Authors: Justin Beer, Anthony Dalton, Harsha Kommanapalli

Date: April 3rd, 2018

Visualizing Branching Processes

**Abstract**

In this case study, we learn about branching processes and how they relate to modern computing with parallel processing. We will build functions that represent a process and these processes will, in turn, generate children processes. Each process takes a certain amount of time to complete and before it can generate any children. We will present a different way to visualize this process generation and completion other than the way that is presented in the textbook on page 296 in Figure 7.4 [1].

**Introduction**

With the latest and greatest processors, CPU’s can now handle more than one process at the time. This is called parallel processing. CPU’s used to be able to handle only one process at a time, but the user would think that it is doing more than one simply because it switched extremely fast between tasks. Now with multi-core CPU’s, for example Intel’s latest 8th generation i7’s which have 4 physical cores, and the advent of hyper-threading, which is creating two virtual cores for every physical core, we can process more information than ever. Today’s i7’s can handle 8 tasks at any given moment in time. However, even with all of the advances in processing power, Data Scientists can still run up against the processing limit due to the massive datasets and tasks performed to them.

**Literature review**

The text we are utilizing, “Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving” [1], teaches us how to build the “familyTree” function, generate process children, and initially visualize the processing time along with the branching and family structure. The initial visualization that is offered in the textbook is in Figure 7.4 on page 296 [1] and is shown below in Figure 1. This visualization shows each generation on the y-axis and the time to complete on the x-axis. The “x” marks on a line are where the birth of any job offspring took place.

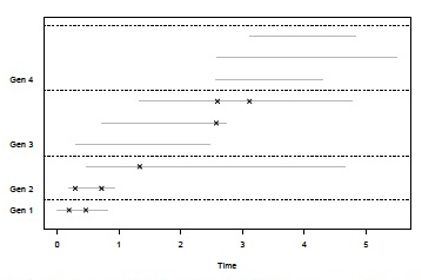


Figure 1. This is Figure 7.4 from page 296 [1].

**Methods**

In order to create a different way to visualize the information in Figure 7.4 on page 296 [1], we needed to utilize the same code that generates the family information. This includes utilizing the “familyTree” and generation functions. The full code base is located in the appendix of this case study.

Once we generated the family, children, and processing / job times, we worked on visualizing the information in a new way. The below Figure 2 shows us the family structure that is generated by the “familyTree” function.

