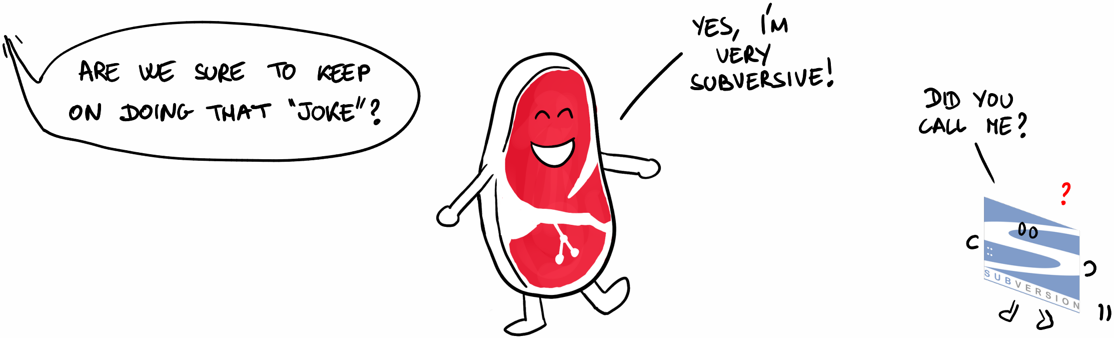
THEORETICAL ANALYSIS OF GIT BISECT

Julien COURTIEL (Université de Caen Normandie)

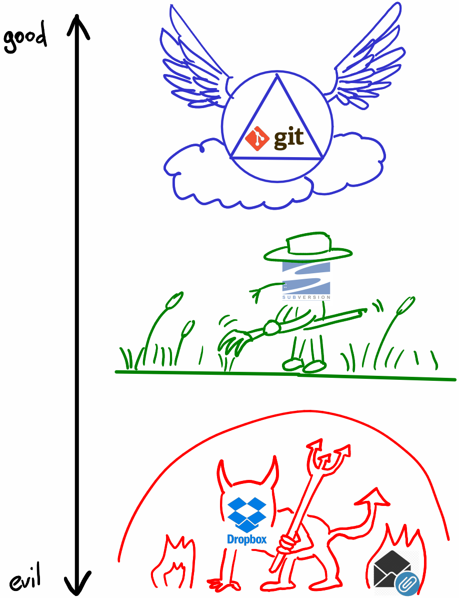
with Paul DORBEC and Romain LECOQ (Université de Caen Normandie)

Séminaire Online du LIGM – 27 avril 2021

Part 1 Looking for the Original Bug



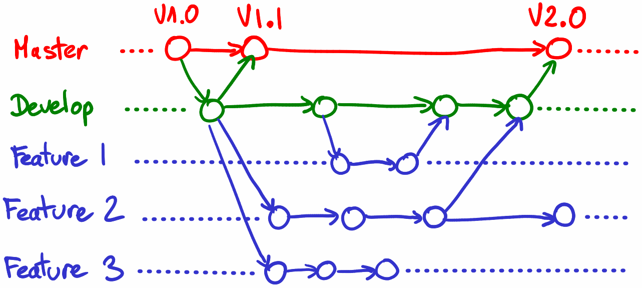
SIMPLE QUESTION: WHAT DO YOU USE TO SHARE FILES WITH YOUR COAUTHORS?



Different ways to share files within a project

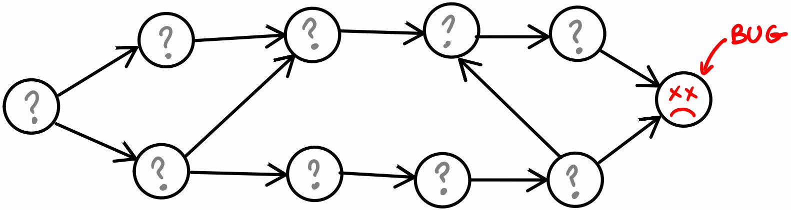
GIT AND ITS COMMIT GRAPH

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



Commit graph

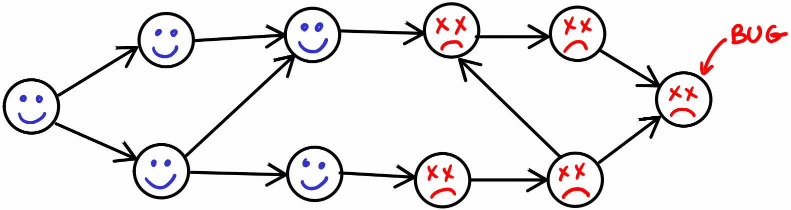
PROBLEM : FINDING THE SOURCE OF A BUG



$\boxed{\text{Input}}$ A commit graph in which a commit is known to be bugged, the other commits may be bugged or bug-free.

$\boxed{\text{Question}}$ Which commit has originally introduced the bug?

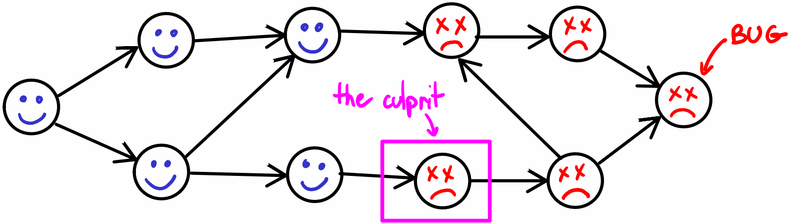
PROBLEM : FINDING THE SOURCE OF A BUG



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?

