Stat.585X: Draft of Project Plan

Lindsay Rutter

April 10, 2014

Project Topic

The project will focus on the construction and interactive visualization of phylogenetic representation of soybean varieties.

Project Motivation

The motivation of the project stems from my interest in bioinformatics and computational biology. Ideally, the product could be used by biologists, geneticists, and agronomists interested in studying how soybean varieties are related. By referencing the interactive visualization of the phylogenetic tree, these scientists may better understand genetic testing results - in this particular dataset, in terms of copy number variants, single nucleotide polymorphisms, protein content, and yield - and use that knowledge in future breeding.

Available Data

The available data consists of a data frame structure that contains 412 direct child-parent relationships between pairs of soybean varieties. These data were collected from frield trials, genetic studies, and United States Department of Agriculture (USDA) bulletins, and date as early as the first decade of the 1900s.

Data Collection and Processing

The data frame format of the soybean varieties has been represented as an interactive phylogenetic tree, in Shiny software produced by Susan VanderPlas:

http://gsoja.agron.iastate.edu:3838/Soybeans/

In the "Genealogy" tab of the Shiny application, the user may choose one or more varieties in the left-panel menu, and immediately view the phylogenetic tree representation of the selected varieties in the right-panel plot as per reactive programming.

However, currently, the right-panel plots are plotted independently for each soybean variety selected, showing a user-selected number of generations surrounding that variety, regardless of its relationship to any other varieties selected.

Instead, it may be useful for biologists to obtain one large plot in the right-panel that merges the selected soybean varieties, thereby determining not only whether or not there exists a relationship between the varieties, but also showing the relationship as a path in the graphical phylogetntic structure, with possibly the varieties (nodes) and the path (edges) between them highlighted for emphasis.

In total, there are about 230 unique soybean varieties present in the tree data frame. The package igraph might be used to determine any pathway relationships between selected varities:

http://cran.r-project.org/web/packages/igraph/index.html

Final Product

If the Shiny application is properly functioning, it will then be extensively commented in the format that will soon be taught in class. This may bring the application one step closer to being available to interested users via an R Package.

Progress as of April 9

1. The first step was examining the data, and perform data cleaning to ensure there are no obvious errors or counterintuitive information about the soybean varieties.

- I discovered that there are many soybean nodes that do not have a year (`NA`). I plan to simply taking the average of all relationships and using that for these cases (which seems to be 9 years).

```r
setwd("/Users/MacOwner/Desktop/Cook/SBTree")

####################### NECESSARY PACKAGES AND RESOURCES #######################
library(plyr)
load("tree.rda")

###################### TEST GENERATIONAL AGE DIFF #############################

# This produces year_rltnshp, which is a data.frame with 412 rows from the
# original tree file However, there are 5 columns that list (in order) child
# name, parent name, year of child, year of parent, and the difference
# between the year of child minus year of parent (which should be positive)

year_rltnship <- data.frame(child = character(), par = character(), child_yr = integer(0),
    par_yr = integer(0), diff = integer(0), stringsAsFactors = FALSE)

for (i in 1:dim(tree)[1]) {
    child = tree$child[i]
    par = tree$parent[i]
    child_yr = tree$year[i]
    par_yr = subset(tree, tree$child == par)$year[1]
    age_diff = child_yr - par_yr
    new_row = c(child, par, child_yr, par_yr, age_diff)
    year_rltnshp[i, ] = new_row
}

# This produces year_rltnshp_noNA, which is a subset of year_rltnshp, except
# that it only contains the 301 rows where there is a calculated value for
# the column representing the difference between child year minus parent
# year

year_rltnship_noNA = subset(year_rltnshp, year_rltnshp$diff != "NA")
```

```r
# I did summary of the generation difference for the 301 cases where its
# value could be calculated. Oddly, the minimum value was negative, (I
# think) meaning that the child is older than the parent!

min(as.numeric(year_rltnship_noNA$diff))  #-16.5

## [1] -16.5

max(as.numeric(year_rltnship_noNA$diff))  #54

## [1] 54

mean(as.numeric(year_rltnship_noNA$diff))  #9.174419

## [1] 9.174
```

