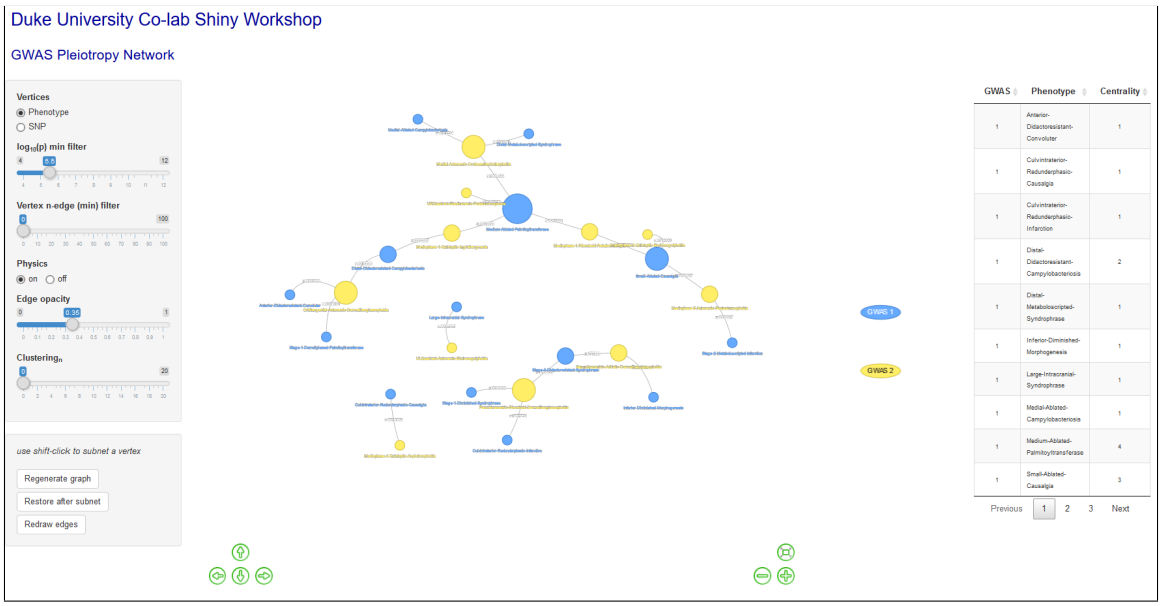

Figure 4: `visNetwork` pleiotropy app, version 3. Inter-GWAS phenotype as vertex, common SNP as edge. Additional Shiny control for clustering vertices. Centrality table included. Subnetting available by clicking a vertex.

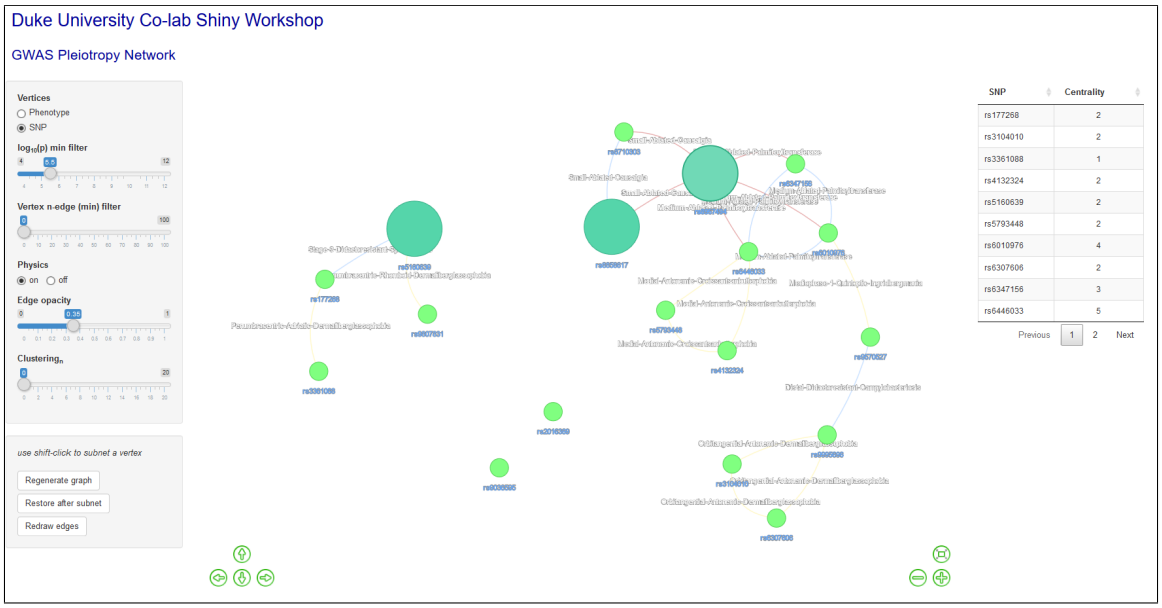

Figure 5: `visNetwork` pleiotropy app, version 3. SNP as vertex, individual edge for each phenotype. Additional Shiny control for clustering vertices. Centrality table included. Subnetting available by clicking a vertex.

9 Integrating `visNetwork` with a Graph Database for Visual Data Exploration

A Shiny app was developed, using `visNetwork`, for the Urea Cycle Disorder (UCD) Project in the Nursing School at Duke University.[1] [2] [3] Participants in the study have a specific, categorical, UCD diagnosis (UCDDx), are indicated as having or not had hyperammonemic (HA) events, and are coded for various clinically observed traits (tremors, cognitive disabilities, etc.). Clinical observations are coded using the Semantic Nomenclature for Medicine, Clinical Terminology (SNOMED CT).[4] Researchers use the app to explore study data and visually identify associations (by unique participant frequency) between UCDDx, HA, and SNOMED CT codes.[5] [6] An interesting feature of the project is its database. Instead of using SQL or an unstructured data format, all participant, diagnosis, SNOMED CT, and prescribed drug data along with their relational associations are represented in a Neo4J graph database.[7] [8] Important features of the app are the ability to select nodes for filtering or expansion. Filtering limits nodes to a select group, while expansion explores the subnet of selected nodes, allowing step-wise exploration, in increasing detail, of participant and medical diagnosis relationships as the researcher encounters them. The following demonstrates construction of a graph and use of node selection, filtering, and expansion operations.

9.1 SNOMED CT Concept Selection

Although multiple concepts can be selected, the example is limited to one, *Involuntary movement*. All immediate members, in the SNOMED CT hierarchy, of *Involuntary movement* will be displayed on the graph. Later, we will expand the *Tremor* sub-concept of *Involuntary movement* to explore this important clinical diagnosis. *Involuntary movement* appears in the SNOMED CT hierarchy under *Clinical finding*, *Clinical history and observation findings*, *Finding of movement*. Figure 6 Shows the concept selector (under "1. Explore concepts"). The selection hierarchy list follows that of the SNOMED CT browser. Once a concept is selected (under "2. Select concepts") the UCD Neo4J graph database is queried ("3. query").

[1] A recent paper, currently under review, that presents important findings in UCD research, is available at https://github.com/tbalmat/Duke-UCD/blob/master/Papers/ASL_PD_FinalAEEMBO.pdf

[2] Additional information on UCD is available at https://www.ncbi.nlm.nih.gov/books/NBK482363/?report=printable

[3] R, Shiny, and Neo4J script sources for the Duke app are available at https://github.com/tbalmat/Duke-UCD/tree/master/ShinyApp

[4] The SNOMED CT browser is available at https://browser.ihtsdotools.org/

[5] Actually, many other variables are explored for association, but the example here is limited to UCDDx, HA, and SNOMED CT code

[6] For an overview of using the app along with some example graphs, see https://github.com/tbalmat/Duke-UCD/blob/master/ShinyApp/Overview/UCDAppOverview01/UCDAppOverview01.pdf

[7] https://neo4j.com/

[8] For a discussion of encoding SNOMED CT concept paths in a Neo4J database, see https://www.sciencedirect.com/science/article/pii/S1532046415001847

