Bayesian Search and Debugging

by Ealdwulf Wuffinga

1 Introduction

Bayesian Inference is a method of reasoning in the presence of uncertainty. Its applied cousin, Bayesian Decision Theory, is the use of Bayesian inference to make decisions which maximise one's expected utility. This paper examines whether Bayesian Search theory (a sub-field of the decision theory) is useful for debugging. It is of particular interest to find methods for diagnosing the cause of an intermittent (non-deterministic) error. The first part of this paper outlines various known debugging techniques which can be viewed as being kinds of search, and then provides an introduction to Bayesian Search Theory. The second part derives two Bayesian search algorithms for particular cases. These are:

- An algorithm for finding the location (in execution time) or a fault, taking into account the cost of observations, and assuming that check-pointing is unavailable.
- An algorithm for binary search which is robust in the case of false-negative observations. This can be used (eg) to locate the point in program history where a bug which causes *intermittent* faults was introduced.

Source code for both algorithms is available.

Because I am writing this in my spare time, this paper is also an invitation to full time researchers to investigate this area, as I cannot guarantee that I will have time to do so myself. Therefore I also list a number of open questions.

1.1 Debugging and search

Many of the activities required to debug a program can be thought of as search [FIXME: citations].

A search can be thought of as consisting of:

– A space to be searched. We may consider more than one co-ordinate system over the space.

– A means of probing locations in the space. We may obtain information about more than the location being probed (eg, searching an ordered list). The information may be probabilistic.

– A strategy for deciding where to probe next, based on the information we have obtained so far.

This is a fairly informal model, just for the purposes of categorising the searches we describe next. A more rigorous, set-based way of describing searches may be found in [3], which is an overview of search as used in various areas of applied mathematics.

During debugging, we may search in various kinds of spaces. Some of these are:

[FIXME: figure out how to fix this damn table]

Space(/Time)	Co-ordinate system(s)	Target	Information hoped to be acquire
Development history	Program versions	First version exhibiting fault	Change that cause fault
Program text			
Space of inputs	depends on kind of input		
Duration of execution	Time,control flow,data flow	state of program at fault	proximate cause of fault
Hypothesis space			

(It may seem a bit of a stretch to call hypothesis testing a search, but we shall see below that the mathematics is similar.)

[Fixme: this section is disjointed]

A given fault can often be located in many of these spaces; the choice of which space to search in a particular case can greatly speed up - or slow down - the debugging process. As can the invention of a 'space' particular to the problem at hand.

In some cases the search target can be more complicated than a single location. In [2], an algorithm ('Delta Debugging') is described which looks for a minimal subset which provokes a fault - it can be used to search in any space, and has been demonstrated in at least the space of inputs and the space of program changes (deltas, from which it gets its name).

The co-ordinate system for a particular space is also interesting. When using a debugger, one usually has to search in the 'control flow' co-ordinate system; but quite often one actually wants to think about the search as being a search of the data flow, and the control flow is a distraction.
[fixme - more to say here?]

1.2 Bayesian Search Theory

There are two kinds of Bayesian search, depending on the source of uncertainty. One is where we have a prior probability distribution over the target space, but the search itself is deterministic. [add examples - Huffman, zhigljavsky's root finding methods]. The other is where the uncertainty also comes from the observations. [add examples - submarines, Search And Rescue]
[Stuff to put here: description of basic 'boxes' search. Iida's etymology of search theory]

2 Bayesian search for debugging

We give two search algorithms, with different applications. The first assumes reliable observations, the second, without.

2.1 Finnegan search

Suppose we are using a debugger to find the instant at which a fault occurs, in a deterministic program. We know it to be in the interval [S,S+L). To make an observation (to see whether the fault has occurs yet) has cost C. If we have gone too far, we must restart the program from the beginning. (I don't know if this problem has a name, but I call it 'Finnegan search' after the rhyme every verse of which ends, 'So poor old Finnegan, had to beginagain!').
Without loss of generality, we take the cost of a single step to be 1, and the cost of restarting the debugger to be zero (not including running the program back to S).

The search strategy is simply a function f(S,L,C) which returns the number of steps to make before the next observation. Obviously, if C is zero f is always one, and if C is large, f is L/2. But what about when C is in between?

