Theoretical analysis of git bisect

Julien Courtiel, Paul Dorbec, Romain Lecoq

Theoretical analysis of git bisect

Julien Courtiel* Paul Dorbec† Romain Lecoq‡

November 4, 2021

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-hard.

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.

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 a 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. 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 *regression*. Even with intensive testing techniques, it seems unavoidable to

*`julien.courtiel@unicaen.fr`, Normandie University, UNICAEN, ENSICAEN, CNRS, GREYC.
†`paul.dorbec@unicaen.fr`, Normandie University, UNICAEN, ENSICAEN, CNRS, GREYC.
‡`romain.lecoq@unicaen.fr`, Normandie University, UNICAEN, ENSICAEN, CNRS, GREYC.

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 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 [7]). 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 Regression Search Problem is polynomial [3, 13], and even linear [12].

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 [10] as well as its approximation version [11], 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 [8, 9];
- the queries return the wrong result (say it is clean while the vertex is bugged, or the converse) with a certain probability [9];
- 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 as bad as possible, 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 do not contain any octopus merge, that is every vertex has indegree at most two. Under such an assumption, we are able to

[1] According to https://www.destroyallsoftware.com/blog/2017/the-biggest-and-weirdest-commits-in-linux-kernel-git-history, a merge of 66 branches happened in the Linux kernel repository.

prove in Section 3 that `git bisect` is an approximation scheme 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 prove this ratio is the best approximation ratio possible for `git bisect`.

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 equal to 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.

1.1 Formal definitions

Throughout the paper, we refer to VCS repositories as graphs, and more precisely as *Directed Acyclic Graphs* (DAG), 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 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 the number letter between two vertical bars (for example $|v|$).

For a DAG D and a subset of vertices $X \subseteq V$, the *induced subgraph* of D on X, denoted $D[X]$, is the digraph with vertex set X, and with an arc from vertex $\boldsymbol{u}$ to vertex $\boldsymbol{v}$ if and only if the corresponding arc is in D.

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.

We consider the problem of identifying the faulty commit in a DAG D, where a bugged vertex $\boldsymbol{b}$ is identified. Usually, since the faulty commit is necessarily an ancestor of $\boldsymbol{b}$, only the induced subgraph on $\boldsymbol{b}$'s ancestors is considered, and thus $\boldsymbol{b}$ is a *sink* (i.e. a vertex with no outgoing edge) accessible from all vertices in the DAG. When the bugged vertex is not specified, it is assumed to be the only unique sink of the DAG.

The problem 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* 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.

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

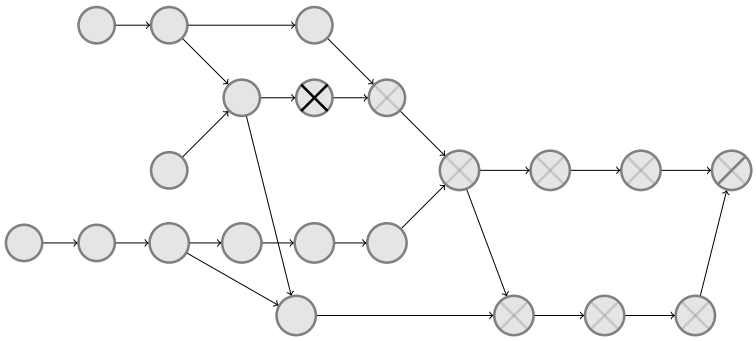


Figure 1: An example of a DAG. The bugged vertices are crossed. The black cross identifies the faulty commit.

If the queried vertex is bugged, then the following strategy is given by the left subtree. If it is clean, the strategy continues on the right subtree. At each query, there are fewer candidates for the faulty commit. Whenever a single candidate remains, the subtree is reduced to a leaf whose label is the only possible faulty commit.

For example, Figure 2 shows a strategy tree for a directed path of size 5. Suppose that the faulty commit is **4**. In this strategy, we query in first **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: A directed path on 5 vertices and a possible strategy for the Regression Search Problem.

The Regression Search Problem is formally defined as follows.

Definition 1. ***Regression Search Problem.***
Input. *A DAG D with a marked vertex* **b***, known to be bugged.*
Output. *A strategy which uses the least number of queries in the worst-case scenario.*

In terms of binary trees, the least number of queries in the worst-case scenario of a strategy corresponds to the height of the tree. For example, if the input DAG is a directed path of size n, we know that there exists a strategy with $\lceil \log_2(n) \rceil$ 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). When the faulty commit

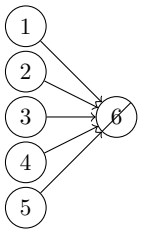


Figure 3: An octopus of size 6

is the sink, we must query all other vertices to make sure that the sink is faulty, regardless of the strategy. Thus, every strategy is equivalent for the Regression Search Problem on the n-vertices octopus, and uses $n-1$ queries in the worst case.

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

Proposition 2. *For any DAG D with n vertices, any 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 quite obvious because one can query at most $n-1$ vertices in the Regression Search Problem. □

From a complexity point of view, the Regression Search Problem is hard: Carmo, Donadelli, Kohayakawa and Laber proved in [6] that computing the least number of queries for the Regression Search Problem is NP-complete[3].

1.2 Description of `git bisect`

As said in the introduction, some VCS have implemented a tool for the Regression Search Problem. The most known one is `git bisect`, but it has its equivalent in `mercurial` (`hg bisect` [5]).

The algorithm `git bisect` is a heuristic based on the classical binary search. It consists of querying the vertices which 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 with n vertices, the* score *of a vertex $\boldsymbol{x}$ is*

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

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

[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 widget they used in the proof of NP-completeness also works for our problem where we do not necessarily forbid such queries.

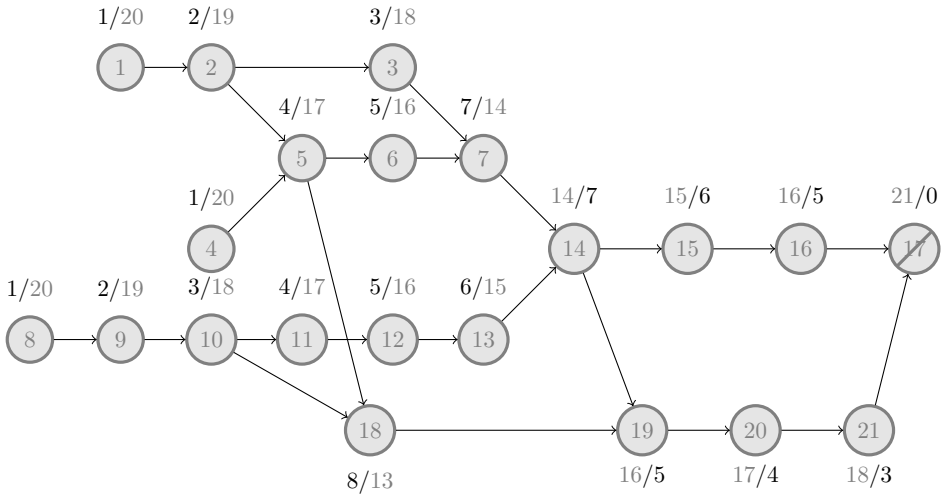


Figure 4: The notation a/b along each vertex indicates that a is the number of ancestors of the vertex, and b is the number of non-ancestors. The score is displayed in black.

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 best score is the score with the maximum value among all.

For example, let us refer to Figure 4: vertex **6** has 5 ancestors (**1**, **2**, **4**, **5** and **6**). Its score is so $\min(5, 21 - 5) = 5$.

We give now a detailed description of `git bisect`.

Algorithm 4 (`git bisect`)**.**
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 best score. If there are several vertices which have the best score, select only one indifferently 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.*

Take for example the DAG from Figure 4. Vertex **18** has the best score (8) so constitutes the first vertex to be queried. If we assume that the faulty commit is **6**, then the query reveals that **18** is clean. So all ancestors of **18** are removed (that are $\mathbf{1}, \mathbf{2}, \mathbf{4}, \mathbf{5}, \mathbf{8}, \mathbf{9}, \mathbf{10}, \mathbf{18}$). Vertex **14** is then queried because it has the best score 7, and so on.

