Master's Thesis

David Insa Cabrera

September 13, 2011

Abstract

Algorithmic Debugging (a.k.a. *Declarative Debugging*) is a semi-automatic debugging technique that allows the programmer to isolate the code where a bug is located without the need to see the source code. To do so, the programmer answers the questions performed by the debugger until it finds the bug and shows it to the user. These questions are chosen by the debugger using a strategy, and thus the reduction of the number of questions is a main objective of this debugging technique. In practice, the strategy that performs less questions has been *Divide & Query* (*D&Q* for short) since its definition 30 years ago. This strategy has been considered optimal in the worst case. In this work we show that D&Q is not optimal and we introduce two new strategies that have proved to be better. One of them is a new version of D&Q where its behavior is improved in some situations. While the other is the first version of a new strategy that is optimal in the worst case. Moreover, we present new techniques that improve the efficiency of Algorithmic Debugging, allowing the debugger to speed-up the questions asked to the user and to reduce the number of questions performed regardless of the used strategy.

Resumen

La *Depuración Algorítmica* (también conocida como *Depuración Declarativa*) es una técnica de depuración semi-automática que permite al programador aislar el código donde se encuentra un bug sin necesidad de ver el código fuente. Para ello, el programador contesta las preguntas que realiza el depurador hasta que éste localiza el error mostrándoselo al usuario. Las preguntas realizadas son escogidas por el depurador utilizando una estrategia, siendo la reducción de la cantidad de preguntas realizadas al usuario uno de los principales objetivos de esta técnica de depuración. En la práctica, la estrategia que hasta ahora realizaba menos preguntas es *Divide & Query* (*D&Q*). Esta estrategia ha sido considerada óptima en el peor caso desde su definición hace tres décadas. En este trabajo demostramos que D&Q no es óptima y mostramos dos nuevas estrategias que son mejores. Una de ellas es una nueva versión de D&Q donde mejoramos su comportamiento en algunas situaciones. Mientras que la otra es la primera versión de una nueva estrategia que sí es óptima en el peor caso. Además, presentamos nuevas técnicas que mejoran la eficiencia de la Depuración Algorítmica, permitiendo acelerar las preguntas que se hacen al usuario y que reducen la cantidad de preguntas realizadas independientemente de la estrategia utilizada.

Contents

Chapter 1

Introduction

1.1 Algorithmic Debugging

Debugging is one of the most important but less automated (and, thus, time-consuming) tasks in the software development process. It is necessary in all paradigms and programming languages both during the development and during the maintenance of software systems. In Shapiro's words [25]:

> "*It is evident that a computer can neither construct nor debug a program without being told (...) what problem the program is supposed to solve (...). No matter what language we use to convey this information, we are bound to make mistakes. Not because we are sloppy and undisciplined, as advocates of some program development methodologies may say, but because of a much more fundamental reason: we cannot know, at any finite point in time, all the consequences of our current assumptions.*"

Unfortunately, the efforts of the scientific community in producing usable and scalable debuggers has been historically low. The programmer is often forced to manually explore the code or iterate over it (using, e.g., breakpoints that allow us to execute the program step by step and inspect computations (manually) at a given point), and this process usually requires a deep understanding of the source code to find the bug.

Algorithmic debugging [25] is a semi-automatic debugging technique that has been extended to practically all paradigms [27], and many techniques [18, 4, 26, 8, 6] have been defined to improve the original proposal [25]. Recent research has produced new advances to increase the scalability of the technique producing new scalable and mature debuggers. The technique produces a dialogue between the debugger and the programmer to find the bugs. Essentially, it relies on the programmer having an *intended interpretation* of the program. In other words, some computations of the program are correct and others are wrong with respect to the programmer's intended semantics. Therefore, declarative debuggers compare the results of sub-computations with what the programmer intended. By asking the programmer questions or using a formal specification the system can identify precisely the location of a program's bug.

We show two different *Algorithmic Debugging Sessions* for two different sorting algorithm (YES and NO answers are provided by the programmer). Note that, to debug the programs, the programmer only has to answer questions. It is not even necessary to see the source code.

Example 1.1.1

Consider this Haskell program inspired in a similar example by [9] that tries to sort a list using *Insertion Sort*:

```
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))

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`.

Example 1.1.2

Consider this Haskell program that tries to sort a list using *mergeSort*:

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

merge [] list = list
merge list [] = list
merge (x:xs) (y:ys) | x <= y    = x : merge xs (y:ys)
                    | otherwise = merge (x:xs) ys

mergeSort [] = []
mergeSort [x] = [x]
mergeSort list = merge (mergeSort left) (mergeSort right)
          where middle = (div (length list) 2)
```

```
                left = take middle list
                right = drop middle list

Starting Debugging Session...
(1)  mergeSort [2,1] = [2]? NO
(2)  merge [2] [1] = [2]? NO
(3)  merge [2] [] = [2]? YES

Bug found in rule:
merge (x:xs) (y:ys) | otherwise = merge (x:xs) ys
```

The debugger points out the part of the code that contains the bug. In this case `otherwise` = `merge`... should be `otherwise` = `y` : `merge`....

Traditionally, declarative debugging consists of two sequential phases: During the first phase a data structure, the *Execution Tree* (ET) [22], is built, which is an intermediate data structure that represents the execution of the program including all subcomputations; while in the second phase this structure is traversed by using a *strategy* and asking questions to an external oracle until the bug is found.

For instance, the ETs of the programs in Example 1.1.1 and 1.1.2 are depicted respectively in Figures 1.1 and 1.1.

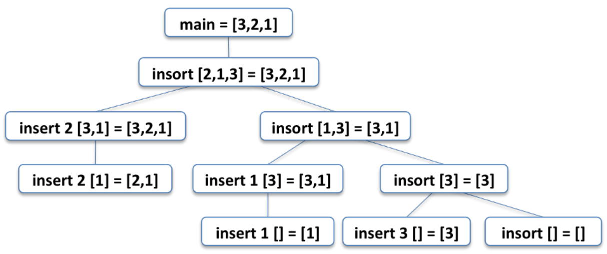


Figure 1.1: ET of the program in Example 1.1.1

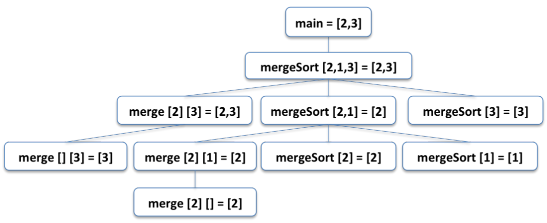


Figure 1.2: ET of the program in Example 1.1.2

The ET contains nodes that represent subcomputations of the program. Therefore, the information of the ET's nodes is different for each programming paradigm (e.g., functions, methods, procedures, etc.), but the construction of the ET is very similar for all of them.

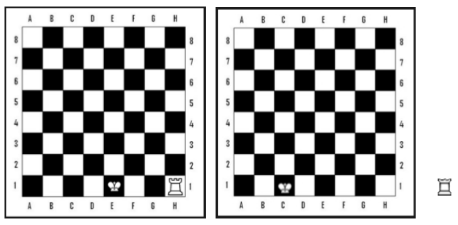
For instance, in the object-oriented paradigm, an ET is constructed as follows: Each node of the ET is associated with a method execution. It contains all the information needed to decide whether the method execution produced a correct result. This information includes the call to the method with its parameters and the result, and the values of all the attributes that are in the scope of this method, before and after the execution (observe, e.g., that exception objects thrown are also in the scope). This information allows the programmer to know whether all the effects of the method execution correspond to her intended semantics. The root node of the ET is the initial method execution of the program (e.g., *main*). For each node n with associated method m, and for each method execution m' done by m, a new node associated with m' is recursively added to the ET as a child of n.

```
public class Chess {
  public static void main(String[] args) {
    Chess p = new Chess();
    Position tower = new Position();
    Position king = new Position();
    king.locate(5,1);
    tower.locate(8,1);
    p.castling(tower,king);
  }
  void castling(Position t,Position k) {
    if (t.x!=8){
        for(int i=1; i<=2; i++) {t.left();}
        for(int i=1; i<=2; i++) {k.right();}
    } else{
        for(int i=1; i<=3; i++) {t.right();}
        for(int i=1; i<=2; i++) {k.left();}
    }
  }
}
class Position {
    int x, y;
    void locate(int a, int b) {x=a; y=b;}
    void up() {y=y+1;}
    void down() {y=y-1;}
    void right() {x=x+1;}
    void left() {x=x-1;}
 }
```

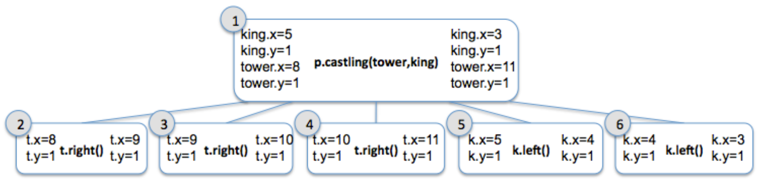
Figure 1.3: Chess program

Example 1.1.3

Consider the Java program in Figure 1.1. This program has a bug, and thus it wrongly simulates some movements on a chessboard. The call of the method `p.castling(tower,king)` produces the (wrong) movement shown in the boards of the following figures.



The next figure depicts the portion of the ET associated with the execution of the method `p.castling(tower,king)`.



1.2 Preliminary definitions

In this section we introduce some notation and formalize the notion of execution tree. For convenience, we consider ETs as labeled trees. We need to formally define the notions of context and method execution before we provide a definition of ET.

Definition 1.2.1 (Context) *Let $\mathcal{P}$ be a program, and m a method in $\mathcal{P}$. The* context *of m is $\{(a, v) \mid a$ is an attribute in the scope of m and v is the value of $a\}$.*

Roughly, the context of a method is composed of all the variables of the program that *could* be affected by the execution of this method. Clearly, these variables can be other objects that in turn contain other variables. In a realistic program, each node contains several data structures that could change during the execution. All this information (before and after the execution) should be visualized together with the call to the method so that the programmer can decide whether it is correct.

Definition 1.2.2 (Method Execution) *Let $\mathcal{P}$ be a program and X an execution of $\mathcal{P}$. Then, each* method execution *done in X is represented by a triple $\mathcal{E} = (b, m, a)$ where m represents the call to the method with its parameters and the returned value, b is the context of the method in m before its execution, and a is the context of the method in m after its execution.*

Thanks to the declarative properties of declarative debugging, we can ignore the operational details of an execution. From the point of view of the debugger an execution is a finite tree of method executions. This can be modeled with the following grammar:

$$T = (b, m[L], a) \qquad L = \epsilon \qquad L = TL$$

where the terminal m is a method of the program and b and a represent the context before and after the execution of the method. For instance, the call `p.castling(tower,king)` in Example 1.1.3 can be represented by the tree:

$$(b_1, \texttt{p.castling(tower,king)}[(b_2, \texttt{t.right()}[], a_1), (a_1, \texttt{t.right()}[], a_2),$$
$$(a_2, \texttt{t.right()}[], a_3), (b_3, \texttt{k.left}[], a_4), (a_4, \texttt{k.left}[], a_5)], a_6)$$

With this tree, we can construct the ET in Example 1.1.3. Roughly speaking, an ET is a tree whose nodes contain method executions and the parent-child relation is defined by the tree produced by the grammar. Formally,

Definition 1.2.3 (Execution Tree) *Given a program $\mathcal{P}$ with a set of method definitions M, and a method execution $\mathcal{E}$, the* execution tree *(ET) of $\mathcal{P}$ w.r.t. $\mathcal{E}$ is a tree $t = (V, E)$ where $\forall v \in V$, v is a method execution, and*

- *The root of the ET is $\mathcal{E}$.*
- *For each method execution $\mathcal{E}_1 = (b, m, a) \in V$, we have a child method execution $\mathcal{E}_2 = (b', m', a') \in V$; $m, m' \in M$ (i.e. $(\mathcal{E}_1 \to \mathcal{E}_2) \in E$), iff*
    1. *during the execution of $\mathcal{E}_1$, $\mathcal{E}_2$ is executed, and*
    2. *the method execution $\mathcal{E}_2$ is started from the definition of m (i.e., m calls m').*

Note that V represents the nodes of the tree and E represents the edges. If the context before and after the execution of the method is not important, we use N to represent the nodes of the tree, so we refer a tree as $ET = (V, E)$ or $ET = (N, E)$ depending on the situation. Note also that we use $(v \to v')$ to denote a directed edge from v to v'.

From now on, we assume that there exists an intended semantics $\mathcal{I}$ of the program being debugged. It corresponds to the model the programmer had in mind while writing the program, and it contains, for each method m and each context b of m before its execution, the expected context a after its execution, that is, $(b, m, a) \in \mathcal{I}$. Moreover, given this atomic information, we are able to deduce judgments of the form $(b, m_1; \ldots; m_n, a)$ with the inference rule Tr, that defines the transitivity for the composition of methods

$$\frac{\overline{(b, m_1, a')} \quad \overline{(a', m_2; \ldots; m_n, a)}}{(b, m_1; \ldots; m_n, a)} \ \mathsf{Tr} \text{ if } n > 1$$

and we say that $\mathcal{I} \models (b, m_1; \ldots; m_n, a)$. Using this intended semantics we can formally define the correctness of method executions:

DEFINITION 1.2.4 (CORRECTNESS OF METHOD EXECUTIONS) *Given a method execution $\mathcal{E}$ and the intended semantics of the program $\mathcal{I}$, we say that $\mathcal{E}$ is* correct *if $\mathcal{E} \in \mathcal{I}$ or $\mathcal{I} \models \mathcal{E}$ and* wrong *otherwise.*

Once the ET is built, in the second phase the debugger uses a strategy to traverse the ET asking the programmer about the correctness of the information stored in each node. If the method execution of a node is wrong, the answer is NO. Otherwise, the answer is YES. Using the answers, the debugger identifies a buggy node (a buggy node is associated with the buggy source code of the program).

DEFINITION 1.2.5 (BUGGY NODE) *Given an ET $t = (V, E)$, a buggy node of t is a node $v \in V$ such that (i) the method execution of v is wrong and (ii) $\forall v' \in V$, $(v \to v') \in E$, v' is correct.*

According to Definition 1.2.5, when all the children of a node with a wrong computation (if any) are correct, the node becomes buggy and the debugger locates the bug in the part of the program associated with this node [23]. A buggy node detects a *buggy method*, which informally stands for methods that return an incorrect context even though all the methods executions performed by them are correct.

LEMMA 1.2.6 (BUGGY METHOD) *Given an ET $t = (V, E)$, and a buggy node $v \in V$ in t with $v = (b, m, a)$, then m contains a bug.*

During the debugging session the user answers questions generated by the debugger. These answers define the state of the nodes, and we call this tree *Marked Execution Tree* (MET), 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 1.2.7 (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 $M : N \to S$ is a marking total function that assigns to all the nodes in N a value in the domain $S = \{Wrong, Undefined\}$.*

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 [12]. For simplicity, we assume here that the user only answers YES or NO.

