THEORETICAL ANALYSIS OF GIT BISECT

Julien COURTEIL (Université de Caen Normandie)

with Paul DORBEC and Romain LECOQ (Université de Caen Normandie)

HERE I AM!

I SAID “GIT BISECT,” NOT “GIT BEEFSTEAK”!

Séminaire Online du LIGM – 27 avril 2021
Part 1  Looking for the Original Bug

**Meat:**
Are we sure to keep on doing that "joke"?

**Other:**
Yes, I'm very subversive.

**Meat:**
DID YOU CALL ME?

**Other:**
Yes, I'm very subversive.
Simple question: What do you use to share files with your coauthors?
Different ways to share files within a project
Git and its commit graph

Git is a distributed version control system where the revisions (or "commits") are arranged as a Directed Acyclic Graph (DAG).

Diagram:
- Master to V1.0
- Develop to V1.0
- Develop to V1.1
- Feature 1 to V1.1
- Feature 2 to V1.1
- Feature 3 to V1.1
- Develop to V2.0
- V1.1 to V2.0
Problem: Finding the source of a bug

Input: A commit graph in which a commit is known to be bugged, the other commits may be bugged or bug-free.

Question: Which commit has originally introduced the bug?
PROBLEM: FINDING THE SOURCE OF A BUG

Input: A commit graph in which a commit is known to be bugged, the other commits may be bugged or bug-free.

Question: Which commit has originally introduced the bug?
Input: A commit graph in which a commit is known to be bugged, the other commits may be bugged or bug-free.

Question: Which commit has originally introduced the bug?
**Problem:** Finding the source of a bug

**Input:** A commit graph in which a commit is known to be bugged, the other commits may be bugged or bug-free.

**Question:** Which commit has originally introduced the bug?

**Assumptions:**
- If a parent of a commit is bugged, then the commit is bugged. *(Monotonous hypothesis)*
- Only one commit has introduced the bug, namely the original bug.
HOW TO CATCH THE FIRST BUG

Only operation: Query of a commit with unknown status
HOW TO CATCH THE FIRST BUG

Only operation: Query of a commit with unknown status

If **bugged**, then the original bug is an ancestor of this commit.

ancestor of a vertex $v = v$ or an ancestor of a parent of $v$.
How to Catch the First Bug

Only operation: Query of a commit with unknown status

If bugged,
then the original bug is an ancestor of this commit

If bug-free,
then the original bug is not an ancestor of this commit
**How to Catch the First Bug**

*Only operation*: Query of a commit with unknown status

If **bugged**, then the original bug is an ancestor of this commit.

If **bug-free**, then the original bug is not an ancestor of this commit.

The original bug is found whenever the remaining graph has only 1 vertex.
**PRECISE DEFINITION OF THE PROBLEM**

**Input:** a DAG where each vertex has an unknown status, except one, which is **bugged**, such that every vertex is an ancestor of this **bugged** vertex.

**Output:** A strategy that finds the **original bug** with a minimal number of queries in the worst-case scenario = optimal strategy.
**PRECISE DEFINITION OF THE PROBLEM**

**Input:** a DAG where each vertex has an unknown status, except one, which is **bugged**, such that every vertex is an ancestor of this **bugged** vertex.

**Output:** A strategy that finds the **original** bug with a minimal number of queries in the worst-case scenario = optimal strategy.

