Unit 12 Case Study: Simulation Study of a Branching Process

Damon Resnick
Collaborators: Rajni Goyal and Jim Hosker

April 5, 2018

Abstract

In this study simulations are run of the branching of programs to run and complete jobs. In this process, parent programs are generated to complete jobs and spawn other programs to complete the next set of jobs. The number of generation of programs, how many jobs they complete, the number of unused programs, and the completion times of these processes are looked at and compared. It was found the found that a slow birth rate and a medium lifetime are best for optimizing the code.

1 Introduction

This case study focuses on simulating a series of jobs and programs to complete in order to understand automation processes better. This will in turn help to write better code in the future to more efficiently complete certain types of automated tasks. A basic family tree structure is used to simulate the creation of parent and offspring programs to complete simulated jobs. More details about the code and the motivations are outlined in chapter seven of "The R Series" text book "Data Science in R A Case Studies Approach to Computational Reasoning and Problem Solving" by Deborah Nolan and Duncan Temple Lang [1].

1.1 Basics of the Simulations

The basics of the simulations used in this study revolve around the generation of the completion times of the jobs that a parent generates to complete some overarching task that requires the processing of many different jobs. Figure 1 shows the start of a basic family tree simulation. The simulation of completing the task starts by generating a parent to complete the first job and spawn other jobs that need to be done to complete the task after the first job. When the parent generates these new jobs, we can think about them as offspring, the parent also generates the time it will take for the offspring to complete that job. Because it is a simulation some offspring that are simulated to have been created after the parent has completed its job are never actually used. Since the parent no longer exists or is "assassinated" the offspring never make it into the next generation. We will call these unused offspring, unborn offspring.

The offspring that are generated before the parent is assassinated become parents themselves in the 2^{nd} generation. Each of these offspring now perform the same processes as the first parent. They complete a job and spawn other jobs/offspring to complete. They also may have unborn offspring as well.

There are two basic parameters that are used to randomly generate the start and end times of these offspring, are lambda (λ) and kappa (κ). Lambda can be thought of as the rate of births of jobs/offspring, while $1/\kappa$ is the expected run or end time for a job/offspring. The details of the math behind the parameters is outlined in the text book as well as by Aldous and Krebs [2]. The basic thing to note is that a small lambda means there will be births often, or many births, while a small kappa means the lifetime or job completion time will be large. Both parameters are used to generate a random value from a distribution of values. The births are generated with a Poisson process while the lifetimes are generated randomly over an exponential distribution [1, 2].

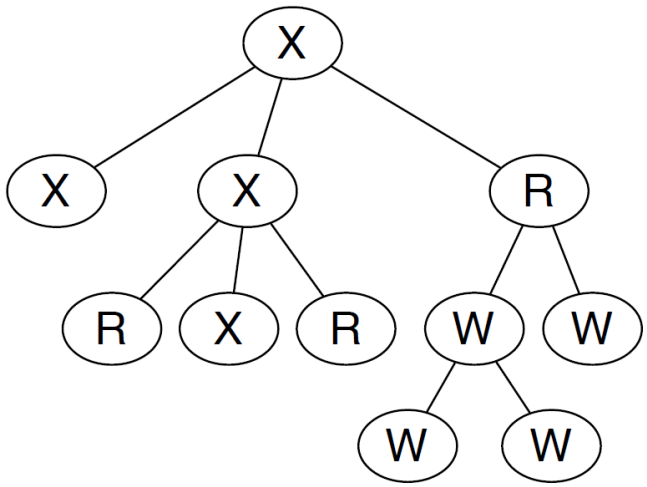

Figure 1: An example of the first four generations of a family branching process, taken from the text and originally from the paper by Aldous and Krebs [2]. You can see one parent at the top generates three offspring that try to complete jobs. Two of these first three offspring, Xs, have completed their jobs. One of those generates offspring that both finish a job and start to run two other jobs. While one of the original offspring, 2nd generation, is still running, R, and has generated two offspring that are waiting, W, to run. The tree in this figure is still running in this diagram.

The last step in this process is knowing how to end the simulations. Because some simulation may end quickly while others continue until stopped externally. In this case a simulation will end if it ends on its own, reaches 200 generations, or 100,000 offspring. For the cases we simulate here, 200 generations are never reached before 100,000 offspring are generated. So a process either dies on its own or reaches the 100,000 offspring limit.