- Another problem I found when cleaning the data, is that there are 16 cases where the parent `year` is younger than (or equal to) the child `year`. For these cases, I also looked at the variable `min.repro.year` in `tree.rda`, but that did not solve the problem.

```r
# I created negRltnshp, a subset of the year_rltnship_noNA where the
# generational difference was less than or equal to zero. This resulted in
# 16 pairs. I also looked at these pairs and tried to determine if using the
# min.repro.year instead from the original tree would fix these problems,
# but it did not always.

negRltshp = subset(year_rltnship_noNA, year_rltnship_noNA$diff <= 0)
```

- What I did to discover this issue is in `testTree.R`. The 16 cases are stored in the variable `negRltshp`.

```
negRltshp

##                    child        par child_yr par_yr  diff
## 17               Bradley     J74-39     1975   1975     0
## 48             D55-4090      Ogden     1945   1953    -8
## 64                  Dare    D52-810     1962   1964    -2
## 66        Davis x Peking      Davis     1958   1966    -8
## 87                Hagood Centennial     1977   1977     0
## 108                  L15      Wayne     1960   1966    -6
## 139              N45-745      Ogden   1951.5   1953  -1.5
## 155          Ogden x CNS      Ogden     1945   1953    -8
## 178               Ransom   D56-1185     1958   1963    -5
## 200                Wells      C1253     1947   1961   -14
## 251             D49-2510      S 100     1945   1945     0
## 274               Dillon Centennial     1977   1977     0
## 282                Essex   S55-7075     1962   1963    -1
## 386 Renville x Capital   Renville     1964   1964     0
## 396               T80-69     J74-40     1975   1975     0
## 406                Wells     C1266R     1947 1963.5 -16.5
```

2. I spoke with scientists and computiational scientists who know more about why these problems may exist in the data.

- Susan VanderPlas explained that the year introduced and year registered can be different by up to 30 years for a given variety, and that this can explain the many cases of children being older than their parents
- It was suggested to me to throw a flag (message) to the user if a child is older than the parent, and indicate to them that the years were forced (imputated) such that the variety is younger than its parents and older than its children

3. Along with Susan and Dr. Cook, I observed the available Shiny applet:

http://gsoja.agron.iastate.edu:3838/Soybeans/

and thought that the greatest-grandparents of the varieties seemed to be `Tokyo` and `AK_004`. As such, I wanted to determine whether or not these two varieties could encompass the majority of the rest of the varieties.

So, I added code to the script `testTree.R` and the relevant lines for me to determine lines that may not be in the union between these two potentially greatest-grandparent varieties:

```r
setwd("/Users/MacOwner/Desktop/Cook/SBTree")

####################### NECESSARY PACKAGES AND RESOURCES #######################
library(plyr)
load("tree.rda")

################## TEST TOTAL NODES IN ALL-ENCOMPASSING TREE ##########################

# There are 206 unique 'child' in the 'tree'
uniqueChild = sort(unique(tree$child))
# There are 165 unique 'parent' in the 'tree'
uniqueParent = sort(unique(tree$parent))
# There are 230 unique nodes in the 'tree'
uniqueNode = sort(union(uniqueChild, uniqueParent))

# These lines were used to generate the nodes from the two large trees
# (Tokyo and AK_004).  The resulting variables were saved to the working
# directory tokyoNodes = vals£df AK004Nodes = vals£df save(tokyoNodes,
# file='tokyoNodes') save(AK004Nodes, file='AK004Nodes')

load("tokyoNodes")
load("AK004Nodes")

# There are 91 unique child in 'Tokyo'
uniqueTokyoChild = unique(tokyoNodes$label)
# There are 61 unique parents in 'Tokyo'
uniqueTokyoParent = unique(tokyoNodes$root)
# There are 91 unique nodes in 'Tokyo'
uniqueTokyoNode = union(uniqueTokyoChild, uniqueTokyoParent)

# There are 104 unique child in 'AK_004'
uniqueAK004Child = unique(AK004Nodes$label)
# There are 67 unique child in 'AK_004'
uniqueAK004Parent = unique(AK004Nodes$root)
# There are 104 unique nodes in 'AK_004'
```

```
uniqueAK004Node = union(uniqueAK004Child, uniqueAK004Parent)

# There are 143 unique nodes in 'AK_004' and 'Tokyo'
tokyoAK004Union = sort(union(uniqueTokyoNode, uniqueAK004Node))

# There are 52 nodes in the intersection
tokyoAK004Int = intersect(uniqueTokyoNode, uniqueAK004Node)

# There are 87 nodes not in Tokyo and AK_004 (230-143)
notInTokyoAK004 = uniqueNode[!(uniqueNode %in% tokyoAK004Union)]
```

