Theoretical analysis of git bisect

Julien Courtiel, Paul Dorbec, and Romain Lecoq

Normandie Univ, UNICAEN, ENSICAEN, CNRS, GREYC, 14000 Caen, France.

Abstract

In this paper, we consider the problem of finding a regression in a version control system (VCS), such as `git`. The set of versions is modelled by a Directed Acyclic Graph (DAG) where vertices represent versions of the software, and arcs are the changes between different versions. We assume that somewhere in the DAG, a bug was introduced, which persists in all of its subsequent versions. It is possible to query a vertex to check whether the corresponding version carries the bug. Given a DAG and a bugged vertex, the Regression Search Problem consists in finding the first vertex containing the bug in a minimum number of queries in the worst-case scenario. This problem is known to be NP-complete.

We study the algorithm used in `git` to address this problem, known as `git bisect`. We prove that in a general setting, `git bisect` can use an exponentially larger number of queries than an optimal algorithm. We also consider the restriction where all vertices have indegree at most 2 (i.e. where merges are made between at most two branches at a time in the VCS), and prove that in this case, `git bisect` is a $\frac{1}{\log_2(3/2)}$-approximation algorithm, and that this bound is tight. We also provide a better approximation algorithm for this case.

Finally, we give an alternative proof of the NP-completeness of the Regression Search Problem, via a variation with bounded indegree.

1   Introduction

In the context of software development, it is essential to resort to Version Control Systems (VCS, in short), like `git` or `mercurial`. VCS enable many developers to work concurrently on the same system of files. Notably, all the *versions* of the project (that is to say the different states of the project over time) are saved by the VCS, as well as the different changes between versions.

Furthermore, many VCS offer the possibility of creating *branches* (i.e. parallel lines of development) and *merging* them, so that individuals can work on their own part of the project, with no risk of interfering with other developers work.

1

Thereby the overall structure can be seen as a Directed Acyclic Graph (DAG), where the vertices are the versions, also named in this context *commits*, and the arcs model the changes between two versions.

The current paper deals with a problem often occurring in projects of large size: searching the origin of a so-called *regression*. Even with intensive testing techniques, it seems unavoidable to find out long-standing bugs which have been lying undetected for some time. Conveniently, one tries to fix this bug by finding the commit in which the bug appeared for the first time. The idea is that there should be few differences between the code source of the commit that introduced the bug and the one from a previous bug-free commit, which makes it easier to find and fix the bug.

The identification of the faulty commit is possible by performing *queries* on existing commits. A query allows to figure out the status of the commit: whether it is *bugged* or it is *clean*. A single query can be very time-consuming: it may require running tests, manual checks, or the compilation of an entire source code. In some large projects, performing a query on a single commit can take up to a full day (for example, the Linux kernel project [8]). This is why it is essential to find the commit that introduced the bug with as few queries as possible.

The problem of finding an optimal solution in terms of number of queries, known as the Regression Search Problem, was proved to be NP-complete by Carmo, Donadelli, Kohayakawa and Laber in [6]. However, whenever the DAG is a tree (oriented from the leaves to the root), the computational complexity of the Regression Search Problem is polynomial [3, 14], and even linear [13].

To our knowledge, very few papers in the literature deal with the Regression Search Problem in the worst-case scenario, as such. The Decision Tree problem, which is known to be NP-complete [11] as well as its approximation version [12], somehow generalises the Regression Search Problem, with this difference that the Decision Tree problem aims to minimise the average number of queries instead of the worst-case number of queries.

Many variations of the Regression Search problem exist:

- the costs of the queries may vary [9, 10];

- the queries return the wrong result (say it is clean while the vertex is bugged or the converse) with a certain probability [10];

- one can just try to find a bugged vertex with at least one clean parent [4].

The most popular VCS today, namely `git`, proposes a tool for this problem: an algorithm named `git bisect`. It is a heuristic inspired by binary search that narrows down at each query the range of the possible faulty commits. This algorithm is widely used and shows excellent experimental results, though to our

knowledge, no mathematical study of its performance have been carried out up to now.

In this paper, we fill this gap by providing a careful analysis on the number of queries that `git bisect` uses compared to an optimal strategy. This paper does not aim to find new approaches for the Regression Search Problem.

First, we show in Section 2 that, in the general case, `git bisect` may be very inefficient, testing about half the commits where an optimal logarithmic number of commits can be used to identify exactly the faulty vertex. But in all the cases where such bad performance occurs, there are large merges between more than two branches,[1] also named *octopus merges*. However, such merges are highly uncommon and inadvisable, so we carry out the study of `git bisect` performances with the assumption that the DAG does not contain any octopus merge, that is, every vertex has indegree at most two. Under such an assumption, we are able to prove in Section 3 that `git bisect` is an approximation algorithm for the problem, never using more than $\frac{1}{\log_2(3/2)} \approx 1.71$ times the optimal number of queries for large enough repositories. We also provide a family of DAGs for which the number of queries used by `git bisect` tends to $\frac{1}{\log_2(3/2)}$ times the optimal number of queries.

This paper also describes in Section 4 a new algorithm, which is a refinement of `git bisect`. This new algorithm, which we call `golden bisect`, offers a mathematical guaranteed ratio of $\frac{1}{\log_2(\phi)} \approx 1.44$ for DAGs with indegree at most 2 where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio. The search of new efficient algorithms for the Regression Search Problem seems to be crucial in software engineering (as evidenced by [4]); `golden bisect` is an example of progress in this direction.

The good performances of `git bisect` and `golden bisect` in the binary case raise a last question. Is the problem still NP-complete if the inputs are restricted to binary DAGs? In Section 5, we consider a variation, the Confined Regression Search Problem (CRSP), which is NP-complete even in the binary case. This variation is equivalent to the Regression Search Problem (RSP) in the general case, so this gives a new proof of the complexity of this problem. However, this does not extend to RSP in the binary case.

1.1 Formal definitions

Throughout the paper, we refer to VCS repositories as graphs, and more precisely as *Directed Acyclic Graphs* (DAGs), i.e., directed graphs with no cycle. The set V of vertices corresponds to the versions of the software. An arc goes from a vertex $\boldsymbol{p}$ to another vertex $\boldsymbol{v}$ if $\boldsymbol{v}$ is obtained by a modification from $\boldsymbol{p}$. We then say that $\boldsymbol{p}$ is a *parent* of $\boldsymbol{v}$. A vertex may have multiple parents in the case of a

---

[1]According to this blog, a merge of 66 branches happened in the Linux kernel repository.

merge. An *ancestor* of $\boldsymbol{v}$ is $\boldsymbol{v}$ itself or an ancestor of a parent of $\boldsymbol{v}$.[2] Equivalently, a vertex is an ancestor of $\boldsymbol{v}$ if and only if it is co-accessible from $\boldsymbol{v}$ (i.e., there exists a path from this vertex to $\boldsymbol{v}$).

We use the convention to write vertices in bold (for example $\boldsymbol{v}$), and the number of ancestors of a vertex with its name between two vertical bars (for example $|v|$).

In our DAGs, we consider that a bug has been introduced at some vertex, named the *faulty commit*. This vertex is unique, and its position is unknown. The faulty commit is supposed to transmit the bug to each of its *descendants* (that is, its children, its grand-children, and so on). Thus, vertices have two possible statuses: *bugged* or *clean*. A vertex is bugged if and only if it has the faulty commit as an ancestor. Other vertices are clean. This is illustrated by Figure 1.
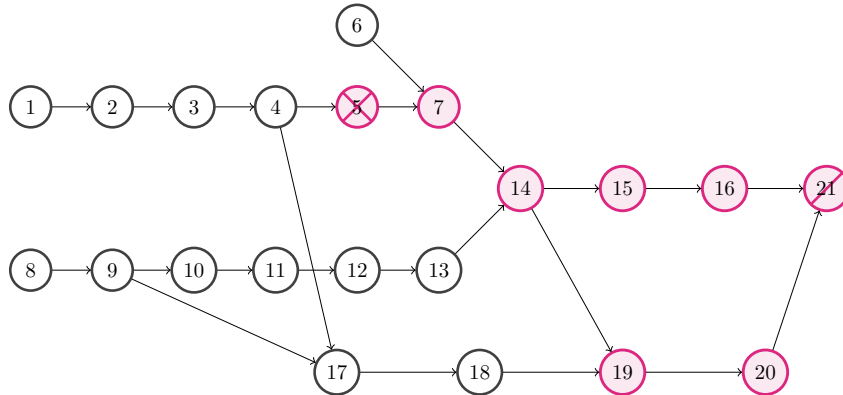

Figure 1: An example of a DAG. The bugged vertices are coloured. The strikeout vertex (**21**) is the marked vertex, known to be bugged. The crossed vertex (**5**) is the faulty commit.

We consider the problem of identifying the faulty commit in a DAG D, where a bugged vertex $\boldsymbol{b}$ is identified. It is addressed by performing *queries* on vertices of the graph. Each query states whether the vertex is bugged or clean, and thus whether or not the faulty commit belongs to its ancestors or not. Once we find a bugged vertex whose parents are all clean, it is the faulty commit.

The aim of the Regression Search Problem is to design a strategy for finding the faulty commit in a minimal number of queries.

Formally, a *strategy* (see for example [7]) for a DAG D is a binary tree S where the nodes are labelled by the vertices of D. Inner nodes of S represent queries. The root of S is the first performed query. If the queried vertex is

---

[2]Usually, $\boldsymbol{v}$ is not considered an ancestor of itself. Though, for simplifying the terminology, we use this special convention here.

bugged, then the following strategy is given by the left subtree. If it is clean, the strategy continues on the right subtree. Whenever the subtree is reduced to a leaf, a single candidate remains. The label of the leaf gives the only possible faulty commit.

For example, Figure 2 shows a strategy tree for a directed path of size 5, where the identified bugged vertex is the last one. Suppose that the faulty commit is **4**. In this strategy, we first query **2**. Since it is clean, we query next **4**, which appears to be bugged. We finally query **3**: since it is clean, we infer that the faulty commit is **4**. We have found the faulty commit with 3 queries. Remark that if the faulty commit was **1**, **2** or **5**, the strategy would use only 2 queries.

Figure 2: *Left.* A directed path on 5 vertices. *Right.* A possible strategy for the Regression Search Problem on the path on 5 vertices.

For a given strategy, the *number of queries in the worst-case scenario* corresponds to the height of the tree. In the above example, this number is 3, occurring when the faulty commit is **3** or **4**.

The Regression Search Problem is formally defined as follows.

Definition 1. *Regression Search Problem.*
Input. A DAG D, a marked vertex b known to be bugged, and an integer k.
Output. Whether there is a strategy that finds the faulty commit in at most k queries in the worst-case scenario.

Since the faulty commit is necessarily an ancestor of b, it is convenient to directly study the induced subgraph on b's ancestors. In this case, b is a *sink* (i.e. a vertex with no outgoing edge) accessible from all vertices in the DAG. Thus, when the bugged vertex is not specified, it is assumed to be the only sink of the DAG.

In the following, *optimal strategies* are strategies that use the least number of queries in the worst case scenario. For example, if the input DAG is a directed path of size n, an optimal strategy uses $\lceil \log_2(n) \rceil$ queries in the worst-case scenario. Indeed, a simple binary search enables to remove half of the vertices at each query.

A second interesting example is what we refer to as an *octopus*. In this digraph, there is a single sink and all other vertices are parent of the sink (see
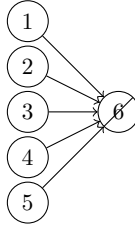
Figure 3: An octopus of size 6.

Figure 3). When the faulty commit is the sink, we must query all other vertices to make sure that the sink is faulty, regardless of the strategy. Thus, any optimal strategy uses $n - 1$ queries in the worst-case scenario.

These two examples actually constitute extreme cases for the Regression Search Problem, as shown by the following proposition.

Proposition 2. *For any DAG D where the marked bugged vertex has n ancestors, an optimal strategy that finds the faulty commit uses at least $\lceil \log_2(n) \rceil$ queries, and at most $n - 1$ queries.*

Proof. Remember that a strategy is a binary tree with at least n leaves, and the number of queries in the worst-case scenario corresponds to the height of the tree. But the height of such a binary tree is necessarily at least $\lceil \log_2(n) \rceil$, which proves the lower bound.

As for the upper bound, it is sufficient to query the $n - 1$ ancestors of the marked bugged vertex to identify the faulty commit. $\square$

From a complexity point of view, the Regression Search Problem is hard: Carmo, Donadelli, Kohayakawa and Laber proved in [6] that the Regression Search Problem is NP-complete.[3] We also provide in Section 5 an alternative proof of its NP-completeness (see Corollary 28).

1.2 Description of `git bisect`

As said in the introduction, some VCS provide a tool for the Regression Search Problem. The most known tool is `git bisect`, but an equivalent exists in `mercurial` (`hg bisect` [5]).

