Probabilistic Declarative Debugging

Lee Naish
University of Melbourne
Melbourne 3010, Australia
lee@cs.mu.oz.au

July 7, 2008

Abstract

We present a probabilistic approach to the search strategy for declarative debugging. We focus on diagnosing wrong answers in pure Prolog programs but the approach can be adapted to other languages (for example, functional languages) and bug symptoms. Drawing information from source code and the execution of passed and failed test cases, different search heuristics are combined using probability theory. This reduces the expected number and complexity of questions. We appeal to rationality to compare several search algorithms. Algorithms such as ours, which use more information and make weaker assumptions, generally have better performance. Examples are also given to illustrate how the algorithm can perform extremely well when certain heuristics are effective, and reasonably well even when the heuristics are ineffective.

Keywords: search strategy, divide and query, dicing, Bayes, logic programming

1 Introduction

Declarative debugging [20] has long been recognized as an attractive way to diagnose errors in logic, constraint and functional programs. The advantages are two-fold. First, the user is only required to understand the relatively simple declarative semantics, instead of the procedural semantics. Complex details of the execution such as backtracking, constraint propagation and lazy evaluation are not exposed to the user. It is only necessary for the user to be clear about what is true in their intended interpretation — what should be computed, rather than how it is computed. Second, the high level reasoning and decision making programmers traditionally perform is done by the debugging system — this is why it was originally named algorithmic debugging in [20]. The debugger ultimately draws the conclusion that a particular part of the source code is buggy by reasoning about bug symptoms (and lack of symptoms) in various parts of the execution. In order to do so, it must make a series of decisions about which parts of the execution should be examined for bug symptoms. The
debugger (hopefully) only examines parts of the execution which may help find the bug — it searches for the bug in a rational way. The term rational debugging is used in [18] [19] to emphasize this point.

In declarative debugging a computation can be seen as a tree and a diagnosis is a single “buggy” node in the tree corresponding to a step in the execution which is at odds with the programmer’s intentions. Even in the earliest work on declarative debugging (for Prolog) [20], several search strategies were proposed. The central idea we propose in this paper is a probabilistic approach to the search problem. We use various heuristics to estimate the probability of a buggy node appearing in a subtree and search the tree in a way which attempts to minimize the expected cost. Our use of the term “probability” may be questioned by some — bugs and their symptoms are arguably not the result of randomness during programming or execution. However, probability theory is a useful tool for reasoning with uncertainty. It gives us systematic methods for combining information from different sources, something we don’t have if we use numerical estimates of likelihood in a more ad hoc way. We give some examples to argue the superiority of our algorithm compared with others proposed. Although we focus on the problem of diagnosing wrong answers in pure Prolog programs without negation, diagnosing wrong answers in functional programs is very similar (see Section 2) and our algorithms can be applied directly. Diagnosing some other bug symptoms, such as inadmissible calls [18] [15] require only minor adaptation. For missing answers (incompleteness) [20] our probabilistic search algorithm could still be used, though the way probabilities are estimated would have to change more substantially. Similarly, application of the technique to other languages may require substantial changes in probability calculations.

Although we do report some experimental results, the contribution of this paper is not a practical implementation. The algorithms presented are based on an appeal to rationality and our implementation and experiments have primarily been used to validate the approach. Rationality is also used to explain the relative performance of various search strategies. We also present a game of dice which is analogous to part of the problem of debugging and reveals the importance of explicitly estimating how “consistent” bugs are (how often the execution of buggy code returns incorrect versus correct results).

This paper is structured as follows. We first give a more detailed background summary of declarative debugging and search strategies. Next we describe a search algorithm which is similar to a previously proposed algorithm but takes advantage of probability and cost estimates to reduce the total expected cost. Next we describe various sources of estimated probabilities and how they can be combined so as to be used by the search algorithm. We provide some experimental results to compare our algorithm with previously proposed algorithms. Finally, we discuss related work and possible further work including a more practical implementation which take resource limits into consideration, then conclude.
2 Background

Declarative debugging of wrong answers in Prolog uses the proof tree (see [9]) associated with the program execution. The nodes of a proof tree are Prolog atomic goals which have succeeded. The children of a node are the successful subgoals in the body of the matching clause used to prove the goal. Nodes may be correct (the goal is valid, all instances are true, in the user’s intended interpretation of the program) or erroneous (not valid) — we use the terminology of [14]. An erroneous node with no erroneous children is buggy. Buggy nodes correspond to clause instances which are not true in the intended interpretation — the body is valid but the head is not.

To diagnose wrong answers in functional programs an evaluation dependence tree (EDT) [17] is used. The relationship with diagnosis of logic programs is given in [13]: if the “flattening” transformation (converting functions of \( N \) arguments to predicates of \( N + 1 \) arguments and nested expressions into conjunctions) is applied to a functional program and a corresponding EDT the result is a logic program and a corresponding proof tree. For strict functional programs and deterministic logic programs the correspondence is exact. For non-strict evaluation of functional programs some subtrees are eliminated — unevaluated applications are leaves of the EDT. For nondeterministic evaluation of logic programs a single top level goal can have multiple answers and multiple proof trees. However, neither of these variations affect our debugging algorithm. Where we refer to (instances of) clauses, (instances of) equations can be used.

A declarative debugger searches the proof tree (or EDT) for a buggy node. It uses information from an oracle to determine if nodes are correct or erroneous. The oracle relies on asking the user to determine correctness of nodes, but generally stores the information so a question is not repeated. It may also allow more general assertions (see [4], for example) about correctness to be made. During the search the debugger finds out (via the oracle) that certain nodes are erroneous and certain others are correct. Every erroneous node is on a path of erroneous nodes going down the tree which terminates at a buggy node. It is normal to search for one bug at a time, so if an erroneous node is found everything except the subtree rooted at that node is eliminated from the search space. Similarly, nodes which are correct, and their descendants, are normally eliminated from the search space. We call these correct subtrees. Erroneous nodes in correct subtrees are rare and typical search algorithms cannot find them. We call the remaining search space the suspect part of the tree. It is a tree with an erroneous root with any correct subtrees removed.

Many search algorithms have been proposed. One of the simplest is a top-down left to right search of the tree. Given an erroneous node, we check the correctness of its children left to right. If an erroneous one is found we recursively search that subtree, otherwise the root of the tree is a buggy node and is returned. For a tree with \( N \) nodes, up to \( N \) questions may be asked. The divide and query algorithm [20] was designed to improve the worst case complexity. Instead of checking a child of the root, we find a subtree whose “weight” (normally the number of nodes) is as close as possible to half the weight of the tree,
and check the root of the subtree. If it is erroneous the rest of the tree can be eliminated and if it is correct the subtree can be eliminated; either way the size of the search is reduced by (approximately) half, resulting in $O(\log N)$ complexity. Constant factors depend on the branching factor — with a branching factor of $M$ the subtree may be of size $1/M$ rather than the ideal case of $1/2$ the tree. In practice, a top-down search is often comparable with divide and query, particularly if the children are ordered in a sensible way (not necessarily left to right), and is much easier to implement. Bottom-up search was also proposed, but this typically performs poorly.

In [18] [19] several other search heuristics were presented. As well as allowing the user to say a node is erroneous, it was suggested that they be able to say a particular sub-term is wrong. Detailed dependency information can then be used to determine where that sub-term was “produced”, and the search directed accordingly. Nodes which do not help produce the sub-term cannot be eliminated from the search space (since if they had failed the incorrect result would have been avoided) but can be given a lower priority in the search. It was also noted that buggy nodes are derived from buggy procedures or clauses. Potentially we can find a subtree which uses half the clauses. If it is erroneous the search space in terms of clauses has been halved. If it is correct the clauses used cannot be eliminated from the search space (since a buggy clause may have both buggy and correct instances) but, again, can be given a lower priority in the search. We call this idea static divide and query as it is based on the (static) source code, which is typically much smaller than the proof tree.

