Document downloaded from:

http://hdl.handle.net/10251/35537

This paper must be cited as:

Insa Cabrera, D.; Silva Galiana, JF. (2011). Optimal divide and query. En Progress in Artificial Intelligence. Springer Verlag (Germany). 7026:224-238. doi:10.1007/978-3-642-24769-9_17.

The final publication is available at

http://link.springer.com/chapter/10.1007/978-3-642-24769-9_17

Copyright Springer Verlag (Germany)

# Optimal Divide and Query*

David Insa and Josep Silva

Universitat Politècnica de València
Camino de Vera s/n, E-46022 Valencia, Spain.
{dinsa,jsilva}@dsic.upv.es

**Abstract.** Algorithmic debugging is a semi-automatic debugging technique that allows the programmer to precisely identify the location of bugs without the need to inspect the source code. The technique has been successfully adapted to all paradigms and mature implementations have been released for languages such as Haskell, Prolog or Java. During three decades, the algorithm introduced by Shapiro and later improved by Hirunkitti has been thought optimal. In this paper we first show that this algorithm is not optimal, 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 is proven optimal, and we introduce some equations that allow the algorithm to identify all optimal solutions.

**Keywords:** Algorithmic Debugging, Strategy, Divide & Query

## 1 Introduction

Debugging is one of the most important but less automated (and, thus, time-consuming) tasks in the software development process. The programmer is often forced to manually explore the code or iterate over it using, e.g., breakpoints, and this process usually requires a deep understanding of the source code to find the bug. *Algorithmic debugging* [17] is a semi-automatic debugging technique that has been extended to practically all paradigms [18]. Recent research has produced new advances to increase the scalability of the technique producing new scalable and mature debuggers. The technique is based on the answers of the programmer to a series of questions generated automatically by the algorithmic debugger. The questions are always whether a given result of an activation of a subcomputation with given input values is actually correct. The answers provide the debugger with information about the correctness of some (sub)computations of a given program; and the debugger uses them to guide the search for the bug until a buggy portion of code is isolated.

*Example 1.* Consider this simple Haskell program inspired in a similar example by [6]. It wrongly (it has a bug) implements the sorting algorithm *Insertion Sort*:

* 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.

```
main = insort [2,1,3]

insort [] = []
insort (x:xs) = insert x (insort xs)

insert x [] = [x]
insert x (y:ys) = if x>=y then (x:y:ys)
                          else (y:(insert x ys))
```