---

[3]In reality, the problem they studied has an extra restriction: a query cannot be performed on a vertex which was eliminated from the set of candidates for the faulty commit (which occurs for example when an ancestor is known to be bugged). However, the gadget they used in the proof of NP-completeness also works for our problem where we do not necessarily forbid such queries.

The algorithm `git bisect` is a greedy algorithm based on the classical binary search. At each step, it keeps only the subgraph where the faulty commit lies and queries the vertex that split the digraph in the most balanced way.

To be more precise, let us define the notion of *score*.

Definition 3 (Score). *Given a DAG D with n vertices, the score of a vertex $\boldsymbol{x}$ of D is designated by $\mathsf{score}_D(\boldsymbol{x})$ or just $\mathsf{score}(\boldsymbol{x})$ if there is no ambiguity. Its value is*

$$\min(|x|, n - |x|),$$

where $|x|$ is the number of ancestors of $\boldsymbol{x}$ (recall that $\boldsymbol{x}$ is an ancestor of itself).
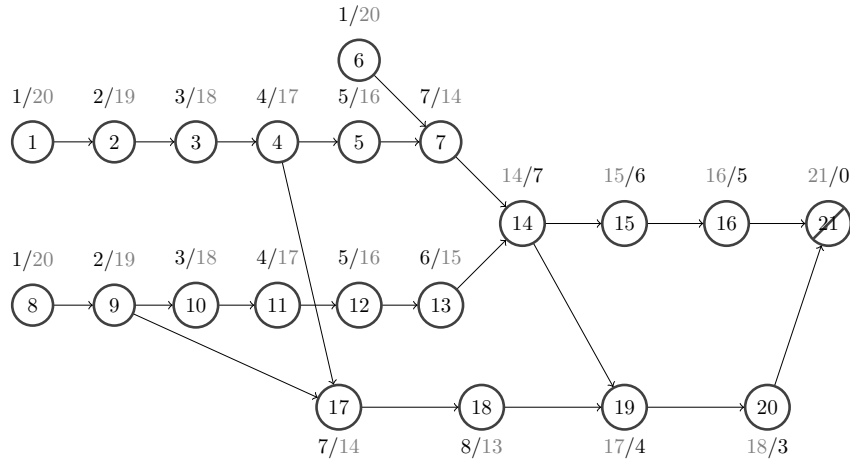

Figure 4: The notation a/b along each vertex indicates that a is the number of ancestors of the vertex, and b is the number of non-ancestors. The score (see Definition 3) is displayed in black.

For example, let us refer to Figure 4: vertex **5** has 5 ancestors (**1**, **2**, **3**, **4** and **5**). So $\mathsf{score}(\boldsymbol{5}) = \min(5, 21 - 5) = 5$.

If vertex $\boldsymbol{x}$ is queried and appears to be bugged, then there remain $|x|$ candidates for the faulty commit: the ancestors of $\boldsymbol{x}$. If the query of $\boldsymbol{x}$ reveals on the contrary that it is clean, then the number of candidates for the faulty commit is $n - |x|$, which is the number of non-ancestors. This is why the score of $\boldsymbol{x}$ can be interpreted as the least number of vertices to be eliminated from the set of possible candidates for the faulty commit, when $\boldsymbol{x}$ is queried. For a DAG, each vertex has a score and the maximum score is the score with the maximum value among all.

A detailed description of `git bisect` is given by Algorithm 1.

Input. A DAG D and a bugged vertex $\boldsymbol{b}$.
Output. The faulty commit of D.
Steps:

1. Remove from D all non-ancestors of $\boldsymbol{b}$.

2. If D has only one vertex, return this vertex.

3. Compute the score for each vertex of D.

4. Query the vertex with the maximum score. If there are several vertices which have the maximum score, select any one then query it.

5. If the queried vertex is bugged, remove from D all non-ancestors of the queried vertex. Otherwise, remove from D all ancestors of the queried vertex.

6. Go to Step 2.

Algorithm 1: `git bisect`

As an example of an execution, consider the DAG from Figure 4. Vertex **18** has the maximum score ($\mathsf{score}(\mathbf{18}) = 8$) so constitutes the first vertex to be queried. If we assume that the faulty commit is **5**, then the query reveals that **18** is clean. So all ancestors of **18** are removed (that are $\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{8}, \mathbf{9}, \mathbf{17}, \mathbf{18}$). Vertex **14** is then queried because it has the new maximum score 5, and so on.
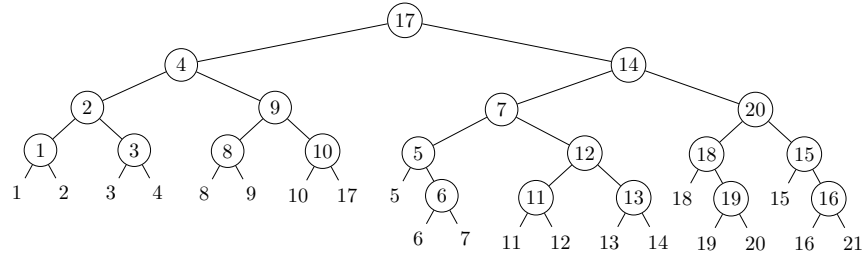

Figure 5: The `git bisect` strategy corresponding to the graph of Figure 4. In case of score equality, the convention we choose consists in querying the vertex with the smallest label.

The whole `git bisect` strategy tree is shown in Figure 5.

Notice that for this DAG, the `git bisect` algorithm uses 6 queries in the worst-case scenario, which is not optimal for this *well-chosen* example as we are going to see later.

The greedy idea behind `git bisect` (choosing the query which partitions the

commits as evenly as possible) is quite widespread in the literature. For example, it was used to find a $(\log(n) + 1)$-approximation for the Decision Tree Problem [1], in particular within the framework of geometric models [2].

2 Worst-case number of queries

This section addresses the complexity analysis of `git bisect` in the worst-case scenario.

In Sections 2, 3 and 4, we consider algorithms that prune all non-ancestors of the marked vertex b. Therefore, all results are stated for DAGs that have one sink, which is the marked vertex, and for which the number of vertices n is also the number of candidates for the faulty commit.

2.1 The comb construction

We describe in this subsection a way to enhance any DAG in such a way the Regression Search Problem can always be solved in a logarithmic number of queries.

Definition 4 (Comb addition). *Let D be a Directed Acyclic Graph with n vertices. Let $v_1 < v_2 < \ldots < v_n$ be a topological ordering of D, that is a linear ordering of the vertices such that if $v_i \to v_j$ is an arc, then $v_i < v_j$.*

We say that we add a comb to D if we add to D:

- *n new vertices $u_1, \ldots, u_n$;*

- *the arcs $v_i \to u_i$ for $i \in \{1, \ldots, n\}$;*

- *the arcs $u_i \to u_{i+1}$ for $i \in \{1, \ldots, n-1\}$.*

The resulting graph is designated by $comb(D)$. The new identified bugged vertex of $comb(D)$ is u_n.

An example of comb addition is shown by Figure 6.

The comb addition depends on the initial topological ordering, but the latter will not have any impact on the following results. This is why we take the liberty of writing $comb(D)$ without any mention of the topological ordering.

Theorem 5. *Let D be a Directed Acyclic Graph with n vertices and such that the number of queries used by the `git bisect` algorithm is x. If we add a comb to D, then the resulting DAG $comb(D)$ is such that:*

- *the optimal strategy uses only $\lceil \log_2(2n) \rceil$ queries;*

- *when n is odd, the `git bisect` algorithm uses $x + 1$ queries.*
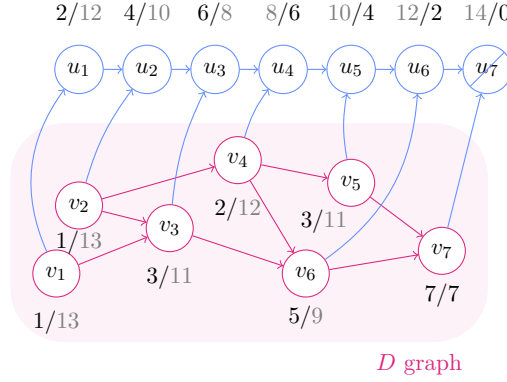
Figure 6: Illustration of the comb addition. The initial digraph is highlighted in pink.

Proof idea On one hand, the optimal strategy for $comb(D)$ can be naturally achieved with a binary search on the u_i vertices. On the other hand, with the assumption that n is odd, `git bisect` will necessarily query v_n first since its score is n, and the scores of the u_i vertices are all even. This explains why the `git bisect` algorithm uses $x + 1$ queries.

Detailed proof. We keep the same notation as Definition 4. For a DAG D and a subset of vertices $X \subseteq V$, the *induced subgraph* of D on X, denoted by $D[X]$, is the digraph with vertex set X, and with an arc from vertex u to vertex v if and only if the corresponding arc is in D.

Claim 5.1. *For all i, u_i has $2i$ ancestors, which are all the vertices u_j and v_j with $j \leq i$. The ancestors of v_i do not change.*

Observe first that no v_i is the head of an arc added in $comb(D)$. Inductively, we infer that the ancestors of v_i do not change.

As for u_i, we prove the claim by induction. Indeed, vertex u_i has two parents which are u_{i-1} and v_i. By induction hypothesis, we can see that all the vertices u_j and v_j with $j < i$ are ancestors of u_i since they are the ancestors of u_{i-1}. Moreover all ancestors v_j of v_i satisfy $j \leq i$ (by topological ordering). Consequently u_i has $2i$ ancestors: itself, v_i and all the ancestors of u_{i-1}.

Claim 5.2. *The optimal number of queries is $\lceil \log_2(2n) \rceil$ for $comb(D)$.*

Let us prove this claim for every digraph D by induction on the number n of vertices of D.

The case $n = 1$ is obvious: if D has only 1 vertex, we query $\boldsymbol{v_1}$ to know whether $\boldsymbol{u_1}$ or $\boldsymbol{v_1}$ is the faulty commit. The number of queries is then $\lceil \log_2(2 \times 1) \rceil = 1$.

Now fix $n > 1$ and let us assume that the claim holds for every digraph D of size smaller than n. We choose as the first query the vertex $\boldsymbol{u_i}$ where $i = \lceil \frac{n}{2} \rceil$.

Depending on whether $\boldsymbol{u_i}$ is bugged or clean, the digraph after this query is either $comb(D)[\boldsymbol{u_1}, \ldots, \boldsymbol{u_i}, \boldsymbol{v_1}, \ldots, \boldsymbol{v_i}]$ or $comb(D)[\boldsymbol{u_{i+1}}, \ldots, \boldsymbol{u_n}, \boldsymbol{v_{i+1}}, \ldots, \boldsymbol{v_n}]$.

Notice that in any case, the resulting digraph is of the form $comb(D')$. Indeed, we just have to choose $D' := D[\boldsymbol{v_1}, \ldots, \boldsymbol{v_i}]$ or $D' := D[\boldsymbol{v_{i+1}}, \ldots, \boldsymbol{v_n}]$, and keep the same topological ordering.

Now we can use the induction hypothesis on $comb(D')$, which has at most $\lceil \frac{n}{2} \rceil$ vertices: we can find a strategy in at most $\lceil \log_2(2 \lceil \frac{n}{2} \rceil) \rceil$ queries to find the faulty commit in $comb(D')$.

The overall number of queries for $comb(D)$ with this strategy is then at most $1 + \lceil \log_2 \left(2 \lceil \frac{n}{2} \rceil \right) \rceil$, which is equal to $\lceil \log_2(2n) \rceil$ whenever $n \geq 1$. By Proposition 2, a strategy with this number of queries must be optimal.

Claim 5.3. *If n is odd, the* `git bisect` *algorithm necessarily uses $x + 1$ queries.*

By Claim 5.1, $\boldsymbol{v_n}$ has n ancestors, and digraph $comb(D)$ has $2n$ vertices. So $\mathsf{score}(\boldsymbol{v_n}) = n$ (hence maximal).

Vertex $\boldsymbol{v_n}$ is the only one to have a maximal score. Indeed, on the one hand, any vertex of the form $\boldsymbol{v_i}$ with $i < n$ has fewer than n ancestors. On the other hand, $\boldsymbol{u_i}$ having $2i$ ancestors, its score must be even, and therefore cannot be maximal if n is odd.

Thus the `git bisect` algorithm is going to choose $\boldsymbol{v_n}$ as first query. If this vertex turns out to be clean, it remains a directed path of length n, inducing $\lceil \log_2(n) \rceil$ `git bisect` queries. If $\boldsymbol{v_n}$ is bugged, then the resulting graph is D, for which the worst-case number of `git bisect` queries is x. Therefore, since $x \geq \lceil \log_2(n) \rceil$ by Proposition 2, the number of `git bisect` queries for $comb(D)$ in the worst-case scenario is $x + 1$. $\qquad\square$

If the initial number of vertices n is even, there is no guarantee that `git bisect` will perform $x + 1$ queries on $comb(D)$ – it depends on whether the first queried vertex is $\boldsymbol{v_n}$ or $\boldsymbol{u_{n/2}}$.