There are many other ideas for improving the search strategy (and we provide several more in this paper), so a question arises as to how we can best combine them. One possibility is to have several different search strategies implemented and have the user pick one. Although this may be suitable for expert users, non-experts are unlikely to know which strategy is best in a given situation. It is also somewhat counter to the philosophy of declarative debugging, where the responsibility for the search strategy is given to the debugger rather than the user. One of the advantages of our proposed approach is that it is relatively simple to combine many search ideas into the same framework. Our key idea is to estimate the probability of each subtree containing a buggy node and use a search algorithm which attempts to minimize the expected cost of diagnosis.

We first describe our search algorithm, which can be applied to any tree with probability estimates for each subtree. Different sets of bug symptoms, such as wrong answers, missing answers or both, and different languages can be diagnosed using different kinds of trees [14], so the search algorithm can potentially be applied to all of these. In Section 5 we suggest an algorithm for computing probability estimates for wrong answer diagnosis, where the tree is a proof tree.
3 Probabilistic Search

Our probabilistic search algorithm can be seen as a variation of binary search. The basic operation of the search algorithm is to probe some element of the search space to determine in which of two parts of the search space the target appears. In the case of classical binary search the probe will be a comparison and a sorted array segment is used so the two parts of the search space are the array segments to the left and right of the probed element. In the case of declarative debugging the probe involves determining the correctness of a node in the tree and the two parts of the search tree are the subtree and the rest of the tree. Determining which node to probe at each stage can require significant computation (possibly evaluating some objective function for each node in order to find the optimal one), but the probes themselves are the most important cost and this is what we attempt to minimize. We know the target exists, so if there is only one element we need no probes. The maximum number of probes to search a space of size $N$ by splitting into two parts of size $XN$ and $(1 - X)N$, where $0 \leq X \leq 1$, is thus:

$$m(1) = 0$$
$$m(N) = 1 + \max(m(XN), m((1 - X)N))$$

It is clear that $m(N)$ is monotonically increasing, which is sufficient to conclude that $X = 1/2$ is optimal, giving standard binary search where $m(N)$ is (around) $\log_2 N$.

3.1 Reducing the expected number of probes

We believe debugging is one of the many search applications where the average case behavior is more important than the worst case. Given a probability distribution for the target being in the different parts of the space, the expected number of probes (which should be close to the average number over many cases) is:

$$e(1) = 0$$
$$e(N) = 1 + Pr(< XN)e(XN) + Pr(> XN)e((1 - X)N)$$ (1)

To simplify presentation we have written the probability terms as if we were searching a sorted array (in general there is no simple $<$ relationship to specify a part of the search space) and assume $X$ is fixed for all probes (in reality, it may vary). If the probabilities are proportional to the search space size the optimal solution is again binary search. We give a more precise version of equation 1 later, but for now we make the (conservative) assumption that $e(N)$ is $O(\log N)$, in which case the optimal choice for $X$ minimizes

$$Pr(< XN) \log X + Pr(> XN) \log(1 - X)$$
The base for the logarithms doesn’t affect the optimal choice used but using base 2 allows simple comparison with binary search or divide and query in the ideal case. These algorithms are optimal if we assume probabilities are proportional to subtree (or sub-array) sizes. In the ideal case, where the probed subtree is exactly half the tree size ($X = 1/2$) this objective function has value $-1$. If the root node has four subtrees of equal size, three probes may be required to reduce the search space by a factor of four, which is $2/3$ of the ideal effectiveness. The expected effectiveness is somewhat better than the worst case, due to the possibility the buggy subtree will be probed earlier: the objective function value for each root child is $-0.81$ rather than $-2/3$ (with probabilities of $1/4$ for each subtree). Given additional clues as to the bug location we can calculate probabilities which are not just proportional to the subtree sizes and the performance may be better than the ideal case for divide and query. For example, if one of the subtrees has a bug probability of $3/4$ and the rest, collectively, have a probability of $1/4$ then probing the root of that subtree will give an objective function value of $-1.6$ and nodes within the subtree may given even better performance.

So far we have been assuming the probabilities sum to one. For debugging, the analogy with binary search breaks down somewhat when we consider there may be several buggy nodes and the sum of the probabilities will generally be greater than one. When we probe a node during debugging, the probability of that subtree will be eliminated from the search depends on the probability of that subtree having a bug. It does not depend on the probability of the rest of the tree having a bug. Thus we want to pick a subtree $S$ which minimizes the following objective function, where $X_S$ is the size of $S$ divided by the size of the whole suspect tree:

$$o(S) = \Pr(S \text{ has bug}) \log X_S + (1 - \Pr(S \text{ has bug})) \log(1 - X_S) \quad (2)$$

### 3.2 Refining the objective function

Equations 1 and 2 attempt only to estimate the number of probes (the traditional measure of the cost when comparing different search strategies) and are somewhat oversimplified, as discussed earlier. In reality, different probes have greatly differing costs. The correctness of some nodes can be determined quickly and easily, but others may require examining extremely large data structures, for example, and be very difficult and time consuming. Complexity of questions is one of the main problems of declarative debugging, and if this could be significantly reduced it would be of great benefit, even if there was a moderate increase in the number of questions. In our refined formulas we therefore use an estimate of the cost $c(N)$ of probing a node $N$. For the experiments reported in this paper we use a cost estimate based on the size $A$ of the atom in the node: $\log_8 (A + 7)$. This is conservative since it grows very slowly (multiplying the size by 8 just increases the cost by around 1) but can still have a significant effect on the complexity of questions asked. A faster growing function is probably more
realistic and would tend to ask even simpler questions, though more of them.

Divide and query can also use the size of atoms as well as the number of nodes to compute the weight of a subtree. However, this can often increase the average cost of diagnosis rather than reduce it. Probes generally split the tree less evenly, the part with fewer nodes having larger atoms. Unless there is a correlation between larger atoms and buggy (not just erroneous) nodes, the number of probes tends to increase. Furthermore, because there is a strong correlation between the size of adjacent nodes whenever large data structures are present, the probed node tends to be larger than average, leading to more costly probes. The examples we use in the rest of this paper illustrate this point.

The choice of probes, and hence the total cost, also depend on information we have gathered to estimate probabilities, and this information can grow as we perform the search. If we probe a node and it is correct, the extra information that some subtree \( S \) is correct can change the probability calculations when the rest of the tree is searched. Section 5 describes how we compute the probability that a subtree \( S \) of tree \( T \) has a bug, using information \( I \), written \( Pr(S, T, I) \).

The expected cost estimate and objective function can thus be defined as follows (\(|T|\) is the size of tree \( T \), \( T \setminus S \) is tree \( T \) with subtree \( S \) deleted and \( \text{correct}(S) \) is the information that subtree \( S \) is correct):

\[
e'(T, I) = \begin{cases} 0 & \text{if } |T| = 1 \\ \min_{S \in T} o'(S, T, I) & \text{otherwise} \end{cases} \\
o'(S, T, I) = c(\text{root}(S)) + Pr(S, T, I)c'(S, I) + (1 - Pr(S, T, I))e'(T \setminus S, I \cup \text{correct}(S))
\]

Ideally we would like to minimize this objective function, but applying these definitions naively takes time exponential in the size of the tree. Using dynamic programming methods to improve the complexity appears difficult due to the way the information \( I \) is gathered. Equations 1 and 2 can be seen as a first order approximation to equation 3, where the recursive calls to \( e' \) are replaced by logarithms, leading to complexity linear in the tree size, and the cost of each probe is one. Clearly, it would be possible to use a \( k \)th order approximation, using equation 3 for \( k \) levels of recursion with complexity \( O(N^k) \), and perhaps vary \( k \) depending on the tree size.

