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)

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?

god-like

good

trash

spawn of the devil

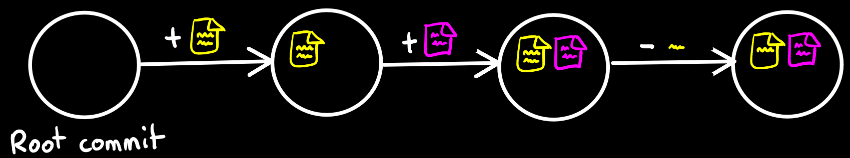
TIER LIST MADE IN MARNE

<u>QUESTION</u>: What do you use to share files with your coauthors?

god-like	git	mercurial
good	SUBVERSION	
trash		Overleaf
spawn of the devil	Dropbox	

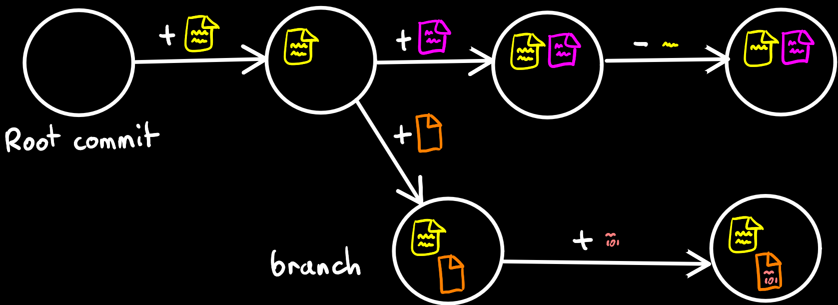
GIT FEATURES

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



GIT FEATURES

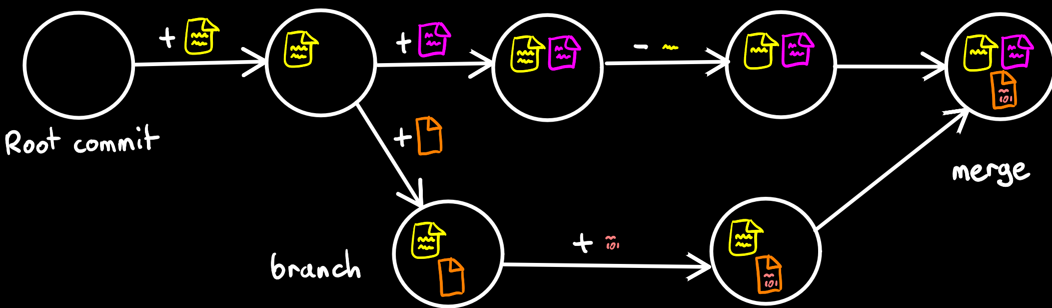
git is a Version Control System (VCS) : it stores all the project states over time.



Git lets you create parallel development branches...

GIT FEATURES

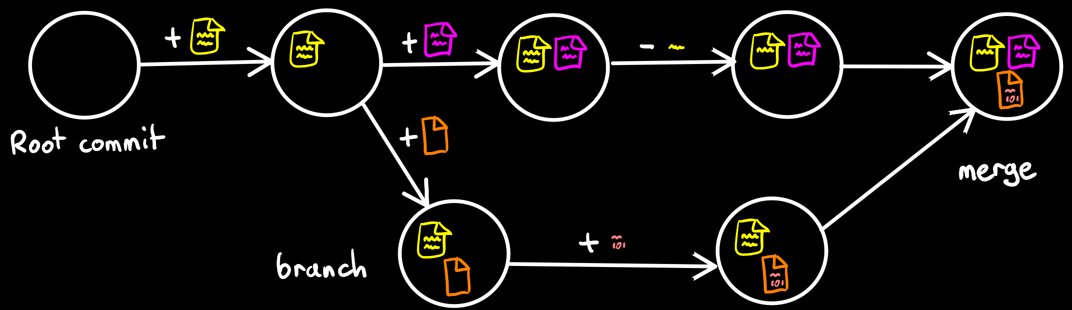
git is a Version Control System (VCS): it stores all the project states over time.



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.

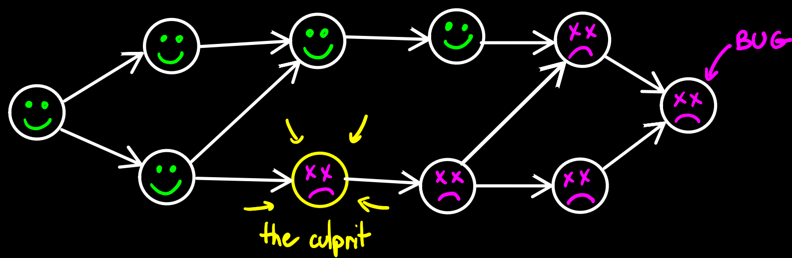


Git lets you create parallel development branches that can be integrated later.

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

PART I
GIT BISECT

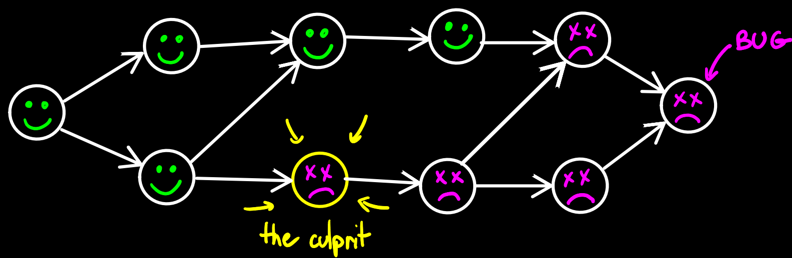
PROBLEM: FINDING A REGRESSION



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

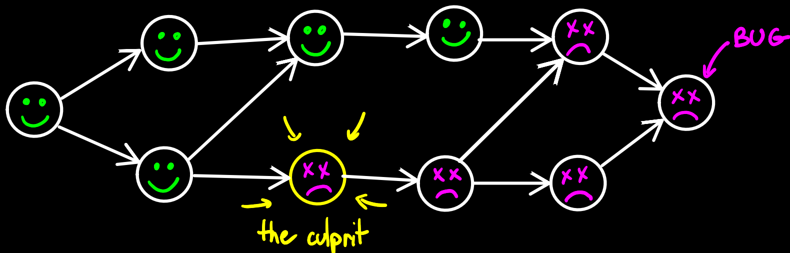


Input A commit graph in which a commit is known to be bugged, the other commits are bugged or clean (= bug-free)

