Probabilistic Declarative Debugging

Lee Naish
Computer Science and
Software Engineering
University of Melbourne

Outline

Declarative Debugging

Search strategy

Estimating probabilities

Examples

Conclusion

But first . . .

“God does not play dice” — Einstein

“God not only plays dice but also sometimes throws them where they cannot be seen” — Hawking

A puzzle for the rational

Suppose N dice of various designs have been created

Each die i, $1 \leq i \leq N$ is thrown S_i times, $S_i > 0$ and on at least one throw of some die the side of the die which comes up is blank — at least one of the dice is defective!

Each die i is thrown K_i more times, $K_i \geq 0$ and no blanks come up

1. What are the odds that die i is defective and
2. assuming it is, what are the odds its next throw comes up blank?

You may want to introduce some simplifying assumptions such as only one of the dice is defective

A puzzle for the rational (cont.)

For example

- Two dice ($N = 2$)
- One suspect throw of each ($S_1 = 1, S_2 = 1$)
- One hundred known correct throws of die 1 and none for die 2 ($K_1 = 100, K_2 = 0$)

It seems most likely that die 1 is ok and die 2 is defective

Diagnosing wrong answers in Prolog

A successful Prolog computation can be viewed as a *proof tree*

Each node is an atomic goal which was proved

Each leaf is an instance of a fact

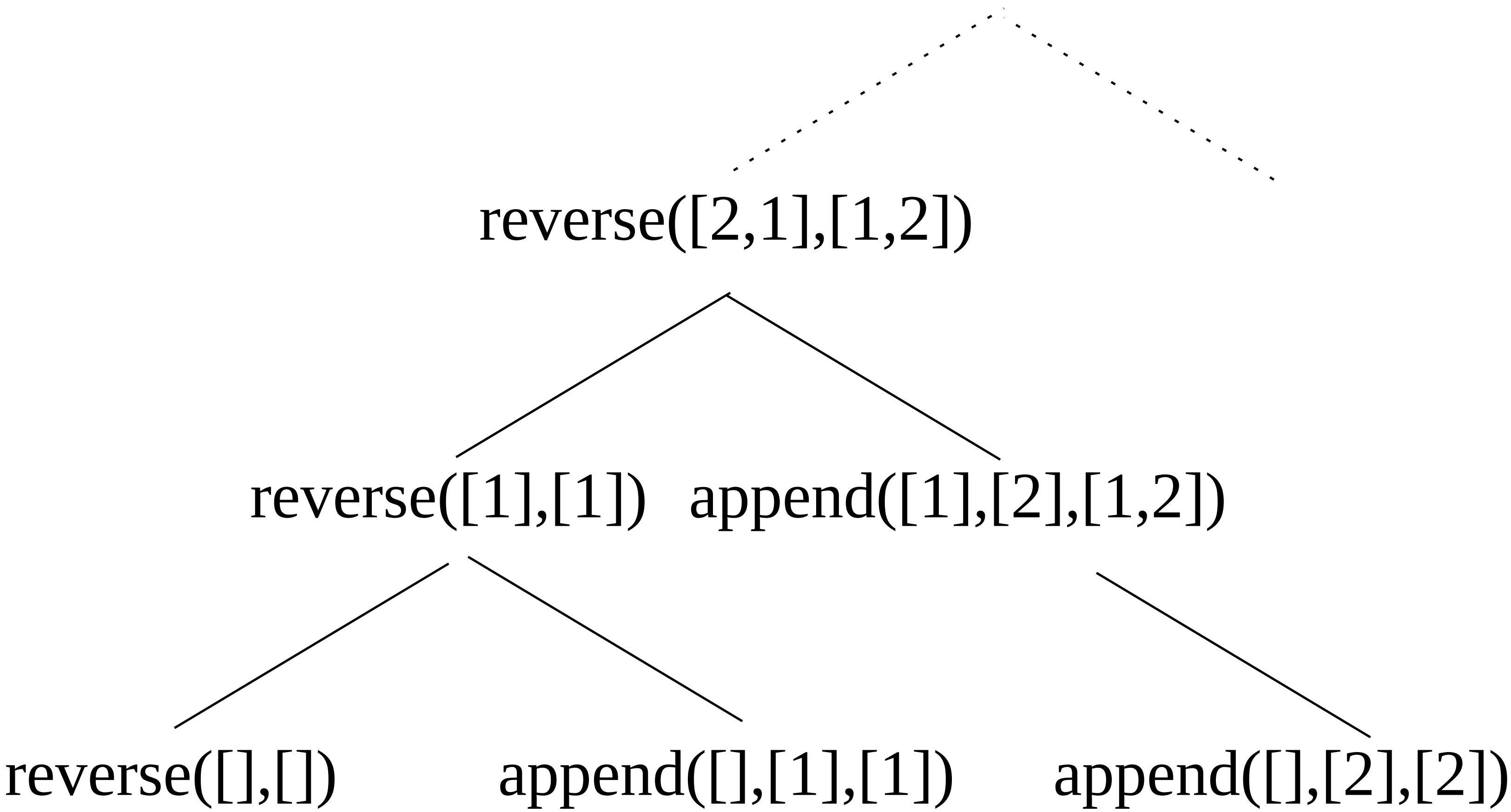
Each internal node is an instance of the head of a rule; the children are instances of subgoals in the body of the rule