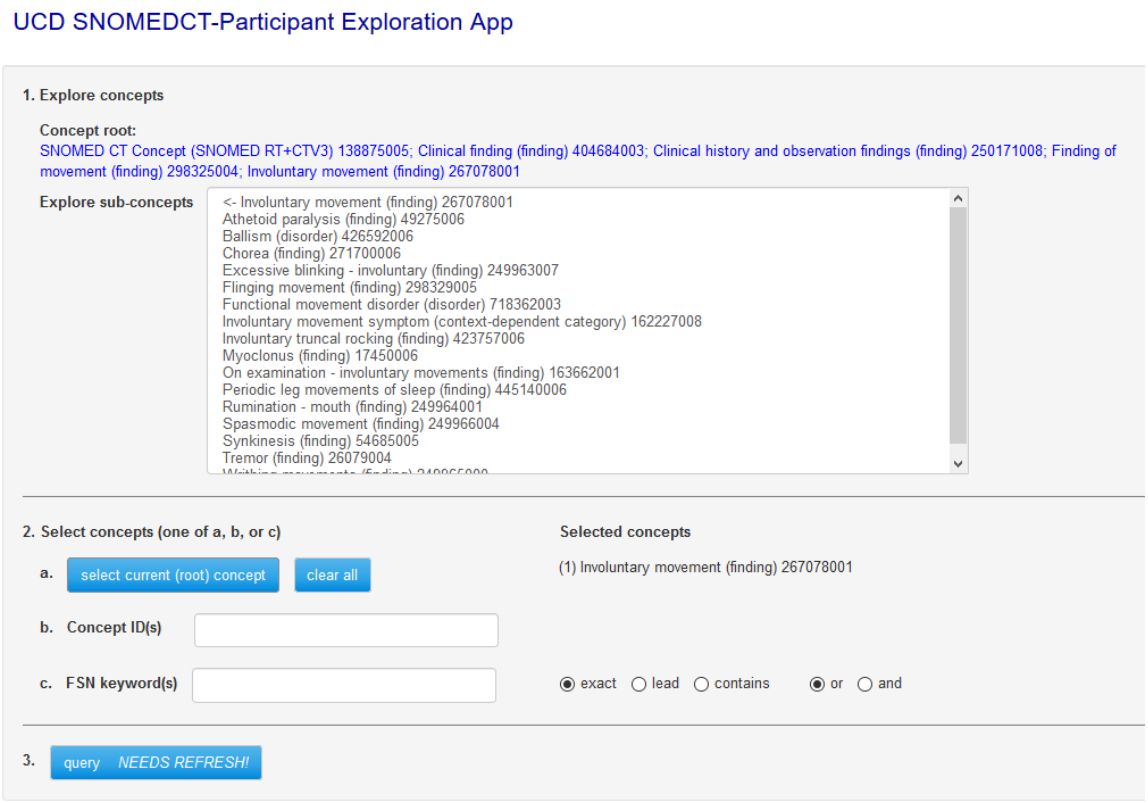

Figure 6: Duke University UCD app. Selection of SNOMED CT *Involuntary movement* concept.

9.2 Define Node and Edge Variables

Figure 7 shows the node and edge variable selection check boxes. Here, we have selected UCDDx, HA, and SNOMED CT concept as nodes, with edges between UCDDx and HA and between UCDDx and concept. Group-HA has also been specified causing the various HA database values to be classified as either HA or not HA. Figure 8 shows the resulting graph. Nodes are colored by variable and sized by number of unique participants assigned to the variable level associated with a node. Edges are sized according to the number of unique participants assigned both variable levels of nodes that are joined by a particular edge. Note that free-form selection of node and edge variables allows construction of a wide variety of graphs. Variable configuration can be changed after graph subsetting and node expansion to allow for exploration of further relationships as study data are visually probed in increasing detail.

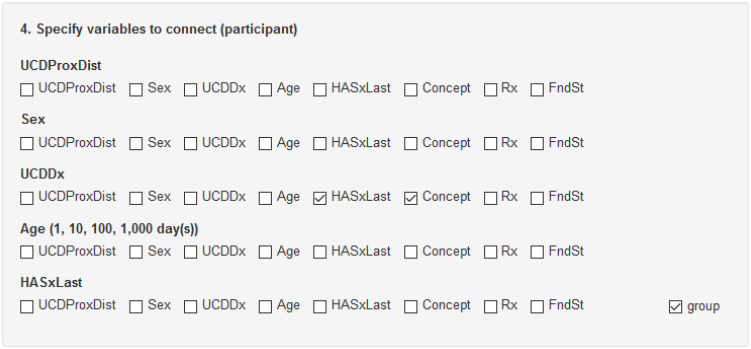

Figure 7: Duke University UCD app. Node and edge variable selection.

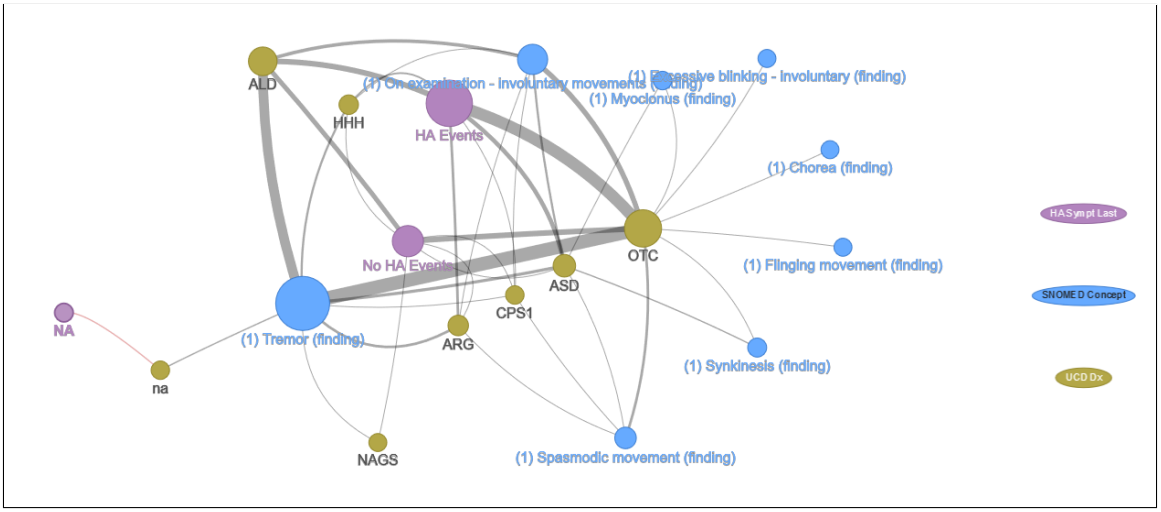

Figure 8: Duke University UCD app. Graph of UCDDx, HA, and *Involuntary movement* SNOMED CT concept.

9.3 Subset Graph to Isolate Variable Levels of Interest

Often, a researcher specializes in particular levels, or varieties, of variables within a system and would like to limit a graph to levels of interest. Figure 9 shows selection of desired nodes (depressing the shift key simultaneous with clicking the left mouse, or shift-click, selects and highlights nodes). The "subnet" button (figure 10, under "7. Explore") executes node subsetting and graph regeneration. Figure 11 shows the resulting graph. Nodes appearing without edges are for reference and indicate variable levels of nodes that shared an edge to a UCDDx node in the complete graph, but not of levels ALD or ASD.

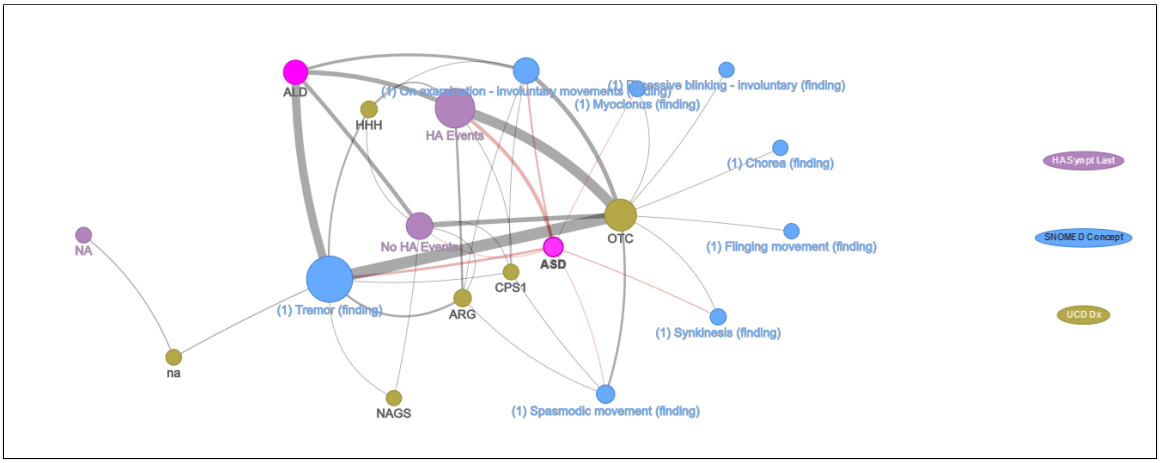

Figure 9: Duke University UCD app. Selection of UCDDx levels ALD and ASD for graph subsetting.

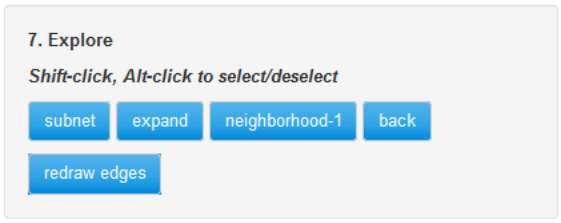

Figure 10: Duke University UCD app. The subnet button used to subset graph to selected nodes.