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



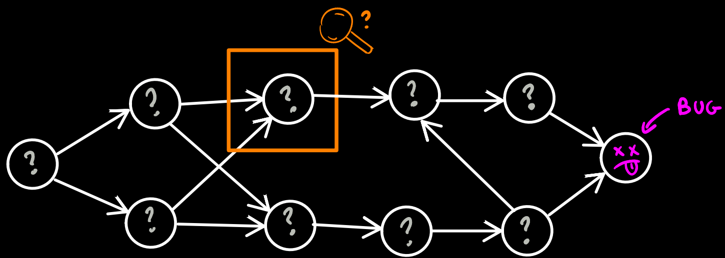
Input A commit graph in which a commit is known to be bugged, the other commits are bugged or clean (= bug-free)

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)

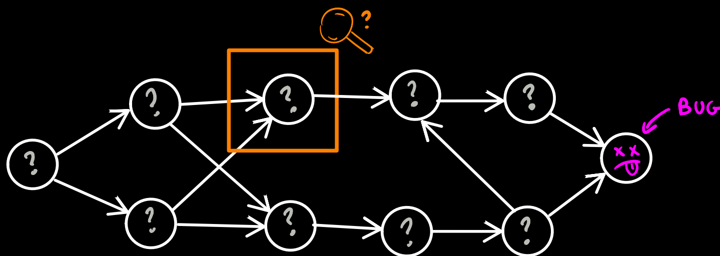
HOW TO INVESTIGATE

Unique operation: QUERY of a commit with unknown status



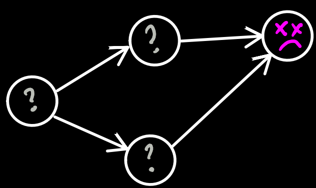
HOW TO INVESTIGATE

Unique operation: QUERY of a commit with unknown status



If bugged,

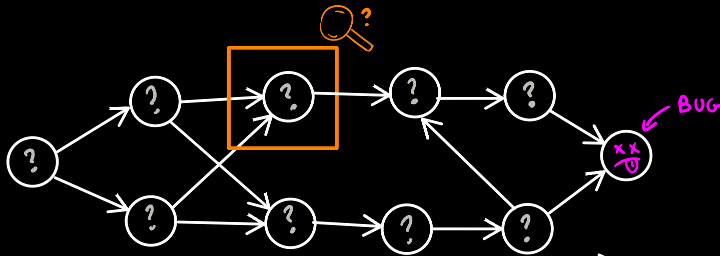
then the faulty commit is an ancestor of this commit



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

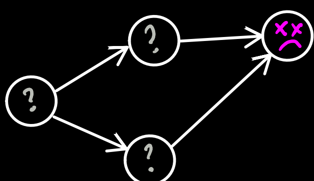
HOW TO INVESTIGATE

Unique operation: QUERY of a commit with unknown status



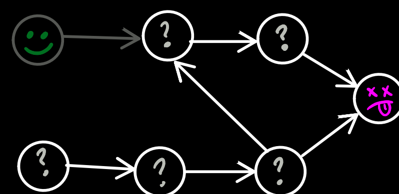
If bugged,

then the faulty commit is an ancestor of this commit



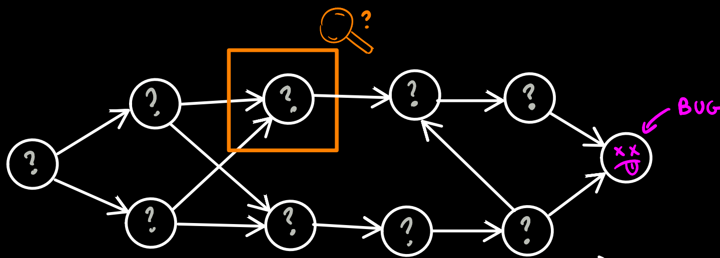
If clean,

then the faulty commit is not an ancestor of this commit

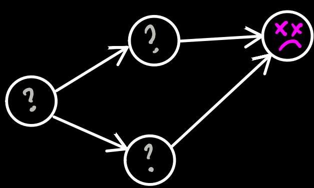


HOW TO INVESTIGATE

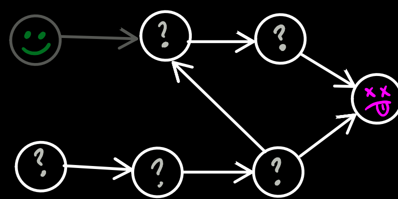
Unique operation: QUERY of a commit with unknown status



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



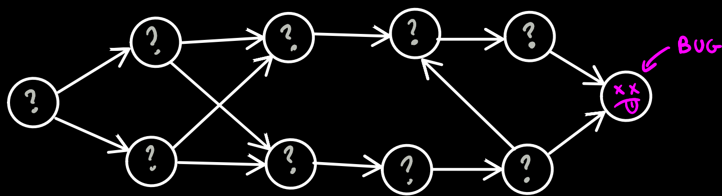
If clean, then the faulty commit is not an ancestor of this commit



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.

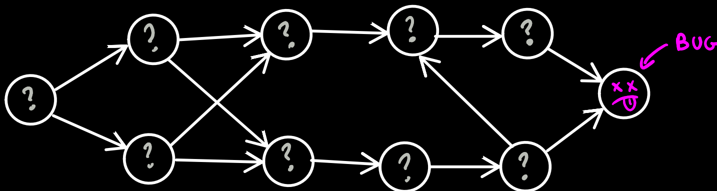


Output: A strategy that finds the faulty commit with a minimal number of queries in the worst-case scenario
= optimal strategy

(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.