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

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

Bug found in rule:
insert x (y:ys) = if x>=y then _ else (y:(insert x ys))
```

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

Typically, algorithmic debuggers have a front-end that produces a data structure representing a program execution—the so-called *execution tree* (ET) [15]—; and a back-end that uses the ET to ask questions and process the answers of the programmer to locate the bug. For instance, the ET of the program in Example 1 is depicted in Figure 1.

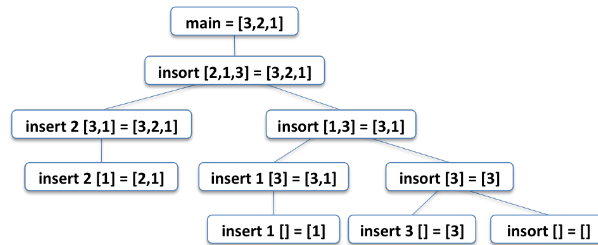


**Fig. 1.** ET of the program in Example 1

The strategy used to decide what nodes of the ET should be asked is crucial for the performance of the technique. Since the definition of algorithmic debugging, there have been a lot of research concerning the definition of new strategies trying to minimize the number of questions [18]. We conducted several experiments to measure the performance of all current algorithmic debugging

strategies. The results of the experiments are shown in Figure 2, where the first column contains the names of the benchmarks; column `nodes` shows the number of nodes in the ET associated with each benchmark; and the other columns represent algorithmic debugging strategies [18] that are ordered according to their performance: Optimal Divide & Query (`D&QO`), Divide & Query by Hirunkitti (`D&QH`), Divide & Query by Shapiro (`D&QS`), Divide by Rules & Query (`DR&Q`), Heaviest First (`HF`), More Rules First (`MRF`), Hat Delta Proportion (`HD-P`), Top-Down (`TD`), Hat Delta YES (`HD-Y`), Hat Delta NO (`HD-N`), Single Stepping (`SS`).

| Benchmark | Nodes | D&QO | D&QH | D&QS | DR&Q | HF | MRF | HD-P | TD | HD-Y | HD-N | SS | Average |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| NumReader | *12* | 28,99 | 28,99 | 31,36 | 29,59 | 44,38 | 44,38 | 49,70 | 49,70 | 49,70 | 49,70 | 53,25 | 41,80 |
| Orderings | *46* | 12,04 | 12,09 | 12,63 | 14,40 | 17,16 | 17,29 | 21,05 | 20,82 | 20,60 | 19,60 | 51,02 | 19,88 |
| Factoricer | *62* | 9,83 | 9,83 | 9,93 | 20,03 | 12,55 | 12,55 | 15,04 | 12,55 | 15,04 | 18,29 | 50,77 | 16,94 |
| Sedgewick | *12* | 30,77 | 30,77 | 33,14 | 30,77 | 34,91 | 34,91 | 43,79 | 43,20 | 43,79 | 43,79 | 53,25 | 38,46 |
| Clasifier | *23* | 19,79 | 20,31 | 22,40 | 21,88 | 22,92 | 23,26 | 32,12 | 31,94 | 32,12 | 34,55 | 51,91 | 28,47 |
| LegendGame | *71* | 8,87 | 8,87 | 8,95 | 16,72 | 11,15 | 11,23 | 14,68 | 13,37 | 14,68 | 16,94 | 50,68 | 16,01 |
| Cues | *18* | 31,58 | 32,41 | 32,41 | 32,41 | 33,24 | 34,63 | 39,06 | 42,11 | 39,06 | 44,32 | 52,35 | 37,60 |
| Romanic | *123* | 6,40 | 10,84 | 11,23 | 13,56 | 7,44 | 11,88 | 13,29 | 13,41 | 13,29 | 13,30 | 50,40 | 15,00 |
| FibRecursive | *4.619* | 0,27 | 0,27 | 0,28 | 1,20 | 0,33 | 0,41 | 3,92 | 0,46 | 3,92 | 0,48 | 50,01 | 5,59 |
| Risk | *33* | 16,78 | 16,78 | 18,08 | 19,38 | 18,69 | 18,69 | 24,31 | 31,14 | 24,31 | 32,79 | 51,38 | 24,76 |
| FactTrans | *198* | 3,89 | 3,89 | 3,93 | 6,22 | 6,58 | 6,58 | 7,37 | 7,16 | 7,24 | 7,50 | 50,25 | 10,06 |
| RndQuicksort | *72* | 8,73 | 8,73 | 8,73 | 11,41 | 12,03 | 12,23 | 13,62 | 13,51 | 12,93 | 14,54 | 50,67 | 15,19 |
| BinaryArrays | *128* | 5,52 | 5,52 | 5,71 | 7,13 | 7,75 | 7,94 | 7,90 | 8,59 | 8,15 | 8,71 | 50,38 | 11,21 |
| FibFactAna | *351* | 2,44 | 2,44 | 2,45 | 5,38 | 7,61 | 7,71 | 6,40 | 8,57 | 7,39 | 5,99 | 50,14 | 9,68 |
| NewtonPol | *7* | 39,06 | 39,06 | 43,75 | 39,06 | 43,75 | 43,75 | 45,31 | 45,31 | 45,31 | 45,31 | 54,69 | 44,03 |
| RegresionTest | *18* | 23,27 | 23,27 | 25,21 | 25,21 | 26,87 | 26,87 | 32,96 | 32,96 | 32,96 | 32,96 | 52,35 | 30,45 |