2 Analysis of the Summary Statistics, Question 9

In the text the summary statistics used to help the analysis of the simulations were the Number of Generations versus the Number of Offspring. Figure 2 shows this for four sets of lambda and kappa values. We will attempt to delve further into this analysis by looking at two other summary statistics Mean Completion Time and Total Number of Unborn Offspring. The Mean Completion Time is just the average lifetime of the all the jobs, and the Total Number of Unborn Offspring is the total number of offspring that did not become parents after their parent was assassinated

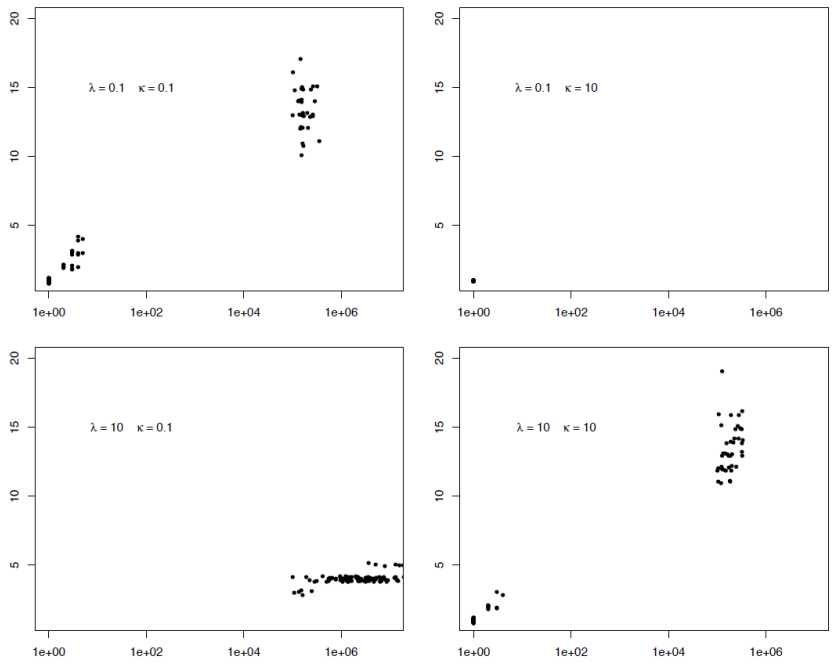

Figure 2: This figure from the text book shows the Number of Generations vs. the Number of Offspring or Jobs Completed for 4 different combinations. Each dot represents 1 of 100 simulations.

The first thing to note about Figure 2 is that very similar outcomes are achieved for ($\lambda = 0.1$, $\kappa = 0.1$) and ($\lambda = 10$, $\kappa = 10$). As you can see both these outcomes show a handful of simulations only getting to a few generations with a great many more simulations make it beyond 10 generations.

In the first case, ($\lambda = 0.1$, $\kappa = 0.1$), there is a high birth rate and a long lifetime which will enable lots of offspring to be born that live long enough to complete jobs and make other generations. This makes a lot of sense. There seems to be a direct relationship between generations and offspring here.

For the case of, ($\lambda = 10$, $\kappa = 10$), the births come slowly, and lifetimes are short. This seems a little odd but since the time scales are not known this is possible. There seems to be a direct relationship between generations and offspring here.

For the case of ($\lambda = 0.1$, $\kappa = 10$) we have many births but short lifetimes. For some reason this resulted in only one generation for all simulations. Not a desirable outcome to say the least.

For the case of ($\lambda = 10$, $\kappa = 0.1$) we have slow birth rates and long lifetimes. These simulations resulted in a large number of jobs being created by only a few generations of offspring. This makes a lot of sense because with a slow birth rate and a long lifetime there could be a great deal of offspring that then have a long time to wait until they start a job. The offspring need to wait until the parent is finished running to start their jobs.

Now that we have a feel for the information that the summary statistics already give us we generate several other graphs like Figure 2 that show the new summary statistics. Figure 3 shows the number of generations versus mean completion time for the same random seeds used for Figure 2 from the text. We can immediately see that again the top left and bottom right graphs are very similar. However, the most interesting part is that since the bottom right simulations have a large kappa then the lifetimes will be much shorter. Since the outcomes are nearly the same I would assume that if you wanted to optimize your code between these two combinations of lambda and kappa you would choose a large lambda and kappa over both being small in order to minimize the total run time.

