Boiler Tube Pulling
An Example of Adaptive Maintenance

Jack Devanney
Center for Tankship Excellence, USA, djw1@c4tx.org

Abstract

Learning is at the core of many (if not all) maintenance problems involving test and replacement of parts. That's why we call it a test. Often the results of a test give us valuable information about the condition of the remaining parts. This paper combines Bayesian reasoning with backwards recursion to develop optimal adaptive testing policies for a range of such problems.

Keywords

Adaptive Maintenance, Optimal Stopping, Bayesian test and replacment, dynamic programming.

1 Pulling Boiler Tubes

In an earlier life, I was a tanker superintendent, responsible for the quadrennial drydocking and overhaul of very large, steam turbine powered tankers. The biggest single maintenance problem in a turbine engine room is the boiler. The boiler is made up of about 1200 tubes carrying 60 bar steam and water. In general, not all and usually very few of the tubes need replacement. However, one cannot determine for certain whether a tube needs replacement without going through the laborious and expensive process of pulling it, that is, removing the tube and inspecting the inside. More importantly, this was the only way to gain information on the overall condition of the boiler. The issue was: when should we stop pulling tubes based on the results of the tubes pulled so far?

Learning is at the core of many maintenance problems. Before opening up a machine, we often have little or no knowledge about its condition. The way we learn about the machine's condition is to remove and test parts. Such tests can be expensive. In some cases, the removal or test is destructive. But if a defective part is not replaced, it will fail in service, in which case the cost is orders of magnitude larger than replacing it. This paper addresses a sub-class of such problem in which we are willing to assume the parts are identical in the sense that, whatever the probabilities are for one part, they are the same for the other parts.

We will begin by assuming that we are willing to characterize the condition of a part as Good or Bad, that is, we have a simple Pass/Fail test.

2 Some Bayesian Preliminaries

We start by assuming Nature is creating bad tubes for this boiler with probability δ. The problem is we don't know and can't know what δ Nature is using. But we can obtain information on δ by pulling tubes and determining whther those tubes are bad or good.

We will take a Bayesian approach. For a Bayesian, δ is just a random variable for which we need a prior density, which we will then update according to Bayes Rule as we observe the results of the tube pulling. It is crucial in a problem like this to differentiate between

1. The unknown and unknowable probability, δ, that Nature is using to generate bad tubes. This is an unchanging state of Nature, which we cannot observe directly.
2. ***Our*** probability density on this state of Nature, $f(\delta|H)$ where H is our current state of knowledge about δ, which will change as the tube pulling results come in.

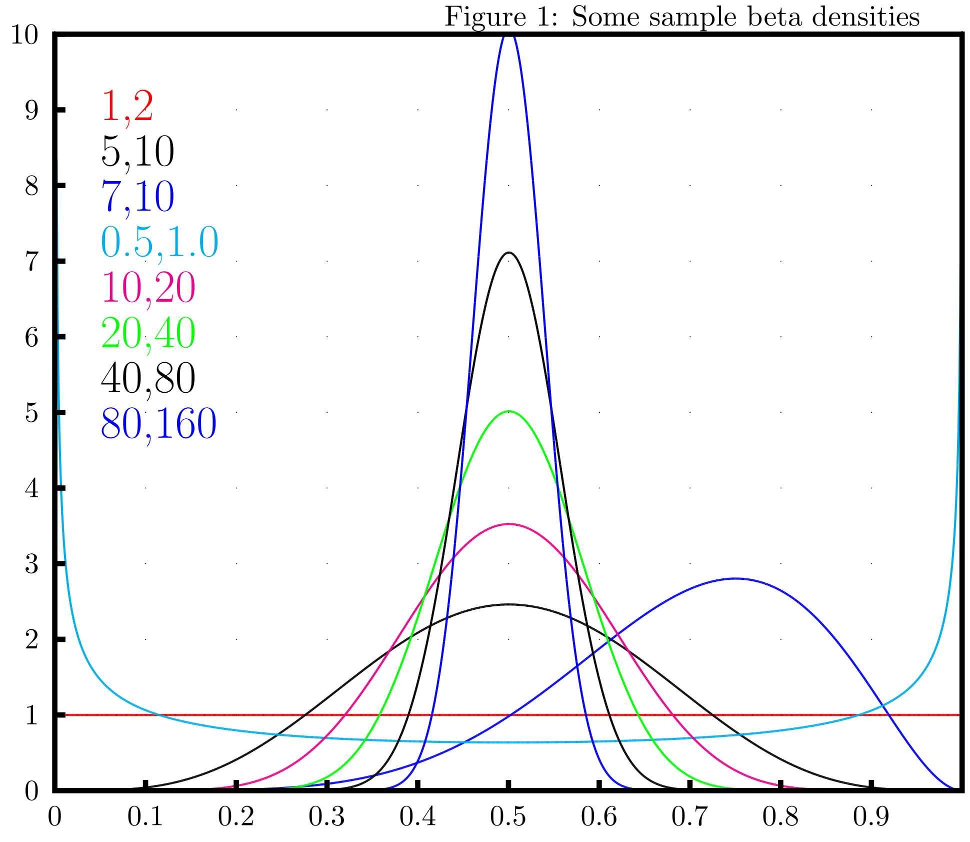
To help us keep these two entirely different probabilities separate, we denote the unknown state of Nature, δ, with a blue Greek letter.