Using even a second order approximation is likely to be prohibitive most of the time. Experimentation has revealed that a first order approximation can work very effectively when there is a small subtree with a high probability of containing a bug. Although the estimated number of probes for the rest of the tree may be significantly over-estimated, this is multiplied by the (small) probability that the subtree is correct. For example, if there is a buggy fact (or equation with no function applications on the right side) in a program the search typically “zooms in” on a leaf of the tree quickly. However, if the buggy node is the root of the tree and the root has several large subtrees a first order approximation works poorly, even if the probability of the root being buggy is computed to be quite high. In order to find the bug we must ultimately probe
all of the children. In all probes but the last, both the subtree and the rest of the tree can be large. We need a $k^{th}$ order approximation, where $k$ is the number of children, in order to avoid the objective function significantly over-estimating the number of probes required. We need to “look ahead” $k$ probes to see the eventually reduction of the search space size to one.

To overcome this problem, we propose a refinement to equation 2 where the cost of each probe is considered and children of the root node are treated specially. This could be refined further and extended to more levels of the tree; even if applied to all nodes the complexity would be $O(N)$. We compute the probability $R$ of the root being a buggy node, which is just the product of the probabilities of each of its subtrees being correct (since the root is known to be erroneous). For each child, if it is correct we say there is probability $1 - R$ of the normal logarithmic cost and a probability $R$ of finding the bug by probing $S_1, S_2, \ldots, S_M$, the siblings of $S$. The minimum of this value and the normal objective function value is used. Note that the logarithmic cost estimates for the number of probes for a subtree $S$ is multiplied by $c_a(S)$, the average cost of probes in $S$.

$$o'(S, T, I) = c(root(S)) + Pr(S, T, I)c_a(S) \log |S| + (1 - Pr(S, T, I))o'(S, T)$$

$$o^*(S, T) = \begin{cases} C_{TS} & \text{if } S \text{ is not a root child} \\ \min(C_{TS}, o^*(S, T)) & \text{otherwise} \end{cases}$$

$$o^*(S, T) = (1 - R)C_{TS} + R \sum_{i=1}^{M} c(S_i)$$

where $C_{TS} = c_a(T \setminus S) \log(|T \setminus S|)$

### 4 Sources of Probability and Cost Estimates

The probabilities used in the search algorithm encapsulate domain knowledge. Our knowledge that the tree has a specific relationship with a program which is an artifact of human ingenuity (and fallibility) allows us to locate bugs more efficiently. In this section we discuss some of the multitude of sources of information we could use. The algorithm we present later uses some of this information explicitly some implicitly and some it ignores, but could potentially be incorporated in the future. There are two distinct kinds of probabilities we can estimate. The first is the correctness of sub-computations (nodes in the tree, which are Prolog atoms in our prototype). Note that if a node is erroneous a buggy node exists somewhere in the subtree but not necessarily at the root. The second is the correctness of sections of source code and their instances (some code may work correctly for some instantiation of variables but not others). We use (instances of) Prolog clauses. This is a more important source of information but is more complex to incorporate because multiple tree nodes are involved. There are also two ways we can obtain estimates. The first is what we call “static” —
unrelated to the program execution. The other is from the “dynamic” execution of the program using the test case(s) supplied to the debugger.

4.1 Static sources

The more important sources of information concerning the likelihood of an atom being erroneous are dynamic, and our current algorithm does not use static information for atoms. However, information about atoms can be incorporated in an ad hoc way by adjusting the cost function. Lowering the cost estimate for atoms considered more likely to be erroneous directs the search to those parts of the tree. Cost estimation can also be refined with information from the programmer and heuristics such as whether the procedure is part of an interface or just local (declarations or the static call graph could be used), whether the procedure has any comments, et cetera.

There are many heuristics for determining if a clause is incorrect. Users often have a reasonable feel for which procedures are suspect and this could be communicated to the debugger. Many static checks for suspicious code have been developed, for example, those incorporated into the NU-Prolog Debugging Environment [16], and could be used for estimating bug likelihood. More complex clauses (by measures such as total size, number of variables and number of conjuncts) are more likely to be buggy. If modification time of clauses or procedures is maintained by a programming environment this could also be used. Simple analysis of comments in code may also be very useful — many embarrassingly long bug searches end close to a comment containing the string “BUG” or similar. Our current algorithm uses information such as this implicitly — it assumes some method is used to compute relatively likelihoods of different clauses being buggy.

4.2 Dynamic sources

The debugger oracle is a very useful source of information about the truth value of atoms, especially if assertions have been provided. With a suitable oracle interface the whole tree can be traversed in search of nodes which can be determined to be erroneous or correct without asking the user any questions. Correct computations are also an excellent source of correct atoms; the root of the proof tree is known to be correct and is typically stored by the oracle, but every other atom in the tree is very likely to be correct also. Polymorphism in declared or inferred types can be used to generalize information about (probably) correct atoms. For example, if we know that append([1],[2],[1,2]) is probably correct (Figure 3 contains the definition) and append/3 is polymorphic in the type of the list elements, we can conclude append([X],[Y],[X,Y]) is probably correct for all X and Y. If users can specify incorrect terms as well as incorrect atoms, dependency information can be used to determine the producers of the term (in Mercury [22] and functional languages this can generally be computed statically). Even without dependency information we can still say a node in the tree containing this term is likely to be erroneous. The producers of the term
must be connected to the root of the tree via a path in which each node contains the term. Occurrences of the term not on such a path could be ignored or given less elevated probability of being erroneous.

Program slicing is based on the idea that bugs are likely to be in parts of the program which are executed in test cases which exhibit buggy behavior but are not executed in test cases which exhibit correct behavior. Originally introduced for imperative programs [23], it has also been applied to logic programs (see [5], for example). Given a proof tree, we can identify clauses used in the suspect part of the tree but not in correct parts of the tree and adjust probabilities accordingly. This is one of the key parts of our algorithm. Measures of how well a set of test cases “covers” a program can be used to refine this idea. Clauses can contain disjunctions and conditionals and a clause body succeeds using a particular execution path [21]. We could consider execution paths used in suspect but not correct parts of the tree, and preferably also note that different execution paths can share some parts, so they are not completely independent. We can also consider different variable instantiations within a single execution path. For example, if all instances of some clause in correct parts of the tree have a certain variable instantiated to the empty list, it does not give us such a strong indication of the correctness of an instance where that variable is a non-empty list.

Work on “most specific” logic programs [11] can be used to combine information on execution paths and clause instances. Clause instances are obtained from correct parts of the tree and disjuncts not on the execution path are replaced by a special value, “bottom”. All instances of each clause can be “anti-unified” to obtain a “more specific” version of the program (the least general instance of the program capable of reproducing all the correct parts of the tree). If a variant of a clause in the original program is obtained, that clause is covered; all paths have been used and every variable has been instantiated to distinct terms with different top level function symbols. Clause instances in the suspect part of the tree which are subsumed by the more specific clauses generated are very likely to be correct; others are less likely to be correct. More specific programs and path and clause usage are ways of summarizing the information in an execution tree using space related to the size of the program. More detailed analysis which considers every clause instance in the tree could also be used if resources permit.

Note that any method which uses correct parts of the tree can benefit from correct test cases as well as correct subcomputations within buggy test cases. Ideally, a programming environment should retain information about correct cases so it can be used to help diagnose incorrect cases. If this is not implemented, a similar result can be achieved by the user explicitly running a more complex goal which contains subgoals which are known to behave correctly, as well as the buggy subgoal. Estimates of the cost of probes could be refined by noting the time it takes to answer questions, whether users answer “don’t know”

\[1\] Technically, the number of execution paths can be exponential in the program size and a more specific program can be arbitrarily larger, but these do not seem to occur in practice.
(which the better debuggers support) and gathering explicit cost information from users.