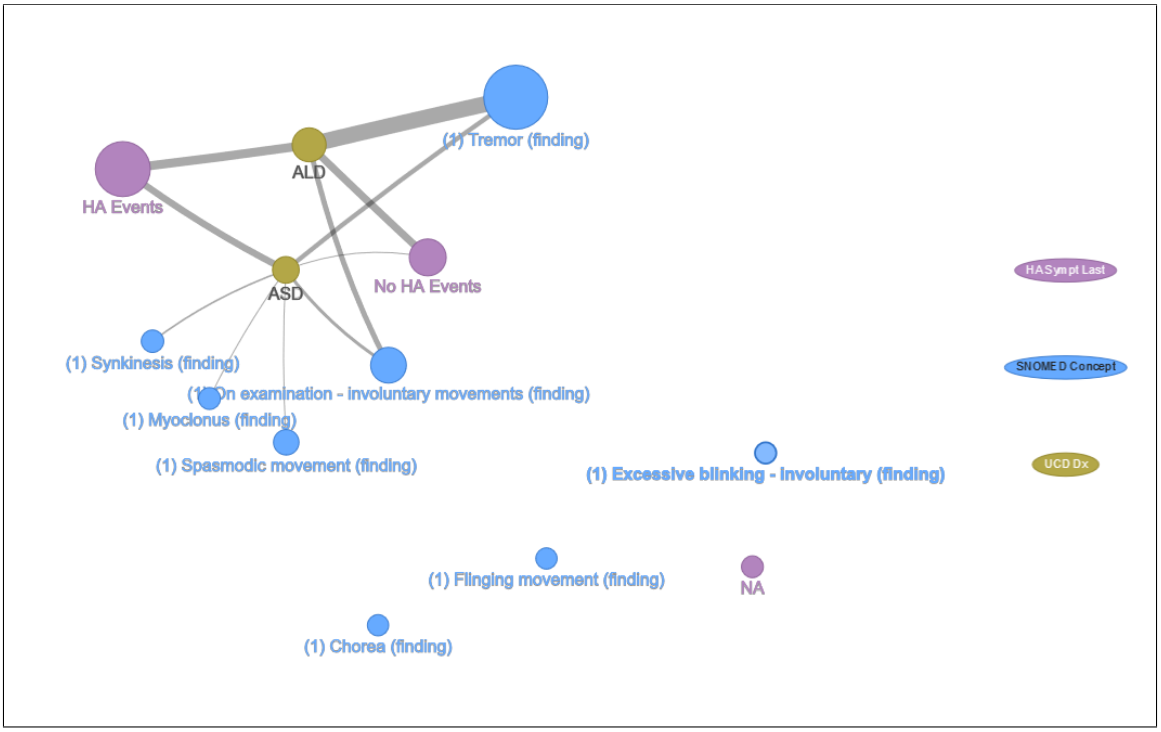

Figure 11: Duke University UCD app. Graph resulting from UCDDx ALD and ASD selection and subsetting.

9.4 Expansion of a Select SNOMED CT Concept Node

In the SNOMED CT hierarchy path explored here, the concept *Tremor* has multiple immediate concept members. To expand the *Tremor* node and present a graph of UCDDx (limited to ALD and ASD, since subsetting remains), HA, and *Tremor* concept members, we select the *Tremor* node (shift-click, figure 12) and click the expand button (figure 10). Note the label ("n=72") that appears when hovering over a node (figure 12). Here, the label indicates that 72 unique participants are assigned a SNOMED CT concept that is an immediate member of *Tremor* (when *Tremor* is selected at this place in the SNOMED CT hierarchy - it may appear in other hierarchical positions with a different number of unique participants assigned). Figure 13 shows the graph produced by expanding *Tremor*. Hover labels in figures 14 through 16 indicate 18 unique participants assigned *Intention tremor*, 17 unique participants assigned UCD diagnosis of ALD, and 12 unique participants assigned both *Intention tremor* and ALD, since corresponding nodes share an edge. Note that *Intention tremor* is associated with (shares an edge) with UCD diagnosis ALD, but not with ASD. This may be an important finding, or it may confirm researcher intuition based on clinical experience.

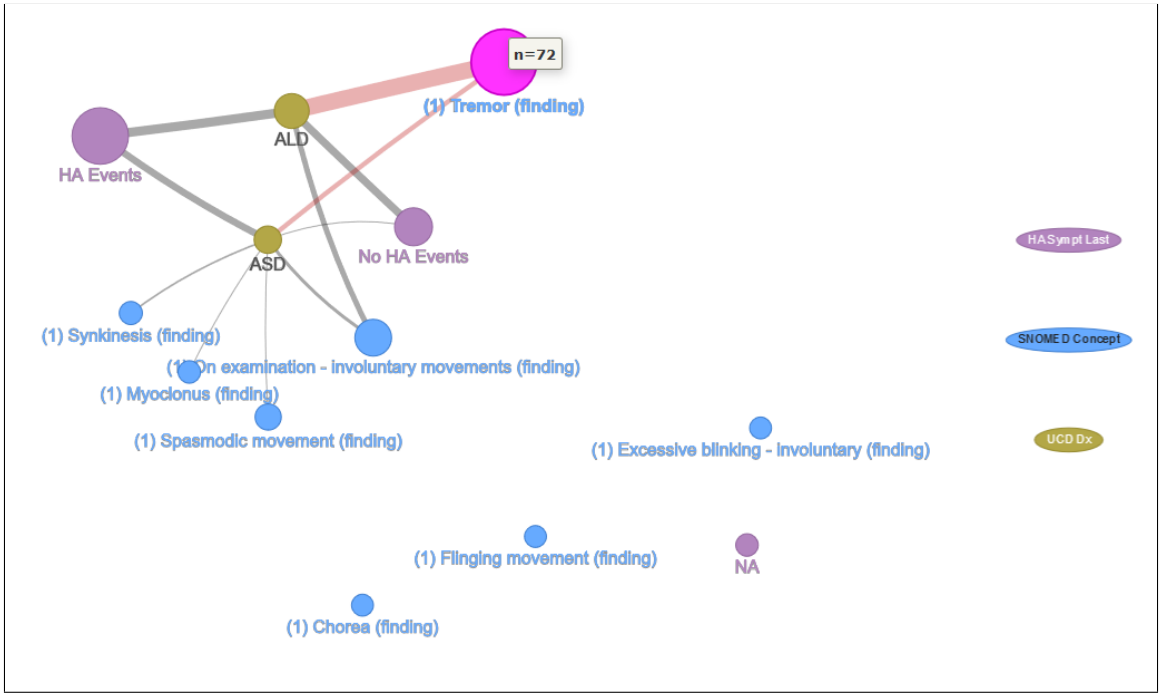

Figure 12: Duke University UCD app. Selection of *Tremor* concept node for expansion.

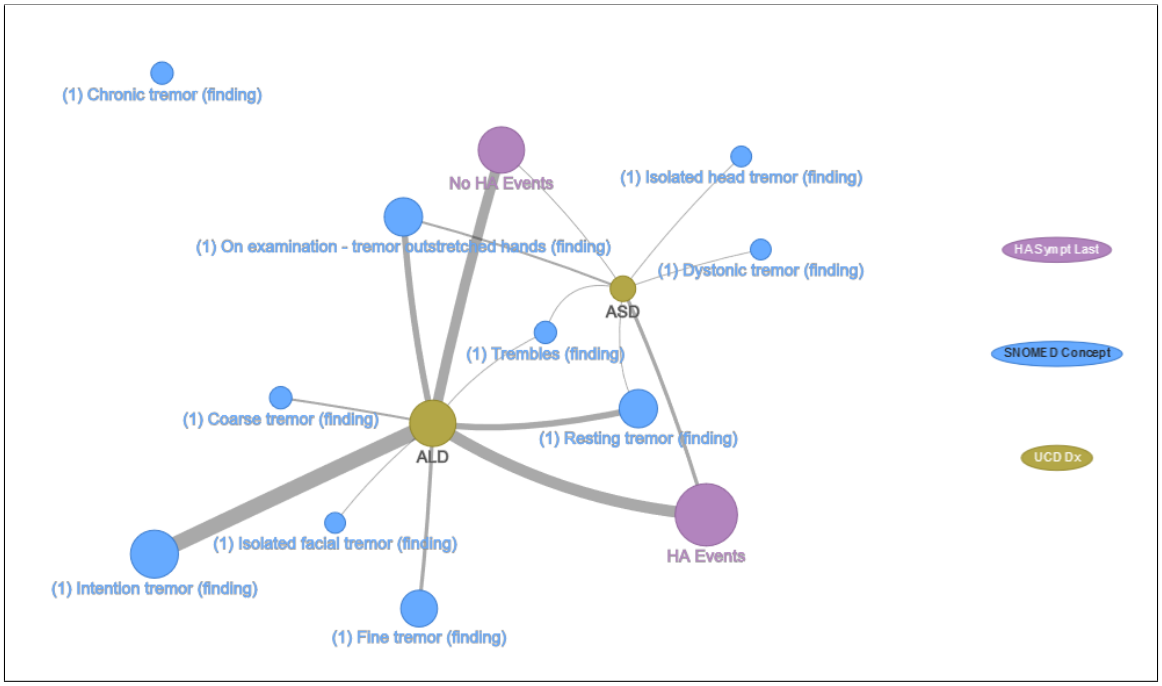

Figure 13: Duke University UCD app. Graph resulting from *Tremor* concept node expansion.

