An Optimal Strategy for Algorithmic Debugging

David Insa and Josep Silva
Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
E-46022 Valencia, Spain.
{dinsa,jsilva}@dsic.upv.es

***Abstract*—Algorithmic debugging is a technique that uses an internal data structure to represent computations and ask about their correctness. The strategy used to explore this data structure is essential for the performance of the technique. The most efficient strategy in practice is Divide and Query that, until now, has been considered optimal in the worst case. In this paper we first show that the original algorithm is inaccurate and moreover, in some situations it is unable to find all possible solutions, thus it is incomplete. Then, we present a new version of the algorithm that solves these problems. Moreover, we introduce a counterexample showing that Divide and Query is not optimal, and we propose the first optimal strategy for algorithmic debugging with respect to the number of questions asked by the debugger.**

***Index Terms*—Algorithmic Debugging, Divide and Query**

I. Introduction

Algorithmic debugging [15], [18] is based on the answers of the programmer to a series of questions generated automatically by the algorithmic debugger. In each question the debugger provides the programmer with the input and output of a (sub)computation (e.g., a function activation) and the programmer states their correctness. This information is used by the debugger to discard correct parts of the code and guide the search for the bug until a buggy portion of code is isolated.

Example 1.1: Consider the following simple (and buggy) Haskell program:

```
main = mix [1, 3] [2, 4]

insert e []      = [e]
insert e (x:xs)
   | e <= x      = e : x : xs
   | otherwise   = x : insert e xs

mix [] ys        = ys
mix (x:xs) ys    = insert x (mix xs xs)
```

An algorithmic debugging session for this program is the following (`YES` and `NO` answers are provided by the programmer):

```
Starting Debugging Session...
(1)  mix [3] [3] = [3]? NO
(2)  mix [] [] = []? YES
(3)  insert 3 [] = [3]? YES

Bug found in rule:
mix (x:xs) ys    = insert x (mix xs xs)
```

The debugger points out the part of the code that contains the bug. In this case `mix xs xs` should be `mix xs ys`. Note that, to debug the program, the programmer only has to answer questions. It is not even necessary to see the code.

Algorithmic debuggers often produce a data structure called the *execution tree* (ET) [13] that represents a program execution. For instance, the ET of the program in Example 1.1 is depicted in Figure I.

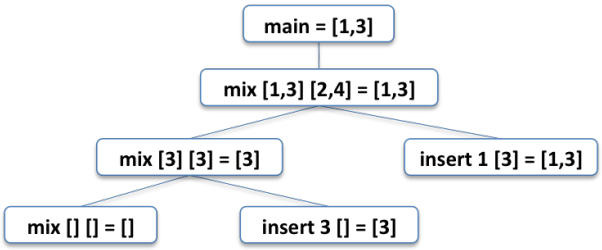


Fig. 1. ET of the program in Example 1.1

The nodes of the ET contain questions, and the strategy used to decide what nodes of the ET should be asked is crucial for the performance of the technique [16].

The algorithmic debugging strategy that presents the best performance in practice is Divide and Query (D&Q) [15], [7]. In fact, from a theoretical point of view, this strategy has been thought optimal in the worst case for almost 30 years, and it has been implemented in almost all current algorithmic debuggers (see, e.g., [5], [6], [8], [14]). In this paper we show that current algorithms for D&Q are suboptimal. We show the problems of D&Q and solve them in a new improved algorithm that is proven optimal. Moreover, the original strategy was only defined for ETs where all the nodes have an individual weight of 1. In contrast, we allow our algorithms to work with different individual weights that can be integer, but also decimal. An individual weight of zero means that this node cannot contain the bug. A positive individual weight approximates the probability of being buggy. The higher the individual weight, the higher the probability. This generalization strongly influences the technique and allows us to assign different probabilities of being buggy to different parts of the program. For instance, a recursive function with higher-order calls should be assigned a higher individual weight than a function implementing a simple base case [16]. The weight of the nodes can also be reassigned dynamically during the debugging session in order to take into account the oracle's answers [6].

We show that the original algorithms are inefficient with ETs where nodes can have different individual weights in the domain of the positive real numbers (including zero) and we redefine the technique for these generalized ETs.

The rest of the paper has been organized as follows. In Section II we recall the algorithmic debugging technique and formalize the strategy D&Q. Then, we show with counterexamples that D&Q is suboptimal and incomplete. In Section III we introduce an improved version of D&Q. In Section IV we define a new strategy for algorithmic debugging that is optimal in all cases. The correctness of the algorithms presented is proven in Section V. Finally, Section VI concludes.

II. Algorithmic Debugging

In this section we recall the algorithmic debugging technique and formalize the strategy D&Q [15]. We start with the definition of *marked execution tree*, that is an ET where some nodes could have been removed because they were marked as correct (i.e., answered YES), some nodes could have been marked as wrong (i.e., answered NO) and the correctness of the other nodes is undefined.

Definition 2.1 (Marked Execution Tree): A *marked execution tree* (MET) is a tree $T = (N, E, M)$ where N are the nodes, $E \subseteq N \times N$ are the edges, and $M : N \to V$ is a marking total function that assigns to all the nodes in N a value in the domain $V = \{Wrong, Undefined\}$.

Initially, all nodes in the MET are marked as $Undefined$. But with every answer of the user, a new MET is produced. Concretely, given a MET $T = (N, E, M)$ and a node $n \in N$, the answer of the user to the question in n produces a new MET such that: (i) if the answer is YES, then this node and its subtree is removed from the MET. (ii) If the answer is NO, then, all the nodes in the MET are removed except this node and its descendants.[1] Therefore, note that the only node that can be marked as $Wrong$ is the root. Moreover, the rest of nodes can only be marked as $Undefined$ because when the answer is YES, the associated subtree is deleted from the MET.

Therefore, the size of the MET is gradually reduced with the answers. If we delete all nodes in the MET then the debugger concludes that no bug has been found. If, contrarily, we finish with a MET composed of a single node marked as wrong, this node is called the *buggy node* and it is pointed to as being responsible for the bug of the program.

All this process is defined in Algorithm 1 where function *selectNode* selects a node in the MET to be asked to the user with function *askNode*. Therefore, *selectNode* is the central point of this paper. In the rest of this section, we assume that *selectNode* implements D&Q. In the following we use E^* to refer to the reflexive and transitive closure of E and E^+ for the transitive closure.

[1]It is also possible to accept *I don't know* as an answer of the user. In this case, the debugger simply selects another node [8]. For simplicity, we assume here that the user only answers YES or NO.

Algorithm 1 General algorithm for algorithmic debugging

Input: A MET $T = (N, E, M)$
Output: A buggy node or $\perp$ if no buggy node is detected
Preconditions: $\forall n \in N,\ M(n) = Undefined$
Initialization: buggyNode $= \perp$

begin
(1) **do**
(2) node = selectNode(T)
(3) answer = askNode(node)
(4) **if** (answer = NO)
(5) **then** M(node) = $Wrong$
(6) buggyNode = node
(7) $N = \{n \in N \mid (\text{node} \to n) \in E^*\}$
(8) **else** $N = N \backslash \{n \in N \mid (\text{node} \to n) \in E^*\}$
(9) **while** $(\exists n \in N, M(n) = Undefined)$
(10) **return** buggyNode
end

A. Divide and Query

D&Q assumes that the individual weight of a node is always 1. Therefore, given a MET $T = (N, E, M)$, the weight of the subtree rooted at node $n \in N$, w_n, is defined recursively as its number of descendants including itself (i.e., $1 + \sum \{w_{n'} \mid (n \to n') \in E\}$).

D&Q tries to simulate a dichotomic search by selecting the node that better divides the MET into two subMETs with a weight as similar as possible. Therefore, given a MET with n nodes, D&Q searches for the node whose weight is closer to $\frac{n}{2}$ [7].

B. Limitations of D&Q

In this section we show that D&Q is suboptimal when the MET does not contain a wrong node (i.e., all nodes are marked as undefined).[2] The intuition beyond this limitation is that the objective of D&Q is to divide the tree by two, but the real objective should be to reduce the number of questions to be asked to the programmer. For instance, consider the MET in Figure 2 (left) where each node is labeled with its weight and the black node is marked as wrong, thus D&Q would select the gray node. The objective of D&Q is to divide the 8 nodes into two groups of 4. Nevertheless, the real motivation of dividing the tree should be to divide the tree into two parts that would produce the same number of remaining questions (in this case 3).