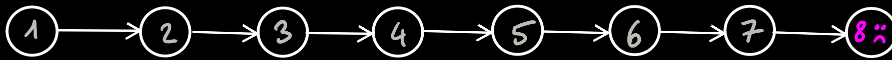
Output: A strategy that finds the faulty commit with a minimal number of queries in the worst-case scenario

= optimal strategy

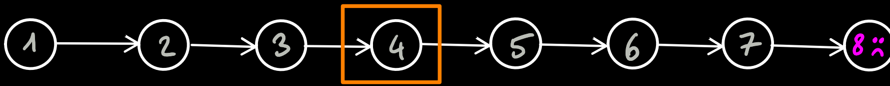
(the faulty commit can be any ancestor of the bugged vertex)

In real life, queries are costly.

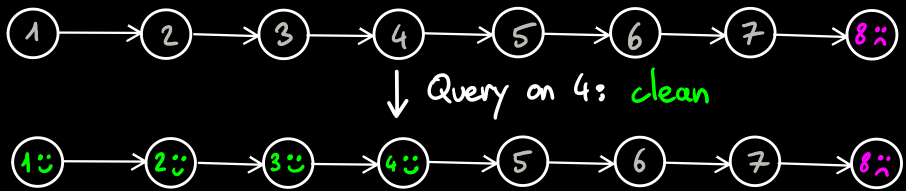
FIRST EXAMPLE : CHAINS



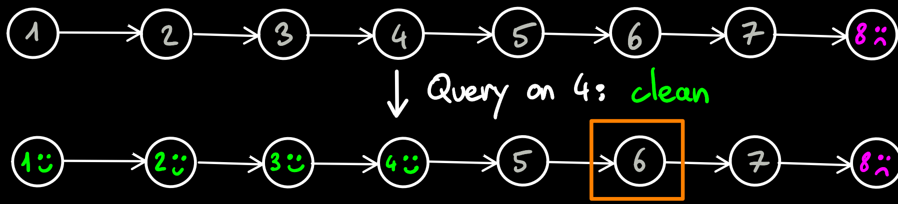
FIRST EXAMPLE : CHAINS



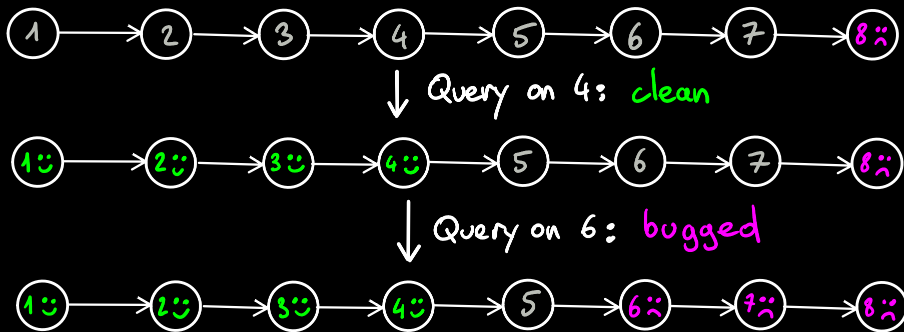
FIRST EXAMPLE : CHAINS



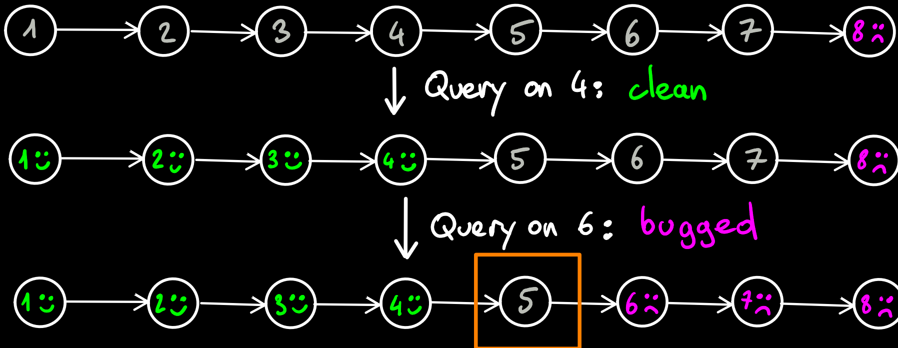
FIRST EXAMPLE : CHAINS



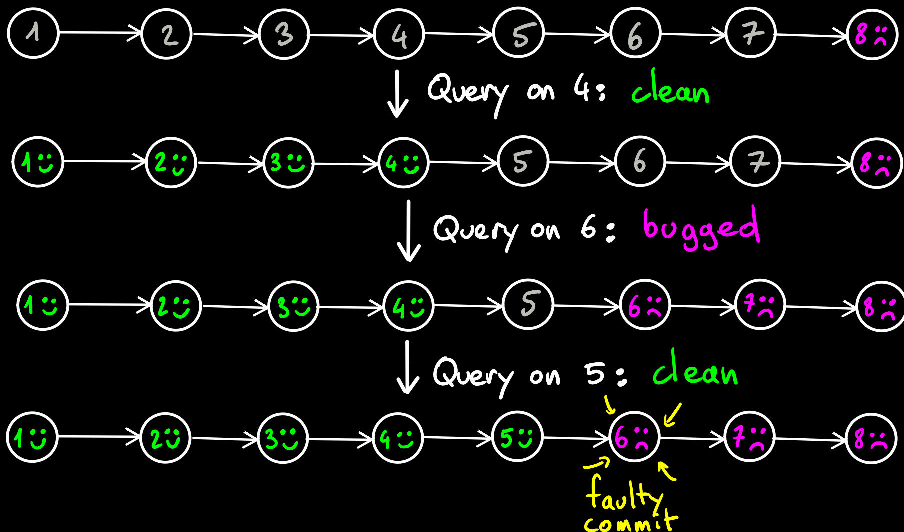
FIRST EXAMPLE : CHAINS



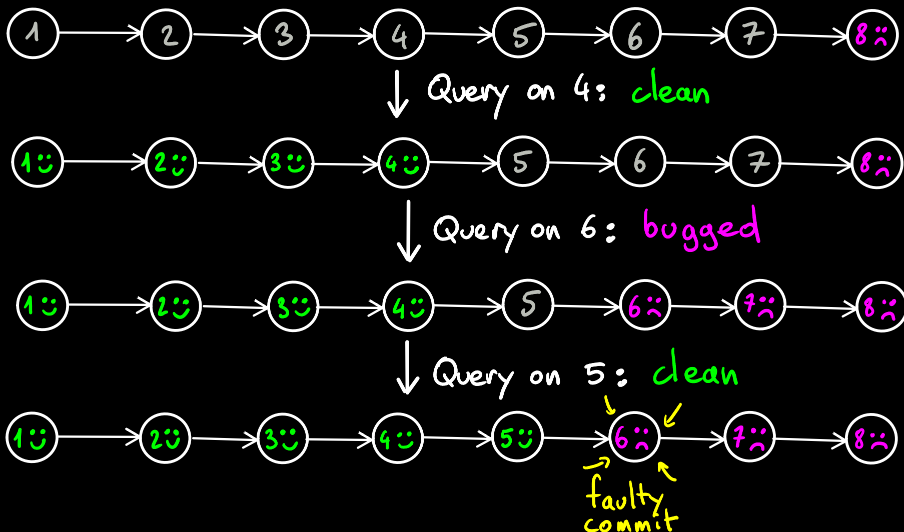
FIRST EXAMPLE : CHAINS



FIRST EXAMPLE : CHAINS

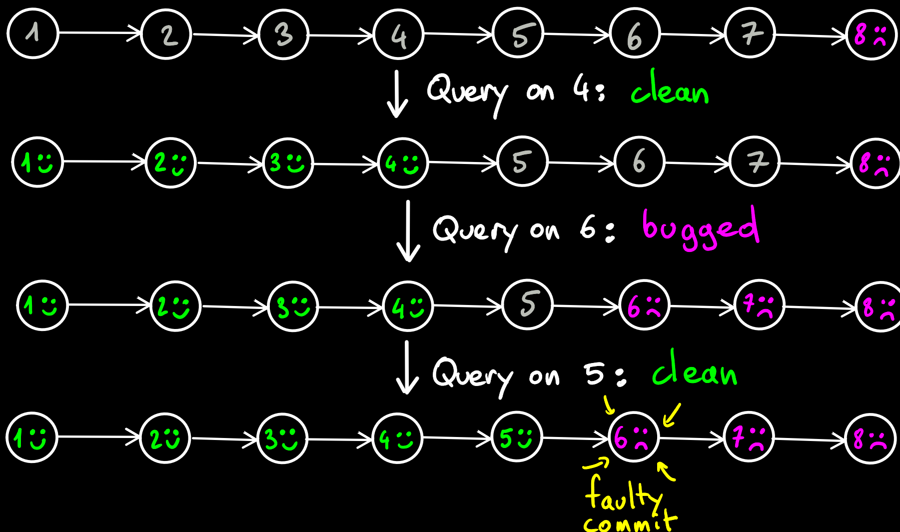


FIRST EXAMPLE : CHAINS



optimal strategy = binary search

