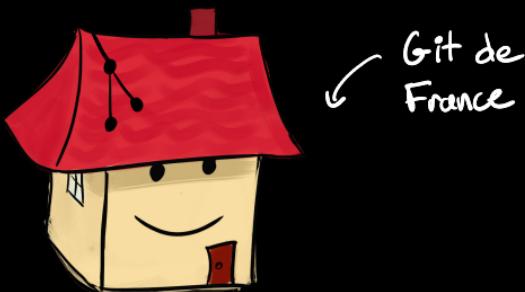


COMBINATORICS OF GIT GRAPHS

Julien COURTIEL
(Université de
Caen Normandie)

WORK 1
with Paul DORBEC
Romain LECOQ
(Université de Caen
Normandie)

WORK 2
with Martin PEPIN
(Université de Caen
Normandie)



Git de
France

LIGM Seminar January 28th 2025

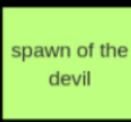
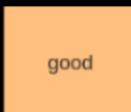
TIER LIST MADE IN MARNE

TIER LIST MADE IN MARNE

QUESTION: What do you use to share files with your coauthors?

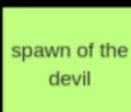
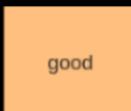
TIER LIST MADE IN MARNE

QUESTION: What do you use to share files with your coauthors?



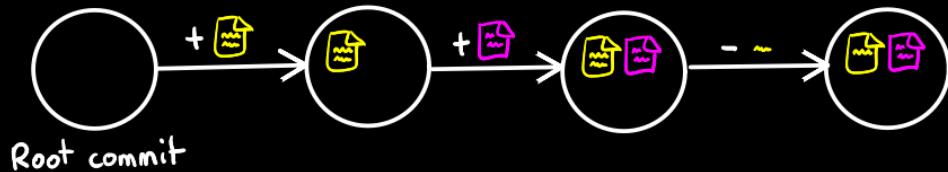
TIER LIST MADE IN MARNE

QUESTION: What do you use to share files with your coauthors?



GIT FEATURES

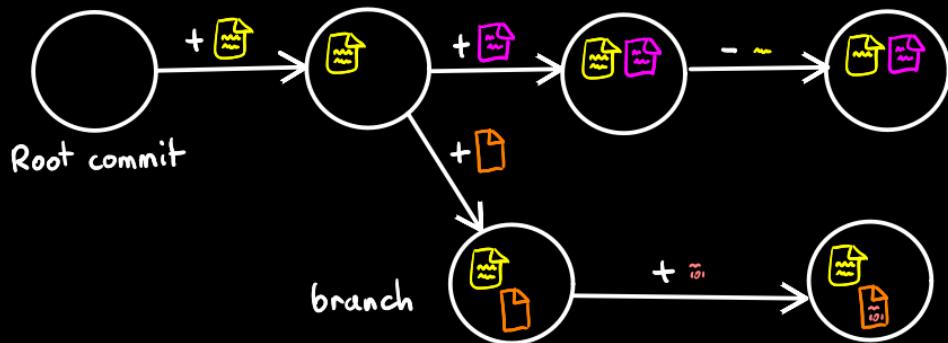
 git is a Version Control System (VCS) :
it stores all the project states over time.



GIT FEATURES



git is a Version Control System (VCS) :
it stores all the project states over time.

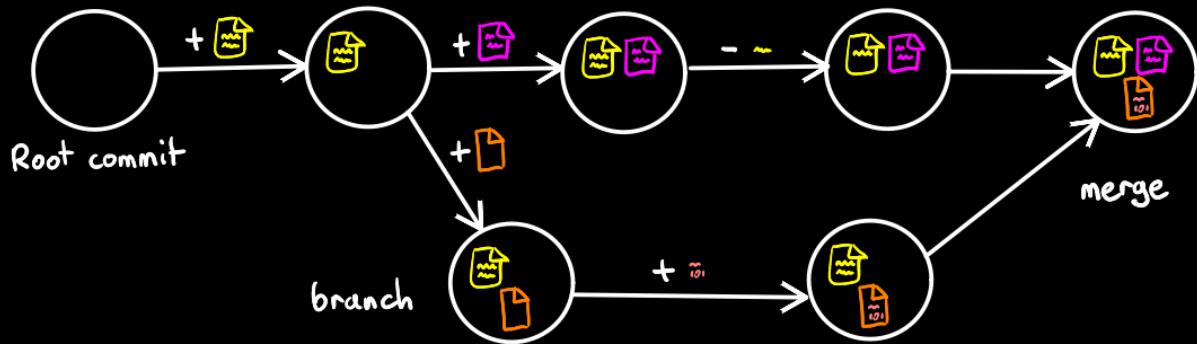


Git lets you create parallel development branches...

GIT FEATURES



git is a Version Control System (VCS) :
it stores all the project states over time.

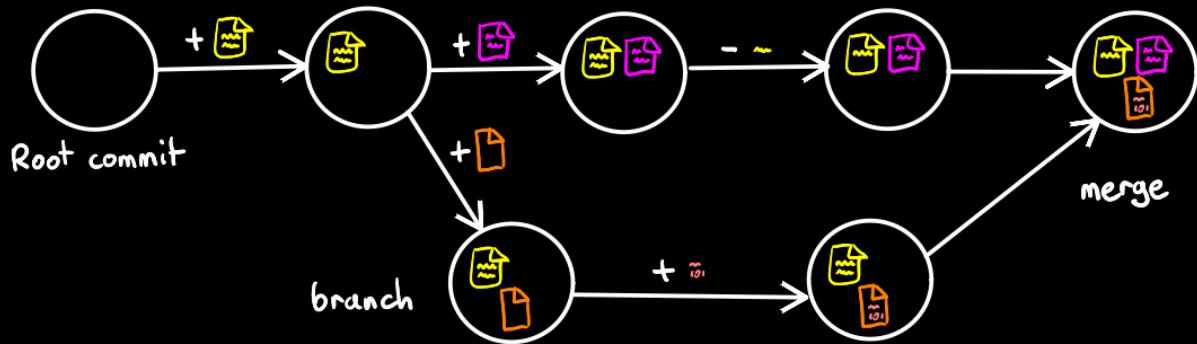


Git lets you create parallel development branches
that can be integrated later.

GIT FEATURES



git is a Version Control System (VCS) :
it stores all the project states over time.



Git lets you create parallel development branches
that can be integrated later.

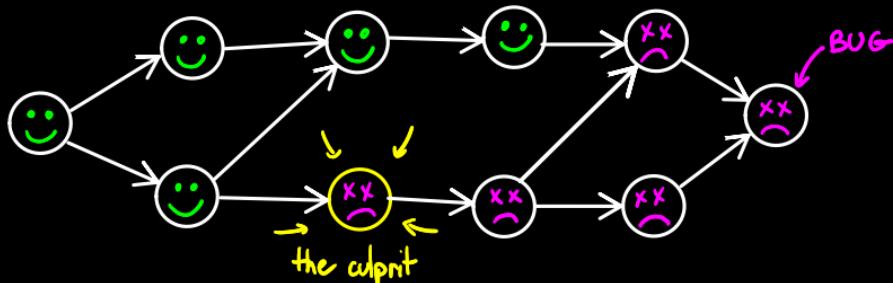
This forms a Directed Acyclic Graph (DAG),
where the vertices are the project states, also named commits.

PART I

Git BISECT



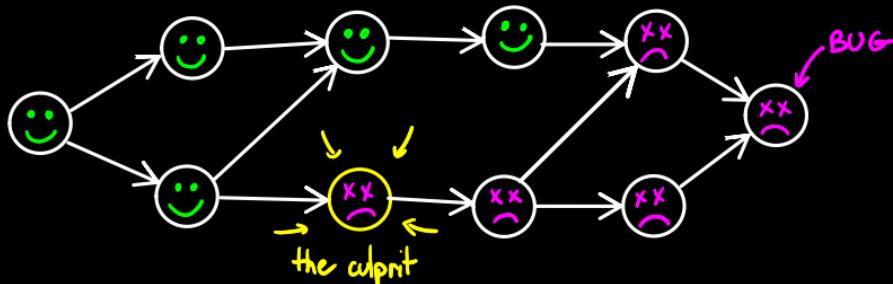
PROBLEM: FINDING A REGRESSION



Input A commit graph in which a commit is known to be bugged, the other commits are bugged or clean (=bug-free)

Question Which commit has originally introduced the bug?

PROBLEM: FINDING A REGRESSION

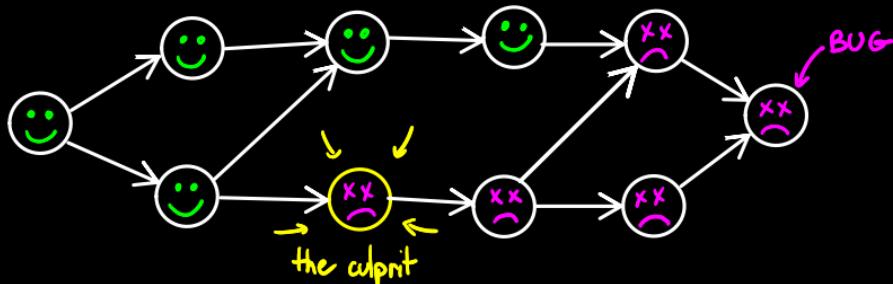


Input A commit graph in which a commit is known to be **bugged**,
the other commits are **bugged** or **clean** (= bug-free)

Question Which commit has originally introduced the bug?

Assumptions If a parent of a commit is bugged, then the commit is bugged.

PROBLEM: FINDING A REGRESSION



Input A commit graph in which a commit is known to be **bugged**,
the other commits are **bugged** or **clean** (= bug-free)

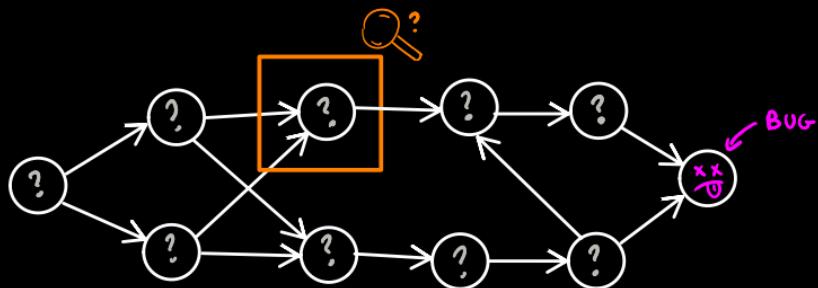
Question Which commit has originally introduced the bug?