However a referee rightly mentioned that the odd hypothesis could be (almost) removed by tweaking the comb construction whenever n is even. Indeed, by deleting the edge from $\boldsymbol{v_{n/2}}$ to $\boldsymbol{u_{n/2}}$, `git bisect` is forced to use $x + 1$ queries in the worst-case scenario, while the following strategy uses $\lceil \log_2(2n) + 1 \rceil$ queries : run a binary search on the path formed by vertices $\boldsymbol{u_1}, \ldots, \boldsymbol{u_n}$, then query all remaining parents of the identified vertex $\boldsymbol{u_i}$ (that is possibly zero, two or one parents depending on whether the identified vertex is respectively $\boldsymbol{u_{n/2}}$, $\boldsymbol{u_{n/2+1}}$ or any other $\boldsymbol{u_i}$) .

2.2 A pathological example for `git bisect`

The following corollary shows the existence of digraphs for which the `git bisect` algorithm totally fails. The optimal number of queries is linear, while the `git bisect` algorithm effectively uses an exponential number of queries.

Theorem 6. *For any integer $k > 2$, there exists a DAG such that the optimal number of queries is k, while the **git bisect** algorithm uses $2^{k-1} - 1$ queries in the worst-case scenario.*
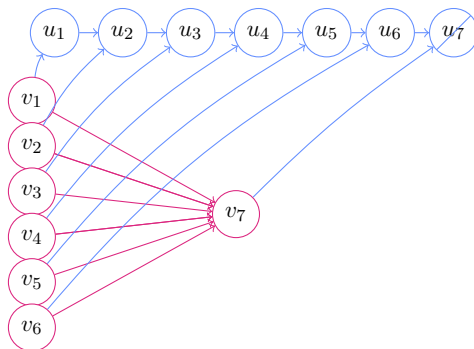

Figure 7: $Comb(D)$ graph where D is an octopus of size 7.

Proof. Choose D as an octopus with $2^{k-1} - 1$ vertices. The number of `git bisect` queries in D is $2^{k-1} - 2$ in the worst-case scenario. The wanted digraph is then $comb(D)$ (see Figure 7 for an illustration). Indeed, by Theorem 5, the `git bisect` algorithm uses $2^{k-1} - 1$ `git bisect` queries to find the faulty commit in $comb(D)$, while an optimal strategy uses $\lceil \log_2 (2^k - 2) \rceil = k$ queries. $\square$

This also shows that the `git bisect` algorithm is not a C-approximation algorithm for the Regression Search Problem, for any constant C.

3 Approximation ratio for binary DAGs

3.1 Results

The pathological input for the `git bisect` algorithm has a very particular shape (see Figure 7): it involves a vertex with a gigantic indegree. However, in the context of VCS, this structure is quite rare. It means that many branches have been merged at the same time (the famous *octopus merge*). Such an operation is strongly discouraged, in addition to the fact that we just showed that `git bisect` becomes inefficient in this situation.

This motivates to define a new family of DAGs, closer to reality:

Definition 7 (Binary digraph)**.** *A digraph is* binary *if each vertex has indegree (that is, the number of ingoing edges) at most* 2*.*

Figure 8: *Left.* A binary DAG. *Right.* A non-binary DAG.

Figure 8 illustrates this definition. If we restrict the DAG to be binary, `git bisect` proves to be efficient.

Theorem 8. *On any binary DAG with n vertices, the number of queries of the* `git bisect` *algorithm is at most* $\frac{\log_2(n)}{\log_2(3/2)}$*.*

Corollary 9. *The algorithm* `git bisect` *is a* $\frac{1}{\log_2(3/2)} \approx 1.71$*-approximation algorithm on binary DAGs.*

3.2   Bounding the number of queries

The key ingredient of the proof lies in the next lemma, which exhibits a core property of binary DAGs. It states that if the DAG is binary, there must be a vertex with a "good" score, i.e., that removes at least approximately one third of the remaining vertices at each query. The overall number of queries is then equal to $\log_{3/2}(n)$.

Lemma 10. *In every binary DAG with n vertices, there exists a vertex $\boldsymbol{v}$ whose number of ancestors, $|v|$, satisfies the double inequality $\frac{n}{3} \leq |v| \leq \frac{2n+1}{3}$.*

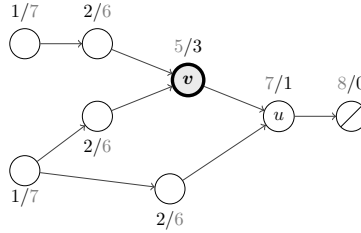The reader can look at Figure 9 for an illustrative example.

Figure 9: The highlighted vertex $\boldsymbol{v}$ is the only one to have its number of ancestors in $[\frac{n}{3}, \frac{2n+1}{3}]$, where $n = 8$.

Proof. [4] If $n = 1$, the only vertex $\boldsymbol{v}$ of the DAG has $|v| = 1$, which satisfies the bound. Then, if $n \geq 2$, let $\boldsymbol{u}$ be a vertex such that $|u| \geq (2n + 2)/3$, chosen so that $|u|$ is as small as possible. Since $|u| \geq 2$, the vertex $\boldsymbol{u}$ has one or two parents. Let $\boldsymbol{v}$ be the parent of $\boldsymbol{u}$ with the most ancestors.

Since $|v| < |u|$ and, by minimality of $|u|$, we have $|v| \leq (2n + 1)/3$. Furthermore, at least half of the $|u| - 1$ strict ancestors of $\boldsymbol{u}$ must be ancestors of $\boldsymbol{v}$. It follows that $|v| \geq \frac{(|u|-1)}{2} \geq \frac{(2n-1)}{6} > \frac{(n-1)}{3}$, i.e., that $|v| \geq \frac{n}{3}$. $\qquad\square$

This lemma is sufficient to prove the logarithmic upper bound for the number of `git bisect` queries.

Proof of Theorem 8. Let D be a DAG with n vertices, and D_k the digraph obtained from D after k `git bisect` queries. Let n_k be the number of vertices in D_k. After each query, the `git bisect` algorithm chooses the vertex $\boldsymbol{v}$ given by Lemma 10 or a vertex with a better score. In any case, the score of the chosen vertex in D_k is greater or equal than $\dfrac{n_k - 1}{3}$. This is why

$$n_{k+1} \leq \frac{2n_k + 1}{3}. \tag{1}$$

We can then show by induction that

$$n_k \leq 1 + \left(\frac{2}{3}\right)^k (n - 1). \tag{2}$$

We distinguish two cases from here.

1. Case $\boldsymbol{n \leq 8}$. We can check the small cases by repetitively using Inequality (1) and keeping the integral part of the right member of the inequality (since we work with integers), thereby obtaining an upper bound $F(n)$ on the least integer k for which $n_k = 1$. For example, if $n = n_0 = 5$, we see that $n_1 \leq \frac{11}{3}$, hence $n_1 \leq 3$. Then $n_2 \leq \frac{7}{3}$ and so $n_2 \leq 2$, and finally $n_3 \leq 1$, which means that the number of queries for a DAG of size 5 is at most $F(5) = 3$. The first values of this upper bound are listed in Table 1. We remark that this is consistent with the $\log_{3/2}(n)$ bound of Theorem 8.

2. Case $\boldsymbol{n \geq 9}$. Let x be the largest number of queries such that $n_x \geq 9$. This means that after $x + 1$ queries, the DAG will have at most 8 vertices and by Table 1, we see that a maximum of 4 extra queries can be required to find the faulty commit from this point. Therefore, the number of `git bisect` queries for D is at most $x + 5$.

Setting $k = x$ in Inequality (2) shows that $8 \leq \left(\frac{2}{3}\right)^x (n - 1)$, hence $x + \log_{\frac{3}{2}}(8) \leq \log_{\frac{3}{2}}(n - 1)$. Since $\log_{3/2}(8) \simeq 5.13$, we see that the number of `git bisect` queries of D is indeed bounded by $x + 5 \leq \log_{\frac{3}{2}}(n - 1)$. $\qquad\square$

---

[4]The authors wish to thank the referee who suggested this more condensed proof.

n	1	2	3	4	5	6	7	8	9	10	11	12	13
Upper bound $F(n)$	0	1	2	3	3	4	4	4	5	5	5	5	6
Approximation for $\log_{\frac{3}{2}}(n)$	0	1.71	2.71	3.42	3.97	4.42	4.80	5.13	5.42	5.68	5.91	6.12	6.33
Approximation for $\log_{\phi}(n) + 1$	1	2.44	3.28	3.88	4.34	4.72	5.04	5.32	5.56	5.78	5.98	6.16	6.33

Table 1: Checking Theorem 8 and Theorem 17 for small sizes

3.3 Tight case

The upper bound of Theorem 8 is asymptotically sharp, as stated by the following proposition.

Proposition 11. *For any integer $k \geq 1$, there exists a* binary *DAG J_k such that*

- *the number of `git bisect` queries on $comb(J_k)$ is $k + \lceil \log_2(k+1) \rceil + 3$;*

- *an optimal strategy for $comb(J_k)$ uses at most $\log_2(\frac{3}{2}) k + \log_2(3k + 7) + 2$ queries.*

(Remember that the comb operation is described by Definition 4.)

Figure 11d shows what J_k looks like for $k = 3$. For this example, `git bisect` uses 7 queries in the worst-case scenario (which occurs for example when c is bugged).

Proof. We first describe a family J_k of graphs that fulfil the properties of Proposition 11.

We start by defining J_k^0, the *backbone* of J_k. It is formed by taking three directed paths on $k + 1$ vertices $x_1 \to x_2 \to \cdots \to x_{k+1}$, $y_1 \to \cdots \to y_{k+1}$ and $z_0 \to \cdots \to z_k$ and merging the three vertices x_{k+1}, y_{k+1} and z_0 into a vertex c (see Figure 11a for an example with $k = 3$).

We construct our final graph J_k from its backbone through $k + 1$ successive digraphs: $J_k^0, J_k^1, \ldots, J_k^k$. For each d starting from 1 to k, let us define

$$\ell_d = \left\lfloor \frac{n_{d-1}}{6} + 1 \right\rfloor$$

where n_{d-1} stands for the number of vertices in J_k^{d-1}. Add a directed path on ℓ_d vertices towards each backbone vertex at distance d from c, namely x_{k+1-d}, y_{k+1-d}, and z_d. Then, add edges from the new parents of x_{k+1-d} and of y_{k+1-d} to the first vertex of the path newly attached to z_d. Also, the new parent of z_d is denoted by z_d'. The reader can refer to Figure 10 for a better understanding of the notation.
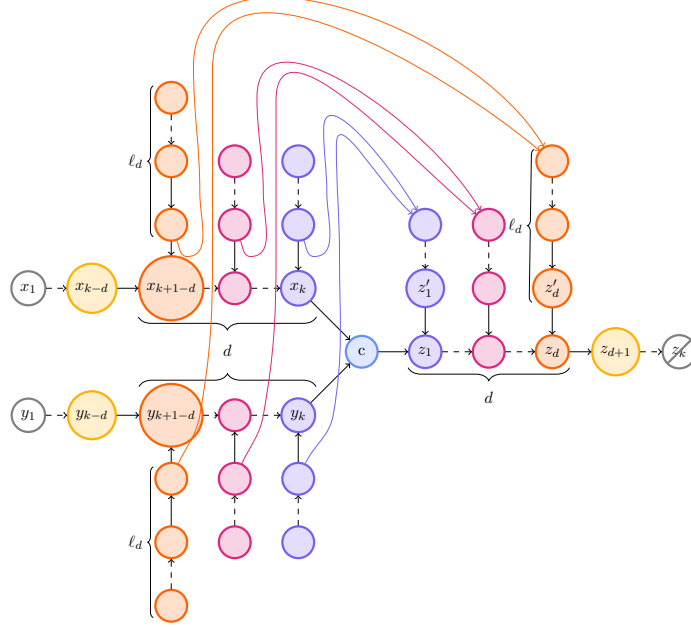
15

Figure 10: the d-th step in the construction of J_k.

We wish the number of vertices in the final graph J_k to be odd in order to use Theorem 5. If J_k^k has an odd number of vertices, then we keep the digraph as such. If this number turns to be even, we just replace ℓ_k by $\ell_k + 1$ in the last step, which increases the number of vertices by 3, and so makes it odd. The resulting digraph is denoted by J_k .

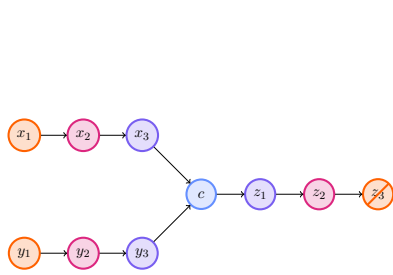The construction for $k = 3$ is shown in Figure 11.