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*. 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 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)              $N = \{n \in N \mid (\text{node} \rightarrow n) \in E^*\}$
(8)      else $N = N \backslash \{n \in N \mid (\text{node} \rightarrow n) \in E^*\}$
(9)  while ($\exists n \in N, M(n) = Undefined$)
(10) return buggyNode

end
```

1.3 Strategies for algorithmic debugging

During the second phase of Algorithmic Debugging a buggy node is searched asking questions to the user. These questions are related to the nodes of the ET, and the selection of the node to be asked is done by a strategy.

Due to the fact that questions are asked in a logical order (i.e., consecutive questions refer to related parts of the computation), *Top-Down search* is the strategy that has been traditionally used (see, e.g., [4, 5, 14]) to measure the performance of different debugging tools and methods. It basically consists of a top-down (assuming that the root is on top), left-to-right traversal of the ET. When the answer to the question of a node is NO, then the next question is associated with one of its children. When the answer is YES, the next question is associated with one of its siblings. Therefore, the node asked is always a child or a sibling of the previous asked node. Hence, the idea is to follow the path of wrong computations from the root of the tree to the buggy node.

However, selecting always the leftmost child does not take into account the size of the subtrees that can be explored. Binks proposed in [2] a variant of Top-Down search in order to consider this information when selecting a child. This variant is called *Heaviest First* because it always selects the child with the biggest subtree. The objective is to avoid selecting small subtrees that have a lower probability of containing a bug. Another important strategy is *Divide and Query* (D&Q) [25], which always selects the node whose subtree's size is the closest one to half the size of the tree (we have used this strategy during the Examples 1.1.1 and 1.1.2). This strategy asks, in general, fewer questions than Top-Down search because it prunes near half of the tree with every question. However, its performance is strongly dependent on the structure of the ET. If the ET is balanced, this strategy is query-optimal.

In general, regardless of the strategy we use, when we ask for a node: If the answer is YES, this node (and its subtree) is pruned; If the answer is NO the search continues in the subtree rooted at this node.

There are many other strategies: variants of Top-Down search [19, 8], variants of D&Q [11], and others [18, 29]. A comparison of strategies can be found in [27]. In general, all of them are strongly influenced by the structure of the ET.

Example 1.3.1

A declarative debugging session for the ET in Example 1.1.3 using D&Q is the following (`YES` and `NO` answers are provided by the programmer):

```
Starting Debugging Session...
(2)  t.x=8,  t.y=1     t.right()    t.x=9,  t.y=1    ? YES
(3)  t.x=9,  t.y=1     t.right()    t.x=10, t.y=1    ? YES
(4)  t.x=10, t.y=1     t.right()    t.x=11, t.y=1    ? YES
(5)  k.x=5,  k.y=1     k.left()     k.x=4,  k.y=1    ? YES
(6)  k.x=4,  k.y=1     k.left()     k.x=3,  k.y=1    ? YES
(1)  king.x=5,  king.y=1      p.castling(tower,king)     king.x=3,   king.y=1    ? NO
     tower.x=8, tower.y=1                                tower.x=11, tower.y=1
Bug found in method: castling(Position t, Position k) of class Chess
```

The debugger points out the buggy method, that contains the bug. In this case, `t.x!=8` should be `t.x==8`.

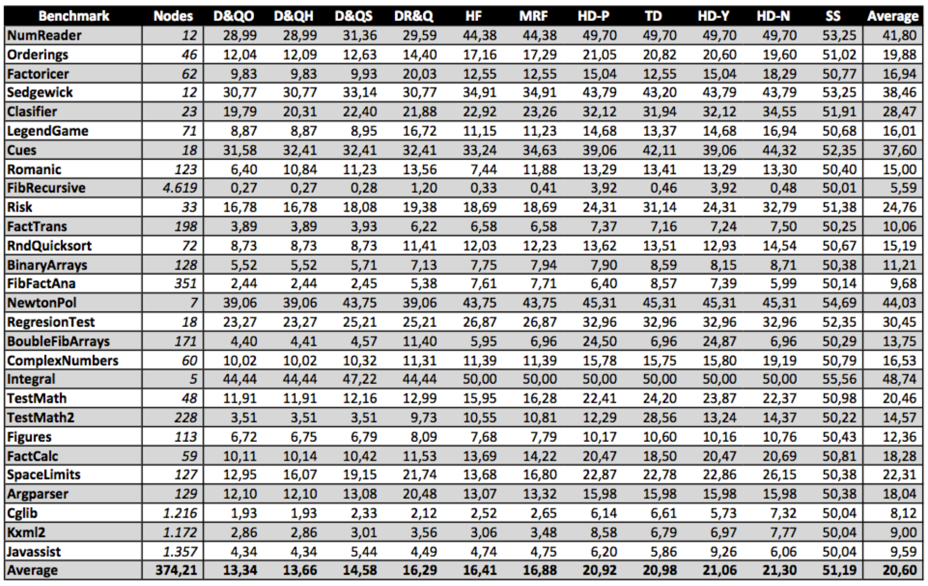
Chapter 2

Optimal Divide & Query

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 has been a lot of research concerning the definition of new strategies trying to minimize the number of questions [27]. 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 [27] 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`).

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 we selected a different node and simulated that it was buggy, thus 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., [7, 8, 12, 24]). In this



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**

Figure 2.1: Performance of algorithmic debugging strategies

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 [27]. The weight of the nodes can also be reassigned dynamically during the debugging session in order to take into account the oracle's answers [8].

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.

2.1 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 [25] and the improved version by Hirunkitti and Hogger [11].

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 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 $\frac{n}{2}$. The original algorithm by Shapiro always selects:

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

Hirunkitti and Hogger noted that this is not enough to divide the MET by 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 $w_{n'} \leqslant \frac{n}{2}$, or
- the lightest node n' whose weight is as close as possible to $\frac{n}{2}$ with $w_{n'} \geqslant \frac{n}{2}$

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

2.1.1 Limitations of current 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) [1]. 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 2.2 (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 2.2 (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 marking of the root. Nevertheless, it is evident that we do not need to ask again for a node that is already

[1] Modern debuggers [12] 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*.

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.

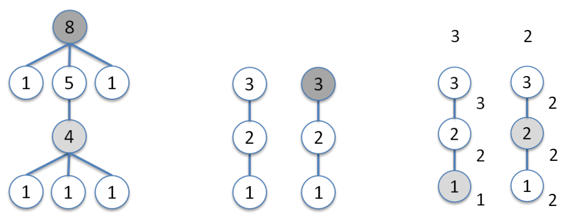


Figure 2.2: Behavior of Divide and Query

In the METs of Figure 2.2 (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 2.2 (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 2.3 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 $\frac{5}{2}$.

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 2.4. 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 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

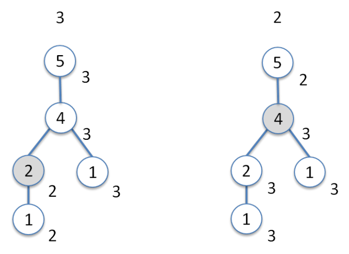


Figure 2.3: Incompleteness of Divide and Query

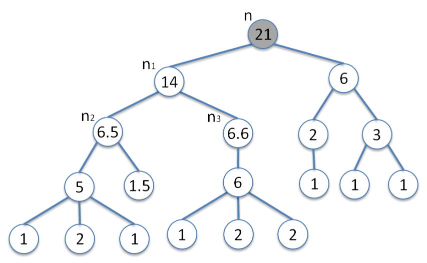


Figure 2.4: MET with decimal individual weights

the MET contains wrong nodes and the nodes of the MET have different individual weights.

2.2 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 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 2.2.1 (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 [15]. 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:

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

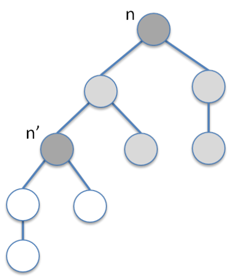


Figure 2.5: Functions Up and Down

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.

2.2.1 Debugging METs where all nodes have the same individual weight $wi \in \mathcal{R}^+$

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

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

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

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

Example 2.2.2

Consider the MET in Figure 2.5. Assuming that the root n is marked as wrong and all nodes have an individual weight of 1, then $\mathit{Sea}(T)$ contains all nodes except n, $\mathit{Up}(n') = 4$ (total weight of the gray nodes), and $\mathit{Down}(n') = 3$ (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:

$$w_n = \mathit{Up}(n') + \mathit{Down}(n') + wi_{n'} \qquad \text{(Equation 1)}$$
$$w_{n'} = \mathit{Down}(n') + wi_{n'} \qquad \text{(Equation 2)}$$

Intuitively, given a node n, what we want to divide by half is the area formed by $\mathit{Up}(n) + \mathit{Down}(n)$. That is, n will not be part of $\mathit{Sea}(T)$ after it has been answered, thus the objective is to make $\mathit{Up}(n)$ equal to $\mathit{Down}(n)$. 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., $|Down(n_1) - Up(n_1)| < |Down(n_2) - Up(n_2)|$). 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 2.2.3 *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 $w_{n_1} > w_{n_2}$, $n_1 \gg n_2$ if and only if $w_n > w_{n_1} + w_{n_2} - wi_n$.*

Theorem 2.2.4 *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 $w_{n_1} > w_{n_2}$, $n_1 \equiv n_2$ if and only if $w_n = w_{n_1} + w_{n_2} - wi_n$.*

Theorem 2.2.3 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 2.2.5 *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 2.2.6 *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 2.2.6 states the objective of optimal D&Q (finding a node n such that $Up(n) = Down(n)$), Theorems 2.2.3, 2.2.4 and 2.2.5 provide a method to approximate this objective (finding a node n such that $|Down(n) - Up(n)|$ is minimum in $Sea(T)$).

An algorithm for Optimal D&Q

Theorems 2.2.3 and 2.2.4 provide equation $w_n \gtrless w_{n_1} + w_{n_2} - wi_n$ 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 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 $n \in N$,
       $\forall n', n'' \in N, wi_{n'} = wi_{n''}$ and $\forall n' \in N, wi_{n'} > 0$
Output: A node $n_{Optimal} \in N$
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 return Best
(6)      Candidate = $n' \mid \forall n''$ with $n', n'' \in$ Children, $w_{n'} \geqslant w_{n''}$
(7)  while ($w_{Candidate} > \frac{w_n}{2}$)
(8)  if ($M(\text{Best}) = Wrong$) then return Candidate
(9)  if ($w_n \geqslant w_{Best} + w_{Candidate} - wi_n$) then return Best
(10)                                     else return Candidate
end
```

Example 2.2.7

Consider the MET in Figure 2.6 where $\forall n \in N, wi_n = 1$ and $M(n) = Undefined$. Observe that Algorithm 2 only needs to apply the equation in

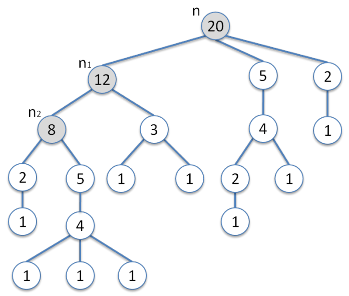


Figure 2.6: Defining a path in a MET to find the optimal node

Theorem 2.2.3 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 $w_n \geqslant w_{n_1} + w_{n_2} - wi_n$ to compare nodes n_1 and n_2. Finally, the algorithm selects n_1.

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 2.2.3, 2.2.4 and 2.2.5. The second point is the key idea of the algorithm and it relies 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: $w_{n_1} > \frac{w_n}{2} \ \wedge \ w_{n_2} \leqslant \frac{w_n}{2}$.
Case 4: $w_{n_1} > w_{n_2} \ \wedge \ w_{n_1} \leqslant \frac{w_n}{2}$.

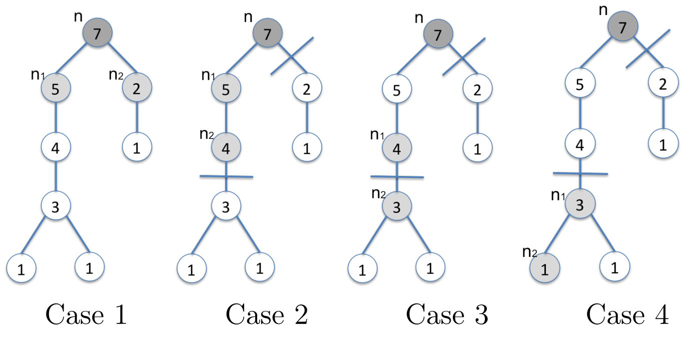


Figure 2.7: Determining the best node in a MET (four possible cases)

We have proven—the individual proofs are part of the proof of Theorem 2.2.8—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 equations of Theorems 2.2.3, 2.2.4 and 2.2.5. 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 2.2.7 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 at node 8 because it is better than all its descendants. And it is also used to determine that nodes 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 2.2.8 (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 2.2.3 in combination with the equation in Theorem 2.2.4 can be used to identify all optimal nodes in the MET. This is implemented in Algorithm 3 that is complete, and thus it returns nodes 2 and 4 in the MET of Figure 2.3 where D&Q can only detect node 2 as optimal.

Algorithm 3 Optimal D&Q (Complete) —SelectNode—

```
Input: A MET $T = (N, E, M)$ whose root is $n \in N$,
       $\forall n', n'' \in N, wi_{n'} = wi_{n''}$ and $\forall n' \in N, wi_{n'} > 0$
Output: A set of nodes $O \subseteq N$
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 return {Best}
(6)      Candidate = $n' \mid \forall n''$ with $n', n'' \in$ Children, $w_{n'} \geqslant w_{n''}$
(7)  while ($w_{Candidate} > \frac{w_n}{2}$)
(8)  Candidates = $\{n' \mid \forall n''$ with $n', n'' \in$ Children, $w_{n'} \geqslant w_{n''}\}$
(9)  if ($M(\text{Best}) = Wrong$) then return Candidates
(10) if ($w_n > w_{Best} + w_{Candidate} - wi_n$) then return {Best}
(11) if ($w_n = w_{Best} + w_{Candidate} - wi_n$) then return {Best} $\cup$ Candidates
(12)                                               else return Candidates
end
```

2.2.2 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 2.4, 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 2.4, it would select node n_3. For this reason, in this section we introduce Algorithm 4, 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 4. Theorem 2.2.9 ensures the finiteness and correctness of Algorithm 4.

Algorithm 4 Optimal D&Q General —SelectNode—

```
Input: A MET $T = (N, E, M)$ whose root is $n \in N$ and $\forall n' \in N, wi_{n'} \geqslant 0$

Output: A node $n_{Optimal} \in N$
Preconditions: $\exists n' \in N$, $M(n') = Undefined$

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

Theorem 2.2.9 (Correctness) *Let $T = (N, E, M)$ be a MET where $\forall n \in N, wi_n \geqslant 0$, then the execution of Algorithm 4 with T as input always terminates producing as output a node $n \in Sea(T)$ such that $\nexists n' \in Sea(T) \mid n' \gg n$.*

2.2.3 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 4 returns a single optimal node, but it is not able to find all optimal nodes.

Given a node (say n), the difference between having an individual weight of zero, wi_n, and having a (total) weight of zero, w_n, 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* [16]. 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 4 to ensure completeness. This refined version is Algorithm 5.

Algorithm 5 Optimal D&Q General (Complete) —SelectNode—

```
Input: A MET $T = (N, E, M)$ whose root is $n \in N$ and $\forall n' \in N, wi_{n'} > 0$

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 return {Best}
(6)      Candidate = $n' \mid \forall n''$ with $n', n'' \in$ Children, $w_{n'} \geqslant w_{n''}$
(7)  while ($w_{Candidate} - \frac{wi_{Candidate}}{2} > \frac{w_n}{2}$)
(8)  Candidates = $\{n' \mid \forall n''$ with $n', n'' \in \text{Children}, w_{n'} - \frac{wi_{n'}}{2} \geqslant w_{n''} - \frac{wi_{n''}}{2}\}$