Assumptions

- If a parent of a commit is bugged, then the commit is bugged.
- Only one commit has introduced the bug, namely the faulty commit (or regression)

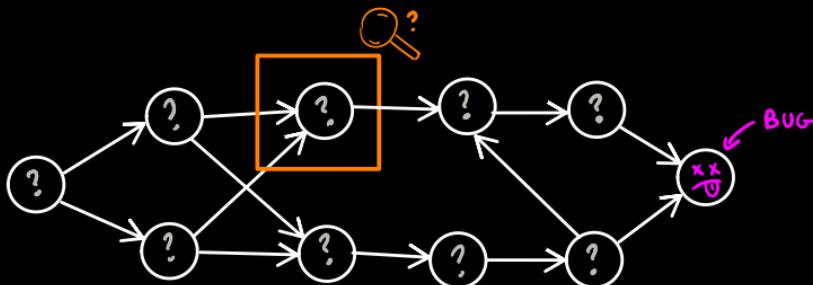
HOW TO INVESTIGATE

Unique operation: QUERY of a commit with unknown status



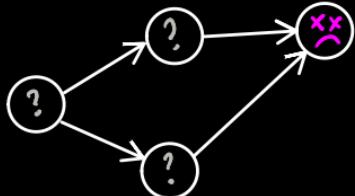
HOW TO INVESTIGATE

Unique operation: QUERY of a commit with unknown status



If bugged,

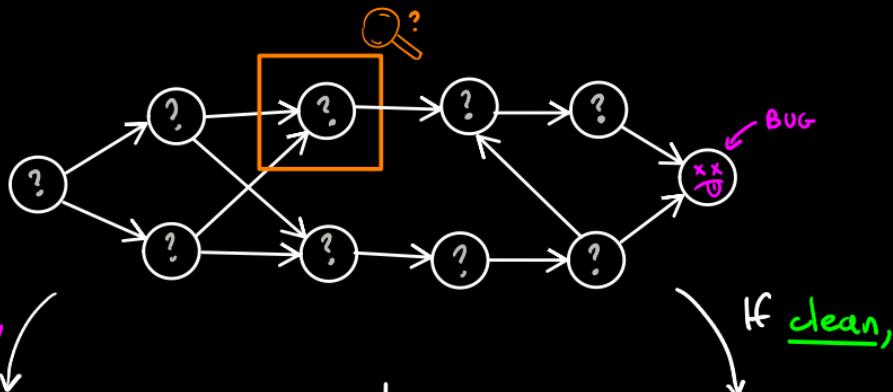
then the faulty commit is an ancestor of this commit



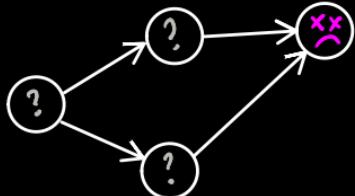
ancestor of a vertex v =
 v or
an ancestor of a parent of v

HOW TO INVESTIGATE

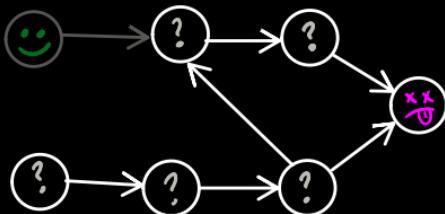
Unique operation: **QUERY** of a commit with unknown status



then the faulty commit is an ancestor of this commit

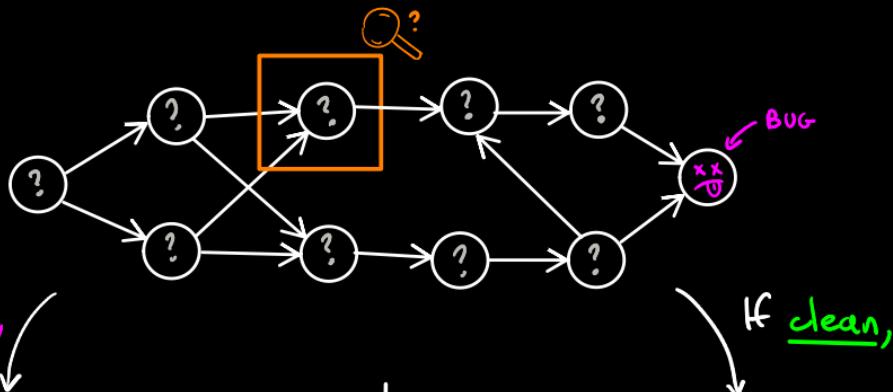


then the faulty commit is not an ancestor of this commit

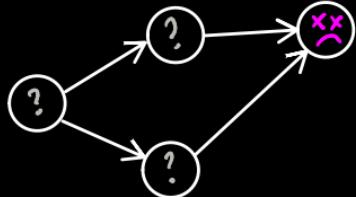


HOW TO INVESTIGATE

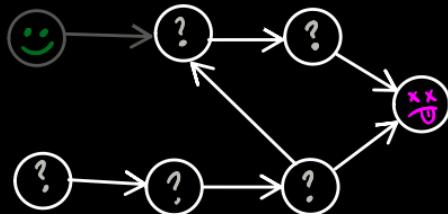
Unique operation: QUERY of a commit with unknown status



If bugged,
then the faulty commit is an ancestor of this commit



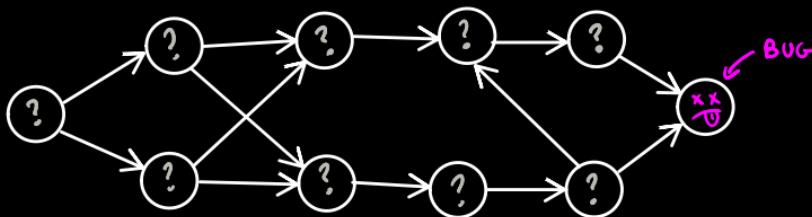
then the faulty commit is not an ancestor of this commit



The faulty commit is found whenever there remains only 1 suspect.

REGRESSION SEARCH PROBLEM

Input: a DAG where each vertex has an unknown status, except one, which is bugged.

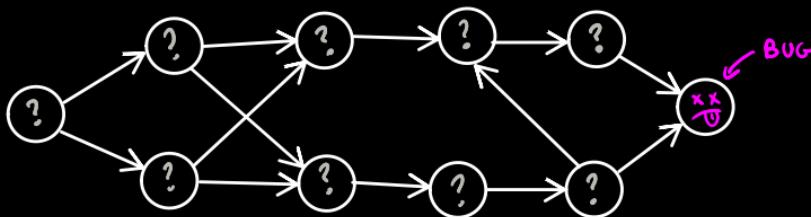


Output: A strategy that finds the faulty commit with a minimal number of queries in the worst-case scenario
= optimal strategy

(the faulty commit can be any ancestor of the bugged vertex)

REGRESSION SEARCH PROBLEM

Input: a DAG where each vertex has an unknown status, except one, which is bugged.



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

(the faulty commit can be any ancestor of the bugged vertex)

In real life, queries are costly.

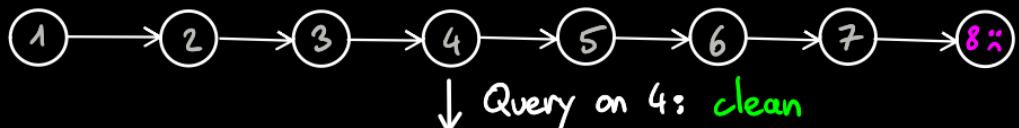
FIRST EXAMPLE : CHAINS



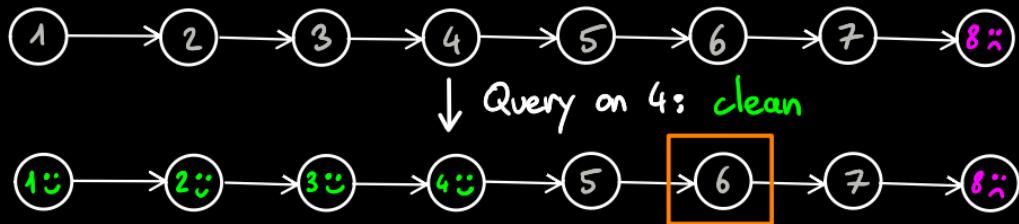
FIRST EXAMPLE : CHAINS



FIRST EXAMPLE : CHAINS



FIRST EXAMPLE : CHAINS



FIRST EXAMPLE : CHAINS



↓ Query on 4: clean



↓ Query on 6: bugged



FIRST EXAMPLE : CHAINS



↓ Query on 4: clean



↓ Query on 6: bugged



FIRST EXAMPLE : CHAINS



↓ Query on 4: clean



↓ Query on 6: bugged



↓ Query on 5: clean



faulty commit

FIRST EXAMPLE : CHAINS



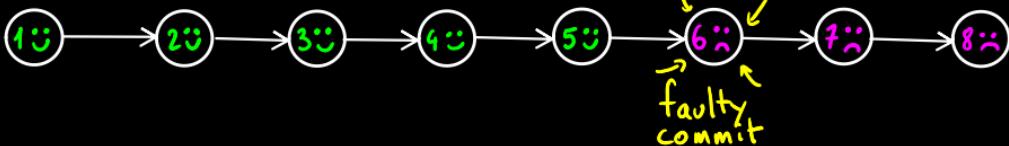
↓ Query on 4: clean



↓ Query on 6: bugged



↓ Query on 5: clean



optimal strategy = binary search

FIRST EXAMPLE : CHAINS



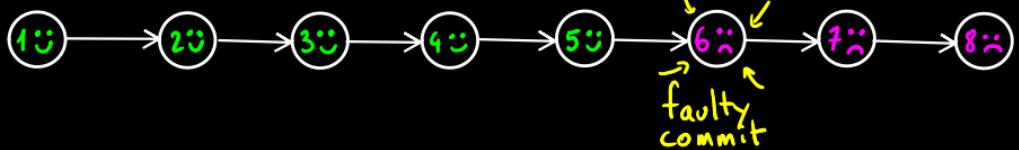
↓ Query on 4: clean



↓ Query on 6: bugged



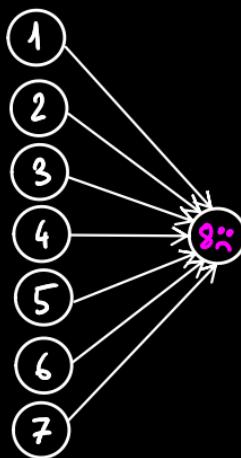
↓ Query on 5: clean



optimal strategy = binary search

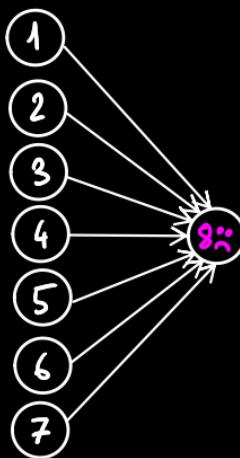
More generally, number of queries in an optimal strategy
in a chain of length n = $\lceil \log_2(n) \rceil$

SECOND EXAMPLE : OCTOPUSES



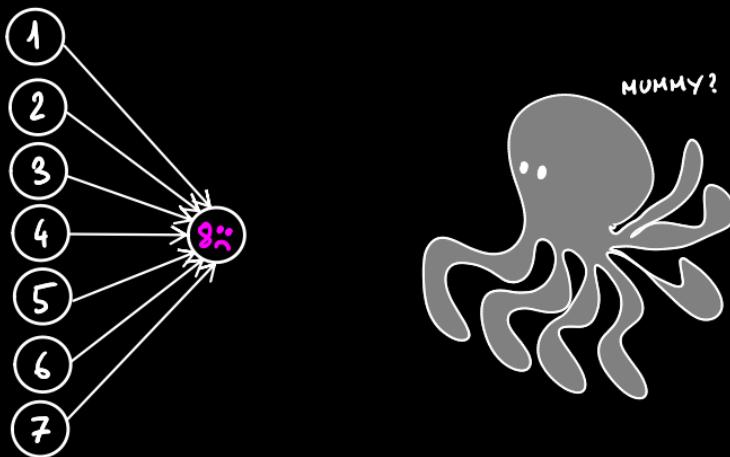
optimal strategy =

SECOND EXAMPLE : OCTOPUSES



optimal strategy = whatever

SECOND EXAMPLE : OCTOPUSES



optimal strategy = whatever

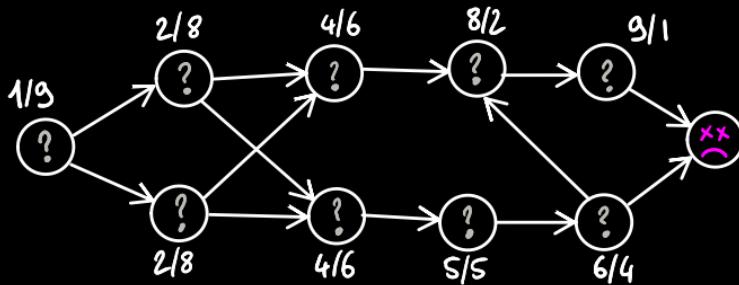
More generally, number of queries in an optimal strategy
in an octopus of size n = $n-1$