| BoubleFibArrays | *171* | 4,40 | 4,41 | 4,57 | 11,40 | 5,95 | 6,96 | 24,50 | 6,96 | 24,87 | 6,96 | 50,29 | 13,75 |
| ComplexNumbers | *60* | 10,02 | 10,02 | 10,32 | 11,31 | 11,39 | 11,39 | 15,78 | 15,75 | 15,80 | 19,19 | 50,79 | 16,53 |
| Integral | *5* | 44,44 | 44,44 | 47,22 | 44,44 | 50,00 | 50,00 | 50,00 | 50,00 | 50,00 | 50,00 | 55,56 | 48,74 |
| TestMath | *48* | 11,91 | 11,91 | 12,16 | 12,99 | 15,95 | 16,28 | 22,41 | 24,20 | 23,87 | 22,37 | 50,98 | 20,46 |
| TestMath2 | *228* | 3,51 | 3,51 | 3,51 | 9,73 | 10,55 | 10,81 | 12,29 | 28,56 | 13,24 | 14,37 | 50,22 | 14,57 |
| Figures | *113* | 6,72 | 6,75 | 6,79 | 8,09 | 7,68 | 7,79 | 10,17 | 10,60 | 10,16 | 10,76 | 50,43 | 12,36 |
| FactCalc | *59* | 10,11 | 10,14 | 10,42 | 11,53 | 13,69 | 14,22 | 20,47 | 18,50 | 20,47 | 20,69 | 50,81 | 18,28 |
| SpaceLimits | *127* | 12,95 | 16,07 | 19,15 | 21,74 | 13,68 | 16,80 | 22,87 | 22,78 | 22,86 | 26,15 | 50,38 | 22,31 |
| Argparser | *129* | 12,10 | 12,10 | 13,08 | 20,48 | 13,07 | 13,32 | 15,98 | 15,98 | 15,98 | 15,98 | 50,38 | 18,04 |
| Cglib | *1.216* | 1,93 | 1,93 | 2,33 | 2,12 | 2,52 | 2,65 | 6,14 | 6,61 | 5,73 | 7,32 | 50,04 | 8,12 |
| Kxml2 | *1.172* | 2,86 | 2,86 | 3,01 | 3,56 | 3,06 | 3,48 | 8,58 | 6,79 | 6,97 | 7,77 | 50,04 | 9,00 |
| Javassist | *1.357* | 4,34 | 4,34 | 5,44 | 4,49 | 4,74 | 4,75 | 6,20 | 5,86 | 9,26 | 6,06 | 50,04 | 9,59 |
| **Average** | **374,21** | **13,34** | **13,66** | **14,58** | **16,29** | **16,41** | **16,88** | **20,92** | **20,98** | **21,06** | **21,30** | **51,19** | **20,60** |

**Fig. 2.** Performance of algorithmic debugging strategies

For each benchmark, we produced its associated ET and assumed that the buggy node could be any node of the ET (i.e., any subcomputation in the execution of the program could be buggy). Therefore, we performed a different experiment for each possible case and, hence, each cell of the table summarizes a number of experiments that were automatized. In particular, benchmark *Factoricer* has been debugged 62 times with each strategy; each time, the buggy node was a different node, and the results shown are the average number of questions performed by each strategy with respect to the number of nodes (i.e., the mean percentage of nodes asked). Similarly, benchmark *Cglib* has been debugged 1216 times with each strategy, and so on.

Observe that the best algorithmic debugging strategies in practice are the two variants of Divide and Query (ignoring our new technique D&QO). Moreover, 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., [4, 5, 8, 16]). 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 [18].

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 2 we recall and formalize the strategy D&Q and we show with counterexamples that it is suboptimal and incomplete. Then, in Section 3 we introduce two new algorithms for D&Q that are optimal and complete. Each algorithm is useful for a different type of ET. Finally, Section 4 concludes. Proofs of technical results can be found in [9].

## 2 D&Q by Shapiro vs. D&Q by Hirunkitti

In this section we formalize the strategy D&Q to show the differences between the original version by Shapiro [17] and the improved version by Hirunkitti [7]. 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 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]

[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 *Correct* or *Wrong*.

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 *buggy node* and it is pointed as responsible of 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.