(9)  Candidate = $n' \in$ Candidates
(10) if ($M$(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
```

2.3 Proofs of technical results

In this section, for the sake of clarity, we use u_n and d_n instead of $Up(n)$ and $Down(n)$ respectively. Moreover, we distinguish between two kinds of METs to prove the theorems of sections 2.2.1 and 2.2.2 respectively.

Definition 2.3.1 (Uniform MET) *A* uniform MET $T = (N, E, M)$ *is a MET, where* $\forall n, n' \in N, wi_n = wi_{n'}$ *and* $\forall n \in N, wi_n > 0$.

Definition 2.3.2 (Variable MET) *A* variable MET $T = (N, E, M)$ *is a MET, where* $\forall n \in N, wi_n \geqslant 0$.

2.3.1 Proofs of Theorems 2.2.3, 2.2.4 and 2.2.5

Here, we prove Theorems 2.2.3, 2.2.4 and 2.2.5 that are used in Algorithm 2 to compare nodes of the MET and determine which of them is better. For the proof of Theorem 2.2.3, we need to prove first the following lemma.

Lemma 2.3.3 *Let* $T = (N, E, M)$ *be a uniform MET whose root is* $n \in N$*, and let* $n_1, n_2 \in Sea(T)$ *be two nodes. Then,* $n_1 \gg n_2$ *if and only if* $u_{n_1} * d_{n_1} > u_{n_2} * d_{n_2}$.

Proof. We prove that $u_{n_1} * d_{n_1} > u_{n_2} * d_{n_2}$ implies that $|d_{n_1} - u_{n_1}| < |d_{n_2} - u_{n_2}|$ and vice versa. This can be shown by developing the equation $u_{n_1} * d_{n_1} > u_{n_2} * d_{n_2}$.
Firstly, note that $w_n = \sum\{wi_{n'} \mid n' \in Sea(T)\}$, then by Equation 1 we know that $w_n = u_{n_1} + d_{n_1} + wi_{n_1} = u_{n_2} + d_{n_2} + wi_{n_2}$. Therefore, as $wi_{n_1} = wi_{n_2} = wi_n$ the optimal division of $Sea(T)$ happens when $u_{n_1} = d_{n_1} = \frac{w_n - wi_n}{2}$. For the sake of simplicity in the notation, let $c = \frac{w_n - wi_n}{2}$ and let $h_1 = c - d_{n_1} = u_{n_1} - c$ and $h_2 = c - d_{n_2} = u_{n_2} - c$. Then,

$u_{n_1} * d_{n_1} > u_{n_2} * d_{n_2}$
Therefore, we replace u_{n_1}, d_{n_1}, u_{n_2} and d_{n_2}:
$(c + h_1) * (c - h_1) > (c + h_2) * (c - h_2)$
$c^2 - h_1 * c + h_1 * c - h_1^2 > c^2 - h_2 * c + h_2 * c - h_2^2$
We simplify:
$c^2 - h_1^2 > c^2 - h_2^2$
$-h_1^2 > -h_2^2$
$h_1^2 < h_2^2$
And finally we obtain that:
$|h_1| < |h_2|$

Hence, if the product $u_{n_1} * d_{n_1}$ is greater than $u_{n_2} * d_{n_2}$ then $|h_1| < |h_2|$ and thus, because h_1 and h_2 represent distances to the center, $n_1 \gg n_2$. ■

Theorem 2.2.3. *Given a uniform MET* $T = (N, E, M)$ *whose root is* $n \in N$*, and given two nodes* $n_1, n_2 \in Sea(T)$*, with* $w_{n_1} > w_{n_2}$*,* $n_1 \gg n_2$ *if and only if* $w_n > w_{n_1} + w_{n_2} - wi_n$.

Proof. By Lemma 2.3.3 we know that if $u_{n_1} * d_{n_1} > u_{n_2} * d_{n_2}$ then $n_1 \gg n_2$. Thus it is enough to prove that $w_n > w_{n_1} + w_{n_2} - wi_n$ implies $u_{n_1} * d_{n_1} > u_{n_2} * d_{n_2}$ and vice versa when $w_{n_1} > w_{n_2}$.

$w_n > w_{n_1} + w_{n_2} - wi_n$
Adding $wi_n - wi_n$:
$w_n > w_{n_1} + w_{n_2} - 2 * wi_n + wi_n$
We replace w_{n_1}, w_{n_2} by Equation 2:
$w_n > d_{n_1} + d_{n_2} + wi_n$
Adding $wi_n * d - wi_n * d$:
$w_n > d_{n_1} + d_{n_2} + wi_n * d + wi_n - wi_n * d$
$w_n > d_{n_1} + d_{n_2} + wi_n * d + wi_n(1 - d)$
Using $d = \frac{d_{n_1}}{d_{n_1} - d_{n_2}}$ we get:
$w_n > d_{n_1} + d_{n_2} + wi_n \frac{d_{n_1}}{d_{n_1} - d_{n_2}} + wi_n(1 - \frac{d_{n_1}}{d_{n_1} - d_{n_2}})$
$w_n > d_{n_1} + d_{n_2} + wi_n \frac{d_{n_1}}{d_{n_1} - d_{n_2}} + wi_n(\frac{d_{n_1} - d_{n_2}}{d_{n_1} - d_{n_2}} - \frac{d_{n_1}}{d_{n_1} - d_{n_2}})$
$w_n > d_{n_1} + d_{n_2} + wi_n \frac{d_{n_1}}{d_{n_1} - d_{n_2}} + wi_n \frac{-d_{n_2}}{d_{n_1} - d_{n_2}}$
$w_n > d_{n_1} + d_{n_2} + wi_n \frac{d_{n_1}}{d_{n_1} - d_{n_2}} - wi_n \frac{d_{n_2}}{d_{n_1} - d_{n_2}}$
Because $d_{n_1} + d_{n_2} = \frac{d_{n_1}^2 - d_{n_2}^2}{d_{n_1} - d_{n_2}}$ then:
$w_n > \frac{d_{n_1}^2 - d_{n_2}^2}{d_{n_1} - d_{n_2}} + \frac{d_{n_1} * wi_n}{d_{n_1} - d_{n_2}} - \frac{d_{n_2} * wi_n}{d_{n_1} - d_{n_2}}$
Because $w_{n_1} > w_{n_2}$ we know by Equation 2 that $d_{n_1} - d_{n_2} > 0$, thus:
$(d_{n_1} - d_{n_2}) * w_n > d_{n_1}^2 - d_{n_2}^2 + d_{n_1} * wi_n - d_{n_2} * wi_n$
$d_{n_1} * w_n - d_{n_2} * w_n > d_{n_1}^2 - d_{n_2}^2 + d_{n_1} * wi_n - d_{n_2} * wi_n$
$d_{n_1} * w_n - d_{n_1}^2 - d_{n_1} * wi_n > d_{n_2} * w_n - d_{n_2}^2 - d_{n_2} * wi_n$
$d_{n_1} * (w_n - d_{n_1} - wi_n) > d_{n_2} * (w_n - d_{n_2} - wi_n)$
As $wi_n = wi_{n_1} = wi_{n_2}$ we replace $w_n - d_{n_1} - wi_n$, $w_n - d_{n_2} - wi_n$ by Equation 1:
$d_{n_1} * u_{n_1} > d_{n_2} * u_{n_2}$

■

Theorem 2.2.4. *Given a uniform MET $T = (N, E, M)$ whose root is $n \in N$, and given two nodes $n_1, n_2 \in Sea(T)$, with $w_{n_1} > w_{n_2}$, $n_1 \equiv n_2$ if and only if $w_n = w_{n_1} + w_{n_2} - wi_n$.*

Proof. The proof is completely analogous to the proof of Theorem 2.2.3. The only difference is that the equation that is developed should be $w_n = w_{n_1} + w_{n_2} - wi_n$. ■

Theorem 2.2.5. *Let $T = (N, E, M)$ be a uniform MET, and let $n_1, n_2 \in Sea(T)$ be two nodes, if $w_{n_1} = w_{n_2}$ then $n_1 \equiv n_2$.*

Proof. We prove that $w_{n_1} = w_{n_2}$ implies $|d_{n_1} - u_{n_1}| = |d_{n_2} - u_{n_2}|$ and thus $n_1 \equiv n_2$:

$w_{n_1} = w_{n_2}$ — we replace w_{n_1}, w_{n_2} by Equation 2
$d_{n_1} + wi_{n_1} = d_{n_2} + wi_{n_2}$ — using $wi_{n_1} = wi_{n_2}$
$d_{n_1} = d_{n_2}$ — using $w_{n_1} = w_{n_2}$
$w_{n_1} - w_n + d_{n_1} = w_{n_2} - w_n + d_{n_2}$ — replacing w_{n_1}, w_{n_2} by Equation 2 and w_n by Equation 1
$(d_{n_1} + wi_{n_1}) - (u_{n_1} + d_{n_1} + wi_{n_1}) + d_{n_1}$
$= (d_{n_2} + wi_{n_2}) - (u_{n_2} + d_{n_2} + wi_{n_2}) + d_{n_2}$ — we simplify
$d_{n_1} - u_{n_1} = d_{n_2} - u_{n_2}$
$|d_{n_1} - u_{n_1}| = |d_{n_2} - u_{n_2}|$

■

Corollary 2.2.6. *Given a uniform MET $T = (N, E, M)$, and given a node $n \in Sea(T)$, then n optimally divides $Sea(T)$ if and only if $u_n = d_n$.*

Proof. If n optimally divides $Sea(T)$ then the product $u_n * d_n$ is maximum, and there does not exist other node $n' \in Sea(T)$ such that $u_{n'} * d_{n'} > u_n * d_n$. This can be easily shown taking into account that the figure of the product is a parabola whose vertex is the maximum value. Therefore, we can compute the maximum by deriving the product.

For simplicity, let $prod = u_n * d_n$ and $sum = u_n + d_n$. Then, we start by transforming the equation $u_n * d_n$ in such a way that it only depends on one of the factors (e.g., u_n):

$u_n * d_n = prod$
We replace d_n :
$u_n * (sum - u_n) = prod$
$u_n * sum - u_n^2 = prod$
We derive the equation and equate it to zero:
$\frac{d}{du_n}(u_n * sum - u_n^2) = 0$
$sum - 2u_n = 0$
And finally we get the value of u_n in the vertex:
$u_n = \frac{sum}{2}$

Now, we can infer d_n from u_n by simply replacing the value of u_n in the equation $u_n + d_n = sum$:

$\frac{sum}{2} + d_n = sum$
$d_n = sum - \frac{sum}{2}$
$d_n = \frac{sum}{2}$
$d_n = u_n$

■

2.3.2 Proof of Theorem 2.2.8

Theorem 2.2.8 states the correctness of Algorithm 2 used when all nodes have the same individual weight. Firstly, we proof the following auxiliary lemma.

Lemma 2.3.4 *Let $T = (N, E, M)$ be a uniform MET whose root is $n \in N$ and $n_1, n_2 \in Sea(T)$ with $w_{n_1} > w_{n_2}$, if $w_n \geqslant w_{n_1} + w_{n_2}$ then $n_1 \gg n_2$.*

Proof. Firstly, by Theorem 2.2.3 we know that if $w_n > w_{n_1} + w_{n_2} - wi_n$ when $w_{n_1} > w_{n_2}$ then $n_1 \gg n_2$. Therefore, as $wi_n > 0$, if $w_n \geqslant w_{n_1} + w_{n_2}$ then $w_n > w_{n_1} + w_{n_2} - wi_n$ and hence $n_1 \gg n_2$. ■

In order to prove the correctness of Algorithm 2, we also need to prove the four cases presented in Section 2.2.1 that are used in the algorithm:

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: $w_{n_1} > \frac{w_n}{2} \;\wedge\; w_{n_2} \leqslant \frac{w_n}{2}$.
Case 4: $w_{n_1} > w_{n_2} \;\wedge\; w_{n_1} \leqslant \frac{w_n}{2}$.

We prove each case in a separate lemma. In case 1, the following lemma shows that given two brother nodes n_1 and n_2, then the heaviest node is better.

Lemma 2.3.5 *Given a uniform 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 $(n \rightarrow n_1) \in E^*$,$(n_1 \rightarrow n_2), (n_1 \rightarrow n_3) \in E$, $n_2 \gg n_3 \vee n_2 \equiv n_3$ if and only if $w_{n_2} \geqslant w_{n_3}$.*

Proof. We prove first that $w_{n_2} \geqslant w_{n_3}$ implies $n_2 \gg n_3 \vee n_2 \equiv n_3$: Trivially, $w_n \geqslant w_{n_2} + w_{n_3}$ because n_2 and n_3 are children of n_1 and n_1 is descendant of n. Therefore, by Lemma 2.3.4 and Theorem 2.2.5, $n_2 \gg n_3 \vee n_2 \equiv n_3$. Now, we prove that $n_2 \gg n_3 \vee n_2 \equiv n_3$ implies $w_{n_2} \geqslant w_{n_3}$: We prove it by contradiction assuming that $w_{n_2} < w_{n_3}$ when $n_2 \gg n_3 \vee n_2 \equiv n_3$, and proving that when $w_{n_2} < w_{n_3}$ and $n_2 \gg n_3 \vee n_2 \equiv n_3$, neither $w_n > w_{n_2} + w_{n_3} - wi_n$ nor $w_n \leqslant w_{n_2} + w_{n_3} - wi_n$ holds. By Theorem 2.2.3 $w_n > w_{n_2} + w_{n_3} - wi_n$ is false because $n_2 \gg n_3 \vee n_2 \equiv n_3$. Moreover, because n_2 and n_3 are brothers, we know that $w_n \geqslant w_{n_2} + w_{n_3}$, and hence $w_n \leqslant w_{n_2} + w_{n_3} - wi_n$ is also false. ■

In case 2, the following lemma ensures that given two nodes n_1 and n_2 such that $n_1 \rightarrow n_2$, if $w_{n_2} > \frac{w_n}{2}$ then n_2 is better.

Lemma 2.3.6 *Given a uniform MET $T = (N, E, M)$ whose root is $n \in N$, and given two nodes $n_1, n_2 \in Sea(T)$, with $(n_1 \rightarrow n_2) \in E$, if $w_{n_2} > \frac{w_n}{2}$ then $n_2 \gg n_1$.*

Proof. We prove the lemma by contradiction assuming that $n_1 \gg n_2$ or $n_1 \equiv n_2$. First, we know that $w_{n_2} = \frac{w_n}{2} + inc_{n_2}$ with $inc_{n_2} > 0$. And we know that $w_{n_1} = \frac{w_n}{2} + inc_{n_2} + wi_n + inc_{n_1}$ with $inc_{n_1} \geqslant 0$, where inc_{n_1} represents the weight of the possible brothers of n_2. By Theorems 2.2.3 and 2.2.4 we know that $w_n \geqslant w_{n_1} + w_{n_2} - wi_n$ when $w_{n_1} > w_{n_2}$ implies $n_1 \gg n_2 \vee n_1 \equiv n_2$.

$$\begin{array}{ll}
w_n \geqslant w_{n_1} + w_{n_2} - wi_n & \text{We replace } w_{n_1}, w_{n_2} \\
w_n \geqslant (\frac{w_n}{2} + inc_{n_2} + wi_n + inc_{n_1}) + (\frac{w_n}{2} + inc_{n_2}) - wi_n & \text{we simplify} \\
w_n \geqslant \frac{w_n}{2} + inc_{n_2} + inc_{n_1} + \frac{w_n}{2} + inc_{n_2} & \\
w_n \geqslant \frac{w_n}{2} + \frac{w_n}{2} + 2 * inc_{n_2} + inc_{n_1} & \\
w_n \geqslant w_n + 2 * inc_{n_2} + inc_{n_1} & \\
0 \geqslant 2 * inc_{n_2} + inc_{n_1} &
\end{array}$$

But, this is a contradiction with $inc_{n_2} > 0$. Hence, $n_2 \gg n_1$. ■

In case 4, the following lemma ensures that given two nodes whose weight is smaller than $\frac{w_n}{2}$ then the heaviest node is better.

Lemma 2.3.7 *Given a uniform MET $T = (N, E, M)$ whose root is $n \in N$, and two nodes $n_1, n_2 \in Sea(T)$, where $\frac{w_n}{2} \geqslant w_{n_1} > w_{n_2}$ then $n_1 \gg n_2$.*

Proof. We can assume that $w_{n_1} = \frac{w_n}{2} - dec_{n_1}$ and $w_{n_2} = \frac{w_n}{2} - dec_{n_2}$ where $dec_{n_2} > dec_{n_1} \geqslant 0$. Moreover, we know that $w_{n_1} + w_{n_2} = \frac{w_n}{2} - dec_{n_1} + \frac{w_n}{2} - dec_{n_2}$ and thus $w_{n_1} + w_{n_2} = w_n - dec_{n_1} - dec_{n_2}$. Therefore, because $dec_{n_2} > dec_{n_1} \geqslant 0$, we deduce that $w_n > w_{n_1} + w_{n_2}$. And as $w_{n_1} > w_{n_2}$ then, by Lemma 2.3.4, $n_1 \gg n_2$. ■

If two nodes n_1 and n_2 are brothers and n_1 is better than n_2 then n_1 is better than any descendant of n_2. The following lemma proves this property that is complementary to Lemma 2.3.5 for case 1.

Lemma 2.3.8 *Given a uniform MET $T = (N, E, M)$ whose root is $n \in N$ and 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 $n_2 \gg n_3 \vee n_2 \equiv n_3$ then $n_2 \gg n_4$.*

Proof. First, n_2 and n_3 are brothers and $n_2 \gg n_3 \vee n_2 \equiv n_3$ then, by Lemma 2.3.5, we know that $w_{n_2} \geqslant w_{n_3}$. We distinguish two cases $w_{n_2} > \frac{w_n}{2}$ and $\frac{w_n}{2} \geqslant w_{n_2}$.
If $\frac{w_n}{2} \geqslant w_{n_2}$ then $\frac{w_n}{2} \geqslant w_{n_3}$ and by Lemma 2.3.7 $n_3 \gg n_4$.
If $w_{n_2} > \frac{w_n}{2}$ then we only have to demonstrate that $\frac{w_n}{2} > w_{n_3}$ and then (as before) by Lemma 2.3.7 $n_3 \gg n_4$.
This can be easily proved having into account that $w_n \geqslant w_{n_2} + w_{n_3}$ because n_2 and n_3 are children of n_1 and n_1 is descendant of n, and that $w_{n_2} = \frac{w_n}{2} + inc_{n_2}$ with $inc_{n_2} > 0$.

$$\begin{array}{ll}
w_n \geqslant w_{n_2} + w_{n_3} & \text{we replace } w_{n_2} \\
w_n \geqslant (\frac{w_n}{2} + inc_{n_2}) + w_{n_3} & \\
w_n - \frac{w_n}{2} \geqslant inc_{n_2} + w_{n_3} & \\
\frac{w_n}{2} \geqslant inc_{n_2} + w_{n_3} & \text{as } inc_{n_2} > 0 \\
\frac{w_n}{2} > w_{n_3} &
\end{array}$$

Therefore as $n_2 \gg n_3 \vee n_2 \equiv n_3$ and $n_3 \gg n_4$ then $n_2 \gg n_4$. ∎

The previous lemmas allow Algorithm 2 to find a path between the root node and an optimal node. The correctness of this algorithm is proved by the following theorem.

Theorem 2.2.8. *Let $T = (N, E, M)$ be a uniform 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: $w_{Candidate}$ strictly decreases in each iteration. Therefore, because N is finite, $w_{Candidate}$ will eventually become smaller or equal to $\frac{w_n}{2}$ and 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.

(Base Case) In the base case, only one iteration is executed. If the condition in Line (5) is satisfied then the root is marked as undefined and it is trivially the optimal node. This optimal node is returned in Line (5). Otherwise, Lines (4) and (6) select the heaviest child of the root, the loop terminates and Lines (9) or (10) return the optimal node.

Note that the root node—when it is marked as *Wrong*—can only be selected in the first iteration. But even in this case, this node is never selected because the root node must have at least one child marked as *Undefined*. Thus Line (5) is not satisfied and Line (6) selects this node. If the condition of the loop is not satisfied, then Line (8) returns the roots' child.

(Induction Hypothesis) We assume as the induction hypothesis that after i iterations, the algorithm has a candidate node $Best \in Sea(T)$ such that $\forall n' \in Sea(T), (Best \to n') \notin E^*, Best \gg n'$.

(Inductive Case) We now prove that the iteration $i + 1$ of the algorithm will select a new candidate node *Candidate* such that $Candidate \gg Best$, 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 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 $w_{Best} > \frac{w_n}{2}$ being n the root of T because in the iteration i the loop did not terminate or because *Best* is the root. Moreover, according to Lines (4) and (6), we know that *Candidate* is the heaviest child of *Best*. We have two possibilities:

- $w_{Candidate} > \frac{w_n}{2}$: In this case the loop does not terminate and $\forall n' \in Sea(T)$, $(Candidate \to n') \notin E^*, Candidate \gg n'$. Firstly, by Lemma 2.3.6 we know that $Candidate \gg Best$, and thus, by the induction hypothesis we know that $\forall n' \in Sea(T), (Best \rightarrow n') \notin E^*, Candidate \gg n'$. By Lemma 2.3.5 $Candidate \gg n' \vee Candidate \equiv n'$ being n' a brother of $Candidate$. But as we know that $w_{Candidate} > \frac{w_n}{2}$ then $Candidate \not\equiv n'$. Moreover, by Lemma 2.3.8 we can ensure that $Candidate \gg n'$ being n' a descendant of a *candidate*'s brother.

- $w_{Candidate} \leqslant \frac{w_n}{2}$: In this case the loop terminates (Line (7)) and by Lemma 2.3.5 we know that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a brother of $Candidate$. Moreover, by Lemma 2.3.8 we can ensure that $Candidate \gg n'$ being n' a descendant of a *candidate*'s brother. Then equation $(w_n \geqslant w_{Best} + w_{Candidate} - wi_n)$ is applied in Line (9) to select an optimal node. Theorems 2.2.3 and 2.2.4 ensures that the node selected is an optimal node because, according to Lemma 2.3.7, for all descendant n' of $Candidate$, $Candidate \gg n'$.

■

2.3.3 Proof of Theorem 2.2.9