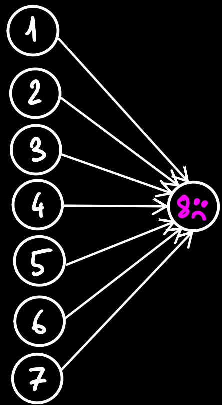
FIRST EXAMPLE : CHAINS



optimal strategy = binary search

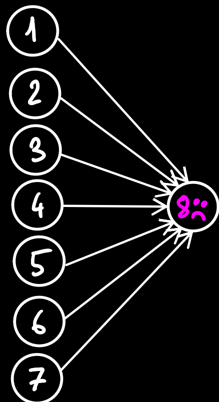
More generally, number of queries in an optimal strategy in a chain of length n = $\lceil \log_2(n) \rceil$

SECOND EXAMPLE : OCTOPUSES



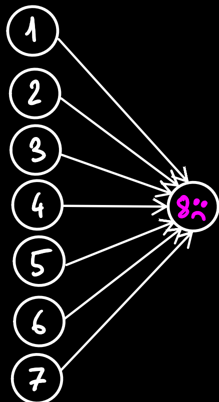
optimal strategy =

SECOND EXAMPLE : OCTOPUSES



optimal strategy = whatever

SECOND EXAMPLE : OCTOPUSES



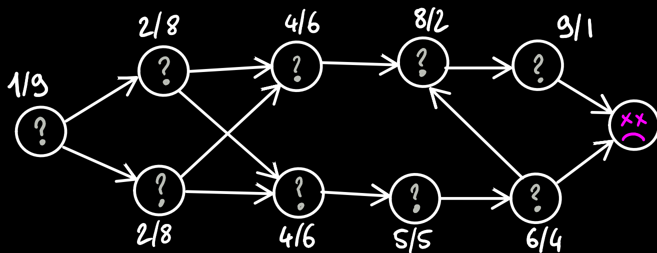
optimal strategy = whatever

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

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

STEP 1:



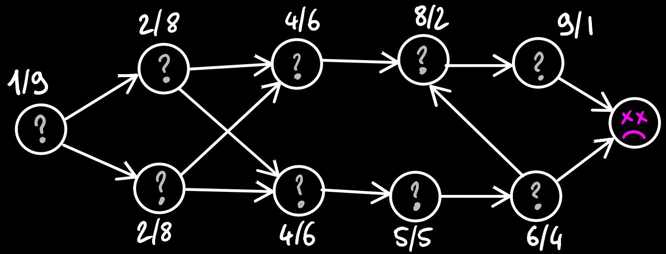
STEP 2 :

STEP 3 :

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

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



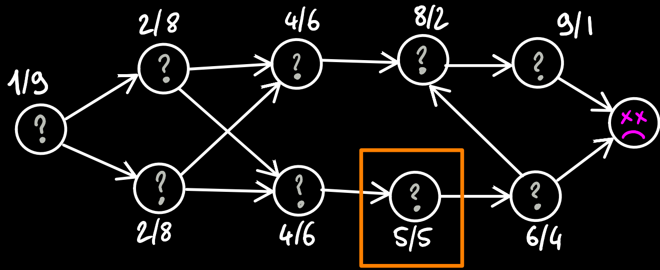
STEP 2 :

STEP 3 :