The problem comes from the fact that D&Q does not take into account the marking of wrong nodes. For instance, observe the two METs in Figure 2 (center) where the black node is marked as wrong. In both cases D&Q would behave exactly in the same way, because it completely ignores the marking of the root. Nevertheless, it is evident that we do not need to ask again for a node that is already marked as wrong to determine whether it is buggy. However, D&Q counts the

[2]Modern debuggers [8] allow the programmer to debug the MET while it is being generated. Thus the root node of the subtree being debugged is not necessarily marked as *Wrong*.

nodes marked as wrong as part of their own weight, and this is a source of inefficiency.

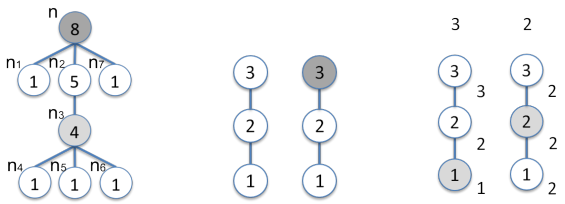


Fig. 2. Behavior of Divide and Query

In the METs of Figure 2 (center) we have two METs. In the one at the right nodes with weight 1 and 2 are optimal, but in the one at the left, only the node with weight 2 is optimal. In both METs D&Q would select either the node with weight 1 or the node with weight 2 (both are equally close to $\frac{3}{2}$). However, we show in Figure 2 (right) that selecting node 1 is suboptimal, and the strategy should always select node 2. Considering that the gray node is the first node selected by the strategy, then the number at the side of a node represents the number of questions needed to find the bug if the buggy node is this node. The number at the top of the figure represents the number of questions needed to determine that there is not a bug. Clearly, as an average, it is better to select first the node with weight 2 because we would perform less questions ($\frac{8}{4}$ vs. $\frac{9}{4}$ considering all four possible cases).

Therefore, D&Q returns a set of nodes that contains the best node, but it is not able to determine which of them is the best node, thus being suboptimal when it is not selected. In addition, the METs in Figure 3 show that D&Q is incomplete. Observe that the METs have 4 nodes, thus D&Q would always select the node with weight 2. However, the node with weight 3 is equally optimal (both need $\frac{12}{5}$ questions as an average to find the bug) but it will be never selected by D&Q because its weight is farther from the half of the tree $\frac{4}{2}$.

Another limitation of D&Q is that it was designed to work with METs where all the nodes have the same individual weight, and moreover, this weight is assumed to be 1. If we work with METs where nodes can have different individual weights and these weights can be any value greater or equal to zero, then D&Q is suboptimal as it is demonstrated by the MET in Figure 4. In this MET, D&Q would select node n_1 because its weight is closer to $\frac{21}{2}$ than any other node.

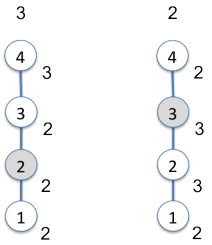


Fig. 3. Incompleteness of Divide and Query

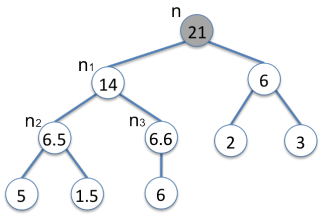


Fig. 4. MET with decimal individual weights

However, node n_2 is the node that better divides the tree in two parts with similar probabilities of containing the bug.

In summary, (1) D&Q is suboptimal when the MET is free of wrong nodes, (2) D&Q is correct when the MET contains wrong nodes and all the nodes of the MET have the same weight, but (3) D&Q is suboptimal when the MET contains wrong nodes and the nodes of the MET have different individual weights.

III. Optimal D&Q

In this section we introduce a new version of D&Q that divides the MET into two parts with the same probability of containing the bug (instead of two parts with the same weight). We introduce a new algorithm that is correct and complete even if the MET contains nodes with different individual weights that can be any decimal number greater than 0. For this, we define the *search area* of a MET as the set of undefined nodes.

Definition 3.1 (Search area): Let $T = (N, E, M)$ be a MET. The *search area* of T, $Sea(T)$, is defined as $\{n \in N \mid M(n) = Undefined\}$.

While D&Q uses the whole T, we only use $Sea(T)$, because answering all nodes in $Sea(T)$ guarantees that we can discover all buggy nodes [9]. Moreover, in the following we refer to the individual weight of a node n with wi_n; and we refer to the weight of a (sub)tree rooted at n with w_n that is recursively defined as:

$$w_n = \begin{cases} \sum \{w_{n'} \mid (n \to n') \in E\} & \text{if } M(n) \neq Undefined \\ wi_n + \sum \{w_{n'} \mid (n \to n') \in E\} & \text{otherwise} \end{cases}$$

Note that, contrarily to standard D&Q, the definition of w_n excludes those nodes that are not in the search area (i.e., the root node when it is wrong). Note also that wi_n allows us to assign any individual weight to the nodes. This is a generalization of D&Q where it is assumed that all nodes have the same individual weight and it is always 1.

For the sake of clarity, given a node $n \in Sea(T)$, we distinguish between three subareas of $Sea(T)$ induced by n: (1) n itself, whose individual weight is wi_n; (2) descendants of n ($Down(n)$), whose weight is

$d_n = \sum \{wi_{n'} \mid n' \in Sea(T) \wedge (n \to n') \in E^+\}$

and (3) the rest of nodes ($Up(n)$), whose weight is

$u_n = \sum \{wi_{n'} \mid n' \in Sea(T) \wedge (n \to n') \notin E^*\}$

Example 3.2: Consider the MET in Figure 5. Assuming that the root n is marked as wrong and all nodes have an individual weight of 1, then $Sea(T)$ contains all nodes except n, $u_{n'} = 3$

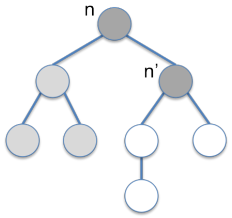


Fig. 5. Functions Up and Down

(total weight of the gray nodes), and $d_{n'} = 3$ (total weight of the white nodes).

Clearly, for any MET whose root is n and a node n', $M(n') = \mathit{Undefined}$, we have that:

$$w_n = u_{n'} + d_{n'} + wi_{n'} \qquad \text{(Equation 1)}$$
$$w_{n'} = d_{n'} + wi_{n'} \qquad \text{(Equation 2)}$$

Intuitively, given a node n, what we want to divide by half is the area formed by $u_n + d_n$. That is, n will not be part of $\mathit{Sea}(T)$ after it has been answered, thus the objective is to make u_n equal to d_n. This is another important difference with traditional D&Q: wi_n should not be considered when dividing the MET. We use the notation $n_1 \gg n_2$ to express that n_1 divides $\mathit{Sea}(T)$ better than n_2 (i.e., $|d_{n_1} - u_{n_1}| < |d_{n_2} - u_{n_2}|$). And we use $n_1 \equiv n_2$ to express that n_1 and n_2 equally divide $\mathit{Sea}(T)$. If we find a node n such that $u_n = d_n$ then n produces an optimal division, and should be selected by the strategy.

In the rest of this section we present an algorithm for optimal D&Q (i.e., that optimally divides the MET by half). In particular, given a MET, Algorithm 2 efficiently determines the best node to divide $\mathit{Sea}(T)$ by half (in the following the *optimal node*). In order to find this node, the algorithm does not need to compare all nodes in the MET. It follows a path of nodes from the root to the optimal node which is closer to the root producing a minimum set of comparisons.