**Algorithm 1** General algorithm for algorithmic debugging

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

**begin**
(1) **do**
(2) node = selectNode($T$)
(3) answer = askNode(node)
(4) **if** (answer = *Wrong*)
(5) **then** $M$(node) = *Wrong*
(6) buggyNode = node
(7) $N = \{n \in N \mid (\text{node} \to n) \in E^*\}$
(8) **else** $N = N \setminus \{n \in N \mid (\text{node} \to n) \in E^*\}$
(9) **while** $(\exists n \in N, M(n) = \mathit{Undefined})$
(10) **return** buggyNode
**end**

Both D&Q by Shapiro and D&Q by Hirunkitti assume 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 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 $\frac{n}{2}$. The original algorithm by Shapiro always selects:

- the heaviest node $n'$ whose weight is as close as possible to $\frac{n}{2}$ with $w_{n'} \leq \frac{n}{2}$

Hirunkitti and Hogger noted that this is not enough to divide the MET by the half and their improved version always selects the node whose weight is closer to $\frac{n}{2}$ between:

- the heaviest node $n'$ whose weight is as close as possible to $\frac{n}{2}$ with $w_{n'} \leq \frac{n}{2}$, or

– the lightest node $n'$ whose weight is as close as possible to $\frac{n}{2}$ with $w_{n'} \geq \frac{n}{2}$

Because it is better, in the rest of the article we only consider Hirunkitti's D&Q and refer to it as D&Q.

### 2.1 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] Moreover, we show that if the MET contains a wrong node, then D&Q is correct (all nodes found divide the MET optimally), but it is incomplete (it cannot find some nodes that optimally divide the MET). The intuition beyond these limitations 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 3 (left) where the black node is marked as wrong and 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 3 (center) where each node is labeled with its weight and the black node is marked as wrong. In both cases D&Q would behave exactly in the same way, because it completely ignores the fact that some nodes are marked as wrong. 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 nodes marked as wrong as part of their own weight, and this is a source of inefficiency.

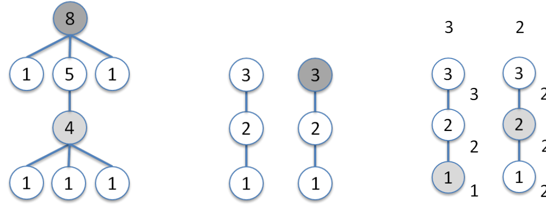

**Fig. 3.** Behavior of Divide and Query

In the METs of Figure 3 (center) 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 3 (right) that selecting node 1 is suboptimal, and the strategy

[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*.

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 4 show that D&Q is incomplete. Observe that the METs have 5 nodes, thus D&Q would always select the node with weight 2. However, the node with weight 4 is equally optimal (both need $\frac{16}{6}$ questions as an average to find the bug) but it will be never selected by D&Q because its weight is far from the half of the tree $\frac{5}{2}$.

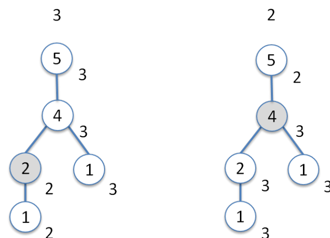


**Fig. 4.** Incompleteness of Divide and Query

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 one. 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 5. In this MET, D&Q would select node $n_1$ because its weight is closer to $\frac{21}{2}$ than any other node. However, node $n_2$ is the node that better divides the tree in two parts with the same probability of containing the bug.

In summary, (1) D&Q is suboptimal when the MET is free of wrong nodes, (2) D&Q is incomplete when the MET contains wrong nodes, (3) D&Q is correct when the MET contains wrong nodes and all the nodes of the MET have the same weight, but (4) D&Q is suboptimal when the MET contains wrong nodes and the nodes of the MET have different individual weights.