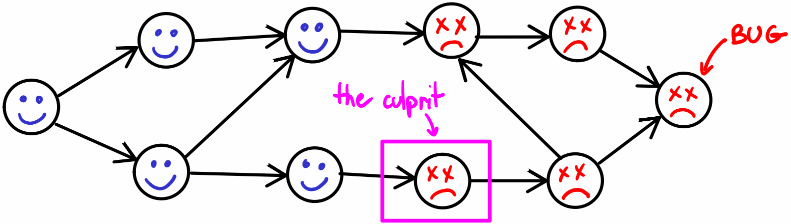
PROBLEM : FINDING THE SOURCE OF A BUG



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



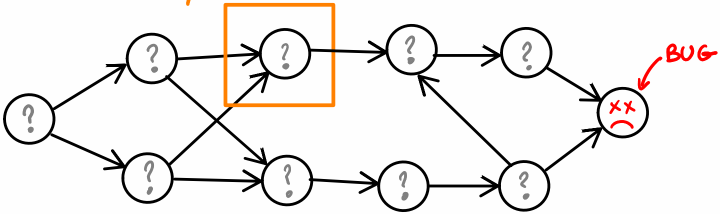
Input A commit graph in which a commit is known to be bugged, the other commits may be bugged or bug-free.

Question Which commit has originally introduced the bug?

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.

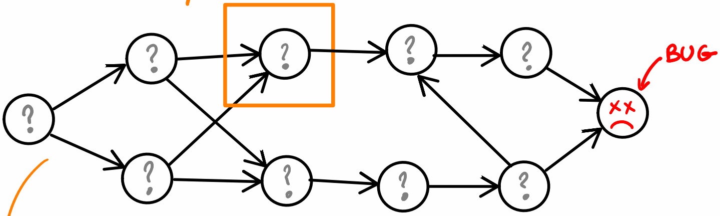
HOW TO CATCH THE FIRST BUG

Only operation: Query of a commit with unknown status



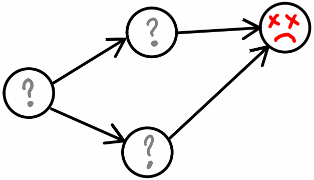
HOW TO CATCH THE FIRST BUG

<u>Only operation</u>: Query of a commit with unknown status



If <u>bugged</u>,

then the original bug is an ancestor of this commit



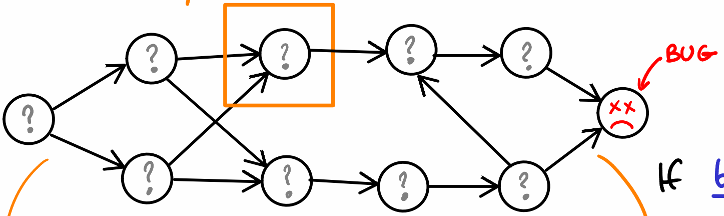
<u>ancestor of a vertex v</u> =

v <u>or</u>

an ancestor of a parent of v

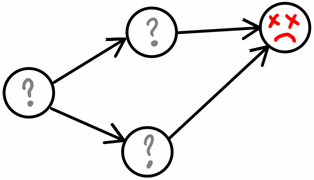
HOW TO CATCH THE FIRST BUG

Only operation: Query of a commit with unknown status



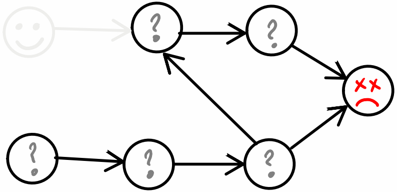
If bugged,

then the original bug is an ancestor of this commit



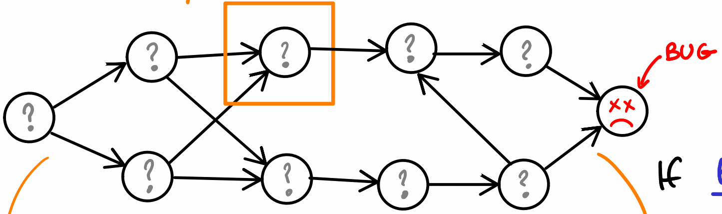
If bug-free,

then the original bug is not an ancestor of this commit



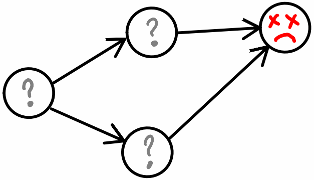
HOW TO CATCH THE FIRST BUG

Only operation: Query of a commit with unknown status



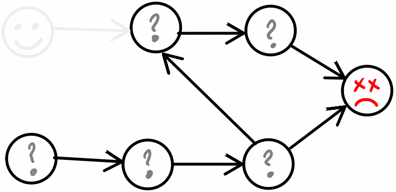
If bugged,

then the original bug is an ancestor of this commit



If bug-free,

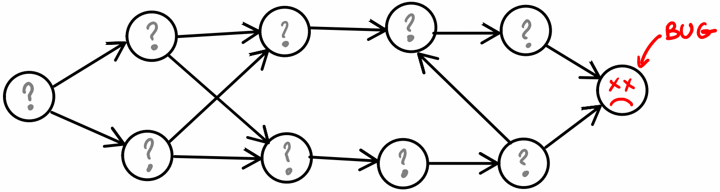
then the original bug is not an ancestor of this commit



The original bug is found whenever the remaining graph has only 1 vertex

PRECISE DEFINITION OF THE PROBLEM

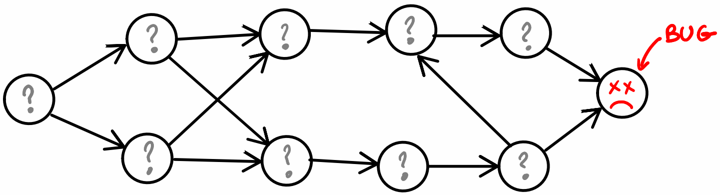
Input: a DAG where each vertex has an unknown status, except one, which is bugged, such that every vertex is an ancestor of this bugged vertex



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

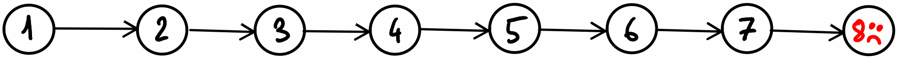
Input: a DAG where each vertex has an unknown status, except one, which is bugged, such that every vertex is an ancestor of this bugged vertex



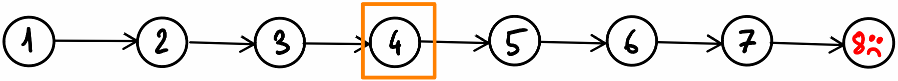
Output: A strategy that finds the original bug with a minimal number of queries in the worst-case scenario = optimal strategy

In real life, queries are costly.

