# Abstract

In this analysis, we find an alternative visualization for the data returned by the *familyTrees*() function. This function generates a tree based on a birth/assassin relationship. This relationship is defined by a root process generating a random number of child processes. Each child process may not begin running until the parent process has completed, but it may create its own child processes before it begins to run. The relationship between parent and child is presented in a visual format displaying how each node in the tree is connected.

# Introduction

In this paper we will analyze how computer processes are generated, how they create new processes, and how those processes are terminated. We want to demonstrate different methods of how you can visually represent the relationship of children processes and the interaction of generating new processes and terminating works. There are a couple of basic principles of when processes run and how they interact with one another.

1. Processes run such that when they are created, they can generate child processes. Children processes can keep being added to a process until it is executed.
2. Child processes must wait until the parent process is done running before it can run, but all children processed of a given parent process can run concurrently and independently.
3. A given parent process can generate multiple child processes, which can in turn generate an innumerable number of child processes

This method is discussed by Aldous and Krebs as a model of queueing processes. [1] In the paper the rate of addition of children is a factor of a Poisson process (rate λ). After the process is executed it then lives for a random amount of time (rate κ). We will cover these numbers later, because for certain values of κ and λ, the processes will never terminate since there will be too many children generated that cannot be terminated.

This dataset is created dynamically through the application by setting a seed and randomly generating a distribution of values according to a Poisson process. The Poisson process is the probability that a random event will occur in a given time. [2] The time interval is fixed and the number of events that occur in a given time is a Poisson random variable. This is represented as the following:

We initialize the program such that the first process creation happens after time 0, with ever process being created after, according to rate λ. We repeat the generation of children processes until the completion time of the parent process; at that time children processes will execute and follow a similar distribution as the parent process. Processes fall into two distinct parts, first the job is created while waiting for the parent to finish then the process the executes after the parent has completed. To create a dataset that follows this premise we need to first create a function to create child processes, as an input we would take the parent’s birth and death time, the rate of creation of child processes (rate λ) and the completion rate of processes (rate κ). Our function outputs a DataFrame with the creation and completion rates of all the processes. Once we have this function then we can run it once or iterate over it multiple times to get an average number of children from our Poisson creation process.

From this DataFrame we can map out the resulting distribution of the births and deaths of all processes. One thing to note is that the rates of λ and κ are critical values to ensuring there is a finite runtime to these processes. With a λ of 1 then we could get an exponentially growing dataset due to the number of children processes being created. This relationship will not be explored in our analysis, but values of λ and κ should be chosen to prevent the possibility of an infinite loop from generating. Fail safes could be incorporated into the process generation function to only allow the Poisson process to run for a certain number of iterations before terminating.

# Background

In our analysis we are employing the use of a stochastic model to help determine the number of children processes created, given a few input variables. A stochastic model is used in this case as opposed to a deterministic model because of the number of variables that could determine the number of processes created from this Poisson model are innumerable. To properly account for the uncertainty in our calculations a stochastic model is employed. Stochastic modeling is employed in other areas where there is a great degree of uncertainty in the final model including stock market returns, weather prediction, deep learning, and quantum computing.

The stochastic method used is a Monte Carlo simulation which is a method of running a random process multiple times to simulate a real-life process. In this case we are running the simulation using a Poisson distribution due to the discrete nature of process creation and execution. A Poisson distribution will represent the number of occurrences of an event in a fixed interval of time. This method of distribution is used because only because the number of a processes are a discrete value and aren’t a continuous value. Only after repeatedly running the simulation will we get a continuous variable representing the mean and standard deviation of the number of processes created over all iterations of our function.

The ability of a Monte Carlos model to accurately simulate a desired process is based on the ability to generate random numbers. If the underlying mechanism is not sufficiently random then bias will be found in the model that may cause inaccuracies in the final output. R relies on the underlying OS to generate any random numbers through using a seed value to generate a number that appears to be random through an algorithm known as a pseudo random number generator also known as deterministic random number generators. These algorithms appear to be random but as their alternate name implies, they can have a deterministic outcome if given the same seed repeatedly. The characteristic of repeatability is desirable when it comes to testing, but randomness is necessary to put this code into production. As it stands the implementation of random in R is sufficient for our purposes, but additional methods of creating truly random variables will be discussed in future works.

# Method

The original image graphically displays the birth and completion of nine processes across four generations of a stochastic birth/assassination process. This data represents a tree structure where a single parent may have from 0 to *n* children. The original visualization is shown in the Figure 1 below. In this figure, each process is represented by a grey line. The length of the line measures the lifetime of the process. The black ‘**x**’ on each line represents a point in time where the current process spawned a new process to run in the following generation. The dotted lines separate the four generations of process. The entire process ends because none of the children in generation 4 spawned any children.