Theorem 2.2.9 states the correctness of Algorithm 4 used in the general case when nodes can have different individual weights. 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 2.3.9 *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 $d_n \geqslant u_n$ in both nodes and $n_1 \rightarrow n_2$ then $n_2 \gg n_1 \vee n_2 \equiv n_1$.

Lemma 2.3.10 *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 $d_{n_2} \geqslant u_{n_2}$ then $n_2 \gg n_1 \vee n_2 \equiv n_1$.*

Proof. We prove that $|d_{n_2} - u_{n_2}| \leqslant |d_{n_1} - u_{n_1}|$ 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 $inc \geqslant 0$, where inc represents the weight of the possible brothers of n_2.
$|d_{n_2} - u_{n_2}| \leqslant |d_{n_1} - u_{n_1}|$
As we know that $d_n \geqslant u_n$ in both nodes:
$d_{n_2} - u_{n_2} \leqslant d_{n_1} - u_{n_1}$
We replace d_{n_1} and u_{n_1}:
$d_{n_2} - u_{n_2} \leqslant (d_{n_2} + wi_{n_2} + inc) - (u_{n_2} - wi_{n_1} - inc)$
$d_{n_2} - u_{n_2} \leqslant d_{n_2} - u_{n_2} + wi_{n_1} + wi_{n_2} + 2inc$
$0 \leqslant wi_{n_1} + wi_{n_2} + 2inc$
Hence, because $wi_{n_1}, wi_{n_2}, inc \geqslant 0$ then $|d_{n_2} - u_{n_2}| \leqslant |d_{n_1} - u_{n_1}|$ 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 $d_n \leqslant u_n$ in both nodes and $n_1 \rightarrow n_2$ then $n_1 \gg n_2 \vee n_1 \equiv n_2$.

Lemma 2.3.11 *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 $d_{n_1} \leqslant u_{n_1}$ then $n_1 \gg n_2 \vee n_1 \equiv n_2$.*

Proof. We prove that $|d_{n_1} - u_{n_1}| \leqslant |d_{n_2} - u_{n_2}|$ 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 $inc \geqslant 0$, where inc represents the weight of the possible brothers of n_2.

$|d_{n_1} - u_{n_1}| \leqslant |d_{n_2} - u_{n_2}|$
As we know that $u_n \geqslant d_n$ in both nodes:
$u_{n_1} - d_{n_1} \leqslant u_{n_2} - d_{n_2}$
We replace d_{n_2} and u_{n_2}:
$u_{n_1} - d_{n_1} \leqslant (u_{n_1} + wi_{n_1} + inc) - (d_{n_1} - wi_{n_2} - inc)$
$u_{n_1} - d_{n_1} \leqslant u_{n_1} - d_{n_1} + wi_{n_1} + wi_{n_2} + 2inc$
$0 \leqslant wi_{n_1} + wi_{n_2} + 2inc$
Hence, because $wi_{n_1}, wi_{n_2}, inc \geqslant 0$ then $|d_{n_1} - u_{n_1}| \leqslant |d_{n_2} - u_{n_2}|$ 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 $d_{n_1} \geqslant u_{n_1}$ then $d_{n_2} \leqslant u_{n_2}$.

Lemma 2.3.12 *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 $d_{n_2} \geqslant u_{n_2}$ then $d_{n_3} \leqslant u_{n_3}$.*

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

If two nodes n_1 and n_2 are brothers and $d_{n_1} \geqslant u_{n_1}$ then $n_1 \gg n_2 \vee n_1 \equiv n_2$. The following lemma proves this property.

Lemma 2.3.13 *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 $d_{n_2} \geqslant u_{n_2}$ then $n_2 \gg n_3 \vee n_2 \equiv n_3$.*

Proof. We prove that $|d_{n_2} - u_{n_2}| \leqslant |d_{n_3} - u_{n_3}|$ holds. First, as n_2 and n_3 are brothers we know that $w_n \geqslant d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3}$, then $w_n = d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3} + inc$ with $inc \geqslant 0$.
$|d_{n_2} - u_{n_2}| \leqslant |d_{n_3} - u_{n_3}|$
As $d_{n_2} \geqslant u_{n_2}$ by Lemma 2.3.12 we know that $u_{n_3} \geqslant d_{n_3}$:
$d_{n_2} - u_{n_2} \leqslant u_{n_3} - d_{n_3}$
We replace u_{n_2} and u_{n_3} using Equation 1:
$d_{n_2} - (w_n - d_{n_2} - wi_{n_2}) \leqslant (w_n - d_{n_3} - wi_{n_3}) - d_{n_3}$
$-w_n + 2d_{n_2} + wi_{n_2} \leqslant w_n - 2d_{n_3} - wi_{n_3}$
$-2w_n \leqslant -2d_{n_2} - 2d_{n_3} - wi_{n_2} - wi_{n_3}$
$2w_n \geqslant 2d_{n_2} + 2d_{n_3} + wi_{n_2} + wi_{n_3}$
$w_n \geqslant d_{n_2} + d_{n_3} + \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
We replace w_n:
$d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3} + inc \geqslant d_{n_2} + d_{n_3} + \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
$wi_{n_2} + wi_{n_3} + inc \geqslant \frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2}$
$\frac{wi_{n_2}}{2} + \frac{wi_{n_3}}{2} + inc \geqslant 0$
Hence, because $wi_{n_2}, wi_{n_3}, inc \geqslant 0$ then $|d_{n_2} - u_{n_2}| \leqslant |d_{n_3} - u_{n_3}|$ 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 $w_{n_1} \geqslant w_{n_2}$ and $d_{n_1} \leqslant u_{n_1}$ then $d_{n_2} \leqslant u_{n_2}$.

Lemma 2.3.14 *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 $w_{n_2} \geqslant w_{n_3}$ and $d_{n_2} \leqslant u_{n_2}$ then $d_{n_3} \leqslant u_{n_3}$.*

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

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

Lemma 2.3.15 *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 $u_{n_2} \geqslant d_{n_2}$ and $u_{n_3} \geqslant d_{n_3}$, $n_2 \gg n_3 \vee n_2 \equiv n_3$ if and only if $w_{n_2} - \frac{wi_{n_2}}{2} \geqslant w_{n_3} - \frac{wi_{n_3}}{2}$.*

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

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

∎

If two nodes n_1 and n_2 are brothers and $d_{n_1} \geqslant u_{n_1}$ 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 2.3.16 *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 $d_{n_2} \geqslant u_{n_2}$ 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 $d_{n3} \leqslant u_{n3}$ when $d_{n_2} \geqslant u_{n_2}$ by Lemma 2.3.12 and then by Lemma 2.3.11 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 $d_{n_1} \leqslant u_{n_1} \wedge d_{n_2} \leqslant u_{n_2}$ and $n_2 \rightarrow^{+} n_3$ 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 2.3.17 *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 $(n \rightarrow n_1) \in E^*$, $(n_1 \rightarrow n_2), (n_1 \rightarrow n_3) \in E$, $(n_3 \rightarrow n_4) \in E^+$, if $d_{n_2} \leqslant u_{n_2}$ and $d_{n_3} \leqslant u_{n_3}$ 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 $d_{n_3} \leqslant u_{n_3}$ and then by Lemma 2.3.11 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 $n_2 \rightarrow^{+} n_3$ then $n_1 \gg n_3$. The following lemma proves this property.

Lemma 2.3.18 *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 $(n \rightarrow n_1) \in E^*$, $(n_1 \rightarrow n_2), (n_1 \rightarrow n_3) \in E$, $(n_3 \rightarrow n_4) \in E^+$, 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 $d_{n_3} \geqslant u_{n_3}$ when $n_2 \gg n_3$. First, as n_2 and n_3 are brothers we know that $w_n \geqslant d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3}$, then $w_n = d_{n_2} + d_{n_3} + wi_{n_2} + wi_{n_3} + inc$ with $inc \geqslant 0$. 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 $d_{n_3} \geqslant u_{n_3}$ and n_2 and n_3 are brothers.

$|d_{n_2} - u_{n_2}| < |d_{n_3} - u_{n_3}|$
As $d_{n_3} \geqslant u_{n_3}$ by Lemma 2.3.12 we know that $u_{n_2} \geqslant d_{n_2}$:
$u_{n_2} - d_{n_2} < d_{n_3} - u_{n_3}$
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 $wi_{n_2}, wi_{n_3}, inc \geqslant 0$. 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 $dec \geqslant 0$, where dec represents the weight of the possible brothers of n_4.

$|d_{n_3} - u_{n_3}| > |d_{n_2} - u_{n_2}| \geqslant |d_{n_4} - u_{n_4}|$
We replace d_{n_4} and u_{n_4}:
$|d_{n_3} - u_{n_3}| > |d_{n_2} - u_{n_2}| \geqslant |(d_{n_3} - wi_{n_4} - dec) - (u_{n_3} + wi_{n_3} + dec)|$
$|d_{n_3} - u_{n_3}| > |d_{n_2} - u_{n_2}| \geqslant |d_{n_3} - wi_{n_4} - dec - u_{n_3} - wi_{n_3} - dec|$
$|d_{n_3} - u_{n_3}| > |d_{n_2} - u_{n_2}| \geqslant |d_{n_3} - u_{n_3} - wi_{n_3} - wi_{n_4} - 2dec|$
Note that $d_{n3} - u_{n3}$ must be positive, thus $d_{n3} > u_{n3}$. But this is a contradiction with $d_{n3} < u_{n3}$. ■

The following lemma ensures that given two nodes n_1 and n_2 where $d_{n_1} \geqslant u_{n_1}$ and $d_{n_2} \leqslant u_{n_2}$ and $n_1 \rightarrow n_2$ then if $w_n \geqslant w_{n_1} + w_{n_2} - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$ is satisfied then $n_1 \gg n_2 \vee n_1 \equiv n_2$.

Lemma 2.3.19 *Given a variable MET $T = (N, E, M)$ and given two nodes $n_1, n_2 \in Sea(T)$, with $(n_1 \rightarrow n_2) \in E$, and $d_{n_1} \geqslant u_{n_1}$, and $d_{n_2} \leqslant u_{n_2}$, $n_1 \gg n_2 \vee n_1 \equiv n_2$ if and only if $w_n \geqslant w_{n_1} + w_{n_2} - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$.*

Proof. First, if $|d_{n_1} - u_{n_1}| \leqslant |d_{n_2} - u_{n_2}|$ then $n_1 \gg n_2$ or $n_1 \equiv n_2$. Thus it is enough to prove that $w_n \geqslant w_{n_1} + w_{n_2} - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$ implies $|d_{n_1} - u_{n_1}| \leqslant |d_{n_2} - u_{n_2}|$ and vice versa when $d_{n_1} \geqslant u_{n_1}$ and $d_{n_2} \leqslant u_{n_2}$.
$w_n \geqslant w_{n_1} + w_{n_2} - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$
We replace w_{n_1}, w_{n_2} using Equation 2:
$w_n \geqslant (d_{n_1} + wi_{n_1}) + (d_{n_2} + wi_{n_2}) - \frac{wi_{n_1}}{2} - \frac{wi_{n_2}}{2}$
$w_n \geqslant d_{n_1} + d_{n_2} + \frac{wi_{n_1}}{2} + \frac{wi_{n_2}}{2}$
$2w_n \geqslant 2d_{n_1} + 2d_{n_2} + wi_{n_1} + wi_{n_2}$
$-2w_n \leqslant -2d_{n_1} - 2d_{n_2} - wi_{n_1} - wi_{n_2}$
$-w_n + 2d_{n_1} + wi_{n_1} \leqslant w_n - 2d_{n_2} - wi_{n_2}$
We replace w_n using Equation 1:
$-(d_{n_1} + u_{n_1} + wi_{n_1}) + 2d_{n_1} + wi_{n_1} \leqslant (d_{n_2} + u_{n_2} + wi_{n_2}) - 2d_{n_2} - wi_{n_2}$
$-d_{n_1} - u_{n_1} - wi_{n_1} + 2d_{n_1} + wi_{n_1} \leqslant d_{n_2} + u_{n_2} + wi_{n_2} - 2d_{n_2} - wi_{n_2}$
$-u_{n_1} + d_{n_1} \leqslant -d_{n_2} + u_{n_2}$
$d_{n_1} - u_{n_1} \leqslant u_{n_2} - d_{n_2}$
As $d_{n_1} \geqslant u_{n_1}$ and $d_{n_2} \leqslant u_{n_2}$:
$|d_{n_1} - u_{n_1}| \leqslant |u_{n_2} - d_{n_2}|$
$|d_{n_1} - u_{n_1}| \leqslant |d_{n_2} - u_{n_2}|$
■

Finally, we prove the correctness of Algorithm 4.

Theorem 2.2.9. *Let $T = (N, E, M)$ be a variable MET, then the execution of Algorithm 4 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 proof of correctness is completely analogous to the proof of Theorem 2.2.8. The only difference is the induction hypothesis and the inductive case:

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

(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 $\{n'' \in Sea(T) | (n' \rightarrow n'') \notin E^*\}$. 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 2.3.9. 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 $\forall n' \in Sea(T)$, $(Candidate \rightarrow n') \notin E^*, Candidate \gg n' \vee Candidate \equiv n'$. Firstly, by Lemma 2.3.10 we know that $Candidate \gg Best \vee Candidate \equiv Best$, and thus, by the induction hypothesis we know that $\forall n' \in Sea(T), (Best \rightarrow n') \notin E^*, Candidate \gg n' \vee Candidate \equiv n'$. By Lemma 2.3.13 we know that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a brother of $Candidate$. Moreover, by Lemma 2.3.16 and 2.3.18 we can ensure that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a descendant of a *candidate*'s brother.

- $d_{Candidate} \leqslant u_{Candidate}$: In this case the loop terminates (Line (7)) and we know by Lemma 2.3.14 that $d_{n'} \leqslant u_{n'}$ being n' any brother of $Candidate$. In Line (8) according to Lemma 2.3.15 we select the Candidate such that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a brother of $Candidate$. Moreover, by Lemma 2.3.17 and 2.3.18 we can ensure that $Candidate \gg n' \vee Candidate \equiv n'$ being n' a descendant of a *candidate*'s brother. Then equation ($w_n \geqslant w_{Best} + w_{Candidate} - \frac{wi_{Best}}{2} - \frac{wi_{Candidate}}{2}$) is applied in Line (10) to select an optimal node. Lemma 2.3.19 ensure that the node selected is an optimal node because, according to Lemma 2.3.11, for all descendant n' of $Candidate$, $Candidate \gg n' \vee Candidate \equiv n'$.

■

Chapter 3

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.

3.1 An optimal strategy for algorithmic debugging

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 3.1.1 (Optimal strategy) *Let ϵ be an algorithmic debugging strategy. Given a MET $T = (N, E, M)$, let s_n^ϵ 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 $t_\epsilon = \sum\limits_{n_i \in N} |s_{n_i}^\epsilon|$. 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 of the MET without considering the structure of this subtree. 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 4) 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 3.1.2 (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*

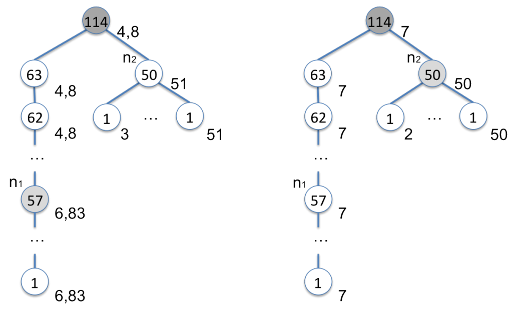


Figure 3.1: MET where D&Q cannot find an optimal node

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 2.2 (left) and assuming that all nodes are marked as undefined:

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

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

Example 3.1.3

Consider the MET in Figure 3.1 (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 3.1.3 showed that D&Q is not an optimal strategy. The question now is whether an optimal strategy exists: is the problem decidable?

Theorem 3.1.4 (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 [12]. 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 3.1.1. The optimal nodes will be the first of the selected sequences.

■

Even though the method described in the proof of Theorem 3.1.4 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.

3.1.1 Selecting the best sequences of questions (sq_n)

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 3.1.4 we first define the notion of valid sequence of questions.

Definition 3.1.5 (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. $\forall n_i, n_j \in sq_n, 1 \leqslant i < j \leqslant m, (n_i \rightarrow n_j) \notin E^*$
2. $N \backslash \{n_j \mid (n_i \rightarrow n_j) \in E^* \wedge n_i \in sq_n\} = \{n\}$

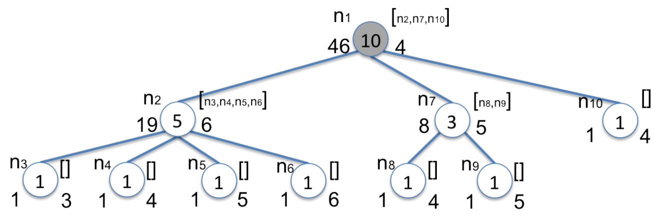
We denote with SQ_n the set of valid sequences of questions of n.

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 3.1.6

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 the root 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[1], 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).

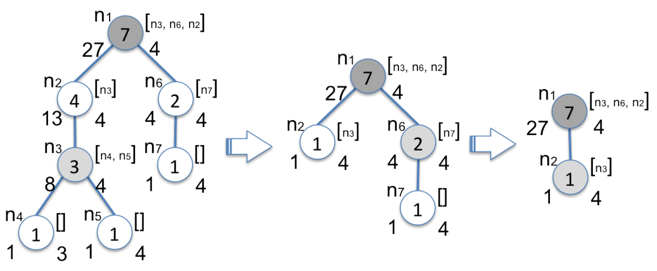
Therefore, to find the optimal nodes, we only have to: (i) Compute Q_n for all nodes in the MET, (ii) add to the MET a fictitious root node, (iii) compute all valid sequences of the root node, (iv) compute the cost associated to each sequence (with Algorithm 6), and (v) select the first node of the sequences with the minimum cost.

[1]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.

Essentially, Algorithm 6 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 4, can be any number greater than or equal to 0.

Example 3.1.7

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 3 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 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 6 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 6 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.

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

Algorithm 6 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
(1) $SQ_n = computeSQ(T, n)$
(2) $sqOptimal = sq_n \in SQ_n \mid \nexists sq'_n \in SQ_n$ .
                      $ComputeQ(T, n, sq_n) > ComputeQ(T, n, sq'_n)$
(3) $QOptimal = ComputeQ(T, n, sqOptimal)$
(4) return $(sqOptimal, QOptimal)$
end

function $ComputeQ(T, n, sq_n)$
begin
(1) $questions = 0$
(2) $indexNode = 0$
(3) $accumNodes = 1$
(4) while $(\{n' | (n \rightarrow n') \in E^*\} \neq \{n\})$
(5)     $node = sq_n[indexNode]$
(6)     $indexNode = indexNode + 1$
(7)     $questions = questions + (Q_{node} + accumNodes * w_{node})$
(8)     $accumNodes = accumNodes + 1$
(9)     $T = AdjustIntermediateNodes(T, n, node)$
    end while
(10) $questions = questions + (accumNodes * wi_n)$
(11) return $questions$
end

function $AdjustIntermediateNodes(T, n, n')$
begin
(1) $O = \{n'' \in N \mid (n' \rightarrow n'') \in E^*\}$
(2) $N = N \backslash O$
(3) $n' = n'' \mid (n'' \rightarrow n') \in E$
(4) while $(n' \neq n)$
(5)     $(\_, Q_{n'}) = ComputeOptimalSequence(T, n')$
(6)     $w_{n'} = w_{n'} - |O|$
(7)     $n' = n'' \mid (n'' \rightarrow n') \in E$
    end while
(8) return $T$
end
```