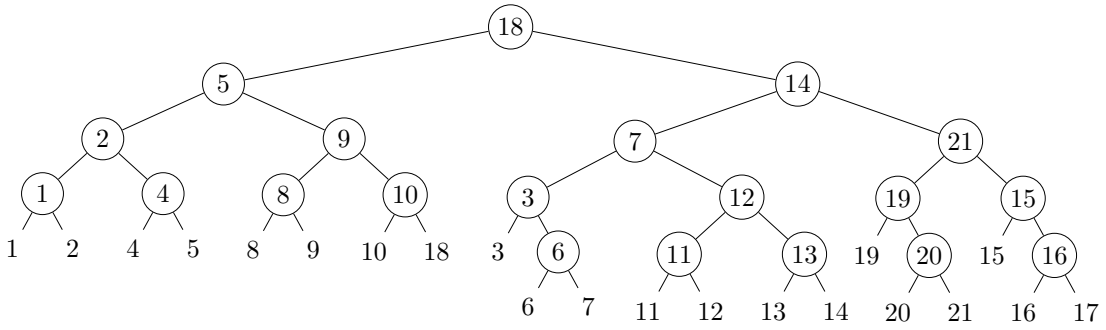


Figure 5: The `git bisect` strategy corresponding to the graph of Figure 4. In case of score equality, the convention we choose consists in querying the vertex with the smallest label.

The whole `git bisect` strategy tree is shown in Figure 5. Notice that for this DAG, the `git bisect` algorithm is optimal since in the worst-case scenario it uses 5 queries and by Proposition 9, we know that any strategy uses at least $\lceil \log_2(21) \rceil = 5$ queries.

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.

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

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

- *n new vertices $\boldsymbol{u_1}, \ldots, \boldsymbol{u_n}$;*
- *the arcs $\boldsymbol{v_i u_i}$ for $i \in \{1, \ldots, n\}$;*
- *the arcs $\boldsymbol{u_i u_{i+1}}$ for $i \in \{1, \ldots, n-1\}$.*

The resulting graph is denoted comb(D). The new identified bugged vertex of comb(D) is $\boldsymbol{u_n}$.

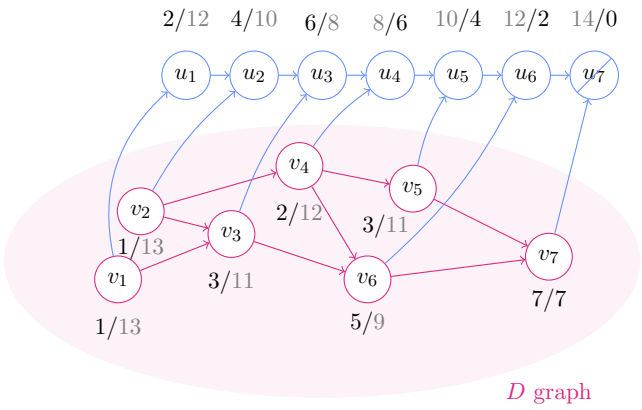


Figure 6: Illustration of the comb addition. The initial digraph is highlighted in pink.

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 to the topological ordering.

Theorem 6. *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.*

Proof. We keep the same notation as Definition 5.

Claim 6.1. *For all i, $\boldsymbol{u_i}$ has $2i$ ancestors, which are all the vertices $\boldsymbol{u_j}$ and $\boldsymbol{v_j}$ with $j \leq i$. The ancestors of $\boldsymbol{v_i}$ do not change.*

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

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

Claim 6.2. *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 $< n$. 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 resulting digraph after this query is either $comb(D)[\boldsymbol{u_1}, \dots, \boldsymbol{u_i}, \boldsymbol{v_1}, \dots, \boldsymbol{v_i}]$ or $comb(D)[\boldsymbol{u_{i+1}}, \dots, \boldsymbol{u_n}, \boldsymbol{v_{i+1}}, \dots, \boldsymbol{v_n}]$. (We use the notation $G[\boldsymbol{t_1}, \dots, \boldsymbol{t_\ell}]$ to denote the subgraph of G induced by the vertices $\boldsymbol{t_1}, \dots, \boldsymbol{t_\ell}$.)

Notice that in any case, the resulting digraph is of the form $comb(D')$. Indeed, we just have to choose $D' := D[\boldsymbol{v_1}, \dots, \boldsymbol{v_i}]$ or $D' := D[\boldsymbol{v_{i+1}}, \dots, \boldsymbol{v_n}]$, 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 strategy with this number of queries must be optimal.

Claim 6.3. *If n is odd, the `git bisect` algorithm necessarily uses $x + 1$ queries.*

By Claim 6.1, $\boldsymbol{v_n}$ has n ancestors, and digraph $comb(D)$ has $2n$ vertices. So the score of $\boldsymbol{v_n}$ is 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 $x + 1$. □

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 will depend on whether the first queried vertex is $\boldsymbol{v_n}$ or $\boldsymbol{u_{n/2}}$.

Moreover, remark that the comb construction shows that the worst-case number of queries required for searching the faulty commit in D can be lower than in a subgraph of D. Indeed, adding a comb to any digraph induces a logarithmic number of queries, even though the initial digraph uses more queries.

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 7. *For any integer $k > 2$, there exists a DAG such that the optimal number of queries is k, while the `git bisect` algorithm always uses $2^{k-1} - 1$ queries.*

Proof. Choose D as an octopus with $2^{k-1} - 1$ vertices. The number of `git bisect` queries is $2^{k-1} - 2$ (like every other strategy). The wanted digraph is then $comb(D)$ (see Figure 7 for an illustration). Indeed, by Theorem 6, the `git bisect` algorithm uses $2^{k-1} - 1$ `git bisect`

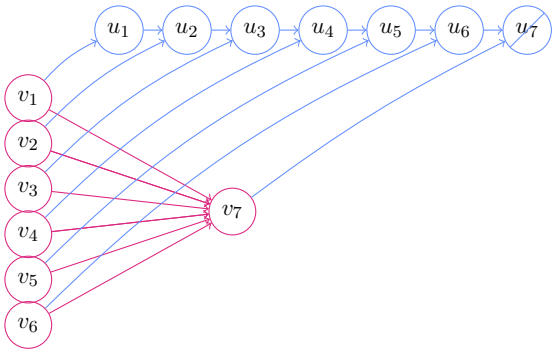


Figure 7: $Comb(D)$ graph where D is an octopus of size 7.

queries to find the faulty commit in $comb(D)$, while an optimal strategy uses $\lceil \log_2 (2^k - 2) \rceil = k$ queries. □

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

3 Approximation scheme 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 8 (Binary digraph)**.** *A digraph is* binary *if each vertex has indegree (that is the number of ingoing edges) at most equal to* 2*.*

(a) A binary DAG. (b) A non-binary DAG.

Figure 8

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

Theorem 9. *On any binary DAG with n vertices, the number of queries of the `git bisect` algorithm is at most* $\dfrac{\log_2(n)}{\log_2(\frac{3}{2})}$.

Corollary 10. *The algorithm* `git bisect` *is a* $\frac{1}{log_2(3/2)} \approx 1.71$ *approximation scheme on binary DAGs.*

3.2 Bounding the number of queries

The next lemma exhibits a core property of the binary DAGs, which will be a key ingredient of the following proofs. It states that if the DAG is binary, there must be a vertex with a good score.

Lemma 11. *In every binary DAG with n vertices, there exists a vertex $\boldsymbol{v}$ such that $|v|$, its number of ancestors, satisfies* $\frac{n-1}{3} < |v| \leq \frac{2n+1}{3}$ *(see Figure 9 for an illustration).*

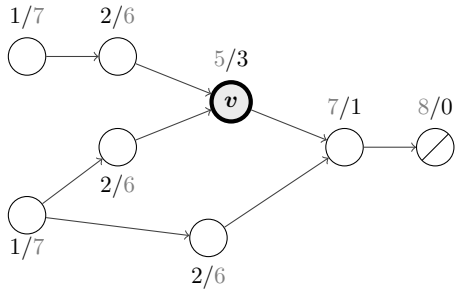


Figure 9: The highlighted vertex $\boldsymbol{v}$ is the only one to have its number of ancestors in $(\frac{n-1}{3}, \frac{2n+1}{3}]$, where $n = 8$.