Things become particularly simple, if we are willing to assume that our prior density on δ, before observing any tubes for this boiler is a member of the Beta family.

$$f_\beta(\delta|k',n') = \frac{(n'-1)!}{(k'-1)!(n'-k'-1)!}\delta^{k'-1}(1-\delta)^{n'-k'-1} \quad 0.0 < \delta < 1.0$$

The Beta is a two parameter family of densities. By varying these two parameters just about any smooth, unimodal density on the interval [0,1] can be approximated. A few of these densities are shown in in Figure 1.

Figure 1: Some sample beta densities



If before pulling any tubes, our density on δ is $f_\beta(\delta|k',n')$, and we pull n tubes and observe k_b defectives, then application of Bayes Rule reveals that our new density on δ is also a Beta; but the Beta with parameters, $(k'+k_b, n'+n)$. By starting out with a Beta, our prior feelings and the test data combine in a remarkably simple and natural manner.

Of course, what we really need in the tube pulling prolem is not our probability density on δ; but ***our*** probability that the next tube is defective. Elementary probability informs us that, if our current density on δ is $f_\beta(\delta|k'+k_b, n'+n)$, then our current probability that the next tube is bad is simply $(k'+k_b)/(n'+n)$.

More generally, if our current density on δ is $f_\beta(\delta|k'+k_b, n'+n)$, our current probability that we will have K defective tubes in the next N pulled is something called the Betabinomial

$$f_{\beta B}(K|N, k' + k_b, n' + n) = \frac{N!(K + k' + k_b - 1)!(N + n' + n - K - k' - k_b - 1)!}{K!(N - k_b)!(N + n' + n)!} \quad K = 0, 1, ...N$$

The mean of the Betabinomial, our average number of defectives in the next N pulled, is simply $N(k' + k_b)/(n' + n)$.

3 Our Decision Tree

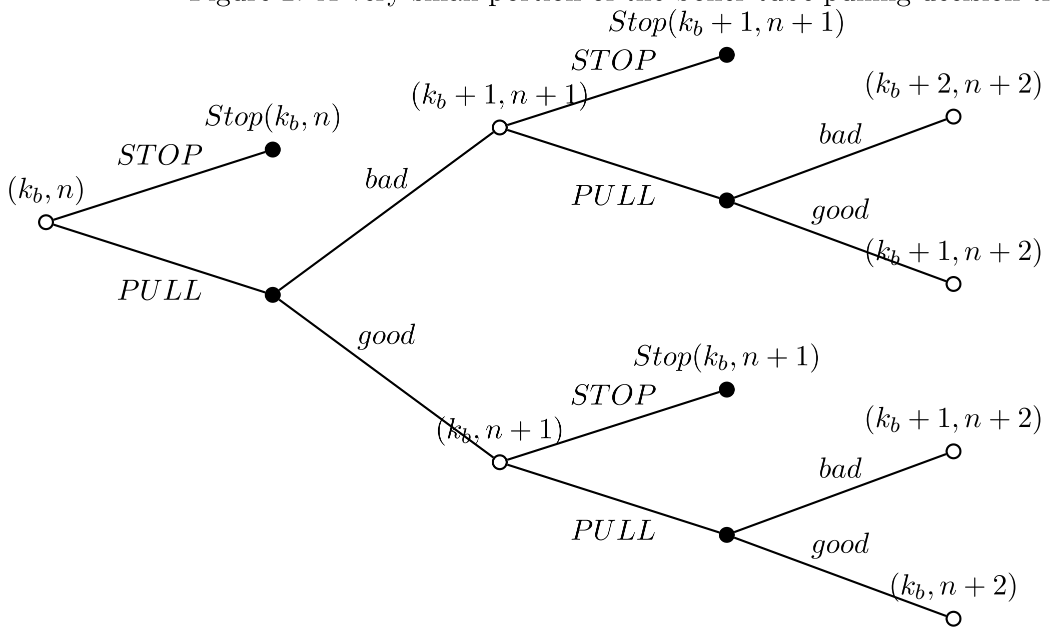
Back to boilers. Now one boiler is much like another, and we had seen lots of boilers. But each boiler has its own life history, its own set of circumstances; its own possible occasions of abuse. ***And we wanted to be conservative.*** We took the wishy-washy attitude that, before seeing any tubes, we had no reason to believe that any of Nature's possible δ's was more likely than any other. We assumed when each ship came into the yard for its third drydocking at age 15 that, δ, was equally likely to be any number between 0.0 and 1.0. This uniform density is shown in red in Figure 1. It is the Beta density with parameters, (1,2).[1]

We had a fairly good estimate of the cost of pulling and replacing a tube, given that the boiler was already opened up. At the time it was about \$200. If a tube failed in service, the boiler had to be shut down for at least 12 hours before it was cool enough for the crew to go inside, find the failed tube and plug it, that is, blank it off. The whole process could take the better part of a day, and the ship would have to operate at reduced power until the next drydocking. These were ships who normally earned about \$50,000 per day. And there was the safety issue. We guestimated the cost of a tube failing in service was \$100,000.