Chapter 4

An Algorithmic Debugger for JAVA

Algorithmic Debugging has the drawback that the data structure used (the ET) can be huge with realistic programs. This is due to the fact that the ET records all the information needed to answer the questions, and this information includes many data structures that can appear in different points of the execution. That is, it needs to store all the changes produced in the value of all the variables used during the execution of the program. As it can be guessed, if the execution is large (e.g., in the object-oriented paradigm a lot of method invocations are performed), this information can imply the storage of gigabytes of data. Therefore, it can be necessary to use a database to store all this information. In addition, the performance of a debugging session is strongly dependent on the strategy used. It selects nodes to be asked to the user in order to direct the search of the bug. The objective of the strategy is to reduce the number of questions that are asked to the user. The less nodes asked, the better. Moreover, the structure of the ET is crucial to reduce this number of questions.

In this section we introduce two new techniques that allow the debugger to optimize the debugging session, solving the two drawbacks presented above. On the one hand we introduce the *Virtual Execution Tree*, a tree that represents an instant of the Execution Tree while it is being generated, together with the arquitecture of the algorithmic debugger that allows us to start the debugging session almost instantly using this tree. On the other hand we also introduce the *Balancing Execution Tree* technique. This technique adds new nodes to the tree in such a way that the structure of the resulting tree is improved, allowing the strategies to perform less questions to find the buggy node.

4.1 Virtual execution trees

Algorithmic Debugging is very powerful thanks to the ET, because it guarantees that the bug will be found whenever the programmer answers the questions of the debugger. Unfortunately, with realistic programs, the ET can be huge (indeed gigabytes) and this is the main drawback of this debugging technique, because scalability has not been solved yet: If the ET is stored in main memory, the debugger is out of memory with big ETs that do not fit. If, on the other hand, it is stored in a database, debugging becomes a slow task because some questions need to explore a big part of the ET; and also because storing the ET in the database is a time-consuming task.

Modern declarative debuggers allow the programmer to freely explore the ET with graphical user interfaces (GUI) that represent computations [5]. The scalability problem also translates to these features, because showing the whole ET (or even the part of the ET that participates in a subcomputation) is often not possible due to memory overflow reasons.

In some languages, the scalability problem is inherent to the current technology that supports the language and cannot be avoided with more accurate implementations. For instance, in Java, current declarative debuggers (e.g., JavaDD [10] and DDJ [5]) are based on the *Java Platform Debugger Architecture* (JPDA) [20] to generate the ET. This architecture uses the *Java Virtual Machine Tools Interface*, a native interface which helps to inspect the state and to control the execution of applications running in the *Java Virtual Machine* (JVM). Unfortunately, the time scalability problem described before also translates to this architecture, and hence, any debugger implemented with the JPDA will suffer the scalability problems. For instance, we conducted some experiments to measure the time needed by JPDA to produce the ET[1] of a collection of medium/large benchmarks. Results are shown in column `ET time` of Table 4.1.3. Note that, in order to generate the ET, the JVM with JPDA needs some minutes, thus the debugging session would not be able to generate the first question until this time.

In this work we propose a new implementation model that solves the three scalability problems, namely, memory, time and graphical visualization of the ET. Clearly, the ET is the bottleneck of the technique, and sometimes (e.g., in Java) it is not possible to generate it fast. Therefore, our model is based on the following question: *Is it possible to start the debugging session before having computed the whole ET?* The answer is yes.

We propose a framework in which the debugger uses the (incomplete) ET while it is being dynamically generated. Roughly speaking, two processes run in parallel. The first process generates the ET and stores it into both a database (the whole ET) and main memory (a part of the ET). The other

[1]These times corresponds to the execution of the program, the production of the ET and its storage in a database.

process starts the debugging session by only using the part of the ET already generated. Moreover, we use a three-cache memories system to speedup the generation of questions and to guarantee that the debugger is never out of memory (including the GUI components).

4.1.1 A new architecture for Declarative Debuggers

This section presents a new architecture in which declarative debugging is not done in two sequential phases, but in two concurrent phases; that is, while the ET is being generated, the debugger is able to produce questions. This new architecture solves the scalability problems of declarative debugging. In particular, we use a database to store the whole ET, and only a part of it is loaded to main memory.

Moreover, in order to make the algorithms that traverse the ET independent of the database caching problems, we use a three-tier architecture where all the components have access to a *virtual execution tree* (VET). The VET is a data structure which is identical to the ET except that some nodes are missing (not generated yet) or incomplete (they only store a part of the method invocation) Hence, standard strategies can traverse the VET because the structure of the ET is kept.

The VET is produced while running the program. For each method invocation, a new node is added to it with the method parameters and the context before the call. The result and the context after the call are only added to the node when the method execution finishes.

Let us explain the components of the architecture with the diagram in Figure 4.1. Observe that each tier contains a cache that can be seen as a view of the VET. Each cache is used for a different task:

Persistence cache. It is used to store the nodes of the VET in the database. Therefore, when the whole VET is in the database, the persistence cache is not used anymore. Basically, it specifies the maximum number of completed nodes that can be stored in the VET. This bound is called *persistence bound* and it ensures that main memory is never overflowed.

Logic cache. It defines a subset of the VET. This subset contains a limited number of nodes (in the following, *logic bound*), and these nodes are those with the highest probability of being asked, therefore, they should be retrieved from the database. This allows us to load in a single database transaction those nodes that are going to be probably asked and thus reducing the number of accesses to the database.

Presentation cache. It contains the part of the VET that is shown to the user in the GUI. The number of nodes in this cache should be limited to ensure that the GUI is not out of memory or it is too slow. The

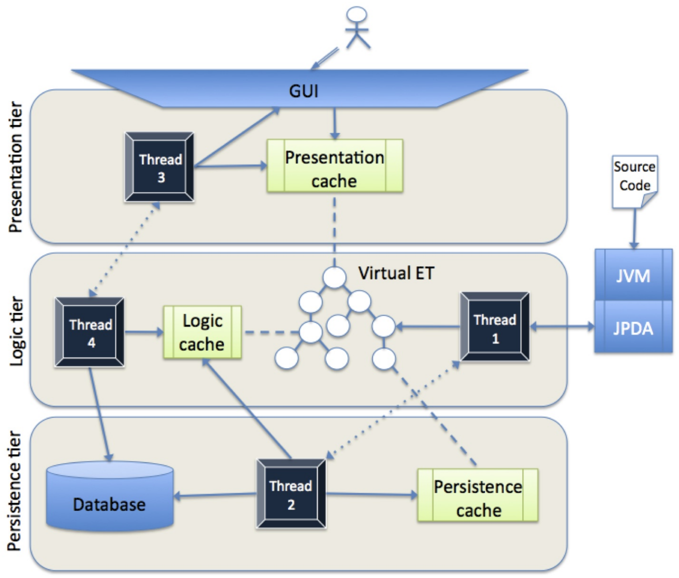


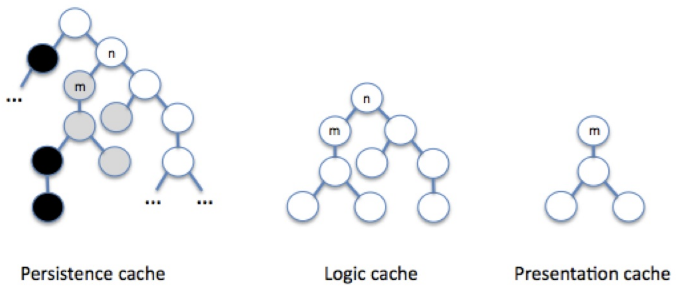
Figure 4.1: Architecture of a scalable declarative debugger

presentation cache defines a subtree inside the logic cache. Therefore, all the nodes in the presentation cache are also nodes of the logic cache. Here, the subtree is defined by selecting one root node and a depth (in the following, *presentation bound*).

The whole VET does not usually fit in main memory. Therefore, a mechanism to remove nodes from it and store them in a database is needed. When the number of complete nodes in the VET is close to the persistence bound, some of them are moved to the database, and only their identifiers remain in the VET. This allows the debugger to keep the whole ET structure in main memory and use identifiers to retrieve nodes from the database when needed.

Example 4.1.1

Consider the following trees:



The tree at the left is the VET of a debugging session where gray nodes are those already completed (their associated method execution already finished); black nodes are completed nodes that are only stored in the database (only their identifiers are stored in the VET), and white nodes are nodes that have not been completed yet (they represent a method execution that has not finished yet). It could be possible that some of the white nodes had a children not generated yet. Note that this VET is associated with an instant of the execution; and new nodes could be generated later. The tree in the middle is the part of the VET referenced by the logic cache, in this case it is a tree, being n the root node and a depth of four, but in general it could contain unconnected nodes. Similarly, the tree at the right is the part of the VET referenced by the presentation cache, with m the root node and a depth of three. Note that the presentation cache is a subset of the logic cache.

The behavior of the debugger is controlled by four threads that run in parallel, one for the presentation tier (Thread 3), two for the logic tier (threads 1 and 4) and one for the persistence tier (Thread 2). Threads 1 and 2 control the generation of the VET and its storage in the database. They collaborate via synchronizations and message passing. Threads 3 and 4 communicate with the user and generate the questions. They also collaborate and are independent of Threads 1 and 2. A description of the threads and their behavior specified with pseudo-code follows:

Thread 1 (Contruction of the VET) This thread is in charge of constructing the VET. It is the only one that communicates with the JPDA and JVM. Therefore, we could easily construct a declarative debugger for another language (e.g., C++) by only replacing this thread. Basically, this thread executes the program and for every method invocation performed, it constructs a new node stored in the VET. When the number of complete nodes (given by function *completeNodes*) is close to the persistence bound, this thread sends to Thread 2 the *wake up* signal. Then, Thread 2 moves some nodes to the database. If the persistence bound is reached, Thread 1 sleeps until enough nodes have been removed from the VET and it can continue generating new nodes.

Thread 2 (Controlling the size of the VET) This thread ensures that the VET always fits in main memory. It controls what nodes of the VET should be stored in main memory, and what nodes should be stored in the database.

When the number of completed nodes in the VET is close to the persistence bound Thread 1 wakes up Thread 2 that removes some[2] nodes

[2]In our implementation, it removes half of the nodes. Our experiments reveal that this

Algorithm 7 Contruction of the VET (Thread 1)

```
Input: A source program $\mathcal{P}$
Output: A VET $\mathcal{V}$
Initialization: $\mathcal{V} = \varnothing$

repeat
(1) Run $\mathcal{P}$ with JPDA and catch event $e$
    case $e$ of
    new method invocation $I$:
(2)     create a new node $N$ with $I$
(3)     add $N$ to $\mathcal{V}$
    method invocation $I$ ended:
(4)     complete node $N$ associated with $I$
(5)     If completeNodes($\mathcal{V}$) == persistenceBound/2
(6)     then send to Thread 2 the wake up signal
(7)     If completeNodes($\mathcal{V}$) == persistenceBound
(8)     then sleep
until $P$ finishes or the bug is found
```

from the VET and copies them to the database. It uses the logic cache to decide what nodes to store in the database. Concretely, it tries to store in the database as many nodes as possible that are not in the logic cache. When it finishes, it sends to Thread 1 the *wake up* signal and sleeps.

Algorithm 8 Controlling the size of the VET (Thread 2)

```
Input: A VET $\mathcal{V}$
Output: An ET stored in a database

repeat
1) Sleep until wake up signal is received
   repeat
2)     Look into the persistence cache for the next completed node $N$ of the VET
3)     if $N$ is not found
4)     then wake up Thread 1
5)          break
6)     else store $N$ in the database
7)     if $N$ is the root node then exit
```

Thread 3 (Interface communication) This thread is the only one that

is a good choice because it keeps Threads 1 and 2 continuously running in a producer-consumer manner.

communicates with the user. It controls the information shown in the GUI with the presentation cache. According to the user's answers, the strategy selected, and the presentation bound, this thread selects the root node of the presentation cache. This task is done question after question according to the programmer answers, ensuring that the question asked (using function *AskQuestion*), its parent, and as many descendants as the presentation bound allows, are shown in the GUI.