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

Let us consider V^+ the set of vertices whose number of ancestors is greater or equal than its number of non-ancestors, and B^+ the subset of V^+ whose vertices have no parent in V^+.

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

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

Also, since $|v| \in V^+$, $|v| \geq n - |v|$ and thus, $|v| \geq \frac{n}{2}$. Thus $\frac{n}{2} \leq |v| < \frac{n}{2} + 1$ which satisfies the lemma whenever $n \geq 4$.

Vertex $\boldsymbol{v}$ has two parents $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively having $|x|$ and $|y|$ ancestors. For the same reasons as above, $\boldsymbol{x}$ and $\boldsymbol{y}$ are not in V^+ so $|x| < \frac{n}{2}$ and $|y| < \frac{n}{2}$.

If any of $\boldsymbol{x}$ or $\boldsymbol{y}$ has more than $\frac{n-1}{3}$ ancestors, then the lemma holds for $\boldsymbol{x}$ or $\boldsymbol{y}$.

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

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

Vertex $\boldsymbol{v}$ thus satisfies the lemma. □

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

Proof of Theorem 9. Let D be a DAG with n vertices, and D_k the digraph obtained from D after k `git bisect` queries. We denote by n_k the number of vertices in D_k. After each query, the `git bisect` algorithm chooses the vertex $\boldsymbol{v}$ given by Lemma 11 or a vertex with a better score. In any case, the score of the chosen vertex in D_k is greater or equal than $\frac{n_k - 1}{3}$. This is why

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

We can then show by induction that

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

Let x be the number of queries for `git bisect` so that x is the first number such that $n_x = 1$. If $n \in \{2, 3, 4, 5, 6, 7\}$, one can check with (1) that n_k is necessary smaller than 2 after $\lfloor \log_2(n) / \log_2(3/2) \rfloor$ steps. So without lost of generality, one can assume that $n > 7$.

If $x \leq 4$, then the proposition holds since $n > 7$ implies $4 < \log_{3/2}(n)$. After each query, we eliminate at least one vertex, so $n_{x-3} \geq 4$. Plugging $k = x - 4$ in (1), one obtains $n_{x-4} \geq \frac{3n_{x-3}-1}{2} \geq 5.5$, hence $n_{x-4} \geq 6$. Similarly, we have $n_{x-5} \geq 9$.

Notice when setting $k = \log_{3/2}\left(\frac{n-1}{8}\right)$ in the right member of (2), one obtains 9. So $x - 5$ must be less than $\log_{3/2}\left(\frac{n-1}{8}\right)$. Thus

$$x < 5 + \log_{\frac{3}{2}}\left(\frac{n-1}{8}\right) < \log_{\frac{3}{2}}(n-1) + \left(5 - \log_{\frac{3}{2}}(8)\right) < \log_{\frac{3}{2}}(n-1)$$

since $\log_{3/2}(8) \simeq 5.13\ldots$ □

3.3 Tight case

The above upper bound is asymptotically sharp, as stated by the following proposition.

Proposition 12. *For any integer k, there exists a* binary *DAG J_k such that*

- *the number of* `git bisect` *queries is $k + \lceil \log_2(k) \rceil + 3$;*
- *an optimal strategy uses at most $\log_2(\frac{3}{2})\, k + \log_2(3k+6) + 2$ queries.*

Consequently, the approximation ratio $1/\log_2(3/2)$ is considered to be optimal for `git bisect`.

Corollary 13. *For any $\varepsilon > 0$, the* `git bisect` *algorithm is not a $\left(\frac{1}{\log_2(3/2)} - \varepsilon\right)$ approximation scheme for binary DAGs.*

Proof of Proposition 12. We start by defining J_k^0, the *backbone* of J_k. It is formed by taking three directed paths on $k+1$ vertices $\boldsymbol{x_1} \to \boldsymbol{x_2} \to \cdots \to \boldsymbol{x_{k+1}}$, $\boldsymbol{y_1} \to \cdots \to \boldsymbol{y_{k+1}}$ and $\boldsymbol{z_0} \to \cdots \to \boldsymbol{z_k}$ and merging the three vertices $\boldsymbol{x_{k+1}}$, $\boldsymbol{y_{k+1}}$ and $\boldsymbol{z_0}$ into a vertex $\boldsymbol{c}$ (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

$$\ell_d = \begin{cases} \dfrac{n_{d-1}+2}{6} & \text{if } n_{d-1} \text{ is even,} \\ \dfrac{n_{d-1}+5}{6} & \text{if } n_{d-1} \text{ is odd,} \end{cases}$$

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 $\boldsymbol{x_{k+1-d}}$, $\boldsymbol{y_{k+1-d}}$, and $\boldsymbol{z_d}$. Then, add edges from the new parents of $\boldsymbol{x_{k+1-d}}$ and of $\boldsymbol{y_{k+1-d}}$ to the first vertex of the path newly attached to $\boldsymbol{z_d}$. Also, the new parent of $\boldsymbol{z_d}$ is denoted by $\boldsymbol{z'_d}$. The reader can refer to Figure 10 for a better understanding of the notation.

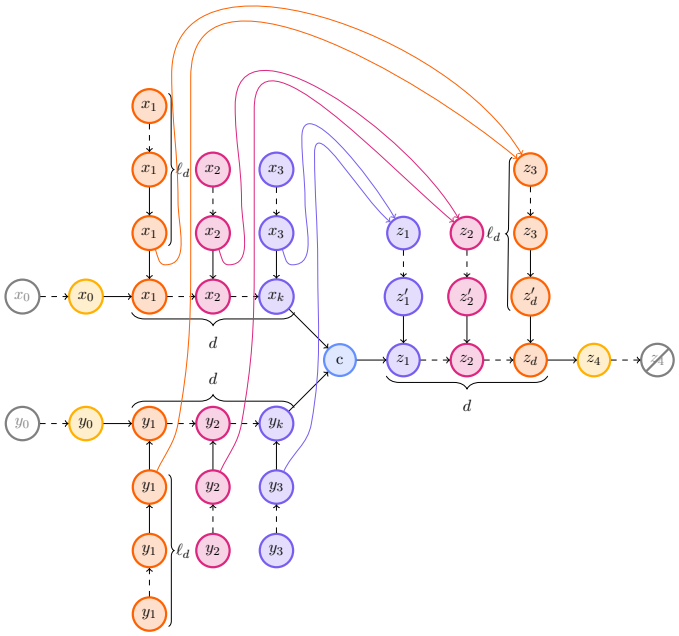
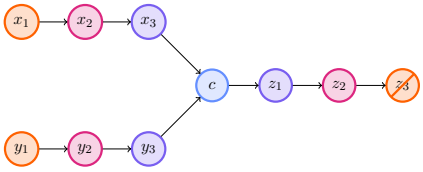


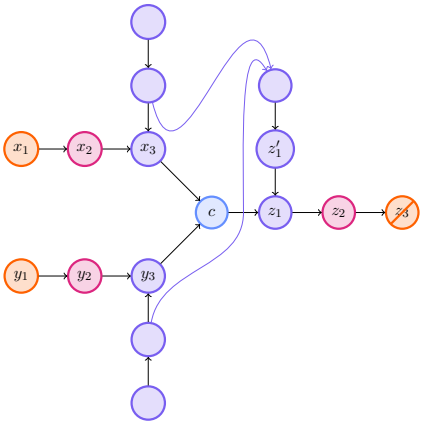
Figure 10: the d-th step in the construction of J_k.

We wish the number of vertices in the final graph J_k to be odd in order to use Theorem 6. 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. We denote by J_k the resulting digraph.

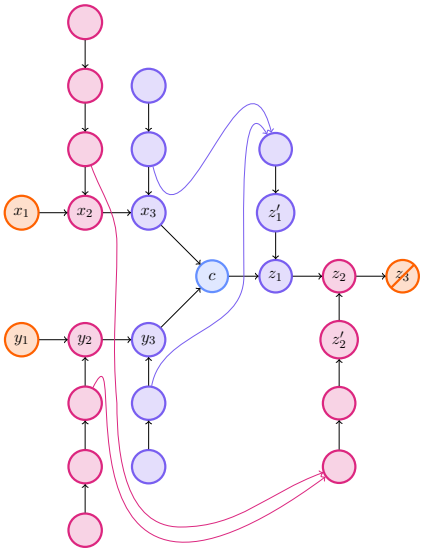
The construction for $k = 3$ is shown in Figure 11.