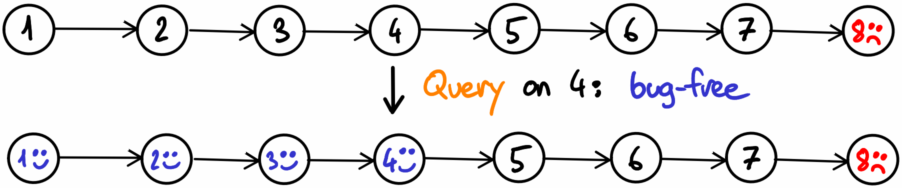
FIRST EXAMPLE: A CHAIN



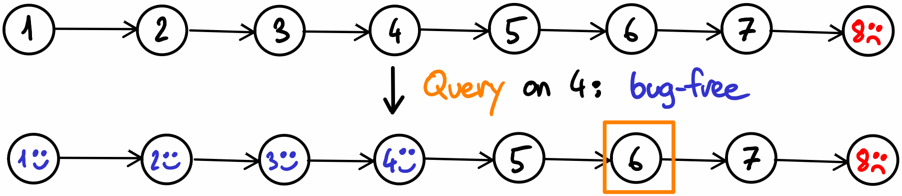
FIRST EXAMPLE: A CHAIN



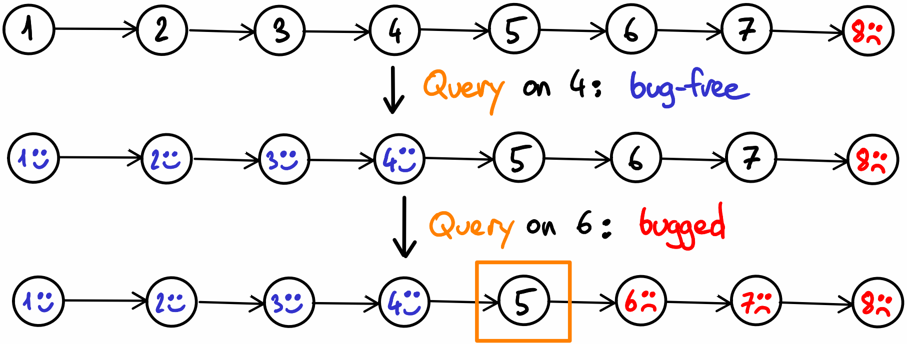
FIRST EXAMPLE: A CHAIN



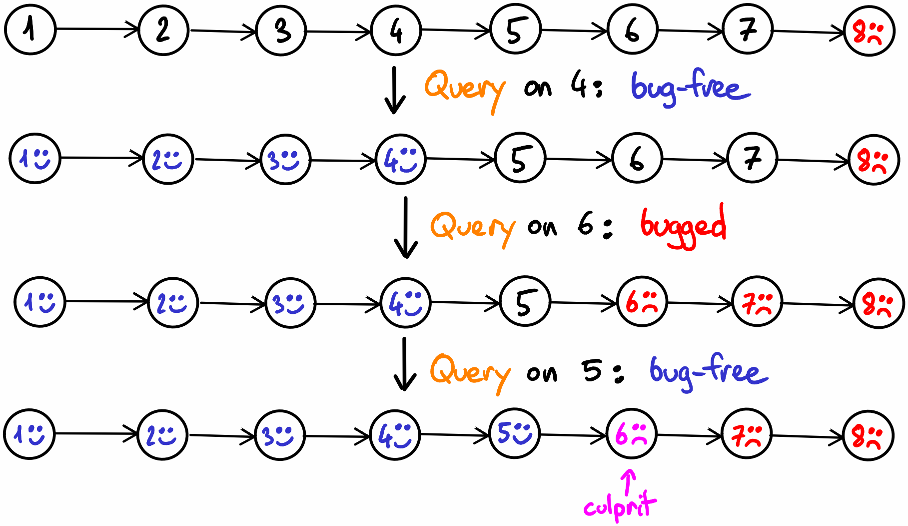
FIRST EXAMPLE: A CHAIN



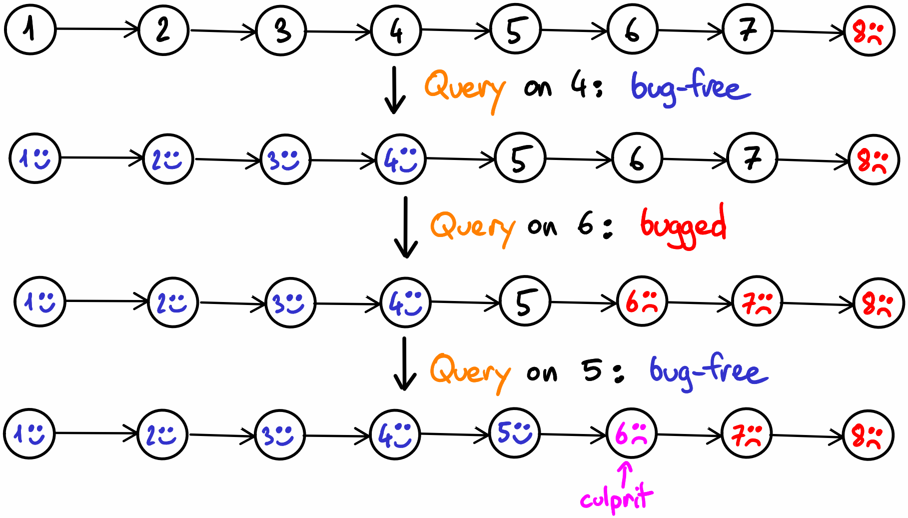
FIRST EXAMPLE: A CHAIN



FIRST EXAMPLE: A CHAIN



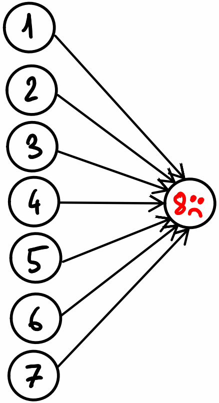
FIRST EXAMPLE: A CHAIN



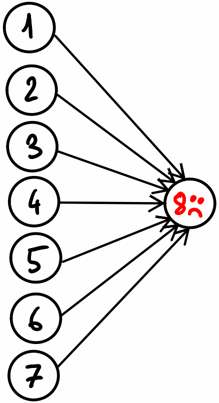
Optimal strategy = binary search

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

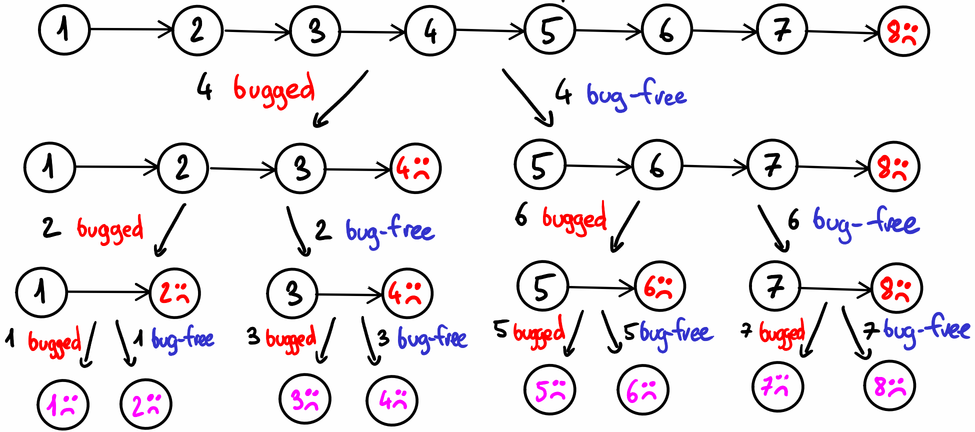


Optimal strategy = whatever

More generally, number of queries in an optimal strategy for a rake of size n = $n-1$