4.3 Learning

The most appropriate way of interpreting the “probabilities” is as a guide to predictive power in the longer term, similar to weather forecasts which give a numerical chance of rain (a style of forecasting which is gaining popularity internationally). If the system computes the chance of a bug (or rain) is 10% in particular circumstances then (over time and/or places within a program) then there should be a bug (rain) in around 10% of the cases where those circumstances arise. If it turns out to be significantly more or less than this, the mechanism for estimating the “probability” (code within the debugging environment or weather model) should be adjusted accordingly. A debugging environment could maintain statistics so probability estimates could be refined over time, either manually or perhaps using machine learning algorithms. The suggestions we make here are a starting point for such refinements.

5 Computing Probabilities

We first discuss some naive approaches, which ignore certain information. Interestingly, this forms a rational basis for explaining the relative performance of various proposed search algorithms. Next we introduce a game of dice which is analogous to the debugging problem. Being a simpler domain where we have fewer preconceptions, this can clarify some issues of rationality and what information is used. Finally we present our algorithm and discuss some possible refinements.

5.1 Naive Approaches

Our search algorithm relies on an estimate of the probability of each subtree in the suspect part of the search space having a buggy node. A naive approach is to assume each node in the tree has a small probability \( \epsilon \) of being buggy. The probability of a subtree of \( N \) nodes having at least one buggy node can be calculated as \( 1 - (1 - \epsilon)^N \). Using the first order approximation of \( N\epsilon \) and assuming all probes have the same cost, our search algorithm is equivalent to the divide and query algorithm, as explained in Section 3.1. Unfortunately, at the heart of many probability calculations there is a dubious assumption of independence. Here \( (1 - \epsilon)^N \) assumes the probability of the \( N \) different nodes being buggy is independent. If we know nothing about the tree nodes or the program it was derived from this is the simplest assumption we can make and thus is rational according to Occam’s Razor. Thus divide and query has a rational basis if we assume there are few (or just one) buggy nodes and certain information is unavailable.
An example where divide and query is clearly not rational is a tree where we know more than half the nodes are in a subtree which computes an \texttt{append/3} call. It is not rational to abandon our (presumably high) expectation that \texttt{append/3} is correct, conclude an \texttt{append/3} node is most likely buggy and probe a node within that subtree. If we know a collection of nodes use the same procedure (or clause) we should generally not assume independent probabilities and the probability that one of the nodes is buggy should be bounded by the probability that the code is buggy. We discuss this further in Section 5.2.

If we have no information about the program, tree nodes or sizes of subtrees then it is reasonable to assume that the subtrees of the root node all have the same size, and top-down search is then the rational choice. The assumption that $\epsilon$ is small is based on our knowledge of how debuggers are typically used. With no knowledge of debugging the rational choice would be much larger, around 0.5, which would make bottom-up search the best choice. Thus the relative performance of bottom-up, top-down and divide and query search algorithms can be explained by an appeal to rationality which is consistent with our approach. All are rational given certain limited knowledge, with more knowledge corresponding to better performance.

Of course there is potentially far more information available about the tree, the program and debugging in general, though some effort may be needed to gather it. Our algorithm uses (some of) this additional information to further improve performance. If some information is inconvenient to obtain or hard to condense into a probability we can consider algorithmic factors when probabilities are estimated. For example, our algorithm inputs estimates of the (prior) probability of each clause being buggy. If the probabilities are high the resulting search is similar to bottom-up whereas if they are very low the search is more like divide and query. It is therefore safer to (if anything) under-estimate such probabilities. Our algorithm also scales up these probabilities (using the knowledge that at least one clause is buggy), so they can be grossly under-estimated without significant performance decrease.

5.2 An Analogy with Dice

An analogy in a more familiar domain is a collection of ten dice, nine of which are normal but one is blank on five sides. Dice are analogous to clauses, the defective one being buggy. Die throws are analogous to tree nodes (or clause instances); blanks are considered buggy and other results correct. If we repeatedly select a die randomly and throw it, the probabilities are independent and the chance of at least one blank in $S$ throws is $1 - (1 - 5/60)^S$, which approaches 1. If we select a die randomly then repeatedly throw the same die the probabilities are not independent and the chance of at least one blank in $S$ throws approaches $1/10$.

More generally, the debugging problem is analogous to having $N$ dice (which may have blanks on any number of sides and may be loaded), corresponding to the $N$ clauses (or equations \textit{et cetera}) in the program. Each die $i$, $1 \leq i \leq N$ is thrown $S_i$ times, $S_i > 0$ and we are told at least one throw of some die is blank;
this corresponds to a suspect tree in which each clause \( i \) is used \( S_i \) times. Each die \( i \) may also be thrown \( K_i \) more times, \( K_i \geq 0 \), with no blanks occurring, corresponding to trees which are known to be correct. Given this information, the challenge is to estimate the following probabilities:

1. the probability that each die \( i \) is defective and
2. the probability a given throw of die \( i \) comes up blank, given that it is defective.

Determining the most rational estimates is complex. We may also have some additional information available. For example, we may be told that defective dice are very rare and this may lead us to introduce simplifying assumptions, such as only one of the dice is defective (there is only one bug).

The first probability is clearly related to which parts of the program are most likely to be buggy. The second is more subtle but important because during debugging we observe instances of code which is executed and the ultimate diagnosis in declarative debugging is an incorrect clause (or equation) instance: the debugger works with throws of dice rather than the dice per sé. Essentially, it is a measure of how “consistent” a bug is. A high value means most executions of the buggy code produce incorrect results, whereas a low value means the bug is spasmodic, being evident in relatively few executions. The consistency of bugs greatly influences the performance of some search strategies, for example, the effectiveness of program slicing.

Note that this measure of bug consistency is a conditional probability. Bayes theorem states the probability of \( A \) given \( B \), written \( Pr(A|B) \) is

\[
Pr(A|B) = \frac{Pr(A)Pr(B|A)}{Pr(B)}
\]

where \( Pr(A) \) and \( Pr(B) \) are the prior probabilities of \( A \) and \( B \), respectively (making no assumptions about the truth of the other proposition). The probability that a clause (die) is buggy given that a corresponding node (throw) is buggy is simply 1. For the dice example above, \( Pr(\text{Blank throw}| \text{Buggy Die}) = (5/60) \times 1/(1/10) = 5/6 \). Using Bayes theorem we can devise a method where, as the \texttt{append/3} subtree size increases, the probability estimate approaches the estimate for the definition being buggy. In our technique several such refinements are used, incorporating the various sources of probability estimates given earlier. From one perspective we are incorporating more knowledge into the search algorithm. From another perspective we are weakening the assumptions of independence. Although some assumptions of independence remain, they are certainly much weaker and more reasonable than those used to justify algorithms such as divide and query from a probabilistic perspective. Similarly, although explicitly estimating prior probabilities may be difficult to do accurately, the alternative is to assume all prior probabilities are equal, which is generally even less accurate.
Algorithm *subtree_probs*:

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

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

% *$P_0(C)$* is the initial estimate of relative likelihood of clause $C$
% being buggy (compared to other clauses); can be much less than
% the actual probability

For each clause $C \in C$

% Let $P_0(C)$ be the prior likelihood of $C$ being buggy

% *$P_1(C)$* is the probability that an instance of clause $C$ with
% a correct body is buggy, given that $C$ is buggy

For each clause $C \in C$

% If $C$ is ground $P_1(C) = 1$, otherwise
% let $0 \leq P_1(C) \leq 1$ maximize $(1 - P_1(C))^{K_C}(1 - (1 - P_1(C))^{S_C})$, where
% $K_C$ is the number of instances of $C$ in correct trees and
% $S_C$ is the number of instances of $C$ in $T$