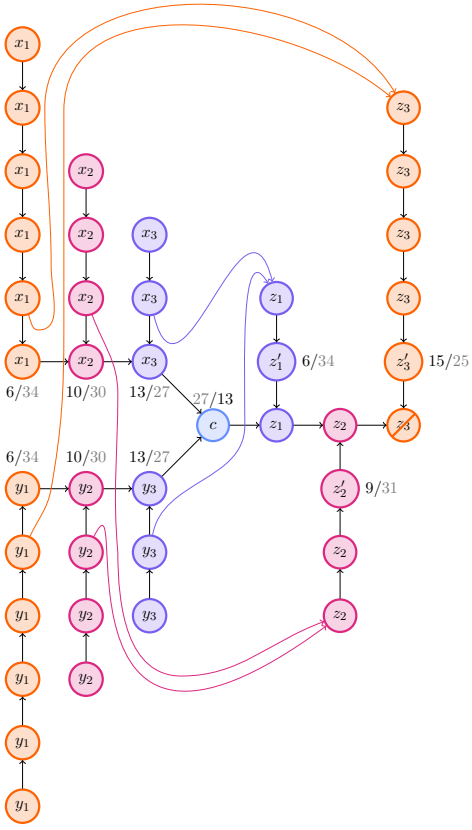
(a) Step 0: J_3^0



(b) Step 1: J_3^1



(c) Step 2: J_3^2



(d) Step 3: $J_3^3 = J_3$

Figure 11: Construction of J_3.

Why is ℓ_d always an integer? Notice that $n_d \equiv 1$ modulo 3, for each d. Indeed, it holds for the backbone J_k^0 since it has $3k+1$ vertices and, each step, we add $3\ell_d$ vertices. Thus, n_d is congruent to 1 (resp. 4) modulo 6 if n_d is odd (resp. even). Equivalently, $n_d + 5$ is a multiple of 6 whenever n_d is odd, like $n_d + 2$ whenever n_d is even. This is why ℓ_d is always an integer.

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

$$n_d = n_{d-1} + 3\ell_d \leq \frac{3}{2} n_{d-1} + \frac{5}{2}.$$

A quick induction shows that

$$n_d \leq \left(\frac{3}{2}\right)^d (3k+6) - 5.$$

Remember that, if n_k was even, we had added 3 vertices in the final digraph. In any case, the number of vertices in J_k is bounded by $\left(\frac{3}{2}\right)^d (3k+6)$.

Claim 13.1. *When $\boldsymbol{c}$ is the faulty commit, the `git bisect` algorithm uses $k + \lceil \log_2(k) \rceil + 2$ 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 $i \in \{0, \ldots, k\}$. In other words, after k `git bisect` queries, we will 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 vertices, for which `git bisect` uses $\lceil \log_2(k) \rceil$ queries to find the faulty commit. This explains why the number of `git bisect` queries is $k + 2 + \lceil \log_2(k) \rceil$.

To do so, we prove by induction on d some construction invariants:

- In J_k^d, vertices $\boldsymbol{x_k}$ and $\boldsymbol{y_k}$ have less ancestors than non-ancestors, while it is the inverse for $\boldsymbol{c}$.
- The scores of $\boldsymbol{x_k}$, $\boldsymbol{y_k}$ and $\boldsymbol{c}$ are the same in J_k^d.
- In J_k^d, the score of $\boldsymbol{x_k}$ is smaller than $n_d/3$.

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

Let us assume now the induction hypotheses true 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 thing 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 two invariants.

As for the score of $\boldsymbol{x_k}$, we just proved that it is equal to the number of ancestors of $\boldsymbol{x_k}$. So, by induction, the score of $\boldsymbol{x_k}$ in J_k^d is smaller than

$$\frac{n_{d-1}}{3} + \ell_d = \frac{n_{d-1} + 3\ell_d}{3} = \frac{n_d}{3},$$

which concludes the induction.

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 $\boldsymbol{x_k}$, or an ancestor of $\boldsymbol{y_k}$, or a descendant of $\boldsymbol{c}$, or an ancestor of a $\boldsymbol{z'_j}$ with $j \in \{1, \dots, m\}$. The vertex $\boldsymbol{x_k}$ having less ancestors than non-ancestors, the ancestors of $\boldsymbol{x_k}$ different from $\boldsymbol{x_k}$ have a worst 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 $j \in \{1, \dots, m\}$.

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, 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 the score of $\boldsymbol{z'_m}$ is equal to its number of ancestors, which is $3\ell_m$, which is bigger than $\frac{n_m}{3}$ 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}.

Claim 13.2. *Proposition 12 is satisfied by comb(J_k).*

It is a direct application of Theorem 6. Recall that the number of vertices in J_k is odd and is bounded by $\left(\frac{3}{2}\right)^d (3k+6)$. □

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 11, which was of paramount importance to understand the structure of binary DAGs, can be generalised as follows.

Lemma 14. *In every Δ-ary DAG with n vertices, there exists a vertex $\boldsymbol{v}$ such that $|v|$, its number of ancestors, satisfies* $\dfrac{n-1}{\Delta+1} < |v| \leq \dfrac{\Delta n + 1}{\Delta + 1}$.

This leads to the following theorem.

Theorem 15. *On any Δ-ary DAG with n vertices, the number of queries of the `git bisect` algorithm is at most* $\dfrac{\log_2(n)}{\log_2(\frac{\Delta+1}{\Delta})}$.

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

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` – theoretically at least.

4.1 Description of `golden bisect`

We design a new algorithm for the Regression Search Problem, which we name ***`golden bisect`***, which is a slight modification of `git bisect`. Thereby, it is not complicated to modify an existing implementation of `git bisect` to enhance it into `golden bisect`.

The difference of `golden bisect` with respect to `git bisect` is that it does not query a vertex with the best score, if the best score is too "low". Under these circumstances, it will restrict its queries to a well-suited subset of vertices. Roughly speaking, if queried, these vertices have the property that the new digraph will have at least a vertex with a really good score.

Let us describe the subsets in question.

Definition 16 (Subsets B^+ and B^-)**.** *Let D be a DAG. We define V^+ as the set of vertices which have more ancestors than non-ancestors. Let B^+ (for "Best" or "Boundary") denote the subset of vertices $\boldsymbol{v}$ of V^+ such that no parent of $\boldsymbol{v}$ belongs to V^+ and B^- be the set of parents of vertices of B^+.*

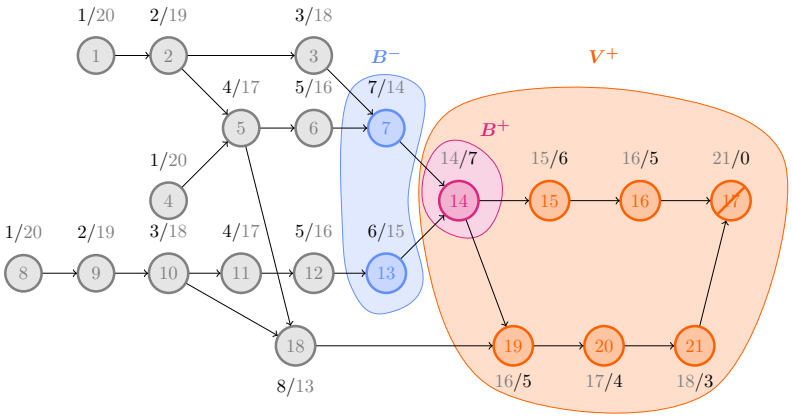


Figure 12: A binary DAG with the 3 sets of vertices V^+, B^+ and B^-.

Superscript + (resp. superscript −) indicates that the vertices have more ancestors (resp. less ancestors) than non-ancestors. The reader can look at Figure 12 for an illustrative example.

Remark that $B^+ \cup B^-$ is the set of vertices that we used to prove Lemma 11.