Example code — naive reverse

```
% reverse of a list
reverse([], []).
reverse([A|As], Bs) :-
    reverse(As, Cs),
    append(Cs, [A], Bs).

% concatenation of two lists
append([], As, As).
append([A|As], Bs, [A|Cs]) :-
    append(As, Bs, Cs).
```

Example proof tree



Diagnosing bugs

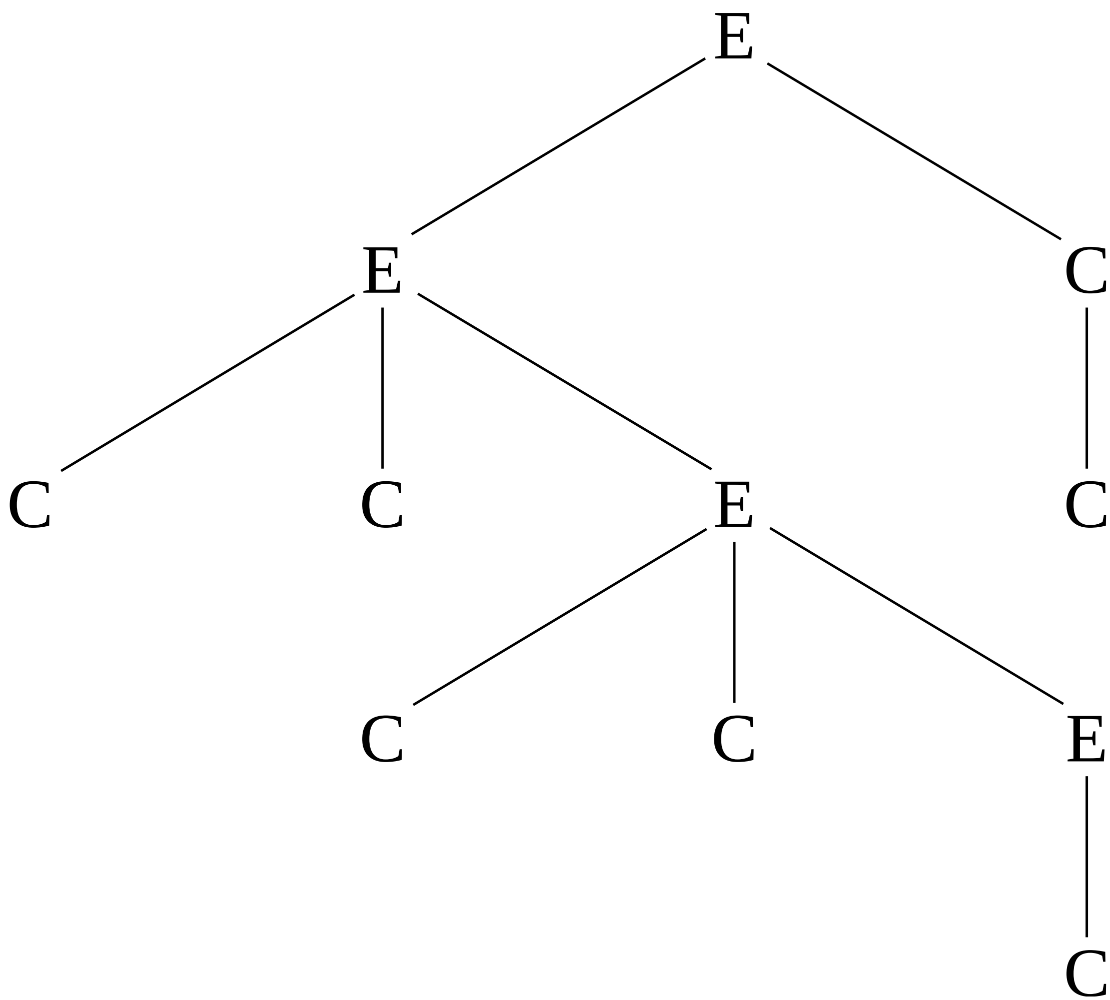
Nodes can be *correct* (*valid* in the intended interpretation) or *erroneous*