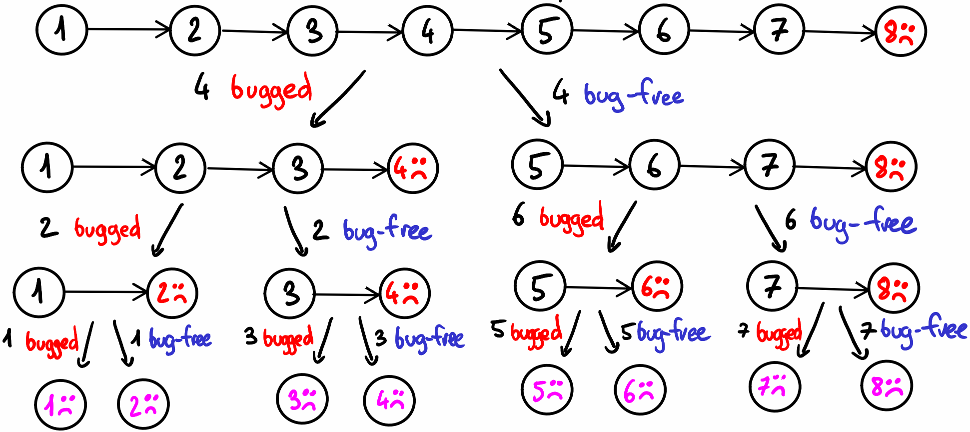
STRATEGY TREE

Strategy tree for binary search:

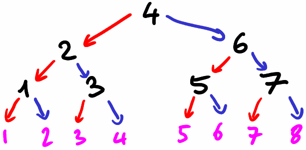


STRATEGY TREE

Strategy tree for binary search:

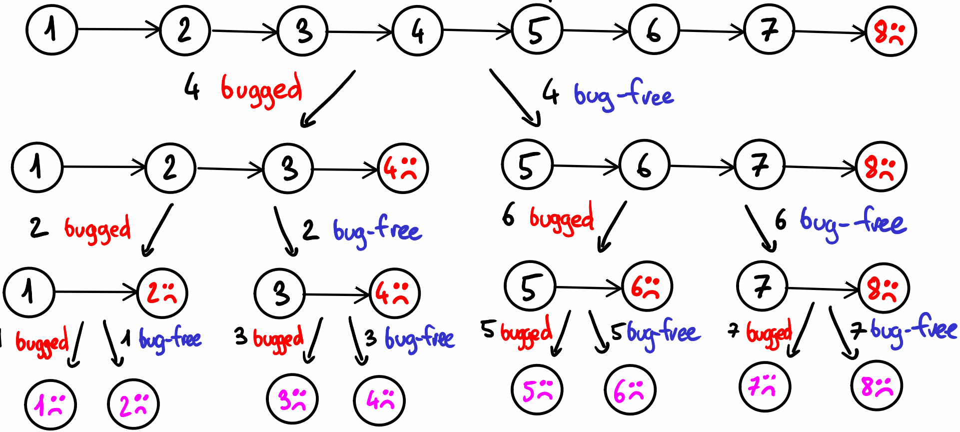


In short:

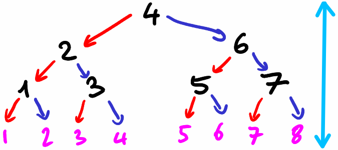


STRATEGY TREE

Strategy tree for binary search:



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 ...

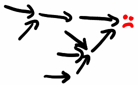
- NP-complete for general DAGs [Carmo Donadelli Kohayakawa Laber 2004]

  Certificate: Strategy tree

  Reduction to: Cover by 3-sets

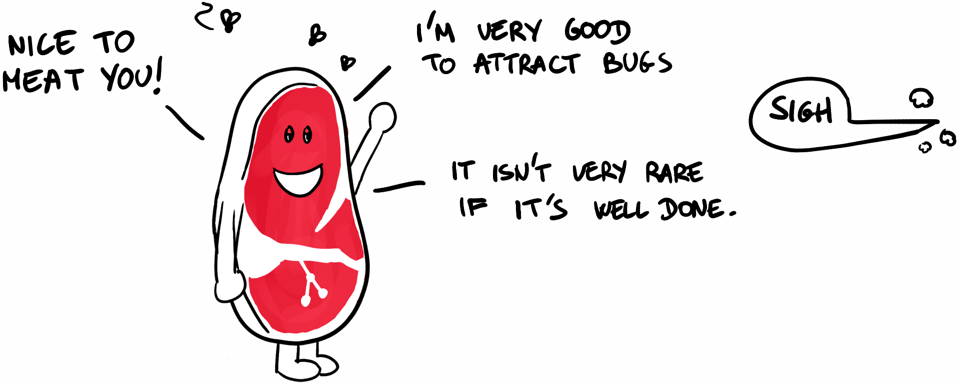
- but polynomial for trees ...