Figure 1 - Original Data Visualization

A screenshot of a social media post

Description automatically generated

The tree is created using two functions, *familyTree* and *genKidsV*. The *familyTree* function calls *genKidsV* to create each generation in the family by passing *genKidsV* a data frame of birth times, completion times and parent ID of the current generation. The nature of the stochastic random model could result in an explosion of children. This is prevented by passing a parameter call *maxGen* to the *familyTree* function to be sure the process of spawning new children terminates at the number defined my *maxGen*. It is assumed that *ith* element on the row of births corresponds to the *ith* row of completions, such that for each node in a generation . Our simulation creates the data set shown in Table 1. This is the data used to create the image in Figure 1.

Table 1 - Data from simulation

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Generation | parentID | kidID | births | completions |
| 1 | NA | 1 | 0.0 | 0.824 |
| 2 | 1 | 1 | 0.191 | 0.928 |
| 2 | 1 | 2 | 0.463 | 4.653 |
| 3 | 1 | 1 | 0.292 | 2.480 |
| 3 | 1 | 2 | .718 | 2.737 |
| 3 | 2 | 3 | 1.335 | 4.783 |
| 4 | 2 | 1 | 2.574 | 4.291 |
| 4 | 3 | 2 | 2.593 | 5.501 |
| 4 | 3 | 3 | 3.111 | 4.834 |

Once we have this data in hand, we want to create a tree structure diagram that represents the parent child relationship inherent in the tree structure. We will use the *data.tree* class to represent the tree. However, looking at the table, the nodes as they are generated by *familyTree* do not have unique names. That is, we cannot uniquely identify a node whether it is referenced in the *kidID* column or in the parentID column. The first *kidID* process born in Generation 1 has the same identifier as the first process born in Generations 2-4. This also causes the problem where we lose track of which *kidID* is associated with which *parentID* across generations.The *data.tree* requires a parent/child relationship with no ambiguity. When converting from a *data.frame* to a *data.tree* we need to provide the path for the parent/child relationship in the data frame variable *pathString*. Converting the data in Table 1 to data.tree we can see that the lack of unique names for each node leaves problems with the data. Table 2 shows we have multiple paths that have the same name but are not referencing the same parent/child relationship. The data created running the *familyTree* function does not contain the information to create a data tree with unique node names however the original visualization implies this information. Using the data from Table 2 allows us to create the date frame which maintains a unique name for each node such that when a node is referenced either as a *parentID* or a *kidID*, it has the same number and that number is unique for that node.

Table 2 – Data.frame for data.tree class

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Generation | parentID | kidID | births | completions |
| 1 | NA | 1 | 0.0 | 0.824 |
| 2 | 1 | 2 | 0.191 | 0.928 |
| 2 | 1 | 3 | 0.463 | 4.653 |
| 3 | 2 | 4 | 0.292 | 2.480 |
| 3 | 2 | 5 | .718 | 2.737 |
| 3 | 3 | 6 | 1.335 | 4.783 |
| 4 | 5 | 7 | 2.574 | 4.291 |
| 4 | 6 | 7 | 2.593 | 5.501 |
| 4 | 6 | 8 | 3.111 | 4.834 |

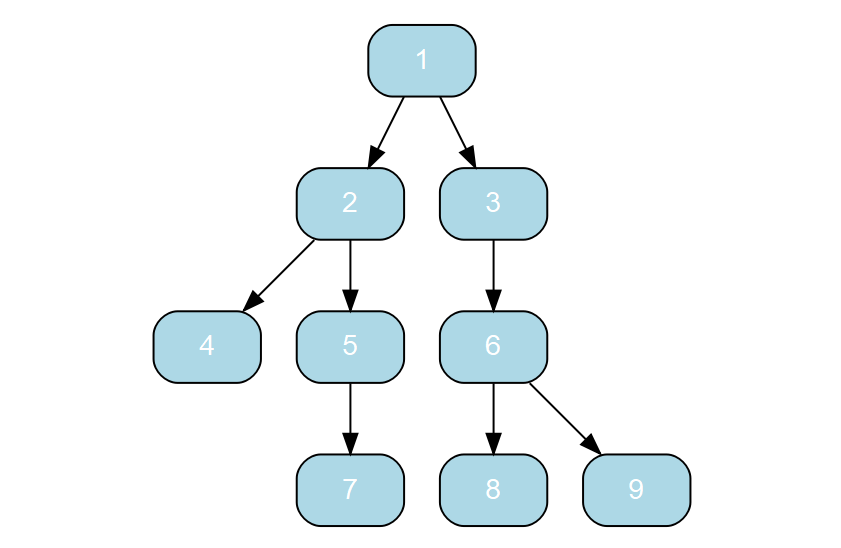
Now that we have each node uniquely named, the path to get from a parent to a child, and vice-versa is available. The variable pathString defines the root/parent/child path to create the tree.