The optimal solution can be defined recursively:

i. $f(S,L,C) = \text{the } k \, s.t. E(S,L,C,k) \text{ is minimised}$

ii. $E(S,1,C,n) = 0$

iii. $E(S,L,C,n) = n + C + (S + E_{\text{opt}}(S,L,C)p + E_{\text{opt}}(S+n, L-n, C)\, q$
(Where $p = \frac{n}{L}$ and $q = 1-p$)

iv. $E_{\text{opt}}(S,L,C) = E(S,L,C,\, f(S,L,C))$

$E(S,\ L,\ C,\ n)$ is the expected cost of the whole search, given that the next observation is after n steps, and the optimal strategy is used subsequently.

This is impractical to use, even with dynamic programming. An approximate solution, which empirically [FIXME - evidence] has a cost very close to the optimal, is to take the number of steps which gives us the best *expected* cost/benefit ratio for this observation. The expected cost is the number of steps, plus C, plus the probability that we have to rerun from the start times the cost of doing so. The expected benefit is the change in entropy of the unknown information.

This is effectively a greedy algorithm, since it maximises immediate return, although it seems misleading to call an algorithm greedy, which maximises benefit/cost rather than just benefit - one might even call it a *prudent* algorithm.

If the best number of steps is >1, it can be estimated by solving for n (or looking up) the equation:

$$1 + \frac{S+C}{L} = \frac{\log(n/L)}{\log(1-n/L)}$$

[Fixme: find derivation, or redo it]

We take the number of steps to be 1 or n, whichever has the better cost/benefit ratio.

2.2 Binary search with false-negative observations

It is fairly common to want to locate the point in a program's history at which a bug was created. Several version-control systems provide support for this search, but all assume that it can be determined reliably whether the bug is present in a given version - IE, the search cannot be used in the presence of false-negatives, such as would be cause if the only known symptom of the bug is intermittent faults.

Here we derive an algorithm which takes into account the presence of false negatives. It is a fairly straightforward modification of the existing Bayesian Search algorithms cited above, for the case of uncertain observations.

In the SAR and military uses of Bayesian search, an observation only tells you about a fixed area around the search, whereas here we assume that an observation anywhere after the bug was introduced may detect the bug. This does not present much of a problem mathematically. More awkward is the fact that we don't know the probability that an observation will detect the bug, whereas for other purposes the detection function can be measured, or derived from physical principles. Fortunately Bayesian inference provides a technique for eliminating unknown variables, called 'Marginalisation'. This requires that we assume a prior distribution over the unknown variable. In this case, we assume the the probability of detection r is uniform in 0...1.

We also assume here that the history is a total order. This may not be true for the case where the revision control system keeps track of where branches have been merged to; then history is a DAG. I seen no reason why this algorithm cannot be generalised to the DAG case, although doing so would probably be messy.

The algorithm needs to do two things: decide where to make each observation, and decide when to stop. In this case, we adopt the simplest approach, which is to always make the observation for which the expected entropy of the posterior distribution is smallest, and to stop when the probability of the most likely location reaches some reasonable value. We do not claim that this results in an optimal one, only a practical one. When search effort is continuous, rather than discrete, minimising entropy often results in an optimal algorithm, but that is not the case here. [fixme: be less wooly]

2.2.1 Definitions

r_k	failure rate at location k
$l_1..l_N$	locations
L	(putative) location of bug

Table 1.

A test at location l_k detects with probability r_k if $L > k$, otheriwse does not detect.

Initialy, we shall assume that $r_k = r$ for all k. Later we shall show how to relax this assumption

2.2.2 Assuming constant r at all locations

Our evidence E is completely represented by:

$t_k, k \in 1...N$	number of tests at locations k which did not detect
$d_k, k \in 1...N$	number of tests at locations k which did detect

Table 2.

Our prior belief is $P(L = l_k) = \mathrm{bl}_k$ (ie, arbitrary) and $P(a < r < b) = b - a$ (ie, uniform p.d.f. on r in 0..1)

We can write down $P(E|L, r)$, as follows:

$$P(E|L,r) = \begin{cases} 0 & \text{if } d_k \neq 0 \text{ for any } k < L \\ \prod_{i=L...N} (1-r)^{t_i} r^{d_i} & \text{otherwise} \end{cases}$$

(NB: this assumes we know the order in which the detections and non-detections occurred, otherwise there is a term $\binom{t_i + d_i}{t_i}$. (fixme: wouldn't this drop out in the end anyway?)

