Modeling Subsystem Reliability

Brennon Brimhall

July 18, 2018

1 The Problem

You're interested in building a robot that will reliably execute its designed functions in a match. But how robust and repairable do you need to make a given subsystem?

First, let's diagram a subsystem.

$$\begin{tikzcd}
w \arrow[loop above] \arrow[rr, bend left] & & b \arrow[loop above] \arrow[ll, bend left]
\end{tikzcd}$$

In this diagram, we have two states – which, while perhaps a bit simplified, is still useful – that indicate a subsystem working (state w), and the subsystem being broken (state b). The arrows between states represent transitions - the robot breaking, getting fixed, staying broken, or staying working. To complete our model and use it for analysis, we need to assign probabilities on each of the transitions. These probabilities are generated by considering a "chunk" or "bucket" of time. It could be a second, or a minute, but in the context of FRC, using the length of a match (2.5 minutes) is pretty convenient. And one more thing: the transitions from each state should sum to 100%.

Let's work with an example – let's talk about a drivetrain. For this example, we'll say that there's a 5% chance a drivetrain will break in a match. It's a pretty good drivetrain; notice that most events have a maximum of about 13 qualification matches (about 22 total matches if you tie in each eliminations series). Let's also say that this drivetrain isn't easily fixable. Maybe it takes, on average, 30 minutes to repair a break there. Let's further suppose that fixing the drivetrain follows a geometric distribution. Then, the probability of fixing the drivetrain in a 2.5 minute window is $\frac{1}{12}$, or about 8.3%. We'll put these weights on our state diagram.

$$\begin{tikzcd}
w \arrow[loop above, ".95"] \arrow[rr, bend left, ".05"] & & b \arrow[loop above, ".917"] \arrow[ll, bend left, ".083"]
\end{tikzcd}$$

So how often are we broken in this example? Well, once we crunch the numbers, it turns out that we're broken about 37.6% of our event's matches – or, in other words, we expect that our drivetrain is broken for about 71 minutes over the course of the tournament. "But our drivetrain shouldn't break in any matches!" you might say. "We've said that it's 95% reliable!" It turns out that that's not quite right based on some assumptions we've made. Let's dig in a bit.

2 The Math

The model I've used here is called a *Markov chain.* Simply put, this model has some finite number of states (two, in our case), and we iterate through them based on the probabilities of each transition. We assume that these state transition probabilities don't change over time.[1]

For example, say that we're working; let's represent this as the vector $< 1, 0 >$. If we follow the transitions, we'll end up saying that we're working 95% of the time, and broken 5% of the time. Let's represent this by the vector $< .95, .05 >$. Another iteration of arrow following, and we get $< .87, .13 >$, then $< .84, .16 >$, and so on. If we keep on iterating, we'll find that we converge on about $< .624, .376 >$.[2]

I won't go into the proof here, but there's a theorem that says that any Markov chain will always converge to some steady state. We can find this steady state easily by solving a system of linear equations:

$$\begin{bmatrix} .95 & .083 \\ .05 & .917 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix}$$

The matrix form might be confusing to some of you, so let's write this in something a bit more familiar:

$$.95x + .083y = x$$
$$.05x + .917y = y$$

We also know that we're dealing with probabilities, so the probability of being in each state need to sum to 1:

$$x + y = 1$$

We can solve this system by using a bit of algebra:

[1] If you're interested in learning more, I suggest digesting the introduction available at `http://setosa.io/ev/markov-chains/`. Unfortunately, the Wikipedia article on Markov chains is pretty thick.

[2] Fans of linear algebra and matrices will notice that each iteration is just a single matrix multiplication:

$$\begin{bmatrix} .95 & .083 \\ .05 & .917 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} .95 \\ .5 \end{bmatrix}$$

$$.05x = .083y$$

$$x = 1 - y$$

$$.05(1 - y) = .083y$$

$$.05 - .05y = .083y$$

$$.05 = .133y$$

$$y = \frac{.05}{.133} = .376$$

$$x = 1 - y = .624$$

This agrees with our repeated matrix multiplication. So it looks like we were right after all.

Now, if you're like me and want to tweak some of the probabilities in our model, solving that system is a lot of work we're going to be repeating each time. If we keep our Markov chain to only two states, and repeat the above algebra using the variables a (for probability we will be transition from broken to working) and b (for probability we transition from working to broken), we will find that, in general, the steady state is:

$$\begin{bmatrix} \frac{a}{a+b} \\ \frac{b}{a+b} \end{bmatrix}$$

What we just did was finding an eigenvector of our matrix of transitions – specifically, the eigenvector associated with the eigenvalue of 1. Every properly constructed Markov chain transition matrix will have this eigenvalue; it's part of the proof I ommitted. For those who aren't quite sure what I'm talking about, eigenvalues and eigenvectors are advanced math concepts that are pretty cool. They kind of show up everywhere once you know how to look for them, such as Google's PageRank algorithm, state-space controllers in FRC, image compression, and more.

3 The Analysis

So, you may feel differently, but I would like a drivetrain that's a bit more reliable at competition. Let's assume that we're trying to make sure that a given breakage won't keep us from being at our full potential in our next match. So that we can be a bit less abstract and a bit more concrete, let's assume that we're playing at the 2018 Tech Valley Regional with Team 20's schedule. On average, they had about 4.5 matches in between qualification match appearances. The tournament had 76 total qualification matches.

Using our existing model, we can play around with the probability of our drivetrain breaking. If we want to decrease the estimated time spent fixing the drivetrain to be within the 11.25 minute window we've given ourselves, we need to make sure our drivetrain only breaks 0.5% of the time, or about 100 times less prone to breaking.

Maybe we should look at making our drivetrain more repairable. If we want to decrease the estimated time spent fixing the drivetrain to be in that same 11.25 minute window, we need to make sure our drivetrain is has an 83% chance of being repaired in a 2.5 minute window. That's a full 10 times more repairable.

I'm not certain that either of those two scenarios are likely – but what happens if we increase both robustness and repairability? As it turns out, if we can make our hypothetical drivetrain only break 1.5% of the time (a factor of 3.3) and make our drivetrain able to be repaired in 10 minutes, on average (an improvement by a factor of 3), we succeed. We'll only spend about 10.8 minutes fixing it over the tournament, on average. In general, this approach – increasing both robustness and repairability – is the best way to ensure overall robot reliability.[3]

[3] "But what about preventative maintenance?" "What about a real robot that's composed of multiple subsystems?" "What if I have multiple failure modes within a subsystem that have varying times to be repaired?" Those are all good questions, and the simple answer is that they require complicating our model. You can add a state for preventative maintenance, and you can add states for ever combination of failure mode (approximately 2^n states for n failure modes). As each state results in an equation to our matrix, this can get pretty hairy by hand. I'd suggest something like Octave, MATLAB, or Mathematica if you decide to go down this route.