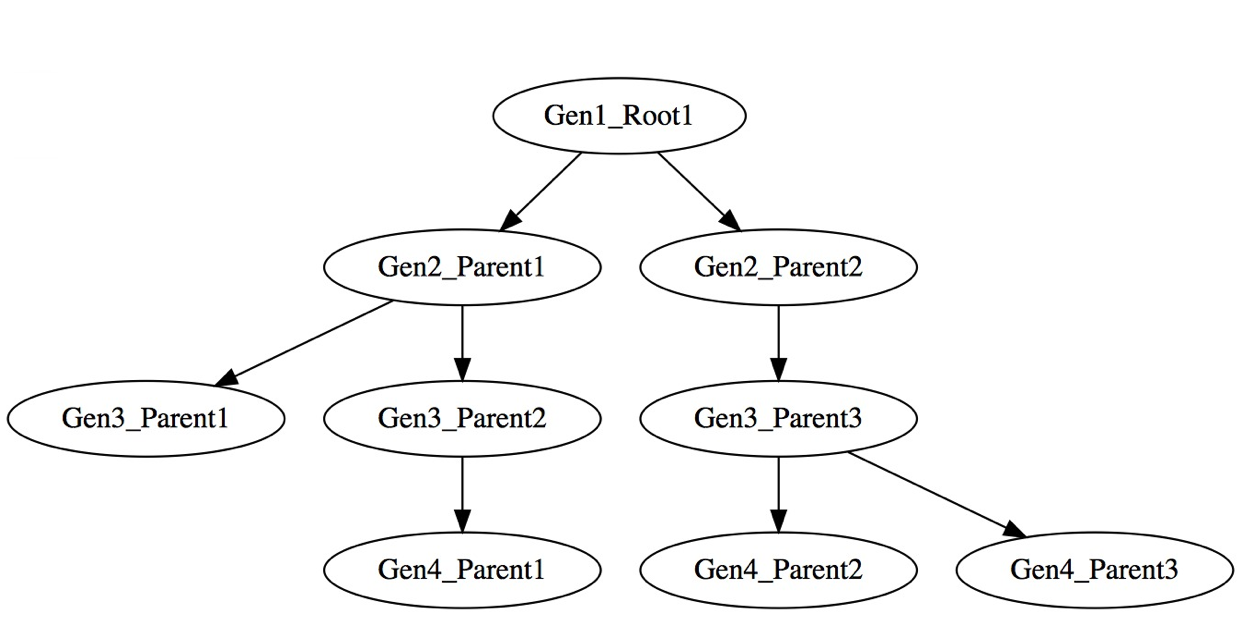


Figure 2. Family Tree structure including Root, Parents, and Children.

The code that generated the family tree can be seen below:

library(data.tree)

tree

fam <- Node$new("Gen1\_Root1")

Gen2ParentID1\_KidID1 <- fam$AddChild("Gen2\_Parent1")

Gen2ParentID1\_KidID2 <- fam$AddChild("Gen2\_Parent2")

Gen3ParentID1\_KidID1 <- Gen2ParentID1\_KidID1$AddChild("Gen3\_Parent1")

Gen3ParentID1\_KidID2 <- Gen2ParentID1\_KidID1$AddChild("Gen3\_Parent2")

Gen3ParentID2\_KidID3 <- Gen2ParentID1\_KidID2$AddChild("Gen3\_Parent3")

Gen4ParentID2\_KidID1 <- Gen3ParentID1\_KidID2$AddChild("Gen4\_Parent1")

Gen4ParentID3\_KidID2 <- Gen3ParentID2\_KidID3$AddChild("Gen4\_Parent2")

Gen4ParentID3\_KidID3 <- Gen3ParentID2\_KidID3$AddChild("Gen4\_Parent3")

fam$births <- 0

fam$completes <- 0.8244658

Gen2ParentID1\_KidID1$births <- 0.1908525

Gen2ParentID1\_KidID1$completes <- 0.9280697

Gen2ParentID1\_KidID2$births <- 0.4631948

Gen2ParentID1\_KidID2$completes <- 4.6525607

Gen3ParentID1\_KidID1$births <- 0.2921203

Gen3ParentID1\_KidID1$completes <- 2.479967

Gen3ParentID1\_KidID2$births <- 0.7177455

Gen3ParentID1\_KidID2$completes <- 2.737410

Gen3ParentID2\_KidID3$births <- 1.3348405

Gen3ParentID2\_KidID3$completes <- 4.782500

Gen4ParentID2\_KidID1$births <- 2.574217

Gen4ParentID2\_KidID1$completes <- 4.291288

Gen4ParentID3\_KidID3$births <- 2.593132

Gen4ParentID3\_KidID3$completes <- 5.500965

Gen4ParentID3\_KidID2$births <- 3.111784

Gen4ParentID3\_KidID2$completes <- 4.834751

plot(fam)

We can see from Figure 2 that we start the first generation with “Root1”. The second generation is then generated from “Root1” and its children are called “Gen2\_Parent1” and “Gen2\_Parent2”. As the job complete they can either finish or branch a new job, the next generation. We can see the “Gen2\_Parent1” generated a new child job called “Gen3\_Parent1”. However, “Gen3\_Parent1” completed its job and did not generate any further jobs / children. Our jobs stop entirely with generation 4 and our processing is complete.

Now we also want to know how long it takes each parent job to either complete or generate a child job. In order to visualize this, we find it simpler to look at a table hierarchy structure, like a folder system. Figure 3 below shows us this structure along with the time to generate a child process “births” and the time to complete its process “completes”.

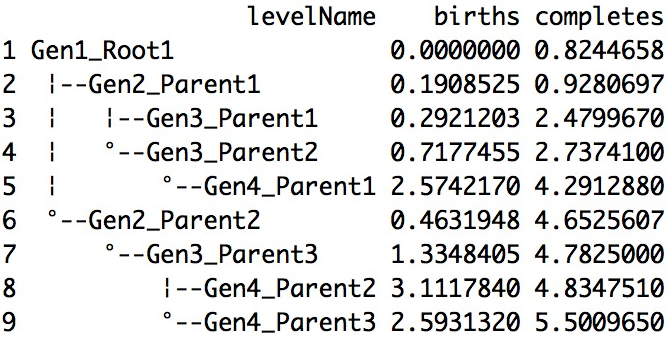


Figure 3. Family Tree Structure as a Hierarchy with Births and Completes.

