## **Abstract**

A computational process consists of multiple jobs, either sequential or parallel. Parallel jobs do not depend on each other and can be performed simultaneously inside the main process, whereas for sequential jobs, the later one cannot begin until the previous one is complete. This branching process can be repeated multiple times. Both the start and end time of these jobs are determined by a stochastic process which is a mathematical model based on probability distributions. In this paper, we construct two functions: one that generates random births during a given time interval and the other that uses the first function to develop a family tree for the entire branching process. Using these functions, we provide an alternative way to visualize the original branching process figure provided in Deborah Nolan and Duncan Temple Lang’ s book, *Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving*.

## **Introduction**

A branching process is a stochastic process commonly used to model population growth. More specifically, a branching process is a mathematical model with some underlying probability distribution that governs the number of children an individual will have [1]. In this paper, we explore a visualization of a randomly generated branching process that compactly encapsulates the duration of each individual’s lifetime and the birth times of each individual’s children across multiple generations.

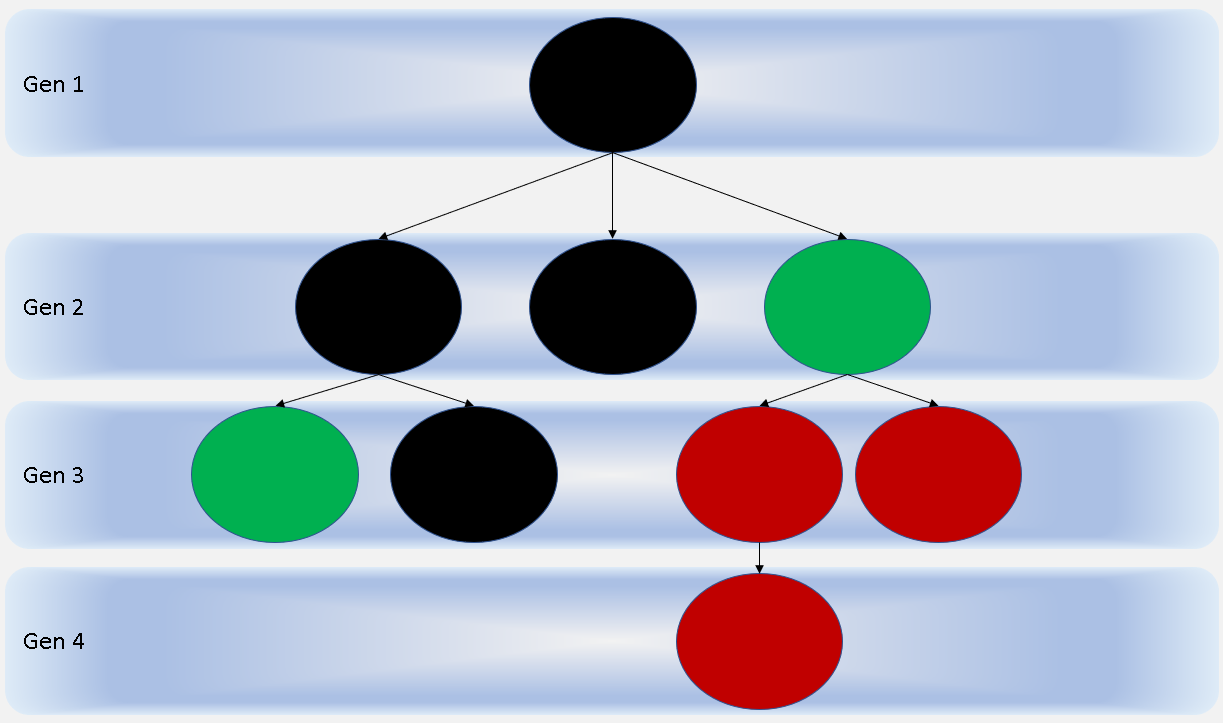
Using the programming language R, we construct a function that generates random births during a given time interval following a Poisson distribution. We create a second function that uses the birth generation function as an input in order to develop a family tree for the entire branching process. We then plot the results of a specific birth process with each generation along the *x*-axis and time along the *y*-axis.

The remainder of this paper is structured as described here. The following section provides an overview of existing approaches in the literature for visualizing branching processes. In the Methods section, we detail our development of the functions necessary for creation of our proposed branching process visualization. We then present our visualization of a four-generation branching process and explain how to interpret it. We finish the paper by highlighting the strengths of our visualization approach and suggesting an enhanced, easier-to-follow illustration of a branching process.