The problem then was to come up with a procedure for deciding after each observation whether to pull yet another tube, or stop and take our chances.

We realized we had a *decision tree* a tiny portion of which is sketched in Figure 2.

Figure 2: A very small portion of the boiler tube pulling decision tree



In a decision tree, our decisions (open circles) alternate with test results (solid circles). At the far left, we are at the node in the tree where we have pulled n tubes and observed k_b bad and have to make a decision. If we stop, we will have to take our chances with the remaining tubes based on our probabilities given k_b and n. We pencil in $Stop(k_b, n)$ as the cost associated with taking this gamble. If we pull another tube, we will pay a cost t, and then Nature will step in and either

[1]The uniform density is also the max entropy (min information) density on the interval [0,1].

show us a bad tube or a good tube, where upon we will have another STOP/PULL decision with a slightly different k_b and n.

If we started out with a uniform prior, we know from Section 2 that ***our*** probability after observing k_b bad tubes in n tries that the next tube is bad is $(k_b + 1)/(n + 2)$ so we can easily fill in ***our*** probabilities on Nature's branches.

4 Backwards is the Way Forward

At this point, we need to make a very important assumption. We assume that our overall goal is to minimize the *average* cost of testing and replacing tubes plus the *average* cost of unreplaced tubes failing in service. This is not an assumption that should be made lightly for it implies that we would not be willing to pay for insurance against the rather substantial losses that we could incur from the tubes failing in service. But our choice of prior was awfully conservative, the ships had two boilers, and we are a large and well-financed tanker owner so even an unusually bad run of tube failures isn't going to break us. With this assumption, our job is to minimize the average or *mean* cost of our decisions.

Stopping is the easy part. Call the the number of tubes in the boiler N. If we stop after replacing n and seeing k_b defective, we know from Section 2, that the number of bad unreplaced tubes will be distributed as a Betabinomial with parameters, $N - n$, $k_b + 1, n + 2$. We know also from Section 2, the average of this density is $(N - n)(k_b + 1)/(n + 2)$. So our average cost of in service failures if we stop at k_b and n is

$$Stop(k_b, n) = \frac{T(N-n)(k_b+1)}{n+2}$$

where T is the cost of a single in service failure.

The interesting part is the "pull again" branch. Our decision tree is enormous but it has a very simple structure. Let's try to take advantage of it. We will call n, the number of tubes pulled, the *stage* variable. After each test, the stage variable increases by 1. We will call k_b, the number bad tubes observed, the *state* variable. After each test, that is at each new stage, the state variable either stays the same or increases by 1. If we somehow knew what the optimal strategy was from stage $(n + 1)$ on *and the average cost of that optimal strategy from* $(n + 1)$ *on*, it would be very easy to decide what to do at stage n.

Let $V(k_b, n + 1)$ be the average cost of following an optimal strategy from $(n + 1)$ on given that at $(n + 1)$ we have seen k_b bad tubes. If we stop at n, our average cost from n on is $Stop(k, n)$. If we pull again, then we will have to pay the cost of pulling that tube t and we will find ourselves in either state k_b or state $k_b + 1$ at stage $n + 1$. But it we find ourselve in state k_b at $n + 1$, we will follow an optimal strategy the rest of the way and the average cost of that strategy is $V(k_b, n + 1)$ Similarly, if we find ourselves in state $k_b + 1$ at $n + 1$, we will do the best we can from there on and the average cost the rest of the way is $V(k_b + 1, n + 1)$. So at n, the average cost of pulling another tube and then doing the best we can whatever happens is

$$Pull(k_b, n) = t + (k_b + 1)/(n + 2)V(k_b + 1, n + 1) + (n - k_b + 1)/(n + 2)V(k_b, n + 1)$$

At n and k_b, the optimal choice is STOP if $Stop(k_b, n)$ is less than this average; otherwise the optimal choice is to test again. At n we will pick the lower of the two and the average cost associated with that choice from stage n on is $V(k_b, n)$. In symbols.

$$V(k_b, n) = min \begin{cases} T(N-n)(k_b+1)/(n+2) \\ t + (k_b+1)/(n+2)V(k_b+1, n+1) + (n-k_b+1)/(n+2)V(k_b, n+1) \end{cases}$$

This equation relates the average cost of the best we can do from stage n on to the average cost of the best we can do from stage $n + 1$ on. $V(k_b, n)$ is called the *optimal value function* for this problem. At this point, you are probably saying: so what! We don't know what $V(k_b, n + 1)$ is, so we certainly don't know what $V(k_b, n)$ is.

Oh ye of little faith. Ye must learn to think backwards. You must jump forward to stage N, where N is the total number of tubes in the boiler. At that point, we will have pulled and replaced all the tubes. There is nothing more to do, and the cost of going the rest of the way is zero, for there is no rest of the way. In short, $V(k_b, N) = 0.0$ for all k_b.