The code to print out the family tree hierarchy is:

print(fam,"births","completes")

From Figure 3 we can see that we have the “levelName” which is simply the family structure, whether a job / process was the root, parent, or had children. The births represent the time it took the process to generate a child process. For example, we see that it took “Gen2\_Parent1” a time of 0.1908525 seconds to generate the child “Gen3\_Parent1”. We also see that “Gen1\_Root1” generated two children and completed its process approximately 0.382 seconds later. Figure 3 also shows us that the total time to complete all processes from “Gen1\_Root1” all the way to “Gen4\_Parent3” took approximately 5.5 seconds. That means it took the CPU on the machine this ran on approximately 5.5 seconds to simulate a process that generated 8 children and completed 9 jobs.

We also want to visualize Figure 3 in a plot so that we can see on a time scale when each generation was created and when each job was completed. In order to do this, we created Figure 4 below which shows us the time it took to generate another child, “births”, on the y-axis and the time it took to complete its job, “completes”, on the x-axis.



Figure 4. Jobs Plotted with Birthing and Completion Times.

We can see in Figure 4 that each generation of jobs is represented by a different color. The bottom left of the graph shows us a time of 0 for births and 0 for completes. The first point we see once more is “Gen1\_Root1” and we see it took just shy of 1 second to complete and its corresponding child “Gen2\_Parent1” was generated 0.19 seconds after “Gen1\_Root1” came into existence. We can also see how “Gen2\_Parent1” created the child “Gen3\_Parent1” and how it was completed before “Gen1\_Root1” created its second child. This graph shows the overlap that can exist in processing where a jobs second child can generate a relatively long time after the first child has generated its own children and completed. The code that produces Figure 4 is:

library(dplyr)

library(ggplot2)

library(dplyr)

library(ggplot2)

library(ggrepel)

dfGen1 = g[[1]]

dfGen2 = g[[2]]

dfGen3 = g[[3]]

dfGen4 = g[[4]]

dfGen1['label']= "Gen1\_Root1"

dfGen1

label2=c("Gen2\_Parent1","Gen2\_Parent2")

dfGen2$label = label2

dfGen2

label3=c("Gen3\_Parent1","Gen3\_Parent2","Gen3\_Parent3")

dfGen3$label = label3

dfGen3

label4=c("Gen4\_Parent1","Gen4\_Parent2","Gen4\_Parent3")

dfGen4$label = label4

dfGen4

p <- ggplot() + geom\_point(data=dfGen1, aes(x=births, y = completes,colour= "Gen1")) + geom\_point(data=dfGen2, aes(x=births,y=completes,colour= "Gen2")) + geom\_point(data=dfGen3, aes(x=births,y=completes,colour= "Gen3")) + geom\_point(data=dfGen4, aes(x=births,y=completes,colour="Gen4"))

q <- p + geom\_text\_repel(data=dfGen2, aes(x=births,y=completes, label = label ))

r <- q + geom\_text\_repel(data=dfGen1, aes(x=births,y=completes, label = label ))

s <- r + geom\_text\_repel(data=dfGen3, aes(x=births,y=completes, label = label ))

t <- s + geom\_text\_repel(data=dfGen4, aes(x=births,y=completes, label = label ))

t + coord\_flip(ylim = c(0,8), xlim = c(0, 4))

**Results**

The initial Figure 7.4 that we attempted to create different visualizations for shows the generations on their own lines along with and “x” to mark when each one completed. Our graphs and tables show this information from several different perspectives so as to complete the picture of how process branching operates. We saw from Figure 2 the tree hierarchy is generated, from Figure 3 the table of births and completes times, and from Figure 4 a graph of all of the generations along with their births and completes times.