A node is *buggy* if it is erroneous but has no erroneous children

It correspond to an incorrect clause instance: the body is valid but the head is not

Our mission is to seek out buggy nodes

A tree with correct and erroneous nodes



More generally ...

We need to reconcile a computation (which has a fault) with a program

A computation can be viewed as a tree(!)

Each node corresponds to an execution of a section of code

Internal nodes use the results of one or more sub-computations

We need a way of determining if a subcomputation is correct

Search strategy

A bottom-up search is possible (but typically performs poorly)

A top-down search which follows a path of erroneous nodes is reasonably effective

Divide and query repeatedly finds a subtree which is (about) half the size of the tree and probes the root of that subtree

If the root is erroneous we narrow the search to the subtree, otherwise we delete the subtree

Around $\log N$ probes are required, which is "optimal"

Doing better than optimal

The *average* case is more important that the worst case

Nodes correspond to code segments and some code segments are more likely to be buggy than others

A large subtree may only use a small part of the code; a small subtree may have a relatively high likelyhood of being erroneous

Some probes are *much* more costly than others — we want to optimise the total cost, not just the number of probes

Note: naively incorporating cost estimates into divide and query can make average performance worse

Doing better than optimal (cont.)

Example: Suppose our intended interpretation for naive reverse was unusual (eg, only first and last elements are swapped)

We run it with a 1000 element list and the answer is wrong...

Divide and query would ask about the reverse of a list of length (around) 700

But unless our interpretation is very contrived this subtree is almost certainly erroneous, and the question asked is complex

Using a length of 10 (say) the question is *much* simpler but its still very likely erroneous — all the code is used, just not as much

Probabilistic search algorithm

We estimate the probability of each node being erroneous and the cost of probing it

The expected total cost of searching tree T using the root of subtree S as the first probe is

- $probe\_cost(root(S))$, plus
- $Pr(S$ erroneous$)\times$ cost of searching S, plus
- $Pr(S$ correct$)\times$ cost of searching $T \setminus S$

We want to pick a subtree S which minimises this

The cost of searching S (and $T \setminus S$) is approximated using log_2 of the size times the average probe cost (full "look-ahead" would take too long)

We use a special case for the root being likely to be buggy

Computing probabilities, naively

Assume we don't know whats in each node in the tree

Each node has a small probability ϵ of being buggy, a subtree of size N has a probability of $1 - (1 - \epsilon)^N$; approximately $N\epsilon$

This leads to divide and query

If we don't know the tree structure, top-down is rational

If we don't know ϵ is small, bottom-up is rational

Computing probabilities, naively (cont.)

But we do know (or can find out) about nodes in the tree

And probabilities are not independent — two nodes may use the same code (for example)

Consider choosing a (possibly defective) die and repeatedly tossing it versus repeatedly choosing a die and tossing it

We can weaken independence assumptions by using *conditional probabilities* and Bayes theorem:

$$Pr(A|B) = Pr(A)Pr(B|A)/Pr(B)$$

So, Pr(a throw is blank) =
Pr(die is defective) $\times$ Pr(a throw is blank | die is defective)

Computing probabilities

Inputs: A suspect tree $\mathcal{T}$, a set of clauses $\mathcal{C}$ with instances in $\mathcal{T}$,
counts of instances of each of these clauses in correct trees

Outputs: For each subtree S of suspect tree $\mathcal{T}$, an estimated
probability $P(S)$ that S contains a buggy node

K_C is the number of instances of C in correct trees

S_C is the number of instances of C in $\mathcal{T}$

```
For each clause C ∈ 𝒞
        Let P0(C) be the prior likelihood of C being buggy

% P1(C) is the probability that an instance of clause C with
% a correct body is buggy, given that C is buggy
For each clause C ∈ 𝒞
        If C is ground P1(C) = 1, otherwise
        let 0 ≤ P1(C) ≤ 1 maximise (1 − P1(C))^KC (1 − (1 − P1(C))^SC)
```

Computing probabilities (cont.)