Algorithm 9 Interface communication (Thread 3)

```
Input: Answers of the user
Output: A buggy node

repeat
(1) ask Thread 4 to produce a question
(2) update presentation cache and GUI visualization
(3) answer = AskQuestion(question)
(4) send answer to Thread 4
until a buggy node is found
```

Thread 4 (Selecting questions) This thread chooses the next question according to a given strategy using function *SelectNextQuestion* that implements standard strategies. With the node selected, the logic cache is updated and all the nodes in the logic cache are loaded from the database. This is done with function *UpdateLogicCache* that uses the node selected as the root, and the logic bound to compute the logic cache. All the nodes that belong to the new logic cache and that do not belong to the previous logic cache are loaded from the database using function *FromDatabaseToET*.

Algorithm 10 Selecting questions (Thread 4)

```
Input: A strategy $\mathcal{S}$ and a VET $\mathcal{V}$
Output: A buggy node

repeat
(1) question = SelectNextQuestion($\mathcal{V}$,$\mathcal{S}$)
(2) missingNodes = UpdateLogicCache()
(3) If (question $\notin \mathcal{V}$) then $\mathcal{V}$ = FromDatabaseToET($\mathcal{V}$,missingNodes)
(4) send question to Thread 3
(5) get answer from Thread 3
until a buggy node is found
```

4.1.2 Redefining the strategies for Declarative Debugging

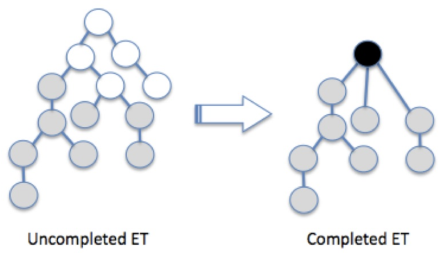
In Algorithm 10, a strategy is used to generate the sequence of questions by selecting nodes in the VET. Nevertheless, all declarative debugging strategies in the literature have been defined for ETs and not for VETs where incomplete nodes can exist. All of them assume that the information of all ET nodes is available. Clearly, this is not true in our context and thus, the strategies would fail. For instance, the first node asked by the strategy Top-Down and its variants is always the root node of the ET. However, this node is the last node completed by Algorithm 9. Hence, these strategies could not even start until the whole ET is completed, and this is exactly the problem that we want to avoid.

Therefore, in this section we propose a redefinition of the strategies for declarative debugging so that they can work with VETs.

A first solution could be to define a transformation from a VET with incomplete nodes to a VET where all nodes are completed. This can be done by inserting a new root node with the equation $1 = 0$. Then, the children of this node would be all the completed nodes whose parent is incomplete. In this way, (i) all nodes of the produced ET would be completed and could be asked; (ii) the parent-child relation is kept in all the subtrees of the ET; and (iii) it is guaranteed that at least one bug (the root node) exists. If the debugging session finishes with the root node as buggy, it means that the node with the "real" bug (if any) has not been completed yet.

Example 4.1.2

Consider the following VETs:



In the VET at the left, gray nodes are completed, and white nodes are incomplete. This VET can be transformed into the VET at the right where all nodes are completed. The new artificial root is the black node which ensures that at least one buggy node exists.

From an implementation point of view, this transformation is inefficient and costly because the VET is being generated continuously by Thread 1, and hence, this transformation should be done repeatedly question after question. In contrast, a more efficient solution is to redefine the strategies so that they ignore incomplete nodes. For instance, Top-Down [1] only asks for the descendants of a node that are completed and that do not have a completed ancestor. Similarly, Binks Top-Down [2] would ask first for the completed descendant that in turn contains more completed descendants. D&Q [25] would ask for the node that divides the VET in two subtrees with the same number of nodes (completed or not), and so on. We refer the interested reader to the source code of our implementation that is publicly available and where all strategies have been reimplemented for VETs.

Even though the architecture presented has been discussed in the context of Java, it can work for other languages with very few changes. Observe that the part of an algorithmic debugger that is language-dependent is the front-end, and our technique relies on the back-end. Once the VET is generated, the back-end can handle the VET mostly independent of the language. In particular, the strategies can traverse the VET being unaware of the meaning of the questions.

4.1.3 Implementation

We have implemented the technique presented in this paper and integrated it into DDJ 2.4. The implementation has been tested with a collection of real applications (e.g., an interpreter, a parser, a debugger, etc).

Table 4.1.3 summarizes the results of the experiments performed. These experiments have been done in an Intel Core2 Quad 2.67 GHz with 2GB RAM.

Benchmark	var. num.	ET size	ET time	node time	cache lim.	ET depth
argparser	8.812	2 MB	22 s.	407 ms.	7/7	7
cglib	216.931	200 MB	230 s.	719 ms.	11/14	18
kxml2	194.879	85 MB	1318 s.	1844 ms.	6/6	9
javassist	650.314	459 MB	556 s.	844 ms.	7/7	16
jtstcase	1.859.043	893 MB	1913 s.	1531 ms.	17/26	57
HTMLcleaner	3.575.513	2909 MB	4828 s.	609 ms.	4/4	17

Table 4.1: Benchmark results

The first column contains the names of the benchmarks. For each benchmark, the second and third columns give an idea of the size of the execution. Fourth and fifth columns are time measures. Finally, sixth and seventh columns show memory bounds. Concretely, column `variables number` shows the number of variables participating (possibly repeated) in the execution considered. Column `ET size` shows the size in MB of the ET when it is completed, this measure has been taken from the size of the ET in the database (of course, it includes compaction). Column `ET time` is the time needed to finish the ET. Column `node time` is the time needed to complete the first node of the ET. Column `cache limit` shows the presentation bound and the depth of the logic cache of these benchmarks. After these bounds, the computer was out of memory. Finally, column `ET depth` shows the depth of the ET after it was constructed.

Observe that a standard declarative debugger is hardly scalable to these real programs. With the standard technique, even if the ET fits in main memory or we use a database, the programmer must wait for a long time until the ET is completed and the first question can be asked. In the worst case, this time is more than one hour. Contrarily, with the new technique, the debugger can start to ask questions before the ET is completed. Note that the time needed to complete the first node is always less than two seconds. Therefore, the debugging session can start almost instantaneously.

The last two columns of the table give an idea of how big is the ET shown in the GUI before it is out of memory. In general, five levels of depth is enough to see the question asked and the part of the computation closely related to this question. In the experiments only HTMLcleaner was out of memory when showing five levels of the ET in the GUI.

All the information related to the experiments, the source code of the benchmarks, the bugs, the source code of the tool and other material can be found at `http://www.dsic.upv.es/~jsilva/DDJ`.

4.2 Balancing execution trees

The debugger explores the ET with a strategy in order to find the computation (i.e., the node) that caused the bug. The performance of all search strategies depends on the size and the structure of the ET and this is the reason why many efforts in the literature try to reduce the size of the ET with strategies that prune a big part of it (e.g., [11, 29]), or with techniques that eliminate redundant and correct nodes (e.g., tree compression [8] and trusting [16]).

In contrast, in this work we show that reducing the size of the ET is as important as optimizing its structure.[3] We introduce a technique that changes the structure of the ET in such a way that the search strategies present an almost optimal behavior. The objective of the technique is to balance the ET (i.e., to transform the ET into a binomial tree) in such a way that search strategies prune half of the ET at every step, because they can always find a node that divides the search area in half. Our experiments with real programs show that the technique reduces (as an average) the number of questions to the programmer by around 30%. Moreover, since the newly introduced nodes try to reproduce the intended behavior of the program the user had in mind, the difficult of the new questions is not critically increased.

Our technique presents three important advantages that make it useful for any declarative debugger. First, it can be easily adapted to other programming languages. We have implemented it for Java, but we have used it with languages such as Haskell or Prolog. Second, the technique is quite simple to implement and can be integrated into any existing declarative debugger with small changes. And third, the technique is conservative. If the questions triggered by the new nodes introduced by the technique are difficult to answer, the user can answer “I don’t know”, and continue the debugging session as in the standard ETs. Moreover, the user can naturally get back to the original ET in case it is needed.

4.2.1 Collapsing and projecting nodes

Even though the strategy Heaviest First significantly improves Top-Down search, its performance strongly depends on the structure of the ET. The more balanced the ET is, the better. Clearly, when the ET is balanced, Heaviest First is much more efficient because it prunes more nodes after every question. If the ET is completely balanced, heaviest first is equivalent to Divide and Query and both are query-optimal.

[3]However, it should be clear that our technique is not incompatible with any of these works. An ET could be first compressed [8] and then balanced with our technique.

Advantages of collapsing and projecting nodes

Our technique is based on a series of transformations that allows us to collapse/project some nodes of the ET. A collapsed node is a new node that replaces some nodes that are then removed from the ET. In contrast, a projected node is a new node that is placed as the parent of a set of nodes that remain in the ET. This section describes the main advantages of collapsing/projecting nodes in the ET:
Balancing execution trees. If we augment an ET with projected nodes, we can strategically place the new projected nodes in such a way that the ET becomes balanced. In this way, the debugger speeds up the debugging session by reducing the number of asked questions.

Example 4.2.1

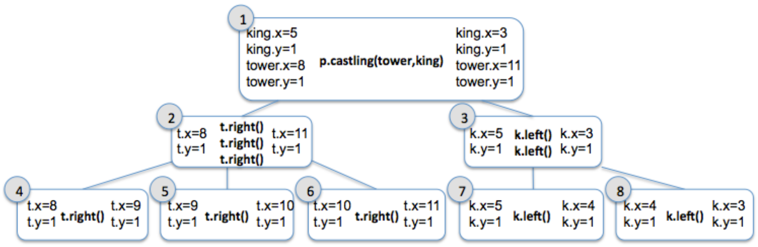


Figure 4.2: Balanced ET of the Example 1.1.3

Consider again the program in Figure 1.1. The portion of the ET associated with the method execution `p.castling(tower,king)` is shown in Example 1.1.3. We can add projected nodes to this ET as depicted in Figure 4.2.1. Note that now the ET becomes balanced, and hence, many strategies ask fewer questions. For instance, in the worst case, using the ET of Figure 1.1.3 the debugger would ask about all the nodes before the bug is found. This is due to the broad nature of this ET that prevents strategies from pruning any nodes. In contrast, using the ET of Figure 4.2.1 the debugger prunes almost half of the tree with every question. In this example, with the standard ET of Example 1.1.3, D&Q produces the following debugging session (numbers refer to the codes of the nodes in the figure):

```
Starting Debugging Session...
(2) YES  (3) YES  (4) YES  (5) YES  (6) YES  (1) NO
Bug found in method: castling(Position t, Position k) of class Chess
```

In contrast, with the ET of Figure 4.2.1, D&Q produces the following session:

```
Starting Debugging Session...
(2) YES  (3) YES  (1) NO
Bug found in method: castling(Position t, Position k) of class Chess
```

Skipping repetitive questions. Declarative debuggers tend to repeat the same (or very similar) question several times when it is associated with a method execution that is inside a loop. In our example, this happens in `for(int i=1; i<=3; i++) {t.right();}`, which is used to move the tower three positions to the right. In this case, the nodes

```
{t.x=8,  t.y=1}  t.right()  {t.x=9,  t.y=1}
{t.x=9,  t.y=1}  t.right()  {t.x=10, t.y=1}
{t.x=10, t.y=1}  t.right()  {t.x=11, t.y=1}
```

could be projected to the node

```
{t.x=8, t.y=1}    t.right(); t.right(); t.right()    {t.x=11, t.y=1}
```

This kind of projection, where all the projecting nodes refer to the same method, has an interesting property: If the projecting nodes are leaves, then they can be deleted from the ET. The reason is that the new projected node and the projecting nodes refer to the same method. Therefore, it does not matter what computation produced the bug, because the bug will necessarily be in this method. Hence, if the projected node is wrong, then the bug is in the method pointed to by this node. When the children of the projected node are removed, we call it *collapsed node.*

Note that, in this case, the idea is not to add nodes to the ET as in the previous case, but to delete them. Because the input and output of all the questions relate to the same attributes (i.e., x and y), then the user can answer them all together, since they are, in fact, a sequence of operations whose output is the input of the next question (i.e., they are chained). Therefore, this technique allows us to treat a set of questions as a whole. This is particularly interesting because it approximates the real behavior intended by the programmer. For instance, in this example, the intended meaning of the loop was to move the tower three positions to the right. The intermediate positions are not interesting for the programmer, only the initial and final ones are meaningful for the intended meaning.

Example 4.2.2

Consider again the ET of Figure 4.2.1. Observe that, if the projected nodes are wrong, then the bug must be in the unique method appearing in the projected node. Thus, we could collapse the node instead of projecting it. Hence, nodes 4, 5, 6, 7, and 8 could be removed; and thus, with only three questions we could discover any bug in any node.

Enhancing the search of declarative debugging. One important problem of declarative debugging strategies is that they must use a given ET without any possibility of changing it. This often prevents declarative debuggers from asking questions that prune a big part of the ET, or from asking questions that concentrate on the regions with a higher probability of containing the

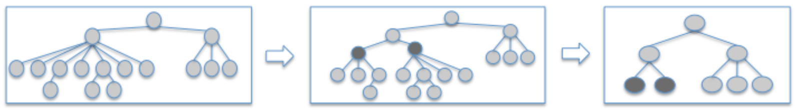


Figure 4.3: Transformation of ETs

bug. The collapse of some subtrees into a single node can help to solve these drawbacks.

The initial idea of this section was to use projected nodes to balance the ET. This idea is very interesting in combination with D&Q, because it can cause the debugging session to be optimal in the worst case (its query complexity is $O(b \cdot log\ n)$, where b is the branching factor and n is the number of nodes in the ET). However, this idea could be further extended in order to force the strategies to ask questions related to parts of the computations with a higher probability of containing the bug. Concretely, we can replace parts of the ET with a collapsed node in order to avoid questions related to this part. If the debugging session determines that the collapsed node is wrong, we can expand it again to continue the debugging session inside this node. Therefore, with this idea, the original ET is transformed into a tree of ETs that can be explored when it is required. Let us illustrate this idea with an example.

Example 4.2.3

Consider the leftmost ET in Figure 4.3. This ET has a root that started two subcomputations. The computation on the left performed ten method executions, while the computation on the right performed only three. Hence, in this ET, all the existing declarative debugging strategies would explore first the left subtree.[4] If we balance the left branch by inserting projected nodes we get the new ET shown on the right of the previous one. This balanced ET requires (on average) fewer questions than the previous one; but the strategies will still explore the left branch of the root first.

Now, let us assume that the debugger identified the right branch as more likely to be buggy (e.g., because it contains recursive calls, because it is non-deterministic, because it contains calls with more arguments involved or with complex data structures...). We can change the structure of the ET in order to make declarative debugging strategies start by exploring the desired branch. In this example we can remove from the ET the nodes that were projected. The new ET is shown on the right of Figure 4.3. With this ET the debugger will explore first the right branch of the root. Observe that it is not necessary that the nodes that were projected refer to the same method. They can be completely different and independent computations.

[4]Current strategies assume that all nodes have the same probability of being buggy, therefore, heavy branches are explored first.

However, if the debugger determines that they are probably correct, they can be omitted to direct the search to other parts of the ET. Of course, they can be expanded again if required by the strategy (e.g., if the debugger cannot find the bug in the other nodes).

Reducing the size of the ET. One important problem of modern declarative debuggers is scalability. With realistic programs, the size of the ET can be huge (indeed gigabytes) and, thus, it does not fit in main memory. The same scalability problem affects graphical user interfaces (GUI). Loading the whole ET in the GUI is often too slow as to be useful, and it is often impossible because, again, all graphical objects do not fit in the graphical memory. Our experiments show that collapsing nodes allows us to increase the number of ET levels shown to the user. For instance, for some programs the debugger is only able to load 4 levels of the ET in the GUI because the next level would produce a memory overflow. With the collapsing technique, we could load 5 levels due to the reduction of nodes. In particular, some loops contain hundreds of nodes that are collapsed into a single one.

Collapsing and projecting algorithms

In this section we define a technique that allows us to balance an ET while keeping the soundness and completeness of declarative debugging. The technique is based on two basic transformations for ETs (namely *collapse leaf chain* and *project chain*, described respectively by Algorithms 11 and 12), and on a new data structure called an *Execution Forest* (EF) that is a generalization of an ET.