**Conclusion**

Branching processes can seem very complex when viewing them from a high level. However, we can use simplified graphs and tables to see how the process is actually very simple at the granular level. We see how each job / process can generate its children and the interplay between when future generation children are created and when the current process is completed. There is a lot that goes on in the background of modern CPU’s and as Data Scientists we need to understand the mechanics. When we work with datasets containing millions of rows and columns and perform functions on them, we end up generating an entire Earth size population of families, parents, and children within our CPU’s that only exist for a brief moment in time before collapsing back to nothing more than a voltage signal.

**Appendix**

**References**

1. Nolan, D., & Lang, D. T. (2015). Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving. (Data science in R.) London: CRC Press.

**Complete Code Base:**

#7.1.1: Monte Carlo generation of 3 Exp(1) rv's

Case12Path = "/Users/Harsha/desktop/Case12"

setwd(Case12Path)

empirical = replicate(6000, sum(rexp(n = 3)))

mean(empirical)

sd(empirical)

sum(empirical <= 5)/length(empirical)

rates = c(seq(0.1, 1, by = 0.1), seq(2, 7, by = 1))

samples = lapply(rates, function(r) {

replicate(6000, sum(rexp(n = 3, rate = r))) })

pdf("BA\_MCexample.pdf", width = 8, height = 5)

par(mar = c(2,4,1,1))

plot(0, 0, xlim = c(-0.1, 10), ylim = c(0, 1), pch = ".",

xlab="", ylab="Cumulative Distribution")

xx = seq(-0.1, 20, by = 0.05)

invisible(lapply(samples[-(19:20)], function(s) {

Fn = ecdf(s) #empirical CDF function

ptile = Fn(xx) # apply empirical cdf to x values

points(x = xx, y = ptile, type = "l")

}))

labs = c(1, 2, 3, 6, 10, 11, 16)

xs = c(8,8,8,6, 4, 2, 0.75)

poss = c(3,3,3, 2, 2, 2,2)

mapply(function(lab, s, x, r, p)

{

Fn = ecdf(s)

text(x = x, y = Fn(x), labels = r, pos = p, cex = 0.8) #location of the labels; note having to compute this

}, lab = labs, s = samples[labs], x = xs, r = rates[labs], p = poss)

dev.off()

#7.2 Exploring the random process

seedx = 116201412

set.seed(seedx)

# Note: we have to run these in order!

kappa = 0.3

# first job's lifetime

d0 = rexp(1, rate = kappa)

d0

lambda = 0.5

# first birth time

birth1 = rexp(1, rate = lambda)

birth1

# simulate next birth time

itime = c(birth1, rexp(1, rate = lambda))

itime

# simulate next one

itime = c(itime, rexp(1, rate = lambda))

itime

cumsum(itime)

itime = c(itime, rexp(1, rate = lambda))

btime = cumsum(itime)

btime # birth times

btime = btime[ btime < d0 ] # only two were generated during first job's lifetime (< d0)

dtime = d0 + rexp(n = length(btime), rate = kappa)

dtime # birth times plus exp(kappa): death times: all start (in parallel after d0 finishes)

# these can potentially generate offspring as well: "birth and assassination process"

btime[1] + rexp(1, rate = lambda) # 6.233543

dtime[1] + rexp(1, rate = kappa) # 12.55685

# note defining signature of function great way to get started!

genKids = function(bTime, cTime, lambda = 0.5, kappa = 0.3) { # four parameters in function definition

# Parent job born at bTime and completes at cTime

# Birth time of first child

mostRecent = rexp(1, rate = lambda) + bTime

kidBirths = numeric()

while (mostRecent < cTime) {

kidBirths = c(kidBirths, mostRecent)

mostRecent = mostRecent + rexp(1, rate = lambda)

}

# generate lifetimes for all offspring

numKids = length(kidBirths)

runtime = rexp(numKids, rate = kappa)

kidCompletes = rep(cTime, numKids) + runtime

data.frame(births = kidBirths,

completes = kidCompletes)

}

set.seed(seedx) # reset the seed in case we lost track

genKids(1, 6)

genKids(1, 6)

genKids(1, 6) # up arrow + return to get a feel for how these are "distributed"