## 3 Optimal D&Q

In this section we introduce a new version of D&Q that tries to divide the MET into two parts with the same probability of containing the bug (instead of two

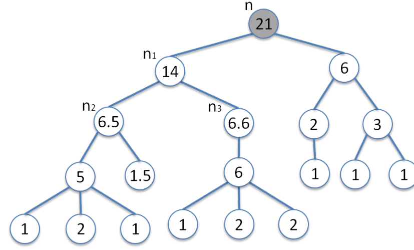


**Fig. 5.** MET with decimal individual weights

parts with the same weight). We introduce new algorithms that are correct and complete even if the MET contains nodes with different individual weights. For this, we define the *search area* of a MET as the set of undefined nodes.

**Definition 2 (Search area).** *Let $T = (N, E, M)$ be a MET. The* search area *of $T$, $Sea(T)$, is defined as $\{n \in N \mid M(n) = \mathit{Undefined}\}$.*

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 [10]. 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:

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

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 an important generalization of D&Q where it is assumed that all nodes have the same individual weight and it is always 1.

### 3.1 Debugging ETs where all nodes have the same individual weight $wi \in \mathcal{R}^+$

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 $n$, whose weight is

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

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

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

*Example 2.* Consider the MET in Figure 6. Assuming that the root $n$ is the only node marked as wrong and all nodes have an individual weight of 1, then $Sea(T)$ contains all nodes except $n$, $Up(n') = 4$ (total weight of the gray nodes), and $Down(n') = 3$ (total weight of the white nodes).

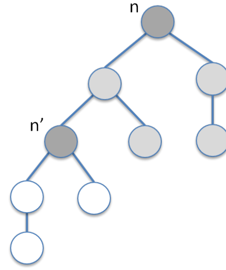


**Fig. 6.** Functions Up and Down

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

$w_n = Up(n') + Down(n') + wi_{n'}$ (Equation 1)

$w_{n'} = Down(n') + wi_{n'}$ (Equation 2)

Intuitively, given a node $n$, what we want to divide by the half is the area formed by $Up(n) + Down(n)$. That is, $n$ will not be part of $Sea(T)$ after it has been answered, thus the objective is to make $Up(n)$ equal to $Down(n)$. 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 $Sea(T)$ better than $n_2$ (i.e., $|Up(n_1) - Down(n_1)| < |Up(n_2) - Down(n_2)|$). And we use $n_1 \equiv n_2$ to express that $n_1$ and $n_2$ equally divide $Sea(T)$. If we find a node $n$ such that $Up(n) = Down(n)$ then $n$ produces an optimal division, and should be selected by the strategy. If an optimal solution cannot be found, the following theorem states how to compare the nodes in order to decide which of them should be selected.

**Theorem 1.** *Given a MET $T = (N, E, M)$ whose root is $n \in N$, where $\forall n, n' \in N, wi_n = wi_{n'}$ and $\forall n \in N, wi_n > 0$, and given two nodes $n_1, n_2 \in Sea(T)$, with $w_{n_1} > w_{n_2}$, if $w_n > w_{n_1} + w_{n_2} - wi_n$ then $n_1 \gg n_2$.*

**Proposition 1.** *Given a MET $T = (N, E, M)$ whose root is $n \in N$, where $\forall n, n' \in N, wi_n = wi_{n'}$ and $\forall n \in N, wi_n > 0$, and given two nodes $n_1, n_2 \in Sea(T)$, with $w_{n_1} > w_{n_2}$, if $w_n = w_{n_1} + w_{n_2} - wi_n$ then $n_1 \equiv n_2$.*

Theorem 1 is useful when one node is heavier than the other. In the case that both nodes have the same weight, then the following theorem guarantees that they both equally divide the MET in all situations.