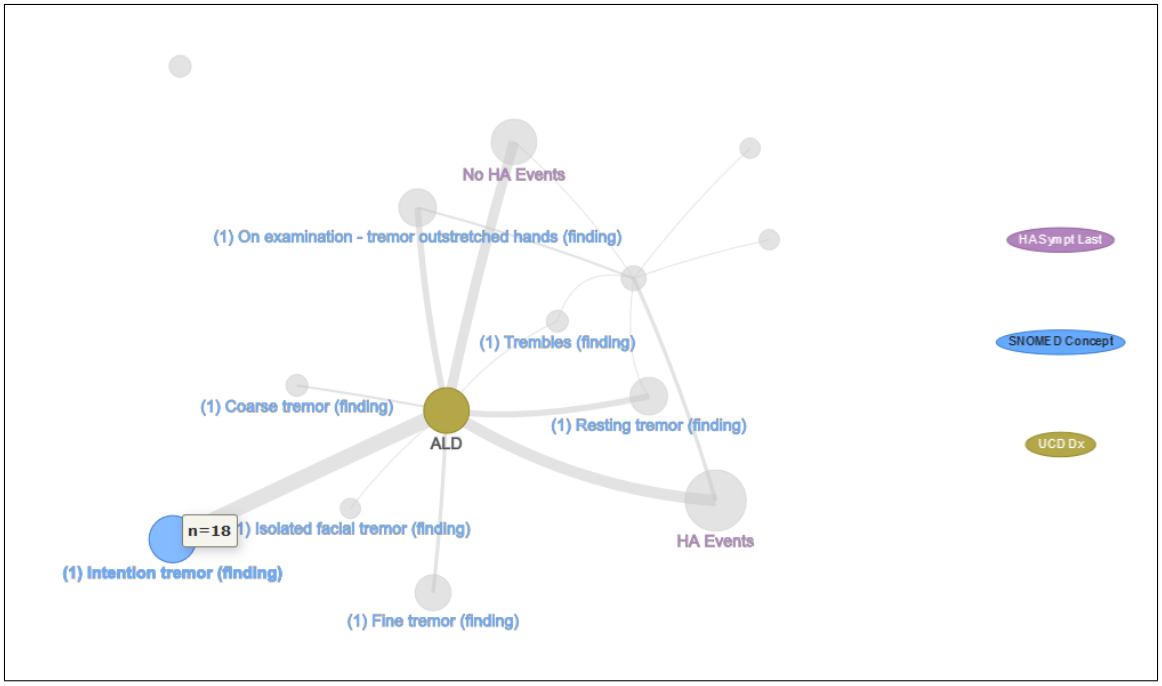

Figure 14: Duke University UCD app. Hover label indicating 18 unique participants assigned *Intention tremor* concept.

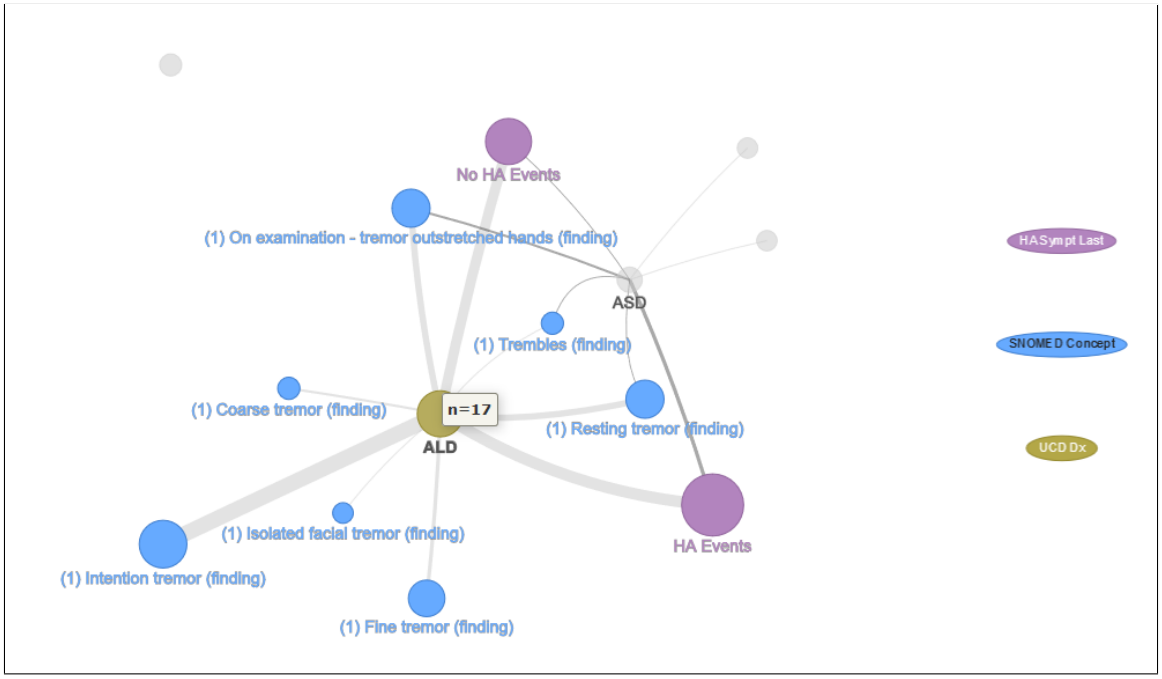

Figure 15: Duke University UCD app. Hover label indicating 17 unique participants assigned UCDDx level of ALD.

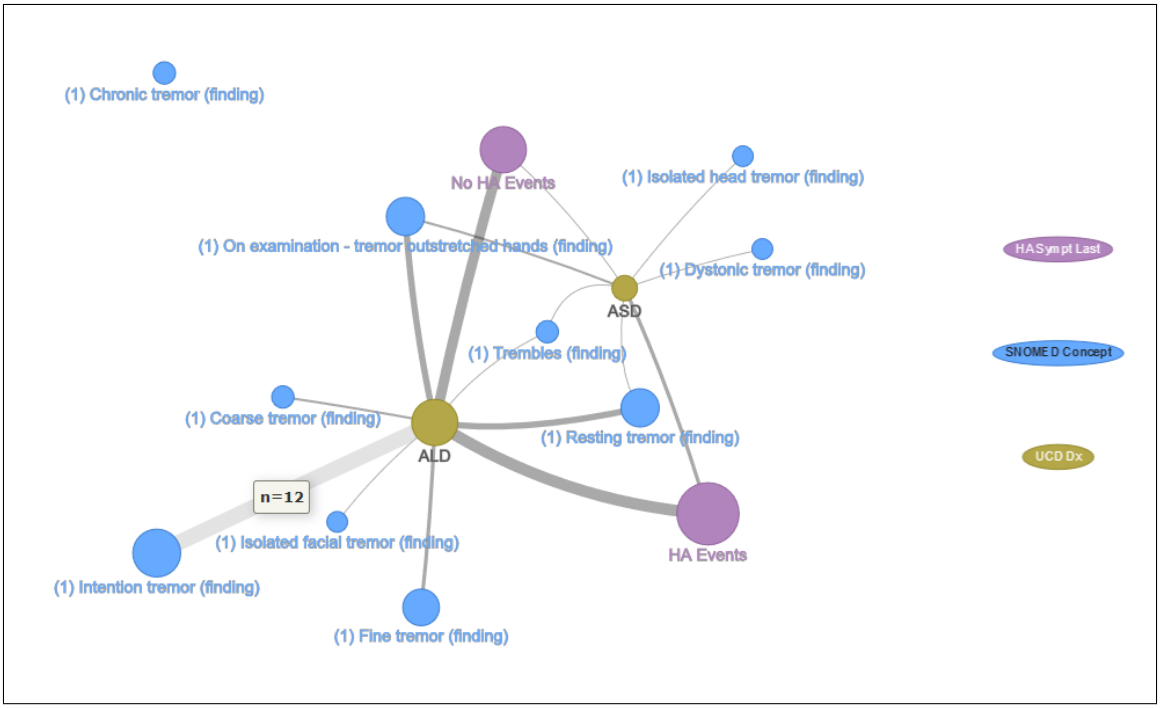

Figure 16: Duke University UCD app. Hover label indicating 12 unique participants assigned (sharing an edge) both *Intention tremor* and UCDDx level of ALD.

9.5 Radial Graph

An alternative to the free-form graphs presented so far is a radial presentation. Figure 17 shows selection of "radial" on the alternative geometry configurator and figure 18 shows the resulting graph (nodes and edges are identical to those in figure 13). Note that concept and UCDDx imbalance are indicated by concepts with a single UCDDx edge, which is, perhaps, more visible in the radial than in the free-form graph.