# we can this recursively:

genBirth = function(currentTime, cTime, births = numeric(), lambda = 0.5) {

# Generate birth time of next job after currentTime

mostRecent = rexp(1, rate = lambda) + currentTime

if (mostRecent > cTime) # note here we "discard" generated data if it doesn't meet the condition

return(births)

else {

births = c(births, mostRecent)

genBirth(currentTime = mostRecent, cTime, births, lambda) # recursive call

}

}

# Effectively, the recursive call replaces the `while` loop

# Deep CS concepts here of "tail-recursion" and convergence

# gen kids recursively:

genKidsR = function(bTime, cTime, lambda = 0.5, kappa = 0.3) {

# Parent job born at bTime and completes at cTime

kidBirths = genBirth(bTime, cTime, lambda = lambda)

# generate lifetimes for all offspring

numKids = length(kidBirths)

runtime = rexp(numKids, rate = kappa)

kidDeaths = rep(cTime, numKids) + runtime

data.frame(births = kidBirths,

completes = kidDeaths)

}

set.seed(seedx) # reset again

numKids = replicate(1000, nrow(genKids(1, 6)))

mean(numKids)

eprobs = table(numKids)/length(numKids)

probs = dpois(x = 0:max(numKids), lambda = 2.5)

plot(eprobs, type = "h",

ylab = "Proportion", xlab = "Number of offspring")

segments(x0 = 0.1 + 0:max(numKids), y0 = rep(0, max(numKids)), # nice plotting function: `segments`; goes along with type "h" + stagger by .1

y1 = probs, col="grey", lwd = 2)

#Very similar distributions

# pdf("numSimOffspring.pdf", width = 8, height = 5)

# oldPar = par(mar = c(4.1, 4.1, 1, 1))

#

# plot(eprobs, type = "h",

# ylab = "Proportion", xlab = "Number of offspring")

# segments(x0 = 0.1 + 0:max(numKids),

# y0 = rep(0, max(numKids)), y1 = probs,

# col="grey", lwd = 2)

#

# par(oldPar)

# dev.off()

# compute the observed proportions vs. theoretical ones

obsCt = table(numKids)

expCt = 1000\* c(dpois(0:8, lambda = 2.5),

ppois(8, lower.tail = FALSE, lambda = 2.5))

# Compute chi^2 statistic: recall! good for comparing frequencies in buckets

stat = sum((obsCt - expCt)^2/expCt)

pchisq(stat, df = 9, lower.tail = FALSE) # 0.9943803 # p-value

# Rewrite above functionality returning a vector, so we can use vector comprehension next (that is, applying a function across a vector)

genKidsU = function(bTime, cTime, lambda = 0.5, kappa = 0.3) {

# Generate the birth times and assassination times

# for the children of a job who is born at bTime

# and completed at cTime.

lambda = (cTime - bTime) \* lambda

numKids = rpois(1, lambda = lambda)

kidBirths = sort(runif(numKids, min = bTime, max = cTime))

# generate lifetimes for each offspring

runtime = rexp(numKids, rate = kappa)

kidDeaths = rep(cTime, numKids) + runtime

return(data.frame(births = kidBirths, completes = kidDeaths))

}

#7.4 profiling code performance

seedx = 116201412

set.seed(seedx)

time1 = system.time( replicate(40, genKids(1, cTime = 9)) )

time2 = system.time( replicate(40, genKidsU(1, cTime = 9)) )

time1/time2

time1 = system.time( replicate(40, genKids(1, cTime = 100)) )

time2 = system.time( replicate(40, genKidsU(1, cTime = 100)) )

time1/time2

set.seed(seedx)

Rprof("profGenKids1.out")

invisible( replicate(1000, genKids(1, cTime = 100)) )

Rprof(NULL)

summaryRprof("profGenKids1.out")$by.self

set.seed(seedx)

Rprof("profGenKidsU.out")

invisible( replicate(1000, genKidsU(1, cTime = 100)) )

Rprof(NULL)

summaryRprof("profGenKidsU.out")$by.self

# 7.5

# apply genKidsU across a vector using `mapply`

genKidsV = function(bTimes, cTimes, lambda = 0.5, kappa = 0.3) {

# bTimes & cTimes - vector of birth and completion times

# Determine how many children each job has

parentAge = cTimes - bTimes

numKids = rpois(n = length(parentAge),

lambda = lambda\*parentAge)