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1:



STEP 2 :

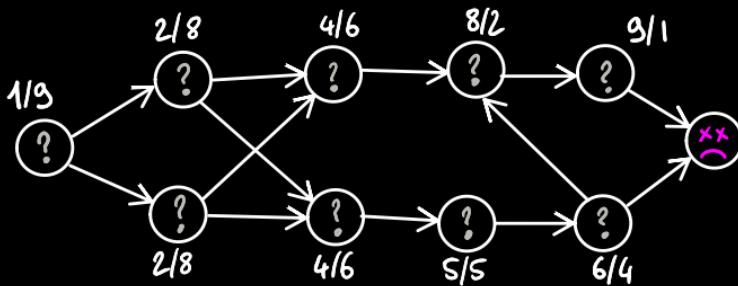
STEP 3 :

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors / non-ancestors for each commit



STEP 2 :

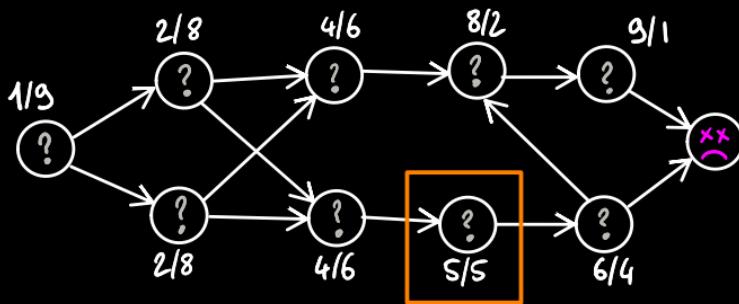
STEP 3 :

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors / non-ancestors for each commit



STEP 2 : Query the vertex with the most balanced ratio

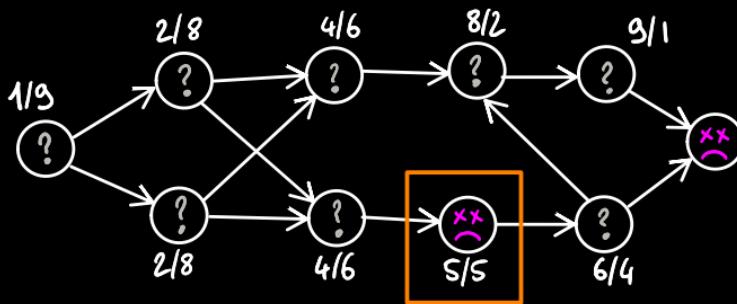
STEP 3 :

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors / non-ancestors for each commit



STEP 2 : Query the vertex with the most balanced ratio

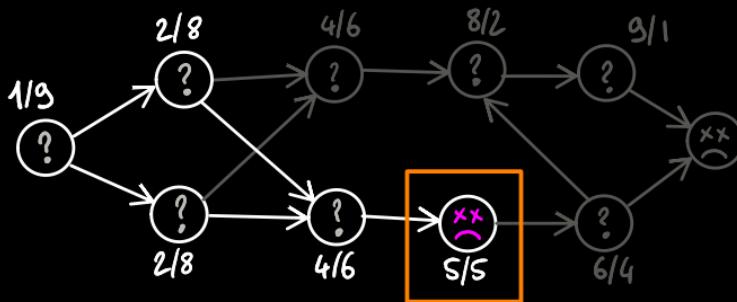
STEP 3 :

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors / non-ancestors for each commit



STEP 2 : Query the vertex with the most balanced ratio

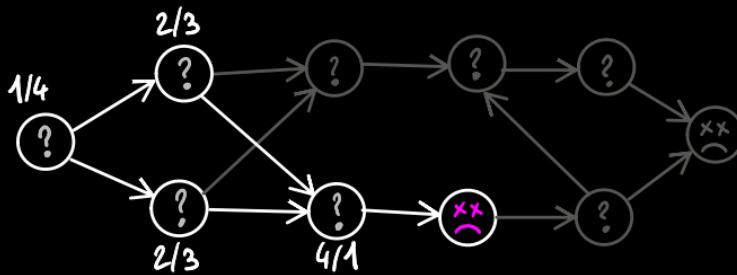
STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors / non-ancestors for each commit



STEP 2 : Query the vertex with the most balanced ratio

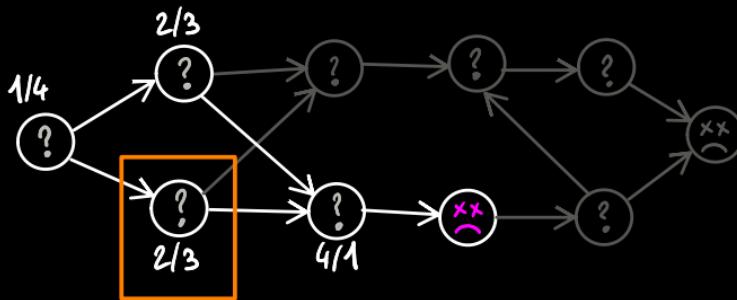
STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors / non-ancestors for each commit



STEP 2 : Query the vertex with the most balanced ratio
a

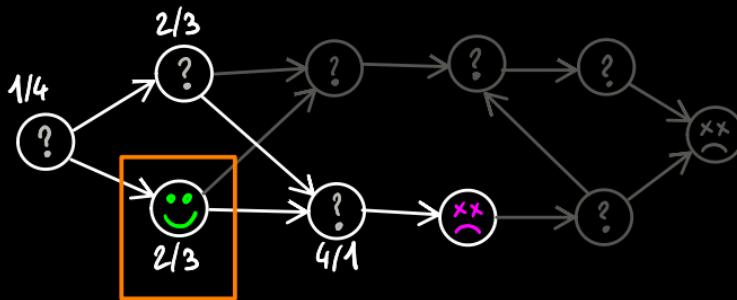
STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors / non-ancestors for each commit



STEP 2 : Query the vertex with the most balanced ratio
a

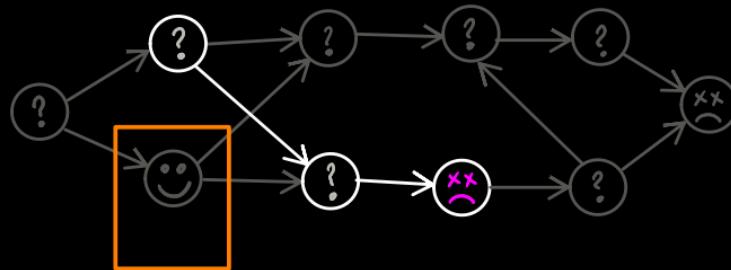
STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors/non-ancestors for each commit



STEP 2 : Query the vertex with the most balanced ratio
a

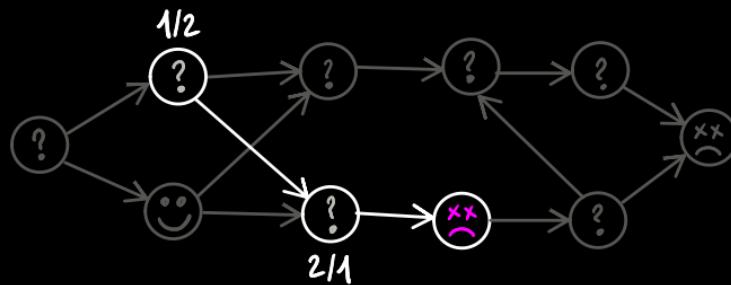
STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors/non-ancestors for each commit



STEP 2: Query the vertex with the most balanced ratio
a

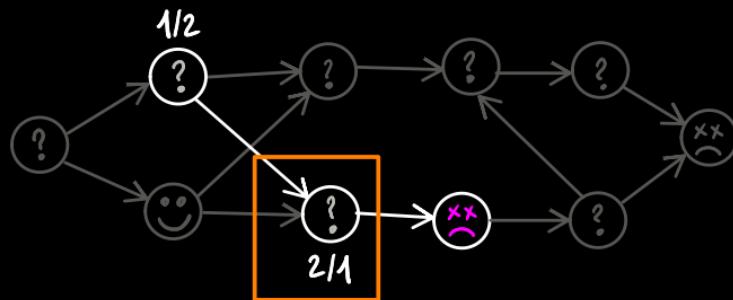
STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors/non-ancestors for each commit



STEP 2 : Query the vertex with the most balanced ratio
a

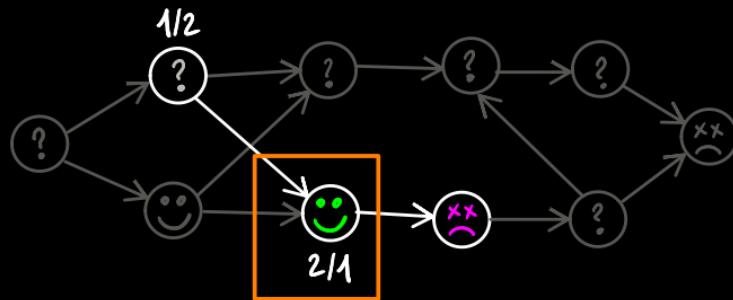
STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors/non-ancestors for each commit



STEP 2 : Query the vertex with the most balanced ratio
a

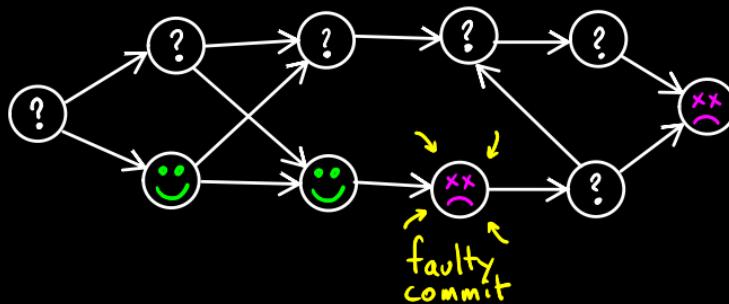
STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors/non-ancestors for each commit



STEP 2: Query the vertex with the most balanced ratio
a

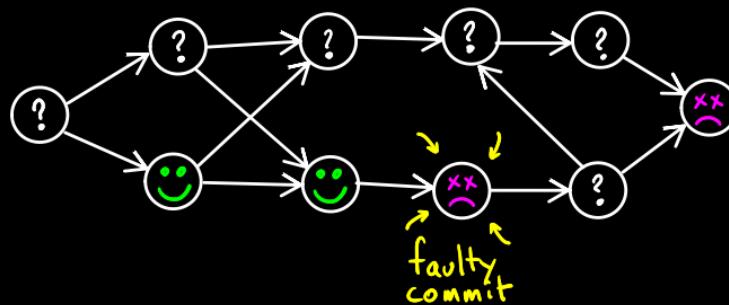
STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM



uses a heuristic to find the faulty commit: git bisect

STEP 1: Compute the number of ancestors/non-ancestors for each commit



STEP 2: Query the vertex with the most balanced ratio

a

STEP 3: Delete the innocent commits and recurse.