## **Background**

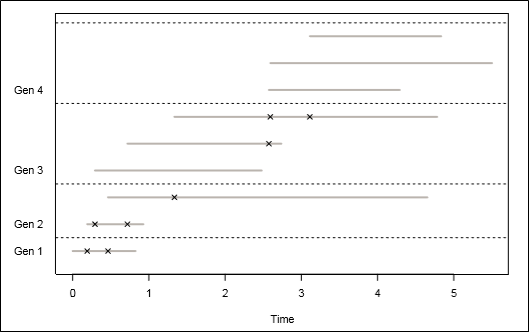
David Aldous and William Krebs developed a set of rules by which an individual, also referred to as a job, in a branching process must adhere. A branching process begins with an initial job from which all subsequent jobs are generated. The job start times are independent and identically distributed. A child job must wait until its parent job completes before it can begin. However, each job can produce its own offspring at any time, even if it is waiting for its parent to complete its job [2].

In Figure 1, an example branching process is illustrated using a tree diagram. Each oval represents a job and the lines connecting one job to another represent a parent-child relationship. The initial job is at the top of the diagram while subsequent generations appear sequentially from top to bottom to the fourth generation. Black ovals are jobs that have finished running, green ovals are jobs that are currently running, and red ovals are jobs that are waiting for their parent jobs to complete. The initial job has three children, two of which have completed their jobs. The third child has yet to complete its task, and it has generated two children, one of which has yielded a grandchild, that are waiting for their parent to finish its job before they can start theirs. Note that this branching process as it appears in Figure 1 has not yet reached its terminal state, as the green and red ovals can potentially produce more children.

Figure 1. Tree diagram of branching process

In their textbook, *Data Science in R: A Case Studies Approach to Computational Reasoning and Problem Solving*, Deborah Nolan and Duncan Temple Lang describe an enhanced method of branching process visualization over the one in Figure 1. Their proposed visualization is a two-dimensional graph with time along the *x*-axis and each generation along the *y*-axis, as shown in Figure 2. Each job’s duration is represented by a horizontal solid gray line while the birth of a child is represented by a black “X.” The generations are separated by a black dashed line. The branching process terminates when none of the solid lines in a generation have an “X” on them [2].

Figure 2. Graph of random branching process



The key advantage of the graph in Figure 2 over the tree in Figure 1 is that the former shows the duration of each job and the relative point in time of each birth whereas the latter does not convey any notion of time elapsed between jobs or generations. In the remainder of this paper, we modify Nolan and Temple Lang’s visualization in a way that we believe makes the graph easier for the human eye to follow.

## **Methods**

As our visualization focuses on birth and completion time of jobs in a randomly generated branching process, the two key components are i) birthtime of a job and ii) job lifetime or age. Birthtime is the random point in time when a job starts, and the lifetime is defined as the total time that the job needs to complete. Hence,

*Job Lifetime or Age = Job Completion Time (cTimes) - Job Birth Time (bTimes)*

In general, job lifetime follows an exponential distribution. The probability density function of an exponential distribution with a rate parameter *κ*, is defined as follows:

κe*-*κx *where* x *> 0*

Birth times statistically follow a Poisson process which denotes the probability distribution of selecting a random point on a time scale. Following a Poisson process, the total number of births in a given time interval is given by:

P *(*k *births in a time interval of 1) =* λkk!*-* λ*/*e *where* k *= 0, 1, 2, …*

The first job in the top most level of the family tree starts at time point 0. Since the interarrival times in a Poisson process independently follow an exponential distribution, we generate the birth times of the offspring from independent exponential distributions.

The function *genKids()* generates births of random offspring using five parameters: i) parent ID, ii) parent’s birth time, iii) parent’s death/completion time, iv) the rate for the interarrival of the offspring (*λ*) and v) the rate for the run time of the offspring (*κ*). This function returns a data frame with information on: i) parent ID, ii) children’s IDs, iii) birth time for children j and iv) completion time for children. The output values depend on the parent’s age which is the difference between the birth and completion time of the parent. The number of children is generated by a Poisson process. If the total number of children are 0, the function returns a null value and the process does not extend to the next generation. For example, *genKids(0, 1, NA, 0.5, 0.3)* yields the output as shown in Table 1, it indicates that the parent in the first generation gives birth to one child job that starts at time 0.1141938 and ends at 14.9441.