Number of vertices in the final digraph. For each $d > 1$, the number of vertices n_d satisfies the inequality

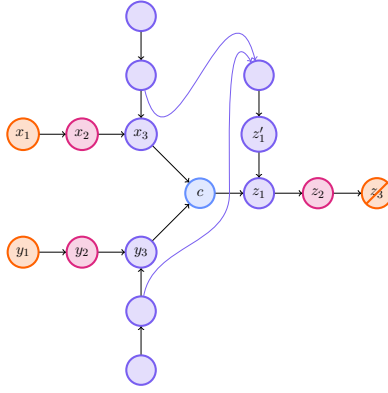$$n_d = n_{d-1} + 3\ell_d \leq \frac{3}{2}n_{d-1} + 3.$$

A quick induction shows that

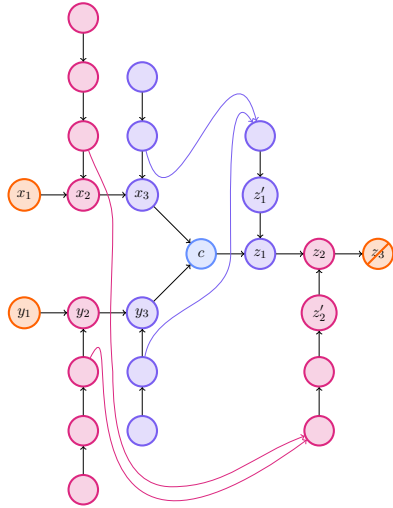$$n_d \leq \left(\frac{3}{2}\right)^d (3k + 7) - 6.$$

Remember that, if n_k is even, we have added 3 vertices in the final digraph. In any case, the number of vertices in J_k is bounded by $\left(\frac{3}{2}\right)^d (3k + 7)$.
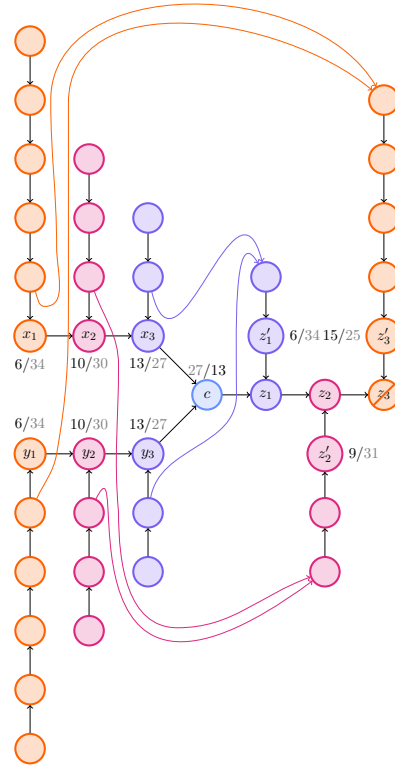
16

(a) Step 0: J_3^0

(b) Step 1: J_3^1

(c) Step 2: J_3^2

(d) Step 3: J_3

Figure 11: Construction of J_3.

17

Claim 11.1. *The sequence $(\ell_d)_{d \in \{1,\dots,k\}}$ is strictly increasing.*

The sequence is reduced to one element for $k = 1$, so we can assume that $k \geq 2$. Let $1 \leq d < k$. By definition of ℓ_d, we have $\ell_{d+1} > \frac{n_d}{6}$ and $\ell_d \leq \frac{n_{d-1}}{6} + 1$. So

$$\ell_{d+1} - \ell_d > \frac{n_d}{6} - \frac{n_{d-1}}{6} - 1 = \frac{\ell_d}{2} - 1.$$

But

$$\ell_d > \frac{n_{d-1}}{6} \geq \frac{n_0}{6} = \frac{3k+1}{6} \geq 2,$$

whenever $k \geq 2$. We conclude from the former that $\ell_{d+1} - \ell_d > 0$.

Claim 11.2. *When c is the faulty commit, the `git bisect` algorithm uses $k + \lceil \log_2(k+1) \rceil + 2$ queries on J_k.*

We are going to show that the resulting digraph just after the i-th step of the `git bisect` algorithm is J_k^{k-i}, for $i \in \{0, \dots, k\}$. In other words, after k `git bisect` queries, we end up with the backbone J_k^0. After we show this fact, the claim is easily proved. Indeed, two extra queries from J_k^0 lead to a binary search on a directed path with $k + 1$ vertices, for which `git bisect` uses $\lceil \log_2(k+1) \rceil$ queries to find the faulty commit. This explains why the number of `git bisect` queries is $k + 2 + \lceil \log_2(k+1) \rceil$. To do so, we prove by induction on d the construction invariants in J_k^d:

- $\mathsf{score}(\boldsymbol{x_k}) = |x_k| = \frac{n_{d-1}}{3}$

- $\mathsf{score}(\boldsymbol{y_k}) = |y_k| = \frac{n_{d-1}}{3}$

- $\mathsf{score}(\boldsymbol{c}) = n_d - |c| = \frac{n_{d-1}}{3}$

- for all $1 \leq i \leq d$, $\mathsf{score}(\boldsymbol{z_i'}) = |z_i'| = 3\ell_i$

All these properties clearly hold for $d = 0$.

Let us assume now the induction hypotheses for $d - 1$. By construction, the number of ancestors of $\boldsymbol{x_k}$ in J_k^d increases by ℓ_d in comparison with its number in J_k^{d-1}, while its number of non-ancestors increases by $2\ell_d$. It is the same for $\boldsymbol{y_k}$. As for $\boldsymbol{c}$, its number of ancestors increases by $2\ell_d$, while its number of non-ancestors increases by ℓ_d. From these observations, we inductively infer the first three invariants. As for the last item, it is obvious that $|z_i'| = 3\ell_i$ by construction. The number of non-ancestors of $\boldsymbol{z_i'}$ is $n_d - 3\ell_i$, which is at least $n_d - 3\ell_d = n_{d-1}$ by Claim 11.1. For $k = 1$, we have $n_0 = 4 > 3 = 3\ell_1$. For $k \geq 2$, we saw in the proof of Claim 11.1 that $\ell_i \geq 2$, hence

$$n_{d-1} \geq 6(\ell_d - 1) \geq 6(\ell_i - 1) \geq 3\ell_i.$$

The score of $\boldsymbol{z_i'}$ is thus $|z_i'| = 3\ell_i$.

Now, let us suppose that the digraph just before the i-th step is J_k^m, where $m = k-i+1$, and let us show that after the i-th step, the digraph becomes J_k^{m-1}. To do so, we have to investigate the scores of all vertices in J_k^m. By construction, each vertex is either an ancestor of $\boldsymbol{x_k}$, an ancestor of $\boldsymbol{y_k}$, a descendant of $\boldsymbol{c}$, or an ancestor of a vertex $\boldsymbol{z'_j}$ with $j \in \{1, \ldots, m\}$. The vertex $\boldsymbol{x_k}$ having fewer ancestors than non-ancestors, the ancestors of $\boldsymbol{x_k}$ different from $\boldsymbol{x_k}$ have a smaller score than $\boldsymbol{x_k}$. Thus the `git bisect` algorithm never queries an ancestor of $\boldsymbol{x_k}$ different from $\boldsymbol{x_k}$. Similarly, we can eliminate every other vertex, excepted $\boldsymbol{x_k}$, $\boldsymbol{y_k}$, $\boldsymbol{c}$ and $\boldsymbol{z'_j}$ with $j \in \{1, \ldots, m\}$.

We already saw that the scores of $\boldsymbol{x_k}$, $\boldsymbol{y_k}$ and $\boldsymbol{c}$ are the same and bounded by $\frac{n_m}{3}$. As for the vertex $\boldsymbol{z'_j}$, its score is equal to $3\ell_j$. Since ℓ_j is strictly increasing by Claim 11.1, we can eliminate every vertex $\boldsymbol{z'_j}$ for $j < m$. It remains to compute the score of $\boldsymbol{z'_m}$. Remark that $6\ell_d > n_{d-1}$ by the definition of ℓ_d. We deduce that

$$n_m = n_{m-1} + 3\ell_m < 9\ell_m.$$

But $\mathsf{score}(\boldsymbol{z'_m}) = |z'_m| = 3\ell_m$, which is bigger than $\dfrac{n_m}{3}$ by the above inequality.

So $\boldsymbol{z'_m}$ is the only vertex with a maximal score; the `git bisect` algorithm will query this vertex. Since $\boldsymbol{c}$ is not an ancestor of $\boldsymbol{z'_m}$, `git bisect` will remove every ancestor of $\boldsymbol{z'_m}$: we recover J_k^{m-1}.

Conclusion. By Theorem 5 and Claim 11.2, the `git bisect` algorithm uses $k + \lceil \log_2(k+1) \rceil + 2 + 1$ queries on $Comb(J_k)$. Also by Theorem 5, since the number of vertices in J_k is odd and is bounded by $\left(\dfrac{3}{2}\right)^k (3k + 7)$, the optimal strategy uses at most $\left\lceil \log_2 \left(2 \left(\dfrac{3}{2}\right)^k (3k+7) \right) \right\rceil \leq \log_2 \left(\frac{3}{2}\right) k + \log_2(3k+7) + 2$ queries. This concludes the proof of Proposition 11. $\qquad\square$

By Proposition 11, we cannot find a better approximation ratio than $\frac{1}{\log_2(3/2)}$ for `git bisect`.

Corollary 12. *For any $\varepsilon > 0$, the **git bisect** algorithm is not a $\left(\frac{1}{\log_2(3/2)} - \varepsilon \right)$- approximation algorithm for binary DAGs.*

3.4 Generalisation for Δ-ary DAGs

For any $\Delta \geq 1$, a DAG is said to be Δ-*ary* if each of its vertices has indegree at most equal to Δ. It is worth noting that the results for binary DAGs can be naturally extended to Δ-ary DAGs.

Indeed, Lemma 10, which is of paramount importance to understand the structure of binary DAGs, can be generalised as follows.

Lemma 13. *In every Δ-ary DAG with n vertices, there exists a vertex $\boldsymbol{v}$ such that $|v|$, its number of ancestors, satisfies $\frac{n-1}{\Delta+1} < |v| \le \frac{\Delta n+1}{\Delta+1}$.*

This leads to the following theorem.

Theorem 14. *On any Δ-ary DAG with n vertices, the number of queries of the* `git bisect` *algorithm is at most $\frac{\log_2(n)}{\log_2\left(\frac{\Delta+1}{\Delta}\right)}$.*

Consequently, the `git bisect` *algorithm is a $\frac{1}{\log_2\left(\frac{\Delta+1}{\Delta}\right)}$-approximation algorithm on Δ-ary DAGs.*

Note that the bound above is tight. Indeed, the previous construction of graphs J_k^d can be extended by merging $\Delta + 1$ paths, Δ of which are analogous to $\boldsymbol{x_1} \to \boldsymbol{x_2} \to \cdots \to \boldsymbol{x_{k+1}}$. We get that $n_d = n_{d-1} + (\Delta+1)\ell_d$, and $\boldsymbol{c}$ as well as each vertex $\boldsymbol{x_k}$ has score $\frac{n_{d-1}}{\Delta+1}$. The `git bisect` algorithm selects the vertex $\boldsymbol{z'_d}$, which has $(\Delta+1)\ell_d$ ancestors.

Choosing an appropriate value for ℓ_d (namely $\lfloor n_{d-1}/\Delta(\Delta+1)\rfloor + 1$), we end up with a graph of order $(1 + 1/\Delta)^k(\Delta k + O(1))$. Thus, the optimal strategy on the comb requires $k\log_2((\Delta+1)/\Delta) + o(k)$ queries.

4 A new algorithm with a better approximation ratio for binary DAGs

In this section, we describe a new algorithm improving the number of queries in the worst-case scenario compared to `git bisect`.

4.1 Description of `golden bisect`

We design a new algorithm for the Regression Search Problem, named *`golden bisect`*, which is a slight modification of `git bisect`. It is so called because it is based on the *golden ratio*, defined as $\phi = \frac{1+\sqrt{5}}{2}$.

The difference of `golden bisect` with respect to `git bisect` is that it may not query a vertex with the maximum score if the maximum score is too small. Let us give some preliminary definitions.

Definition 15 (Subsets $B^{\ge}$ and $B^{<}$)**.** *Let D be a DAG. We define $V^{\ge}$ as the set of vertices which have more ancestors than non-ancestors. Let $B^{\ge}$ (for "Best" or "Boundary") denote the subset of vertices $\boldsymbol{v}$ of $V^{\ge}$ such that no parent of $\boldsymbol{v}$ belongs to $V^{\ge}$ and let $B^{<}$ be the set of parents of vertices of $B^{\ge}$.*

The reader can look at Figure 12 for an illustrative example. Note that the score of a vertex $\boldsymbol{v}$ with $|v|$ ancestors is $n - |v|$ if $\boldsymbol{v} \in V^{\ge}$, or $|v|$ if $\boldsymbol{v} \notin V^{\ge}$.