Definition 4.2.4 (Execution Forest) *An* execution forest *is a tree $t = (V, E)$ whose internal vertices V are method executions and whose leaves are either method executions or execution forests.*

Roughly speaking, an EF is an ET where some subtrees have been replaced (i.e., collapsed) by a single node. Observe that this recursive definition of EF is more general than the definition of ET because an ET is an instance of an EF where no collapsed nodes exist. We can now define the two basic transformations of our technique. Both transformations are based on the notion of *chain.* Informally, a chain is formed by an ordered set of sibling method executions in which the final context produced by a method of the chain is the initial context of the next method. Chains often represent a set of method executions performed one after the other during an execution. Formally,

Definition 4.2.5 (Chain) *Given an EF $t = (V, E)$ and a set $C \subset V$ of n nodes with associated method executions $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n$ we say that C is a*

Algorithm 11 Collapse Leaf Chain

```
**Input:** An EF $t = (V, E)$ and a set of nodes $C \subset V$
**Output:** An EF $t' = (V', E')$
**Preconditions:** $C$ is a chain with nodes $(a_1, m_1, a_2)$, $(a_2, m_2, a_3)$, …, $(a_n, m_n, a_{n+1})$ and $\nexists\ v \in V\ .\ (c \to v) \in E$, with $c \in C$
begin
1) $parent = u \in V$ such that $\forall c \in C, (u \to c) \in E$
2) $colnode = (a_1, m, a_{n+1})$ with $m = m_1; m_2; \ldots; m_n$
3) $V' = (V \setminus C) \cup \{colnode\}$
4) $E' = ((E \setminus \{(parent \to v) \in E \mid v \in C\}) \cup \{(parent \to colnode)\}$
end
return $t' = (V', E')$
```

Algorithm 12 Project Chain

```
**Input:** An EF $t = (V, E)$ and a set of nodes $C \subset V$
**Output:** An EF $t' = (V', E')$
**Preconditions:** $C$ is a chain with nodes $(a_1, m_1, a_2)$, $(a_2, m_2, a_3)$, …, $(a_n, m_n, a_{n+1})$
begin
1) $parent = u \in V$ such that $\forall c \in C, (u \to c) \in E$
2) $projnode = (a_1, m, a_{n+1})$ with $m = m_1; m_2; \ldots; m_n$
3) $V' = V \cup \{projnode\}$
4) $E' = ((E \setminus \{(parent \to v) \in E \mid v \in C\}) \cup \{(parent \to projnode)\} \cup \{(projnode \to c) \mid c \in C\}$
end
return $t' = (V', E')$
```

chain *iff (i)* $\exists v \in V$ *such that* $\forall c \in C\ .\ (v \to c) \in E$, *and (ii)* $\forall j, 1 \leqslant j \leqslant n-1$, *if* $\mathcal{E}_j = (a_j, m_j, a_{j+1})$ *then* $\mathcal{E}_{j+1} = (a_{j+1}, m_{j+1}, a_{j+2})$.

The first condition ensures that all the elements in the chain are siblings. The second one ensures that for all nodes in the chain the final context of a node is the initial context of the next chained node. It is common to find chains when one or more methods are executed inside a loop. As explained in Example 4.2.2, when the chain is formed by a single method without children that is repeated, all the nodes that form the chain can be collapsed into a single node and the chain deleted from the EF. This collapsed node only needs to show to the user the initial and the final context of the chain.

The other transformation for chains is based on the projection of a chain producing a new (parent) node whose question represents the whole chain (see Example 4.2.1). In this case, the chain remains in the EF. This transformation is very fitting to balance the EF.

The basic transformations of chains are described by Algorithms 11 and 12. Algorithm 11 is in charge of collapsing chains, which consists in creating a new node *colnode* with initial context the context before applying the first method of the chain, final context the context after applying the last method of the chain, and with the composition of the methods in the chain as associated method. Then, the nodes in the chain (and thus their links) are removed from the tree and the new node is linked to the parent of the nodes in C, thus reducing the size of the EF. Algorithm 12 is in charge of projecting chains, and works in a similar way to the algorithm before. Given a tree t and a chain C in the tree, it removes from t the links between each $c \in C$ and its parent *parent*, and then introduces a new node *projnode*, built as explained before, which is linked to each c as their new parent, and to *parent* as its new child. As explained in the previous section, although this algorithm introduces a new node it improves the performance of the strategy by balancing the EF.

Algorithm 13 is in charge of removing the chains of leaves that can be collapsed. It first computes in the initialization all the chains of nodes that are leaves and are related to the same methods. Then, for each of these chains, it applies Algorithm 11 to collapse them by removing the chain from the tree and adding the corresponding collapsed node.

Our method for balancing EFs is implemented by Algorithm 14. This algorithm first uses Algorithm 13 to shrink the EF (line 1); and then it balances this shrunken EF by projecting some nodes. The objective is to divide the tree into two parts with the same weight. Therefore, we first compute the half of the size of the EF (lines 4 and 5). If a child of the root is already heavier than half the size of the tree, then, the weight of this node is not taken into account in the balancing process because the question associated to this node will be the first question asked (lines 9-15). Otherwise, it projects the part of a chain whose weight is as close as possible to half the weight of the root (lines 16-24). This allows us to prune half of the subtree when asking a question associated to a projected node. In the case that the heavier node (lines 13-15) or the projected chain (lines 18-19) belongs to a bigger chain, it must be cut with function *cutChain* producing new (smaller) chains that are also processed. Of course, the size of the chains already processed is not taken into account when dividing the successive (sub)chains.

The algorithm finishes when no more chains can be projected. Trivially, because the number of chains and their length are finite, termination is ensured. In addition, Algorithm 14 is able to balance the EF while it is being computed. Concretely, the algorithm should be executed for each node of the EF that is completed (i.e., the result of the method execution is already calculated, thus all the children of this node are also completed). Note that this means that the algorithm is applied bottom-up to the nodes of the EF. Hence, when balancing a node, all the descendants of the node have been already balanced. This also means that modern debuggers that are able to debug programs with uncompleted ETs [12] can also use the technique, because the ET can be balanced while it is being computed.

We showed in the previous section that collapsing nodes can be very useful. However, collapsing nodes is not always a good idea, because they may introduce difficult questions that delay the debugging session. After several experiments, we found a situation where collapsing nodes often produces good results; this situation happens when the collapsed nodes form a chain. Therefore, the algorithm only projects and collapses nodes that belong to a chain.[5] When the chain is composed by a single function and all the nodes of the chain are leaves, the whole chain can be replaced by a single collapsed node. All the collapsed nodes are computed first, and then the projected nodes are calculated. If the chain is very long, it can be cut in several subchains to be projected and thus balance the EF. In order to cut chains we use the function *cutChain*(chain $\{c_1, ..., c_n\}$, int i, j), that removes from the chain a subchain delimited by indexes i and j.

Another conclusion of our experiments is that all these transformations must be done only when the questions about the produced collapsed or projected nodes are not very hard to answer. A good measure to achieve this is counting the number of changes in the state produced by the chain. In our implementation, we took as a design decision that no collapsed nor projected node contains more than five changes in the state. In particular, we only collapse/project chains where the number of *different* attributes changed is not higher than five. Note that, in object-oriented languages, attributes can be objects. Therefore, in our implementation, any change in the state of the object-valued attribute is taken into account as a change in the chain. Of course, if we drop the 5-changes restriction, the balancing process would be much more accurate producing very efficient ETs but after intensive testing with the tool we preferred to produce more conservative balanced ETs with less projections but ensuring that questions associated to projections are easy to answer. In any case, the last release of the tool allows us to change (or even remove) this parameter.

4.2.2 Correctness

Our technique for balancing EFs is based on the transformations presented in the previous section. We show first that the execution tree is complete (an ET with a wrong root contains a buggy node) and sound (a buggy node is associated with a buggy method).

[5]Note that, since these chains are usually found when a loop or a recursive call is used, our approach generates nodes whose questions are very close to the intended meaning the programmer had in mind while developing the program and thus, although the new questions comprise a bigger context, they may be even easier to answer than the "atomic" ones.

Algorithm 13 Shrink EF

```
Input: An EF $t = (V, E)$
Output: An EF $t' = (V', E')$
Preconditions: Given a node $v$, $v.method$ is the name of the method.
Initialization: $t' = t$, set $\mathcal{S}$ contains all the chains of $t$ s.t.
$\forall s = \{c_1, \ldots, c_n\} \in \mathcal{S}, \forall x,\ 1 \leqslant x \leqslant n-1,\ c_x.method == c_{x+1}.method,\ \wedge$
$\nexists\ v \in V\ .\ (c \to v) \in E$, with $c \in s$
begin
1)  while ($\mathcal{S} \neq \emptyset$)
2)      take a chain $s \in \mathcal{S}$
3)      $\mathcal{S} = \mathcal{S} \backslash \{s\}$
4)      $t' = collapseChain(t', s)$
    end while
end
return $t'$
```

Theorem 4.2.6 (Completeness and soundness of EFs) *Given an EF with a wrong root, it contains a buggy node which is associated with a buggy method.*

Completeness and soundness are kept after our transformations. We prove that an EF with a buggy node will still have a buggy node after any number of collapses or projections.

Theorem 4.2.7 (Chain Collapse Correctness) *Let $t = (V, E)$ and $t' = (V', E')$ be two EFs, being the root of t wrong, and let $C \subset V$ be a chain such that all nodes in the chain are leaves and they have the same associated method. Given $t' =$* `collapseChain(t,C)`*,*

1. *t' contains a buggy node.*
2. *Every buggy node in t' is associated with a buggy method.*

Theorem 4.2.8 (Chain Projection Correctness) *Let $t = (V, E)$ and $t' = (V', E')$ be two EFs, and let $C \subset V$ be a chain such that $t' =$* `projectChain(t,C)`*.*

1. *All buggy nodes in t are also buggy nodes in t'.*
2. *Every buggy node in t' is associated with a buggy method.*

We provide in this section an interesting result related to the projection of chains. This result is related to the incompleteness of the technique when it is used intra-session (i.e., in a single debugging session trying to find one particular bug). Concretely, the following result does not hold: *A buggy node can be found in an EF if and only if it can be found in its balanced version.*

Algorithm 14 Shrink & Balance EF

```
Input: An EF $t = (V, E)$ whose root is $root \in V$
Output: An EF $t' = (V', E')$
Preconditions: Given a node $v$, $v.weight$ is the size of the subtree of $v$.
begin
1)  $t' = shrink(t)$
2)  $children = \{v \in V' \mid (root \rightarrow v) \in E'\}$
3)  $\mathcal{S} = \{s \mid s$ is a chain in $children\}$
4)  $rootweight = root.weight$
5)  $weight = rootweight/2$
6)  while $(\mathcal{S} \neq \emptyset)$
7)    $child = c \in children \mid \nexists c' \in children, c \neq c' \wedge c'.weight > c.weight$
8)    $distance = |weight - child.weight|$
9)    if $(child.weight \geqslant weight$ or $\nexists s, i, j$ s.t. $s = \{c_1, \ldots, c_n\} \in S \wedge$
        $(|W - weight| < distance)$ with $W = \sum_{x=i}^{j} c_x.weight)$
10)   then $children = children \backslash \{child\}$
11)     $rootweight = rootweight - child.weight$
12)     $weight = rootweight/2$
13)     if $(\exists s \in \mathcal{S}$ such that $s = \{c_1, \ldots, c_n\}$ and $child = c_i, 1 \leqslant i \leqslant n)$
14)     then $(s_{ini}, s_{end}) = cutChain(s, i, i)$
15)       $\mathcal{S} = (\mathcal{S} \backslash \{s\}) \cup s_{ini} \cup s_{end}$
        end if
      else
16)     find an $s, i, j$ such that $s = \{c_1, \ldots, c_n\} \in S$ and $\sum_{x=i}^{j} c_x.weight$ is
as close as possible to $weight$
17)     $s' = \{c_i, \ldots, c_j\}$
18)     $(s_{ini}, s_{end}) = cutChain(s, i, j)$
19)     $\mathcal{S} = (\mathcal{S} \backslash \{s\}) \cup s_{ini} \cup s_{end}$
20)     $t' = projectChain(t', s')$
21)     for each $c \in s'$
22)       $rootweight = rootweight - c.weight$
        end for each
23)     $children = (children \backslash s')$
24)     $weight = rootweight/2$
      end if
    end while
end
return $t' = (V', E')$
```

In general, our technique ensures that all the bugs that caused the wrong behavior of the root node (i.e., the wrong final state of the whole program) can be found in the balanced EF. This means that all the buggy nodes that are responsible of the wrong behavior are present in the balanced EF.

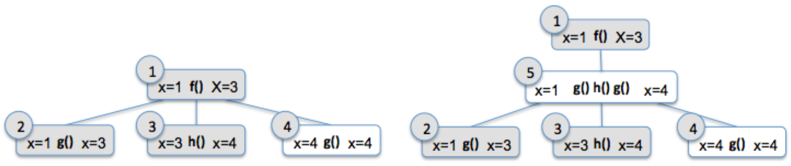


Figure 4.4: New buggy nodes revealed

However, declarative debugging can find bugs by a fluke. Those nodes that are buggy nodes in the EF but did not cause the wrong behavior of the root node can be undetectable with some strategies in the balanced version of the EF. The opposite is also true: It is possible to find bugs in the balanced EF that were undetectable in the original EF:

Example 4.2.9

Consider the EFs in Figure 4.4. The EF on the right is the same as the one on the left but a new projected node has been added. If we assume the following intended semantics, then grey nodes are wrong and white nodes are right:

$x = 1\ f()\ x = 2 \quad x = 4\ h()\ x = 4 \quad x = 3\ h()\ x = 3 \quad x = 4\ g()\ x = 4 \quad x = 1\ g()\ x = 4$

Note that in the EF on the left, only nodes 2 and 3 are buggy. Therefore, all the strategies will report these nodes as buggy, but never node 1. However, node 1 contains a bug but it is undetectable by the debugger until nodes 2 and 3 have been corrected. Nevertheless, observe that nodes 2 and 3 did not produce the wrong behavior of node 1. They simply produced two errors that, in combination, produced by a fluke a global correct behavior.

Now, observe in the EF on the right that node 1 is buggy and thus detectable by the strategies. In contrast, nodes 2 and 3 are now undetectable by Top-Down search (they could be detected by D&Q). Thanks to the balancing process, it has been made explicit that three different bugs are in the EF.

4.2.3 Implementation

We have implemented the technique presented in this paper and integrated it into the debugger Declarative Debugger for Java DDJ 2.4 [13]. The implementation allows the programmer to activate the transformations of the technique and to parameterize them in order to adjust the size of the projected/collapsed chains. It has been tested with a collection of small to large programs including real applications (e.g., an interpreter, a compiler, an XSLT processor, etc.) producing good results, as summarized in Table 4.2.

Each benchmark has been evaluated assuming that the bug could be in any node. This means that each row of the table is the average of a number of

Benchmark	ET nodes	Proj./Col.	Proj./Col. nodes	Bal. Time	Quest.	Q. Bal.	%
NumReader	12 nodes	0/0	0/0 nodes	0 ms.	6,46	6,46	0,00%
Orderings	72 nodes	2/14	5/45 nodes	0 ms.	11,47	8,89	22,46%
Factoricer	62 nodes	7/0	17/0 nodes	0 ms.	13,89	7,90	43,09%
Sedgewick	41 nodes	3/8	7/24 nodes	0 ms.	18,79	7,52	59,95%
Clasifier	30 nodes	4/7	10/20 nodes	0 ms.	15,52	6,48	58,21%
LegendGame	93 nodes	12/20	28/40 nodes	0 ms.	16,00	9,70	39,36%
Cues	19 nodes	3/1	8/2 nodes	0 ms.	10,40	8,20	21,15%
Romanic	123 nodes	20/0	40/0 nodes	0 ms.	25,06	16,63	33,66%
FibRecursive	6724 nodes	19/1290	70/2593 nodes	344 ms.	38,29	21,47	43,92%
Risk	70 nodes	7/8	19/43 nodes	0 ms.	30,69	10,28	66,50%
FactTrans	198 nodes	5/0	12/0 nodes	0 ms.	18,96	14,25	24,88%
RndQuicksort	88 nodes	3/3	9/0 nodes	0 ms.	12,88	10,40	19,20%
BinaryArrays	132 nodes	7/0	18/0 nodes	0 ms.	15,56	10,58	32,03%
FibFactAna	380 nodes	3/29	9/58 nodes	0 ms.	30,13	29,15	3,27%
NewtonPol	46 nodes	1/3	2/40 nodes	0 ms.	23,09	4,77	79,35%
RegresionTest	18 nodes	1/0	3/0 nodes	0 ms.	6,84	6,26	8,46%
BoubleFibArrays	214 nodes	0/40	0/83 nodes	0 ms.	12,42	12,01	3,33%
ComplexNumbers	68 nodes	17/9	37/18 nodes	16 ms.	20,62	10,20	50,53%
StatsMeanFib	104 nodes	3/20	6/56 nodes	0 ms.	12,33	11,00	10,81%
Integral	25 nodes	0/2	0/22 nodes	0 ms.	8,38	3,38	59,63%
TestMath	51 nodes	1/2	2/5 nodes	0 ms.	12,77	11,65	8,73%
TestMath2	267 nodes	7/13	16/52 nodes	31 ms.	66,47	58,33	12,24%
Figures	116 nodes	8/3	16/6 nodes	0 ms.	13,78	12,17	11,66%
FactCalc	105 nodes	3/11	8/32 nodes	0 ms.	19,81	12,64	36,19%
SpaceLimits	127 nodes	38/0	76/0 nodes	0 ms.	40,85	29,16	28,61%
Argparser	129 nodes	31/9	70/37 nodes	16 ms.	20,78	12,71	38,85%
Cglib	1216 nodes	67/39	166/84 nodes	620 ms.	80,41	65,01	19,15%
Javassist	1357 nodes	10/8	28/24 nodes	4.745 ms.	79,52	77,50	2,54%
Kxml2	1172 nodes	260/21	695/42 nodes	452 ms.	79,61	28,21	64,56%
HTMLcleaner	6047 nodes	394/90	1001/223 nodes	8.266 ms.	169.49	138,85	18,08%
Jtestcase	4151 nodes	299/27	776/54 nodes	1.328 ms.	85,05	80,52	5,32%

Table 4.2: Benchmark results

experiments. For instance, `cglib` was tested 1.216 times (i.e., the experiment was repeated choosing a different node as buggy, and all nodes were tried). For each benchmark, column `ET nodes` shows the size of the ET evaluated; column `Proj./Col.` shows the number of projected/collapsed nodes inserted into the EF; column `Proj./Col. nodes` shows the number of nodes that were projected and collapsed by the debugger; column `Bal. time` shows the time needed by the debugger to balance the whole EF; column `Quest.` shows the average number of questions done by the debugger before finding the bug in the original ET; column `Q. Bal.` shows the average number of questions done by the debugger before finding the bug in the balanced ET; finally, column `(%)` shows the improvement achieved with the balancing technique. Clearly, the balancing technique has an important impact in the reduction of questions with a mean reduction of 30% using Top-Down.