Algorithm 2 Optimal D&Q —SelectNode in Algorithm 1—

Input: A MET $T = (N, E, M)$ whose root is $n \in N$ and $\forall n' \in N, wi_{n'} \geq 0$
Output: A node $n_{Optimal} \in N$
Preconditions: $\exists n' \in N,\ M(n') = \mathit{Undefined}$

begin
(1) Candidate = n
(2) **do**
(3) Best = Candidate
(4) Children = $\{m \mid (\text{Best} \rightarrow m) \in E\}$
(5) **if** (Children = $\emptyset$) **then return** Best
(6) Candidate = $n' \mid \forall n''$ with $n', n'' \in$ Children, $w_{n'} \geq w_{n''}$
(7) **while** $(w_{Candidate} - \frac{wi_{Candidate}}{2} > \frac{w_n}{2})$
(8) Candidate = $n' \mid \forall n''$ with $n', n'' \in$ Children, $w_{n'} - \frac{wi_{n'}}{2} \geq w_{n''} - \frac{wi_{n''}}{2}$
(9) **if** $(M(\text{Best}) = \mathit{Wrong})$ **then return** Candidate
(10) **if** $(w_n \geq w_{Best} + w_{Candidate} - \frac{wi_{Best}}{2} - \frac{wi_{Candidate}}{2})$ **then return** Best
(11) **else return** Candidate
end

Example 3.3: Consider the MET in Figure 6 where $\forall n \in N, wi_n = 1$ and $M(n) = \mathit{Undefined}$.

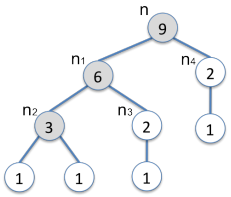


Fig. 6. Defining a path in a MET to find the optimal node

Observe that Algorithm 2 traverses the MET top-down from the root selecting at each level the heaviest node until we find a node whose weight minus its individual weight divided by two is smaller or equal than the half of the MET ($\frac{w_n}{2}$), thus, defining a path in the MET that is colored in gray. Finally, the algorithm selects n_1.

One important idea of the algorithm is the identification of a path from the root to the buggy node. In particular, when we use Algorithm 2 and compare two nodes n_1, n_2 in a MET whose root is n, we find five possible cases (see Figure 7) to define this path:

Case 1: $d_{n_1} > u_{n_1} \ \wedge\ d_{n_2} \leq u_{n_2} \ \wedge\ n_1, n_2$ are brothers
Case 2: $d_{n_1} \leq u_{n_1} \ \wedge\ d_{n_2} \leq u_{n_2} \ \wedge\ n_1, n_2$ are brothers
Case 3: $d_{n_1} > u_{n_1} \ \wedge\ d_{n_2} > u_{n_2} \ \wedge\ (n_1 \rightarrow n_2) \in E$
Case 4: $d_{n_1} > u_{n_1} \ \wedge\ d_{n_2} \leq u_{n_2} \ \wedge\ (n_1 \rightarrow n_2) \in E$
Case 5: $d_{n_1} \leq u_{n_1} \ \wedge\ d_{n_2} \leq u_{n_2} \ \wedge\ (n_1 \rightarrow n_2) \in E$

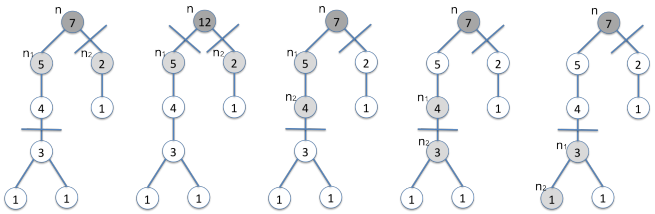


Fig. 7. Determining the best node in a MET (five possible cases)

In cases 1 and 5 n_1 is better (i.e., $n_1 \gg n_2$ or $n_1 \equiv n_2$); in case 3 n_2 is better; and in cases 2 and 4 the best node must be determined comparing them. Observe that these results allow the algorithm to determine the path to the optimal node that is closer to the root. For instance, in Example 3.3 case 1 is used to select a child, e.g., node n_1 instead of node n_4. Case 3 is used to go down and select node n_1 instead of node n. Case 2 is used to select node n_2 instead of node n_3 (this is done with line (8) of the algorithm). Case 5 is used to stop going down at node n_2 because it is better than all its descendants. And it is also used to determine that nodes n_3 and n_4 are better than all their descendants. Finally, case 4 is used to select the optimal node, n_1 instead of n_2 (this is done with line (10) of the algorithm). Note that D&Q could have selected node n_2 that is equally close to $\frac{9}{2}$ than node n_1; but it is suboptimal because $u_{n_1} = 3$ and $d_{n_1} = 5$ whereas $u_{n_2} = 6$ and $d_{n_2} = 2$.

The correctness of Algorithm 2 is stated by the following theorem.

Theorem 3.4 (Correctness): Let $T = (N, E, M)$ be a MET where $\forall n \in N, wi_n \geq 0$, then the execution of Algorithm 2 with T as input always terminates producing as output a node $n \in Sea(T)$ such that $\nexists n' \in Sea(T) \mid n' \gg n$.

Algorithm 2 always returns a single optimal node. However, a small modification can make the algorithm to return all optimal nodes, thus being complete: changing line (8) to collect all candidates instead of one, and updating accordingly the output lines to return a set. Moreover, line (10) should be split to distinguish between the cases '$>$' (returning $\{Best\}$) and '$=$' (returning $candidates \cup \{Best\}$).

IV. Divide by Queries

An strategy should be consider optimal with respect to the number of questions generated if and only if the average number of questions asked with any MET is minimum. Note that we can compute this average by assuming that the bug can be in any node of the MET, and thus computing the number of questions asked for each node using Algorithm 1. In this section we call *optimal node* to the first node asked by an optimal strategy (instead of the node that better divides the MET by the half).

Definition 4.1 (Optimal strategy): Let ϵ be an algorithmic debugging strategy. Given a MET $T = (N, E, M)$, let s^{ϵ}_{n} the sequence of questions made by Algorithm 1 with strategy ϵ and considering that the only buggy node in T is $n \in N$. Let $t_{\epsilon} = \sum_{n_i \in N} |s^{\epsilon}_{n_i}|$. We say that ϵ is optimal if and only if for any MET $\nexists \epsilon' \; . \; t_{\epsilon} > t_{\epsilon'}$.

In this section we show that any version of D&Q is and will be suboptimal. The reason is that D&Q tries to prune the biggest possible subtree, but it ignores the structure of the MET. In practice, pruning complex subtrees that are more difficult to explore is very important, but this is ignored by D&Q. This means that our version of D&Q (Algorithm 2) is optimal in the sense that it optimally divides the MET by the half. But it is not an optimal strategy because in total (considering all questions needed to find the bug) it can perform more questions than necessary.

Because we compute the cost of a strategy based on the number of questions asked, we need a formal definition for sequence of questions.

Definition 4.2 (Sequence of questions): Given a MET $T = (N, E, M)$ and two nodes $n_1, n_2 \in N$, a *sequence of questions* of n_1 with respect to n_2, $sq(n_1, n_2)$, is formed by all nodes asked by Algorithm 1 when the first node selected by function $selectNode(T)$ is n_2 and the only buggy node in T is n_1.

This means that the sequence of questions is completely dependent on the used strategy. For instance, using standard D&Q with the MET in Figure 2 (left) and assuming that all nodes are marked as undefined:

$sq(n, n_3) = [n_3, n_1, n_2, n_7, n]$ $\quad$ $sq(n_3, n) = [n, n_3, n_4, n_5, n_6]$

$sq(n, n) = [n, n_3, n_1, n_2, n_7]$ $\quad$ $sq(n_3, n_3) = [n_3, n_4, n_5, n_6]$

We now show a counterexample where D&Q cannot find an optimal node.

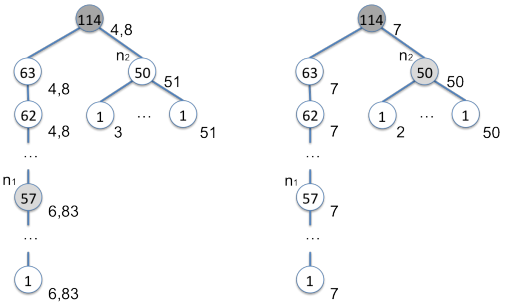


Fig. 8. MET where D&Q cannot find an optimal node

Example 4.3: Consider the MET in Figure 8 (left) where the number at the right of the node represents $|sq(n, n')|$ being n the node itself and n' the gray node. D&Q would select node n_1 because its weight is closer to $\frac{114}{2} = 57$. However, n_2 is the only optimal node, and it produces less questions than n_1 even though its weight is far from 57.