Using the sets defined above, we propose a refinement of Lemma 10.
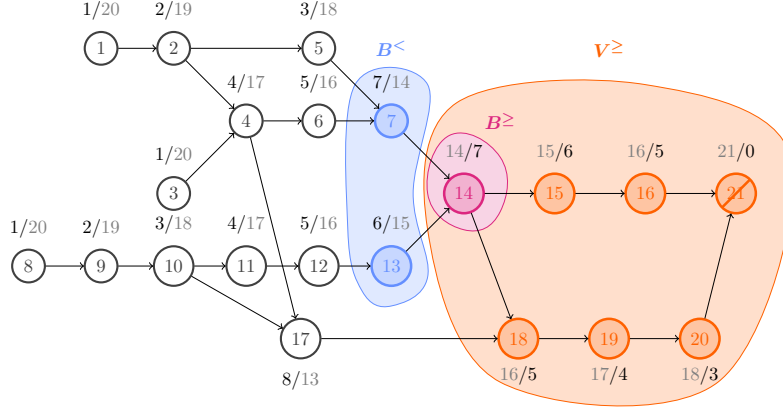
Figure 12: A binary DAG with the 3 sets of vertices $V^{\geq}$, $B^{\geq}$ and $B^{<}$.

Lemma 16. *Given any DAG with n vertices, there exists a vertex $\boldsymbol{v} \in B^{\geq} \cup B^{<}$ such that $\mathsf{score}(\boldsymbol{v}) \geq \dfrac{n-1}{3}$.*

Proof. The lemma is obvious whenever $n \leq 3$ (choose a vertex with no parent).

Let us choose any $\boldsymbol{v}$ in $B^{\geq}$. Since the graph is binary, $\boldsymbol{v}$ has 1 or 2 parents. Let us study both cases separately.

Vertex v has only one parent p. Thus, $\boldsymbol{p}$ has exactly $|v| - 1$ ancestors and $n - |v| + 1$ non-ancestors. Since $\boldsymbol{p}$ is not in $V^{\geq}$, $|v| - 1 < n - |v| + 1$, and thus $|v| < \frac{n}{2} + 1$.

Also, since $\boldsymbol{v} \in V^{\geq}$, $\mathsf{score}(\boldsymbol{v}) = n - |v| \geq \frac{n}{2} - 1$, which satisfies the lemma whenever $n \geq 4$.

Vertex v has two parents x and y, belonging to $B^{<}$, respectively having $|x|$ and $|y|$ ancestors. If any of $\boldsymbol{x}$ or $\boldsymbol{y}$ has $\dfrac{n-1}{3}$ or more ancestors, then the lemma holds for $\boldsymbol{x}$ or $\boldsymbol{y}$, since $\mathsf{score}(\boldsymbol{x}) = |x|$ and $\mathsf{score}(\boldsymbol{y}) = |y|$.

Let us assume the contrary, that is $|x| < \dfrac{n-1}{3}$ and $|y| < \dfrac{n-1}{3}$. But aside itself, every ancestor of $\boldsymbol{v}$ must be an ancestor of $\boldsymbol{x}$ or $\boldsymbol{y}$. Hence

$$|v| \leq |x| + |y| + 1 < \frac{n-1}{3} + \frac{n-1}{3} + 1 = \frac{2n+1}{3}.$$

Since $\boldsymbol{v}$ is in $V^{\geq}$, $\mathsf{score}(\boldsymbol{v}) \geq n - \dfrac{2n+1}{3} = \dfrac{n-1}{3}$. Thus $\boldsymbol{v}$ satisfies the condition of the lemma. $\qquad\square$

A description of `golden bisect` is given by Algorithm 2.

Now, let us describe the behavior of the `golden bisect` algorithm on the example of Figure 12. We have $21/\phi^2 \approx 8.02$. The maximum score 8 is smaller

Input. A DAG D and a bugged vertex $\boldsymbol{b}$.
Output. The faulty commit of D.
Steps:

1. Remove from D all non-ancestors of $\boldsymbol{b}$.

2. If D has only one vertex, return this vertex.

3. Compute the score for each vertex of D.

4. **If the maximum score is at least $\frac{n}{\phi^2} \approx 38.2\% \times n$ (where $\phi = \frac{1+\sqrt{5}}{2}$), query a vertex with the maximum score.**

5. **Otherwise, query a vertex of $B^{\geq} \cup B^{<}$ which has the maximum score among vertices of $B^{\geq} \cup B^{<}$, even though it may not be the overall maximum score.**

6. If the queried vertex is bugged, remove from D all non-ancestors of the queried vertex. Otherwise, remove from D all ancestors of the queried vertex.

7. Go to Step 2.

(The differences with `git bisect` are displayed in **bold**.)
Algorithm 2: `golden bisect`

than this number, so we run Step 5 instead of Step 4. Thus as its first query, `golden bisect` chooses **7**, which belongs to $B^{<}$ and has score 7. Another possible first query is to choose **14**, which has the same score as **7**, but belongs to $B^{\geq}$. In both cases, `golden bisect` uses 5 queries in the worst-case scenario (see Figure 13 for a possible strategy tree whenever **7** is queried). This diverges from `git bisect`, which would pick **18** (with a score of 8) with 6 queries in the worst-case scenario.

The `golden bisect` strategy is not always better than `git bisect`: Figure 14 shows an example of binary DAG where `git bisect` is thriftier than `golden bisect`.

In this figure, vertex $\boldsymbol{a}$ is queried first by `golden bisect` contrary to `git bisect` which starts with vertex $\boldsymbol{b}$. Each grey rectangle represents a directed path and the number inside is the number of vertices.

To understand why `golden bisect` is less efficient than `git bisect`, both strategies are also presented under the form of trees where the nodes represent the induced subgraphs. The directed paths are not developed here and the colored squares represent the number of additional queries used to find the faulty commit
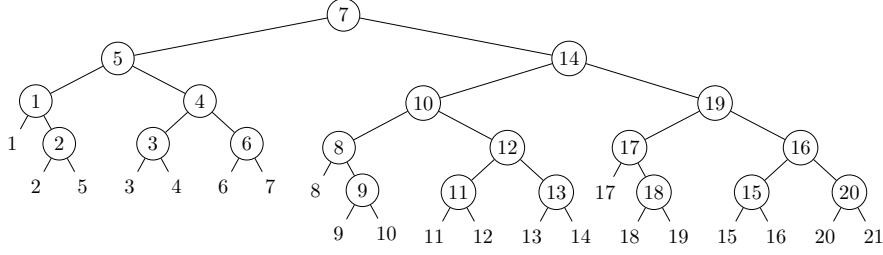
Figure 13: A `golden bisect` strategy tree for the digraph of Figure 12. In case of equality of score, the vertex with the smallest label is chosen.

in the worst-case scenario.

4.2 Results for `golden bisect` on binary DAGs

This subsection lists the main results about the complexity analysis of `golden bisect`. First, note that Theorem 6 also holds for `golden bisect`, so the general case (i.e, whenever the DAGs are not necessarily binary) is as bad as `git bisect`.

As for binary DAGs, we establish that the `golden bisect` algorithm has a better upper bound for the number of queries, in comparison with `git bisect`.

Theorem 17. *On any binary DAG with n vertices, the number of `golden bisect` queries is at most $\log_\phi(n) + 1 = \frac{\log_2(n)}{\log_2(\phi)} + 1$, where ϕ is the golden ratio.*

Proof idea The `golden bisect` algorithm has the remarkable following property: starting from a graph with n vertices, either the subgraph remaining after one query is of size at most $\frac{n}{\phi}$, or the subgraph obtained after two queries is of size at most $\frac{n}{\phi^2}$. If we admit this point, the proof of Theorem 17 has no difficulty.

The reason why we have such a guarantee on the size of the remaining graph after one or two queries comes from the choices of the sets $B^\geq$ and $B^<$. If `golden bisect` first queries a bugged vertex of $B^\geq$ with a "bad" score, then the parents of this vertex must have a "really good" score in the new resulting graph.

Let us take a critical example: `golden bisect` queries a bugged vertex of $B^\geq$, let us say q, with $\mathsf{score}(q) = (n-1)/3$ — which is the worst possible score for such vertices, by Lemma 16. In this case, each of the two parents of q has the really good score of $(n-1)/3$ in the new graph, which is approximately half of its size. So, even if the first query just removes one third of the vertices, the size of the graph after two queries is more or less $n/3$ (which is smaller than n/ϕ^2). Similar arguments hold whenever the first query concerns a vertex of $B^<$.

The ratio $1/\phi$ appears in fact whenever we try to balance what could go wrong after one query and what could go wrong after two queries.
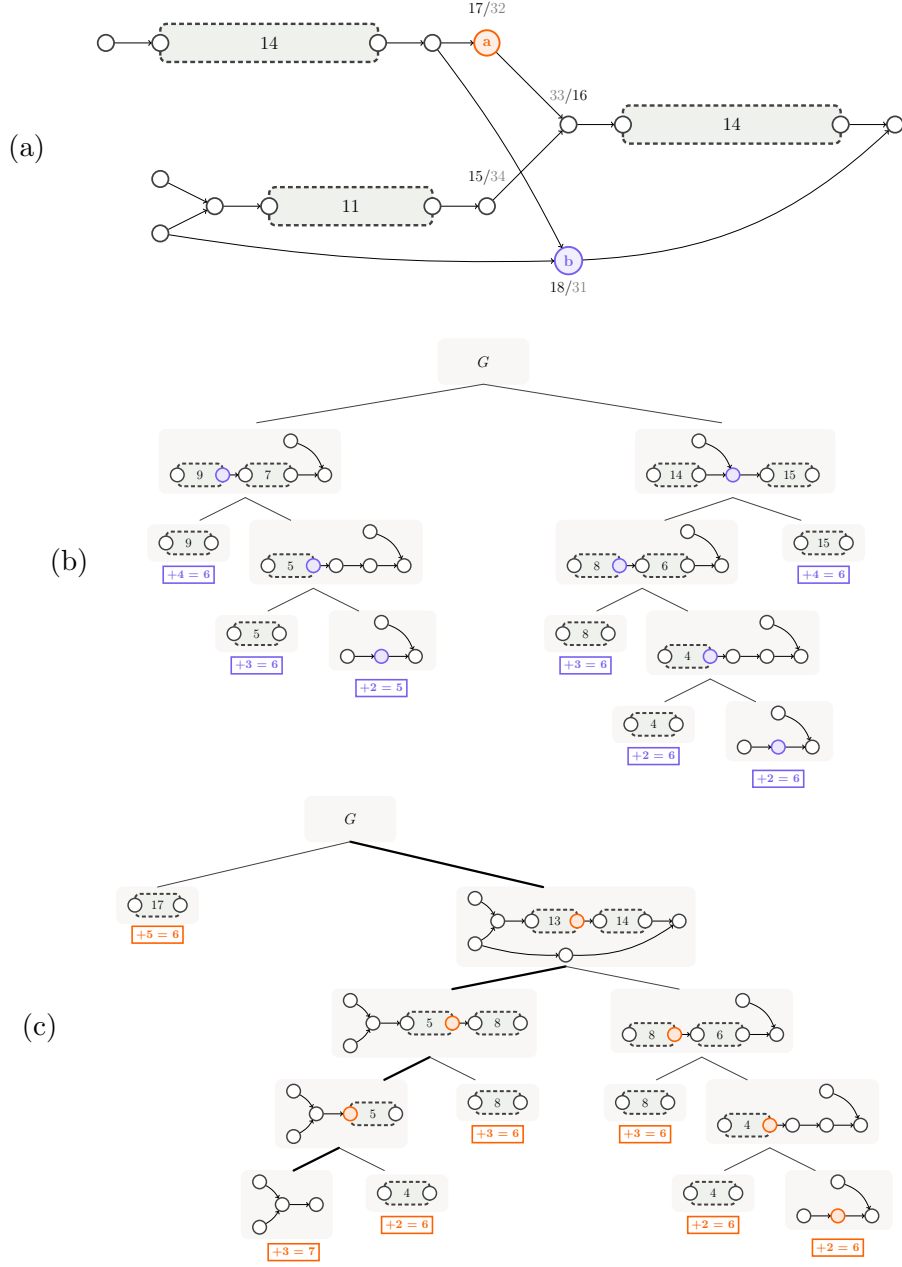
Figure 14: (a) A graph of size 49 where `golden bisect` uses one more query than `git bisect`. Dashed boxes represent directed paths. (b) The strategy tree for `git bisect`. (c) The strategy tree for `golden bisect`.