# Determine the birth and completion times of the children

mapply(function(n, min, max) {

births = sort(runif(n, min, max))

runtimes = rexp(n, rate = kappa)

completes = rep(max, n) + runtimes

data.frame(births, completes)

},

n = numKids , min = bTimes, max = cTimes,

SIMPLIFY = FALSE)

}

# check the above work with a unit test 7.6

bTimes1 = 1:3

cTimes1 = c(3, 10, 15)

seed1 = 12062013

set.seed(seed1)

kids = genKidsV(bTimes1, cTimes1)

kids

kids2 = genKidsV(bTimes1, cTimes1)

sapply(kids2, nrow)

set.seed(seed1)

kids3 = genKidsV(bTimes = bTimes1, cTimes = cTimes1)

identical(kids, kids3)

# 7.7 Generate kids vectorized returning the ids of the kids and parents

genKidsV = function(bTimes, cTimes, parentID, lambda = 0.5, kappa = 0.3) {

# Determine how many children each job has

parentAge = cTimes - bTimes

numKids = rpois(n = length(parentAge), lambda = lambda \* parentAge)

if (sum(numKids) == 0) return(NULL)

# Determine the birth times of the children

kidStats =

mapply(function(n, min, max) {

births = sort(runif(n, min, max))

runtimes = rexp(n, rate = kappa)

completes = rep(max, n) + runtimes

data.frame(births, completes)

},

n = numKids , min = bTimes, max = cTimes,

SIMPLIFY = FALSE)

return(data.frame(parentID = rep(parentID, numKids),

kidID = 1:sum(numKids),

births = unlist(lapply(kidStats, "[[", "births")),

completes = unlist(lapply(kidStats,"[[", "completes"))

))

}

set.seed(seed1)

genKidsV(bTimes1, cTimes1, parentID = letters[1:3])

#7.8 Family tree

familyTree = function(lambda = 0.5, kappa = 0.3, maxGen = 10) {

# maxGen - maximum number of generations to observe

# Return value - a list with 1 data frame per generation.

allGens = vector(mode = "list", length = maxGen)

# Generate the root of the tree

allGens[[1]] = data.frame(parentID = NA, kidID = 1, births = 0,

completes = rexp(1, rate = kappa))

# Generate future generations, one at a time.

for (i in 2:maxGen) {

nextGen = genKidsV(bTimes = allGens[[ (i - 1) ]]$births,

cTimes = allGens[[ (i - 1) ]]$completes,

parentID = allGens[[ (i - 1) ]]$kidID,

lambda = lambda, kappa = kappa)

if (is.null(nextGen)) return(allGens[ 1:(i - 1) ])

allGens[[ i ]] = nextGen

}

return(allGens)

}

set.seed(seed1)

tree = familyTree(lambda = 0.4, kappa = 1, maxGen = 10)

pdf("BA\_FamilyTree.pdf", width = 8, height = 5)

oldPar = par(mar = c(4.1, 4.1, 0.5, 0.5))

set.seed(seed1)

g = familyTree(lambda = 0.4, kappa = 1, maxGen = 10)

g

print(g)

maxLife = max(sapply(g, function(gen) max(gen$completes)))

numFamily = sum(sapply(g, nrow))

plot(0,0, ylim = c(0.5, numFamily + 0.5), xlim = c(0, maxLife),

xlab = "Time", ylab = "", type ="n",

axes = FALSE)

box()

axis(1)

numGen = length(g)

numKids = sapply(g, nrow)

treeN = g[ 2:(numGen + 1) ]

birthNum = c(0, cumsum(sapply(g, nrow))[ -length(g)])

axis(2, at = birthNum + 1,

labels = paste("Gen", 1:numGen), tick = FALSE, las = 1)

mapply(function(gen, nextGen, birthNum) {

birthOrder = birthNum + (1:nrow(gen))

segments(x0 = gen$births, x1 = gen$completes, y0 = birthOrder,

lwd = 3, col = "grey")

abline(h = 0.5 + max(birthOrder), lty = 2, col="black" )

if (all(!is.na(nextGen$births)))

points(x = nextGen$births, y = birthOrder[nextGen$parentID],

pch = 4)

},

gen = g, nextGen = treeN, birthNum = birthNum )

par(oldPar)

dev.off()

g

--------------------------

#try to visualise above in a different way

#Q.6 Figure 7.4 is a custom visualization of the birth and completion times for a tree. Design an alternative custom visualization of the return value from familyTree().