Table 3 - Parent/Child relationship

|  |  |  |
| --- | --- | --- |
| parentID | kidID | pathString |
| 0 | 1 | GEN / 1 / 1 |
| 0 | 2 | GEN / 1 / 2 |
| 1 | 1 | GEN / 1 / 1 |
| 1 | 2 | GEN / 1 / 2 |
| 2 | 3 | GEN / 1 / 3 |
| 4 | 3 | GEN / 2/ 3 |
| 5 | 3 | GEN / 2 / 3 |
| 5 | 2 | GEN / 3 / 2 |

Once we have data formatted correctly, it is just a matter of calling the plot library to display the final parent/child tree as seen below in Figure 2 - Parent/child plot from familyTree function

Figure 2 - Parent/child plot from familyTree function



# Results

Each of the visualizations has its strengths and limitations. The alternate visualization we derived using the tree structure provides an easier way to visually associate a parent node with a child node. It is clear the which children were spawned be each individual parent. The lifetimes of each process are not graphically conveyed through this tree structure and that is a limitation of the visual.

The original visual did not convey the parent/child relationship as clearly as this visual. The relative lifetime of each node compared to others is readily apparent and a good estimate of the absolute amount of time each process is alive can be ascertained.

The ‘**x**’ marking the birth of each new process can be lined up with the process in the next generation, but it is not always easy to see which process the child of the which parent is. This confusion is most obvious when looking at the two latest process in generation 3. It is not obvious which process in generation 3 spawned which process in generation 4. This relationship is obvious in the alternate visual we produced.

This limitation of the original function required us to manually create a data frame that uniquely identified the nodes. Given the way *genKidsV* and *familyTree* functions are written, this type of tree structure is not available from the data. Creating this type of data structure is not scalable.

# Future Work

The immediate future work to make *familyTree* and *genKidsV* more usable would be to have each them work to identify individual nodes. Given that information both of these visuals would be possible and likely others.

Future work could also come into play with a couple of different methods including representing different models with our technique and using different methods of generating our random numbers. It was mentioned that we did not consider the full range of the values that λ and κ can take, rather we limited the model to only using values that gave finite ranges. Other work could be done to visualize what happens to the model when the values provided don’t allow for all processes to complete. Plotting this relationship would rely on representing the visuals with a method that can represent exponentially different values, such as a log plot. It would also be difficult to represent the creation and execution of individual processes and instead the creation and termination times could be aggregated across generations.

Our method of generating random functions relies on the rexp function from Python, which is a pseudorandom function that generates a mathematically computed value from a seed that appears to be random. The seed could be selected from several sources including milliseconds from the computer’s clock, to keystrokes from the user, to a static number for testing purposes. The problem with pseudorandom number generators is that while they can appear random, there can be issues if the underlying seed is not sufficiently random. When creating statistical models this would bias in the model when compared to what is being simulated. A method to create more random variables could rely on quantum computers to create truly random variables. Research by Peter Bierhorst is showing promise in the ability to generate random numbers due to the physical properties of Quantum Physics. [3] While Bierhort’s machine is used to cryptography, its uses could be extended out to other fields where true randomness is a requirement.

# References

|  |  |
| --- | --- |
| [1] | D. Aldous and W. Krebs, "The 'Birth-and-Assassination' Process.," *Statistics and Probability Letters,* vol. 10, pp. 427-430, 1990. |
| [2] | D. N. Lang and D. T. Lang, "Chapter 7 - Simulation Study of a Branching Process," in *Data Science in R - A Case Studies Approach to Computational Reasoning and Problem Solving*, Boca Rotan, CRC Press, 2015, p. 50. |
| [3] | S. Chen, "Quantum Mechanics Could Solve Cryptography’s Random Number Problem," Wired, 11 April 2018. [Online]. Available: https://www.wired.com/story/quantum-mechanics-could-solve-cryptographys-random-number-problem/. [Accessed 18 November 2018]. |

# Appendix

R Code

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])

familyTree = function(lambda = 0.5, kappa = 0.3, maxGen = 10) { # Mike\_t begin

# 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)

}

#

# Create data from family Tree and

# convert to a flat data frame

#

set.seed(seed1)

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

df <- ldply(g,data.frame)

#

# Create the data frame with unique node names

# and convert to data.tree

#

testf <- data.frame(parentID=c(1,1,2,2,3,5,6,6), kidID=c(2,3,4,5,6,7,8,9))

treef <- FromDataFrameNetwork(testf)

print(treef,"parentID","kidID","births","completes")

SetNodeStyle(treef,style="filled,rounded",fillcolor="LightBlue",

shape="box",fontname="helvetica")

plot(treef)