---

In real life, queries are costly.
FIRST EXAMPLE: A CHAIN

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8?
FIRST EXAMPLE: A CHAIN
FIRST EXAMPLE: A CHAIN

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

Query on 4: bug-free
FIRST EXAMPLE: A CHAIN

Query on 4: bug-free
FIRST EXAMPLE: A CHAIN

Query on 4: bug-free

Query on 6: bugged
FIRST EXAMPLE: A CHAIN

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

Query on 4: bug-free

1😊 → 2😊 → 3😊 → 4😊 → 5 → 6 → 7 → 8

Query on 6: bugged

1😊 → 2😊 → 3😊 → 4😊 → 5 → 6😊 → 7😊 → 8

Query on 5: bug-free

1😊 → 2😊 → 3😊 → 4😊 → 5😊 → 6😊 → 7😊 → 8

culprit
FIRST EXAMPLE: A CHAIN

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

Query on 4: bug-free

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

Query on 6: bugged

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

Query on 5: bug-free

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

Optimal strategy = binary search

More generally, number of queries in an optimal strategy for a chain of length $n = \lceil \log_2(n) \rceil$
SECOND EXAMPLE: A RAKE
SECOND EXAMPLE: A RAKE

Optimal strategy = whatever

More generally, number of queries in an optimal strategy for a rake of size \( n = n-1 \)
STRATEGY TREE

Strategy tree for binary search:

1 — 2 — 3 — 4 — 5 — 6 — 7 — 8

4 bugged

1 — 2 — 3 — 4

2 bugged

1 bugged 1 bug-free

1 — 2 — 3 — 4

2 bug-free

3 bugged 3 bug-free

5 — 6 — 7 — 8

6 bugged 6 bug-free

5 — 6 — 7 — 8

5 bugged 5 bug-free

7 bugged 7 bug-free
STRATEGY TREE

Strategy tree for binary search:

In short:
STRATEGY TREE

Strategy tree for binary search:

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

2 bug-free

4 bugged

6 bug-free

6 bugged

4 bug-free

5 bug-free

7 bug-free

5 bugged

3 bug-free

2 bug-free

In short:

height of a strategy tree = number of requests in the worst-case scenario
COMPLEXITY OF THE PROBLEM

Finding the number of queries in an optimal strategy is ...

- **NP-complete** for general DAGs [Carmo Donadelli Kohayakawa Labor 2004]
  
  Certificate: Strategy tree
  Reduction to: Cover by 3-sets

- but **polynomial** for trees ...
  [Ben-Asher Farchi Newman 2000]

  ... more precisely, linear. [Mozes Onak Weimann 2008]
PART II - GIT
BI-FRICKING-SECT

I'M VERY GOOD TO ATTRACT BUGS

IT ISN'T VERY RARE IF IT'S WELL DONE.

NICE TO MEAT YOU!
LET ME INTRODUCE YOU GIT BISECT

Git uses a heuristic algorithm to find the original bug: git bisect

originally written by Linus Torvalds himself
now maintained by Junio Hamano

In the source code of `git bisect`:

```c
/*
 * This is a truly stupid algorithm, but it's only
 * used for bisection, and we just don't care enough.
 */
```
LET ME INTRODUCE YOU GIT BISECT

*git* uses a heuristic algorithm to find the original bug: *git bisect*

Step 1: Compute the number of ancestors/number of non-ancestors for each vertex:
LET ME INTRODUCE YOU GIT BISECT

**git** uses a heuristic algorithm to find the original bug: **git bisect**

**Step 1:** Compute the number of ancestors/number of non-ancestors for each vertex:

```
1/9 -> 2/8
     ↓    ↓
     ↘    ↘
   ?     ?
     ↑    ↑
   ?     ?
     ↓    ↓
   2/8   4/6
     ↓    ↓
   ?     ?
     ↓    ↓
   4/6   5/5
     ↓    ↓
   ?     ?
     ↓    ↓
   8/2   6/4
     ↓    ↓
   ?     ?
     ↓    ↓
   ?     ?
     ↓    ↓
   ?     ?
     ↓    ↓
   9/1   x
```

**Step 2:** Query on a vertex with the most balanced ratio (max of both numbers)

**Step 3:** Recurse
**LET ME INTRODUCE YOU TO GIT BISECT**

Git uses a heuristic algorithm to find the original bug: **git bisect**

**Step 1:** Compute the number of ancestors/number of non-ancestors for each vertex:

![Diagram of a tree with numbers and question marks](image)

**Step 2:** Query on a vertex with the most balanced ratio (max of both numbers)

**Step 3:** Recurse
LET ME INTRODUCE YOU GIT BISECT

Git uses a heuristic algorithm to find the original bug: `git bisect`

**Step 1:** Compute the number of ancestors/number of non-ancestors for each vertex:

**Step 2:** Query on a vertex with the most balanced ratio (max of both numbers)

**Step 3:** Recurse
Let me introduce you to Git Bisect.

Git uses a heuristic algorithm to find the original bug: Git bisect.