This revealed that 87 varieties are not descendants of `Tokyo` and `AK_004`. So, I needed to devise a new plan to generate a complete layout of the tree.

```
str(notInTokyoAK004)

##  chr [1:87] "(Lincoln x Richland) x Korean" "Altona" "Amsoy-71" ...
```

4. I addressed this issue by creating a script called `testIGraph.R`:

- I wanted to develop a visualization of the complete tree, as this is currently not possible in the Shiny application. As such, I created an output .csv file in this scipt that can then be read into the software `Cytoscape`. The software allows for the visualization of the graph and calculation of graph theoretical parameters:

  http://www.cytoscape.org/

  using the following code:

```
# Must use igraph version 0.7.0

setwd("/Users/MacOwner/Desktop/Cook/SBTree")

library(plyr)
library(reshape2)
library(ggplot2)
library(stringr)
library(igraph)

load("tree.rda")
# Total tree nodes (412*2)
treeGraph = as.data.frame(cbind(tree$child, tree$parent))
# Remove any rows with 'NA' relationship (340*2)
treeGraph = treeGraph[-which(is.na(tree$parent)), ]
# Add an edge weight to each pair of vertices (all of weight value equal to
# one)
treeGraph = cbind(treeGraph, rep(1, dim(treeGraph)[1]))
# Add column names the tree
colnames(treeGraph) = c("child", "parent", "edgeWt")

# Write the tree as a .cvs file to be read into Cytoscape
sbTreeGraph = write.csv(treeGraph, "sbTreeGraphTest.csv")
```