QUESTION: Does git bisect always give an optimal strategy?

HOW GOOD IS GIT BISECT?

QUESTION: Does git bisect always give an optimal strategy?

HOW GOOD IS GIT BISECT?

QUESTION: Does git bisect always give an optimal strategy?

NO The Regression Search Problem is NP-complete.

[Carmo Donadelli
Kohayakawa (Laber 2006)]

[We've also proved it!]

But is it really that bad?

HOW GOOD IS GIT BISECT?

QUESTION: Does git bisect always give an optimal strategy?

NO The Regression Search Problem is NP-complete.

[Carmo Donadelli
Kohayakawa (Laber 2006)]

[We've also proved it!]

But is it really that bad?

YEAH!

HOW GOOD IS GIT BISECT?

QUESTION: Does git bisect always give an optimal strategy?

NO The Regression Search Problem is NP-complete.

[Carmo Donadelli
Kohayakawa (Laber 2006)]

[We've also proved it!]

But is it really that bad?

YEAH!

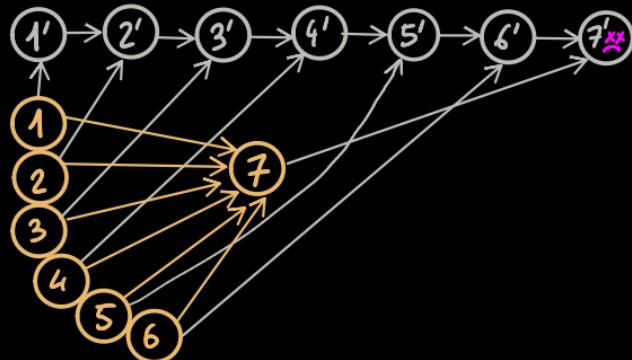
Proposition For any k , there exists a DAG such that
an optimal strategy uses k queries
and git bisect always uses $2^{k-1} - 1$ queries.

HOW GOOD IS GIT BISECT?

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 $k=4$:

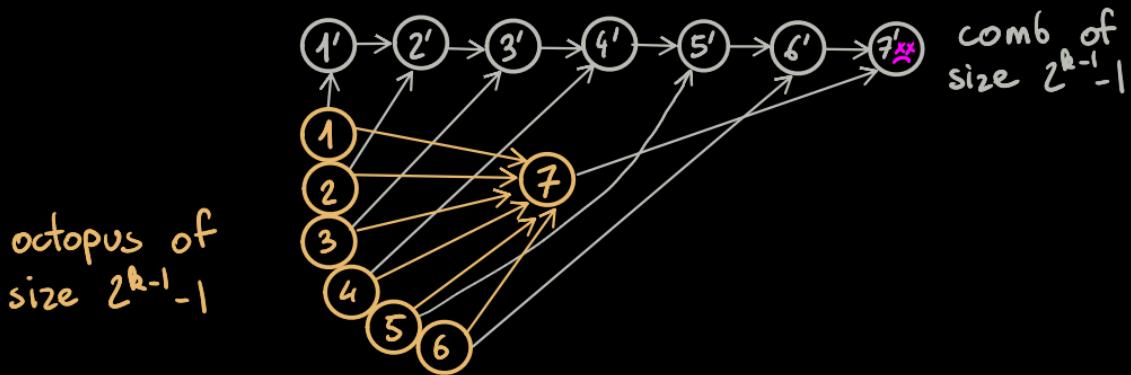


HOW GOOD IS GIT BISECT?

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 $k=4$:

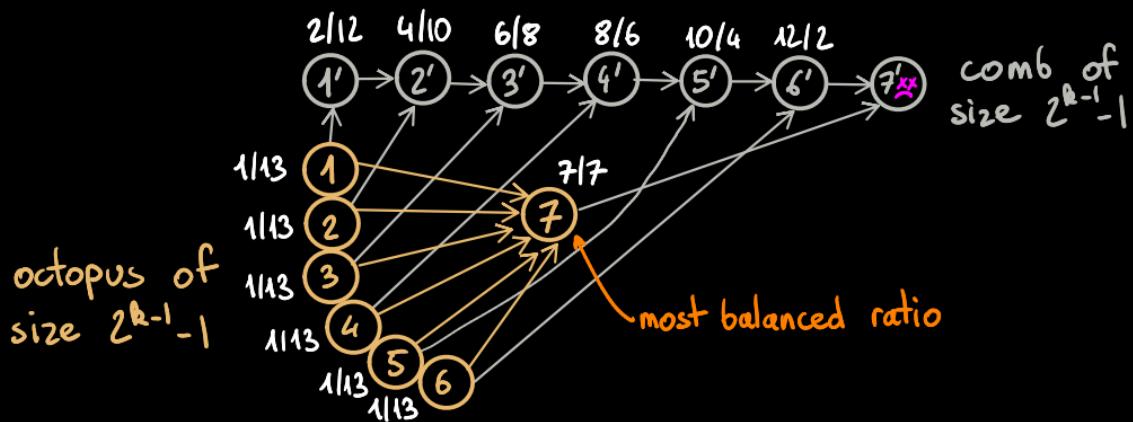


HOW GOOD IS GIT BISECT?

Proposition

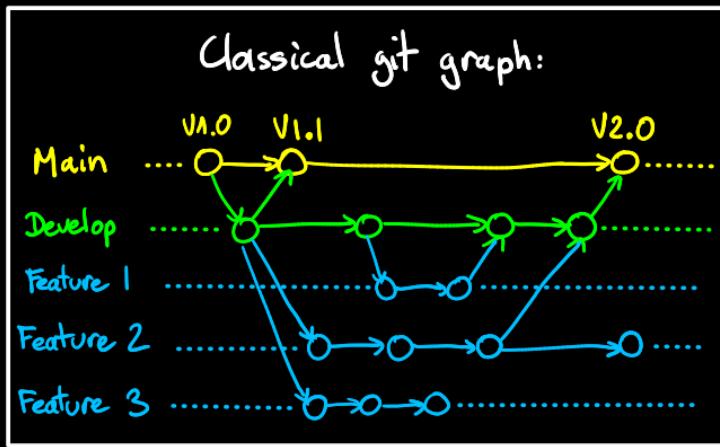
For any k , there exists a DAG such that
an optimal strategy uses k queries
and git bisect always uses $2^{k-1}-1$ queries.

Proof for $k=4$:



BACK TO REALITY ?

Octopus substructures are unrealistic



Usually, we never merge more than 2 branches.

(Otherwise it is called an octopus merge)

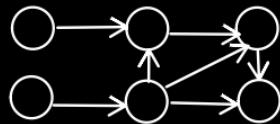


BINARY DAGS

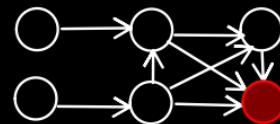
Definition

binary DAG = DAG where the vertices have indegree ≤ 2

Ex:



Good



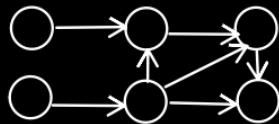
Bad

BINARY DAGS

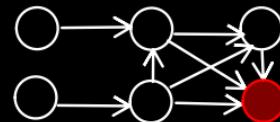
Definition

binary DAG = DAG where the vertices have indegree ≤ 2

Ex:



Good



Bad

Theorem

git bisect is a $\frac{1}{\log_2(\frac{3}{2})}$ - approximation algorithm

when it is used on binary DAGs.

$\frac{1}{\log_2(\frac{3}{2})} \approx 1.71$ is the optimal constant.

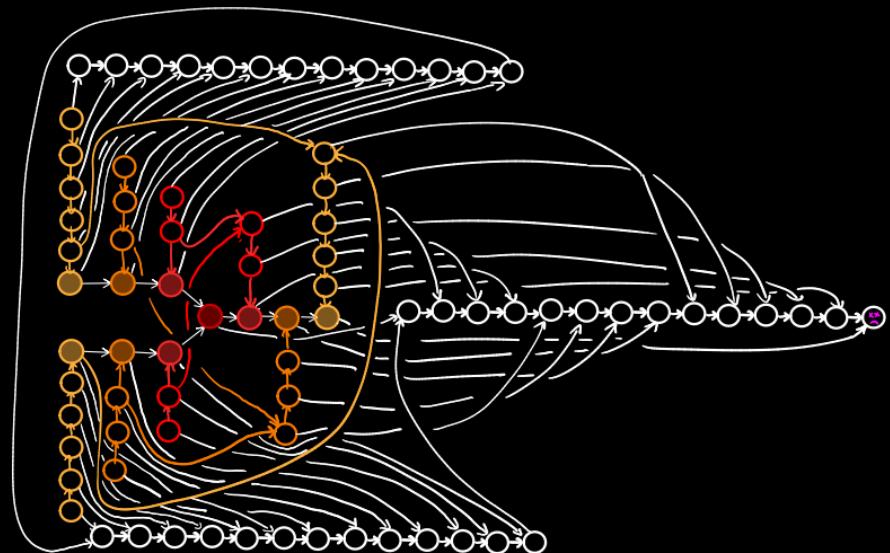
BINARY DAGS

Theorem

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

$$\frac{1}{\log_2(\frac{3}{2})} \approx 1.71 \text{ is the optimal constant.}$$

Example of
a mean 
binary DAG



GOLDEN BISECT

We've improved git bisect in the worst-case scenario

→ new algorithm : ★ golden^{*} bisect ★

GOLDEN BISECT

We've improved git bisect in the worst-case scenario

→ new algorithm : ★ golden_{*} bisect ★

Idea
behind
golden
bisect

- If git bisect queries a vertex with score $\geq \frac{\text{nb vertices}}{\phi^2}$, query the same
- Otherwise, query some special vertex even if it has not best score.

GOLDEN BISECT

We've improved git bisect in the worst-case scenario

→ new algorithm : ★ golden_{*} bisect ★

Idea
behind
golden
bisect

- If git bisect queries a vertex with score $\geq \frac{\text{nb vertices}}{\phi^2}$, query the same
- Otherwise, query some special vertex even if it has not best score.

Theorem

golden bisect is a $\frac{1}{\log_2(\phi)}$ - approximation algorithm

for binary DAGs, where $\phi = \text{golden ratio.}$
 $= (1 + \sqrt{5}) / 2$

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

QUESTION

Have we really proved that git bisect is bad?

QUESTION

Have we really proved that git bisect is bad?

- The examples of DAGs where git bisect performs poorly shouldn't occur in real life
- Sometimes git bisect is better than golden bisect

QUESTION

Have we really proved that git bisect is bad?

- The examples of DAGs where git bisect performs poorly shouldn't occur in real life
- Sometimes git bisect is better than golden bisect

More relevant (?)
question

What is the average-case
complexity of git bisect?

QUESTION

Have we really proved that `git bisect` is bad?

- The examples of DAGs where `git bisect` performs poorly shouldn't occur in real life
- Sometimes `git bisect` is better than golden bisect

More relevant (?)
question

What is the average-case
complexity of `git bisect`?

This question calls for many more,
notably what is a random Git graph?