We want $P(L|E)$. To get this we must marginalise $P(L, r|E)$.

$$P(L|E) = \int_{r \in 0...1} P(L, r|E)$$

By Bayes theorem,

$$P(L, r|E) = \frac{P(E|L, r)P(L, r)}{P(E)}$$

$P(L, r)$ is just the prior $P(L)P(r)$.

$P(E)$ must be found by marginalising $\mathrm{pdf}(E, L, r)$ over both L and r

$$\mathrm{pdf}(E, L, r) = P(E|L, r)P(L)\mathrm{pdf}(r)$$

$$\begin{aligned} P(E) &= \sum_{i \in 1...N} \int_{r=0...1} P(E, L, r)\mathrm{dr} \\ &= \sum_{i \in 1...N} \int_{r=0...1} P(E|L, r)P(L)\mathrm{pdf}(r)\mathrm{dr} \end{aligned}$$

define $g(E, L)$ as the integral:

$$g(E, L) = \int_{r=0...1} P(E|L, r)P(L)\mathrm{pdf}(r)$$

$$g(E,L) = \begin{cases} 0 & \text{if } d_k \neq 0 \text{ for any } k < L \\ \int_{r=0...1} \prod_{i=L...N} (1-r)^{t_i} r^{d_i} P(L)\mathrm{dr} & \text{otherwise} \end{cases}$$

define $T_l = \sum_{i=L\ldots N} t_i$ and $D_l = \sum_{i=L\ldots N} d_i$. Then $g(E, L)$ is:

$$g(E,L) = \begin{cases} 0 & \text{if } d_k \neq 0 \text{ for any } k < L \\ \int_{r=0\ldots 1} (1-r)^{T_L} r^{D_{L_i}} P(L)\mathrm{dr} & \text{otherwise} \end{cases}$$

$$g(E,L) = \begin{cases} 0 & \text{if } d_k \neq 0 \text{ for any } k < L \\ \mathrm{Beta}(D_L+1, T_L+1)P(L) & \text{otherwise} \end{cases}$$

$$P(E) = \sum_{i \in 1\ldots N} g(E,i)$$

In fact, $g(E, L)$ is also, by the same reasoning, $P(E|L, r)P(L, r)$ marginalised over r, which is the remaining piece we need to write down $P(E|L)$:

$$P(E|L) = \frac{g(E,L)}{\sum_{i \in 1\ldots N} g(E,i)}$$

We also need to calculate $P(D_i|E)$, the probability that a measurement at l_i will detect given the evidence E collected so far.

$$P(D_i|E) = \frac{P(D_i, E)}{P(E)}$$

We already know how to calculate $P(E)$: $P(D_i, E)$ is the same except with another detection added to d_i.

2.2.3 Multiple r_i

If we cannot assume that r is constant at all locations, there are a couple of possibilities:

- Assume nothing about the r_i
- take the possibility that r is constant into account, by assigning some prior probability to the hypothesis that this is the case.

The latter is the best course of action if we have some reason for believing that constant r is more likely than would be represented by an independent uniform distribution of all the r_i. One reason for proceeding in this fashion is that an intermittent fault is likely to have some underlying rate, constant in all locations, but in some locations it may be (partially) obscured.

2.2.4 Independent r_i

In the case where we assume nothing about the r_i, we assign them each an independent uniform prior in 0..1.

$P(E|L, r_0 \ldots r_N)$ is now:

$$P(E|L, r_0 \ldots r_N) = \begin{cases} 0 & \text{if } d_k \neq 0 \text{ for any } k < L \\ w & \\ \prod_{i=L\ldots N} (1-r_i)^{t_i} r_i^{d_i} & \text{otherwise} \end{cases}$$

$$P(L|E) = \int_{r_0} \ldots \int_{r_N} P(L, r_0 \ldots r_N | E)$$

The above calculation all works out the same, with the extra integrals everywhere, and we get:

$$g(E,L) = \begin{cases} 0 & \text{if } d_k \neq 0 \text{ for any } k < L \\ \prod_{i \in L\ldots N} \mathrm{Beta}(d_i+1, t_i+1)P(L) & \text{otherwise} \end{cases}$$

2.2.5 Entropy calculation

Now, we need to calculate the expected entropy after the observation. For this we need:

- The probability that the observation will detect, at each location ($P(D_i|E)$, above)

- For each L_i, the entropy of the probability distribution $P(L|E_d)$ where E_d is the evidence if we have another detection at i
- The same entropies, for the case when we do not detect at L_i.