% Scale down relative likelihoods of clauses being buggy
% using number of instances in correct subtrees and $P_1$ values

For each clause $C \in C$

% $P_2(C) = P_0(C)(1 - P_1(C))^{K_C}$, where $K_C$ is as above

% *$P_3(C)$* is the probability that clause $C$ is buggy
% given that at least one clause is buggy

For each clause $C \in C$

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

% *$P_4(S, C)$* is the probability that an instance of $C$ in $S$ is buggy

For each clause $C \in C$ and subtree $S$ of $T$

% $P_4(S, C) = P_3(C)(1 - (1 - P_1(C))^{M_C})$, where
% $M_C$ is the number of occurrences of $C$ in $S$

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

For each subtree $S$ of $T$

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

% *$P(S)$* is the probability that a clause instance in $S$ is buggy
% given that a clause instance in $T$ is buggy

For each subtree $S$ of $T$

% $P(S) = P_5(S)/P_5(T)$

Figure 1: Algorithm for computing probabilities of subtrees being buggy
5.3 Our Algorithm

The algorithm is given in figure 1; here we discuss various details. As well as the inputs described in figure 1, we assume a method for computing prior relative likelihoods of clauses being buggy based on (possibly) the code and additional clues as to likely bug locations from the user or other sources. Ideally, we would like to start with accurate prior probabilities \( P_0(C) \) of each clause \( C \) being buggy. Unfortunately, this is rather difficult, though the machine learning suggestion in Section 4.3 may help. However, our approach still works well if these prior probabilities are underestimated by a constant factor. Instead of attempting to estimate accurate probabilities it is sufficient to choose relative likelihoods of clause being buggy which are much smaller than actual probabilities (around one millionth, say). Static information and possibly information from the programmer can be used to increase or decrease \( P_0(C) \) compared to this base value. We don’t propose any systematic method for this, but suggest a factor of more than ten with just static sources is placing too much reliance on them. If the static analysis is accurate it should be reinforced by the dynamic information anyway.

As mentioned, to weaken assumptions of independence and obtain a measure of bug consistency we can estimate the probability of a clause instance being buggy given that the clause is buggy (essentially the proportion of instances of the clause in the suspect tree which are buggy). Furthermore, we can ignore clause instances for which the body is false. An instance where both the head and body are false is indicative of a bug somewhere lower in the tree but is not buggy in itself. Thus, for each suspect clause \( C \), we estimate the probability \( P_1(C) \) of any instance of \( C \) in the tree being buggy given that \( C \) is buggy and the body instance is true. With very low \( P_1 \) values, the probability of a subtree containing a bug is more related to the number of clause instances used (the size of the subtree) than the number of clauses, and our search algorithm behaves like divide and query. With high values, our search algorithm gives more weight to the program slicing heuristic and behaves more like static divide and query. We treat ground clauses as a special case since the probability is clearly 1. For non-ground clauses we pick a value which gives the “best” explanation the observations: the choice of \( P_1(C) \) maximizes the probability that all occurrences of \( C \) in correct trees are correct and least one occurrence in the suspect subtree is buggy. We discuss alternative ways of computing \( P_1(C) \) in section 5.4.

The probability of a clause \( C \) being buggy given static information and information that certain nodes contain true atoms (correct computations) is the prior probability of \( C \) being buggy times the probability of the nodes being true given that \( C \) is buggy. These \( P_2(C) \) values are based on the \( P_0(C) \) values and are thus are relative likelihoods rather than realistic probabilities. They are further reduced by a factor dependent of the number of times \( C \) is used in correct trees \( (P_2(C) = P_0(C) \) if \( C \) is not used in any correct tree). The algorithm assumes the correctness of instances of a buggy clause at different tree nodes are independent. It could be refined by considering only the number of distinct instances of a clause or using more sophisticated measures of similarity. For
example, if a subset of the instances are considered to cover the clause then subsequent instances could affect the overall probability less. It is possible that correct trees contain buggy clause instances and our algorithm could be refined to take account of this. For example, slightly smaller $P_1$ values could be used (currently it is assumed all $K_C$ instances of clause $C$ are correct).

Knowing at least one clause is buggy allows us to scale up each (possibly tiny) value by the same factor so they become realistic probability estimates. The sum of the $P_3(C)$ values will always be greater than 1 but if the $P_0$ (and $P_2$) values approach zero the sum approaches 1. Thus choosing very small $P_0(C)$ values corresponds to assuming there is a single buggy clause and simply scaling the $P_2$ values so the sum is 1 is a reasonable alternative to our more complex formula for $P_3$ (and is what our current implementation uses). From the perspective of the debugger we are always searching for a single bug and it does not hurt to assume every other clause is correct. Also, it is helpful to optimize the search for the “last” bug (the hardest one to find) and divide and query essentially assumes an extremely low probability. When there are many bugs a higher probability would be appropriate, but in this case we can often find one easily, with a simple test case, a small proof tree and few questions even with a far from optimal search strategy.

The way we compute the probability of a subtree $S$ of the suspect part of the tree containing a buggy instance of clause $C$ (given static information and information from correct computations), $P_4(S, C)$ assumes independence (as with the correct trees, we could just consider distinct clause instances, et cetera). Similarly, computing $P_5(S)$, the probability that at least one clause instance in $S$ is buggy, we assume that the probabilities of different clauses being buggy is independent. The $P_5$ values are scaled up to give the final probabilities using the information that at least one clause instance in the suspect subtree is buggy.

As mentioned previously, we currently only support atom-based heuristics by ad hoc adjustment of the function which estimates the cost of a single probe. We simply make a few brief comments here. First, care must be taken with assumptions of independence. Clearly, atoms are more likely to be erroneous if they are derived from procedures which are likely to be buggy. However, the search algorithm already takes this into account; only information which is independent of clause probabilities should generally be used. Second, applying atom-based heuristics just locally within the tree is unlikely to be as effective as a more global analysis. For example, assume the user says atoms containing 42 are considered likely to be erroneous. If there are only a few such atoms in the tree it may be reasonable to change the search significantly, whereas if there are many such atoms (especially if some are found in correct sub-computations) less attention should be paid to the information. Third, procedural information may be used to affect probabilities. For example, a procedure which outputs 42 is more likely to be buggy than one which inputs 42 given the information above. Although declarative debugging can be done with only declarative information, procedural information can provide heuristics to help the search.
Figure 2: $P_1$ values computed using maximum, median and Bayesian methods

### 5.4 Alternative methods of computing $P_1$

In our first experiments we used a fixed $P_1$ value for all clauses. Many buggy clauses have all their instances buggy, so a relatively high $P_1$ value is appropriate. However, for spasmodic bugs there can be many correct instances of the clause used in the test cases and few (perhaps just one) buggy instances. Having a constant $P_1$ value which is too high directs the search away from such bugs, leading to poor performance — the estimated probability that a clause $C$ is buggy, $P_3(C)$, becomes very small when many correct instances are found. With $P_1 = 0.8$, just ten correct occurrences of a clause reduces its $P_3$ value by a factor of around ten million. It is a mistake to explain correct instances of a clause $C$ only by a (very) small $P_3(C)$ value. Instead, they should be explained by a combination of small $P_3(C)$ and $P_1(C)$ values.

With our proposed method of maximizing the probability of the observations, $P_1(C) = 1$ if there are no known instances of $C$ in correct computations. As the search progresses, $P_1(C)$ drops (sometimes quite substantially) when instances of $C$ are found in a correct computation and increases (if it is not 1) when suspect instances of $C$ are eliminated from the search space. Instead of using the value corresponding to the maximum probability (for discreet values it would be called the mode), we could use some other measure of central tendency. For example, we could use the median, the 50th percentile of the integral of $(1 - P_1(C))^{K_C} (1 - (1 - P_1(C))^{S_C})$ between 0 and 1. Figure 2 gives some sample $P_1$ values computed using both maximum and median methods.