**Theorem 2.** *Let $T = (N, E, M)$ be a MET where $\forall n, n' \in N, wi_n = wi_{n'}$ and $\forall n \in N, wi_n > 0$, and let $n_1, n_2 \in Sea(T)$ be two nodes, if $w_{n_1} = w_{n_2}$ then $n_1 \equiv n_2$.*

**Corollary 1.** *Given a MET $T = (N, E, M)$ where $\forall n, n' \in N, wi_n = wi_{n'}$ and $\forall n \in N, wi_n > 0$, and given a node $n \in Sea(T)$, then $n$ optimally divides $Sea(T)$ if and only if $Up(n) = Down(n)$.*

While Corollary 1 states the objective of optimal D&Q (finding a node $n$ such that $Up(n) = Down(n)$), Theorems 1 and 2 provide a method to approximate this objective (finding a node $n$ such that $|Up(n) - Down(n)|$ is minimum in $Sea(T)$).

**An algorithm for Optimal D&Q.** Theorem 1 and Proposition 1 provide equation $w_n \geq w_{n_1} + w_{n_2} - wi_n$ to compare two nodes $n_1, n_2$ by efficiently determining $n_1 \gg n_2$, $n_1 \equiv n_2$ or $n_1 \ll n_2$. However, with only this equation, we should compare all nodes to select the best of them (i.e., $n$ such that $\nexists n', n' \gg n$). Hence, in this section we provide an algorithm that allows us to find the best node in a MET with a minimum set of node comparisons.

Given a MET, Algorithm 2 efficiently determines the best node to divide $Sea(T)$ by the 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)

**Input:** A MET $T = (N, E, M)$ whose root is $n \in N$,
$\forall n_1, n_2 \in N, wi_{n_1} = wi_{n_2}$ and $\forall n_1 \in N, wi_{n_1} > 0$
**Output:** A node $n' \in N$
**Preconditions:** $\exists n \in N$, $M(n) = Undefined$

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

*Example 3.* Consider the MET in Figure 7 where $\forall n \in N, wi_n = 1$ and $M(n) =$ *Undefined*. Observe that Algorithm 2 only needs to apply the equation in Theorem 1 once to identify an optimal node. Firstly, it traverses the MET top-down from the root selecting at each level the heaviest node until we find a node whose weight is smaller than the half of the MET $(\frac{w_n}{2})$, thus, defining a path in the MET that is colored in gray. Then, the algorithm uses the equation $w_n \geq w_{n_1} + w_{n_2} - wi_n$ to compare nodes $n_1$ and $n_2$. Finally, the algorithm selects $n_1$.

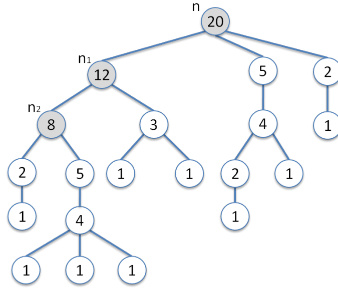


**Fig. 7.** Defining a path in a MET to find the optimal node

In order to prove the correctness of Algorithm 2, we need to prove that (1) the node returned is really an optimal node, and (2) this node will always be found by the algorithm (i.e., it is always in the path defined by the algorithm).

The first point can be proven with Theorems 1 and 2. The second point is the key idea of the algorithm and it relays on an interesting property of the path defined: while defining the path in the MET, only four cases are possible, and all of them coincide in that the subtree of the heaviest node will contain an optimal 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 four possible cases:

**Case 1:** $n_1$ and $n_2$ are brothers.
**Case 2:** $w_{n_1} > w_{n_2} \ \wedge \ w_{n_2} > \frac{w_n}{2}$.
**Case 3:** $w_{n_1} > \frac{w_n}{2} \ \wedge \ w_{n_2} \leq \frac{w_n}{2}$.
**Case 4:** $w_{n_1} > w_{n_2} \ \wedge \ w_{n_1} \leq \frac{w_n}{2}$.

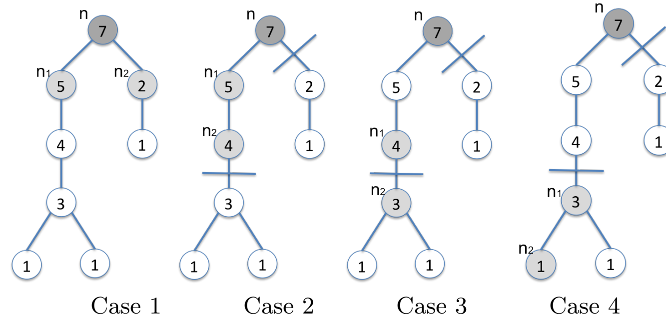


**Fig. 8.** Determining the best node in a MET (four possible cases)

We have proven—the individual proofs are part of the proof of Theorem 3—that in cases 1 and 4, the heaviest node is better (i.e., if $w_{n_1} > w_{n_2}$ then $n_1 \gg n_2$); In case 2, the lightest node is better; and in case 3, the best node must be