As first corollary, `golden bisect` is a better approximation algorithm than `git bisect` (in the binary case):

Corollary 18. *For every $\varepsilon > 0$, `golden bisect` is a $\left(\frac{1}{\log_2(\phi)} + \varepsilon \right)$-approximation algorithm on binary DAGs with a sufficiently large size.*

This also gives an upper bound for the optimal number of queries in the worst-case scenario, using the fact that no power of ϕ is an integer and thus that $\lfloor \log_\phi(n) + 1 \rfloor = \lceil \log_\phi(n) \rceil$.

Corollary 19. *For any binary DAG D with n vertices, the optimal number* opt *of queries in the worst-case scenario satisfies*

$$\lceil \log_2(n) \rceil \leq \text{opt} \leq \lceil \log_\phi(n) \rceil.$$

Note that the latter corollary is an analogue of Proposition 2, but for binary DAGs. The lower bound is satisfied for a large variety of DAGs, the most obvious ones being the directed paths. As for the upper bound, there is a 4-vertices graph, commonly named claw (see Figure 15), that uses $\lceil \log_\phi(4) \rceil = 3$ queries in the worst-case scenario.
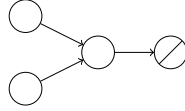

Figure 15: Claw graph.

4.3 Proof of the upper bound

We prove here the upper bound for the number of queries used by `golden bisect` for binary graphs.

Recall that $\phi = \dfrac{1 + \sqrt{5}}{2}$ is the golden ratio. We also have $1 + \phi - \phi^2 = 0$, and thus $n - \frac{n}{\phi} = \frac{n}{\phi^2}$.

Lemma 20. *For any binary DAG with $n \geq 14$ vertices,*

(i) *either the `golden bisect` reduces the searching area to at most $\frac{n}{\phi}$ in one query,*

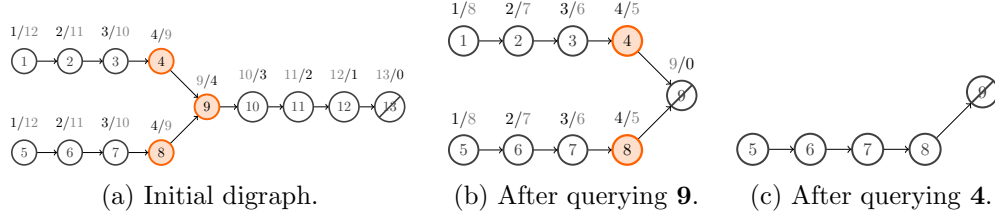(ii) *or it reduces the searching area to at most $\frac{n}{\phi^2}$ in two queries.*

(a) Initial digraph. (b) After querying **9**. (c) After querying **4**.

Figure 16: First two steps of `golden bisect` for a DAG of size 13.

Note that the lemma does not hold for $n = 13$, as shown by Figure 16. Here, the digraph after 1 `golden bisect` step has 9 vertices, which is larger than $\frac{13}{\phi} \approx 8.03$, and after 2 `golden bisect` steps, it has 5 vertices, which is larger than $\frac{13}{\phi^2} \approx 4.96$.

Proof. If there exists a vertex $\boldsymbol{v}$ in D with $\mathsf{score}(\boldsymbol{v}) \geq \frac{n}{\phi^2}$, then item (i) holds since `golden bisect` will query such a vertex via Step 4, and then there will remain at most $n - \frac{n}{\phi^2} = \frac{n}{\phi}$ vertices.

Thus we can assume that all vertices have a score smaller than $\frac{n}{\phi^2}$. Note that under this assumption, every vertex of $B^{\geq}$ has exactly two parents. Indeed, if a vertex $\boldsymbol{z} \in B^{\geq}$ has only one parent, say $\boldsymbol{x}$, then

$$\mathsf{score}(\boldsymbol{x}) = |x| = |z| - 1 \geq \frac{n}{2} - 1 \geq \frac{n}{\phi^2}$$

(the last inequality is true whenever $n \geq 9$).

Let us start by proving a useful claim.

Claim 20.1. *Suppose that after the first query in D, the resulting digraph, say D', satisfies the following two properties :*

- *$|D'|$, the number of vertices of D', is greater than $\frac{n}{\phi}$;*

- *there exists a vertex $\boldsymbol{v}$ in D' with no more than $\frac{n}{\phi^2}$ ancestors in D', and no more than $\frac{n}{\phi^2}$ non-ancestors in D'.*

Then the score of $\boldsymbol{v}$ in D' is greater or equal than $\frac{|D'|}{\phi^2}$, and item ii of the lemma holds.

Proof. Let $\#a$ and $\#na$ be respectively the number of ancestors and non-ancestors of $\boldsymbol{v}$ in D', with $\#a \leq \frac{n}{\phi^2}$ and $\#na \leq \frac{n}{\phi^2}$.

First notice that the score of $\boldsymbol{v}$ in D' is the minimum between $\#a$ and $\#na$. So, if we show that both $\#a$ and $\#na$ are no less than $\frac{|D'|}{\phi^2}$, the first part of the claim is proved.

26

Thus

$$\#a = |D'| - \#na \ge |D'| - \frac{n}{\phi^2} = \frac{|D'|}{\phi^2} + \frac{1}{\phi}\left(|D'| - \frac{n}{\phi}\right)$$

(the last equality can be derived from the identity $1 = \frac{1}{\phi} + \frac{1}{\phi^2}$). But since $|D'| > \frac{n}{\phi}$ by hypothesis, we deduce that $\#a \ge \frac{|D'|}{\phi^2}$. The numbers $\#a$ and $\#na$ play symmetric roles in this claim, so we can similarly infer that $\#na \ge \frac{|D'|}{\phi^2}$. Thereby we have proved that $\mathsf{score}_{D'}(\boldsymbol{v}) \ge \frac{|D'|}{\phi^2}$.

As for the second part of the claim, `golden bisect` will run Step 4 and query $\boldsymbol{v}$ or a vertex with a larger score. The searching area is thus reduced to at most $\frac{n}{\phi^2}$ vertices. $\qquad\square$

Let $\boldsymbol{z}$ be the first vertex queried by `golden bisect`. Note that $\boldsymbol{z}$ belongs to $B^{\ge}$ or $B^{<}$. In either case, we are going to show that if (i) fails, then the hypotheses of the above claim are satisfied, and consequently (ii) holds.
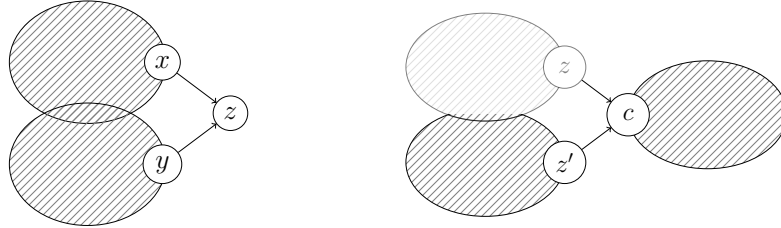
(a) First queried vertex $\boldsymbol{z}$ is in $B^{\ge}$, and $\boldsymbol{z}$ is bugged.

(b) First queried vertex $\boldsymbol{z}$ is in $B^{<}$, and $\boldsymbol{z}$ is clean.

Figure 17: Possible digraphs after the first query of `golden bisect`.

Case 1: $\boldsymbol{z} \in B^{\ge}$. Since $\boldsymbol{z} \in V^{\ge}$ by hypothesis, its score corresponds to the number of non-ancestors, thus $\mathsf{score}_D(\boldsymbol{z}) = n - |z| \le \frac{n}{\phi^2}$. If $\boldsymbol{z}$ is clean, only the non-ancestors of $\boldsymbol{z}$ remain after one step of `golden bisect`, which is fewer than $\frac{n}{\phi^2}$ vertices, and (i) holds.

Suppose now that $\boldsymbol{z}$ is bugged. Let D' denote the DAG obtained from D after querying $\boldsymbol{z}$ (constisting in only the ancestors of $\boldsymbol{z}$, which becomes the new marked sink, see Figure 17(a)). Note that D' has $|z| = n - \mathsf{score}_D(\boldsymbol{z})$ vertices, which is greater than $n - \frac{n}{\phi^2} = \frac{n}{\phi}$.

Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be the parents of $\boldsymbol{z}$, and assume that $|x| \ge |y|$. Since $|x| = \mathsf{score}_D(\boldsymbol{x}) \le \frac{n}{\phi^2}$, vertex $\boldsymbol{x}$ has at most $\frac{n}{\phi^2}$ ancestors in D'. Moreover, vertex $\boldsymbol{x}$ has $|z| - |x|$ non-ancestors in D'. But since a non-ancestor of $\boldsymbol{x}$ in D' is either $\boldsymbol{z}$ or an ancestor of $\boldsymbol{y}$, we have

$$|z| - |x| \le |y| + 1$$

and because $|y| \leq |x| \leq \mathsf{score}_D(z) = n - |z|$, we deduce

$$|z| - |x| \leq |x| + 1 \leq n - |z| + 1,$$

hence

$$3(|z| - |x|) \leq (|z| - |x|) + (|x| + 1) + (n - |z| + 1) = n + 2.$$

Thus x has at most $\frac{n+2}{3}$ non-ancestors, which is smaller than $\frac{n}{\phi^2}$ whenever $n \geq 14$. The hypotheses of Claim 20.1 hold, which concludes Case 1.

Case 2: $z \in B^<$. Since z belongs to $B^<$, it has a child in $B^\geq$, denoted by c. By assumption, c has 2 parents, let z' be the other parent of c (also in $B^<$).

If z is bugged, then there remain at most $\mathsf{score}(z) = |z| < \frac{n}{\phi^2}$ vertices, which makes (i) true. So assume that the queried vertex z is clean, and after one step of `golden bisect`, we end up with a new DAG D', obtained from D by removing all ancestors of z (see Figure 17(b)). Note that D' has $n - |z|$ vertices, which is greater than $n - \mathsf{score}_D(z) > \frac{n}{\phi}$.

Note that the non ancestors of c in D' are exactly the non ancestors of c in D, and thus c has no more than $\frac{n}{\phi^2}$ non ancestors in D'. Thus if c has no more than $\frac{n}{\phi^2}$ ancestors in D', c satisfies the hypotheses of Claim 20.1 and the result holds.

Assume that c has more than $\frac{n}{\phi^2}$ ancestors in D', which means that

$$|c| - |z| \geq \frac{n}{\phi^2} \geq n - |c|.$$

Since $\frac{n}{\phi^2}$ cannot be an integer, $|c| - |z| \geq n - |c| + 1$.

Moreover because $n - |c| = \mathsf{score}_D(c) \leq \mathsf{score}_D(z) = |z|$, we have

$$3(n - |c| + 1) \leq (n - |c| + 1) + (|z| + 1) + (|c| - |z|) = n + 2.$$

So the number of non-ancestors of z' in D', that is $n - |c| + 1$, is less than $\frac{n+2}{3} < \frac{n}{\phi^2}$ when $n \geq 14$. Moreover, the number of ancestor of z' in D' is no more than its number of ancestors in D, which satisfies $|z'| = \mathsf{score}_D(z') \leq \frac{n}{\phi^2}$. So z' satisfies the hypotheses of Claim 20.1 and the result holds. $\qquad\square$

We can now establish the upper bound for the overall number of `golden bisect` queries.

Proof of Theorem 17. We prove by induction on n that for any binary DAG with n vertices, the number of `golden bisect` queries is at most $\log_\phi(n) + 1$.

The base case contains all graphs up to $n = 13$. To prove it, we use Lemma 16 to show that `golden bisect` eliminates at least $\frac{n-1}{3}$ vertices at the first step. So the maximal number of queries for size n is bounded by one plus the maximal number of queries for size $n - \lceil \frac{n-1}{3} \rceil$. The first values are given by the sequence

$F(n)$ in Table 1. We remark that the second row is bounded above by the last row. So the property holds for $n \leq 13$.

As for $n \geq 14$, the induction is straightforward by Lemma 20. Indeed, if item i is satisfied, then the number of `golden bisect` queries is bounded by $1 + \left(\log_\phi \left(\frac{n}{\phi} \right) + 1 \right) = 1 + \log_\phi(n)$. If item ii is satisfied, then it is also bounded by $2 + \left(\log_\phi \left(\frac{n}{\phi^2} \right) + 1 \right) = 1 + \log_\phi(n)$. $\qquad \square$

4.4 Fibonacci trees

In order to prove the sharpness of the constant $\frac{1}{\log_2(\phi)}$ from Corollary 18, we define a new family of digraphs: the *Fibonacci trees*.

Definition 21 (Fibonacci trees)**.** *For $i \geq 1$, the i-th Fibonacci tree F_i is defined as followed.*

F_1 is a single vertex, F_2 is an arc between two vertices, and for $i \geq 2$, F_{i+1} is a sink with two parents, one being the sink of a tree F_i and the other the sink of a tree F_{i-1}.