  [Ben-Asher Farchi Newman 2000]



... more precisely, linear. [Mozes Onak Weimann 2008]

PART II - GIT BI-FRICKING-SECT



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

it's him

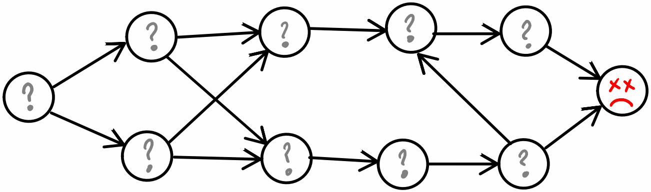
In the source code of git bisect:

```
/*
 * 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

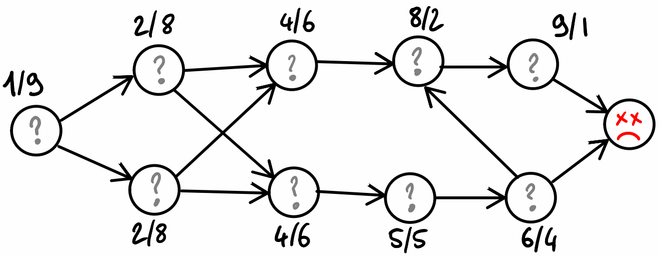
Step 1: Compute the number of ancestors/number of non-ancestors for each vertex:



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:



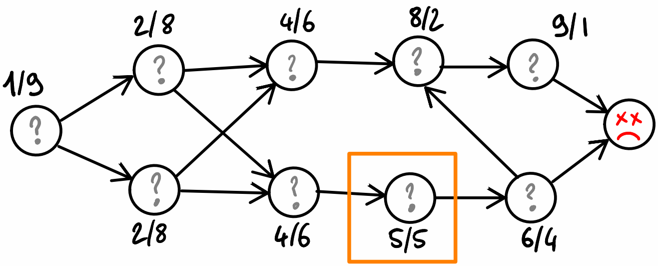
Step 2: Query on a vertex with the most balanced ratio (max of both numbers)

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:



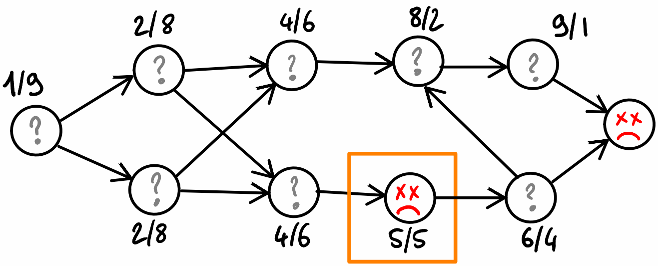
Step 2: Query on a vertex with the most balanced ratio (max of both numbers)

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:



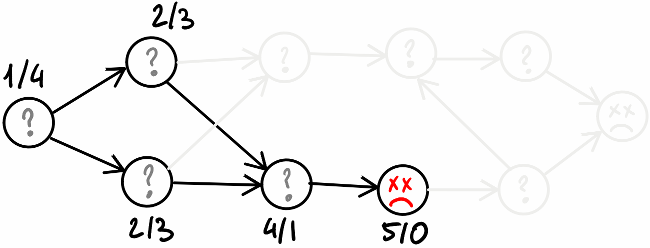
Step 2: Query on a vertex with the most balanced ratio (max of both numbers)

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:



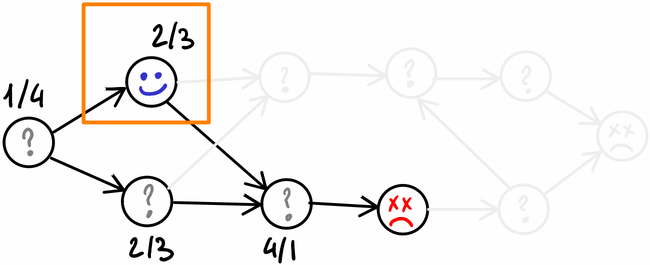
Step 2: Query on a vertex with the most balanced ratio (max of both numbers)

Step 3: Recurse

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:



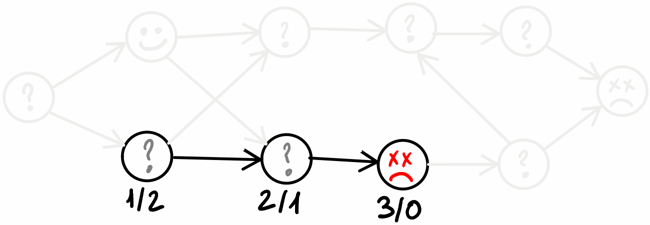
Step 2: Query on a vertex with the most balanced ratio (max of both numbers)

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:



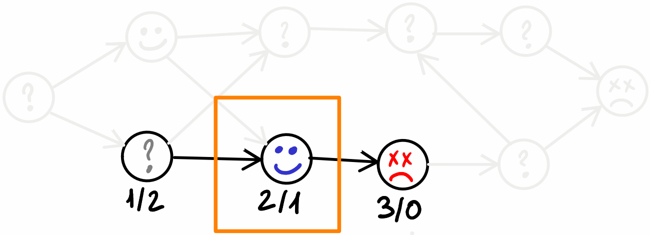
Step 2: Query on a vertex with the most balanced ratio (max of both numbers)

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:



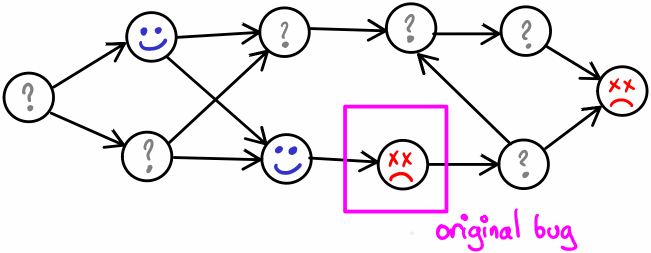
Step 2: Query on a vertex with the most balanced ratio (max of both numbers)

Step 3: Recurse

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:



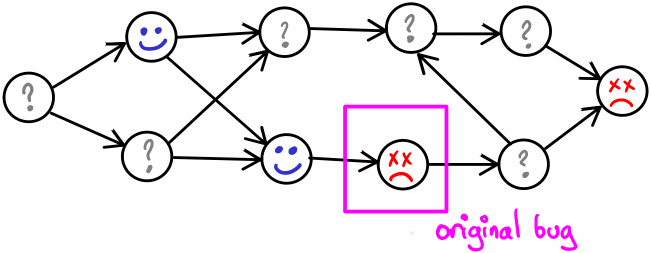
Step 2: Query on a vertex with the most balanced ratio (max of both numbers)