Table 1. Example output of *genKids(0, 1, NA, 0.5, 0.3)*

|  |  |  |  |
| --- | --- | --- | --- |
| parentID | childID | births | completes |
| NA | 1 | 0.1141938 | 14.9441 |

The next function, *familyTree()*, creates all the generations that exist in the branching process. We use three parameters in this function: *λ*, *κ* and *maxGen*. *λ* and *κ* are the same as defined in *genKids()*. *maxGen* is used to restrict the maximum number of generations, a family can have, thereby prohibiting the branching process from running indefinitely. *familyTree()* uses *genKids()* as a nested function to generate children for every generation in the tree and stores the children’s data frame in a variable named *nextGen*. *familyTree()* runs until *nextGen* reaches *maxGen* or *nextGen* becomes null, whichever comes first. Table 2 shows the output of *familyTree(0.5, 0.3, 3)*. The job in Generation 1 creates four jobs in Generation 2. The first, second and third jobs from Generation 2 produce seven, one and three child jobs, respectively, in Generation 3. The fourth job in Generation 2 has no children of its own. The birth and completion time for all the children are also displayed in Table 2. Since *maxGen* is set to 3 in this function, the branching process does not continue beyond Generation 3.

Table 2. Example output of *familyTree(0.5, 0.3, 3)*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Gen 1 | parentID | childID | births | completes |
|  | NA | 1 | 0 | 7.730216 |
| Gen 2 | **parentID** | **childID** | **births** | **completes** |
|  | 1 | 1 | 1.630282 | 10.659611 |
|  | 1 | 2 | 4.801337 | 11.583503 |
|  | 1 | 3 | 5.304041 | 9.519773 |
|  | 1 | 4 | 6.340001 | 13.879598 |
| Gen 3 | **parentID** | **childID** | **births** | **completes** |
|  | 1 | 1 | 3.114654 | 10.87029 |
|  | 1 | 2 | 5.266424 | 11.75423 |
|  | 1 | 3 | 5.477547 | 19.01639 |
|  | 1 | 4 | 6.235651 | 14.72795 |
|  | 1 | 5 | 6.322852 | 11.85526 |
|  | 1 | 6 | 8.734475 | 11.59822 |
|  | 1 | 7 | 9.859066 | 10.68623 |
|  | 2 | 8 | 8.02407 | 12.02093 |
|  | 3 | 9 | 6.530571 | 11.14012 |
|  | 3 | 10 | 7.33153 | 10.22969 |
|  | 3 | 11 | 8.63974 | 10.73554 |

The output in Table 2 can then be used to create a visualization of the random branching process, such as the one shown in Figure 2.

## **Results**

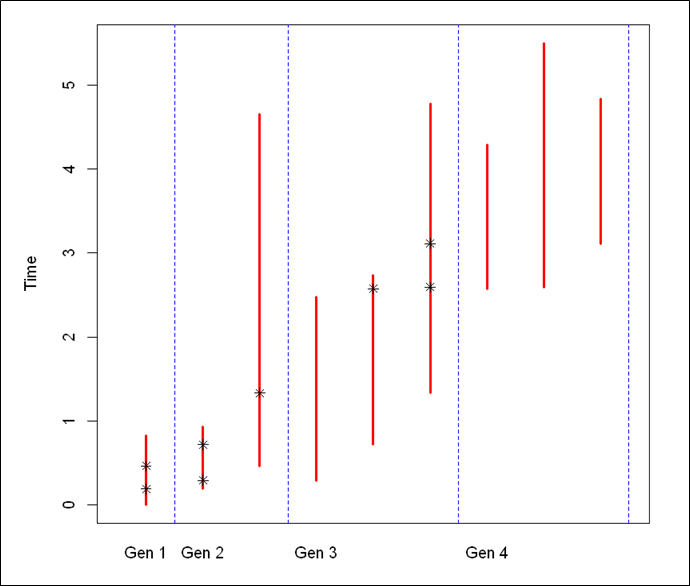
In our study, we set *λ* and *κ* at 0.5 and 0.3, respectively, in *genKids()*. The function *familyTree()* uses *λ* = 0.4, *κ* = 1 and *maxGen* = 10. With a seed value of 12062013, *familyTree()* produces an output as displayed in Table 3.

Table 3. Final output of *familyTree(0.4, 1, 10)* for visualization

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Gen 1 | parentID | childID | births | completes |
|  | NA | 1 | 0 | 0.8244658 |
| Gen 2 | **parentID** | **childID** | **births** | **completes** |
|  | 1 | 1 | 0.1908525 | 0.9280697 |
|  | 1 | 2 | 0.4631948 | 4.6525607 |
| Gen 3 | **parentID** | **childID** | **births** | **completes** |
|  | 1 | 1 | 0.2921203 | 2.479967 |
|  | 1 | 2 | 0.7177455 | 2.73741 |
|  | 2 | 3 | 1.3348405 | 4.7825 |
| Gen 4 | **parentID** | **childID** | **births** | **completes** |
|  | 2 | 1 | 2.574217 | 4.291288 |
|  | 3 | 2 | 2.593132 | 5.500965 |
|  | 3 | 3 | 3.111784 | 4.834751 |

This branching process has four generations. We have one job in the first generation, two in the second and three in both the third and fourth generations. As none of the jobs in the fourth generation had a child, the entire branching process ends after the fourth generation.