In the MET at the left we start asking node n_1. If the bug is located in the subtree of node n_1, because it is a deep subtree, we would ask an average of $\log_2 57 = 5,83$ questions to each node ($+1$ because we have initially asked node n_1). If the bug were not in this subtree, then after asking node n_1 we would explore the subtree of node n_2. If the bug is located in this subtree we must ask all nodes until we find the bug, and all these nodes have to consider the question already asked to node n_1; in total we ask $(\sum_{i=3}^{51} i) + 51 = 1374$ questions. If the bug were not located in the right brach, there only remain the 7 top nodes in the left branch (including the root). There we ask an average of $\log_2 7 = 2,8$ questions to each node ($+2$ because we already asked nodes n_1 and n_2). In total we ask $57 * 6,83 + 1374 + 7 * 4,8 = 1796,91$ questions, and an average of $\frac{1796,91}{114} = 15,76$ questions.

If, contrarily, we start asking node n_2 (see the MET at the right) and the bug is located in this subtree, we ask $(\sum_{i=2}^{50} i) + 50 = 1324$ questions. If the bug is located in the other branch, after asking node n_2 we still have 64 nodes in depth; therefore, with $\log_2 64 = 6$ questions ($+1$ because we have initially asked node n_2) we will find the bug. In total, we ask $1324 + 64 * 7 = 1772$ questions, and an average of $\frac{1772}{114} = 15,54$ questions.

Example 4.3 showed that D&Q is not an optimal strategy. The question now is whether an optimal strategy exists: is the problem decidable?

Theorem 4.4 (Decidability): Given a MET, finding all optimal nodes is a decidable problem.

Proof: We show that at least one finite method exists to find all optimal nodes. Firstly, we know that the size of the MET is finite because the question in the root can only be completed if the execution terminated, and hence the number of subcomputations—and nodes—is finite [8]. Because the tree is finite, we know (according to Algorithm 1) that any sequence of questions asked by the debugger (no matter the strategy used) is also finite because at most all nodes of the tree are asked once. Therefore, the number of possible

sequences is also finite. This guarantees that we can compute all possible sequences and select the best sequences according to the equation in Definition 4.1. The optimal nodes will be the first of the selected sequences. ■

Even though the method described in the proof of Theorem 4.4 is effective, it is too expensive because it needs to compute all possible sequences of questions. In the rest of this section we present a more efficient method to compute all optimal nodes.

We now present a method to select the best sequences of questions (sq_n) for the nodes in a MET. For the sake of clarity, in the following when we talk about the sequence of questions of a node, we assume that this node is wrong and that the sequence contains a set of nodes that after they have been asked, they allow us to know whether the node is buggy or not.

In order to formalize the method described in the proof of Theorem 4.4 we first define the notion of valid sequence of questions.

Definition 4.5 (Valid sequences of questions of a node): Let $T = (N, E, M)$ be a MET whose root is $n \in N$. A sequence of questions $sq_n = [n_1, \ldots, n_m]$ for n is valid if:

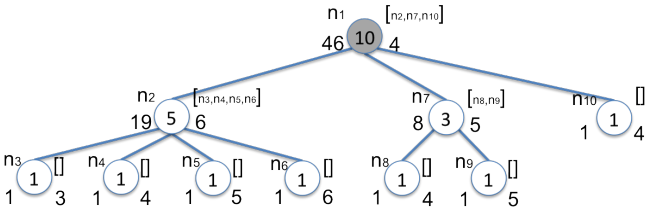
1) $\forall n_i, n_j \in sq_n, 1 \leq i < j \leq m, (n_i \rightarrow n_j) \notin E^*$
2) $N \backslash \{n_j \mid (n_i \rightarrow n_j) \in E^* \wedge n_i \in sq_n\} = \{n\}$

We denote with SQ_n the set of valid sequences of questions of n.

Intuitively, the valid sequences of questions of a root node n are all those sequences formed with non-repeated nodes that (1) a node in the sequence cannot be descendant of a previous node in the sequence, and (2) after having pruned all the subtrees whose roots are the nodes of the sequence, node n has not descendants.

The next example shows that if we label each node of a MET with a valid sequence of questions, then it is possible to know how many questions do we need to ask to find the buggy node.

Example 4.6: Consider the next tree:



Here, nodes are labeled with their weight (inside) their identifier (top left), an optimal sequence of questions for this node (top right), $|sq(n, n_1)|$ where n is the node (bottom right), and the sum of $|sq(n', n)|$ for all node n' of the subtree of this node (n) (bottom left).

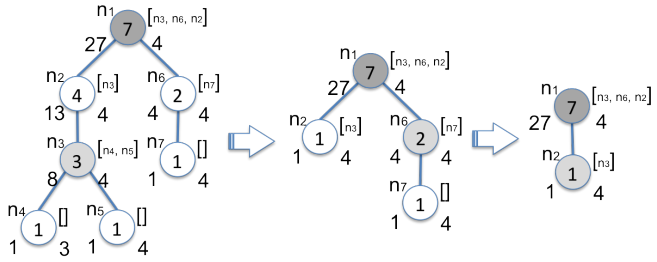
There exist many valid sequences for node n_1, (e.g. $[n_2, n_7, n_{10}]$, $[n_2, n_{10}, n_7]$, $[n_3, n_2, n_7, n_{10}]$, etc.). If we consider the sequence of the figure ($[n_2, n_7, n_{10}]$) and taking into account that we start asking in the root node[3], then we can easily determine the number of questions needed to find the bug in any node n. We refer to this number with q_n. For instance, we need 4 questions to find the bug in node n_1 ($[n_1, n_2, n_7, n_{10}]$). Similarly, $q_{n_4} = 4$ ($[n_1, n_2, n_3, n_4]$) and $q_{n_7} = 5$ ($[n_1, n_2, n_7, n_8, n_9]$). Observe that when we reach node n_7 and mark it as wrong we continue the sequence of questions of this node ($[n_8, n_9]$).

Once we have computed q for all descendants of node n, we can also compute the n's number at bottom left (referred to as Q_n) by adding all qs. In the figure, we see that $Q_{n_1} = 46$, $Q_{n_2} = 19$ and (trivially) the leafs only need one question (the node itself).

Therefore, to find the optimal nodes, we only have to: (i) Compute Q_n for all nodes in the MET, (ii) add to the MET a fictitious root node, (iii) compute all valid sequences of the root node, (iv) compute the cost associated to each sequence (with Algorithm 3), and (v) select the first node of the sequences with the minimum cost.

Essentially, Algorithm 3 is used to compute Q_n of a given node. For this, it compositionally computes the number of questions that should be asked to each node. This is done by taking into account the individual weight of each node that, as with Algorithm 2, can be any number greater than or equal to 0.

Example 4.7: Consider the following tree at the left with depth 4, where we want to compute the cost Q_{n_1} associated to the sequence $[n_3, n_6, n_2]$.



Function $ComputeQ$ takes the first element of the sequence (n_3) and computes the number of questions to be done if the bug is located in its subtree. This is $Q_{n_3} = 8$. Therefore, no matter where the bug is, we have to ask one extra question for each node (the one in the root n_1). Thus we have a total of 8+3=11 questions. If the bug is not located in the subtree of n_3, then we should continue asking questions of the sequence having pruned this subtree. Therefore, we should prune this tree and recalculate Q for the ancestors of node n_3. This is done with function $AdjustIntermediateNodes$ producing the tree with a depth of 2 in the middle of the figure above. In this new tree, Q_{n_2} has been recomputed and its value is 1.

Then, we proceed with the next question in the sequence (n_6). Now we have Q_{n_6}=4 questions. But now we have to take into account two extra questions (one for n_1 and one

[3]Note that this does not mean that valid sequences must necessarily start asking the root node. Given any MET we can add a new fictitious parent of the root and compute the optimal sequence of questions associated to this new node.

for n_3) for each node in the subtree of n_6: 2*2=4 questions. Hence we have a total of (8+3)+(4+4)=19 questions. If the bug is not located in the subtree of node n_6 we prune it producing the tree at the right of the figure and we ask the next question in the sequence: n_2. If the bug is n_2, then we have to ask one question (n_2) plus the extra questions done before (n_1, n_3 and n_6). Thus we have a total of (8+3)+(4+4)+(1+3)=23 questions. Finally, if the bug is located in the root, we have to ask 4 questions: the root node itself and all questions in the sequence. Thus, the final value of Q_{n_1} is (8+3)+(4+4)+(1+3)+(4)=27.