Figure 18 shows the six first Fibonacci trees. Noting $|F_i|$ the number of vertices of the i-th Fibonacci tree F_i, we have by construction, that

$$|F_i| = |F_{i-1}| + |F_{i-2}| + 1, \quad |F_1| = 1 \quad \text{and} \quad |F_2| = 2.$$

This recurrence has for solution $|F_i| = fib_{i+2} - 1$, where fib_i is the i-th Fibonacci number.

We can establish an optimal strategy for the Fibonacci trees.

Theorem 22. *For any $i \geq 1$, the optimal strategy for the i-th Fibonacci tree F_i uses $i - 1$ queries in the worst-case scenario.*

We decompose this proof in two claims.

Claim 22.1. *For $i \geq 1$, we define F_i' as a sink with one parent which is the sink of an i-th Fibonacci tree F_i. For any F_i and F_{i-1}' with $i \geq 2$, there exists a strategy which finds the faulty commit in at most $i - 1$ queries.*

Proof. We prove the claim by induction on $i \geq 2$. The base case $i = 2$ is obvious since F_2 and F_1' are each composed of two vertices including the marked bugged vertex, and that in this case one query is required to discover the faulty commit. Suppose by induction we have a strategy for F_{i-1}' and F_i with $i - 1$ queries.

To find the faulty commit in F_{i+1}, we query the sink of the subtree F_i. If this sink is bugged, then we continue with our strategy on F_i. If this sink is clean, then by removing this subtree from F_{i+1}, we recognize F_{i-1}' and use the

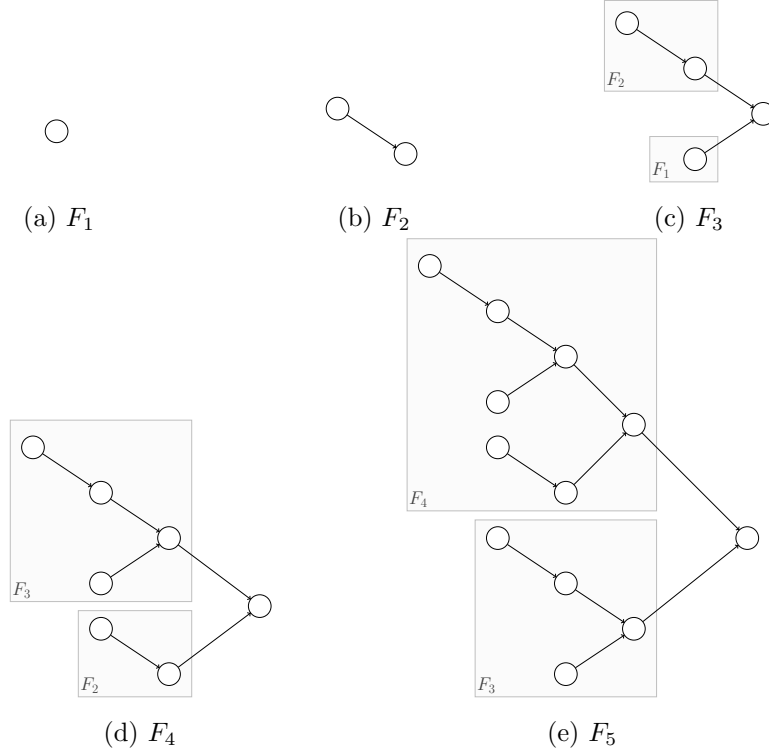(a) F_1 (b) F_2 (c) F_3

(d) F_4 (e) F_5

Figure 18: First Fibonacci trees.

corresponding strategy. In both cases, we have performed no more than $1+(i-1)$ queries.

To find the faulty commit in F_i', we query the parent of the sink. If it is bugged, then we continue with our strategy on the resulting F_i. Otherwise, the sink of F_i' is the faulty commit. This strategy uses at most $1 + (i - 1)$ queries, as required.   □

Claim 22.2. *If T is a tree containing two subtrees isomorphic to Fibonacci trees F_k and F_{k+1} and whose sinks are non-ancestors of each other (cf Figure 19 top), then, for any strategy searching for the faulty commit in T, there exists a vertex v in T such that this strategy uses at least $k+1$ queries to identify v as the faulty commit.*

Proof. We prove this claim by induction on k. For $k = 1$, T must contain at least 4 vertices, so by Proposition 2, one must query at least 2 vertices.

Now suppose the claim statement true for a positive integer $k-1$, and consider a strategy for the Regression Search Problem on a tree T strictly containing
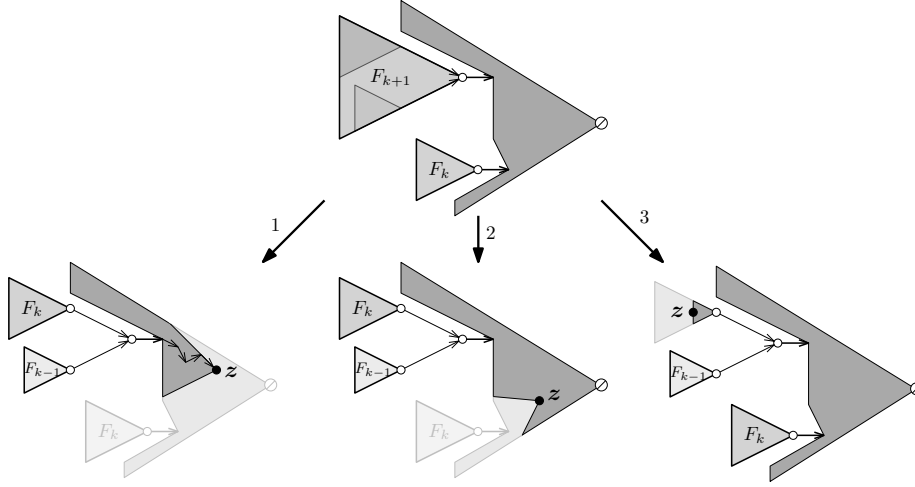
Figure 19: Illustration of the proof of Claim 22.2.

F_{k+1} and F_k. Let z be the first query of this strategy. Let us investigate every possibility for z (the reader can refer to Figure 19 for an illustration):

1. **The root of the subtree F_{k+1} is an ancestor of z.** Then we force the faulty commit to be an ancestor of z (i.e. z is bugged). Then after querying z, there remains all ancestors of z, which contains F_{k+1}, which, by definition of Fibonacci trees, strictly contains F_k and F_{k-1}. By induction hypothesis, we need to query k extra vertices to find the faulty commit.

2. **The root of the subtree F_{k+1} is not an ancestor of z and z is not in the subtree F_{k+1}.** Here the faulty commit will be a non-ancestor of z (i.e. z is not bugged). Like in the previous case, the remaining digraph will include F_{k+1}, hence copies of F_k and F_{k-1}. We then use the induction hypothesis.

3. **z is in the subtree F_{k+1}, but it is not its root.** We set z to be clean so that the faulty commit will be among the non-ancestors of z. The subtree F_{k+1} contains two disjoint copies of F_{k-1}, one of which is in the non-ancestors of z. By hypothesis, T includes also another copy of F_k. So the query of z leads to a tree containing F_k and F_{k-1}: the induction hypothesis indicates that we need k other queries.

For each of these three possibilities, the strategy uses in total $k+1$ queries, which concludes the induction. □

Conclusion of the proof of Theorem 22. The Fibonacci tree F_i contains disjoint copies of F_{i-1} and F_{i-2}. By Claim 22.2, any strategy, in particular an optimal

one, uses at least $i - 1$ queries to find the faulty commit in F_i in the worst-case scenario. The optimal number of queries is then exactly $i - 1$, because by Claim 22.1, there exists a strategy with that many queries. $\qquad\square$

The first consequence of Theorem 22 is that the upper bound $\lceil \log_\phi(n) \rceil$ from Corollary 19 is asymptotically sharp:

Corollary 23. *Any optimal strategy solving the Regression Search Problem for any Fibonacci tree of size $n \geq 7$ uses $\lceil \log_\phi(n) \rceil - 2$ queries in the worst-case scenario.*

The authors do not know if there exist an infinity of graphs for which solving the Regression Search Problem requires exactly $\lceil \log_\phi(n) \rceil$ queries.

Proof of Corollary 23. Recall that $|F_i| = fib_{i+2} - 1$. Using that $fib_{i+2} = \left(\phi^{i+2} - (-\phi)^{-i-2} \right) / \sqrt{5}$, we get

$$\log_\phi(|F_i|) = i + 2 - \log_\phi(\sqrt{5}) + \varepsilon_i,$$

where

$$\varepsilon_i = \log_\phi \left(1 - \frac{\sqrt{5}}{\phi^{i+2}} - \frac{1}{(-\phi^2)^{i+2}} \right)$$

which increases and tends to 0, hence is in absolute value no more than $|\varepsilon_4| \leq 0.3$ whenever $i \geq 4$. If $i \geq 4$, we have $\lceil \log_\phi(|F_i|) \rceil = i + 2 - \lfloor \log_\phi(\sqrt{5}) + 0.3 \rfloor = i + 1$. We conclude by Theorem 22. $\qquad\square$

The previous corollary demonstrates that the Fibonacci trees are inherently flawed for the Regression Search Problem. They are the less pathological analogues of octopuses, but in the context of binary DAGs.

Finally, we show that $\frac{1}{\log_2(\phi)}$ is the good approximation ratio for `golden bisect`.

Corollary 24. *For $\varepsilon > 0$, `golden bisect` is not a $\left(\dfrac{1}{\log_2(\phi)} - \varepsilon \right)$ approximation algorithm.*

Proof. The idea is to add a comb (see Definition 4) to the i-th Fibonacci tree F_i to approach the $\frac{1}{\log_2(\phi)}$ ratio.

Indeed, Theorem 5 still holds if we replace "`git bisect`" by "`golden bisect`" since both algorithms will query first v_n, which is the only vertex with the maximal score n. Thus we apply this theorem for all i such that $|F_i|$ is odd, i.e., whenever i is congruent to 1 modulo 3.

We deduce that $comb(F_i)$ is a binary DAG for which:

- the number of `golden bisect` queries is $\lceil \log_\phi(|F_i|) \rceil - 1$ (see Corollary 23),

- the optimal number of queries is $\lceil \log_2(|F_i|) \rceil + 1$.

The ratio of these two numbers makes a number tending to $\frac{1}{\log_2(\phi)}$, whenever i goes to $+\infty$. This is why `golden bisect` cannot be a $\left(\frac{1}{\log_2(\phi)} - \varepsilon \right)$ approximation algorithm, for any $\varepsilon > 0$. $\qquad \square$

5   Is the binary case NP-complete?

Though `git bisect` and `golden bisect` are non-optimal algorithms, the Regression Search Problem (RSP – see Definition 1) on binary DAGs is not proved to be NP-hard. We still do not know whether it is the case, but we show the NP-completeness of a new related problem, the Confined Regression Search Problem (CRSP). It is a reformulation of RSP in the general case but not in the binary case.

In the Confined Regression Search Problem, we consider a DAG D with additional information on some vertices. A vertex is said to be *innocent* if it is not the faulty vertex, i.e., if it is not the one that introduced the bug. It is still possible to query an innocent vertex since it can be bugged or clean. CRSP consists in searching the faulty vertex given a possibly empty set I of innocent vertices. Conversely, it confines the faulty vertex to be in the complementary set of I. The DAG does not necessarily have any bugged vertex any more.

The decision version of the Confined Regression Search Problem is formally defined as follows.

Definition 25. *Confined Regression Search Problem*
Input. *A DAG D, a subset I of innocent vertices, and an integer k.*
Output. *Whether there is a strategy that determines in at most k queries in the worst-case scenario whether D has a bugged vertex, and if it is the case, which one is the faulty vertex.*

Figure 20 illustrates the equivalence between instances of CRSP and RSP in the general case. The reductions work with the following transformations.

- from CRSP to RSP: create a bugged vertex b and add arcs from all non-innocent vertices to b.

- from RSP to CRSP: delete the bugged vertex b and all its descendants. Set as innocent all vertices which were not ancestors of b.

The strategies are preserved in both reductions, but with the following change: if the DAG in CRSP has no bugged vertex then the marked vertex b is the faulty vertex in RSP and *vice versa*. However, note that though the reduction from RSP

(a) Instance of CRSP.

(b) Instance of RSP.