Alternatively, the Bayesian philosophy (which we have used elsewhere) could be embraced fully. In the Bayesian view we have prior expectations of $P_1$ values and these expectations, not just the observations, should be taken into account. With one correct occurrence and 99 suspect occurrences, the “best” explanation (using the maximum method and ignoring any prior expectations about $P_1$; see Figure 2) is only 4.5% of the instances are buggy. This intuitively may seem rather low, because most bugs we encounter are exhibited in more than

<table>
<thead>
<tr>
<th>$S_C$</th>
<th>$K_C$</th>
<th>$P_1$ maximum</th>
<th>$P_1$ median</th>
<th>$P_1$ Bayesian U=0.2, R=3</th>
<th>$P_1$ Bayesian U=0.5, R=50</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>1.000</td>
<td>0.653</td>
<td>0.781</td>
<td>0.888</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.500</td>
<td>0.500</td>
<td>0.561</td>
<td>0.610</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1.000</td>
<td>0.545</td>
<td>0.722</td>
<td>0.866</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0.226</td>
<td>0.329</td>
<td>0.430</td>
<td>0.508</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0.129</td>
<td>0.191</td>
<td>0.212</td>
<td>0.271</td>
</tr>
<tr>
<td>1</td>
<td>9</td>
<td>0.100</td>
<td>0.148</td>
<td>0.165</td>
<td>0.209</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>1.000</td>
<td>0.505</td>
<td>0.706</td>
<td>0.863</td>
</tr>
<tr>
<td>99</td>
<td>1</td>
<td>0.045</td>
<td>0.300</td>
<td>0.386</td>
<td>0.491</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>0.014</td>
<td>0.024</td>
<td>0.025</td>
<td>0.032</td>
</tr>
<tr>
<td>1</td>
<td>99</td>
<td>0.010</td>
<td>0.017</td>
<td>0.017</td>
<td>0.021</td>
</tr>
</tbody>
</table>
4.5% of the instances of the clause. Our expectations can be modelled by a probability distribution. For example, very simple code which manipulates data structures commonly has consistent bugs (though spasmodic bugs are still possible) whereas complex numerical code has spasmodic bugs relatively often. Our implementation supports distributions which are linear in the interval [0, 1), with a special case for 1. The function in the interval is specified by the ratio $R$ of the prior probabilities of very consistent bugs to very inconsistent bugs. Additionally, a value $U$ gives the prior probability of a totally consistent bug (this becomes irrelevant as soon as an incorrect instance is found). The median method corresponds to having a uniform probability distribution for $P_1 (U = 0, R = 1)$. A low $R$ value such as this may be appropriate for some complex clauses. For simpler clauses, higher values for both $U$ and $R$ are reasonable — Figure 2 includes two examples. For ground clauses $U = 1$ is the rational choice.

Although the maximum method seems less rational, it often performs better than the other methods. For simple bugs where the program slicing heuristic is effective and there are no correct instances of the buggy clause, the very high $P_1$ value leads to outstanding performance. For spasmodic bugs where every clause has multiple correct instances and the search space is large, the $P_1$ values quickly become small. They typically underestimate the true values, but this simply makes the search algorithm more like divide and query, which is a good default when heuristics are not effective. We conjecture that our algorithm with the maximum method asymptotically has the same number of probes as divide and query if the size of the program is fixed. Going back to the dice game analogy, the maximum method is better than we might expect because the payoff is not symmetric. Consistently overestimating $P_1$ can be much worse than underestimating it and a rational choice should take this into consideration. The dice game we described relates to a single probe; a more complex version is required to model strategies over several probes. For simply ranking clauses according to the probability of them being buggy (see Section 7) we would expect the method using prior probability distributions to be most effective.

6 Experiments

A simple thought experiment illustrates how our algorithm performs far more rationally than top-down search or divide and query when the tree is much larger than the program. Consider the case where the suspect subtree is a naive reverse computation with a list length $L$ (see Figure 3.) Note that although we normally consider this code to be correct, someone may have written this code (or equivalent) with different intentions in mind. The tree has height $L+1$ and number of instances of the different clauses is 1, $L$, $L$ and $L(L-1)/2$, respectively; the number of nodes $N = (L+1)(L+2)/2$. With top-down search which selects the recursive child first, the next probe would be naive reverse computation with a list length $L-1$. Divide and query attempts to
% 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).

Figure 3: Definition of naive reverse

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>L=4</th>
<th>L=16</th>
<th>L=64</th>
<th>L=256</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>N=15</td>
<td>N=153</td>
<td>N=2145</td>
<td>N=33153</td>
</tr>
<tr>
<td>Divide and query</td>
<td>6</td>
<td>78</td>
<td>1081</td>
<td>16653</td>
</tr>
<tr>
<td>(P_3(C) = {1.0, 1.0, 1.0, 1.0})</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>(P_3(C) = {1.0, 0.8, 0.8, 0.8})</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>(P_3(C) = {1.0, 0.5, 0.5, 0.5})</td>
<td>3</td>
<td>6</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>(P_3(C) = {1.0, 0.1, 0.1, 0.1})</td>
<td>1</td>
<td>1</td>
<td>36</td>
<td>55</td>
</tr>
<tr>
<td>median, (K_C = {1,1,1,1})</td>
<td>6</td>
<td>10</td>
<td>21</td>
<td>28</td>
</tr>
<tr>
<td>median, (K_C = {1,1,5,50})</td>
<td>6</td>
<td>10</td>
<td>45</td>
<td>136</td>
</tr>
<tr>
<td>maximum, (K_C = {1,1,1,1})</td>
<td>6</td>
<td>21</td>
<td>171</td>
<td>1653</td>
</tr>
<tr>
<td>maximum, (K_C = {1,1,5,50})</td>
<td>3</td>
<td>21</td>
<td>171</td>
<td>1830</td>
</tr>
</tbody>
</table>

Figure 4: Size of first subtree probed for naive reverse

halve the tree size, the list length being reduced by a factor of (around) \(\sqrt{2}\),
to 0.71L. With \(L = 256\) (\(N = 33153\)), the first probe for divide and query
will be the subtree which computes the reverse of a list of length 181 (16653
nodes). Although the tree has just half as many nodes, common sense suggests
it is still almost certain to be buggy. It is hard to think of a realistic intended
interpretation of such a simple four clause program for which the larger tree
exhibits a bug but the smaller tree does not. Divide and query with weights
computed using atom size will probe an even larger subtree, with a larger atom
at its root.

The behavior of our algorithm depends in part on the \(P_0(C)\) estimates for
each of the four clauses. If we have no clues as to the bug location and use the
same very low \(P_0(C)\) for each clause and have no known correct subtrees, the
\(P_3(C)\) values will be around 0.25. Assuming a constant \(P_1(C)\) value of 0.8 for all
(non-ground) clauses, the probability of the reverse subtree for list length 2 (6
nodes) being buggy is close to 0.95. With such a high probability of this subtree
being buggy, probing a node higher in the tree is only worthwhile if there are
several thousand nodes. Figure 4 shows how the tree size relates to the size of
the first subtree probed for divide and query and variations of our algorithm
with different ways of computing the \(P_1\) values. Lower \(P_1\) values generally lead
to somewhat larger subtrees being probed (there are some exceptions because
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), % correct
    merge(Ss1, Ss2, Ss).

% 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). % correct

Figure 5: Buggy merge_sort/1 definition

the reverse/2 fact is ground and hence always has a $P_1$ value of 1). Using
the maximum method, with 50 correct and 32640 suspect instances, $P_1$ for the
append/3 rule is 0.0002. Even with such tiny values, that rational choice for
the next probe is around one tenth the size of that used by divide and query.