The previous example illustrates the work done by Algorithm 3 to compute Q. It basically computes and sums the number of questions asked for each node. For this, it has to take into account the sequence of questions in order to decide how many questions are cumulated when a new subtree is explored.

Algorithm 3 uses function $computeSQ(T, n)$ to compute all possible valid sequences of questions associated to node n. Therefore, because this function returns all possible sequences for the node, then the strategy is optimal. However, note that this function could be restricted to behave as other strategies of the literature. For instance, we could adapt it to work as the strategy *Top-Down* [1] if we restrict the sequences returned to those where all elements in the sequence are children of n. This is equivalent to adding the following restriction to Definition 4.5:

3) $\forall n' \in sq_n, (n \rightarrow n') \in E$

V. Proofs of Technical Results

We first define a MET where individual weights can be decimal numbers:

Definition 5.1 (Variable MET): A *variable MET* $T = (N, E, M)$ is a MET, where $\forall n \in N, wi_n \geq 0$.

Theorem 3.4 states the correctness of Algorithm 2. For the proof of this theorem we define first some auxiliary lemmas. The following lemma ensures that $w_{n_1} - \frac{wi_{n_1}}{2} > \frac{w_n}{2}$ used in the condition of the loop implies $d_{n_1} > u_{n_1}$.

Lemma 5.2: Given a variable MET $T = (N, E, M)$ whose root is $n \in N$ and a node $n_1 \in Sea(T)$, $d_{n_1} > u_{n_1}$ if and only if $w_{n_1} - \frac{wi_{n_1}}{2} > \frac{w_n}{2}$.

Proof: We prove that $w_{n_1} - \frac{wi_{n_1}}{2} > \frac{w_n}{2}$ implies $d_{n_1} > u_{n_1}$ and vice versa.

$w_{n_1} - \frac{wi_{n_1}}{2} > \frac{w_n}{2}$
$2w_{n_1} - wi_{n_1} > w_n$
We replace w_{n_1} using Equation 2:
$2(d_{n_1} + wi_{n_1}) - wi_{n_1} > w_n$
$2d_{n_1} + wi_{n_1} > w_n$
$d_{n_1} > w_n - d_{n_1} - wi_{n_1}$
We replace $w_n - d_{n_1} - wi_{n_1}$ using Equation 1:
$d_{n_1} > u_{n_1}$

■

The following lemma ensures that given two nodes n_1 and n_2 where $d_n \geq u_n$ in both nodes and $n_1 \rightarrow n_2$ then $n_2 \gg n_1 \vee n_2 \equiv n_1$ (**Case 3**).

Algorithm 3 Compute Q_n

Input: A MET $T = (N, E, M)$ and a node $n \in N$
Output: Q_n

(1) $(sq_n, Q_n) = ComputeOptimalSequence(T, n)$
(2) **return** Q_n

function $ComputeOptimalSequence(T, n)$
begin
(3) $SQ_n = computeSQ(T, n)$
(4) $sqOptimal = sq_n \in SQ_n \mid \nexists sq'_n \in SQ_n \; . \; ComputeQ(T, n, sq_n) > ComputeQ(T, n, sq'_n)$
(5) $QOptimal = ComputeQ(T, n, sqOptimal)$
(6) **return** $(sqOptimal, QOptimal)$
end

function $ComputeQ(T, n, sq_n)$
begin
(7) $questions = 0$
(8) $indexNode = 0$
(9) $accumNodes = 1$
(10) **while** $(\{n' | (n \rightarrow n') \in E^*\} \neq \{n\})$
(11) $\quad node = sq_n[indexNode]$
(12) $\quad indexNode = indexNode + 1$
(13) $\quad questions = questions + (Q_{node} + accumNodes * w_{node})$
(14) $\quad accumNodes = accumNodes + 1$
(15) $\quad T = AdjustIntermediateNodes(T, n, node)$
end while
(16) $questions = questions + (accumNodes * wi_n)$
(17) **return** $questions$
end

function $AdjustIntermediateNodes(T, n, n')$
begin
(18) $O = \{n'' \in N \mid (n' \rightarrow n'') \in E^*\}$
(19) $N = N \backslash O$
(20) $n' = n'' \mid (n'' \rightarrow n') \in E$
(21) **while** $(n' \neq n)$
(22) $\quad (\_, Q_{n'}) = ComputeOptimalSequence(T, n')$
(23) $\quad w_{n'} = w_{n'} - |O|$
(24) $\quad n' = n'' \mid (n'' \rightarrow n') \in E$
end while
(25) **return** T
end

Lemma 5.3: Given a variable MET $T = (N, E, M)$ and given two nodes $n_1, n_2 \in Sea(T)$, with $(n_1 \rightarrow n_2) \in E$, if $d_{n_2} \geq u_{n_2}$ then $n_2 \gg n_1 \vee n_2 \equiv n_1$.

Proof: We prove that $|d_{n_2} - u_{n_2}| \leq |d_{n_1} - u_{n_1}|$ holds. First, we know that $d_{n_1} = d_{n_2} + wi_{n_2} + inc$ and $u_{n_1} = u_{n_2} - wi_{n_1} - inc$ with $inc \geq 0$, where inc represents the weight of the possible brothers of n_2.

$|d_{n_2} - u_{n_2}| \leq |d_{n_1} - u_{n_1}|$
As we know that $d_n \geq u_n$ in both nodes:
$d_{n_2} - u_{n_2} \leq d_{n_1} - u_{n_1}$
We replace d_{n_1} and u_{n_1}:
$d_{n_2} - u_{n_2} \leq (d_{n_2} + wi_{n_2} + inc) - (u_{n_2} - wi_{n_1} - inc)$
$d_{n_2} - u_{n_2} \leq d_{n_2} - u_{n_2} + wi_{n_1} + wi_{n_2} + 2inc$
$0 \leq wi_{n_1} + wi_{n_2} + 2inc$

Hence, because wi_{n_1}, wi_{n_2}, $inc \geq 0$ then $|d_{n_2} - u_{n_2}| \leq |d_{n_1} - u_{n_1}|$ is satisfied and thus $n_2 \gg n_1 \vee n_2 \equiv n_1$. ■

The following lemma ensures that given two nodes n_1 and n_2 where $d_n \leq u_n$ in both nodes and $n_1 \rightarrow n_2$ then $n_1 \gg n_2 \vee n_1 \equiv n_2$ (**Case 5**).

Lemma 5.4: Given a variable MET $T = (N, E, M)$ and given two nodes $n_1, n_2 \in Sea(T)$, with $(n_1 \rightarrow n_2) \in E$, if $d_{n_1} \leq u_{n_1}$ then $n_1 \gg n_2 \vee n_1 \equiv n_2$.

Proof: We prove that $|d_{n_1} - u_{n_1}| \leq |d_{n_2} - u_{n_2}|$ holds. First, we know that $d_{n_2} = d_{n_1} - wi_{n_2} - inc$ and $u_{n_2} = u_{n_1} + wi_{n_1} + inc$ with $inc \geq 0$, where inc represents the weight of the possible brothers of n_2.

$|d_{n_1} - u_{n_1}| \leq |d_{n_2} - u_{n_2}|$
As we know that $u_n \geq d_n$ in both nodes:
$u_{n_1} - d_{n_1} \leq u_{n_2} - d_{n_2}$
We replace d_{n_2} and u_{n_2}:
$u_{n_1} - d_{n_1} \leq (u_{n_1} + wi_{n_1} + inc) - (d_{n_1} - wi_{n_2} - inc)$
$u_{n_1} - d_{n_1} \leq u_{n_1} - d_{n_1} + wi_{n_1} + wi_{n_2} + 2inc$
$0 \leq wi_{n_1} + wi_{n_2} + 2inc$

Hence, because wi_{n_1}, wi_{n_2}, $inc \geq 0$ then $|d_{n_1} - u_{n_1}| \leq |d_{n_2} - u_{n_2}|$ is satisfied and thus $n_1 \gg n_2 \vee n_1 \equiv n_2$. ∎

The following lemma ensures that given two brother nodes n_1 and n_2, if $d_{n_1} \geq u_{n_1}$ then $d_{n_2} \leq u_{n_2}$.