The mean completion times were generated in the function exptOne():

```
vcomp = c(0)
pcomp = c(0)
for(i in 1:length(aTree)){
  vc <- aTree[[i]]$completes
  pa <- aTree[[i]]$parentAssin
  vcomp = append(vcomp,vc)
  pcomp = append(pcomp,pa)
}
  mcomp <- mean(vcomp)
```

A loop was created in exptOne() to generate the mean completion times for each job.

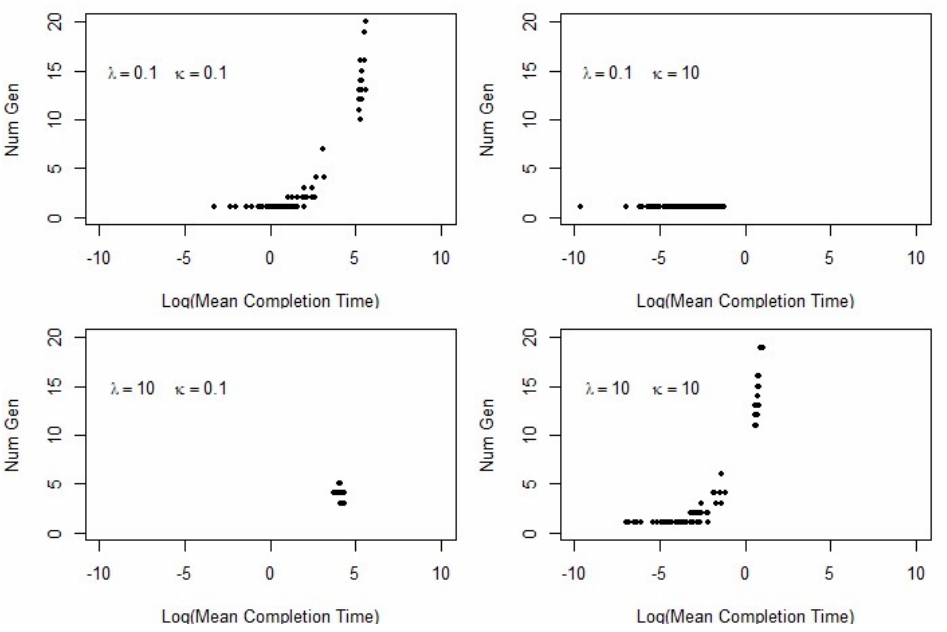

Figure 3: This figure shows the Number of Generations vs. Mean Completion Time for 4 different combinations of lambda and kappa. The x-axis is on a log scale.

Let's explore this a bit further and see if we can find the best set of lambdas and kappas to complete the task in the most efficient way. Figure 4 shows the Number of Jobs versus the Mean Completion

Time. Remember number of jobs is the same as the number of offspring. We can again see a similar result between the top left and bottom right, but now suddenly, the bottom left graph jumps out at us! Clearly the number of jobs completed on average for the bottom left is higher than the other two. However, the completion times are still much higher than the bottom right, so I believe it is now easy to say that simulations with $\lambda = 10$ and $\kappa = 10$ generally out-performs the other simulations with different combinations of these two parameters, but a slightly smaller kappa may be more efficient.