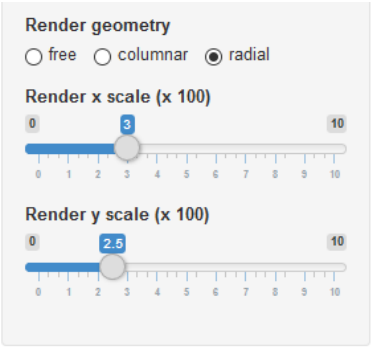

Figure 17: Duke University UCD app. Radial graph layout specification.

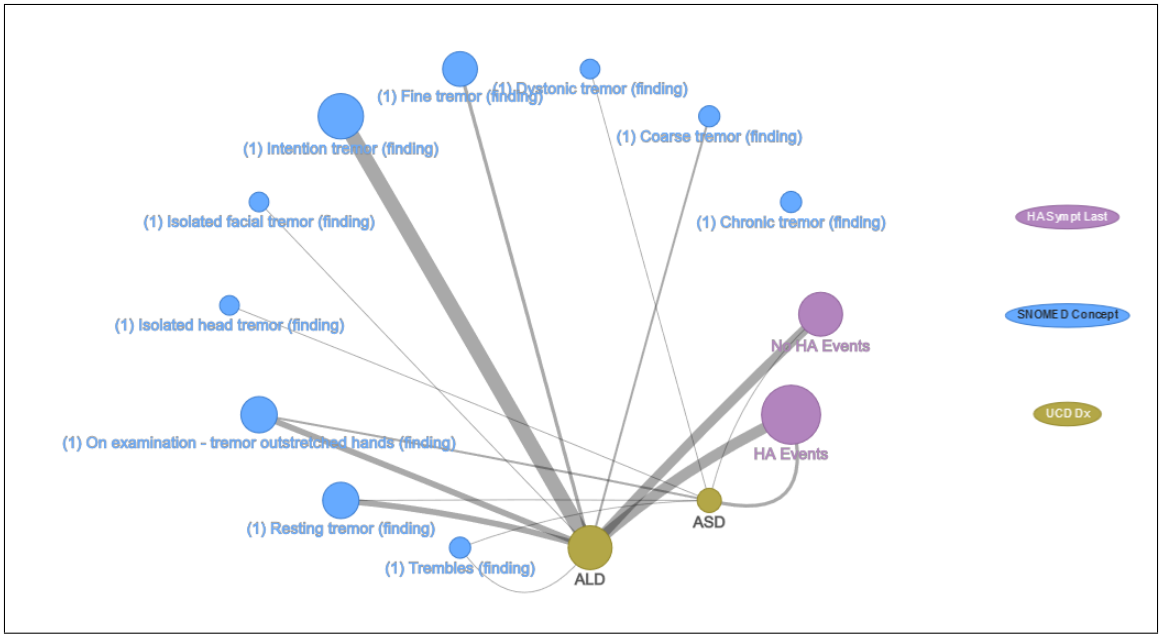

Figure 18: Duke University UCD app. Radial graph layout.

9.6 Interactions

Graphs presented so far assume that relationships between levels of two variables hold regardless of the level of a third variable. However, given our data, it may be the case that relationships between, say UCDDx and HA, depend on the level of SNOMED CT concept. For instance, if UCDDx of ALD is associated with concept *Trembles* only when HA is positive, we want our graph to indicate this interaction with corresponding edges. Figure 19 shows interaction controls. Figures 20 and 21 show the resulting graph in free-form and columnar (bipartite) formats, respectively.

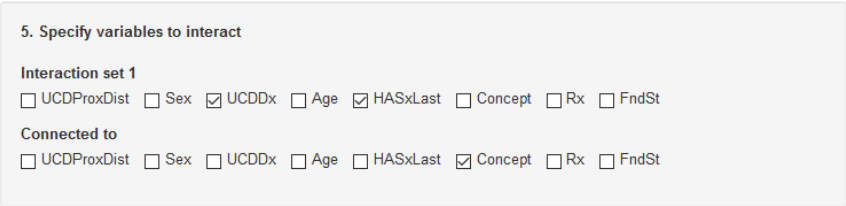

Figure 19: Duke University UCD app. Specifying interactions.

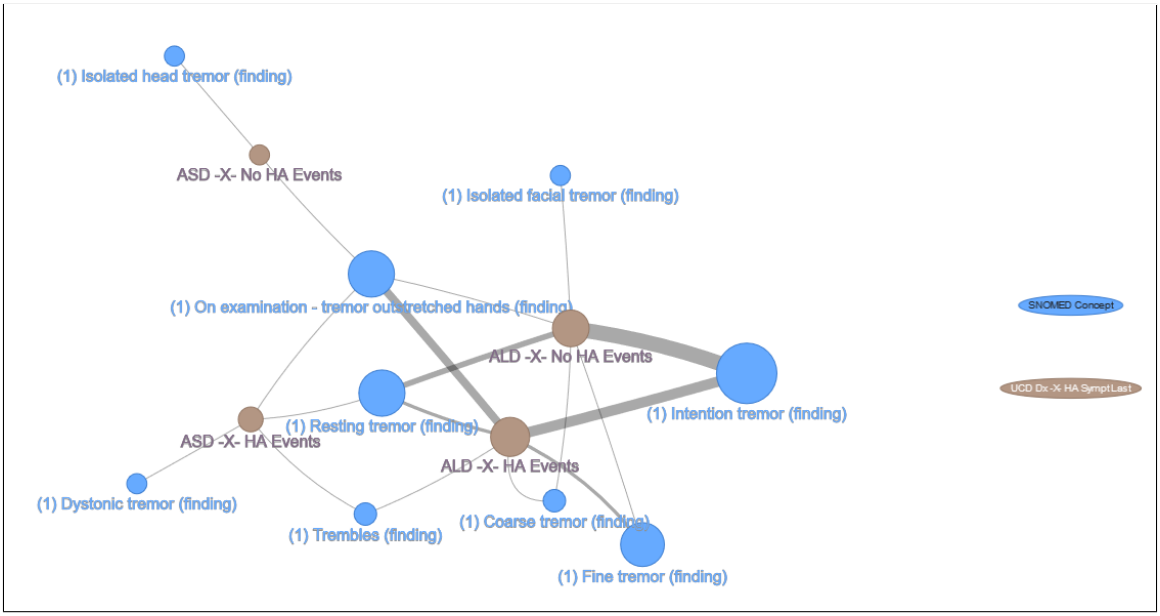

Figure 20: Duke University UCD app. Interaction graph, free-form.

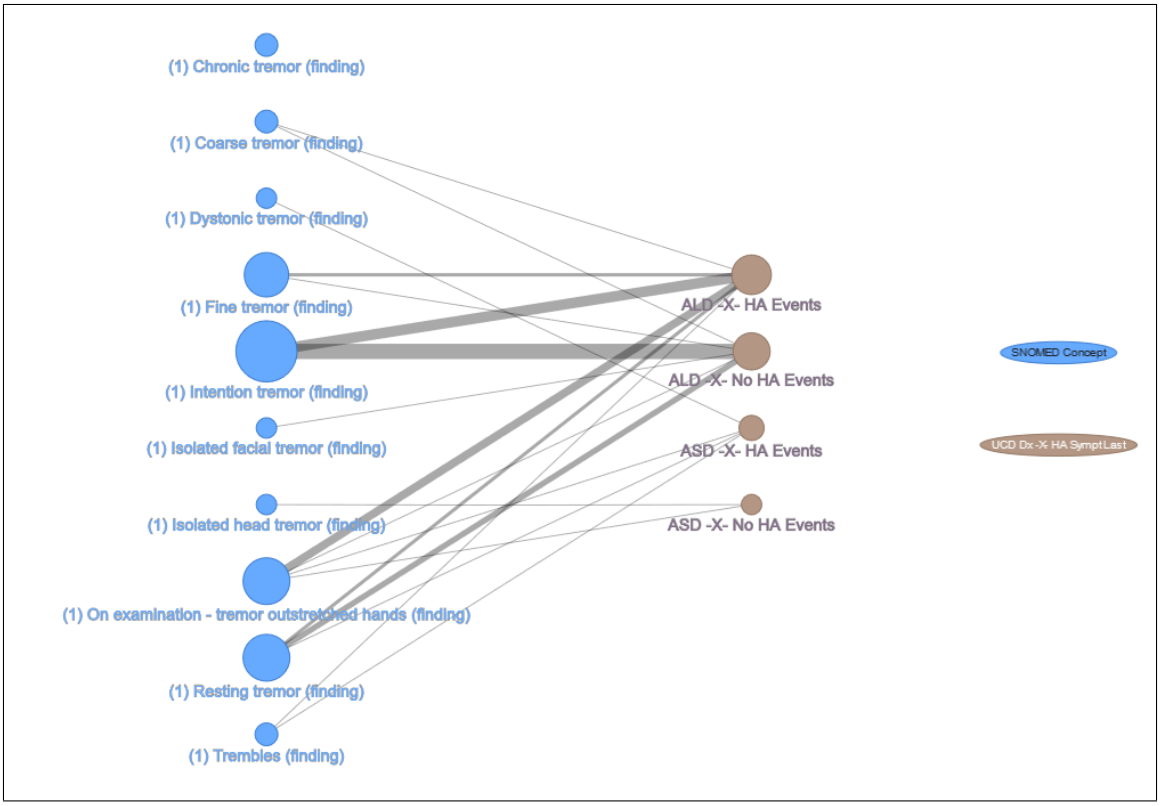

Figure 21: Duke University UCD app. Interaction graph, bipartite.