This sounds like we need $O(N^2)$ probabilities. However, they are actually made up from only $O(N)$ likelihoods $g(E, L)$ because of the following:

- if $E_a = E_b$ execpt that some d_i in E_a is replaced by $d_i + 1$ in E_b, then $g(E_a, L_j) = g(E_b, L_j)$ for all $j < i$
- if $E_a = E_b$ execpt that some t_i in E_a is replaced by $t_i + 1$ in E_b, then $g(E_a, L_j) = g(E_b, L_j)$ for all $j < i$
- if $E_a = E_b$ execpt that some d_i in E_a is replaced by $d_i + 1$ in E_b, and d_{i+1} in E_b is replace by $d_{i+1} + 1$ in E_a, then $g(E_a, L_j) = g(E_b, L_j)$ for all $j < i$ and j>i
- if $E_a = E_b$ execpt that some t_i in E_a is replaced by $t_i + 1$ in E_b, and t_{i+1} in E_b is replace by $t_{i+1} + 1$ in E_a, then $g(E_a, L_j) = g(E_b, L_j)$ for all $j < i$ and j>i

For Shannon entropy, there is no obvious way to make use of this fact. However, Reny entropy H_α has a simpler algebraic structure:

$$H_\alpha(X) = \frac{1}{1-\alpha} \log\left(\sum_{i=1}^{n} p_i^\alpha \right)$$

In the limit as α tends to 1, H_α converges to the shannon entropy. So we use the Renyi entropy instead, and pick an α close to 1 in order to approximate the Shannon entropy. (In fact other α might be better, since Renyi entropy originated in Search theory [check this], but I have not looked into this.)

By using the Renyi entropy, we can calculate all the entropies in $O(N)$ time. This is still expensive, but it is sufficient to make the algorithm useful when the cost of making an observation is high. As a rough measure, it takes my (2006 Athlon) PC 1.5 seconds to compute the decision when N=1000. The calculation could probably be made faster, but it will remain $O(N)$ if we have to calculate the entropies and probabilities explicitly. So an interesting question is whether an algorithm with the same behavior can be devised, which does not explicitely calculate $O(N)$ values.

[add more here about the behavior of the algorithm]

This algorithm opens the possibility of tracking down intermittent bugs in an automated way, by leaving a machine to test versions of a program until one has a sufficient probability of being where the bug was introduced.

3 Conclusions

[write some conclusions]

3.1 Open questions

1. Are there any other useful spaces over which to search, during debugging?
2. Can one write a program such that *searching in it* is efficient? Is this the same as, or in conflict with, or independent of, the program being well written in existing senses?
3. Questions about binary search with false-negatives:
   i. Is maximising change-in-entropy optimal?
   ii. Can we take into account the cost of switching from one location to another being greater than retesting the current location?

iii. Is there a procedure which makes its decision in less than $O(n)$ time, which has the same behaviour as calculating all the entropies?

4. Can Zeller's 'Delta debugging' algorithm be generalised efficiently to the false-negative case? A straightforward approach would require calculating $O(2^N)$ probabilities, so this is not obvious. In [1],[4], Pollock claims that huge numbers of probabilities is a generic problem with Bayesian inference, and advocates an alternative approach based on his 'Nomic Probabilities'.

Bibliography

[1] *J. L. Pollock, Knowledge and Skepticism*, chapter Reasoning defeasibly about probabilities. MIT Press, 2008.

[2] Holger Cleve and Andreas Zeller. Finding failure causes through automated testing. In *Automated and Algorithmic Debugging*, 2000.

[3] J.H. O'Geran, H.P. Wynn, and A.A. Zhigljavsky. Search. *Acta Applicandae Mathematicae*, 1991.

[4] J. L. Pollock. Probable probabilities. Technical report, OSCAR Project, 2007.