Step 1: Compute the number of ancestors/number of non-ancestors for each vertex:

Step 2: Query on a vertex with the most balanced ratio (max of both numbers).

Step 3: Recurse.
**Let Me Introduce You To Git Bisect**

Git uses a heuristic algorithm to find the original bug: `git bisect`.

**Step 1:** Compute the number of ancestors/number of non-ancestors for each vertex:

1. **Step 2:** Query on a vertex with the most balanced ratio (max of both numbers).

2. **Step 3:** Recurse.
LET ME INTRODUCE YOU GIT BISECT

**git** uses a heuristic algorithm to find the original bug: **git bisect**

**Step 1:** Compute the number of ancestors/number of non-ancestors for each vertex:

```
1/2  
2/1  
3/0  
```

**Step 2:** Query on a vertex with the most balanced ratio (max of both numbers)

**Step 3:** Recurse
Let me introduce you Git Bisect

Git uses a heuristic algorithm to find the original bug: Git Bisect

**Step 1:** Compute the number of ancestors/number of non-ancestors for each vertex:

**Step 2:** Query on a vertex with the most balanced ratio (max of both numbers)

**Step 3:** Recurse
**LET ME INTRODUCE YOU** **GIT BISECT**

*git* uses a heuristic algorithm to find the original bug: *git bisect*

**Step 1:** Compute the number of ancestors/number of non-ancestors for each vertex:

```
```

**Step 2:** Query on a vertex with the most balanced ratio (max of both numbers)

**Step 3:** Recurse
LET ME INTRODUCE YOU GIT BISECT

Git uses a heuristic algorithm to find the original bug: `git bisect`

**Step 1:** Compute the number of ancestors/number of non-ancestors for each vertex:

**Step 2:** Query on a vertex with the most balanced ratio (max of both numbers)

**Step 3:** Recurse
How good is Git Bisect?

Strategy tree of git bisect for this example:

Number of queries in the worst-case scenario of git bisect: 4
Number of queries of an optimal strategy: 4
How good is git bisect?

Strategy tree of git bisect for this example:

Number of queries in the worst-case scenario of git bisect: 4
Number of queries of an optimal strategy: 4

Question: Can git bisect always find an optimal strategy?
How good is git bisect?

Question: Can git bisect always find an optimal strategy?

Quick answer: absolutely no, it can’t
How good is Git Bisect?

**Question**: Can git bisect always find an optimal strategy?

**Quick answer**: absolutely no, it can't

**Proposition**: For any $k$, there exists a DAG such that an optimal strategy uses $k$ queries and git bisect always uses $2^{k-1} - 1$ queries.
A COUNTER-EXAMPLE

Proposition: For any $k$, there exists a DAG such that an optimal strategy uses $k$ queries and git bisect always uses $2^{k-1} - 1$ queries.

Proof for $k = 4$:

- Rake of size $2^{k-1} - 1$ to $7' \times$ of size $2^{k-1} - 1$
A COUNTER-EXAMPLE

Proposition
For any \( k \), there exists a DAG such that an optimal strategy uses \( k \) queries and git bisect always uses \( 2^{k-1} - 1 \) queries.

Proof for \( k = 4 \)

Proof of size \( 2^{k-1} - 1 \)

comb of size \( 2^{k-1} - 1 \)
BACK TO REALITY?