Essentially, our debugger produces the EF and transforms it by collapsing and projecting nodes by using Algorithm 14. Finally, it is explored with standard strategies starting the debugging session at any node selected by the user. If we observe again Algorithms 11, 12, and 13, a moment of thought should convince the reader that their cost is linear with the branching factor of the EF. In contrast, the cost of Algorithm 14 is quadratic with the branching factor of the EF. On the practical side, our experiments reveal that the average cost of a single collapse (considering the 1.675 collapses) is 0,77 msec, and the average cost of a single projection (considering the 1.235 projections) is 17,32 msec. Finally, the average cost for balancing an EF is

2.818,25 msec.

Our algorithm is very conservative because it only collapses or projects nodes that belong to a chain. Our initial experiments showed that if we do not apply any restriction in the use of chains or in the size of them, the results produce EFs that are much more balanced. These results (considering all 23.257 experiments) produced a query reduction of 42%. However, this reduction comes with a cost: the complexity of the questions may be increased. Therefore, we only apply the transformations when the question produced is not complicated (i.e., using chains of at most five changes, see Section 4.2.1). This has produced good results, but sometimes the question of a collapsed/projected node can be still hard to answer. Even in this case, our implementation ensures that if the programmer is able to find the bug with the standard ET, she will also be able with the balanced EF. That is, the introduction of projected nodes cannot cause the debugging session to stop, because our debugger allows the programmer to answer “I don’t know”, skipping the current question and continuing the debugging process with the other questions (e.g., with the children).

4.2.4 Related work

Besides our approach, there exist other transformations devoted to reducing the size of the ET, and thus the number of questions performed. Our implementation allows us to balance an already generated ET, or it allows us to automatically generate the balanced ET. This can be done by collapsing or projecting nodes during their generation. However, conceptually, our technique is a post-ET generation transformation.

The most similar approach is the Tree Compression technique introduced by Dave and Chitil [8]. This approach is also a conservative approach that transforms an ET into an equivalent (smaller) ET where the same bugs can be detected. The objective of this technique is essentially different: it tries to reduce the size of the ET by removing redundant nodes, and it is only applicable to recursive calls.

Another approach which is related to ours was presented in [22], where a transformation for list comprehensions of functional programs was introduced. In this case, it is a source code (rather than an ET) transformation to translate list comprehensions into equivalent functions that implement the iteration. The ET produced can be further transformed to remove the internal nodes of the ET reducing the size of the final ET as in the tree compression technique. Both techniques are orthogonal to the balancing of the ET, thus they both can be applied before balancing.

4.2.5 Proofs of technical results

We present here the proofs of the results of the Balancing technique.
Lemma 1.2.6 (Buggy method). *Given an ET $t = (V, E)$, and a buggy node $v \in V$ in t with $v = (b, m, a)$, then m contains a bug.*

Proof. Because v is buggy, then the method execution (b, m, a) is wrong, thus either $(b, m, a) \notin \mathcal{I}$ or $\mathcal{I} \not\models (b, m, a)$. Moreover, by Definition 1.2.3, we have a child of v for each call to a method done from the definition of m. But we know by Definition 1.2.5 that for all child v' of v, $v' \in \mathcal{I}$ or $\mathcal{I} \models v'$ Hence, m must contain a bug. ■

Proposition 1 *Let t be a EF with a wrong root. Then t contains a buggy node.*

Proof. We prove the claim by induction on the size of t. (Base case) t only contains one node b. Then b is buggy, because it is wrong and it has no children. (Induction hypothesis) t contains i nodes and at least one of them is buggy. (Inductive case) t contains $i + 1$ nodes. In this case we have a tree of i nodes that, by the induction hypothesis, does contain a buggy node b plus one extra node n. If n is not the child of b, then b is buggy. If n is the child of b, then either n is correct, and thus b is buggy; or n is wrong and hence, it is buggy because it has no children. ■

The following lemma ensures that the transformations for collapsing and projecting nodes are sound.

Lemma 4.2.10 (Soundness of projections and collapses) *Let $v = (b, m_1; \ldots; m_n, a)$ be a collapsed or projected node in an EF t. If v is buggy, then it contains a buggy method.*

Proof. We have two possibilities: (1) v is a collapsed node. In this case v has not children, and because v is wrong, $(b, m_1; \ldots; m_n, a) \notin \mathcal{I}$; therefore, trivially, at least one method m_i, $1 \leqslant i \leqslant n$, is buggy. (2) v is a projected node. This case is impossible because a projected node cannot be buggy. The reason is that if all the children of v are correct, then v is correct by Definition 1.2.4 using the inference rule Tr. Otherwise, at least one child is wrong, but then, v cannot be buggy by Definition 1.2.5. ■

Theorem 4.2.6 (Completeness and soundness of EFs). *Given an EF with a wrong root, it contains a buggy node which is associated with a buggy method.*

Proof. The first point is proved by Proposition 1, while the second one is proved by Lemmas 1.2.6 and 4.2.10. ■

Theorem 4.2.7 (Chain Collapse Correctness). *Let $t = (V, E)$ and $t' = (V', E')$ be two EFs, being the root of t wrong, and let $C \subset V$ be a chain such that all nodes in the chain are leaves and they have the same associated method. Given $t' =$ `collapseChain`(t, C),*

1. *t' contains a buggy node.*
2. *Every buggy node in t' is associated with a buggy method.*

Proof. For the first item, only leaf nodes can be collapsed, therefore, the root node could only be collapsed if it is the only node of t. However, even in this case we have that $\nexists v \in V$ such that $(v \to r) \in E$ being r the root of t. Therefore, according to Definition 4.2.5, r is not a chain and thus it cannot be collapsed. Hence, the root of t' is the same as the root of t, and thus t' contains a buggy node by Proposition 1.

Now, we prove that any buggy node of t' is associated with a buggy method. Let $v \in V$ be the parent node of the chain C, and let $w \in V'$ be the collapsed node of C. We consider three cases:

- $u \in V', v \neq u \neq w$ is buggy. In this case the collapse does not influence the buggy node u and thus the claim follows by Lemma 1.2.6.
- v is buggy in t'. This case is trivial, because v is wrong and w is correct by Definition 1.2.5. Therefore, the new node w can be inferred with the rule Tr and thus the method in v is wrong according to Lemma 1.2.6.

  This case is particularly interesting because it reveals a phenomenon: node v is not changed by the transformation and thus it belongs to both trees t and t'. However, it could be possible that v is not buggy in t but it is buggy in t'. This happens because w has somehow hidden some error in the chain—some wrong intermediate result that was visible in the chain is now hidden because only the initial and final contexts are shown—, revealing a new bug located in v.
- w is buggy in t'. Then, either the result or the final context of w are wrong. Hence, since both the result and the final context are produced by the nodes in C, we know that at least one node $c \in C$ is also wrong. Because c is a leaf and it is wrong, then it is a buggy node in t and it is associated with a buggy method m by Lemma 1.2.6. According to Algorithm 11 w is associated with a method execution $(b, m_0 \ldots m \ldots m_n, a)$, and thus it is associated with a buggy method.

■

The proof for projections is more general because it does not require the root of the EF to be buggy, and because it proves that all buggy nodes remain after the projection.

Theorem 4.2.8 (Chain Projection Correctness). *Let $t = (V, E)$ and $t' = (V', E')$ be two EFs, and let $C \subset V$ be a chain such that $t' =$* `projectChain(t,C)`*.*

1. *All buggy nodes in t are also buggy nodes in t'.*
2. *Every buggy node in t' is associated with a buggy method.*

Proof. Let $v \in V$ be the parent node of the chain C, and let $w \in V'$ be the projected node of C. We consider an arbitrary buggy node $u \in V$ and show that it is also buggy in t'. Five cases are possible:

- u is the root of t. It is easy to see that t' has the same root as t, since the node added by `projectChain` requires a parent node in the tree (i.e., the root cannot be projected). Thus, if the root of t is buggy, then the root of t' is also buggy.
- $u \neq v, u \neq w$ and $u \neq c \in C$. In this case the projection does not influence the buggy node u nor its children and thus u is also buggy in t'.
- $u = v$. This means that u is the parent of the chain C. If it is buggy, then by Definition 1.2.5 all nodes in C are correct. Then, as shown in the proof of Lemma 4.2.10, w is correct. Hence, u is also buggy in t'.

  In the general case, v will be buggy in t, and also in t' ($\forall c \in C$, c will be correct; and thus w is also correct). However, it could be possible that v is correct in t, two nodes $c_1, c_2 \in C$ were wrong, and their combined (wrong) effects produced a correct result. In that case, both errors would be hidden in the projection node (but of course they would remain in c_1 and c_2). As a result, a new buggy node (v) not present in t would appear in t'.
- $u = c \in C$. This means that c is wrong and all its children correct. Since the children of c have not been modified by `projectChain`, c was buggy in t and it is also buggy in t'.

In all cases, the buggy node is associated with a buggy method by Lemmas 1.2.6 and 4.2.10. ■

Chapter 5

Conclusions

Declarative debugging is a powerful debugging technique that has been adapted to practically all programming languages. During three decades, Divide & Query has been the more efficient algorithmic debugging strategy. On the practical side, all current algorithmic debuggers implement D&Q [3, 5, 8, 12, 17, 18, 21, 22, 24], and experiments [4, 28] (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.

An important 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).

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.

In addition to providing two new strategies that reduce the number of questions performed by the debugger, we have shown a new arquitecture that allows the programmer to strart the debugging session almost instantly.

As we have mentioned a main problem of the technique is its low level of scalability both in time and memory. With realistic programs the huge size of the internal data structures handled makes the debugging session impractical and too slow to be productive. We have proposed the use of VETs as a suitable solution to these problems. This data structure has two important advantages: It is prepared to be partially stored in main memory, and completely stored in secondary memory. This ensures that it will always fit in main memory and thus solves the memory scalability problem. In addition, it can be used during a debugging session before it is completed. For this, we have implemented a version of standard declarative debugging strategies able to work with VETs. This solves the time scalability problem as demonstrated by our experiments. In our implementation, the programmer can control how much memory is used by the GUI components, and by the strategies thanks to the use of three cache memories. The most important result is that experiments confirm that, even with large programs and long running computations, a debugging session can start to ask questions after only few seconds.

Another contribution of this work is a new technique that allows us to automatically balance standard ETs. This results in fewer questions made by the debugger regardless of the strategy used. This technique has been implemented, and experiments with real applications confirm that it has a positive impact on the performance of declarative debugging. From a theoretical point of view, two important results have been proved. The projection and the collapse of nodes do not prevent finding bugs, and the bugs found after the transformations are always real bugs.

We refer all readers interested in the implementation to the webpage of the DDJ project: `http://users.dsic.upv.es/~jsilva/DDJ`. In this page the reader can see both the source code and the experiments of all the techniques showed during this work.

Chapter 6

Future work

- **Divide by Queries:** In this work we have described the principles that an optimal strategy must follow, and we have introduced an initial version of one of them. This strategy has the property of being compositional. That is, it can be computed during the generation of the ET. However, the excesive amount of time and memory needed with this approach has not been solved yet. The problem is caused by the high number of possible sequences of questions. Then, a main future objective is to reduce the amount of these possible sequences.

- **Loop Expansion:** During the execution of a program a lot of loops can be executed. When an iterative loop (e.g., in the imperative paradigm) is executed, an extremely broad tree is generated. These kinds of trees are more difficult to debug due to their width. In contrast, recursive loops generate deep trees that are easier to debug. If we transform these loops into recursive calls we would obtain trees that are better structured and, thus, they can be easily explored to find the buggy node.

- **Execution Tree Interface:** The debugger presented in this work is a declarative debugger for Java. Consequently, the ET is composed of nodes that contain the information related to the execution of methods. This information is stored in such a way that when it is shown to the user, she is not aware of the programming language used. The debugger can be adapted to another language by only changing the *front-end.* Thus, a different front-end is necessary for each language. Another possibility is to create an Execution Tree Interface that contains, in plain text, all the information stored in the nodes. Thus, if another debugger transforms its ET in this interface we could debug other languages without the need of creating a front-end for each language we want to debug.

- **Aproximating nodes weight:** A weight is usually associated to a node by a strategy. Tipically, this weight represents the amount of descendant nodes, but regardless of their size or their complexity. Moreover, each node is related to the execution of a method, maybe with the same arguments or not. Therefore other strategies assign the same individual weight to all the nodes related to the same method, without having into account that the amount of code executed by each node is different in each case. In order to accurate these weights, it would be interesesting to calculate the percentage of code executed by each node with respect to the number of lines of the method.

- **Eclipse plugin:** DDJ (the declarative debugger designed during this work) inputs the Java classes and the arguments of the main method. Then, DDJ executes all the program and generates the ET that represents all the computations made during the execution. Next, the user answers the questions generated by the debugger, and the debugger considers all the nodes produced during the execution of the program. Normally, a programmer is not interested in debugging all the program, but only the execution of a method and its descendants. We would like to adapt DDJ to an Eclipse plugin in such a way that it will allow the user to select a method and debug only the subtree related to the execution of this method. Of course, if the user is interested in debugging all the program, she only has to select the main method and the ET is generated as usual.

Bibliography

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

[2] D. Binks. *Declarative Debugging in Gödel.* PhD thesis, University of Bristol, 1995.

[3] B. Brael 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.

[4] 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.

[5] R. Caballero, C. Hermanns, and H. Kuchen. 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.

[6] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. A declarative debugger for maude functional modules. *Electronic Notes Theoretical Computer Science*, 238(3):63–81, 2009.

[7] 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.

[8] 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.

[9] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimóthy. Generalized Algorithmic Debugging and Testing. *LOPLAS*, 1(4):303–322, 1992.

[10] H. Girgis and B. Jayaraman. JavaDD: a Declarative Debugger for Java. Technical Report 2006-07, University at Buffalo, March 2006.

[11] 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.

[12] 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.

[13] D. Insa and J. Silva. Debugging with Incomplete and Dynamically Generated Execution Trees. In *Proc. of the 20th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2010)*, Austria, 2010.

[14] G. Kokai, J. Nilson, and C. Niss. GIDTS: A Graphical Programming Environment for Prolog. In *Workshop on Program Analysis For Software Tools and Engineering (PASTE'99)*, pages 95–104. ACM Press, 1999.

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

[16] Yong Luo and Olaf Chitil. Algorithmic debugging and trusted functions. Technical report 10-07, University of Kent, Computing Laboratory, UK, August 2007.

[17] 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.

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

[19] M. Maeji and T. Kanamori. Top-Down Zooming Diagnosis of Logic Programs. Technical Report TR-290, ICOT, Japan, 1987.

[20] Sun Microsystems. Java Platform Debugger Architecture - JPDA, 2010. Available from URL: `http://java.sun.com/javase/technologies/core/toolsapis/jpda/`.

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

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

[23] H. Nilsson and P. Fritzson. Algorithmic Debugging for Lazy Functional Languages. *Journal of Functional Programming*, 4(3):337–370, 1994.

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

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

[26] J. Silva. Algorithmic debugging strategies. In *Proc. of International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006)*, pages 134–140, 2006.

[27] 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.

[28] 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`.

[29] J. Silva and O. Chitil. Combining Algorithmic Debugging and Program Slicing. In *Proc. of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*, pages 157–166. ACM Press, 2006.