% Scale down relative likelihoods of clauses being buggy
% using number of instances in correct subtrees and P_1 values
For each clause $C \in \mathcal{C}$

$$P_2(C) = P_0(C)(1 - P_1(C))^{K_C}$$

% $P_3(C)$ is the probability that clause C is buggy
% given that at least one clause is buggy
For each clause $C \in \mathcal{C}$

$$P_3(C) = P_2(C)/(1 - \prod_{C' \in \mathcal{C}}(1 - P_2(C')))$$

% $P_4(S, C)$ is the probability that an instance of C in S is
% buggy
For each clause $C \in \mathcal{C}$ and subtree S of $\mathcal{T}$

$$P_4(S, C) = P_3(C)(1 - (1 - P_1(C))^{M_C}),$$ where

M_C is the number of occurrences of C in S

Computing probabilities (cont.)

% $P_5(S)$ is the probability that a clause instance in S is
% buggy

`For each subtree` S `of` $\mathcal{T}$

$$P_5(S) = 1 - \prod_{C \text{ in } S}(1 - P_4(S, C))$$

% $P(S)$ is the probability that a clause instance in S is
% buggy given that a clause instance in $\mathcal{T}$ is buggy

`For each subtree` S `of` $\mathcal{T}$

$$P(S) = P_5(S)/P_5(\mathcal{T})$$

Computing P_1

P_1 can be thought of as a measure of how “consistent” a bug is

Lots of bugs are very consistent but if P_1 is too high, search is directed away from “spasmodic” bugs

We pick P_1 to maximise the probability of the observations

We could use the median of the probability distribution instead

Or different percentiles depending on (eg) clause complexity

Or estimate prior probability distributions

The maximum method results in finding consistent bugs very quickly and behaviour like divide and query for spasmodic bugs

Comparative P_1 values

S_C	2	1	10	9	5	1
K_C	0	1	0	1	5	9
P_1 (max)	1.000	0.500	1.000	0.226	0.129	0.100
P_1 (med)	0.653	0.500	0.545	0.359	0.191	0.148
P_1 (U=.2,R=3)	0.781	0.561	0.722	0.430	0.212	0.165

S_C	100	99	50	1
K_C	0	1	50	99
P_1 (max)	1.000	0.045	0.014	0.010
P_1 (med)	0.505	0.300	0.024	0.017
P_1 (U=.2,R=3)	0.706	0.386	0.025	0.017

Size of first subtree probed for reverse

Algorithm	L=4 N=15	L=16 N=153	L=64 N=2145	L=256 N=33153
Divide and query	6	78	1081	16653
$P_1(C) = \{1.0,\ 1.0,\ 1.0,\ 1.0\}$	3	6	6	6
$P_1(C) = \{1.0,\ 0.8,\ 0.8,\ 0.8\}$	3	6	6	10
$P_1(C) = \{1.0,\ 0.5,\ 0.5,\ 0.5\}$	3	6	10	15
$P_1(C) = \{1.0,\ 0.1,\ 0.1,\ 0.1\}$	1	1	36	55
$P_1(C)$ med. $K_C = \{1,1,1,1\}$	6	10	21	28
$P_1(C)$ med. $K_C = \{1,1,5,50\}$	6	10	45	136
$P_1(C)$ max. $K_C = \{1,1,1,1\}$	6	21	171	1653
$P_1(C)$ max. $K_C = \{1,1,5,50\}$	3	21	171	1830

Buggy merge sort

```
merge_sort(Us, Ss) :-
        length(Us, N),
        msort_n(N, Us, Ss, _). % last arg should be []

% Ss is first N element of Us sorted, RestUs is the rest.
% First clause only used for merge_sort of empty list.
msort_n(0, Us, [], Us).
msort_n(1, [U|Us], [U], Us).
msort_n(N, Us, Ss, RestUs) :-
        N > 1,
        N1 is N // 2,
        msort_n(N1, Us, Ss1, Us2),
        msort_n(N1, Us2, Ss2, RestUs),               % BUG
        % N2 is N-N1, msort_n(N2, Us2, Ss2, RestUs), % OK
        merge(Ss1, Ss2, Ss).
```