Lemma 17. *Given any DAG with n vertices, there exists a vertex $\boldsymbol{v} \in B^+ \cup B^-$ such that the score of $\boldsymbol{v}$ is at least $\dfrac{n-1}{3}$.*

Proof. Exactly the same proof as Lemma 11. □

Now, let us describe the `golden bisect` algorithm. It is so called because it is based on the *golden ratio*, which is defined as $\phi = \dfrac{1+\sqrt{5}}{2}$.

Algorithm 18 (`golden bisect`)**.**
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 best score is at least*** $\frac{n}{\phi^2} \approx 38.2\% \times n$ ***(where*** $\phi = \frac{1+\sqrt{5}}{2}$***), query a vertex with this score.***
5. ***Otherwise, query a vertex of*** $B^+ \cup B^-$ ***which has the best score among vertices of*** $B^+ \cup B^-$***, even though it is not the overall best 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.)*

For example, consider the digraph from Figure 12. We have $21/\phi^2 \approx 8.02$. The best score 8 is smaller than this number, so we run Step 5 instead of Step 4. Thus as its first query, `golden bisect` chooses indifferently **7** or **14**, which respectively belong to B^- and B^+, and which have score 7. It diverges from `git bisect`, which picked **18** (score 8) instead.

For a full example, the reader can refer to the strategy tree in Figure 13. Note that, even if it is different from `git bisect`, the `golden bisect` strategy uses 5 queries in the worst-case scenario.

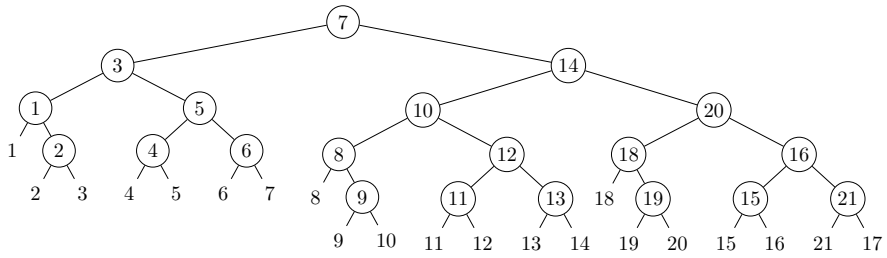


Figure 13: The `golden bisect` strategy tree for the digraph of Figure 12. In case of equality of score, the vertex with the least label is chosen.

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 7 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 have a better upper bound for the number of queries, in comparison with `git bisect`.

Theorem 19. *On any binary DAG with n vertices, the number of `golden bisect` queries is at most $\log_\phi(n) + 1 = \dfrac{\log_2(n)}{\log_2(\phi)} + 1$, where ϕ is the golden ratio.*

As first corollary, since no power of ϕ is an integer, the number of `golden bisect` queries for a binary DAG of size n is also at most $\lceil \log_\phi(n) \rceil = \left\lceil \frac{\log_2(n)}{\log_2(\phi)} \right\rceil$. We can also deduce that it is a better approximation scheme than `git bisect` (in the binary case):

Corollary 20. *For every $\varepsilon > 0$, `golden bisect` is a $\left(\dfrac{1}{\log_2(\phi)} + \varepsilon\right)$-approximation algorithm on binary DAGs with a sufficiently large size.*

Finally, this also gives an upper bound for the optimal number of queries in the worst-case scenario, given a binary DAG of size n.

Corollary 21. *For any binary DAG D with n vertices, the optimal number opt of queries for the Regression Search Problem satisfies*

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

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 14), that uses $\lceil \log_\phi(4) \rceil = 3$ queries in the worst-case scenario.

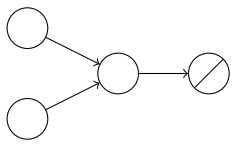


Figure 14: Claw graph.

4.3 Proof of the upper bound

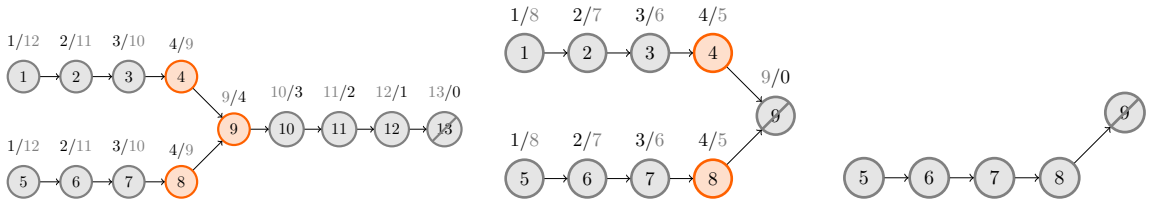
Recall that $\phi = \dfrac{1+\sqrt{5}}{2}$ is the golden ratio. We also have $1+\phi-\phi^2 = 0$, and thus $n - \frac{n}{\phi} = \frac{n}{\phi^2}$.

Lemma 22. *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.

Note that the lemma does not hold for $n = 13$, as shown by Figure 15. 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 $\frac{13}{\phi^2} \approx 4.96$.



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

Figure 15: First two steps of `golden bisect` for a DAG of size 13

Proof. If the best score of a vertex in D is at least $\frac{n}{\phi^2}$, then item (i) holds since there will remain at most $n - \frac{n}{\phi^2} = \frac{n}{\phi}$ vertices.

Let us suppose then that all vertices have score less than $\frac{n}{\phi^2}$ and show that (ii) is satisfied.

Then let $\boldsymbol{z}$ be the first vertex queried by `golden bisect`, and $|z|$ be its number of ancestors. In this case, $\boldsymbol{z}$ belongs to B^+ or B^-.

Figure 16 sketches all the cases in this proof.

Case 1: $\boldsymbol{z} \in B^+$. Since $\boldsymbol{z} \in V^+$ by hypothesis, its score corresponds to the number of non-ancestors, which is $n - |z|$. Thus, if $\boldsymbol{z}$ is clean, then after one step of `golden bisect`, one only keeps as many vertices as the score of $\boldsymbol{z}$, which is less than $\frac{n}{\phi^2}$, and (ii) holds.

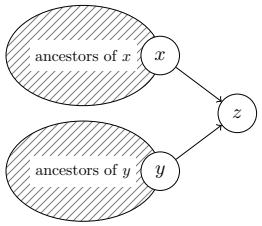
Suppose now that $\boldsymbol{z}$ carries a bug. Vertex $\boldsymbol{z}$ must have 2 parents. Indeed, if it has only one parent, then by the same reasoning as in the proof of Lemma 11, the score of $\boldsymbol{z}$ is an integer greater than $\frac{n}{2} - 1$, which is greater than $\frac{n}{\phi^2}$ for $n \geq 4$. This contradicts the hypothesis on the score of $\boldsymbol{z}$.

Let us denote by D' the DAG obtained from D after querying $\boldsymbol{z}$. The new marked sink is $\boldsymbol{z}$. Let us call $\boldsymbol{x}$ and $\boldsymbol{y}$ the parents of $\boldsymbol{z}$, and assume that $|x|$ and $|y|$, the respective numbers of ancestors of these vertices, satisfy $|x| \geq |y|$.

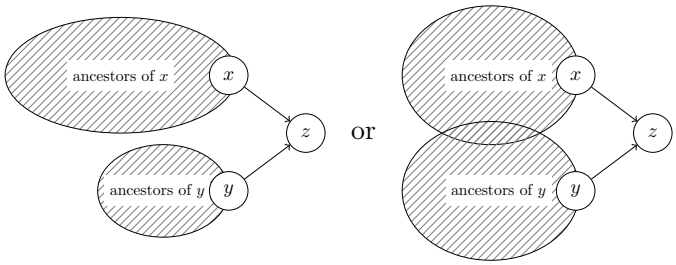
We study two complementary cases:

Case 1a: $|x| = |y|$ and $\boldsymbol{x}$ and $\boldsymbol{y}$ do not have any common ancestor. In this case, $|z| = |x| + |y| + 1 = 2|x| + 1$, and `golden bisect` will indifferently choose $\boldsymbol{x}$ or $\boldsymbol{y}$ as second query, which leaves $|x| + 1$ vertices.