Lemma 5.5: Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given three nodes $n_1 \in N$ and $n_2, n_3 \in Sea(T)$, with $(n \to n_1) \in E^*$, $(n_1 \to n_2), (n_1 \to n_3) \in E$, if $d_{n_2} \geq u_{n_2}$ then $d_{n_3} \leq u_{n_3}$.

Proof: We prove it by contradiction assuming that $d_{n_3} > u_{n_3}$ when $d_{n_2} \geq u_{n_2}$ and they are brothers. First, we know that as n_2 and n_3 are brothers then $u_{n_2} \geq w_{n_3}$ and $u_{n_3} \geq w_{n_2}$. Therefore, if $d_{n_3} > u_{n_3}$ then $d_{n_2} \geq u_{n_2} \geq w_{n_3} \geq d_{n_3} > u_{n_3} \geq w_{n_2} \geq d_{n_2}$ that implies $d_{n_2} > d_{n_2}$ that is a contradiction itself. ∎

If two nodes n_1 and n_2 are brothers and $d_{n_1} \geq u_{n_1}$ then $n_1 \gg n_2 \vee n_1 \equiv n_2$ (**Case 1**). The following lemma proves this property.

Lemma 5.6: Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given three nodes $n_1 \in N$ and $n_2, n_3 \in Sea(T)$, with $(n \to n_1) \in E^*$, $(n_1 \to n_2), (n_1 \to n_3) \in E$, if $d_{n_2} \geq u_{n_2}$ then $n_2 \gg n_3 \vee n_2 \equiv n_3$.

Proof: We prove that $|d_{n_2} - u_{n_2}| \leq |d_{n_3} - u_{n_3}|$ holds. First, as n_2 and n_3 are brothers we know that $w_n \geq d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3}$, then $w_n = d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3} + inc$ with $inc \geq 0$.

$|d_{n_2} - u_{n_2}| \leq |d_{n_3} - u_{n_3}|$
As $d_{n_2} \geq u_{n_2}$ by Lemma 5.5 we know that $u_{n_3} \geq d_{n_3}$:
$d_{n_2} - u_{n_2} \leq u_{n_3} - d_{n_3}$
We replace u_{n_2} and u_{n_3} using Equation 1:
$d_{n_2} - (w_n - d_{n_2} - wi_{n_2}) \leq (w_n - d_{n_3} - wi_{n_3}) - d_{n_3}$
$-w_n + 2d_{n_2} + wi_{n_2} \leq w_n - 2d_{n_3} - wi_{n_3}$
$-2w_n \leq -2d_{n_2} - 2d_{n_3} - wi_{n_2} - wi_{n_3}$
$2w_n \geq 2d_{n_2} + 2d_{n_3} + wi_{n_2} + wi_{n_3}$
$w_n \geq d_{n_2} + d_{n_3} + \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
We replace w_n:
$d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3} + inc \geq d_{n_2} + d_{n_3} + \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
$wi_{n_2} + wi_{n_3} + inc \geq \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
$\frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2} + inc \geq 0$

Hence, because wi_{n_2}, wi_{n_3}, $inc \geq 0$ then $|d_{n_2} - u_{n_2}| \leq |d_{n_3} - u_{n_3}|$ is satisfied and thus $n_2 \gg n_3 \vee n_2 \equiv n_3$. ∎

The following lemma ensures that given two brother nodes n_1 and n_2, if $w_{n_1} \geq w_{n_2}$ and $d_{n_1} \leq u_{n_1}$ then $d_{n_2} \leq u_{n_2}$.

Lemma 5.7: Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given three nodes $n_1 \in N$ and $n_2, n_3 \in Sea(T)$, with $(n \to n_1) \in E^*$, $(n_1 \to n_2), (n_1 \to n_3) \in E$, if $w_{n_2} \geq w_{n_3}$ and $d_{n_2} \leq u_{n_2}$ then $d_{n_3} \leq u_{n_3}$.

Proof: We prove it by contradiction assuming that $d_{n_3} > u_{n_3}$ when $w_{n_2} \geq w_{n_3}$ and $d_{n_2} \leq u_{n_2}$ and they are brothers. First, we know that as n_2 and n_3 are brothers then $u_{n_2} \geq w_{n_3}$ and $u_{n_3} \geq w_{n_2}$. Therefore, if $d_{n_3} > u_{n_3}$ then $d_{n_3} > u_{n_3} \geq w_{n_2} \geq w_{n_3} \geq d_{n_3}$ that implies $d_{n_3} > d_{n_3}$ that is a contradiction itself. ∎

If two nodes n_1 and n_2 are brothers and $u_{n_1} \geq d_{n_1} \wedge u_{n_2} \geq d_{n_2}$ then, if $w_{n_1} - \frac{wi_{n_1}}{2} \geq w_{n_2} - \frac{wi_{n_2}}{2}$ is satisfied then $n_1 \gg n_2 \vee n_1 \equiv n_2$ (**Case 2**). The following lemma proves this property.

Lemma 5.8: Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given three nodes $n_1 \in N$ and $n_2, n_3 \in Sea(T)$, with $(n \to n_1) \in E^*$, $(n_1 \to n_2), (n_1 \to n_3) \in E$, and $u_{n_2} \geq d_{n_2}$ and $u_{n_3} \geq d_{n_3}$, $n_2 \gg n_3 \vee n_2 \equiv n_3$ if and only if $w_{n_2} - \frac{wi_{n_2}}{2} \geq w_{n_3} - \frac{wi_{n_3}}{2}$.

Proof: First, if $|d_{n_2} - u_{n_2}| \leq |d_{n_3} - u_{n_3}|$ then $n_2 \gg n_3 \vee n_2 \equiv n_3$. Thus it is enough to prove that $w_{n_2} - \frac{wi_{n_2}}{2} \geq w_{n_3} - \frac{wi_{n_3}}{2}$ implies $|d_{n_2} - u_{n_2}| \leq |d_{n_3} - u_{n_3}|$ and vice versa when $u_n \geq d_n$ in both nodes and they are brothers.

$w_{n_2} - \frac{wi_{n_2}}{2} \geq w_{n_3} - \frac{wi_{n_3}}{2}$
$2w_{n_2} - wi_{n_2} \geq 2w_{n_3} - wi_{n_3}$
We replace w_{n_2} and w_{n_3} using Equation 2:
$2(d_{n_2} + wi_{n_2}) - wi_{n_2} \geq 2(d_{n_3} + wi_{n_3}) - wi_{n_3}$
$2d_{n_2} + wi_{n_2} \geq 2d_{n_3} + wi_{n_3}$
We add $-w_n$:
$-w_n + 2d_{n_2} + wi_{n_2} \geq -w_n + 2d_{n_3} + wi_{n_3}$
$w_n - 2d_{n_2} - wi_{n_2} \leq w_n - 2d_{n_3} - wi_{n_3}$
We replace w_n using Equation 1:
$(d_{n_2} + u_{n_2} + wi_{n_2}) - 2d_{n_2} - wi_{n_2} \leq (d_{n_3} + u_{n_3} + wi_{n_3}) - 2d_{n_3} - wi_{n_3}$
$-d_{n_2} + u_{n_2} \leq -d_{n_3} + u_{n_3}$
$u_{n_2} - d_{n_2} \leq u_{n_3} - d_{n_3}$
As $u_n \geq d_n$ in both nodes:
$|u_{n_2} - d_{n_2}| \leq |u_{n_3} - d_{n_3}|$
$|d_{n_2} - u_{n_2}| \leq |d_{n_3} - u_{n_3}|$

∎

If two nodes n_1 and n_2 are brothers and $d_{n_1} \geq u_{n_1}$ and $n_2 \to^+ n_3$ then, if $n_1 \equiv n_2$ then $n_1 \gg n_3 \vee n_1 \equiv n_3$. The following lemma proves this property.

Lemma 5.9: Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given four nodes $n_1 \in N$ and $n_2, n_3, n_4 \in Sea(T)$, with $(n \to n_1) \in E^*$, $(n_1 \to n_2), (n_1 \to n_3) \in E$, $(n_3 \to n_4) \in E^+$, if $d_{n_2} \geq u_{n_2}$ and $n_2 \equiv n_3$ then $n_2 \gg n_4 \vee n_2 \equiv n_4$.