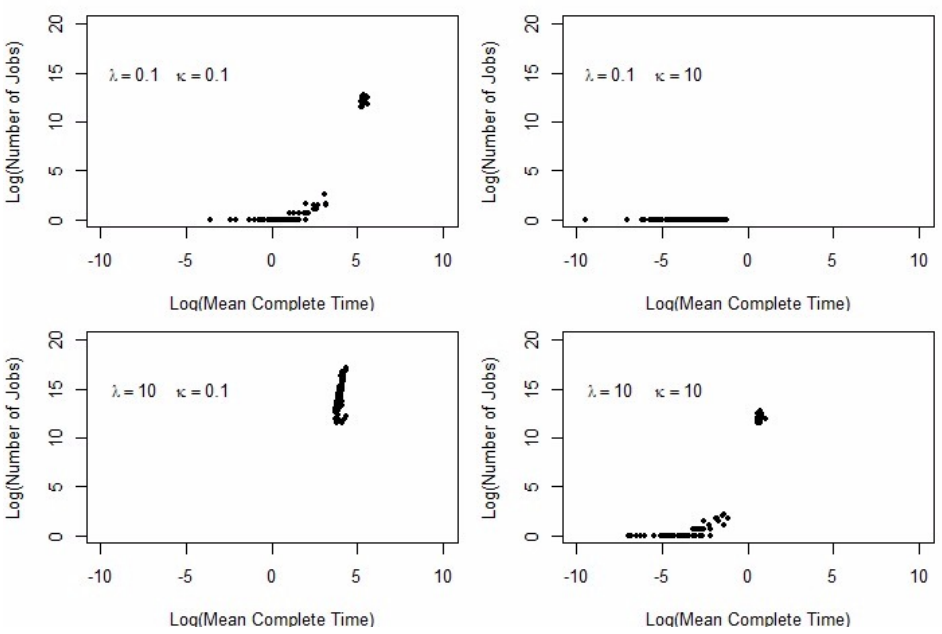

Figure 4: This figure shows the Number of Jobs vs. Mean Completion Time for 4 different combinations of lambda and kappa. The x and y axes are on a log scale.

Remember the bottom left case, ($\lambda = 10$, $\kappa = 0.1$), has a slow birth rate but a long lifetime. In a sense, slow a steady win's certain kinds of races. And those simulations complete the most jobs. So, if you were looking for the parameters to complete the most jobs the most often you would probably go with this combination. However, if you were feeling lucky you may decide to go with $\lambda = 10$ and $\kappa = 0.1$ because it is more likely to finish much faster. For that matter you may go with a large lambda of 10 but make the kappa value somewhere between 0.1 and 10.

The last summary stat we looked at was the Number of Unborn Offspring. Remember that the number of unborn offspring is simply the total number of jobs that did not get generated because their start times were after their parents had died. They were simply not born. This summary statistic was just a curiosity that we came up with hoping it might tell us something interesting. Figure 5 shows the relationship of the number of unborn offspring with the number of jobs. You can see that there is a basic linear relationship between the number of unborn offspring and the number of jobs/offspring. This makes sense because the more jobs there are the more unborn there should be. The slope of this relationship looks the same for the top left and bottom right cases but looks smaller for the bottom left case. The slower birth rate in the bottom left coupled with the long lifetimes makes for fewer unborn per those not unborn.

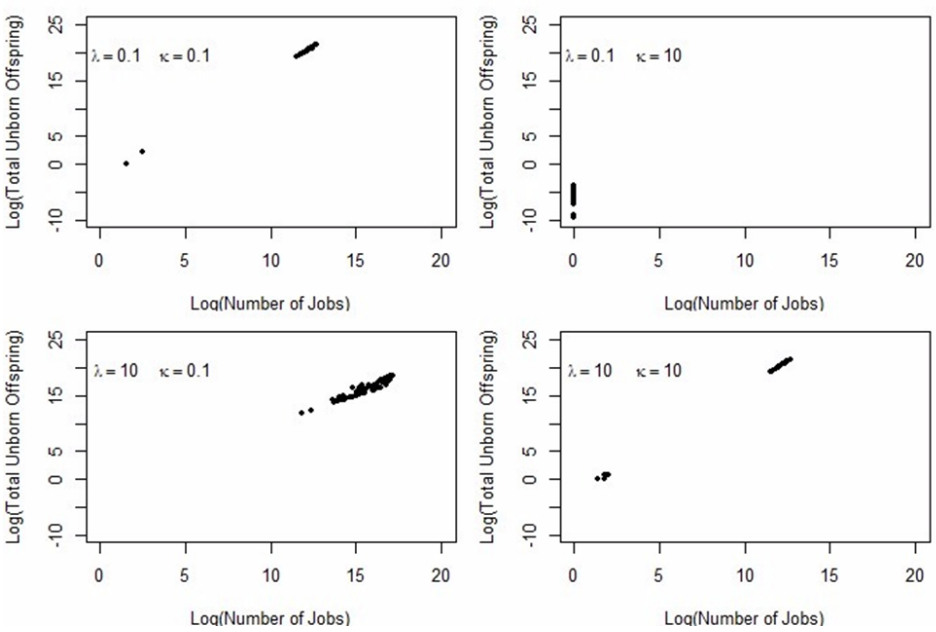

Figure 5: This figure shows the Number of Unborn Offspring versus the Number of Jobs for 4 different combinations of lambda and kappa. The x and y axes are on a log scale.