PART II

RANDOM GIT GRAPHS

ongoing work

IT'S MY NEIGHBORHOOD!

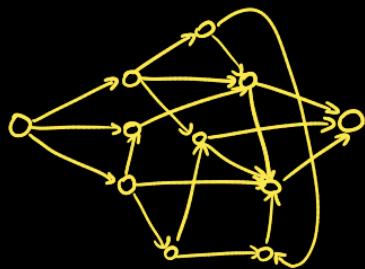


WHICH GRAPHS TO CONSIDER ?

In  git, every DAG

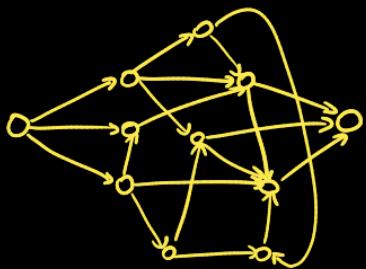
without restriction

can be generated...



WHICH GRAPHS TO CONSIDER ?

In  git, every DAG without restriction can be generated...

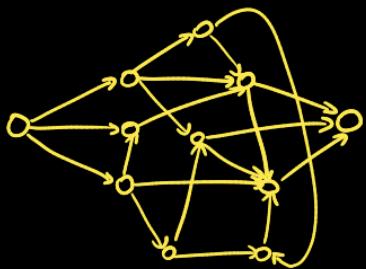


... but many projects follow a workflow

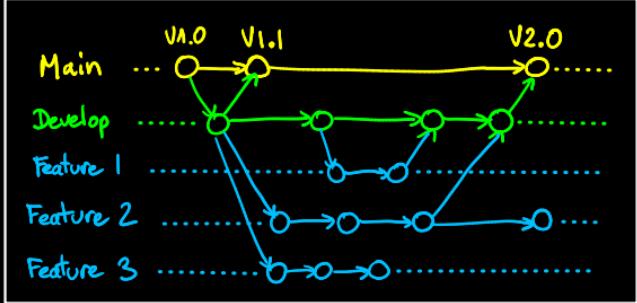


WHICH GRAPHS TO CONSIDER ?

In  , every DAG without restriction can be generated ...



... but many projects follow a workflow



In the following, we consider a simple workflow but widely used in industry: the feature branch workflow

GIT GRAPH

DEFINITION

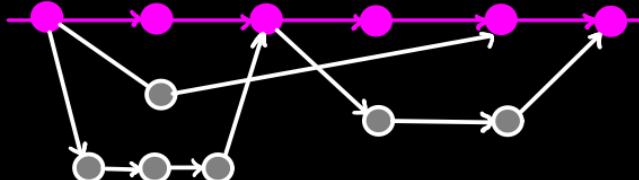
(feature branch)
Git graph

= DAG with

- a main branch (path of magenta vertices)
- 0, 1 or several ⁺feature branches, paths of ≥ 1 white vertices starting and ending on magenta vertices
- indegree ≤ 2 for all vertices

previously defined in [Lecoq 2024]

e.g.



ILLUSTRATED RULES	
OK	✓
KO	✗

The table illustrates rules for a Git graph. The 'OK' row shows a magenta vertex with two outgoing arrows to white vertices, and a white vertex with two incoming arrows from magenta vertices. The 'KO' row shows a magenta vertex with three outgoing arrows to white vertices, and a white vertex with three incoming arrows from magenta vertices.

OBJECTIVES

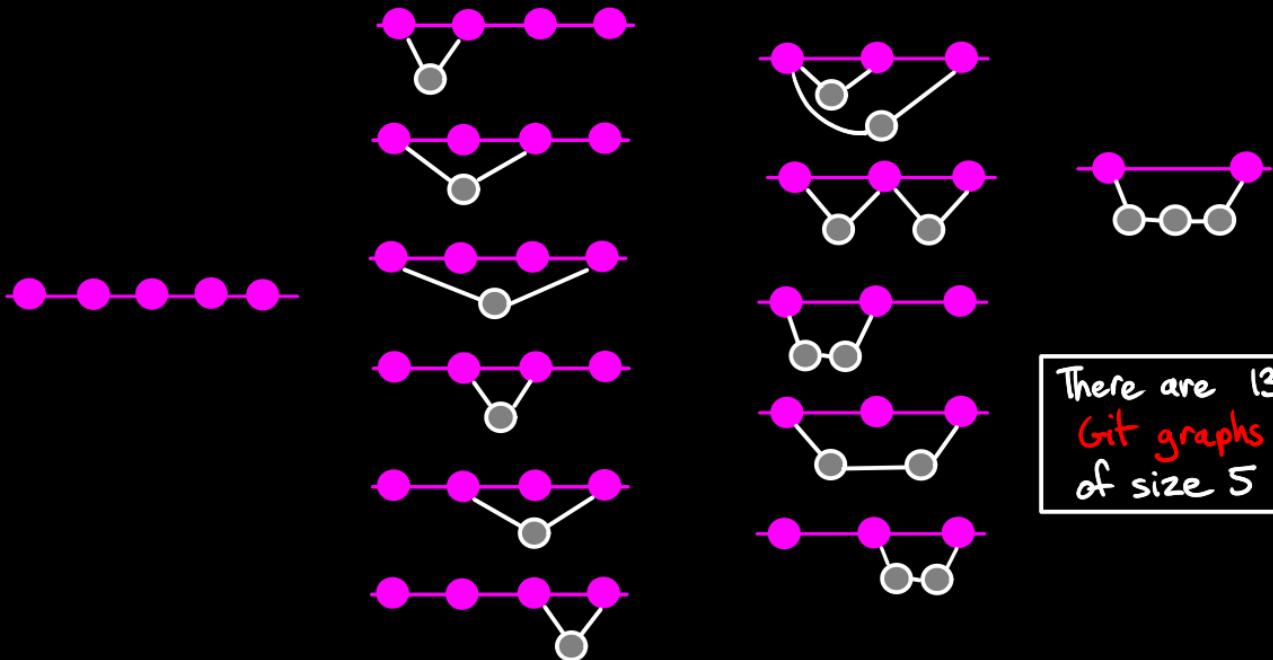
G
O
A
L
S

- Count Git graphs (exact & asymptotic)
- Sample a Git graph uniformly at random given a size n and a number k of magenta vertices

G
O
A
L
S

OBJECTIVES

- Count Git graphs (exact & asymptotic)
- Sample a Git graph uniformly at random given a size n and a number k of magenta vertices



There are 13
Git graphs
of size 5

RECURSIVE DECOMPOSITION

Decomposition

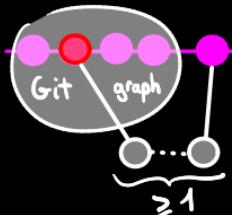


=

or



or



Recurrence

$$g_{n,k} = g_{n-1, k-1} + \sum_{l \geq 1} (k-1) g_{m-1-l, k-1} \quad \text{for } n \geq 1$$

where $g_{n,k}$:= number of Git graphs with n vertices,
 k of them being magenta

Differential Equation for the Generating Function

$$G(\gamma, u) = 1 + \gamma u G(\gamma, u) + \frac{\gamma^2 u^2}{1-\gamma} \frac{\partial G}{\partial u} (\gamma, u)$$

$$\text{where } G(\gamma, u) = \sum_{n \geq 0} \sum_{k \geq 0} g_{n,k} \gamma^n u^k$$



$G(\gamma, u)$ is not analytic.



SANDWICHING $g_{n,k}$

$g_{n,k}$ = number of Git graphs with n vertices, k of them being magenta

$$\left(\frac{n-k-1}{k-2}\right) (k-1)! \leq g_{n,k} \leq \left(\frac{n-2}{k-2}\right) (k-1)! \quad \text{if } k \leq \frac{n+1}{2}$$

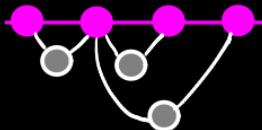


SANDWICHING $g_{m,k}$

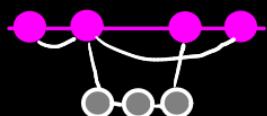
$g_{m,k}$ = number of Git graphs with m vertices, k of them being magenta

$$\binom{m-k-1}{k-2} (k-1)! \leq g_{m,k} \leq \binom{m-2}{k-2} (k-1)! \quad \text{if } k \leq \frac{m+1}{2}$$

nb of Git graphs
where every magenta
vertex has indegree 2



nb of Git graphs
where every magenta
vertex has indegree 2
(except the 1st one)
but branches with 0
white vertex are allowed



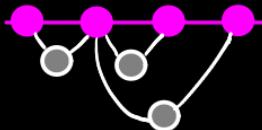


SANDWICHING $g_{n,k}$

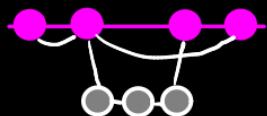
$g_{n,k}$ = number of Git graphs with n vertices, k of them being magenta

$$\binom{n-k-1}{k-2} (k-1)! \leq g_{n,k} \leq \binom{n-2}{k-2} (k-1)! \quad \text{if } k \leq \frac{n+1}{2}$$

nb of Git graphs where every magenta vertex has indegree 2



nb of Git graphs where every magenta vertex has indegree 2
(except the 1st one)
but branches with 0 white vertex are allowed



$$\frac{(k-1)!}{(2^{k-n}-1)!} \leq g_{n,k} \leq \binom{n-2}{k-2} \frac{(k-1)!}{(2^{k-n}-1)!} \quad \text{if } k > \frac{n+1}{2}$$

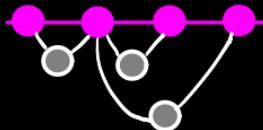


SANDWICHING $g_{n,k}$

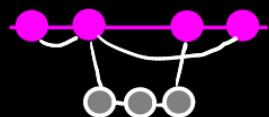
$g_{n,k}$ = number of Git graphs with n vertices, k of them being magenta

$$\binom{n-k-1}{k-2} (k-1)! \leq g_{n,k} \leq \binom{n-2}{k-2} (k-1)! \quad \text{if } k \leq \frac{n+1}{2}$$

nb of Git graphs where every magenta vertex has indegree 2



nb of Git graphs where every magenta vertex has indegree 2 (except the 1st one) but branches with 0 white vertex are allowed



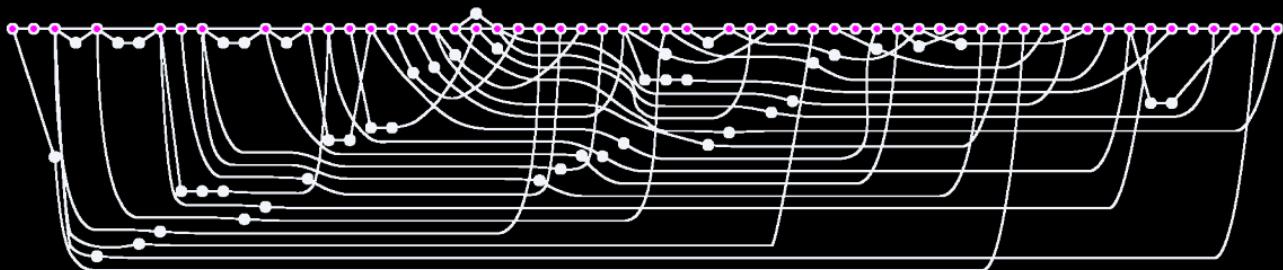
$$\frac{(k-1)!}{(2^{k-n}-1)!} \leq g_{n,k} \leq \binom{n-2}{k-2} \frac{(k-1)!}{(2^{k-n}-1)!} \quad \text{if } k > \frac{n+1}{2}$$