Step 3: Recurse

LET ME INTRODUCE YOU GIT BISECT

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

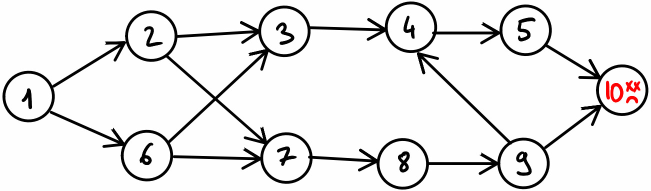
Step 1: Compute the number of ancestors/number of non-ancestors for each vertex:



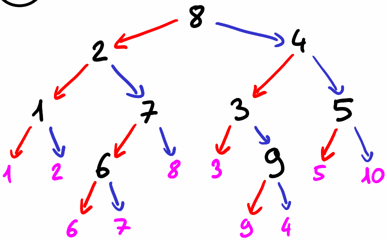
Step 2: Query on a vertex with the most balanced ratio (max of both numbers)

Step 3: Recurse

HOW GOOD IS GIT BISECT?



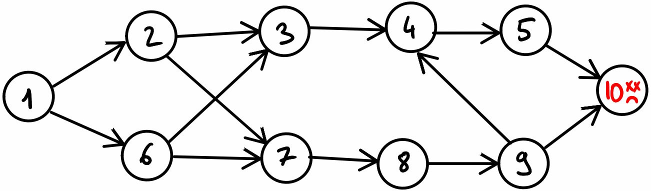
Strategy tree of git bisect for this example:



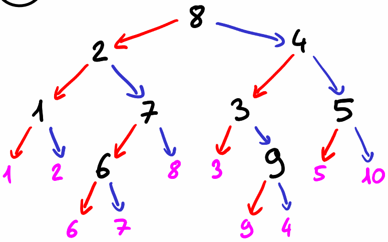
Number of queries in the worst-case scenario of git bisect: 4

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?

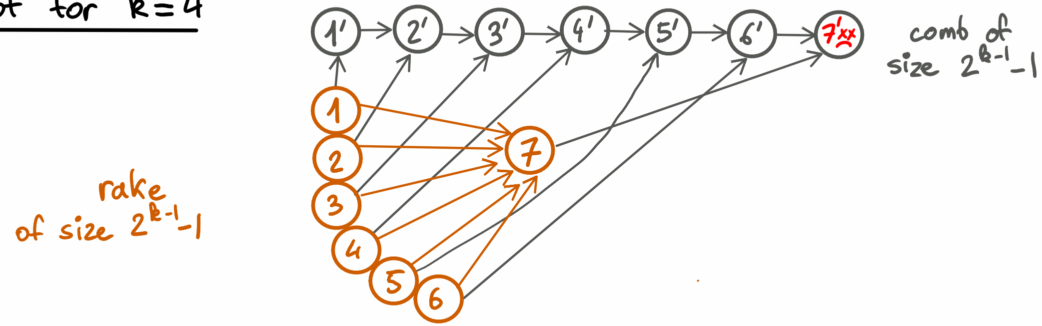
Quick answer: absolutely no, it can't

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.

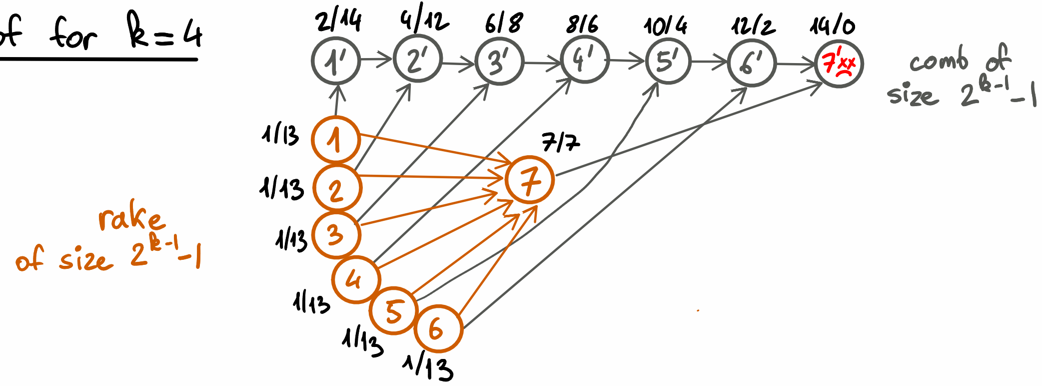
Proof for $k=4$



A COUNTER-EXAMPLE

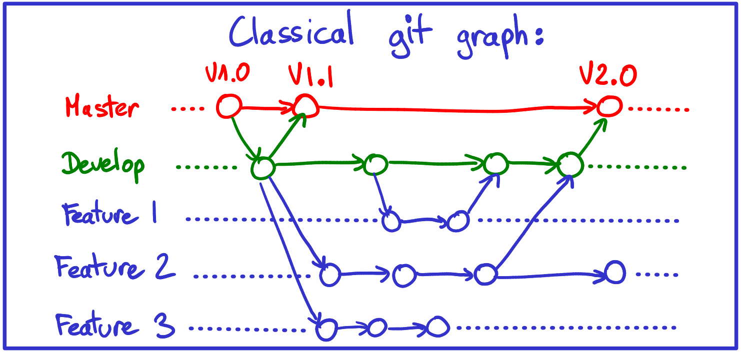
Proposition For any k, there exists a DAG such that an optimal strategy uses k queries and git bisect always uses $2^{k-1}-1$ queries.

Proof for $k=4$

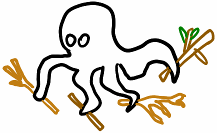


BACK TO REALITY?

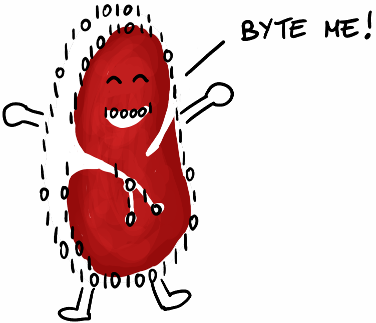
Rake substructures are unrealistic



Usually, we never merge more than 2 branches.

(Otherwise, it is called an octopus merge )

PART III - GIT BISECT ON BINARY DAGs



BINARY DAG

Definition

binary DAG = DAG where the vertices have indegree ≤ 2

Ex:

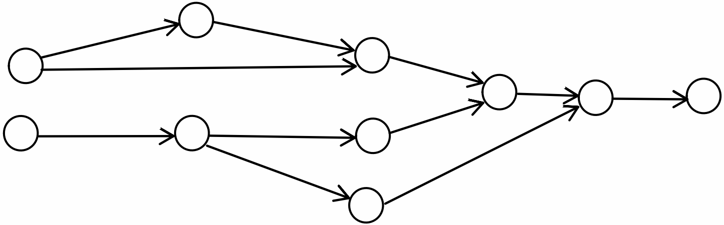
Theorem git bisect is a $\dfrac{1}{\log_2\left(\frac{3}{2}\right)}$ - approximation algorithm when it is used on binary DAGs.

$$\frac{1}{\log_2\left(\frac{3}{2}\right)} \simeq 1{,}71 \text{ is the optimal constant.}$$

EXISTENCE OF A BALANCED VERTEX

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

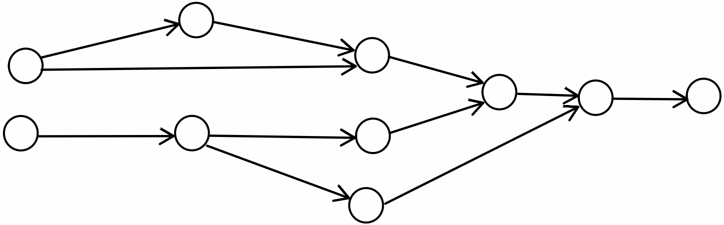
$$\frac{n-1}{3} < x \leq \frac{2n+1}{3}$$