determined with the equation of Theorem 1. 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 case 1 is used to select a child, e.g., node 12 instead of node 5 or node 2, and node 8 instead of node 3. Case 2 is used to go down and select node 12 instead of node 20. Case 4 is used to stop going down and stop at node 8 because it is better than all its descendants. And it is also used to determine that node 2, 3 and 5 are better than all their descendants. Finally, case 3 is used to select the optimal node, 12 instead of 8. Note that D&Q could have selected node 8 that is equally close to $\frac{20}{2}$ than node 12; but it is suboptimal because $Up(8) = 12$ and $Down(8) = 7$ whereas $Up(12) = 8$ and $Down(12) = 11$.

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

**Theorem 3 (Correctness).** *Let $T = (N, E, M)$ be a MET where $\forall n, n' \in N, wi_n = wi_{n'}$ and $\forall n \in N, wi_n > 0$, 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, the equation in Theorem 1 in combination with the equation in Proposition 1 can be used to identify all optimal nodes in the MET. In particular, we could add a new line between lines (7) and (8) of Algorithm 2 to collect all candidates instead of one (see Theorem 2):

Candidates $= \{n' \in$ Children $\mid \forall n'' \in$ Children $, w_{n'} \geq w_{n''}\}$

then, in line (8) we could replace Candidate by Candidates, and we could modify lines (9) and (10) to return both Best and Candidates when the equation is an equality (see Proposition 1):

**if** $(w_n > w_{Best} + w_{Candidate} - wi_n)$ **then return** {Best}
**if** $(w_n = w_{Best} + w_{Candidate} - wi_n)$ **then return** {Best} $\cup$ Candidates
**else return** Candidates

With this modifications the algorithm is complete, and it returns nodes 2 and 4 in the MET of Figure 4 where D&Q can only detect node 2 as optimal.

### 3.2 Debugging METs where nodes can have different individual weights in $\mathcal{R}^+ \cup \{0\}$

In this section we generalize divide and query to the case where nodes can have different individual weights and these weights can be any value greater or equal to zero. As shown in Figure 5, in this general case traditional D&Q fails to identify the optimal node (it selects node $n_1$ but the optimal node is $n_2$). The algorithm presented in the previous section is also suboptimal when the individual weights can be different. For instance, in the MET of Figure 5, it would select node $n_3$. For this reason, in this section we introduce Algorithm 3, a general algorithm able to identify an optimal node in all cases. It does not mean that Algorithm 2 is useless. Algorithm 2 is optimal when all nodes have the same weight, and in that case, it is more efficient than Algorithm 3. Theorem 4 ensures the finiteness and correctness of Algorithm 3.

**Algorithm 3** Optimal D&Q General (SelectNode)

**Input:** A MET $T = (N, E, M)$ whose root is $n \in N$ and $\forall n_1 \in N, wi_{n_1} \geq 0$
**Output:** A node $n' \in N$
**Preconditions:** $\exists n \in N,\ M(n) = \mathit{Undefined}$

**begin**
(1) Candidate $= n$
(2) **do**
(3) Best = Candidate
(4) Children $= \{m \mid (\text{Best} \to m) \in E\}$
(5) **if** (Children $= \emptyset$) **then break**
(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** ($M$(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**

**Theorem 4 (Correctness).** *Let $T = (N, E, M)$ be a MET where $\forall n \in N, wi_n \geq 0$, then the execution of Algorithm 3 with $T$ as input always terminates producing as output a node $n \in Sea(T)$ such that $\nexists n' \in Sea(T) \mid n' \gg n$.*

### 3.3 Debugging METs where nodes can have different individual weights in $\mathcal{R}^+$

In the previous section we provided an algorithm that optimally selects an optimal node of the MET with a minimum set of node comparisons. But this algorithm is not complete due to the fact that we allow the nodes to have an individual weight of zero. For instance, when all nodes have an individual weight of zero, Algorithm 3 returns a single optimal node, but it is not able to find all optimal nodes.