Now we move back to stage $(N-1)$ plug zero into the V's on the right side of the above equation, and calculate $V(k_b, N-1)$ for all k_b, keeping track of our decision for each k, $D(k_b, N-1)$. We will set $D(k_b, N-1)$ to S if the optimal decision is to stop, and P if it's to pull another tube.

Now we can move back to stage $(N-2)$ plugging $V(k_b, N-1)$ into the right side of above equation and computing $V(k_b, N-3)$ and the corresponding set of optimal decisions. We continue in this backwards manner until we get all the way back to $n = 0$. At this point we have two tables $V(k_b, n)$ and $D(k_b, n)$ for all n from 0 to N and all k_b from 0 to n.

Now it is a trivial matter to start moving moving forward. We begin at $(k_b = 0, n = 0)$, if $D(0, 0)$ is P we pull a tube; otherwise we close up the boiler and send the ship to sea. If we pull a tube, we will get to either $(k_b = 0, n = 1)$ or $(k_b = 1, n = 1)$. we check the corresponding $D(k_b, 1)$ and do what it tells us. Eventually, $D(k_b, n)$ will tell us to stop pulling tubes or we will get all the way to stage N.

It is important to note that this procedure is truly adaptive. Not only does it tell us what to do for any possible state nature might put us in, not only does it account for the change in our probabilities as the test results come in, but in calculating the optimal value table it weights the chances of future learning and the value to be obtained from this knowledge in deciding what to do now.

Admittedly this is a brute force method.[2] Some may worry about the amount of computation involved. Don't, at least not for boilers. For a 1200 tube boiler, on my ordinary desktop computing the tables take 3.5 seconds using completely unoptimized code. This is pretty remarkable when one considers that Nature can send us down 2^{1200} possible paths. 2^{1200} is a number so large that most computers cannot display it. The saving grace, of course, is that at any point in our decision tree, we don't care which path got us there, we only care about how many bad tubes we have seen.

5 Mini-boiler Blues

To make all this math concrete, let's do a boiler. A real boiler has over a thousand tubes and, while a modern computer can handle that in a couple of seconds, a 1000x1000 table is a bit big to put on a page.

So we will do a 20 tube mini-boiler using precisely the same procedure and the same little program that we would use for a full sized boiler. But I have buggered with the costs of pulling and the costs of failure to make the results for the mini-boiler more interesting. The results for a test cost of \$200 and a failure cost of \$1000 are shown in Tables 1 and 2. In these tables, there is a column for each stage, n (the number of tubes pulled), and a row for each state k_b (the number of bad tubes seen). Since k_b can't be bigger than n, the tables take on a triangular form.

Table 1 is the optimal decision table. For these numbers, the optimal policy can be summarized succinctly:

- If you have seen no bad tubes in first 5 pulled, stop.
- If you have seen only 1 bad tubes in first 10 pulled, stop.
- If you have seen only 2 bad tubes in first 14 pulled, stop.
- If you have seen only 3 bad tubes in first 18 pulled, stop.
- Otherwise, pull them all. The long and the short and the tall.

Table 2 shows the optimal value table. If we had just gone ahead and pulled all 20 tubes, the average cost would have been \$4000. With these numbers, the procedure is quite conservative, and going in, we have a high probability of pulling all the tubes. So at (0,0) our average cost is only

[2]This backwards, brute force technique, developed largely by Richard Bellman, is called *dynamic programming.* But this phrase is used in other contexts with completely different meanings. So we will stick with backwards, brute force, despite the fact that "brute force" is unfair to a method that turns 2^{1200} calculations into 1200 x 600.

about \$172 lower than \$4000. Bummer! But if Nature's probability of producing a bad tube really is low, then we have a good chance of seeing no bad tubes in the first few pulled, and the average cost drops rapidly.

Table 3 shows our probability that the next tube is bad for each combination of stage and state.[3]

As would be expected for a uniform prior, before pulling any tubes, our probability starts off at 0.5, but drops quickly if we see no bad tubes in the first few pulled. If we pull 5 tubes and none are bad, our probability is down to 0.045 and it's better to take our chances with the last 15 tubes. But once we seen one or two bad tubes, it's very difficult to get our probability back down to this level, in part because there is a good chance we will see more bad tubes.

In a problem like this, starting off with a uniform prior — essentially throwing away all relevent past experience — guarantees that you will do at least a handful of tests for just about any reasonable set of costs. So you should only use this assumption in situations where you would have done some tests in the absence of any of these Bayesian ruminations. That's why we applied this procedure only to boilers that were over 15 years old, and starting to show some signs of age. We knew we needed to pull some tubes, we just didn't know how many. At that point, a uniform prior is a reasonable, if conservative, choice.

The input to our little table generating program consists of only three numbers:

N The number of parts.

t The cost of inspecting/replacing.

T The cost of failure in service.

Yet the procedure applies to a wide range of maintenance problems. And in situations where you are not prepared to throw away all your hard earned knowledge, you can simply change (k', n'), your initial prior parameters, from (1,2) to something that more closely matches your experience.