The snippet of code bellow shows how we added to the code to create the unborn numbers. This bit of code was added in a loop for the familyTree function used in exptOne():

```
parentAss = length(unique(allGens[[ (i - 1) ]]$kidID)) -
    length(intersect(unique(allGens[[ (i - 1) ]]$kidID) ,unique(nextGen$parentID)))
  allGens[[ i ]]$parentAssin =  parentAss
```

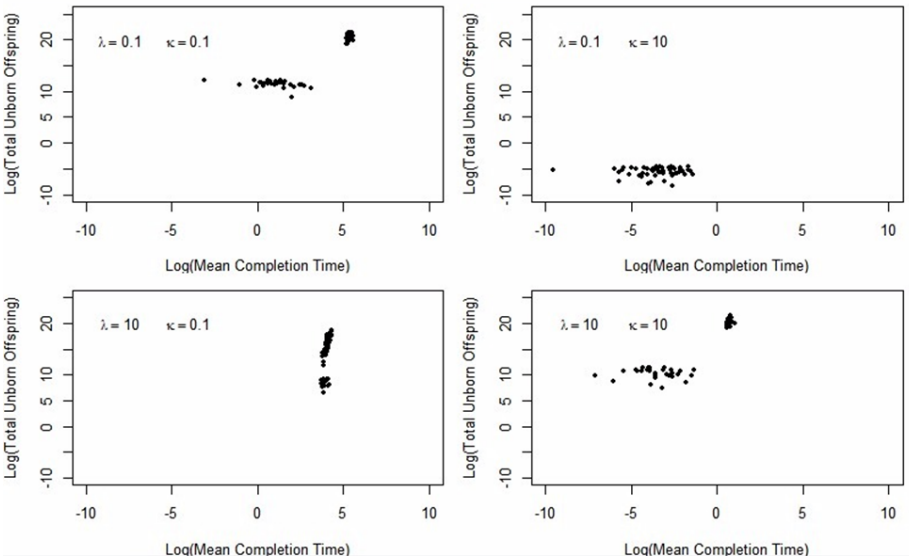

Figure 6: This figure shows the Number of Unborn Offspring versus the Mean Completion Time for 4 different combinations of lambda and kappa. The x and y axes are on a log scale.

The number of unborn were calculated simply using the total number of kids generated in one generation minus the number of parents in the generation. The unborn kids are defined as this difference.

Looking at the unborn versus mean completion time we see again how efficient and quick the bottom right case is while as expected the bottom left has the largest completion times. Figure 6 shows this in detail. These sets of graphs are very similar to Figure 4 as the number of unborn offspring are linearly related to the number of jobs.

5 Conclusion

In this study simulations were run of the branching of programs to run and complete jobs. In this process, parent programs were generated to complete jobs and spawn other programs to complete the next set of jobs. The number of generations of programs, how many jobs they complete, the number of unused programs, and the mean completion times of these processes were looked at and compared. We found that the $\lambda = 10$ and $\kappa = 0.1$ paring seems to guarantee the most jobs completed but since it does not take advantage of using a large number of generations it cannot complete these jobs quickly. Clearly a slow birth rate is desirable but a kappa value between 0.1 and 10 is probably optimal to maximize jobs completed in the fastest amount of time. The new summary statistics both confirm the earlier findings and offer new insights.

References

[1] "The R Series" text book "Data Science in R A Case Studies Approach to Computational Reasoning and Problem Solving" by Deborah Nolan and Duncan Temple Lang, CRC Press, 2015.

[2] "The 'Birth-and-Assassination' Process", by David Aldous and William Krebs. Statistics and Probability Letters, 10:427–430, 1990.

Appendix A: Code

Listed here is the code that was changed or added in order to answer the questions. The rest of the code is in an attached .txt file.

Please see attachment for entirety of code.

The mean completion times were generated in the function exptOne():

```
 vcomp = c(0)
 pcomp = c(0)
 for(i in 1:length(aTree)){
   vc <- aTree[[i]]$completes
   pa <- aTree[[i]]$parentAssin
   vcomp = append(vcomp,vc)
   pcomp = append(pcomp,pa)
 }
   mcomp <- mean(vcomp)
```

A loop was created in exptOne() to generate the mean completion times for each job.

The snippet of code bellow shows how we added to the code to create the unborn numbers. This bit of code was added in a loop for the familyTree function used in exptOne():

```
parentAss = length(unique(allGens[[ (i - 1) ]]$kidID)) -
     length(intersect(unique(allGens[[ (i - 1) ]]$kidID) ,unique(nextGen$parentID)))
   allGens[[ i ]]$parentAssin =  parentAss
```

The number of unborn were calculated simply using the total number of kids generated in one generation minus the number of parents in the generation. The unborn kids are defined as this difference.