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

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



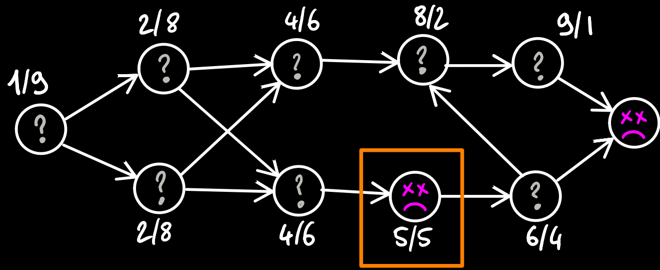
STEP 2: Query the vertex with the most balanced ratio

STEP 3:

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

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



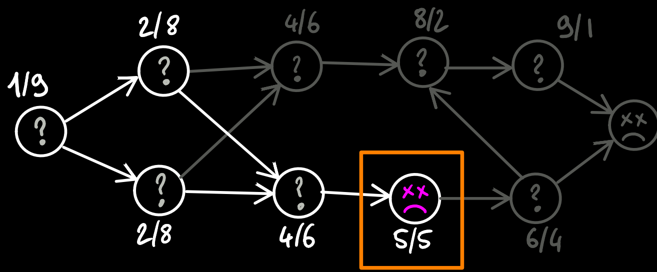
STEP 2 : Query the vertex with the most balanced ratio

STEP 3 :

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

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



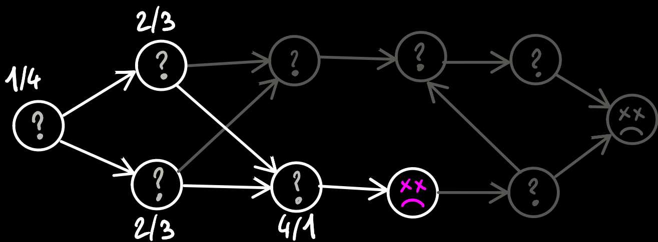
STEP 2 : Query the vertex with the most balanced ratio

STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

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



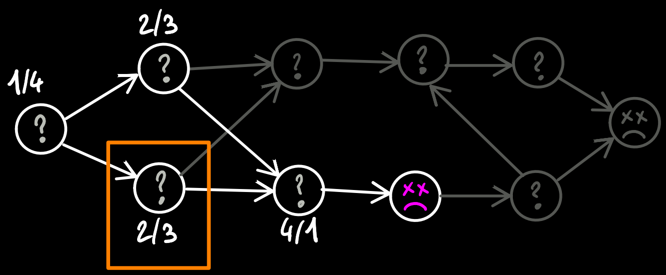
STEP 2: Query the vertex with the most balanced ratio

STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

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



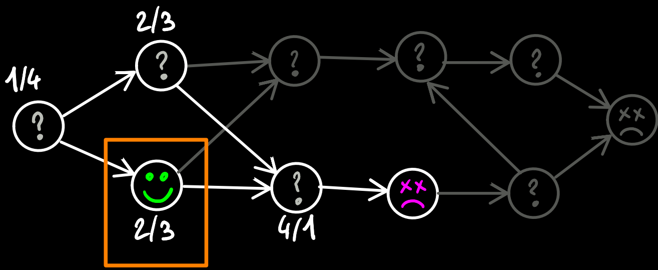
STEP 2: Query ~~the~~ a vertex with the most balanced ratio

STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

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



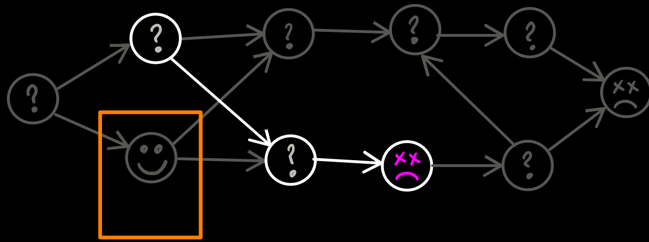
STEP 2: Query ~~the~~ a vertex with the most balanced ratio

STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

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



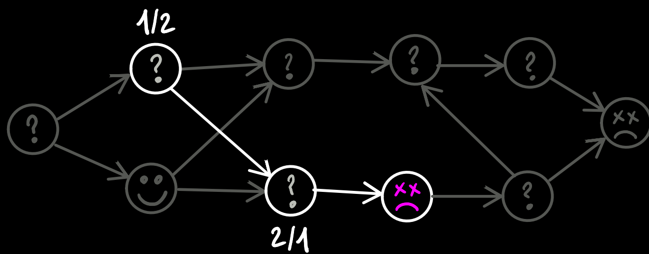
STEP 2: Query ~~the~~ a vertex with the most balanced ratio

STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

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



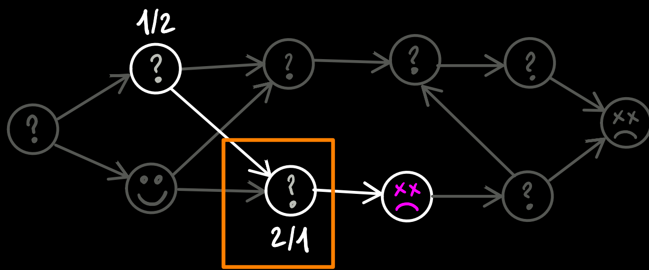
STEP 2: Query ~~the~~ a vertex with the most balanced ratio

STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

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



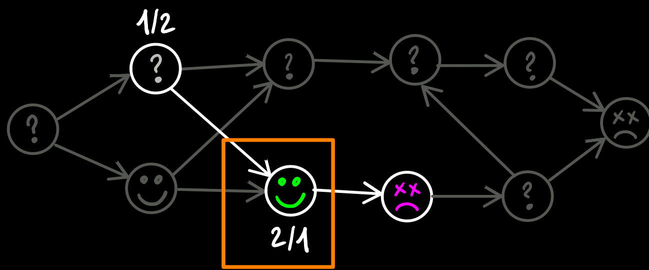
STEP 2: Query ~~the~~ a vertex with the most balanced ratio

STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

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



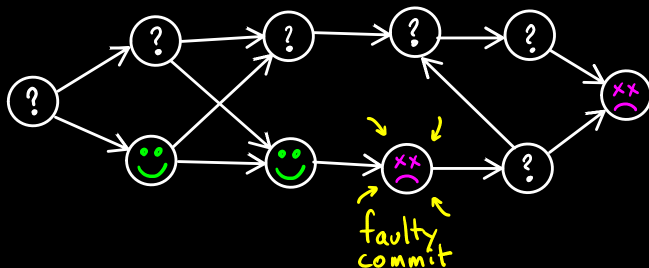
STEP 2: Query ~~the~~ a vertex with the most balanced ratio

STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM

git uses a heuristic to find the faulty commit: git bisect

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



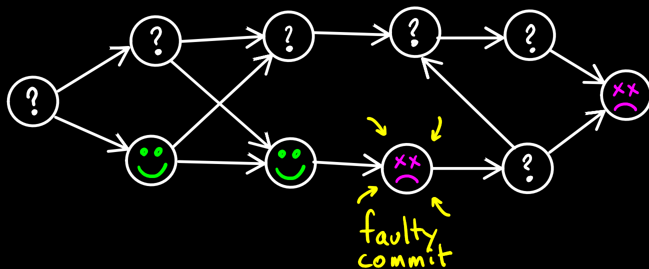
STEP 2: Query ~~the~~ a vertex with the most balanced ratio

STEP 3: Delete the innocent commits and recurse.

THE GIT BISECT ALGORITHM

git 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~~ a vertex with the most balanced ratio

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?

$\boxed{\text{NO}}$ The Regression Search Problem is NP-complete.

[Carmo Donadelli Kohayakawa Laber 2004]

[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 2004] [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?

$\boxed{\text{NO}}$ The Regression Search Problem is NP-complete.

[Carmo Donadelli Kohayakawa Laber 2004]

[We've also proved it!]

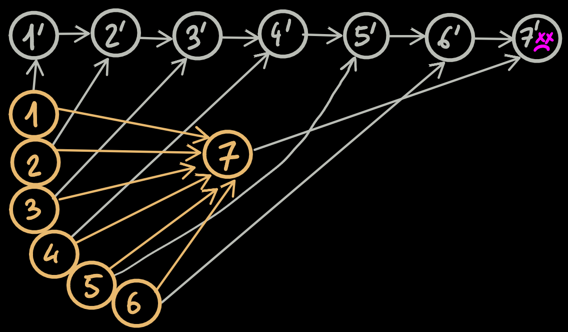
But is it really that bad? $\boxed{\text{YEAH}}$

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.

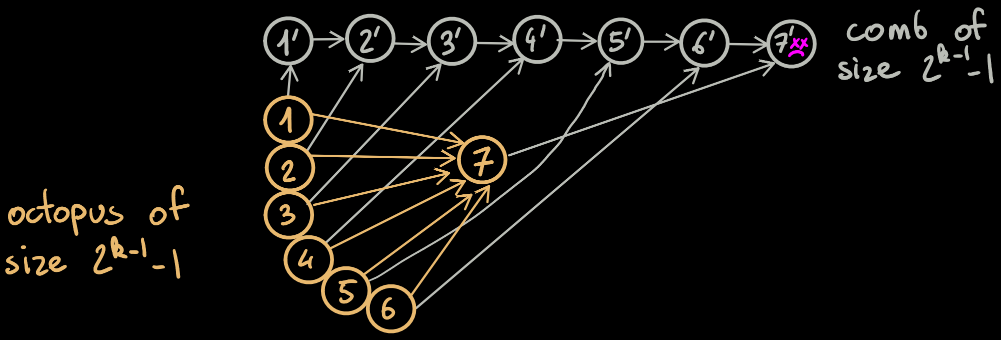
Proof for $k=4$:



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.

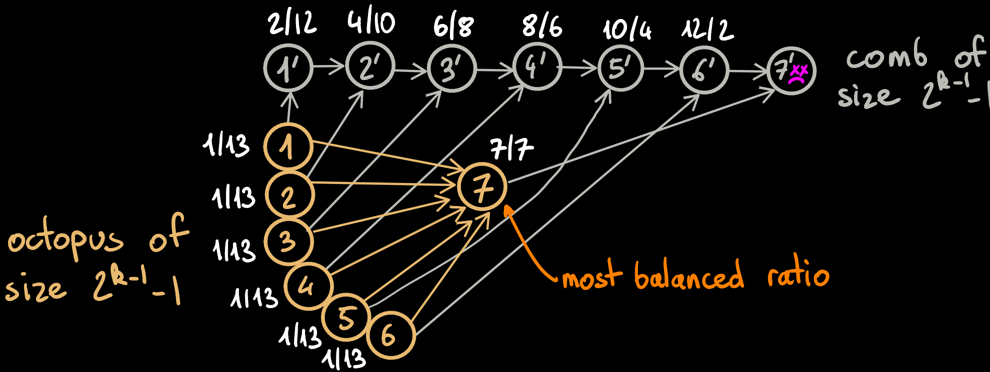
Proof for $k=4$:



HOW GOOD IS GIT BISECT?

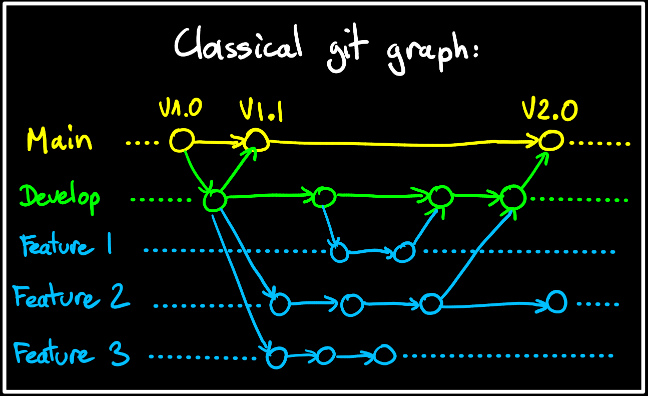
Proposition For any k, there exists a DAG such that an optimal strategy uses k queries and git bisect always uses $2^{k-1}-1$ queries.

Proof for $k=4$:



BACK TO REALITY?

Octopus substructures are unrealistic



Usually, we never merge more than 2 branches.