Proof: This can be trivially proved having into account that $d_{n_3} \leq u_{n_3}$ when $d_{n_2} \geq u_{n_2}$ by Lemma 5.5 and then by Lemma 5.4 we know that $n_3 \gg n_4 \vee n_3 \equiv n_4$ and as $n_2 \equiv n_3$ then $n_2 \gg n_4 \vee n_2 \equiv n_4$. ∎

If two nodes n_1 and n_2 are brothers and $d_{n_1} \leq u_{n_1} \wedge d_{n_2} \leq u_{n_2}$ and $n_2 \to^+ n_3$ then, if $n_1 \equiv n_2$ then $n_1 \gg n_3 \vee n_1 \equiv n_3$. The following lemma proves this property.

Lemma 5.10: Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given four nodes $n_1 \in N$ and $n_2, n_3, n_4 \in Sea(T)$, with $(n \to n_1) \in E^*$, $(n_1 \to n_2), (n_1 \to n_3) \in E$, $(n_3 \to n_4) \in E^+$, if $d_{n_2} \leq u_{n_2}$ and $d_{n_3} \leq u_{n_3}$ and $n_2 \equiv n_3$ then $n_2 \gg n_4 \vee n_2 \equiv n_4$.

Proof: This can be trivially proved having into account that $d_{n_3} \leq u_{n_3}$ and then by Lemma 5.4 we know that $n_3 \gg n_4 \vee n_3 \equiv n_4$ and as $n_2 \equiv n_3$ then $n_2 \gg n_4 \vee n_2 \equiv n_4$. ∎

If two nodes n_1 and n_2 are brothers and $n_1 \gg n_2$ and $n_2 \to^+ n_3$ then $n_1 \gg n_3$. The following lemma proves this property.

Lemma 5.11: Given a variable MET $T = (N, E, M)$ whose root is $n \in N$, and given four nodes $n_1 \in N$ and $n_2, n_3, n_4 \in Sea(T)$, with $(n \to n_1) \in E^*$, $(n_1 \to n_2), (n_1 \to n_3) \in E$, $(n_3 \to n_4) \in E^+$, if $n_2 \gg n_3$ then $n_2 \gg n_4$.

Proof: We show that if $n_2 \gg n_3$ then $d_{n_3} < u_{n_3}$. We prove it by contradiction assuming that $d_{n_3} \geq u_{n_3}$ when $n_2 \gg n_3$. First, as n_2 and n_3 are brothers we know that $w_n \geq d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3}$, then $w_n = d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3} + inc$ with $inc \geq 0$. Therefore, if $|d_{n_2} - u_{n_2}| < |d_{n_3} - u_{n_3}|$ then $n_2 \gg n_3$. Thus it is enough to prove that $|d_{n_2} - u_{n_2}| < |d_{n_3} - u_{n_3}|$ is not satisfied when $d_{n_3} \geq u_{n_3}$ and n_2 and n_3 are brothers.

$|d_{n_2} - u_{n_2}| < |d_{n_3} - u_{n_3}|$
As $d_{n_3} \geq u_{n_3}$ by Lemma 5.5 we know that $u_{n_2} \geq d_{n_2}$:
$u_{n_2} - d_{n_2} < d_{n_3} - u_{n_3}$
We replace u_{n_2} and u_{n_3} using Equation 1:
$(w_n - d_{n_2} - wi_{n_2}) - d_{n_2} < d_{n_3} - (w_n - d_{n_3} - wi_{n_3})$
$w_n - 2d_{n_2} - wi_{n_2} < 2d_{n_3} - w_n + wi_{n_3}$
$2w_n < 2d_{n_2} + 2d_{n_3} + wi_{n_2} + wi_{n_3}$
$w_n < d_{n_2} + d_{n_3} + \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
We replace w_n:
$d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3} + inc < d_{n_2} + d_{n_3} + \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
$wi_{n_2} + wi_{n_3} + inc < \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
$\frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2} + inc < 0$

But, this is a contradiction with $wi_{n_2}, wi_{n_3}, inc \geq 0$. Hence, $d_{n_3} < u_{n_3}$.

Now we show that, if $n_2 \gg n_3$ then $n_2 \gg n_4$. We prove it by contradiction assuming that $n_4 \gg n_2 \vee n_4 \equiv n_2$ when $n_2 \gg n_3$. First, we know that $d_{n_3} < u_{n_3}$. Therefore we know that $d_{n_4} = d_{n_3} - wi_{n_4} - dec$ and $u_{n_4} = u_{n_3} + wi_{n_3} + dec$ with $dec \geq 0$, where dec represents the weight of the possible brothers of n_4.

$|d_{n_3} - u_{n_3}| > |d_{n_2} - u_{n_2}| \geq |d_{n_4} - u_{n_4}|$
We replace d_{n_4} and u_{n_4}:
$|d_{n_3} - u_{n_3}| > |d_{n_2} - u_{n_2}|$
$|d_{n_2} - u_{n_2}| \geq |(d_{n_3} - wi_{n_4} - dec) - (u_{n_3} + wi_{n_3} + dec)|$
$|d_{n_2} - u_{n_2}| \geq |d_{n_3} - wi_{n_4} - dec - u_{n_3} - wi_{n_3} - dec|$
$|d_{n_2} - u_{n_2}| \geq |d_{n_3} - u_{n_3} - wi_{n_3} - wi_{n_4} - 2dec|$

Note that $d_{n_3} - u_{n_3}$ must be positive, thus $d_{n_3} > u_{n_3}$. But this is a contradiction with $d_{n_3} < u_{n_3}$. ■

The following lemma ensures that given two nodes n_1 and n_2 where $d_{n_1} \geq u_{n_1}$ and $d_{n_2} \leq u_{n_2}$ and $n_1 \to n_2$ then if $w_n \geq w_{n_1} + w_{n_2} - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$ is satisfied then $n_1 \gg n_2 \vee n_1 \equiv n_2$ (**Case 4**).

Lemma 5.12: Given a variable MET $T = (N, E, M)$ and given two nodes $n_1, n_2 \in Sea(T)$, with $(n_1 \to n_2) \in E$, and $d_{n_1} \geq u_{n_1}$, and $d_{n_2} \leq u_{n_2}$, $n_1 \gg n_2 \vee n_1 \equiv n_2$ if and only if $w_n \geq w_{n_1} + w_{n_2} - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$.

Proof: First, if $|d_{n_1} - u_{n_1}| \leq |d_{n_2} - u_{n_2}|$ then $n_1 \gg n_2$ or $n_1 \equiv n_2$. Thus it is enough to prove that $w_n \geq w_{n_1} + w_{n_2} - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$ implies $|d_{n_1} - u_{n_1}| \leq |d_{n_2} - u_{n_2}|$ and vice versa when $d_{n_1} \geq u_{n_1}$ and $d_{n_2} \leq u_{n_2}$.

$w_n \geq w_{n_1} + w_{n_2} - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$
We replace w_{n_1}, w_{n_2} using Equation 2:
$w_n \geq (d_{n_1} + wi_{n_1}) + (d_{n_2} + wi_{n_2}) - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$
$w_n \geq d_{n_1} + d_{n_2} + \frac{wi_{n_1}}{2} + \frac{wi_{n_2}}{2}$
$2w_n \geq 2d_{n_1} + 2d_{n_2} + wi_{n_1} + wi_{n_2}$
$-2w_n \leq -2d_{n_1} - 2d_{n_2} - wi_{n_1} - wi_{n_2}$
$-w_n + 2d_{n_1} + wi_{n_1} \leq w_n - 2d_{n_2} - wi_{n_2}$
We replace w_n using Equation 1:
$-(d_{n_1} + u_{n_1} + wi_{n_1}) + 2d_{n_1} + wi_{n_1} \leq (d_{n_2} + u_{n_2} + wi_{n_2}) - 2d_{n_2} - wi_{n_2}$
$-d_{n_1} - u_{n_1} - wi_{n_1} + 2d_{n_1} + wi_{n_1} \leq d_{n_2} + u_{n_2} + wi_{n_2} - 2d_{n_2} - wi_{n_2}$
$-u_{n_1} + d_{n_1} \leq -d_{n_2} + u_{n_2}$
$d_{n_1} - u_{n_1} \leq u_{n_2} - d_{n_2}$
As $d_{n_1} \geq u_{n_1}$ and $d_{n_2} \leq u_{n_2}$:
$|d_{n_1} - u_{n_1}| \leq |u_{n_2} - d_{n_2}|$
$|d_{n_1} - u_{n_1}| \leq |d_{n_2} - u_{n_2}|$

■

Finally, we prove the correctness of Algorithm 2.