(c) Instance of RSP.
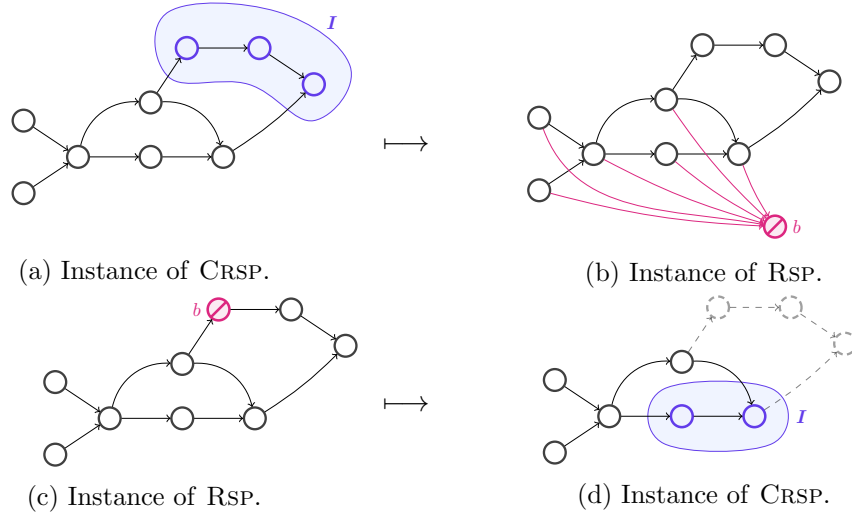
(d) Instance of CRSP.

Figure 20: *Top.* Example of a transformation from an instance of CRSP to an instance of RSP. *Bottom.* Example of a transformation from an instance of RSP to an instance of CRSP.

to CRSP preserves the indegree of the DAG, the reduction from CRSP to RSP creates a vertex with a large indegree.

To prove that CRSP is NP-hard even restricted to binary DAGs, we show that there is a polynomial reduction from the problem BOUNDED (2,3)-SAT (BSAT), proved to be NP-complete in [15].

Definition 26. *Bounded (2,3)-SAT*
Input. *A Boolean formula in Conjonctive Normal Form with the following restrictions:*

- *each clause contains 2 or 3 literals,*

- *each variable is present in at most 3 clauses.*

Output. *Whether there is an assignment of the variables that satisfies the formula.*

Note that we can assume without loss of generality that each literal of any instance of BSAT is present in at most 2 clauses. Indeed, if a literal appears in 3 clauses, the opposite literal does not appear in the formula. Assigning the corresponding variable so that the literal is true makes the three clauses satisfied. Thus they can be removed from the formula without changing its satisfiability.

Theorem 27. *Confined Regression Search Problem is NP-hard even when the inputs are restricted to binary DAGs.*

Proof. We show that there exists a polynomial reduction from BSAT to CRSP with inputs restricted to binary DAGs, named BIN-CRSP. The reduction algorithm takes as input a Boolean formula with n variables and m clauses and computes a binary DAG D in polynomial time as follows.

First, for each variable X_i, create two vertices x_i and $\overline{x_i}$, two *branching* vertices b_i and $\overline{b_i}$, and one *control* vertex ct_i. Connect b_i and ct_i to x_i, and $\overline{b_i}$ and ct_i to $\overline{x_i}$ (see Figure 21 for an example of this *variable gadget*).

Then for each clause C_j, create a vertex c_j. For each literal x_i (resp. $\overline{x_i}$) in the clause C_j, add an arc from c_j to b_i (resp. $\overline{b_i}$). Finally, create three isolated vertices t_1, t_2 and t_3. Set as innocent all vertices except t_1, t_2, t_3, the $(c_j)_{1 \leq j \leq m}$ and the $(ct_i)_{1 \leq j \leq n}$ vertices.

Note that since every literal appears in at most two clauses, each vertex b_i and $\overline{b_i}$ has indegree at most two, and the resulting DAG is binary. Moreover, the size of the Boolean formula and the size of the DAG are polynomials with respect to $(n + m)$.

For example, Figure 21 shows the DAG from the reduction of the following formula in BSAT form:

$$(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3}).$$

We show that the previous transformation is a reduction: the Boolean formula has a satisfiable assignment if and only if there is a strategy using $n + 3$ queries in the worst-case scenario.

First, let us suppose that F has a satisfiable assignment $X = \{X_1, X_2, \ldots, X_n\}$ where X_i is true or false and let us describe a strategy that solves BIN-CRSP.

For each i between 1 and n, query x_i if X_i is true, $\overline{x_i}$ if X_i is false. If none of the queries reveals a bugged vertex, query the three vertices t_1, t_2, t_3. If one is discovered as bugged then it is the faulty vertex. If they are all clean then there is no faulty vertex in the DAG. Otherwise, some vertex x_i or $\overline{x_i}$ is found to be bugged. Without loss of generality, let us assume it is a vertex x_i corresponding to a positive literal. Then, we query the parents of b_i (i.e. the vertices corresponding to clauses where x_i appears), which is done in no more than two queries. If a parent of b_i is discovered as bugged, then it is the faulty vertex. Otherwise, ct_i is the faulty vertex.

Conversely, let us suppose there exists a strategy solving BIN-CRSP in at most $n + 3$ queries. We prove that the Boolean formula F has a satisfiable assignment.

Consider the scenario with no faulty vertex in the DAG. We show that exactly $n + 3$ queries are required and that a solution of the formula can be deduced from which vertices have been queried. All the non-innocent vertices must be cleared to guaranty that the DAG has no faulty vertex, which can be done by querying the vertex itself or one of its descendants.
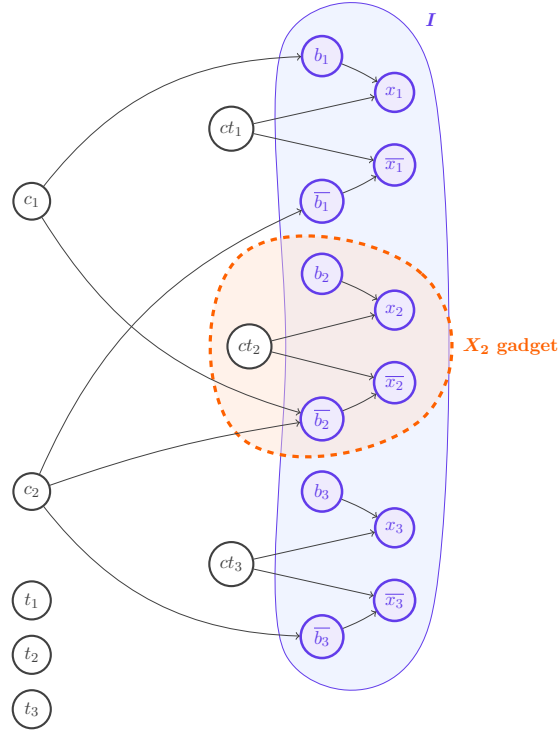
Figure 21: The graph resulting from the reduction of the formula $(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$.

Concerning the three terminal vertices t_1, t_2 and t_3, they are isolated so the strategy must query each of them to know that they are clean. To know that a ct_i vertex is clean we must query the ct_i vertex itself or one of its two descendants x_i or $\overline{x_i}$. Thus we must make at least one query in each group $\{ct_i, x_i, \overline{x_i}\}$ for $1 \le i \le n$. By our assumption that the strategy uses at most $n+3$ queries, there must be exactly one query in each group $\{ct_i, x_i, \overline{x_i}\}$.

For each $1 \le i \le n$, we assign X_i as

$$X_i = \begin{cases} \text{true} & \text{if } x_i \text{ is queried} \\ \text{false} & \text{if } \overline{x_i} \text{ is queried} \\ \text{true (arbitrarily)} & \text{if } ct_i \text{ is queried} \end{cases}$$

Since all clause vertices must be cleared, at least one of the descendants of each c_j is queried (c_j cannot be queried since we have already done $n+3$ queries). This implies that some literal of the corresponding clause C_j is true in the above assignment. □

As a corollary we have an alternative proof of the NP-completeness of RSP.

36

Corollary 28. *The problems* RSP, CRSP *and* BIN-CRSP *are NP-complete.*

Proof. On one hand, certificates for these 3 problems are strategy trees with a polynomial number of nodes. Thus CRSP-bin, CRSP and RSP are in NP. On the other hand, BIN-CRSP is NP-hard by Theorem 27. As a generalisation of BIN-CRSP, CRSP is NP-hard. Since RSP is equivalent to CRSP, as shown above, RSP is also NP-hard. □

In the light of the above reduction, let us explain why proving the NP-hardness of the Regression Search Problem for binary graphs (abbreviated BIN-RSP) seems to be arduous. Observe that the reduction from BSAT to BIN-CRSP (as well as the existing reductions in the literature) encodes the assignment of the n variables of the SAT formula into a sequence of queries from a specific scenario. Thereby the certificate is a strategy tree of depth at least n. By Theorem 17, any binary DAG requiring this number of queries to solve BIN-RSP has at least ϕ^{n-1} vertices and thus would not be polynomial in the size of the formula.

Therefore, to find a reduction from a SAT problem to BIN-RSP, it is not possible to encode the assignment of the variables in a single branch of the strategy tree. One needs to consider the tree widthwise, and use the sets of queried vertices in different scenarios. We do not know of any reduction using this thorny approach yet.

6 Conclusion

In summary, this paper has established that `git bisect` can be very inefficient on very particular digraphs, but under the reasonable hypothesis that merges must not concern more than 2 branches each, it is proved to be a good approximation algorithm. This study has also developed a new algorithm, `golden bisect`, which displays better theoretical results than `git bisect`.

Notably, some open questions remain:

- Is BIN-RSP, the Regression Search Problem for binary DAGs, still NP-complete, as discussed in the previous section?

- How do `git bisect` and `golden bisect` compare in terms of average-case complexity?

- In `git bisect` and in `golden bisect`, one never queries vertices which were eliminated from the set of candidates for the faulty commit. However, we could speed up the procedure by never removing any vertex after queries. For example, consider the DAG from Figure 7. If we choose v_7 as first query and it is bugged, then we remove all u_i (the non-ancestors of v_7).

37

However, querying the vertices u_i in the comb would be more efficient. Could we obtain an improved algorithm by authorising such queries?

- If we restrict the DAGs to be trees (oriented from the leaves to the root) with unbounded indegree, are `git bisect` and `golden bisect` good approximation algorithms? We conjecture that they are $2-$approximation algorithms for trees. (We have found examples where the ratio is 2.)

Finally it would be interesting to study the number of queries in the worst-case scenario when the input DAG is taken at random. Indeed, most of the examples described in this paper are not very likely to appear in reality. The notion of randomness for a digraph emanating from a VCS is therefore quite interesting and deserves to be developed. One could for example define a theoretical probabilistic model based on existing workflows. It would be also quite useful to use random samplers for VCS repositories in order to constitute benchmarks on demand.

Aknowledgments

References

[1] Micah Adler and Brent Heeringa. Approximating optimal binary decision trees. *Algorithmica*, 62(3-4):1112–1121, 2012.

[2] Esther M. Arkin, Henk Meijer, Joseph S. B. Mitchell, David Rappaport, and Steven S. Skiena. Decision trees for geometric models. *Internat. J. Comput. Geom. Appl.*, 8(3):343–363, 1998.

[3] Yosi Ben-Asher, Eitan Farchi, and Ilan Newman. Optimal search in trees, 1999.

[4] Jaroslav Bendík, Nikola Benes, and Ivana Cerna. Finding regressions in projects under version control systems. *CoRR*, 2017.

[5] Benoit Boissinot. *hg bisect mercurial manpage*.

[6] R. Carmo, J. Donadelli, Y. Kohayakawa, and E. Laber. Searching in random partially ordered sets. *Theoret. Comput. Sci.*, 321(1):41–57, 2004.

[7] Ferdinando Cicalese, Tobias Jacobs, Eduardo Sany Laber, and Marco Molinaro. On the complexity of searching in trees and partially ordered structures. *Theoretical Computer Science*, 412:6879–6896, 2011.

[8] Christian Couder. Fighting regressions with git bisect, 2009.

[9] Dariusz Dereniowski, Adrian Kosowski, Przemysł aw Uznański, and Mengchuan Zou. Approximation strategies for generalized binary search in weighted trees. 80:Art. No. 84, 14, 2017.

[10] Ehsan Emamjomeh-Zadeh, David Kempe, and Vikrant Singhal. Deterministic and probabilistic binary search in graphs. pages 519–532, 2016.

[11] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Lett.*, 5(1):15–17, 1976/77.

[12] Eduardo S. Laber and Loana Tito Nogueira. On the hardness of the minimum height decision tree problem. *Discrete Appl. Math.*, 144(1-2):209–212, 2004.

[13] Shay Mozes, Krzysztof Onak, and Oren Weimann. Finding an optimal tree searching strategy in linear time. pages 1096–1105, 2008.

[14] Krzysztof Onak and Pawel Parys. Generalization of binary search: Searching in trees and forest-like partial orders. pages 379–388, 2006.

[15] Craig A. Tovey. A simplified np-complete satisfiability problem. *Discrete Applied Mathematics*, 8(1):85–89, 1984.