Given a node, the difference between having an individual weight of zero and having a (total) weight of zero should be clear. The former means that this node did not cause the bug, the later means that none of the descendants of this node (neither the node itself) caused the bug. Surprisingly, the use of nodes with individual weights of zero has not been exploited in the literature. Assigning a (total) weight of zero to a node has been used for instance in the technique called *Trusting* [11]. This technique allows the user to trust a method. When this happens all the nodes related to this method and their descendants are pruned from the tree (i.e., these nodes have a (total) weight of zero).

If we add the restriction that nodes cannot be assigned with an individual weight of zero, then we can refine Algorithm 3 to ensure completeness. This refined version is Algorithm 4.

**Algorithm 4** Optimal D&Q General (SelectNode)

**Input:** A MET $T = (N, E, M)$ whose root is $n \in N$ and $\forall n_1 \in N, wi_{n_1} > 0$
**Output:** A set of nodes $O \subseteq N$
**Preconditions:** $\exists n \in N,\ M(n) = \mathit{Undefined}$

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

## 4 Conclusion

During three decades, D&Q has been the more efficient algorithmic debugging strategy. On the practical side, all current algorithmic debuggers implement D&Q [1, 3, 5, 8, 12–16], and experiments [2, 19] (see also http://users.dsic.upv.es/~jsilva/DDJ/#Experiments) demonstrate that D&Q 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.

The main contribution of this work is a new algorithm for D&Q that is optimal in all cases; including a generalization of the technique 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 (see column `D&QO` in Figure 2). The implementation—including the source code—and the experiments are publicly available at: http://users.dsic.upv.es/~jsilva/DDJ.

## References


1. 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.
2. 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.
3. 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.
4. 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.
5. 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.
6. P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimóthy. Generalized Algorithmic Debugging and Testing. *LOPLAS*, 1(4):303–322, 1992.
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. D. Insa and J. Silva. Optimal Divide and Query (extended version). *Available in the Computing Research Repository (http://arxiv.org/abs/1107.0350)*, July 2011.
10. J. W. Lloyd. Declarative Error Diagnosis. *New Gen. Comput.*, 5(2):133–154, 1987.
11. Y. Luo and O. Chitil. Algorithmic debugging and trusted functions. Technical report 10-07, University of Kent, Computing Laboratory, UK, August 2007.
12. 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.
13. I. MacLarty. *Practical Declarative Debugging of Mercury Programs*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne, 2005.
14. 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.
15. H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, May 1998.
16. B. Pope. *A Declarative Debugger for Haskell*. PhD thesis, The University of Melbourne, Australia, 2006.
17. E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.
18. 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.
19. 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`.