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

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



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

The wanted vertex must be v or one of its parents.

EXISTENCE OF A BALANCED VERTEX

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

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

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

EXISTENCE OF A BALANCED VERTEX

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

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

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

At each step, git 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

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

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.

$$n \text{ vertices} \xrightarrow{1 \text{ step}} 2n/3 \text{ vertices} \xrightarrow{1 \text{ step}} 4n/9 \text{ vertices} \longrightarrow \cdots \xrightarrow{1 \text{ step}} 0$$

number of git bisect queries $\approx \log_{\frac{3}{2}}(n)$

optimal number of queries $\geqslant \log_2(n)$

TIGHTENING THE BOUND

<u>The hard part</u>: proving that $\frac{1}{\log_2\left(\frac{3}{2}\right)}$ 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 $\leq k \log_2\left(\frac{3}{2}\right) + \log_2(3k+6) + 4$

STEP 1 : MAXIMIZING THE NUMBER OF GB QUERIES

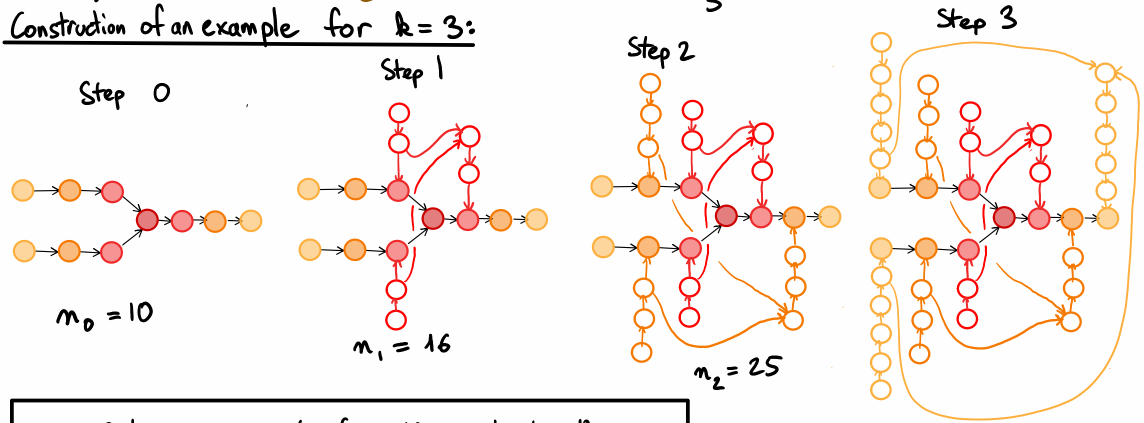
Binary DAG such that git bisect eliminates $\frac{1}{3}$ of its vertices for the k first steps?

<u>Construction of an example for $k = 3$:</u>

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?

<u>Construction of an example for $k = 3$:</u>



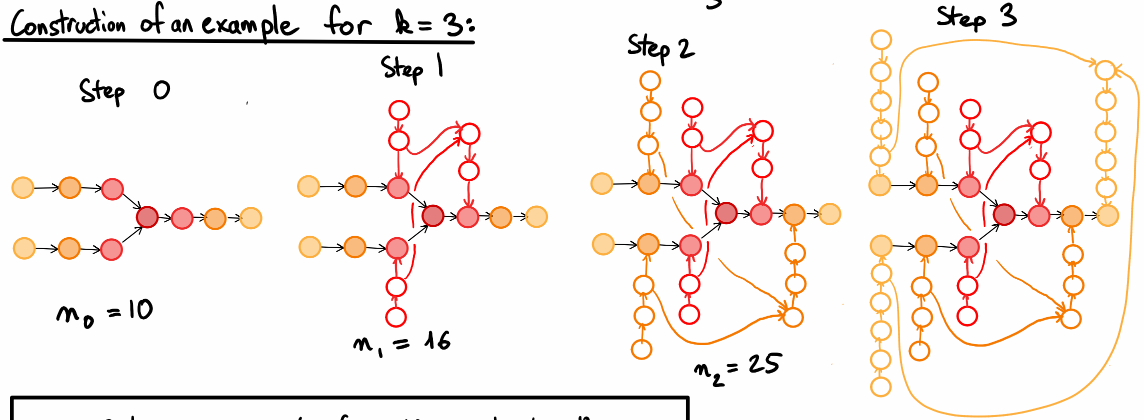
Rule $\quad n_i$ = nb of vertices at step i

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

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 $k = 3$:



Rule $\quad n_i$ = nb of vertices at step i

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

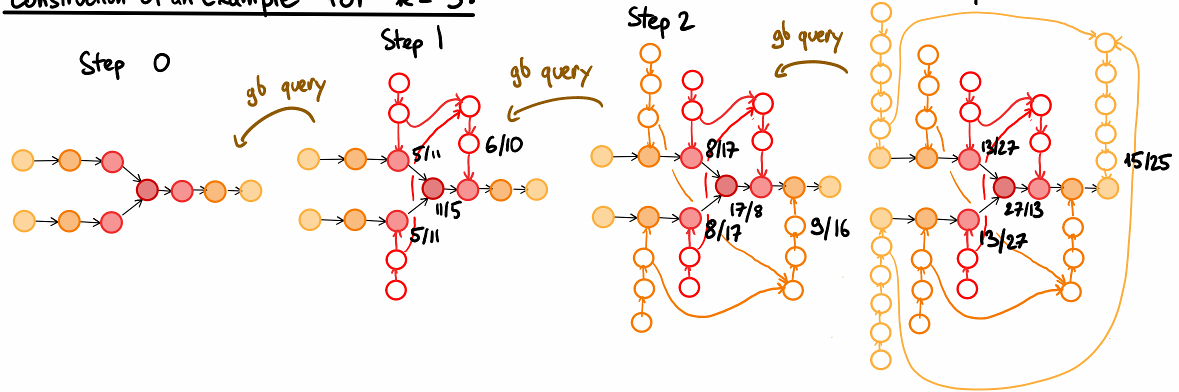
$\simeq 3k \times \left(\frac{3}{2}\right)^k$ vertices

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 k first steps?

Construction of an example for $k = 3$:



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

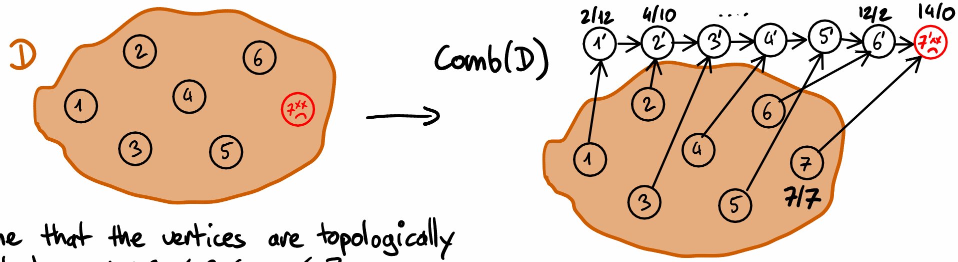
number of git bisect queries $\geq k$

STEP 2: TRAPPING GIT BISECT

Theorem

Let D be a DAG with n vertices. such that git bisect uses gb queries.
There exists a DAG 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 gb+1 queries.

Proof:



We assume that the vertices are topologically sorted: $1 < 2 < 3 < \dots < 7$

A BETTER ALGORITHM? THE GOLDEN BISECTION

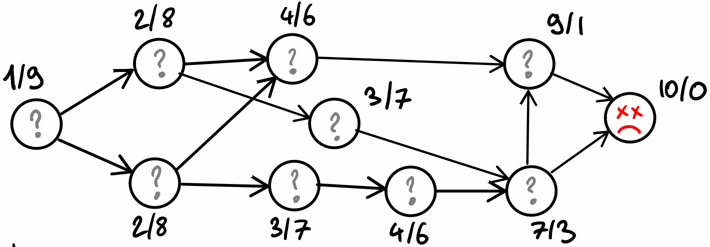
theoritical analysis of git bisect → new (better?) algorithm

A BETTER ALGORITHM? THE GOLDEN BISECTION

theoritical analysis of git bisect → new (better?) algorithm

★ THE GOLDEN BISECTION ★

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



Step 2: M = { 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

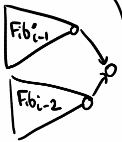
Step 3: Recurse

THEORITICAL ANALYSIS OF THE GOLDEN BISECTION

Theorem The golden bisection is a $\frac{1}{\log_2(\phi)}$ - approximation algorithm for binary DAGs, where ϕ = golden ratio

$\frac{1}{\log_2(\phi)} \simeq 1{,}44$ is the optimal constant.

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

$\mathrm{Fib}_0 = \circ \quad \mathrm{Fib}_1 = \circ \longrightarrow \circ \quad \mathrm{Fib}_i =$ 

PERSPECTIVES

→ Write an article

→ Experimental results : git bisect vs golden bisection

→ Average-case analysis
  - Good model of random git graph (not Erdös-Rényi) ?
  - How to sample them
  - Theoritical analysis of git bisect

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

THANK YOU!