Figure 5 shows a (slightly) larger example, a buggy definition of merge sort.
The code initially computes the length of the list then uses msort_n/4 to sort
the first N elements and return the rest of the list (which should be empty at
the top level); this method avoids multiple list scans to split lists in half. There
is a “cut and paste” error in the definition of merge/3 which results in larger
elements being replaced by smaller ones; all instances of this clause are buggy
and buggy nodes have just one child (we assume calls to built-in procedures
are ignored for debugging purposes). The recursive case for msort_n/4 is also
buggy, losing an element from the sorted list if N is odd. This bug will show up
if the list length at the top level is not a power of two but not all instances of
the clause are buggy, it appears higher in the tree and buggy nodes have three
Figure 6: Finding the consistent \texttt{merge\_sort/1} bug: two “worst case” sessions children.

Suppose we initially have a single test case for \texttt{merge\_sort/2} with a list of length some power of two, use very small $P_0$ values with facts all having values 1/4 of those used for all the rules and the maximum method for computing $P_1$. For lengths of 4 to 64 (tree size 15 and 378, respectively), the first probe will be a \texttt{msort\_n/4} node where the first argument is 2 (subtree size 5, $P$ value of 0.68). There are typically several such nodes with the same probabilities and the search algorithm ensures the one with the least cost (smallest atom) is probed. It is the rightmost such atom in the tree and the list in the second argument has two elements; the leftmost such atom has an additional 62 elements in the second and fourth arguments. The \texttt{msort\_n/4} node may be correct or erroneous, depending on whether the last two elements of the input list are sorted or not. If the node is erroneous there is a subtree which uses the buggy \texttt{merge/3} clause and this is probed next, followed by its child, and the bug is found. If the \texttt{msort\_n/4} node is correct the remaining suspect tree is much larger (373 nodes) but the probabilities are recomputed so there is a better idea of the likely bug location. The buggy clause and the \texttt{merge\_sort/2} clause have higher $P_3$ values and much higher $P_1$ values than the other clauses (the two recursive \texttt{merge\_3} clauses have $P_3$ values of 0.22 and 0.23 and $P_1$ values of 0.03 and 1). The second subtree probed again has just a single occurrence of the buggy clause (see Figure 6). Although the computed probability is not particularly high ($P = 0.35$), the fact that only two probes would be required to find the bug and the cost of both are relatively small make it the optimal choice. Thus only three simple questions are asked in all cases compared with 7 significantly more complex questions for divide and query. Using our conservative logarithmic cost function our algorithm out-performs divide and query by a factor of about 3. If a linear cost measure is used the factor is between 12 and 27, depending on where in the tree the buggy node is found.

For a 128 element list (around 830 tree nodes) divide and query asks 8 ques-
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]).

Figure 7: Finding the more spasmodic bug without prior test cases.

tions, with the extra question almost as large as all the other questions combined. With our algorithm the first probe is typically the smallest msort_n/4 node with first argument 4 which uses both merge/3 recursive clauses (if this node has a high cost or no such node exists the questions are the same as for shorter lists). This node must be erroneous. The next probe is a merge/3 subtree of size two; if it doesn’t use the buggy clause the next probe is to similar subtree which does. The second example in Figure 6 illustrates this worst case for a 128 element list: a more complex first question and an extra question about merge/3. These examples demonstrate the algorithm has outstanding performance when the program slicing heuristic is effective.

Having found and corrected the bug in merge/3, the programmer may make the mistake of deleting all the information gained from the debugging session then later discover merge_sort/2 is still buggy. Figure 7 shows a sample debugging session chosen to illustrate how our algorithm can perform less well. Despite the relatively large number of probes, we believe each choice of probe is reasonable under the circumstances; in some sense the choices are unlucky rather than foolish. There are 15 instances of the buggy clause used but only 2 are incorrect (those used for sorting five elements). They are at level 3 (of 7) in the tree and are neither top-most nor bottom-most occurrences of this clause.

The first probe is a merge/3 subtree of size 3 (the merge/3 bug in the previous example distorted subsequent clause usage in merge/3, resulting in a different first probe). The second probe is the smallest msort_n/4 node with 2 in the first argument and which uses a different merge/3 fact to that in the first probe. Although this node is correct, the probe is not wasted because it is a child of the buggy node which is eventually found. At this point, all clauses but that of merge_sort/2 are known to have correct instances, leading to the child of the root being probed. The rest of the search proceeds as divide and query would, except for the fact that the first question about msort_n/4 has already been
?– 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?
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)

Figure 8: Finding the more spasmodic bug with prior test cases

asked. Thus our algorithm results in two more questions than divide and query. If the list is duplicated (the length is doubled) our algorithm asks the same questions, only one more than divide and query, and if the length is quadrupled the same number of questions asked for both algorithms. For examples such as these there are several buggy $msort\_n/4$ nodes at the same level of the tree, with subtrees of similar sizes. Our algorithm generally finds the right-most one, since the right-most $msort\_n/4$ nodes have the smallest atoms and least costly probes. Because $msort\_n/4$ nodes have three subtrees, each with strictly less than half the nodes, divide and query generally finds the one with the greatest weight, usually requiring much more complex probes, especially if weights are computed using atom sizes.

Our final example debugging session, Figure 8, locates the same bug, but assumes the debugger has information from two prior (correct) test cases for $merge\_sort/2$ with 16-element lists. There are only two instances of the top level $merge\_sort/2$ clause, thirty instances of the buggy clause and between 15 and 43 instances of the other clauses. The $merge\_sort/2$ clause therefore has the highest $P_3$ value and by far the lowest $P_1$ value, and its child is the first probed. The rest of the probes are same as those used by divide and query due to the relatively low $P_1$ values. The first probe can be eliminated by having more small test cases, increasing the number of correct occurrences of the $merge\_sort/2$ clause without greatly increasing the count for the other clauses. Even with somewhat misleading prior test cases and bugs for which our heuristics are ineffective, the performance of our algorithm is typically not much worse than divide and query and in cases where the heuristics are effective its performance can be dramatically better, especially if the tree is very large.

7 Related work

We have already compared our work with related declarative debugging work such as divide and query. One of the contributions of declarative debugging is to view the computation as a tree rather than a sequence of states. The tree nodes can be queried for correctness in a natural way due to the semantics of the programming languages it has been applied to. Complex execution strategies such as coroutining, backtracking and lazy evaluation can be hidden and
strategies for searching the tree can be automated.

There is also some related work on bug localization which has been applied to imperative programs. The Tarantula system [6] takes information about which statements are used in successful and failed test cases and displays the program statements with varied color and brightness to be used as a heuristic for (manual) debugging. Color is used to indicate how likely a statement is buggy, varying from red through yellow to green. Brightness is used to indicate confidence — if a statement is used in all successful or all failed test cases it will maximally bright; if it is never used it is black. By comparing Tarantula with our work it can be seen that both can be improved by using more information, and Tarantula could use a more rational approach to computing colors.

In comparing Tarantula and a similar visual debugging system \( \chi \)Slice [3] it was noted that Tarantula used multiple failed test cases to identify likely bugs whereas \( \chi \)Slice uses just one [6]. Similarly, declarative debugging systems use a single suspect subtree. There is a general assumption that probes are costly and hence there is no advantage in searching for more than one suspect subtree. However, some probes cost little (if the oracle is partially automated), and multiple suspect subtrees can give us additional information which can be used to improve the search strategy. For example, the probabilistic approach could potentially be modified to use this information, or the suspect subtree with the lowest estimated cost could be searched with our proposed algorithm. Although Tarantula may use more test cases, each test case provides less information — it is known whether a particular statement is used in each test case, but not how often (this is known as “binarization”). It would be reasonable to say that a statement being used 100 times in a single passed test case provides more evidence of its correctness than it being used once in each of two passed test cases. Clearly this information could be gathered and Tarantula modified to make use of it.