Figure 3. Transposed graph of random branching process



Based on the branching process in Table 3, we create a visualization as shown in Figure 3. In this graph, we have generations plotted on *x*-axis and time on *y*-axis, which is a transposition of the graph in Figure 2. The dashed blue lines separate generations from each other. Each vertical red line indicates a single job in the process. The total number of jobs in the process are represented by the total number of red lines which is equal to the total number of rows in Table 3. The black stars on the red lines denote the birth of a child job. It should be noted that the number of red lines in a generation (except Generation 1) matches the number of stars in the previous generation. Moreover, the origination of a red line in a subsequent generation always matches with the time point of a black star in the previous generation. None of the red lines in Generation 4 has a black star, indicating no children were born in this generation. As a result, the random branching process ends at this level.

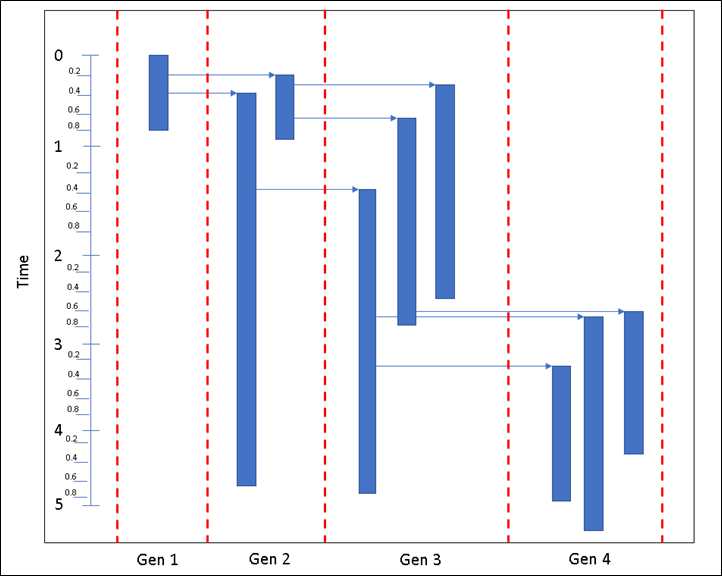
## **Conclusions**

Our work produces a tabular as well as a graphical representation of a branching process. The output showcases all the jobs across generations, identifies their relationships and calculates their start and end times. The visualization offers a transposed version of the original figure shown in Nolan and Temple Lang’ s book. This alternative visualization is easier to follow because the generation chronology flows from left to right, which is more natural than from bottom to top. The use of colors in the graph helps the reader to quickly distinguish between generations. Overall, the visualization has achieved our objective and demonstrates our thorough understanding of a branching process mechanism and the code required for creating the graph.

## **Future Improvements**

Creativity has no limit and being a key component in data visualization, it widens the scope to for improvement. Since branching processes resemble tree structures, it is ideal to present a branching process with a tree visualization. The biggest challenge in a tree visualization would be the absence of any conventional axis system which is essential to present length of a process in terms of time. Moreover, both variables, generation and time, cannot be presented on the same axis.

Figure 4. Proposed improvement



However, as shown in Figure 4, adding arrows from the parent job to origination of a child job would be a more precise indication for the birth time of a child. Addition of a minor scale for time is also suggested as it helps in visualizing smaller differences in time.

## **References**

1. Chapter 6: Branching Processes: The Theory of Reproduction. The University of Auckland, New Zealand. https://www.stat.auckland.ac.nz/~fewster/325/notes/ch6.pdf
2. Deborah Nolan and Duncan Temple Lang, “Case Studies in Data Science with R”. University of California, Berkeley and University of California, Davis. 2015, pp. 277-299. http://www.rdatasciencecases.org

## **Appendix – R Code**

genKids = 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"))

))

}

genKids(0, 1, NA, 0.5, 0.3)

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 = genKids(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)

}

familyTree(0.5, 0.3, 3)

seed1 = 12062013

set.seed(seed1)

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

tree

set.seed(seed1)

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

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

numFamily = sum(sapply(g, nrow))

maxLife

numFamily

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 )

seed1 = 12062013

set.seed(seed1)

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

g

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

numFamily = sum(sapply(g, nrow))

maxLife

numFamily

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

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

axes = FALSE)

box()

axis(2)

numGen = length(g)

numKids = sapply(g, nrow)

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

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

axis(1, at = birthNum + 1,

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

mapply(function(gen, nextGen, birthNum) {

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

segments(y0 = gen$births, y1 = gen$completes, x0 = birthOrder,

lwd = 3, col = "red")

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

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

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

pch = 4)

},

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