Consequence: $\sum_{n,k} g_{n,k} z^n u^k$ is not analytic, but some asymptotic analysis can be done -

MOST GIT GRAPHS LOOK ALIKE

Theorem

In a Git graph of size n taken uniformly at random,
the number of magenta vertices is $\frac{n}{2} + \sigma(n)$

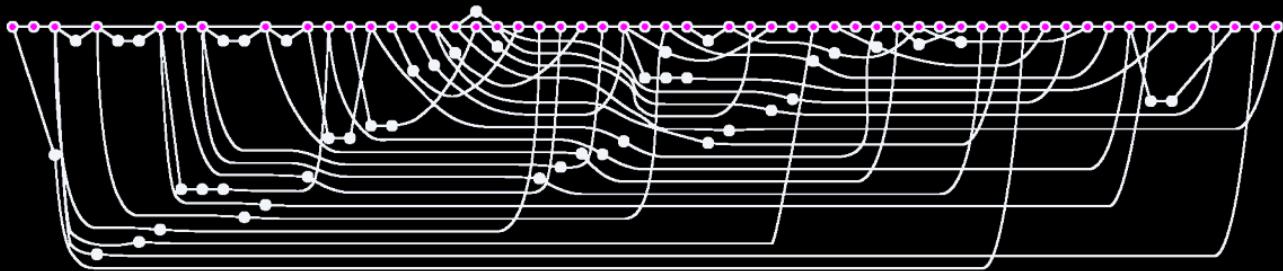


random Git graph of size 100

MOST GIT GRAPHS LOOK ALIKE

Theorem

In a **Git graph** of size n taken uniformly at random,
the number of magenta vertices is $\frac{n}{2} + \sigma(n)$



random **Git graph** of size 100

Sampling a **Git graph** is more interesting if
we fix size n and number of magenta vertices

TRANSFORMING THE EQUATION

Recurrence	$g_{m,k} = g_{m-1,k-1} + \sum_{l \geq 1} (k-l) g_{m-1-l,k-1}$
Differential Equation	$G(\gamma, u) = 1 + \gamma u G(\gamma, u) + \frac{\gamma^2 u^2}{1-\gamma} \frac{\partial G}{\partial u} (\gamma, u)$

Usual trick:

Ordinary
Generating
Function

$$\sum_{m,k \geq 0} g_{m,k} \gamma^m u^k$$

$G(\gamma, u),$
not analytic \times

TRANSFORMING THE EQUATION

Recurrence	$g_{m,k} = g_{m-1,k-1} + \sum_{l \geq 1} (k-l) g_{m-1-l,k-1}$
Differential Equation	$G(z, u) = 1 + z u G(z, u) + \frac{z^2 u^2}{1-z} \frac{\partial G}{\partial u}(z, u)$

Usual trick:

Ordinary Generating Function

$$\underbrace{\sum_{m,k \geq 0} g_{m,k} z^m u^k}_{G(z, u)},$$

not analytic \times

Borel transform

Exponential Generating Function

$$\underbrace{\sum_{m,k \geq 0} \frac{g_{m,k}}{m!} z^m u^k}_{\text{analytic, but no pretty equation } \times}$$

TRANSFORMING THE EQUATION

Recurrence	$g_{m,k} = g_{m-1,k-1} + \sum_{l \geq 1} (k-l) g_{m-1-l,k-1}$
Differential Equation	$G(\gamma, u) = 1 + \gamma u G(\gamma, u) + \frac{\gamma^2 u^2}{1-\gamma} \frac{\partial G}{\partial u} (\gamma, u)$

Usual trick:

Ordinary Generating Function

$$\underbrace{\sum_{n,k \geq 0} g_{m,k} \gamma^n u^k}_{G(\gamma, u)}$$

not analytic \times

Borel transform
on u

$$\tilde{G}(\gamma, u) = \sum_{n,k \geq 0} \frac{g_{m,k}}{k!} \gamma^n u^k \quad \text{and}$$

analytic \checkmark

Borel transform

Exponential Generating Function

$$\underbrace{\sum_{n,k \geq 0} \frac{g_{m,k}}{m!} \gamma^n u^k}_{\text{analytic}}$$

but no pretty equation \times

Differential Equation for \tilde{G}

$$\frac{\partial \tilde{G}}{\partial u} = \gamma \tilde{G} + \frac{\gamma^2 u}{1-\gamma} \frac{\partial \tilde{G}}{\partial u}$$

\checkmark

TRANSFORMING THE EQUATION

Recurrence	$g_{m,k} = g_{m-1,k-1} + \sum_{l \geq 1} (k-l) g_{m-1-l,k-1}$
Differential Equation	$G(\gamma, u) = 1 + \gamma u G(\gamma, u) + \frac{\gamma^2 u^2}{1-\gamma} \frac{\partial G}{\partial u} (\gamma, u)$

$$\tilde{G}(\gamma, u) = \sum_{m,k \geq 0} \frac{g_{m,k}}{k!} \gamma^m u^k$$

analytic ✓

and

Differential Equation for \tilde{G}

$$\frac{\partial \tilde{G}}{\partial u} = \gamma \tilde{G} + \frac{\gamma^2 u}{1-\gamma} \frac{\partial \tilde{G}}{\partial u}$$

✓

TRANSFORMING THE EQUATION

Recurrence

$$g_{m,k} = g_{m-1,k-1} + \sum_{l \geq 1} (k-l) g_{m-1-l,k-1}$$

Differential
Equation

$$G(z, u) = 1 + z u G(z, u) + \frac{z^2 u^2}{1-z} \frac{\partial G}{\partial u}(z, u)$$

$$\tilde{G}(z, u) = \sum_{m, k \geq 0} \frac{g_{m,k}}{k!} z^m u^k$$

analytic ✓

and

Differential Equation for \tilde{G}

$$\frac{\partial \tilde{G}}{\partial u} = z \tilde{G} + \frac{z^2 u}{1-z} \frac{\partial \tilde{G}}{\partial u}$$

✓

Theorem

$$\tilde{G}(z, u) = \left(1 - \frac{z^2 u}{1-z}\right)^{-\frac{1-z}{z}}$$

this can
be solved!

How can it be exploited?

DIFFERENT PERSPECTIVE

Is there a combinatorial explanation for the formula

$$\tilde{G}(r_0, u) = \left(1 - \frac{r_0^2 u}{1 - r_0}\right)^{-\frac{1-r_0}{r_0}} ?$$

DIFFERENT PERSPECTIVE

Is there a combinatorial explanation for the formula

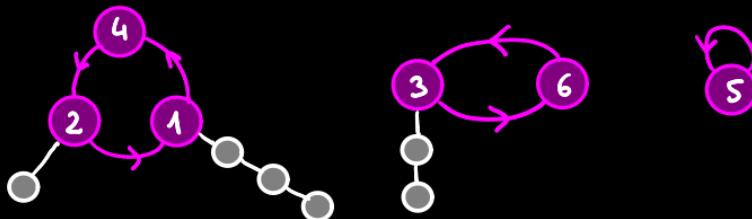
$$\tilde{G}(r_0, u) = \left(1 - \frac{r_0^2 u}{1 - r_0}\right)^{-\frac{1 - r_0}{r_0}} = \exp\left(\frac{1}{\frac{r_0}{1 - r_0}} \ln\left(\frac{1}{1 - \frac{u r_0}{1 - r_0}}\right)\right) ?$$

DIFFERENT PERSPECTIVE

Definition

set of cycles of
magenta vertices labeled from 1 to k
where a chain of white unlabeled vertices
is attached to each magenta vertex,
except to the ones having the
largest label in their cycles.

e.g.:



Is there a combinatorial explanation for the formula

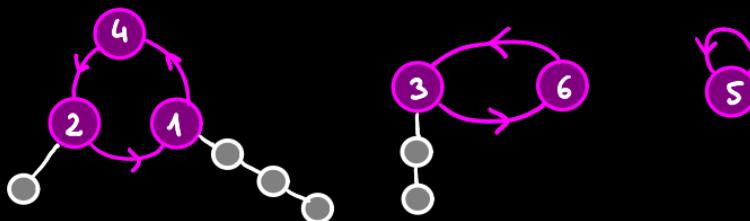
$$\tilde{G}(z_0, u) = \left(1 - \frac{z_0^2 u}{1-z_0}\right)^{-\frac{1-z_0}{z_0}} = \exp\left(-\frac{1}{\frac{z_0}{1-z_0}} \ln\left(\frac{1}{1-u z_0 \frac{z_0}{1-z_0}}\right)\right)?$$

DIFFERENT PERSPECTIVE

Definition

set of cycles of
magenta vertices labeled from 1 to k
where a chain of white unlabeled vertices
is attached to each magenta vertex,
except to the ones having the
largest label in their cycles.

e.g.:



Is there a combinatorial explanation for the formula

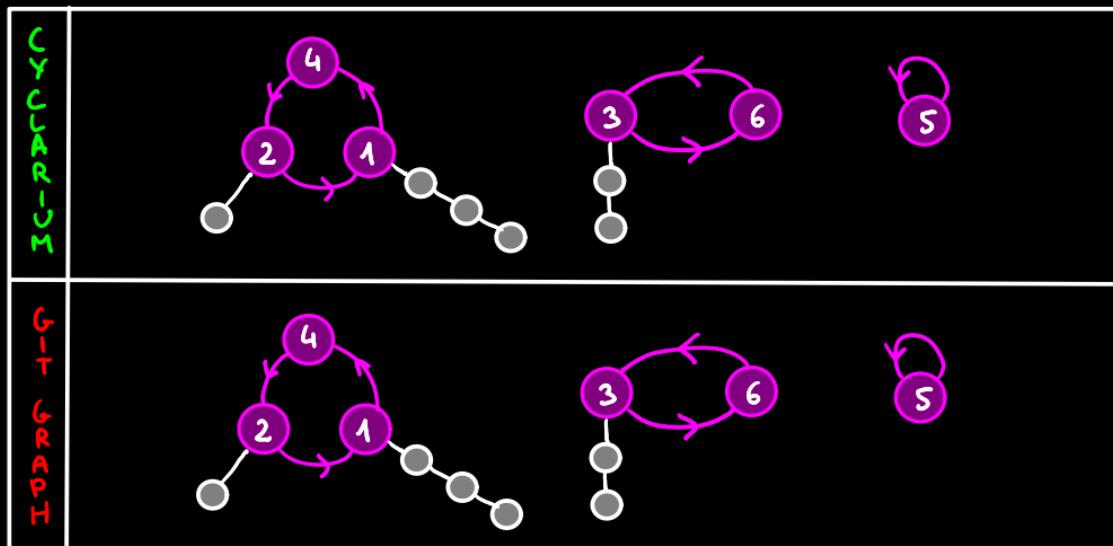
$$\tilde{G}(z_0, u) = \left(1 - \frac{z_0^2 u}{1-z_0}\right)^{-\frac{1-z_0}{z_0}} = \exp\left(-\frac{1}{\frac{z_0}{1-z_0}} \ln\left(\frac{1}{1-u z_0 \frac{z_0}{1-z_0}}\right)\right)$$

It's the generating function of cycariums!