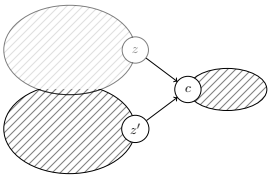
Since $\boldsymbol{z}$ has been preferred to $\boldsymbol{x}$ in D (and $\boldsymbol{x} \in B^-$), the score $|x|$ of $\boldsymbol{x}$ is no more than the score $n - |z|$ of $\boldsymbol{z}$. This translates by $|x| \leq n - (2|x| + 1)$, or $|x| + 1 \leq \frac{n-1}{3} + 1$. Since $\frac{n-1}{3} + 1 < \frac{n}{\phi^2}$ for $n \geq 14$, this case is proved.



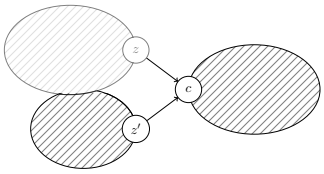
(a) **Case 1a:** $|x| = |y|$ and $\boldsymbol{x}$ and $\boldsymbol{y}$ does not have any common ancestor.



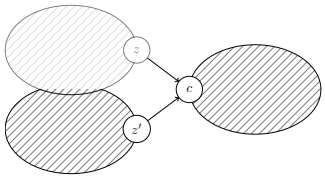
(b) **Case 1b:** $|x| \geq |y| + 1$ or the vertices $\boldsymbol{x}$ and $\boldsymbol{y}$ have the same number of ancestors and share a common ancestor.



(c) **Case 2a:** in D', number of ancestors of $\boldsymbol{z'}$ > number of non-ancestors of $\boldsymbol{c}$



(d) **Case 2b:** in D', number of ancestors of $\boldsymbol{z'}$ < number of non-ancestors of $\boldsymbol{c}$



(e) **Case 2c:** in D', number of ancestors of $\boldsymbol{z'}$ = number of non-ancestors of $\boldsymbol{c}$

Figure 16: Subcases of the proof in Lemma 22.

Case 1b: $|x| \geq |y|+1$ or the vertices x and y have the same number of ancestors and share a common ancestor. Both cases in the assumption give $|y| + 1 - |xy| \leq |x|$, where $|xy|$ stands for the number of common ancestors of $\boldsymbol{x}$ and $\boldsymbol{y}$. Since $|z| = 1 + |x| + |y| - |xy|$, we have that in both cases, $|z| - |x|$ is no more than $|x|$. Thus the score of $\boldsymbol{x}$ in D' is $|z| - |x|$.

We wish to show that in D', `golden bisect` selects a query according to Step 4. In other words, we want to prove that a vertex has a score in D' no less than $\dfrac{|z|}{\phi^2}$, more precisely that vertex $\boldsymbol{x}$ satisfies this condition.

First observe that in D, the score $|x|$ of $\boldsymbol{x}$ and the score $n - |z|$ of $\boldsymbol{z}$ are by hypothesis less than $\frac{n}{\phi^2}$. Thus, we have

$$|x| < \frac{n}{\phi^2} \quad \text{and} \quad |z| > n - \frac{n}{\phi^2} = \frac{n}{\phi}.$$

Therefore the score $|z| - |x|$ of $\boldsymbol{x}$ in D' satisfies

$$|z| - |x| > |z| - \frac{n}{\phi^2} = \frac{|z|}{\phi} + \frac{|z|}{\phi^2} - \frac{n}{\phi^2} = \frac{|z|}{\phi^2} + \frac{|z| - \frac{n}{\phi}}{\phi} > \frac{|z|}{\phi^2}.$$

Thus `golden bisect` will query $\boldsymbol{x}$ or a vertex with a better score. But querying $\boldsymbol{x}$ will keep $|x|$ vertices if $\boldsymbol{x}$ is bugged, and $|z| - |x|$ otherwise. So the number of remaining vertices after the second query is at most $|x| < \frac{n}{\phi^2}$, and (ii) is true.

Case 2: $\boldsymbol{z} \in B^-$. Since $\boldsymbol{z}$ belongs to B^-, it has a child in B^+. Denote it $\boldsymbol{c}$, and $|c|$ its number of ancestors. As above, we can assume that $\boldsymbol{c}$ has more than 1 parent, because

otherwise the score of $\boldsymbol{c}$ would be better than $\frac{n}{2} - 1$. Let $\boldsymbol{z'}$ be the other parent of $\boldsymbol{c}$ (also in B^-) and $|z'|$ its number of ancestors.

If $\boldsymbol{z}$ is bugged, then there remain at most $|z| < \frac{n}{\phi^2}$ vertices, which makes (i) true. So assume that the queried vertex $\boldsymbol{z}$ 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 $\boldsymbol{z}$. Note that D' has $n - |z|$ vertices.

Let us show first the following claim, which is going to be helpful for the two last subcases (2b and 2c).

Claim 22.1. *If there exists a vertex $\boldsymbol{v}$ of D' which have $\leq \frac{n}{\phi^2}$ ancestors in D' and $\leq \frac{n}{\phi^2}$ non-ancestors in D', then the score of $\boldsymbol{v}$ in D' is greater or equal than $\frac{n-|z|}{\phi^2}$. Thus `golden bisect` will query a vertex with the best score, as stated by Step 4.*

Proof. Let $\#a$ and $\#na$ be respectively the number of ancestors and non-ancestors of $\boldsymbol{v}$ in D' so that $\#a \leq \frac{n}{\phi^2}$ 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 $\frac{n-|z|}{\phi^2}$, the claim is proved.

Since there are $n - |z|$ vertices in D', we have

$$\#a + \#na = n - |z|.$$

So

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

(the last equality can be derived after some calculations from the identity $1 = \frac{1}{\phi} + \frac{1}{\phi^2}$). But the score $|z|$ of $\boldsymbol{z}$ in D is by hypothesis less than $\frac{n}{\phi^2}$. We then deduce that

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

The numbers $\#a$ and $\#na$ play symmetric roles in this claim, so we can similarly infer that $\#na \geq \frac{n-|z|}{\phi^2}$. Thereby we have finally proved that the score of $\boldsymbol{v}$ is $\geq \frac{n-|z|}{\phi^2}$. □

Let us resume the proof of Lemma 22.

Case 2a: in D', the number of ancestors of $\boldsymbol{z'}$ is greater than the number of non-ancestors of $\boldsymbol{c}$. In D', the number of ancestors of $\boldsymbol{z'}$ is $|z'|$ or less. But $|z'|$ was the score of $\boldsymbol{z'}$ in D and did not exceed $\frac{n}{\phi^2}$ in D. Moreover, the number of non-ancestors of $\boldsymbol{z'}$ in D' is equal to the number of non-ancestors of $\boldsymbol{c}$ plus one, which is by hypothesis less than or equal to the number of ancestors of $\boldsymbol{z'}$. Thus, the number of ancestors and the number of non-ancestors of $\boldsymbol{z'}$ in D' are no more than $\frac{n}{\phi^2}$.

So by Claim 22.1, `golden bisect` will query $\boldsymbol{z'}$ or a vertex with a better score. But if $\boldsymbol{z'}$ is queried, the resulting graph would have at most $|z'|$ vertices, which is less than $\frac{n}{\phi^2}$. Therefore (ii) holds.

Case 2b: in D', the number of ancestors of z' is less than the number of non-ancestors of c. The number of ancestors of $\boldsymbol{c}$ in D' is the number of ancestors of $\boldsymbol{z'}$ in D' plus one, which is less than or equal to $n - |c|$, the number of non-ancestors of c. Moreover, $n - |c|$ was also the score of $\boldsymbol{c}$ in D, which is less than $\frac{n}{\phi^2}$ by hypothesis.

So $\boldsymbol{c}$ satisfies the assumptions of Claim 22.1, which shows, that the second query is about $\boldsymbol{c}$, or a vertex with a better score. It yields a DAG with at most $n - |c|$ vertices, which is less than $\frac{n}{\phi^2}$. Here again (ii) holds.