Rake substructures are unrealistic

Classical git graph:

- **Master**
  - V1.0
  - V1.1
  - V2.0
- **Develop**
- **Feature 1**
- **Feature 2**
- **Feature 3**

Usually, we never merge more than 2 branches.

(Otherwise, it is called an octopus merge.)
PART III - GIT BISECT ON BINARY DAGs

BYTE ME!
**Definition**

binary DAG = DAG where the vertices have indegree ≤ 2

**Ex:**

Good

Bad

**Theorem**

git bisect is a $\frac{1}{\log_2\left(\frac{3}{2}\right)}$-approximation algorithm when it is used on binary DAGs.

$\frac{1}{\log_2\left(\frac{3}{2}\right)} \approx 1.71$ is the optimal constant.
**EXISTENCE OF A BALANCED VERTEX**

**Lemma**  
In any binary graph of length $n$, there exists a vertex such that its number $x$ of ancestors satisfies

$$\frac{n-1}{3} \leq x \leq \frac{2n+1}{3}$$
EXISTENCE OF A BALANCED VERTEX

Lemma In any binary graph of length $n$, there exists a vertex such that its number $x$ of ancestors satisfies
\[
\frac{m-1}{3} \leq x \leq \frac{2n+1}{3}
\]

Where is it? Consider $\nu = \text{vertex with the least number } x \text{ of ancestors among those that has more ancestors than non-ancestors.} \ (\frac{m}{2} \leq x)$

The wanted vertex must be $\nu$ or one of its parents.
EXISTENCE OF A BALANCED VERTEX

Lemma In any binary graph of length n, there exists a vertex such that its number x of ancestors satisfies
\[ \frac{n-1}{3} \leq x \leq \frac{2n+1}{3} \]

Proof of the \( \frac{1}{\log_2 \left( \frac{3}{2} \right)} \) -approximation:
EXISTENCE OF A BALANCED VERTEX

Lemma: In any binary graph of length \( n \), there exists a vertex such that its number \( x \) of ancestors satisfies

\[
\frac{n-1}{3} \leq x \leq \frac{2n+1}{3}
\]

Proof of the \( \frac{1}{\log_2 \left( \frac{3}{2} \right)} \)-approximation:

At each step, \textit{gib bisect} chooses a query which eliminates at least \( 1/3 \) of the vertices.
EXISTENCE OF A BALANCED VERTEX

Lemma In any binary graph of length $n$, there exists a vertex such that its number $x$ of ancestors satisfies

$$\frac{n-1}{3} \leq x \leq \frac{2n+1}{3}$$

Proof of the $\frac{1}{\log_2 \left( \frac{3}{2} \right)}$-approximation:

At each step, git bisect chooses a query which eliminates at least $1/3$ of the vertices.

$$\text{number of git bisect queries} \approx \log_2 \left( \frac{3}{2} \right) (m)$$

$$\text{optimal number of queries} \geq \log_2 (m)$$
TIGHTENING THE BOUND

The hard part: proving that $\frac{1}{\log_2(\frac{3}{2})}$ is optimal

Existence of a problematic binary DAG for git bisect?

**Proposition**

Let $k$ be any number.

There exists a binary DAG such that
- number of git bisect queries = $k + \lceil \log_2(k) \rceil + 2$
- optimal number of queries $\leq k \log_2(\frac{3}{2}) + \log_2(3k+6) + 4$
STEP 1: MAXIMIZING THE NUMBER OF GB QUERIES

Binary DAG such that git bisect eliminates $\frac{1}{3}$ of its vertices for the $k$ first steps?

Construction of an example for $k = 3$: 

STEP 1: MAXIMIZING THE NUMBER OF GB QUERIES

Binary DAG such that git bisect eliminates $\frac{1}{3}$ of its vertices for the $k$ first steps?