The color of a statement in Tarantula is determined by the percentage of passed test cases which use the statement divided by the sum of the percentages of passed cases using the statement and failed cases using the statement. Color discrimination between different suspect statements is reduced by increasing the number of passed test cases which don’t use the code. An extreme case is a system with two entry points, one of which calls only correct code and the other calls buggy code. By executing enough test cases for the correct entry point, all the code using in incorrect test cases can be made uniformly red. Code which is not used in any failed test cases (and test cases which only use such code) may be useful for validation but are best ignored for debugging purposes.

A more subtle criticism is the relative color of two statements used in the same number of failed test cases depends only on the ratio of the number of passed test cases they have been used in. Suppose there is a single failed test case which uses two statements, \( S_1 \) and \( S_2 \). The same colors (and brightness) are used whether \( S_1 \) and \( S_2 \) are used in one and four passed test cases, respectively, or in 25 and 100 passed test cases. However, it would be rational to have more distinction between the two statements in the latter case, where there is more information. In our probabilistic approach (using equal tiny \( P_0 \) values
and the maximum method for \( P_1 \) the ratio of \( P_3 \) values is 1:0.328 and 1:0.254, respectively. As the number of passed test case grows the \( P_3 \) ratio approaches the ratio of the number of passed test cases, but for small numbers of tests this approximation is inaccurate.

The metric used for ranking in Tarantula is just one of many which have been proposed for debugging and also other domains [12] and some of them have been compared empirically for debugging [1][2]. Effectiveness of declarative debugging overall cannot easily be compared with such systems because declarative debugging uses a tree to structure the computation and support an iterative search for the bug instance. However, our method of computing \( P_3 \) values could be used just for ranking, allowing (simplistic) comparison. We would expect the more sophisticated ways of computing \( P_1 \), using prior probability distributions, would perform better than the other methods we have suggested.

The SOBER system [8] is also a ranking method which uses probability theory, philosophically similar to ours, but uses different information. Instead of using statement counts it uses counts of the truth and falsity of certain predicates, such as conditions in if-then-else statements, and these predicates are ranked. The “probability” that each predicate is true is determined in successful and failed test cases; differences are used for ranking (more complex ranking metrics have also been proposed [7]). For example, if our buggy merge/3 was coded using a conditional, the predicate \( A>B \) would be always false in passed test cases (probability zero) but would have a significantly higher probability for failed test cases. Although such information may not point directly to the bug, it can be very helpful for a programmer. One advantage of the approach is that it allows additional domain knowledge to be incorporated. For example, predicates can be automatically added to compare the result of a C function with zero — something which often has semantic significance. It may be that diagnosis methods using predicates and statement execution counts of could be combined to good effect. For example, by adding dummy conditionals which do nothing but test a predicate, the same domain knowledge can be reflected in the statement execution counts of “then” and “else” branches. However, for the most spasmodic bugs, it is doubtful that any ranking method can perform well overall as the signal to noise ratio in the data gathered will be extremely small. Impirical studies [2] comparing accuracy of ranking with “error detection accuracy”, \( q_e \) (essentially, a binarized approximation to \( P_1 \)) suggest some ranking methods still perform reasonably well with \( q_e \) as little as 1%, which is encouraging.

Delta debugging [24] is very effective at diagnosing certain classes of bugs. This method runs the program multiple times with different subsets of the bug-inducing input in order to find two very similar inputs, only one of which exhibits the bug symptom. An oracle is needed to determine if the bug symptom appears in a run — if the symptom is a program crash this is simple (note that declarative languages attempt to automate such things as memory management which are the main cause of crashes). Failure cause-effect chains are then used to provide a relatively concise explanation of the bug symptom. These are state-oriented and it is not clear how usefully they could be adapted to the declarative
paradigm. However, any method which can simplify the test data which causes a bug symptom is useful in any diagnosis system.

8 Further Work

There are clearly many areas in which our work can be extended. One important area we have discussed is estimating the consistency of bugs \((P_i)\) values. The way we have approximated Equation 3 to obtain our objective function could also be revisited. Currently we use the logarithms of subtree sizes, which is reasonable if the bug is expected to be spasmodic. However, for consistent bugs the number of probes is more like the logarithm of the number of clauses used in the subtrees; some interpolation between these two based on probability estimates could be used. Heuristics for estimating prior probabilities and probe costs also need development. Another possible source of probabilities is to allow users to give probabilistic answers to questions rather than “don’t know” (or definite answers which may be incorrect).

Any real application of this work will also have to address resource considerations, particularly memory. It is not feasible to store the whole tree explicitly, yet our current algorithm relies on it. Divide and query suffers from the same potential problem, but the Mercury\(^2\) debugger has an implementation of (an approximation to) divide and query which uses a modest amount of memory: the tree is incrementally constructed top-down by re-executing parts of the program \([10]\). Where a subtree is not explicitly constructed, some information is retained, so its (approximate) size can be computed, for example. Clause usage information can be maintained in a similar way.

Search for the optimal node according to our objective function is more difficult than for divide and query, but a top-down search using heuristics may often find a node with a near-optimal value (or at least significantly better than divide and query). The Mercury system currently has a “deep profiler” which accumulates information such as procedure and clause usage using a data structure which may be very useful for computing approximate probabilities and navigating down the tree. Instead of a tree, a graph is used; cycles are introduced by recursive calls. For example, a single \texttt{merge/3} sub-computation would introduce a cycle and use constant space though the two textual calls to \texttt{merge/3} from \texttt{msort_n/4} would accumulate information separately. Alternatively, a non-top-down search could be used, constructing some nodes without constructing their ancestors. For example, constructing a node corresponding to a bottom-most occurrence of each clause would be very helpful for diagnosing consistent bugs.

There may also be other applications of our approach. One possibility\(^3\) is to use the algorithm for generating or simplifying test cases to help validate programs. Given a complex test case where the correctness of the result is very hard to determine, our algorithm can be run to produce a sequence of probes.

---

\(^2\)http://www.cs.mu.oz.au/mercury

\(^3\)Suggested by Ralph Beckett
If any are incorrect a bug can be diagnosed, otherwise each one helps validate the program. The probes have (relatively) low cost and automatically produce good coverage of the program (relative to the complex test case). The algorithm estimates probabilities of nodes being erroneous given that the top level test case is erroneous, and since the probed nodes never have small probabilities, each correct probe contributes to our confidence that the top level test is correct. Depending on the relative importance of validation and diagnosis when a bug symptom is found, the objective function could be adjusted.

9 Conclusion

People draw on many sources of information when debugging; prior suspicions of where the bug may be, correct computations, incorrect sub-terms and dependencies, et cetera. Ideally, automated systems should do the same — debugging is a difficult task and to make it easier, all potentially useful information should be considered. However, there must be some way of combining information from disparate sources and making it available in a form which can be used directly in a debugging algorithm, something which (to our knowledge) has not previously been done. In this paper we have suggested using probability theory to ultimately compute estimates of the likelihood of subcomputations being buggy. This, combined with knowledge of the sizes of the subcomputations and estimates of the cost of probing different nodes is used to determine the next question to ask in the search algorithm. The effect of the information is to reduce the expected number and complexity of questions asked in order to find a bug.

Our approach has been guided by rationality and results of experimentation. Experimentation has revealed that explicit estimation of how consistent bugs are is important, something which (to our knowledge) has not been done previously. Interestingly, other search algorithms can also be considered rational if we assume limited information is available; the algorithms which perform better use more information. Unfortunately, there is actually too much potentially useful information in the execution in most practical cases. Some compromises must be introduced in order to limit memory use especially. Precisely which compromises are made is best determined by detailed implementation considerations which we have not addressed. We have built very simple prototypes of some of the ideas presented here, but are keen to see a more substantial implementation.

References