[3]I say again. ***These are not Nature's probability of making a tube defective. Nature's probability is a single, unchanging number which we cannot observe directly.***

Table 1: Optimal Decision Table: Number of tubes = 20 Test cost= 200 Failure cost= 1000

STATE	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	P	P	P	P	P	P	S	S	S	S	S	S	S	S	S	S	S	S	S	S	S
1		P	P	P	P	P	P	P	P	P	P	S	S	S	S	S	S	S	S	S	S
2			P	P	P	P	P	P	P	P	P	P	P	P	P	S	S	S	S	S	S
3				P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	S	S
4					P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	S
5						P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	S
6							P	P	P	P	P	P	P	P	P	P	P	P	P	P	S
7								P	P	P	P	P	P	P	P	P	P	P	P	P	S
8									P	P	P	P	P	P	P	P	P	P	P	P	S
9										P	P	P	P	P	P	P	P	P	P	P	S
10											P	P	P	P	P	P	P	P	P	P	S
11												P	P	P	P	P	P	P	P	P	S
12													P	P	P	P	P	P	P	P	S
13														P	P	P	P	P	P	P	S
14															P	P	P	P	P	P	S
15																P	P	P	P	P	S
16																	P	P	P	P	S
17																		P	P	P	S
18																			P	P	S
19																				P	S
20																					S

Table 2: Optimal Value Table: Number of tubes = 20 Test cost= 200 Failure cost= 1000

STATE	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	3828	3463	3107	2757	2415	2079	1750	1444	1200	1000	833	692	571	467	375	294	222	158	100	48	0
1		3792	3577	3355	3127	2892	2653	2407	2158	1904	1646	1385	1143	933	750	588	444	316	200	95	0
2			3600	3399	3198	2996	2792	2588	2381	2174	1964	1752	1539	1322	1104	882	667	474	300	143	0
3				3400	3200	3000	2800	2600	2400	2200	1999	1799	1599	1398	1197	996	795	594	392	190	0
4					3200	3000	2800	2600	2400	2200	2000	1800	1600	1400	1200	1000	800	600	400	200	0
5						3000	2800	2600	2400	2200	2000	1800	1600	1400	1200	1000	800	600	400	200	0
6							2800	2600	2400	2200	2000	1800	1600	1400	1200	1000	800	600	400	200	0
7								2600	2400	2200	2000	1800	1600	1400	1200	1000	800	600	400	200	0
8									2400	2200	2000	1800	1600	1400	1200	1000	800	600	400	200	0
9										2200	2000	1800	1600	1400	1200	1000	800	600	400	200	0
10											2000	1800	1600	1400	1200	1000	800	600	400	200	0
11												1800	1600	1400	1200	1000	800	600	400	200	0
12													1600	1400	1200	1000	800	600	400	200	0
13														1400	1200	1000	800	600	400	200	0
14															1200	1000	800	600	400	200	0
15																1000	800	600	400	200	0
16																	800	600	400	200	0
17																		600	400	200	0
18																			400	200	0
19																				200	0
20																					0

Table 3: Our probability next tube bad: Number of tubes = 20 Test cost= 200 Failure cost= 1000

STATE	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	0.500	0.333	0.250	0.200	0.167	0.143	0.125	0.111	0.100	0.091	0.083	0.077	0.071	0.067	0.062	0.059	0.056	0.053	0.050	0.048	0.045
1		0.667	0.500	0.400	0.333	0.286	0.250	0.222	0.200	0.182	0.167	0.154	0.143	0.133	0.125	0.118	0.111	0.105	0.100	0.095	0.091
2			0.750	0.600	0.500	0.429	0.375	0.333	0.300	0.273	0.250	0.231	0.214	0.200	0.188	0.176	0.167	0.158	0.150	0.143	0.136
3				0.800	0.667	0.571	0.500	0.444	0.400	0.364	0.333	0.308	0.286	0.267	0.250	0.235	0.222	0.211	0.200	0.190	0.182
4					0.833	0.714	0.625	0.556	0.500	0.455	0.417	0.385	0.357	0.333	0.312	0.294	0.278	0.263	0.250	0.238	0.227
5						0.857	0.750	0.667	0.600	0.545	0.500	0.462	0.429	0.400	0.375	0.353	0.333	0.316	0.300	0.286	0.273
6							0.875	0.778	0.700	0.636	0.583	0.538	0.500	0.467	0.438	0.412	0.389	0.368	0.350	0.333	0.318
7								0.889	0.800	0.727	0.667	0.615	0.571	0.533	0.500	0.471	0.444	0.421	0.400	0.381	0.364
8									0.900	0.818	0.750	0.692	0.643	0.600	0.562	0.529	0.500	0.474	0.450	0.429	0.409
9										0.909	0.833	0.769	0.714	0.667	0.625	0.588	0.556	0.526	0.500	0.476	0.455
10											0.917	0.846	0.786	0.733	0.688	0.647	0.611	0.579	0.550	0.524	0.500
11												0.923	0.857	0.800	0.750	0.706	0.667	0.632	0.600	0.571	0.545
12													0.929	0.867	0.812	0.765	0.722	0.684	0.650	0.619	0.591
13														0.933	0.875	0.824	0.778	0.737	0.700	0.667	0.636
14															0.938	0.882	0.833	0.789	0.750	0.714	0.682
15																0.941	0.889	0.842	0.800	0.762	0.727
16																	0.944	0.895	0.850	0.810	0.773
17																		0.947	0.900	0.857	0.818
18																			0.950	0.905	0.864
19																				0.952	0.909
20																					0.955