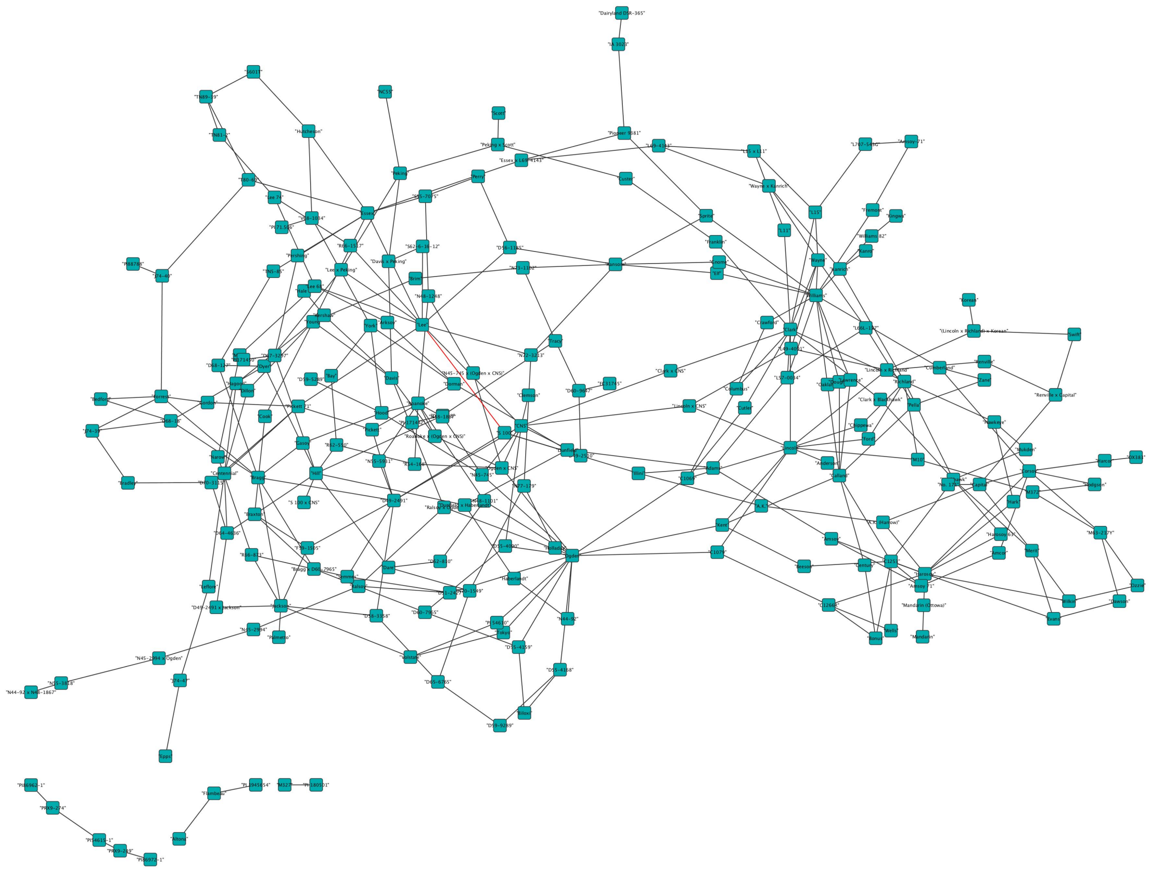

Figure 1: Complete Cytoscape Tree

- I created two main functions in the `testIGraph.R` script:
  - getBasicStatistics()
  - getPath()
- In order to run these functions that contain functions from the igraph package, I needed to convert the tree object into a graph that can be read by igraph.

```
# Convert the data frame object into an object that can be read by igraph
# package
mygraph = graph.data.frame(treeGraph, directed = T)
```

- I also created five test cases, knowing that lineage `Falmbeau` is not connected to any other varieties, and that `Brim` and `Bedford` are connected by a direct ancestor/descendant path, and hence their order mattered for this directed graph. Finally, `Tokyo` and `Narow` could provide an example of cousins, that is not a direct ancestor/descenant lineage.

```
v1 = "Brim"
v2 = "Bedford"
v3 = "Flambeau"
v4 = "Tokyo"
v5 = "Narow"
```

- The first script `getBasicStatistics` returns the basic graph theoretical measurements of the inputted graph. I will need to eventually return the statistics, rather than directly printing to console, as is currently:

```r
# This function prints to console the basic graph theoretical measurements
# of the inputted graph.
getBasicStatistics = function(mygraph) {
    # Get edge and node count from 'structure.info' function of igraph
    numNodes = vcount(mygraph)
    numEdges = ecount(mygraph)
    # Determine if the graph is connected or not from 'clusters' function of
    # igraph
    isConnected = is.connected(mygraph)
    # Determine the number of connected components in the graph from 'clusters'
    # function of igraph
    numComponents = no.clusters(mygraph)
    # Compute the average path length of the graph
    connected = FALSE
    if (isConnected) {
        connected = TRUE
    }
    avePathLength = average.path.length(mygraph, directed = F, unconnected = !isConnected)
    # Determine the log(N) value of the graph
    logN = log(numNodes)
    # Determine the network diameter
    graphDiameter = diameter(mygraph, directed = F, unconnected = !isConnected,
        weights = NULL)
    # Print all statistics to console
    print(paste("The current graph is connected?: ", isConnected))
    print(paste("The current graph has ", numComponents, " connected components."))
    print(paste("The average path length of the graph is: ", avePathLength))
    print(paste("The diameter of the graph is: ", graphDiameter))
    print(paste("The number of nodes in the graph is: ", numNodes))
    print(paste("The number of edges in the graph is: ", numEdges))
    print(paste("The log of the network size is: ", logN))
}
```