10 Alternative Visualizations

Script location: https://github.com/tbalmat/Duke-UCD/blob/master/ShinyApp/AlternativeVisualization/UCDEdgeBundle.r

The following instructions use the `ggraph()` function of the `ggraph` package to produce various hierarchical and edge bundled graphs that may serve as alternatives to network graphs.[9] Each set of instructions is executed after querying the UCD graph database mentioned in section 9. Variables in the queried result set are participant, the interaction between UCD diagnosis and hyperammonemia classification, SNOMED concept, and prescribed medication.

10.1 Create Hierarchy, Vertices, and Initial Graph

```
##############################################################################################
# Create hierarchy
# The dendrogram has an origin for groups, an origin for each group, and leaves (try
# ggraph with circular=F)
# From and to, here, instruct from which graph origin a line originates and where it
# terminates
# The groups originate at the origin and termnate at a group
# The leaves originate at a group and terminate in a leaf (the unique label within the
# data for a group)
##############################################################################################

h <- rbind(data.frame("from"="origin", "to"="origin",
                      "n"=length(unique(y[,"participantID"]))),
           data.frame("from"="origin", "to"=v,
                      "n"=length(unique(y[,"participantID"]))),
do.call(rbind, lapply(v,
  function(v)
    setNames(aggregate(1:nrow(y), by=list(rep(v, nrow(y)), y[,v]),
               function(k) length(unique(y[k,"participantID"]))),
             c("from", "to", "n")))))

##############################################################################################
# Create vertices
##############################################################################################

vertex <- setNames(h[,c("to", "from", "n")], c("name", "v", "n"))

##############################################################################################
# Create igraph hierarchical dendrogram object
##############################################################################################

g <- graph_from_data_frame(h, vertices=vertex)
```

10.2 Hierarchical Dendrogram Edges Indication Pathways Between Node Types

Script:

```
# Review plot and group origins
ggraph(g, layout='dendrogram', circular=F) +
  geom_edge_diagonal() +
  theme_void()
```

Result:

[9] For more information on the `ggraph` package, see https://ggraph.data-imaginist.com/reference/index.html

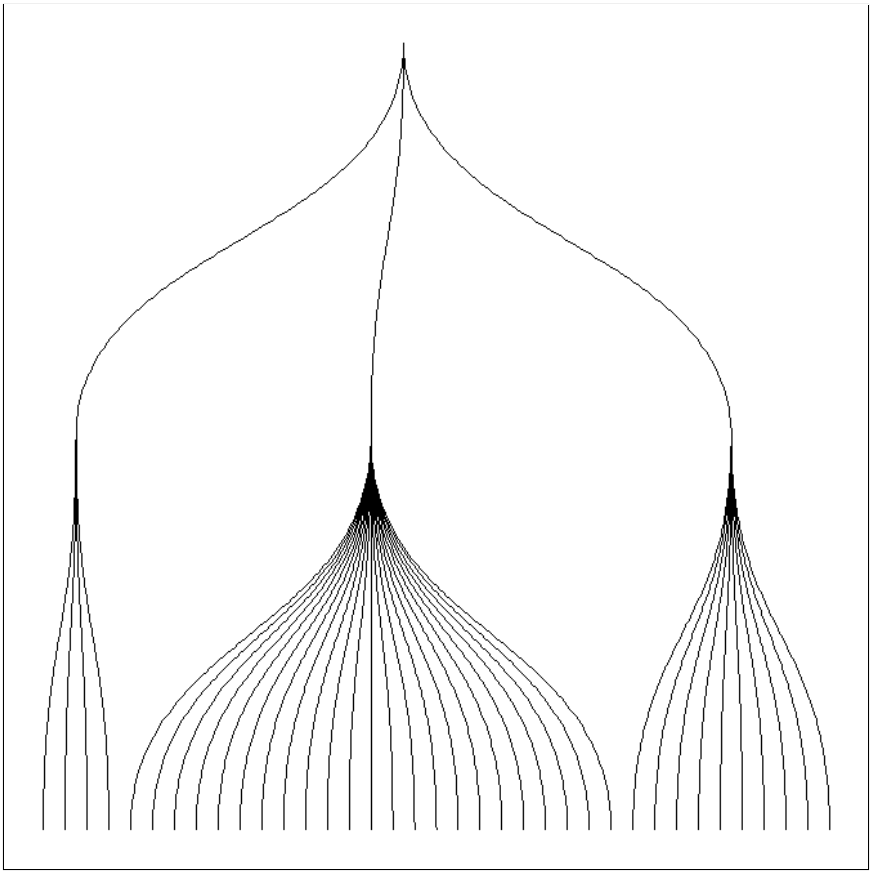

Figure 22: Hierarchical UCD dendrogram showing pathways between node types

10.3 Hierarchical Dendrogram Edges and Nodes

Script:

```
# Add points
ggraph(g, layout='dendrogram', circular=F) +
  # x, y coordinates are generated by ggraph() (use str(ggraph(...)) to review elements)
  # n is taken from the vertex data frame used to construct g
  geom_node_point(aes(x=x, y=y, size=n)) +
  geom_edge_diagonal() +
  theme_void()
```

Result:

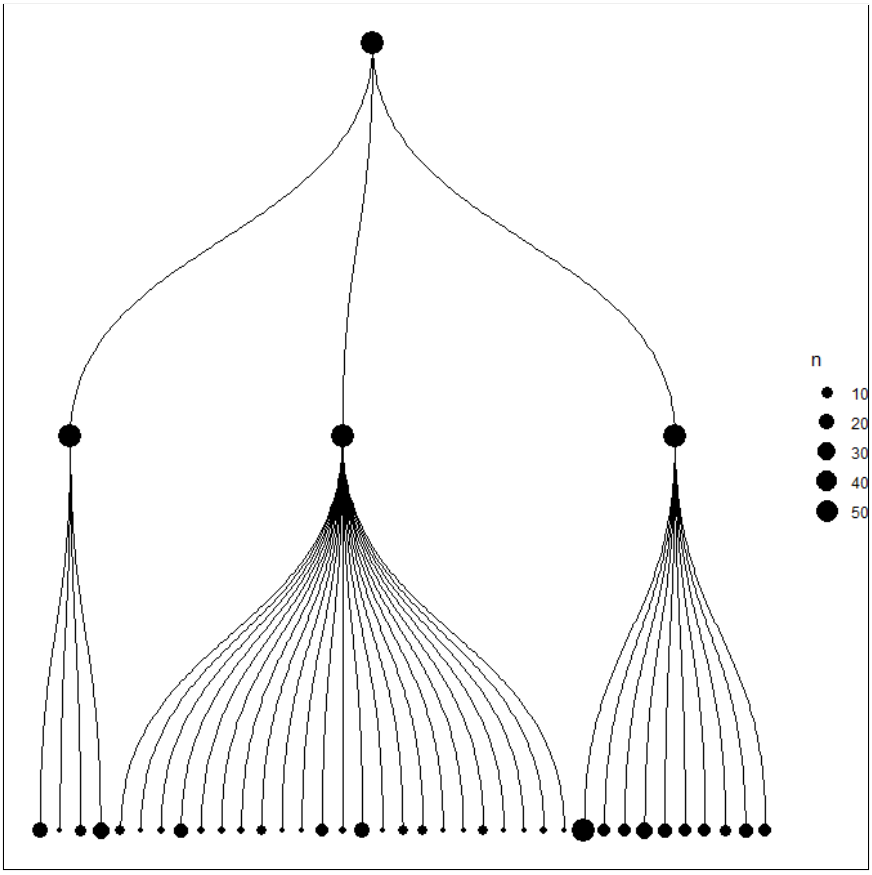

Figure 23: Hierarchical UCD dendrogram with edges and nodes

10.4 Hierarchical Dendrogram Edges Indicing Unique Participant Frequency by Node Type

Script:

```
# Add points
ggraph(g, layout='dendrogram', circular=F) +
  # x, y coordinates are generated by ggraph() (use str(ggraph(...)) to review elements)
  # n is taken from the vertex data frame used to construct g
  geom_node_point(aes(x=x, y=y, size=n)) +
  geom_edge_diagonal() +
  theme_void()

# Add labels
ggraph(g, layout='dendrogram', circular=F) +
  # Coordinates:
  # Plot extents are (0,0) to (n_leaves-1,n_levels-1), where n_levels is the number of levels in th
  # hierarchy (one for origin to origin, one for the groups with from="origin," and one for leaves)
  # leaves are placed at x=0..n_leaves-1, y=0
  # x, y coordinates are generated by ggraph() (use str(ggraph(...)) to review elements)
  # v and n are taken from the vertex data frame used to construct g
  geom_node_point(aes(x=x, y=y, size=n,
    color=ifelse(v!="origin", v, ifelse(name!="origin", name, "Participants")))) +
  geom_node_text(aes(x=ifelse(v!="origin", x, x+1),
    y=ifelse(v!="origin", y-0.2, y),
    label=ifelse(name!="origin", name, "Participant"), #paste("(", x, ",", y, ")", sep=""),
    color=ifelse(v!="origin", v, ifelse(name!="origin", name, "Participants")),
    angle=ifelse(v!="origin", 90, 0),
```

```
  vjust=0.5,
  hjust=ifelse(v!="origin", 1, 0),
  # Note that node and text sizes are from disjoint spaces, but ggplot does not
  # support multiple scale_size functions
  # Therefore, scale text the range of point sizes, which are based on participant
  # counts
  size=ifelse(v!="origin", 10, ifelse(name!="origin", 20, 30)))) +
scale_color_manual(values=setNames(c("black",
  rgb(matrix(col2rgb("#FFEE66"), nrow=1)/390),
  rgb(matrix(col2rgb("#66AAFF"), nrow=1)/300),
  rgb(matrix(col2rgb("#88AAAA"), nrow=1)/280)), c("Participants", v)),
  guide=F) +
#scale_size_manual(values=c("t1"=3, "t2"=4, "t3"=5), guide=F) +
geom_edge_diagonal() +
expand_limits(x=c(NA, length(which(h[,"from"]!="origin"))+5), y=c(-2.5, NA)) +
theme_void() + theme(plot.margin=margin(0.25, 0.25, 0.25, 0.25, "in"))
```

Result (n indicates unique participant frequency):

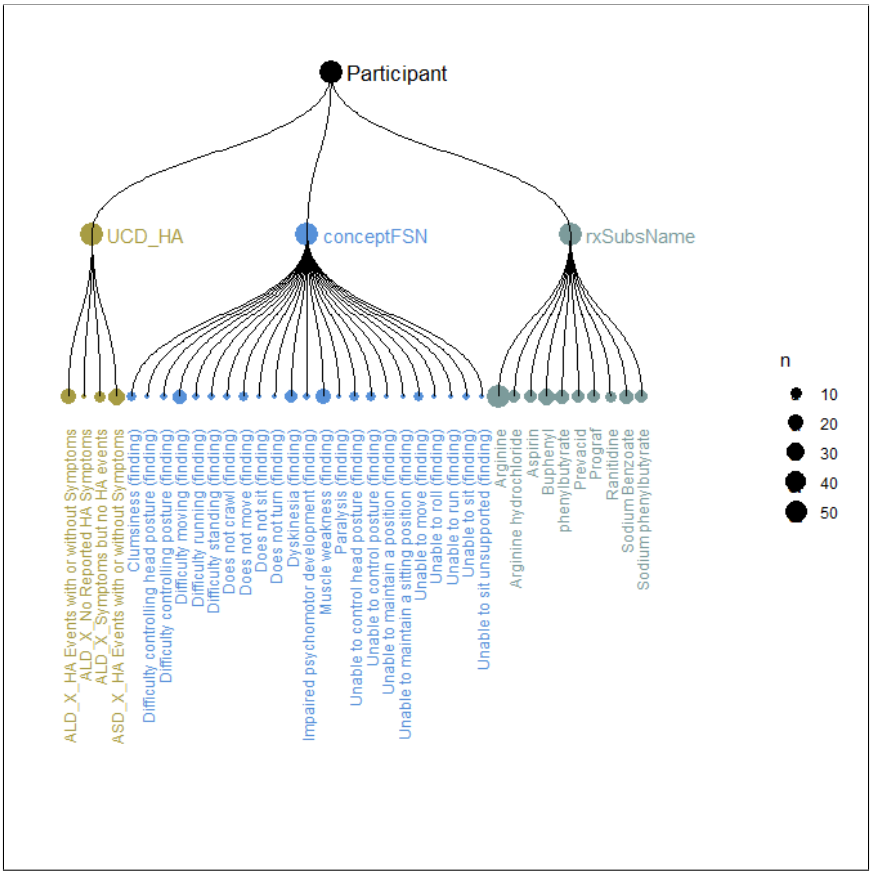

Figure 24: Hierarchical UCD dendrogram with nodes sized to indicate unique participant frequency

10.5 Radial Edge Bundled Graph (points and edges)

Script:

```
# Include points (nodes)
ggraph(g, layout='dendrogram', circular=T) +
  # x, y coordinates are generated by ggraph() (use str(ggraph(...)) to review elements)
```

```
# Filter=leaf omits points for graph and group origins
geom_node_point(aes(x=x*1.05, y=y*1.05, filter=leaf)) +
# from and to are taken from the edge data frame
geom_conn_bundle(data=get_con(from=edge[,"from"], to=edge[,"to"]),
color="blue", alpha=0.1, tension=0.75) +
theme_void()
```

Result:

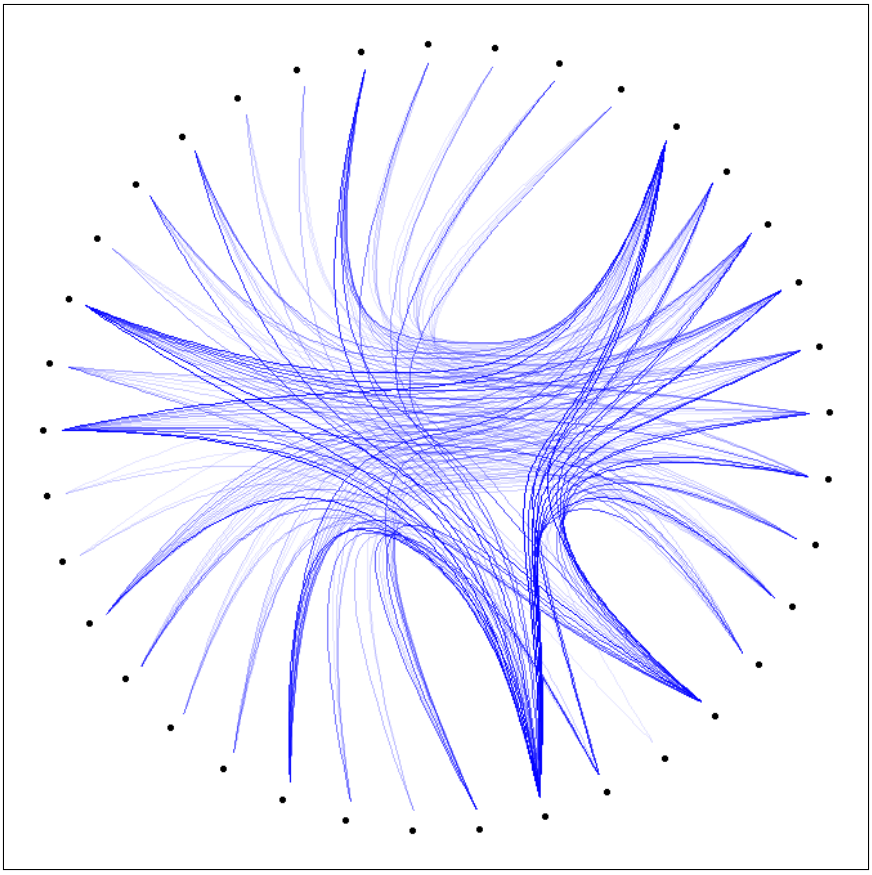

Figure 25: UCD radial edge bundled graph, nodes and edges only

10.6 Radial Edge Bundled Graph with Edges, Nodes Sized for n, and Node Labels

Script:

```
# Draw points, labels, and edges, sized and colored
xyfactor <- 3
glim <- c(-10, 10)
tsize <- 3.5
ecolor <- setNames(c(rgb(matrix((col2rgb("#FFEE66")+col2rgb("#66AAFF"))/2, nrow=1)/300),
rgb(matrix((col2rgb("#FFEE66")+col2rgb("#88AAAA"))/2, nrow=1)/300),
rgb(matrix((col2rgb("#66AAFF")+col2rgb("#88AAAA"))/2, nrow=1)/300)),
apply(t(combn(1:3, 2)), 1, function(j) paste(v[j[1]], "_", v[j[2]], sep="")))

ggraph(g, layout='dendrogram', circular=T) +
  # Draw arcs first, so that points overlay ends
  # x, y coordinates are generated by ggraph() (use str(ggraph(...)) to review elements)
  # from and to are taken from the edge data frame
  geom_conn_bundle(data=get_con(from=edge[,"from"], to=edge[,"to"], v12=edge[,"v12"]),
```

```
    aes(x=x*xyfactor, y=y*xyfactor, color=v12), edge_width=0.35, alpha=0.1, tension=0.75) +
  scale_edge_color_manual(values=ecolor) +
  # v and n are taken from the vertex data frame used to construct g
  geom_node_point(aes(filter=leaf, x=x*xyfactor, y=y*xyfactor, size=n, color=v), alpha=0.8) +
  scale_size_continuous(range=c(3, 10)) +
  geom_node_text(aes(x=xyfactor*1.2*x, y=xyfactor*1.2*y, filter=leaf, label=name, color=v,
    angle=ifelse((node_angle(x, y)+270)%%360 > 180, 0, 180)+node_angle(x, y),
    vjust=ifelse((node_angle(x, y)+270)%%360 > 180, 0.25, 0.5)), size=tsize, hjust="outward") +
  scale_color_manual(values=setNames(c(rgb(matrix(col2rgb("#FFEE66"), nrow=1)/390),
    rgb(matrix(col2rgb("#66AAFF"), nrow=1)/300),
    rgb(matrix(col2rgb("#88AAAA"), nrow=1)/280)), v)) +
theme_void() +
theme(legend.position="none") +
expand_limits(x=glim, y=glim)
```