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

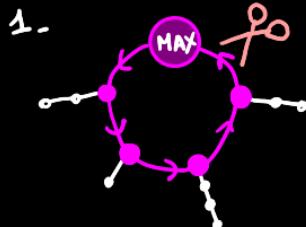


BIJECTION

Proposition

There is a bijection from cyclariums to Git graphs:

STEPS



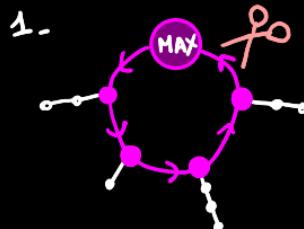
CYCLARIUM	GIT GRAPH
1-	<p>A diagram showing a Git graph corresponding to the cyclarium in step 1. It consists of 5 nodes. Nodes 1 and 2 are connected by a bidirectional edge. Node 3 is a single node. Node 4 is a single node. Node 5 is a single node. There are no edges between nodes 3, 4, and 5.</p>
STEPS	GIT GRAPH
2-	<p>A diagram showing a Git graph corresponding to the cyclarium in step 2. It consists of 5 nodes. Nodes 1 and 2 are connected by a bidirectional edge. Node 3 is a single node. Node 4 is a single node. Node 5 is a single node. There are no edges between nodes 3, 4, and 5.</p>

BIJECTION

Proposition

There is a bijection from cyclarums to Git graphs:

STEPS



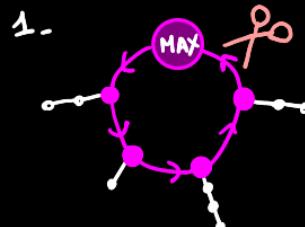
CYCLARIUM	Git Graph
<p>A diagram of a cyclarium with 6 nodes. Nodes 1, 2, and 4 are highlighted in purple, while 3, 5, and 6 are grey. Directed edges between highlighted nodes are purple, while edges between grey nodes and between a grey node and a purple node are grey.</p>	<p>A diagram of a Git graph with 6 nodes. Nodes 1, 2, and 4 are purple, while 3, 5, and 6 are grey. Directed edges between purple nodes are purple, while edges between grey nodes and between a grey node and a purple node are grey.</p>
Git Graph	

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

STEPS



2.



CYLARIUM	4	2	1	3	6	5
GIT GRAPH	4	2	1	6	3	5

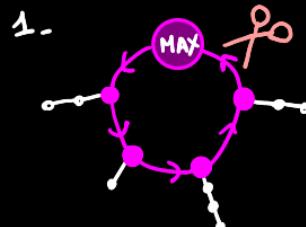
The diagram illustrates the mapping between a cylarium and a Git graph. The top row shows the cylarium structure with nodes 4, 2, 1, 3, 6, and 5. The bottom row shows the corresponding Git graph structure. Nodes 4, 2, 1, and 3 form a sequence where each node has a downward arrow pointing to the next node. Nodes 6 and 5 are isolated nodes. The nodes are color-coded: purple for nodes 4, 2, 1, 3, and 6; grey for nodes 5 and 1's child node; and white for 2's child node and 3's child nodes.

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

STEPS



2.



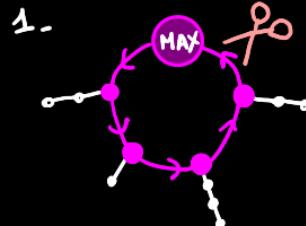
CYLARIUM	Git Graph
Git Graph	Git Graph

BIJECTION

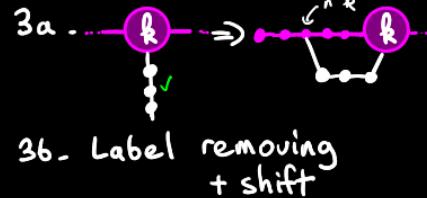
Proposition

There is a bijection from cylariums to Git graphs:

STEPS



3. Right to left:



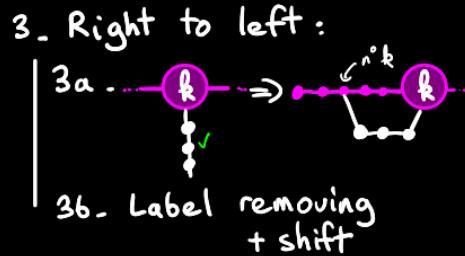
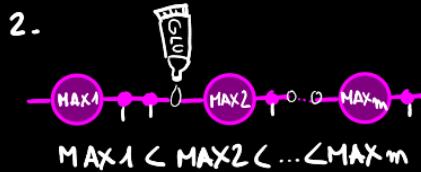
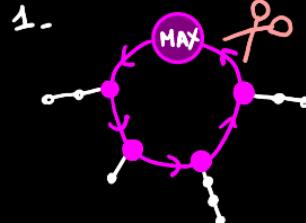
CYLARIUM	Git Graph
Git Graph	Git Graph

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

STEPS



CYLARIUM	Git Graph
Git Graph	Git Graph

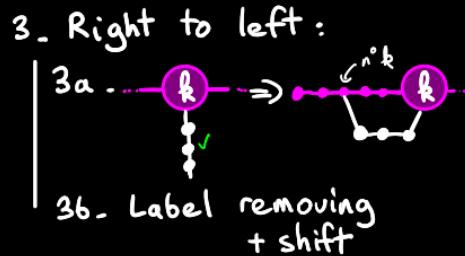
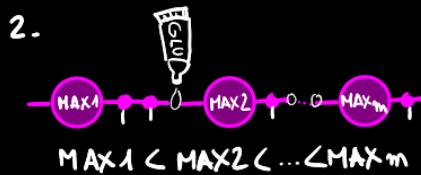
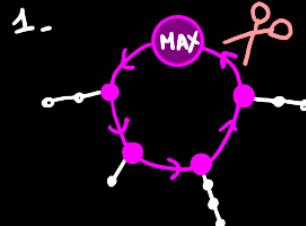
The diagram shows two rows of graphs. The top row is labeled "CYLARIUM" and the bottom row is labeled "GIT GRAPH". The first graph in each row is a cylarium with nodes 1, 2, 3, 4, and 5. The second graph in each row is a Git graph with nodes 4, 2, 1, 5, 6, and 3. A red arrow labeled "n°3" points from the cylarium node 1 to the Git graph node 1. A red arrow also points from the Git graph node 3 back to the cylarium node 5.

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

STEPS



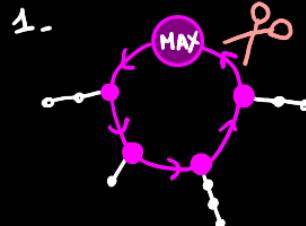
CYLARIUM	Git Graph
Git Graph	

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

STEPS



3. Right to left:



3b. Label removing + shift

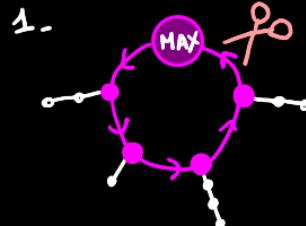
CYLARIUM	Git Graph
Git Graph	Git Graph

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

STEPS



3. Right to left:



3b. Label removing + shift

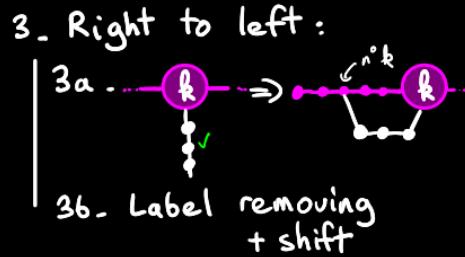
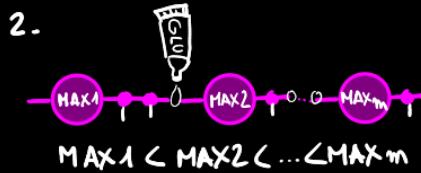
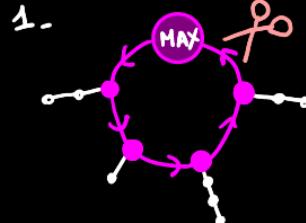
CYLARIUM	Git Graph

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

STEPS



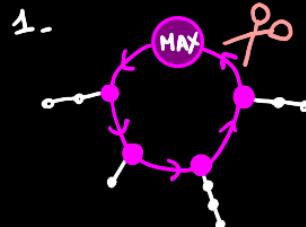
CYLARIUM	Git Graph

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

STEPS



2.



3. Right to left:



3b. Label removing + shift

CYLAR IUM	Git Graph
Git Graph	Git Graph

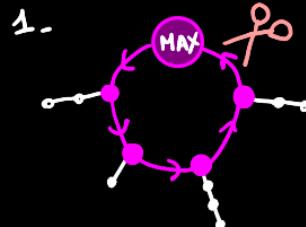
The diagram shows two rows of graphs. The top row, labeled 'CYLAR IUM' on the left, contains three separate components: a cluster of four nodes (1, 2, 3, 4) with a cycle between them, a cluster of two nodes (5, 6) with a cycle between them, and a single node (5) by itself. The bottom row, labeled 'GIT GRAPH' on the left, contains two identical clusters of four nodes each. Each cluster has a central node (1) connected to three other nodes. Nodes 1, 2, and 3 form a triangle, while node 4 is connected to nodes 1 and 2. Red arrows point from the top row to the bottom row, indicating the mapping process.

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

STEPS



2.



3. Right to left:



3b. Label removing + shift

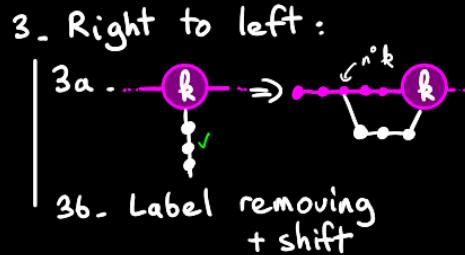
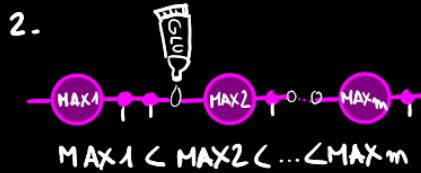
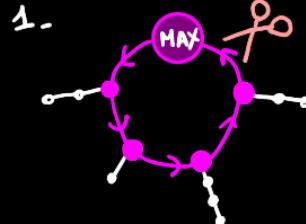
CYLARIUM	Git Graph

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

STEPS



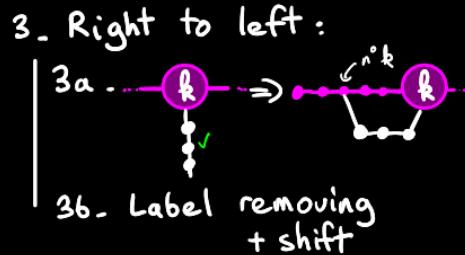
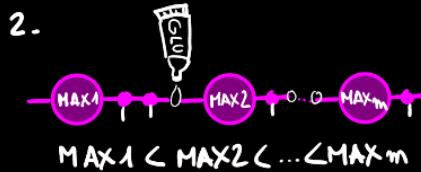
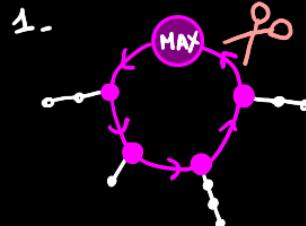
CYLARIUM	Git Graph
Git Graph	