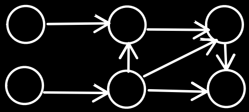
(Otherwise it is called an octopus merge )

BINARY DAGS

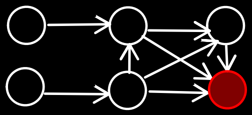
Definition

binary DAG = DAG where the vertices have indegree ≤ 2

Ex:



Good



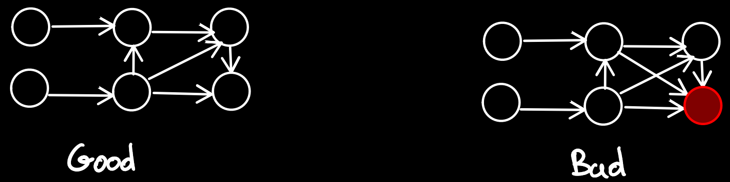
Bad

BINARY DAGS

Definition

binary DAG = DAG where the vertices have indegree ≤ 2

Ex:



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

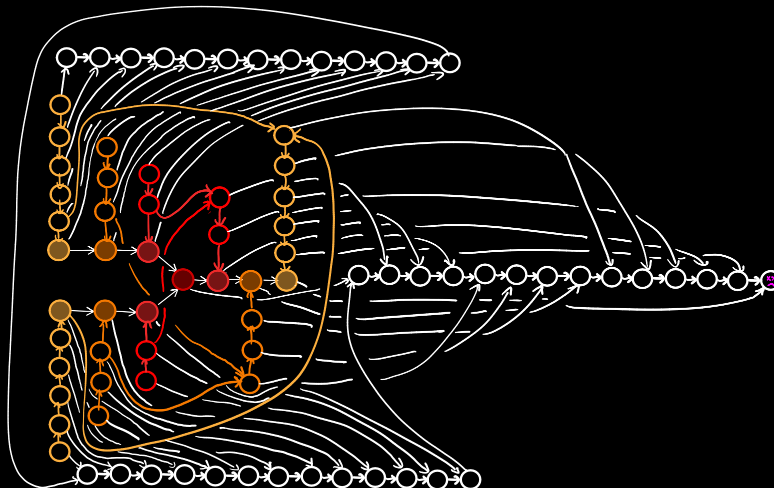
$$\frac{1}{\log_2\left(\frac{3}{2}\right)} \simeq 1{,}71$$ is the optimal constant.

BINARY DAGS

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

$\dfrac{1}{\log_2\left(\frac{3}{2}\right)} \simeq 1{,}71$ 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

$\longrightarrow$ 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 ϕ = golden ratio.
$= (1+\sqrt{5})/2$

$\frac{1}{\log_2(\phi)} \simeq 1{,}44$ 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

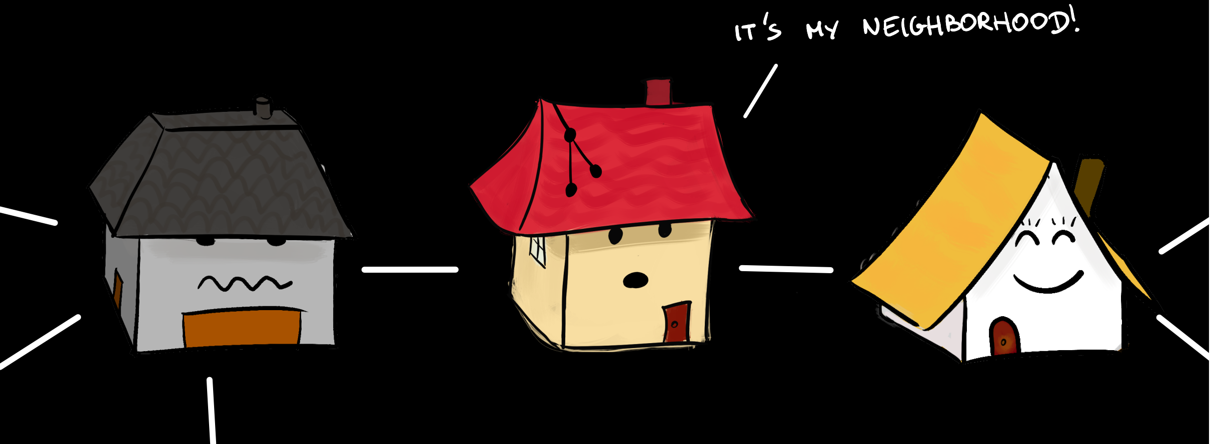
→ Sometimes git bisect is better than golden bisect

More relevant(?) question	What is the average-case complexity of git bisect?

This question calls for many more, notably <u>what is a random Git graph?</u>

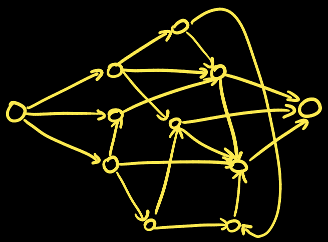
Part II
Random Git Graphs

ongoing work



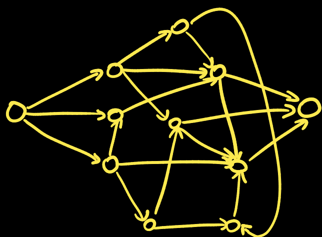
WHICH GRAPHS TO CONSIDER ?

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

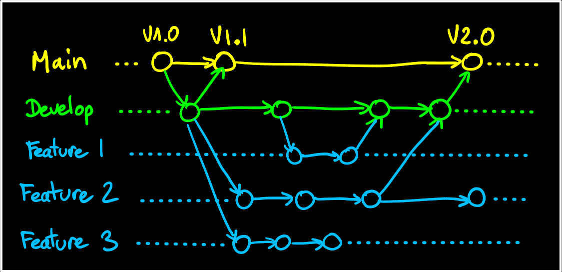


WHICH GRAPHS TO CONSIDER ?