6 BBF Programming

Backward, brute force programming is trivial. Here's the entire Perl code for the boiler tube pulling problem.

```
#!/usr/bin/perl
# calculate and print out optimal value and decision tables
# for boiler tube pulling problem
#
$N=20;                            # Number of tubes in boiler
$t=200.0;                         # cost of pulling a tube
$T=1000.0;                        # cost of tube failing in service
$k_prior = 1;                     # prior parameters
$n_prior = 2;

for ($k = 0; $k <= $N; $k++) { # Boundary condition at Stage N
    $V[$k][$N] = 0.0;             # optimal value function table
    $D[$k][$N] = 'S';             # optimal decision table
}

for ($n = $N-1; $n >= 0; $n--) { # loop over stages going backwards
    for ($k = 0; $k <= $n; $k++) {  # loop over states at this stage
        $p_bad = ($k + $k_prior) / ($n + $n_prior);
        $stop_cost = $T * ($N - $n) * $p_bad;
        $pull_cost = $t + $p_bad  * $V[$k+1][$n+1]
                        + (1.0 - $p_bad) * $V[$k][$n+1];
        if ($stop_cost < $pull_cost) {
            $V[$k][$n] = $stop_cost;
            $D[$k][$n] = 'S';
        } else {
            $V[$k][$n] = $pull_cost;
            $D[$k][$n] = 'P';
        }
    }
}

# print out optimal value function and optimal decision tables
open(VTABLE, ">vtable");
open(DTABLE, ">dtable");
printf(VTABLE "\nSTATE|");
printf(DTABLE "\nSTATE|");
for ($n = 0; $n <= $N; $n++) {
    printf(VTABLE "%5.0f|", $n);
    printf(DTABLE "%5.0f|", $n);
}
printf(VTABLE "\n");
printf(DTABLE "\n");
for ($k = 0; $k <= $N; $k++) {
    printf(VTABLE "%5d|", $k);
    printf(DTABLE "%5d|", $k);
    for ($n = 0; $n < $k; $n++) { #skip all n's smaller than k
        printf(VTABLE "     |");
        printf(DTABLE "     |");
    }
    for ($n = $k; $n <= $N; $n++) {
        printf(VTABLE "%5.0f|", $V[$k][$n]);
        printf(DTABLE "%5s|",   $D[$k][$n]);
    }
    printf(VTABLE "\n");
    printf(DTABLE "\n");
}
```

All the work is done in lines 15 to 30. It is hard to imagine anything simpler.

Notice also that at stage n, we need only the optimal value function at stage $n+1$. So if memory is a problem, we can move the optimal value function for stage $n+2$ to disk as soon as we finish the calculations at stage $n+1$ with nil effect on computation time. From a memory point of view, it doesn't matter how many stages we have.

A little reflection will reveal that storing the decision table is unnecessary, for the optimal decision for any combination of state and stage can be recovered from the optimal value table. The optimal decision at any stage and state is the decision that results in equality between the left hand and the right hand sides of the recursion relation Section 4.

7 Real World Implementation Issues

Needless to say, all our probabilities, tables, etc are based on a model, a gross simplification of the real world. This means the results must be applied with a great deal of judgement.

We have already talked about not using a uniform prior unless you are willing to throw away your past knowledge, ***unless you know you want to do enough testing so that the test results are much more important than any past experience***.

Another crucially important issue is real world tests are rarely pure pass/fail. Boiler tubes are not like kids: girl or boy. The condition of a tube can range from pristine to so lousy we are amazed that it hasn't already failed. This is why we did not throw away the optimal value table, after we had used it to compute the optimal decision table. In fact, we kept not only the optimal value table, but also the average cost of stopping table $Stop(k_b, n)$ for every possible (k_b, n) and the average cost of pulling table $Pull(k_b, n)$. We gave our superintendent in charge of the overhaul access to these two tables.

If when the superintendent got to (k_b, n), the tubes were looking really good, he was instructed to look at $Stop(k_b, n)$ and $Pull(k_b, n)$, and, if they were not all that different, cheat toward stopping. However, if when he got to (k_b, n), a lot of the tubes we had called "good" were really pretty marginal, he was instructed to cheat toward pulling.[4]

The superintendents who were originally very leary of this intrusion on their professional judgement came to appreciate the guidance. Well, some of them did.