  I could then run my function on the entire graph:

```r
getBasicStatistics(mygraph)

## [1] "The current graph is connected?:  FALSE"
## [1] "The current graph has  4  connected components."
## [1] "The average path length of the graph is:  5.33374645892351"
## [1] "The diameter of the graph is:  13"
## [1] "The number of nodes in the graph is:  223"
## [1] "The number of edges in the graph is:  340"
## [1] "The log of the network size is:  5.40717177146012"
```

- The second function `getPath` returns a list of vertices that form the path between the pair of inputted vertices. If not path exists, the function returns character(0). The third parameter is a boolean variable indicating whether or not the graph is directional. The function will look at both directions for a directional graph to make sure the user accidentally did not input the vertices in the reverse order.

```r
# This function determines the shortest path between the two inputted
# vertices, and takes into account whether or not the graph is directed. If
# there is a path, the list of vertices of the path will be returned. If
# there is not a path, a list of character(0) will be returned. Note: For a
# directed graph, the direction matters. However, this function will check
# both directions and return the path if it exists.
getPath = function(v1, v2, isDirected) {
    # If the tree is directed
    if (isDirected) {
        # Convert the data frame object into an object that can be read by igraph
        # package
        mygraph = graph.data.frame(treeGraph, directed = T)
        pathVertices = character()
        # We need to look at both forward and reverse cases of directions, because
        # the user may not know the potential direction of a path between the two
        # vertices
        pathVIndicesForward = get.shortest.paths(mygraph, v1, v2, weights = NA,
            output = "vpath")$vpath[[1]]
        pathVIndicesReverse = get.shortest.paths(mygraph, v2, v1, weights = NA,
            output = "vpath")$vpath[[1]]
        # If there is a path in the forward direction, then we save the names of the
        # vertices in that order
        if (length(pathVIndicesForward) != 0) {
            for (i in 1:length(pathVIndicesForward)) {
                pathVertices = c(pathVertices, get.vertex.attribute(mygraph,
                  "name", index = pathVIndicesForward[i]))
            }
        }
        # If there is a path in the reverse direction, then we save the names of the
        # vertices in that order
        if (length(pathVIndicesReverse) != 0) {
            for (i in 1:length(pathVIndicesReverse)) {
                pathVertices = c(pathVertices, get.vertex.attribute(mygraph,
                  "name", index = pathVIndicesReverse[i]))
            }
        }
    }
    # If the tree is undirected
    if (!isDirected) {
        mygraph = graph.data.frame(treeGraph, directed = F)
        pathVertices = character()
        # The direction does not matter, any shortest path between the vertices will
        # be listed
        pathVIndices = get.shortest.paths(mygraph, v1, v2, weights = NA, output = "vpath")$vpath[[1
        if (length(pathVIndices) != 0) {
            for (i in 1:length(pathVIndices)) {
                pathVertices = c(pathVertices, get.vertex.attribute(mygraph,
                  "name", index = pathVIndices[i]))
            }
        }
    }
    # Return the shortest path, if it exists
    pathVertices
}
```

I could then call the function, such as:

```
getPath(v1, v2, F)

## [1] "Brim"    "Young"   "Essex"   "T80-69"  "J74-40"  "Forrest" "Bedford"
```

Future Work

Now that I have paths working between any two inputted varieties, I would like to create a new version of the Shiny application for the phylogentic soy bean trees, such that only the vertices included in the path are displayed. This would make the visualization of the relationship between the varieties more clear to the user, without all the extraneous vertices causing visual noise around the vertices of interest.