Case 2c: in D', the number of ancestors of z' is equal to the number of non-ancestors of c. Note that every vertex of D is either an ancestor of $\boldsymbol{z}$ (number: $|z|$), or a non-ancestor of $\boldsymbol{c}$ (number: $n - |c|$), or an ancestor of $\boldsymbol{z'}$ in D' (number by hypothesis: $n - |c|$), or $\boldsymbol{c}$ (number: 1), hence $n = 2(n - |c|) + |z| + 1$.

But we have $n - |c| \leq |z|$ since the score of $\boldsymbol{c}$ in D is no more than the score of $\boldsymbol{z}$. Therefore $n \geq 3(n - |c|) + 1$ and $(n - |c|) + 1 \leq \frac{n-1}{3} + 1$, which is less than $\frac{n}{\phi^2}$ whenever $n \geq 14$. Thus $\boldsymbol{c}$ (like $\boldsymbol{z'}$) satisfies the hypotheses of Claim 22.1 and by the same reasoning as the previous cases, (ii) is satisfied. □

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

Proof of Theorem 19. 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 is a bit tedious since we need to prove it for $n \leq 13$. The idea is to use Lemma 17 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 $n - \left\lceil \frac{n-1}{3} \right\rceil$. The first values give the following array:

n	1	2	3	4	5	6	7	8	9	10	11	12	13
Bound for `golden bisect`	0	1	2	3	3	4	4	4	5	5	5	5	6
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

We remark that the second row is bounded by the last row. So the property holds for $n \leq 13$.

As for $n \geq 14$, the induction is straightforward by Lemma 22. Indeed, if (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 (ii) is satisfied, then it is also bounded by $2 + \left(\log_\phi\left(\frac{n}{\phi^2}\right) + 1\right) = 1 + \log_\phi(n)$. □

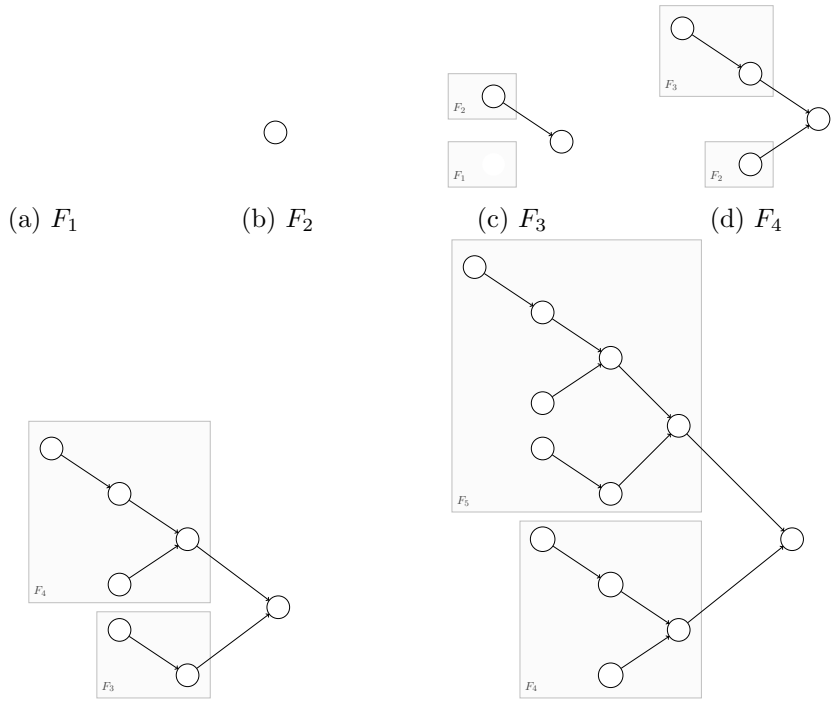
4.4 Fibonacci trees

In order to prove the sharpness of the constant $\frac{1}{\log_2(\phi)}$ from Corollary 20, we define a new family of digraphs: *Fibonacci trees*.

Definition 23 (Fibonacci trees)**.** *For $i \geq 0$, the i-th Fibonacci tree F_1 is defined as followed.*

F_1 is an empty tree, F_2 is a single vertex, and for $i \geq 3$, 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 17 shows the 6 first Fibonacci trees. We can establish an optimal strategy for the Fibonacci trees.



(a) F_1 (b) F_2 (c) F_3 (d) F_4

(e) F_5 (f) F_6

Figure 17: First Fibonacci trees

Theorem 24. *For any $i \geq 2$, the optimal strategy for the i-th Fibonacci tree F_i uses $i-2$ queries in the worst-case scenario.*

Proof. We decompose this proof in two claims.

Claim 24.1. *The `git bisect` algorithm and the `golden bisect` algorithm both use $i-2$ queries to find the faulty commit in F_i in the worst-case scenario.*

Proof. The best score in F_i is only achieved for the root of the subtree F_{i-1}, and it is equal to the size of F_{i-1} (which is $fib_i - 1$, where fib_i is the i-th Fibonacci number). So both `git bisect` and `golden bisect` will choose this vertex as first query. The worst-case scenario is whenever this vertex is bugged, and so whenever the graph after the first query is F_{i-1}. We then proceed to an induction and see there remain at this point $i-3$ queries. □

Claim 24.2. *If T is a tree strictly containing as disjoint copies the Fibonacci trees F_k and F_{k+1} (cf Figure 18 top), then, for any strategy searching for the faulty commit in T, there exists a vertex $\boldsymbol{v}$ in T such that this strategy uses at least k queries to find $\boldsymbol{v}$ as the faulty commit.*

Proof. We prove this claim by induction on k. It is clear for $k = 1$, since if T strictly includes F_2, which is a single vertex, as a copy, then we need at least 1 query to know whether this single vertex carries a bug or not.

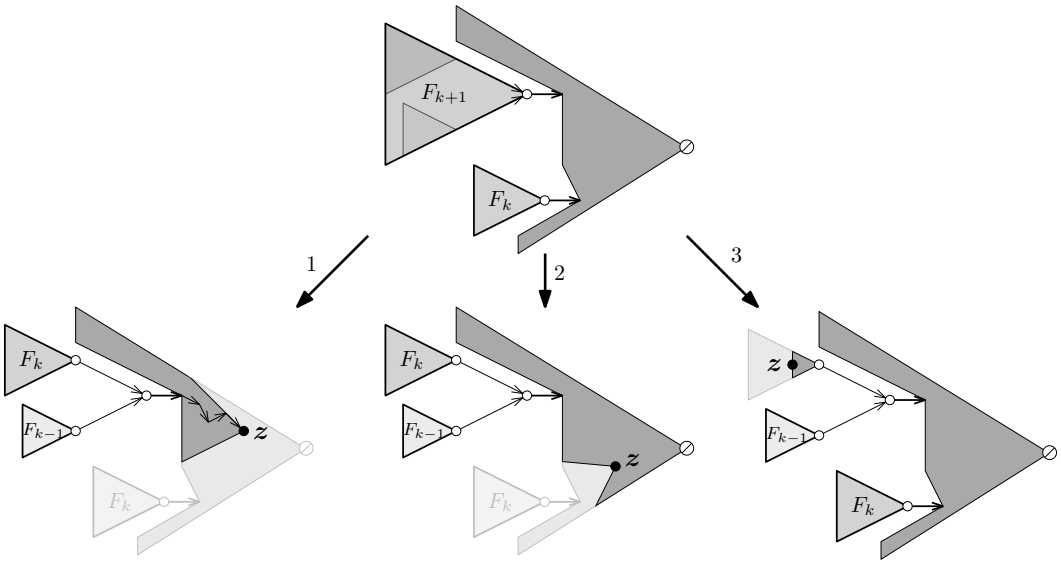


Figure 18: Illustration of the proof of Claim 24.2

Now suppose the claim statement true for an integer $k-1$, and consider a strategy for the Regression Search Problem on a tree T strictly containing F_{k+1} and F_k. Let $\boldsymbol{z}$ be the first query of this strategy. Let us investigate every possibility for $\boldsymbol{z}$ (the reader can refer to Figure 18 for an illustration):

1. **The root of the subtree F_{k+1} is an ancestor of $\boldsymbol{z}$.** Then we force the faulty commit $\boldsymbol{v}$ to be an ancestor of $\boldsymbol{z}$ (i.e. $\boldsymbol{z}$ is bugged). Then after querying $\boldsymbol{z}$, there remains all ancestors of $\boldsymbol{z}$, 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 $k-1$ extra vertices to find the faulty commit.