Construction of an example for $k = 3$:

Step 0

$\mathbf{n}_0 = 10$

Step 1

$\mathbf{n}_1 = 16$

Step 2

$\mathbf{n}_2 = 25$

Step 3

Rule $\mathbf{n}_i = \text{nb of vertices at step } i$

Add 3 chains of length

\[
\begin{cases} 
\frac{\mathbf{n}_i + 2}{6} & \text{if } \mathbf{n}_i \text{ is even} \\
\frac{\mathbf{n}_i + 5}{6} & \text{if } \mathbf{n}_i \text{ is odd}
\end{cases}
\]
STEP 1: MAXIMIZING THE NUMBER OF GB QUERIES

Binary DAG such that git bisect eliminates $\frac{1}{3}$ of its vertices for the $k$ first steps?

Construction of an example for $k = 3$:

Step 0

$m_0 = 10$

Step 1

$n_1 = 16$

Step 2

$n_2 = 25$

Step 3

Rule $m_i = \text{nb of vertices at step } i$

Add 3 chains of length $\begin{cases} \frac{m_i + 2}{6} & \text{if } m_i \text{ is even} \\ \frac{m_i + 5}{6} & \text{if } m_i \text{ is odd} \end{cases}$

$\simeq 3k \times \left(\frac{3}{2}\right)^k$ vertices

number of git bisect queries $\geq k$
STEP 1: MAXIMIZING THE NUMBER OF GB QUERIES

Binary DAG such that git bisect eliminates $\frac{1}{3}$ of its vertices for the first $k$ steps?

Construction of an example for $k = 3$:

\[ \leq 3k \times \left(\frac{2}{3}\right)^k \text{ vertices} \]

\[ \text{number of git bisect queries} \geq k \]
STEP 2: TRAPPING GIT BISECT

**Theorem**

Let $D$ be a DAG with $n$ vertices such that git bisect uses $g_0$ queries. There exists a DAG $\text{Comb}(D)$ with $2n$ vertices such that an optimal strategy uses $\lceil \log_2(n) \rceil + 1$ queries and if $n$ is odd, git bisect uses $g_0+1$ queries.

**Proof:**

We assume that the vertices are topologically sorted: $1 < 2 < 3 < \ldots < 7$
A BETTER ALGORITHM? THE GOLDEN BISECTION

theoretical analysis of git bisect \rightarrow new (better?) algorithm
A BETTER ALGORITHM? THE GOLDEN BISECTION

Theoretical analysis of git bisect -> new (better?) algorithm

**THE GOLDEN BISECTION**

Step 1: Compute the number of ancestors/number of non-ancestors for each vertex:

```
[Middle edge diagram]
```

Step 2: \( M = \{ \text{vertices with the least number of ancestors amongst those that have more ancestors than non-ancestors} \} \)

Query on a vertex with the most balanced ratio amongst vertices of \( M \) or parents of vertices of \( M \).

Step 3: Recurse
The theoretical analysis of the golden bisection

Theorem: The golden bisection is a \( \frac{1}{\log_2(\phi)} \)-approximation algorithm for binary DAGs, where \( \phi \) = golden ratio \( \frac{1}{\log_2(\phi)} \approx 1.44 \) is the optimal constant.

Problematic DAG for the golden bisection = \text{Comb(Fib}_n\text{)}
where \( \text{Fib}_n \) is the \( n \)-th Fibonacci tree.

\[
\begin{pmatrix}
\text{Fib}_0 = 0 & \text{Fib}_1 = 0 \rightarrow 0 & \text{Fib}_n =
\end{pmatrix}
\]

\[
\begin{array}
\text{Fib}_{n-1} \\
\text{Fib}_{n-2}
\end{array}
\]
PERSPECTIVES

→ Write an article

→ Experimental results: git bisect vs golden bisection

→ Average-case analysis
  o Good model of random git graph (not Erdős–Rényi)?
  o How to sample them
  o Theoretical analysis of git bisect

→ Efficiency of git bisect on trees?
  Conjecture: git bisect = 2-approximation on trees
THANK YOU!

SHOULD I SAY I'M A GOOD BYE-STEAK?

YOU SHOULDN'T SAY ANYTHING.