#HN : initial generation visualisation

library(dplyr)

library(ggplot2)

library(dplyr)

library(ggplot2)

library(ggrepel)

dfGen1 = g[[1]]

dfGen2 = g[[2]]

dfGen3 = g[[3]]

dfGen4 = g[[4]]

dfGen1['label']= "Gen1\_Root1"

dfGen1

label2=c("Gen2\_Parent1","Gen2\_Parent2")

dfGen2$label = label2

dfGen2

label3=c("Gen3\_Parent1","Gen3\_Parent2","Gen3\_Parent3")

dfGen3$label = label3

dfGen3

label4=c("Gen4\_Parent1","Gen4\_Parent2","Gen4\_Parent3")

dfGen4$label = label4

dfGen4

p <- ggplot() + geom\_point(data=dfGen1, aes(x=births, y = completes,colour= "Gen1")) + geom\_point(data=dfGen2, aes(x=births,y=completes,colour= "Gen2")) + geom\_point(data=dfGen3, aes(x=births,y=completes,colour= "Gen3")) + geom\_point(data=dfGen4, aes(x=births,y=completes,colour="Gen4"))

q <- p + geom\_text\_repel(data=dfGen2, aes(x=births,y=completes, label = label ))

r <- q + geom\_text\_repel(data=dfGen1, aes(x=births,y=completes, label = label ))

s <- r + geom\_text\_repel(data=dfGen3, aes(x=births,y=completes, label = label ))

t <- s + geom\_text\_repel(data=dfGen4, aes(x=births,y=completes, label = label ))

t + coord\_flip(ylim = c(0,8), xlim = c(0, 4))

# Visual # 1 - Hand Coded - Dalton Code set

install.packages("data.tree")

library(data.tree)

tree

fam <- Node$new("Gen1\_Root1")

Gen2ParentID1\_KidID1 <- fam$AddChild("Gen2\_Parent1")

Gen2ParentID1\_KidID2 <- fam$AddChild("Gen2\_Parent2")

Gen3ParentID1\_KidID1 <- Gen2ParentID1\_KidID1$AddChild("Gen3\_Parent1")

Gen3ParentID1\_KidID2 <- Gen2ParentID1\_KidID1$AddChild("Gen3\_Parent2")

Gen3ParentID2\_KidID3 <- Gen2ParentID1\_KidID2$AddChild("Gen3\_Parent3")

Gen4ParentID2\_KidID1 <- Gen3ParentID1\_KidID2$AddChild("Gen4\_Parent1")

Gen4ParentID3\_KidID2 <- Gen3ParentID2\_KidID3$AddChild("Gen4\_Parent2")

Gen4ParentID3\_KidID3 <- Gen3ParentID2\_KidID3$AddChild("Gen4\_Parent3")

fam$births <- 0

fam$completes <- 0.8244658

Gen2ParentID1\_KidID1$births <- 0.1908525

Gen2ParentID1\_KidID1$completes <- 0.9280697

Gen2ParentID1\_KidID2$births <- 0.4631948

Gen2ParentID1\_KidID2$completes <- 4.6525607

Gen3ParentID1\_KidID1$births <- 0.2921203

Gen3ParentID1\_KidID1$completes <- 2.479967

Gen3ParentID1\_KidID2$births <- 0.7177455

Gen3ParentID1\_KidID2$completes <- 2.737410

Gen3ParentID2\_KidID3$births <- 1.3348405

Gen3ParentID2\_KidID3$completes <- 4.782500

Gen4ParentID2\_KidID1$births <- 2.574217

Gen4ParentID2\_KidID1$completes <- 4.291288

Gen4ParentID3\_KidID3$births <- 2.593132

Gen4ParentID3\_KidID3$completes <- 5.500965

Gen4ParentID3\_KidID2$births <- 3.111784

Gen4ParentID3\_KidID2$completes <- 4.834751

print(fam,"births","completes")

plot(fam)