2. **The root of the subtree F_{k+1} is not an ancestor of $\boldsymbol{z}$ and $\boldsymbol{z}$ is not in the subtree F_{k+1}.** Here $\boldsymbol{v}$ will be a non-ancestor of $\boldsymbol{z}$ (i.e. $\boldsymbol{z}$ 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. **$\boldsymbol{z}$ is in the subtree F_{k+1}, but it is not its root.** We set $\boldsymbol{z}$ to be clean so that $\boldsymbol{v}$ will be among the non-ancestors of $\boldsymbol{z}$. The subtree F_{k+1} contains two disjoint copies of F_{k-1}, so at least a whole copy of F_{k-1} is included at the same time in the non-ancestors of $\boldsymbol{z}$ and in F_{k+1}. By hypothesis, T included also another copy of F_k. So the query of $\boldsymbol{z}$ leads to a tree containing F_k and F_{k-1}: the induction hypothesis indicates that we need $k-1$ other queries.

For each of these 3 possibilities, the strategy uses in total k queries, which concludes the induction. □

Conclusion of the proof of Theorem 24. The Fibonacci tree F_i contains disjoint copies of F_{i-1} and F_{i-2}. By Claim 24.2, any strategy, in particular an optimal one, uses at least

$i-2$ queries to find the faulty commit in F_i in the worst-case scenario. The optimal number of queries is then exactly $i-2$, because by Claim 24.1, `git bisect` and `golden bisect` use that many queries (and so are optimal). □

The first consequence of Theorem 24 is that the upper bound $\lceil \log_\phi(n) \rceil$ from Corollary 21 is asymptotically sharp:

Corollary 25. *Any strategy solving the Regression Search Problem for any Fibonacci tree of size $n \geq 7$ uses $\lceil \log_\phi(n) \rceil - 2$ queries.*

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. The only known graph to satisfy this is the 4-vertices claw (see Figure 14). We leave this as an open question.

Proof of Corollary 25. Let $|F_i|$ be the number of vertices of the i-th Fibonacci tree F_i. By construction, we have

$$|F_i| = |F_{i-1}| + |F_{i-2}| + 1, \quad |F_1| = 0 \quad \text{and} \quad |F_2| = 1.$$

This recurrence has for solution $|F_i| = fib_{i+1} - 1$, where fib_i is the i-th Fibonacci number. Using that $fib_{i+1} = \left(\phi^{i+1} - (-\phi)^{-i-1}\right)/\sqrt{5}$, we get

$$\log_\phi(|F_i|) = i + 1 - \log_\phi(\sqrt{5}) + \varepsilon_i,$$

where

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

which tends to 0 and is in absolute value smaller than 0.3 (a bit less than $2 - \log_\phi(\sqrt{5})$) whenever $i \geq 5$. If $i \geq 5$, we have $\lceil \log_\phi(|F_i|) \rceil = i + 1 - \lfloor \log_\phi(\sqrt{5}) + 0.3 \rfloor = i$. We conclude by Theorem 24. □

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 26. *For $\varepsilon > 0$, `golden bisect` is not a $\left(\frac{1}{\log_2(\phi)} - \varepsilon\right)$ approximation scheme.*

Proof. The idea is to add a comb (see Definition 5) to the i-th Fibonacci tree F_i to approach the $\frac{1}{\log_2(\phi)}$ ratio.

Theorem 6 indeed works similarly if we use `golden bisect` instead of `git bisect`. Thus, $comb(F_i)$ is a binary DAG for which:

- the number of `golden bisect` queries is $\lceil \log_\phi(|F_i|) \rceil - 1$ (see Corollary 25),
- 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 scheme, for any $\varepsilon > 0$. □

5 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 theoretic results than `git bisect`.

The natural next step will to conduct experimental studies. The authors are currently implementing `git bisect` and `golden bisect`, and are going to put them to the test thanks to benchmarks.

Notably, some open questions remain, and hopefully answers will be found through the experiments.

Here is a list of such open questions:

- Even if `golden bisect` is a better approximation algorithm than the `git bisect` algorithm, it does not mean that `golden bisect` is overall better than `git bisect`. Does there exist some instance of binary DAG for which `golden bisect` is worst than the `git bisect` algorithm?
- 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. Even if we choose $\boldsymbol{v_7}$ as first query, we would like to still query the vertices $\boldsymbol{u_i}$ in the comb. Could we improve `git bisect` by authorising such queries?
- When we restrict the DAGs to be binary, is the Regression Search Problem still NP-complete?
- Is it possible to remove the ε of the approximation ratio of `golden bisect` in Corollary 20?
- Is it possible to find a family of binary DAGs for which the regression search problem requires at least $\lceil \log_\phi(n) \rceil$ queries where n is the number of vertices?
- If we restrict the DAGs to be trees (oriented from the leaves to the root), is `git bisect` a good approximation algorithm? We conjecture that `git bisect` is a $2-$approximation scheme for trees. (We have found examples where the ratio (number of `git bisect` queries)/(optimal number of queries) is 2.)

Finally we envisage studying the number of queries in the worst-case scenario, but whenever the input DAG is taken at random. Indeed, most of the examples described in this paper are not very likely to exist in reality. The notion of randomness for a digraph emanating from

a VCS is therefore quite interesting and deserves to be developed. We could for example define a theoretical probabilist model based on existing workflows. It will be also quite useful to use random samplers for VCS repositories in order to constitute benchmarks on demand.

References

[1] Micah Adler and Brent Heeringa. Approximating optimal binary decision trees. *Algorithmica*, 62(3-4):1112–1121, 2012. URL: https://doi.org/10.1007/s00453-011-9510-9, doi:10.1007/s00453-011-9510-9.

[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. URL: https://doi.org/10.1142/S0218195998000175, doi:10.1142/S0218195998000175.

[3] Yosi Ben-Asher, Eitan Farchi, and Ilan Newman. Optimal search in trees, 1999. doi:10.1137/S009753979731858X.

[4] Jaroslav Bendík, Nikola Benes, and Ivana Cerna. Finding regressions in projects under version control systems. *CoRR*, 2017. URL: http://arxiv.org/abs/1708.06623, arXiv:1708.06623.

[5] Benoit Boissinot. *hg bisect mercurial manpage*. URL: https://www.mercurial-scm.org/wiki/BisectExtension.

[6] R. Carmo, J. Donadelli, Y. Kohayakawa, and E. Laber. Searching in random partially ordered sets. *Theoret. Comput. Sci.*, 321(1):41–57, 2004. URL: https://doi.org/10.1016/j.tcs.2003.06.001, doi:10.1016/j.tcs.2003.06.001.

[7] Christian Couder. Fighting regressions with git bisect, 2009. URL: https://git-scm.com/docs/git-bisect-lk2009.

[8] Dariusz Dereniowski, Adrian Kosowski, Przemysł aw Uznański, and Mengchuan Zou. Approximation strategies for generalized binary search in weighted trees. In *44th International Colloquium on Automata, Languages, and Programming*, volume 80 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages Art. No. 84, 14. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2017.

[9] Ehsan Emamjomeh-Zadeh, David Kempe, and Vikrant Singhal. Deterministic and probabilistic binary search in graphs. In *STOC'16—Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, pages 519–532. ACM, New York, 2016. URL: https://doi.org/10.1145/2897518.2897656, doi:10.1145/2897518.2897656.

[10] Laurent Hyafil and Ronald L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Lett.*, 5(1):15–17, 1976/77. doi:10.1016/0020-0190(76)90095-8.

[11] 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. doi:10.1016/j.dam.2004.06.002.

[12] Shay Mozes, Krzysztof Onak, and Oren Weimann. Finding an optimal tree searching strategy in linear time. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1096–1105. ACM, New York, 2008.

[13] Krzysztof Onak and Pawel Parys. Generalization of binary search: Searching in trees and forest-like partial orders. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2006), 21-24 October 2006, Berkeley, California, USA, Proceedings*, pages 379–388. IEEE Computer Society, 2006. URL: `https://doi.org/10.1109/FOCS.2006.3`, `doi:10.1109/FOCS.2006.32`.