Another important issue is the assumption that all parts face identical conditions. This is never completely true, although some machines such as condensers come close. But a boiler really consists of at least three different kinds of tubes: steam generating, superheater, and economizer. Each section of the boiler faces different temperature and scaling conditions. In such situations, it may make sense to divide the machine into sub-machines, and treat each such sub-machine independently. This partitioning turns out to have important computational advantages, as we shall see.

8 Varients on Boiler Tube Pulling

We have purposely focused on only the simplest version of the boiler tube pulling problem. Among the possible variations are:

1. Unlike boiler tube pulling, in many maintenance problems, the test is non-destructive; and it's cheaper to simply re-insert a good old part than to have to replace a bad part with a brand new part. Pulling diesel engine pistons is an example from my old day job.
   The recursion relation ship becomes

$$V(k_b, n) = min \begin{cases} T(N-n)(k_b+1)/(n+2) \\ (k_b+1)/(n+2)(t_b + V(k_b+1, n+1)) + (n-k_b+1)/(n+2)(t_g + V(k_b, n+1)) \end{cases}$$

   where t_b is the cost of testing and replacing a bad part with a new part, and t_g is the cost of testing and re-inserting a good old part.
2. Defective parts don't necessarily fail between now and the next overhaul. Good parts don't necessarily not fail between now and the next overhaul. However, defective parts have a higher rate of failure than good parts.
   In this case, the mean cost of stopping at stage N after observing k_b defectives is

$$Stop(k_b, n) = \frac{T(N-n)}{n+2}(P_b(k_b+1) + P_g(n-k_b+1))$$

[4] Near the Red-Black interface in Table 1, the difference between the average cost of stopping and the average cost of testing some more is usually not large. But the further away from this interface the superintendent gets the bigger this difference will become. Notice that backwards brute force has calculated what to do and its cost for combinations of stage and state which we could never have gotten into if we had strictly followed Table 1. I'd call that very thoughtful.

where P_b is the probability that a defective part fails and P_g is the probability that a non-defective part fails. The boundary condition at stage N is no longer zero, but TP_gN reflecting the fact that we are starting out with N non-defective parts.

3. Sometimes the cost of failure in service is not linear in the number of failures. For example, the system can withstand one failure and keep running, but two failures is a disaster.
   In this case, the mean cost of stopping at stage N after observing k_b defectives is

$$Stop(k_b, n) = \sum_{i=1}^{N-n} f_{\beta B}(K|N-n, k'+k_b, n'+n)C(K)$$

   where $C(K)$ is the non-linear cost of K in service failures.[5]

4. Sometimes it's much cheaper to test parts in batches than to test them one by one. That is, the cost of pulling and testing m tubes in a batch $t(m)$ is not tm. In this case, the decision at any stage is not: do we stop or do we test one more part; but rather: do we stop or test and, if test, how may parts in this batch.

$$V(k_b, n) = min \begin{cases} T(N-n)(k_b+1)/(n+2) \\ t(m) + \sum_{K=0}^{m} f_{\beta B}(K|m, k_b+1, n+2)V(k_b+K, n+m)) \end{cases}$$

   where the minimum is over $m = 0(stop), 1, 2, ..., M$ where M is the largest batch size, we want to consider and K is the number of defectives observed in the batch. This varient is interesting in that we can "skip" stages.[6]

5. Combinations of the above.

Using backwards, brute force (aka dynamic programming) it is possible to handle all of these situations with little more computational travail than for the simplest case. (See footnotes for caveats.) That's pretty amazing.

The reason for this flexibility is that backwards, brute force does not depend on the form of the stop or pull cost functions, but only on the fact that you can divide the total average loss at stage n into the average loss incurred at this stage plus the average loss from stage $(n+1)$ on.

9 Yes/No/Maybe

In many situations, a test can have more than two possible outcomes, more than just Pass or Fail. In USA politics, polled voters are often divided into Democrats, Republicans, and Smart. In the boiler tube problem, the tube test might result in Good/Marginal/Bad. Or we might have four possible outcomes, or five, or ....

There is a straight forward generalization of the Beta family for 2-D and 3-D and whatever-D problems. It is called the multi-variate Beta, $f_\beta(k'_1, k'_2, ..., k'_{M-1}, n')$.

If we use a member of this family, amd then do n tests and observe k_1 occurances of the first possible outcome, k_2 occurances of the second possible outcome, k_3 occurances of the third possible outcome, and so on, our posterior is a multi-variate Beta whose parameters are $(k'_1 + k_1, k'_2 + k_2, ...k'_{M-1} + k_{M-1}, n' + n)$.[7] and our new probability on the mth outcome in the next test is $(k'_m + k_m, n' + n)$.

With this background, it is easy to write down the recursion relation for the optimal value function for the M-outcome variant of the simple boiler tube pulling problem. The corresponding computer program is also a trivial generalization of the two outcome problem. So the method easily generalizes to the M-outcome varient for any M.

[5]In general, computing the Beta-binomial for all its parameters can be very costly. In many real world problems, the probability of K being larger than, say, 4 or 5 is negligible in which case the numbers are doable. If this is not the case, then the factorials will have to be approximated by Stirling's formula or its varients.