Result:

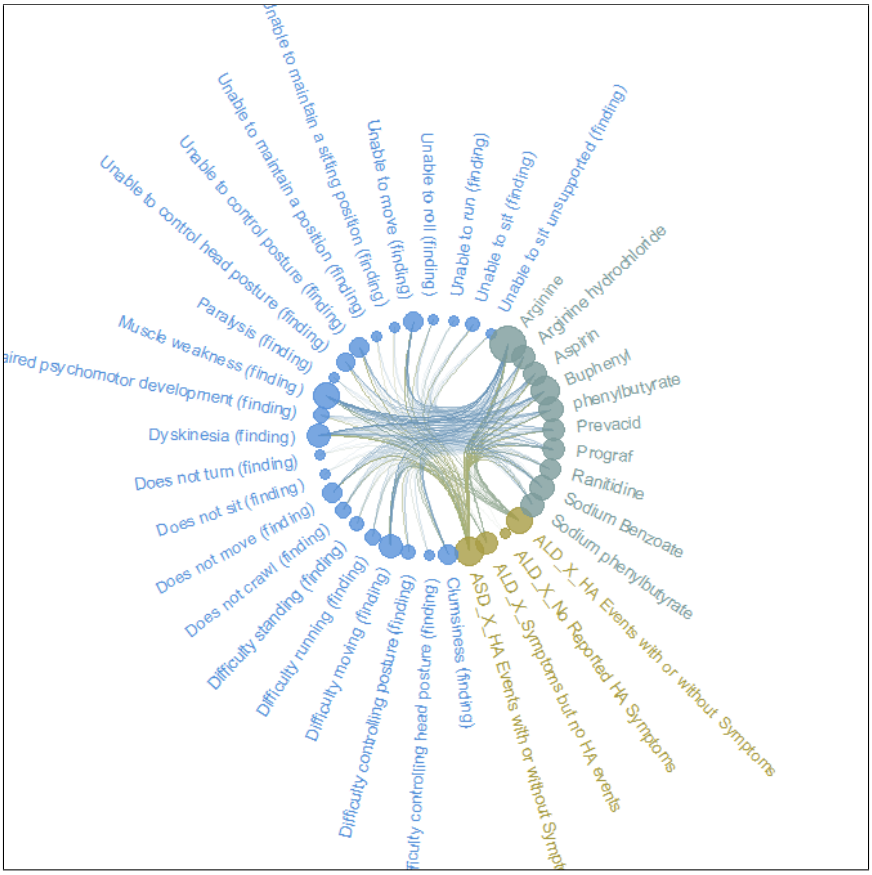

Figure 26: UCD radial edge bundled graph with Nodes Sized for n, and Node Labels, and Edges Colored by Node Type

10.7 Hiearchical Dendrogram with Nodes, Labels, and Edges

Script:

```
ggraph(g, layout='dendrogram', circular=F) +
  # Coordinates:
  # Plot extents are (0,0) to (n_leaves-1,n_levels-1), where n_levels is the number of levels
  # in the hierarchy (one for origin to origin, one for the groups with from="origin," and one
  # for leaves)
```

```
# leaves are placed at x=0..n_leaves-1, y=0
# Draw arcs first, so that points overlay ends
geom_edge_diagonal(alpha=0.5) +
# x, y coordinates are generated by ggraph() (use str(ggraph(...)) to review elements)
# from and to are taken from the edge data frame
geom_conn_bundle(data=get_con(from=edge[,"from"], to=edge[,"to"], v12=edge[,"v12"]),
  aes(x=x, y=y, color=v12), edge_width=0.35, alpha=0.1, tension=0.75) +
scale_edge_color_manual(values=ecolor, guide=F) +
# v and n are taken from the vertex data frame used to construct g
geom_node_point(aes(x=x, y=y, size=n,
  color=ifelse(v!="origin", v, ifelse(name!="origin", name, "Participants")))) +
geom_node_text(aes(x=ifelse(v!="origin", x, x+1),
                   y=ifelse(v!="origin", y-0.2, y),
label=ifelse(name!="origin", name, "Participant"),
  color=ifelse(v!="origin", v, ifelse(name!="origin", name, "Participants")),
  angle=ifelse(v!="origin", 90, 0),
  vjust=0.5,
  hjust=ifelse(v!="origin", 1, 0),
  # Note that node and text sizes are from disjoint spaces, but ggplot does not
  # support multiple scale_size functions
  # Therefore, scale text the range of point sizes, which are based on participant
  # counts
  size=ifelse(v!="origin", 10, ifelse(name!="origin", 20, 30)))) +
scale_color_manual(values=setNames(c("black",
  rgb(matrix(col2rgb("#FFEE66"), nrow=1)/390),
  rgb(matrix(col2rgb("#66AAFF"), nrow=1)/300),
  rgb(matrix(col2rgb("#88AAAA"), nrow=1)/280)), c("Participants", v)),
  guide=F) +
#scale_size_manual(values=c("t1"=3, "t2"=4, "t3"=5), guide=F) +
expand_limits(x=c(NA, length(which(h[,"from"]!="origin"))+5), y=c(-2.5, NA)) +
theme_void() + theme(plot.margin=margin(0.25, 0.25, 0.25, 0.25, "in"))
```

Result:

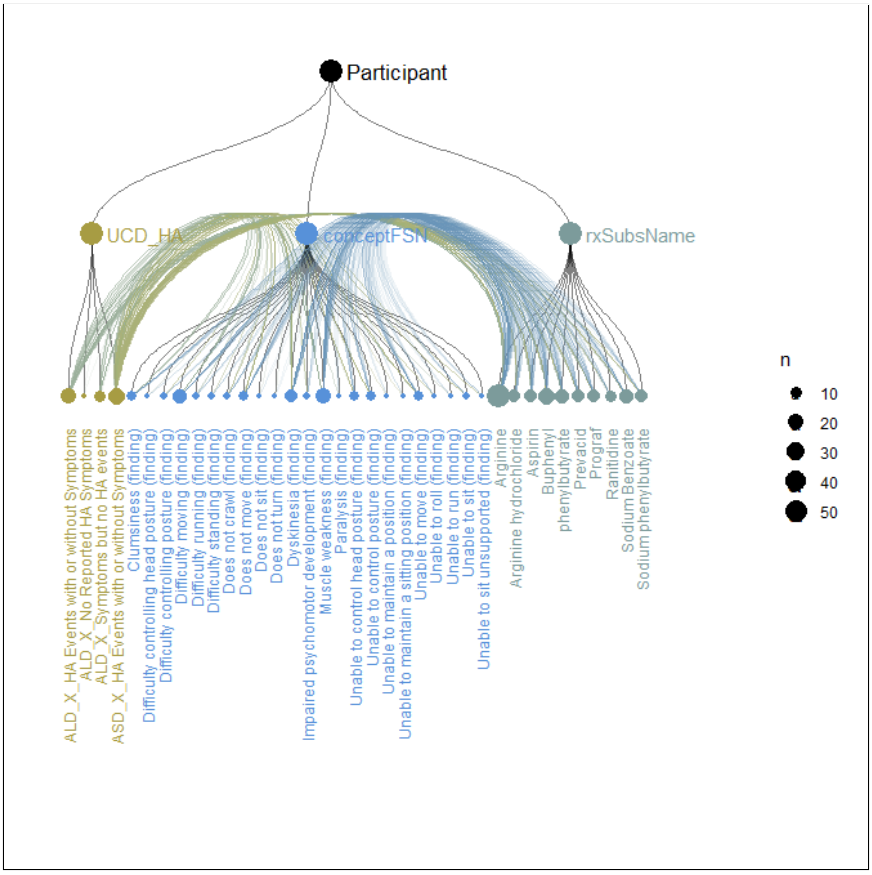

Figure 27: Hierarchical UCD dendrogram with edges relating nodes by participant

11 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