Theorem 3.4. *Let $T = (N, E, M)$ be a variable MET, then the execution of Algorithm 2 with T as input always terminates producing as output a node $n \in Sea(T)$ such that $\nexists n' \in Sea(T) \mid n' \gg n$.*

Proof: The finiteness of the algorithm is proved thanks to the following invariant: each iteration processes one single node, and the same node is never processed again. Therefore, because N is finite, the loop will terminate.

The correctness can be proved showing that after any number of iterations the algorithm always finishes with an optimal node. We prove it by induction on the number of iterations performed.

(Induction Hypothesis) After i iterations, the algorithm has a candidate node $Best \in Sea(T)$ such that $\forall n' \in Sea(T), (Best \to n') \notin E^*, Best \gg n' \vee Best \equiv n'$.

(Inductive Case) We prove that the iteration $i + 1$ of the algorithm will select a new candidate node $Candidate$ such that $Candidate \gg Best \vee Candidate \equiv Best$, or it will terminate selecting an optimal node.
Firstly, when the condition in Line (5) is satisfied $Best$ and $Candidate$ are the same node (say n'). According to the induction hypothesis, this node is better or equal than any other of the nodes in the set $\{n'' \in Sea(T) | (n' \to n'') \notin E^*\}$. Therefore, because n' has no children, then it is an optimal node; and it is returned in Line (5). Otherwise, if the condition in Line (5) is not satisfied, Line (7) in the algorithm ensures that $w_{Best} - \frac{wi_{Best}}{2} > \frac{w_n}{2}$ being n the root of T because in the iteration i the loop did not terminate or because $Best$ is the root (observe that an exception can happen when all nodes have an individual weight of 0. But in this case all nodes are optimal, and thus the node returned by the algorithm is optimal). Then we know that $d_{Best} > u_{Best}$ by Lemma 5.2. Moreover, according to Lines (4) and (6), we know that $Candidate$ is the heaviest child of $Best$. We have two possibilities:

- $d_{Candidate} > u_{Candidate}$: In this case the loop does not terminate and $\forall n' \in Sea(T)$, $(Candidate \to n') \notin E^*, Candidate \gg n' \vee Candidate \equiv n'$. Firstly, by Lemma 5.3 we know that $Candidate \gg Best \vee Candidate \equiv Best$, and thus, by the induction hypothesis we know that $\forall n' \in Sea(T), (Best \to n') \notin E^*, Candidate \gg n' \vee Candidate \equiv n'$. By Lemma 5.6 we know that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a brother of $Candidate$. Moreover, by Lemma 5.9 and 5.11 we

can ensure that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a descendant of a $candidate$'s brother.

- $d_{Candidate} \leq u_{Candidate}$: In this case the loop terminates (Line (7)) and we know by Lemma 5.7 that $d_{n'} \leq u_{n'}$ being n' any brother of $Candidate$. In Line (8) according to Lemma 5.8 we select the Candidate such that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a brother of $Candidate$. Moreover, by Lemma 5.10 and 5.11 we can ensure that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a descendant of a $candidate$'s brother. Then equation $(w_n \geq w_{Best} + w_{Candidate} - \frac{wi_{Best}}{2} - \frac{wi_{Candidate}}{2})$ is applied in Line (10) to select an optimal node. Lemma 5.12 ensure that the node selected is an optimal node because, according to Lemma 5.4, for all descendant n' of $Candidate$, $Candidate \gg n' \vee Candidate \equiv n'$.

■

VI. Conclusion

During three decades, Divide & Query has been the more efficient algorithmic debugging strategy. On the practical side, all current algorithmic debuggers implement D&Q [2], [4], [6], [8], [10], [11], [12], [13], [14], and experiments [3], [17] (see also http://users.dsic.upv.es/~jsilva/DDJ/#Experiments) demonstrate that it performs on average 2-36% less questions than other strategies. On the theoretical side, because D&Q intends a dichotomic search, it has been thought optimal with respect to the number of questions performed, and thus research on algorithmic debugging strategies has focused on other aspects such as reducing the complexity of questions.

In this work we show that in some situations current algorithms for D&Q are incomplete and inefficient because they are not able to find all optimal nodes, and sometimes they return nodes that are not optimal. We have identified the sources of inefficiency and provided examples that show both the incompleteness and incorrectness of the technique.

A relevant contribution of this work is a new algorithm for D&Q that optimally divides the ET even in the case where all nodes of the ET can have different individual weights in $\mathcal{R}^+ \cup \{0\}$. The algorithm has been proved terminating and correct. And a slightly modified version of the algorithm has been provided that returns all optimal solutions, thus being complete. We have implemented the technique and experiments show that it is more efficient than all previous algorithms.

Other important contributions are the proof that D&Q is not optimal in the worst case as supposed, and the definition of the first optimal strategy for algorithmic debugging.

The implementation—including the source code—and the experiments are publicly available at: http://users.dsic.upv.es/~jsilva/DDJ.

Acknowledgements

This work has been partially supported by the Spanish *Ministerio de Ciencia e Innovación* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052.

References

[1] E. Av-Ron. *Top-Down Diagnosis of Prolog Programs*. PhD thesis, Weizmanm Institute, 1984.

[2] B. Braßel and F. Huch. The Kiel Curry system KiCS. In *Proc of 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and 21st Workshop on (Constraint) Logic Programming (WLP 2007)*, pages 215–223. Technical Report 434, University of Würzburg, 2007.

[3] R. Caballero. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *Proc. of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05)*, pages 8–13, New York, USA, 2005. ACM Press.

[4] R. Caballero. Algorithmic Debugging of Java Programs. In *Proc. of the 2006 Workshop on Functional Logic Programming (WFLP'06)*, pages 63–76. Electronic Notes in Theoretical Computer Science, 2006.

[5] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. A Declarative Debugger for Maude Functional Modules. *Electronic Notes in Theoretical Computer Science*, 238:63–81, June 2009.

[6] T. Davie and O. Chitil. Hat-delta: One Right Does Make a Wrong. In *Seventh Symposium on Trends in Functional Programming, TFP 06*, April 2006.

[7] V. Hirunkitti and C. J. Hogger. A Generalised Query Minimisation for Program Debugging. In *Proc. of International Workshop of Automated and Algorithmic Debugging (AADEBUG'93)*, pages 153–170. Springer LNCS 749, 1993.

[8] D. Insa and J. Silva. An Algorithmic Debugger for Java. *In Proc. of the 26th IEEE International Conference on Software Maintenance*, 0:1–6, 2010.

[9] J. W. Lloyd. Declarative Error Diagnosis. *New Gen. Comput.*, 5(2):133–154, 1987.

[10] W. Lux. Münster Curry User's Guide (release 0.9.10 of may 10, 2006). Available at: http://danae.uni-muenster.de/~lux/curry/user.pdf, 2006.

[11] I. MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2005.

[12] L. Naish, P. W. Dart, and J. Zobel. The NU-Prolog Debugging Environment. In A. Porto, editor, *Proceedings of the Sixth International Conference on Logic Programming*, pages 521–536, Lisboa, Portugal, June 1989.

[13] H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, May 1998.

[14] B. Pope. *A Declarative Debugger for Haskell*. PhD thesis, The University of Melbourne, Australia, 2006.

[15] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.

[16] J. Silva. A Comparative Study of Algorithmic Debugging Strategies. In *Proc. of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06)*, pages 143–159. Springer LNCS 4407, 2007.

[17] J. Silva. An Empirical Evaluation of Algorithmic Debugging Strategies. Technical Report DSIC-II/10/09, UPV, 2009. Available from URL: http://www.dsic.upv.es/~jsilva/research.htm#techs.

[18] J. Silva. A Survey on Algorithmic Debugging Strategies. *Advances in Engineering Software*, 42(11):976–991, 2011.