[6]Computationally, this means we must keep the optimal value table for the next M stages in core if we are to avoid myriad disk reads.

[7]Sometimes called the Dirichlet density.

But we do have a computation problem. The state of the system at any stage is now an $M - 1$ dimensional vector. Suppose we are pulling boiler tubes and we decide to divide the results into Bad, Marginal, and Good. Then after pulling n tubes, we will have k_b bad tubes, k_m marginal tubes, and the rest are bad. If we have a 1000 tube boiler, then the number of possible states when the 1000th tube is pulled is 1000 * 1001 / 2. The number of states has gone up be a factor of 500. In general, the number of possible states at the final stage is

$$S(N, M) = \frac{(N + M - 1)!}{N!(M - 1)!}$$

where N is the number of tubes and M is the number of possible outcomes in an individual test.[8] This is a number that quickly gets out of hand as M grows as Table 4 shows. This is called the curse of dimensionality.

Table 4: Number of States

	Number of Possible Outcomes			
TUBES	2	3	4	5
10	11	66	286	1001
50	51	1326	23426	316251
100	101	5151	176851	4598126
500	501	125751	21084251	2656615626
1000	1001	501501	167668501	42084793751

Memory is usually not the critical factor.[9] With multi-gigabyte memorys, we can fit problems involving one hundred million states or more onto a modern desktop. The problem is computation time. The average number of states during the computation is approximately half $S(N, M)$. Table 5 showns the total number of stage-state combinations for a range of N and M. We must compute the recursion relation this many times.

Table 5: Number of State-Stage Combinations

	Number of Stage-State Combos			
TUBES	2	3	4	5
10	5.500e+01	3.300e+02	1.430e+03	5.005e+03
50	1.275e+03	3.315e+04	5.857e+05	7.906e+06
100	5.050e+03	2.576e+05	8.843e+06	2.299e+08
500	1.252e+05	3.144e+07	5.271e+09	6.642e+11
1000	5.005e+05	2.508e+08	8.383e+10	2.104e+13

On my very average desktop using totally un-optimized code, it takes about 3 seconds to do a million combos. If we are prepared to let such a machine run over-night, we could handle about 10^{10} combos. These ruminations point to a rough limit of 1000x3, or 500x4, or 200x5 before we have to do something exotic.

With repect to the latter, there are a number of possibilities, including:

Parallelizing BBF parallizes easily and efficiently. If we have a number of processors at our disposal, we can assign each processor a portion of the state space. At any stage, n, the processors have no need to communicate with each other, they only need access to the optimal value function computed at the last (n+1)st stage, which could be stored in common memory. After all the processors have computed, their computations at stage n, the results become the new common memory, and everybody moves on to stage $n - 1$. There is nil overhead

[8] $S(N, M)$ is the number of possible ways of putting N balls into M cells.
[9] This certainly was not true ten or more years ago.

associated with dividing the problem among multiples CPU's. If a problem justifies the use of a cluster, we can scale up nearly linearly. A 100 CPU cluster would allow us to compute problems in the 10^{12} range over-night.

Aggregation Aggregation is the process of lumping similar states into a single aggregate state. For example, in the boiler tube pulling problem, we might decide to count defective tubes in twos. In other words, treat seeing 250 defective and 251 defective as the same state, thereby halving the number of states. Of course, in doing the recursion at aggregate state 125, we don't know if the actual unaggregated state is 250 or 251 bad. The usual way around this is to assume that all the aggegated state's composite states are equally likely at which point one can compute the transition probabilities necessary to do the recursion. The efficacy of aggregation is very problem dependent, but there are plenty of problem areas where aggregation can lead to intelligent if not near-optimal solutions. If you think about it, lumping our test results into two or three categories is a form of aggregation.

Partitioning It is often possible to obtain a near optimal solution by *partitioning.* Suppose we have a machine made up of a 1000 identical parts and we need to divide the test outcomes into 5 categories. Tables 4 and 5 indicate we are out of business. We pretend our machine is two machines A and B, each made up of 500 parts. We compute the optimal value table for A, do the tests, and close up machine A as soon as the optimal decision table tells us to. Suppose this happens after we have seen n_A tubes for which k_{A1} fell into category 1, K_{A2} fell into category 2, and so on. We now re-compute the optimal decision table; but this time our prior has parameters, $k'_1 + k_{A1}, k'_2 + k_{A2}, \ldots$. We don't throw away the information we learned from A in doing B. But we may stop testing on A prematurely relative to the true optimum, because the algorithm for A can't see the value of more information for B.

If we partition into halves, we will gain 2^{M-1} in both computational effort and memory. And we can partition into thirds or quarters or whatever is required computationally if we are willing to accept the correspondingly sub-optimal, but often still intelligent solution.

We have already seen an example of partioning, when we divided the boiler into three separate sections: steam-generating tubes, superheater, and economizer. Sometimes partitioning is quite natural.

In short, by a combination of parallizing and judicious use of aggregation and partitioning, optimal or near-optimal maintenance policies can be obtained for quite large problems by backwards brute force. I really don't know why more use has not been made of it.