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

STEPS



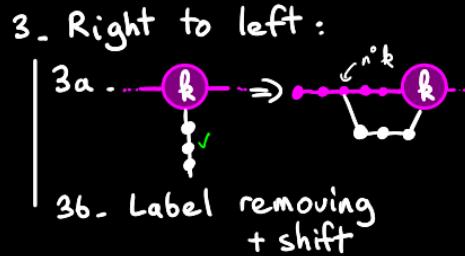
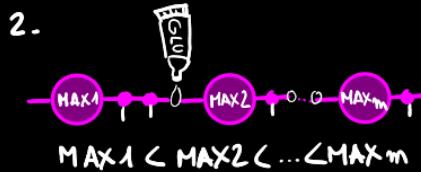
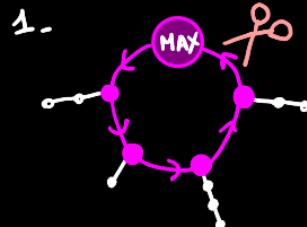
CYLARIUM	Git Graph
Git Graph	

BIJECTION

Proposition

There is a bijection from cylariums to Git graphs:

STEPS



CYLARIUM	Git Graph
Git Graph	Cylarium

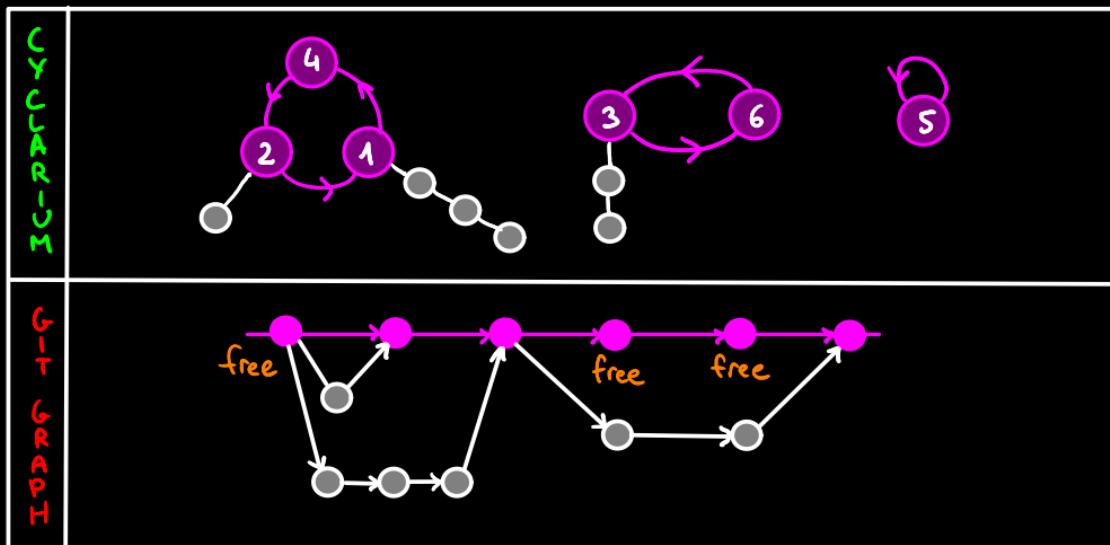
The diagram shows two rows of graphs. The top row, labeled 'CYLARIUM' on the left, contains three graphs: a small cylarium with nodes 1, 2, 4, and a larger one with nodes 3, 5, 6. The bottom row, labeled 'GIT GRAPH' on the left, contains two graphs: a complex branched graph on the left and a simple linear chain with a loop on the right.

BIJECTION

Proposition

There is a bijection from cyclariums to Git graphs:
sending

vertices	→	vertices
magenta vertices	→	magenta vertices
cycles	→	<u>free vertices</u> i.e magenta vertices of indegree ≤ 1
cycle lengths	→	gaps between free vertices

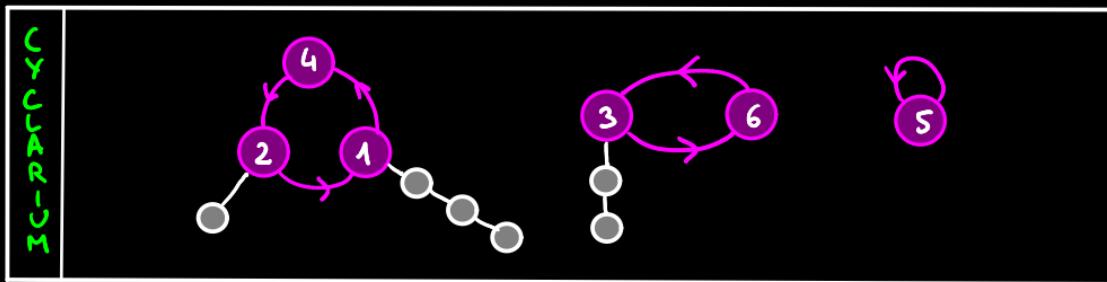


A NICE FORMULA

Proposition

There is a bijection from cyclariums to Git graphs:
sending

vertices	→ vertices
magenta vertices	→ magenta vertices
cycles	→ <u>free vertices</u> i.e. magenta vertices of indegree ≤ 1
cycle lengths	→ gaps between free vertices



Corollary

$$g_{m,k} = \sum_{f=1}^{k-1} \left[\begin{matrix} k \\ f \end{matrix} \right] \binom{m-k-1}{k-f-1} \quad (k < m)$$

where $g_{m,k}$ = number of Git graphs counted by vertices & magenta vertices
and $\left[\begin{matrix} : \\ : \end{matrix} \right]$ = (unsigned) Stirling number of 1st kind

RANDOM MODEL

“Boltzmann model” (exponential in α , ordinary in β)

Fix $\beta > 0^*$ and $\alpha > 0^*$.

We wish to draw a *Git graph* δ with a weight proportional to $\beta^{\# \text{vertices in } \delta} \frac{\alpha^{\# \text{magenta vertices in } \delta}}{(\# \text{magenta vertices in } \delta)!}$
 (Size is not fixed)

Examples

$$P(\text{---}) \propto 1$$

$$P(\bullet) \propto \beta \alpha$$

$$P(\bullet\bullet\bullet\bullet\bullet) \propto \beta^5 \frac{\alpha^4}{24}$$

$$P(\bullet\bullet\bullet\bullet) \propto \beta^5 \frac{\alpha^3}{6}$$

*: in the disk of convergence of \tilde{G}

RANDOM MODEL

“Boltzmann model” (exponential in u , ordinary in γ)

Fix $\gamma > 0^*$ and $u > 0^*$.

We wish to draw a Git graph δ with a weight proportional to $\frac{\gamma^{\# \text{vertices in } \delta}}{\tilde{G}(\gamma, u)} \frac{u^{\# \text{magenta vertices in } \delta}}{(\# \text{magenta vertices in } \delta)!}$

equal

(Size is not fixed)

$$\text{where } \tilde{G}(\gamma, u) = \sum_{m, k \geq 0} \frac{g_{m, k}}{k!} \gamma^m u^k = \left(1 - \frac{\gamma^2 u}{1 - \gamma}\right)^{-\frac{1 - \gamma}{\gamma}}$$

Examples

$$P(\text{---}) = \frac{1}{\tilde{G}(\gamma, u)}$$

$$P(\bullet) = \frac{\gamma u}{\tilde{G}(\gamma, u)}$$

$$P(\text{--- --- --- ---}) = \frac{\gamma^5 u^4}{\tilde{G}(\gamma, u) 24}$$

$$P(\text{--- --- --- ---}) = \frac{\gamma^5 u^3}{\tilde{G}(\gamma, u) 6}$$

*: in the disk of convergence of \tilde{G}

RANDOM MODEL

Proposition

Let γ be a random **Graph** sampled with respect to the previous Boltzmann model, conditioned to have size n

$$\mathbb{E}(\# \text{magenta vertices}(\gamma)) \sim \frac{1 - \rho_u}{2 - \rho_u} n$$

$$\mathbb{V}(\# \text{magenta vertices}(\gamma)) \sim \frac{\rho_u(1 - \rho_u)}{(2 - \rho_u)^3} n$$

$$\text{where } \rho_u = \frac{\sqrt{1 + 4u} - 1}{2u}$$

Proof : Transfer Theorem from $\tilde{G}(z, u) = \left(1 - \frac{z^2 u}{1 - z}\right)^{-\frac{1 - z}{z}}$

RANDOM MODEL

Proposition

Let γ be a random **Git Graph** sampled with respect to the previous Boltzmann model, conditioned to have size n

$$\mathbb{E}(\# \text{magenta vertices}(\gamma)) \sim \frac{1 - \rho_u}{2 - \rho_u} n$$

$$\mathbb{V}(\# \text{magenta vertices}(\gamma)) \sim \frac{\rho_u(1 - \rho_u)}{(2 - \rho_u)^3} n$$

$$\text{where } \rho_u = \frac{\sqrt{1 + 4u} - 1}{2u}$$

Proof : Transfer Theorem from $\tilde{G}(z_1, u) = \left(1 - \frac{z_1^2 u}{1 - z_1}\right)^{-\frac{1-z_1}{z_1}}$

Consequence : A random generator for **Git graphs** with $\approx n$ vertices and $\approx k$ magenta vertices ($k \leq \frac{n}{2}$)

1. Tune u so that $\frac{1 - \rho_u}{2 - \rho_u} = \frac{k}{n}$

3. Make a Boltzmann sampler with parameters z_1 and u for cyclariums.

2. Tune z_1 so that $z_1 = \rho_u - \frac{1 - \rho_u}{n}$

4. Bijection to Git graphs

PERSPECTIVES ABOUT RANDOM GIT GRAPHS

→ Asymptotic behaviour

- Asymptotic equivalent of # Git graphs of size n ? ▷ Collaboration with Fang
- Limit Laws
- Phase transition?

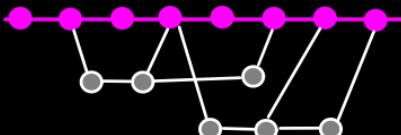
PERSPECTIVES ABOUT RANDOM GIT GRAPHS

→ Asymptotic behaviour

- Asymptotic equivalent of # Git graphs of size n ? ▷ Collaboration with Fang
- Limit Laws
- Phase transition?

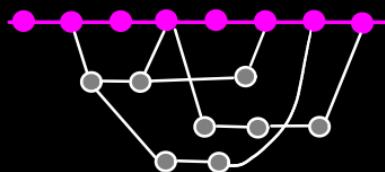
→ Other random models ▷ Collaboration with Clement + Marechal

Phoenix
graphs



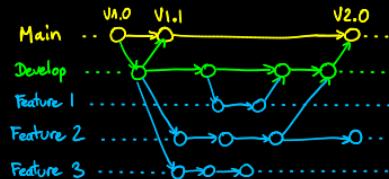
merged branches can be reborn

Fork Anywhere
graphs



branches can be born anywhere
but must be merged into main

More involved
workflows



LINKS BETWEEN GIT GRAPHS & GIT BISECT

Still to do: Average-case complexity of git bisect
where the input is taken w.r.t the Boltzmann distribution.

LINKS BETWEEN GIT GRAPHS & GIT BISECT

Still to do: Average-case complexity of git bisect
where the input is taken w.r.t the Boltzmann distribution.

But also: - Is there a polynomial algorithm for the
Regression Problem when the input is a Git graph?

(We proved that git bisect fails to be optimal for some Git graphs)

- Is the Regression Problem NP-complete when the input is binary?
- Other algorithms from Version Control Systems to be analyzed?

THANK YOU!



STEAK UN
AU REVOIR!

GIT À LA NOIX! —