Buggy merge sort (cont.)

```
% merge of two sorted lists
merge([], Ss, Ss).
merge([S|Ss], [], [S|Ss]).
merge([A|As], [B|Bs], [A|Ss]) :-
        A =< B,
        merge(As, [B|Bs], Ss).
merge([A|As], [B|Bs], [B|Ss]) :-
        A > B,
        merge(As, [B|Bs], Ss).                          % BUG
        % merge([A|As], Bs, Ss).                        % OK
```

The more consistent bug — two worst cases

```
?- wrong(merge_sort([⟨(up to) 56 elements⟩,3,5,8,1,2,4,6,7],
      [⟨(up to) 56 elements⟩,1,1,1,1,2,4,6,7])).  % 378 nodes
msort_n(2,[6,7],[6,7],[]) valid?  y
merge([8],[1],[1,1]) valid?  n
merge([],[1],[1]) valid?  y
Bug:  merge([8],[1],[1,1]) :- merge([],[1],[1]).
```

```
?- wrong(merge_sort([⟨120 elements⟩,3,5,8,1,2,4,6,7],
      [⟨120 elements⟩,1,1,1,1,2,4,6,7])).  % 830 nodes
msort_n(4,[3,5,8,1,2,4,6,7],[1,1,1,1],[2,4,6,7]) valid?  n
merge([3],[5],[3,5]) valid?  y
merge([8],[1],[1,1]) valid?  n
merge([],[1],[1]) valid?  y
Bug:  merge([8],[1],[1,1]) :- merge([],[1],[1]).
```

The more spasmodic bug

```
 ?-wrong(merge_sort([9,0,3,5,8,1,2,4,6,7],[0,1,2,3,4,5,8,9])).
merge([5,9],[8],[5,8,9]) valid?  y
msort_n(2,[2,4,6,7],[2,4],[6,7]) valid?  y
msort_n(10,[9,0,3,5,8,1,2,4,6,7],[0,1,2,3,4,5,8,9],[6,7])
            valid?  n
msort_n(5,[8,1,2,4,6,7],[1,2,4,8],[6,7]) valid?  n
merge([1,8],[2,4],[1,2,4,8]) valid?  y
merge([8],[1],[1,8]) valid?  y
msort_n(2,[8,1,2,4,6,7],[1,8],[2,4,6,7]) valid?  y
Bug:  msort_n(5,[8,1,2,4,6,7],[1,2,4,8],[6,7]) :-
            5 > 1,
            2 is 5 // 2,
            msort_n(2,[8,1,2,4,6,7],[1,8],[2,4,6,7]),
            msort_n(2,[2,4,6,7],[2,4],[6,7]),
            merge([1,8],[2,4],[1,2,4,8]).
```

. . . with prior tests cases

```
 ?-wrong(merge_sort([9,0,3,5,8,1,2,4,6,7],[0,1,2,3,4,5,8,9])).
msort_n(10,[9,0,3,5,8,1,2,4,6,7],[0,1,2,3,4,5,8,9],[6,7])
            valid?  n
msort_n(5,[8,1,2,4,6,7],[1,2,4,8],[6,7]) valid?  n
merge([1,8],[2,4],[1,2,4,8]) valid?  y
msort_n(2,[2,4,6,7],[2,4],[6,7]) valid?  y
merge([8],[1],[1,8]) valid?  y
msort_n(2,[8,1,2,4,6,7],[1,8],[2,4,6,7]) valid?  y
Bug:  ⟨as before⟩
```

Tarantula

Graphical tool; color of statement = $\frac{\%passed}{\%passed + \%failed}$

Could consider number of times code is used in each test case

For debugging, best ignore code which is never used in a failed test case and ignore test cases which only use such code

Percentages lose information

Declarative debugging could use more than one failed test case (suspect tree)

Conclusion

The relative performance of different search strategies can be explained by appealing to rationality: bottom-up $\rightarrow$ top-down $\rightarrow$ divide and query $\rightarrow$ our algorithm $\rightarrow \ldots$

For "spasmodic" bugs $\log N$ probes is the best we can do

For "consistent" bugs some search strategies perform much better

Passed and failed test cases can be used to help estimate bug consistency and adapt the search strategy accordingly

Probability theory can be used to control the search strategy and reconcile multiple sources of information

The variation in probe costs should be considered