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

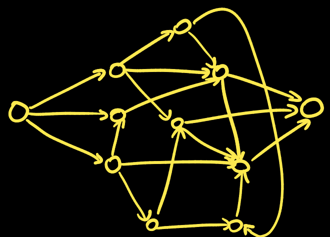


... but many projects follow a workflow

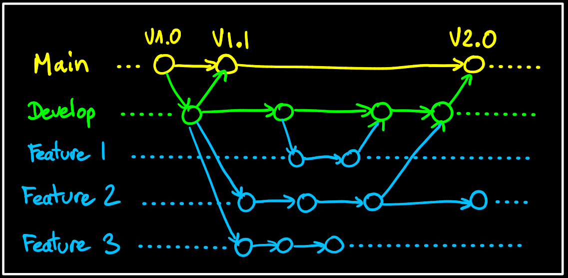


WHICH GRAPHS TO CONSIDER ?

In git, 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 <u>feature branch workflow</u>

GIT GRAPH

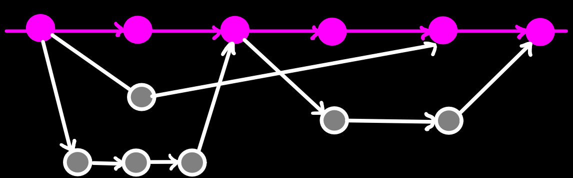
DEFINITION

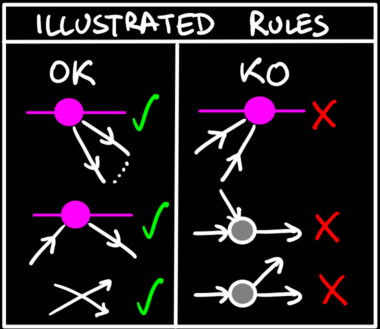
(feature branch)
Git graph = DAG with

- 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.



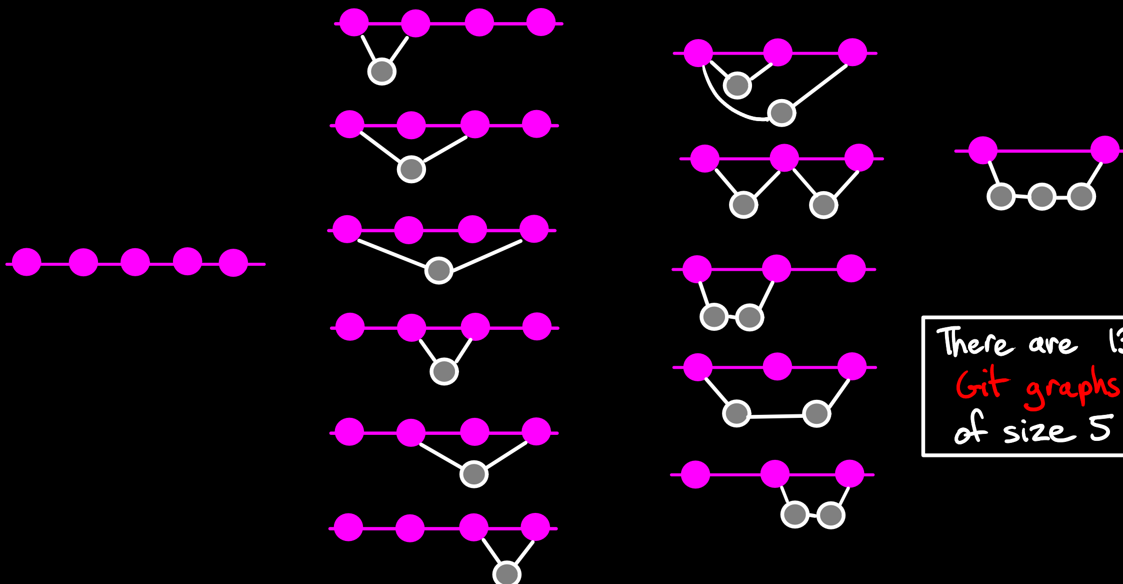


OBJECTIVES

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

OBJECTIVES

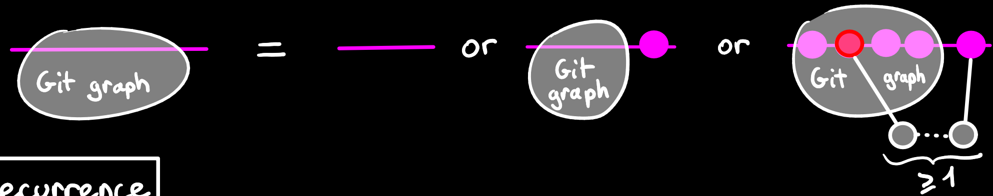
GOALS	→ Count Git graphs (exact & asymptotic) → Sample a Git graph uniformly at random given a size n and a number k of magenta vertices



There are 13 Git graphs of size 5

RECURSIVE DECOMPOSITION

Decomposition



Recurrence

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

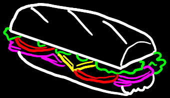
where $g_{n,k} :=$ number of Git graphs with n vertices, k of them being magenta

Differential Equation for the Generating Function

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

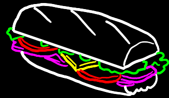
where $G(z,u) = \sum_{n \geq 0} \sum_{k \geq 0} g_{n,k}\, z^n u^k$

⚠ $G(z,u)$ is not analytic.

 SANDWICHING $g_{n,k}$

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

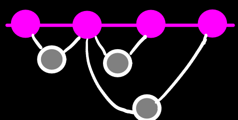
$$\boxed{\binom{n-k-1}{k-2}(k-1)! \leq g_{n,k} \leq \binom{n-2}{k-2}(k-1)! \quad \text{if } k \leq \frac{n+1}{2}}$$

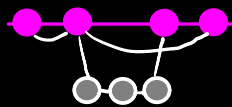


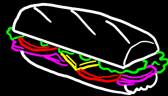
SANDWICHING $g_{n,k}$

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

$$\boxed{\underbrace{\binom{n-k-1}{k-2}(k-1)!}_{\text{nb of Git graphs where every magenta vertex has indegree 2}} \leq g_{n,k} \leq \underbrace{\binom{n-2}{k-2}(k-1)!}_{\text{nb of Git graphs where every magenta vertex has indegree 2 (except the 1st one) but branches with 0 white vertex are allowed}} \quad \text{if } k \leq \frac{n+1}{2}}$$



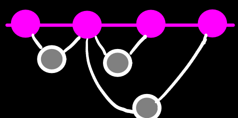


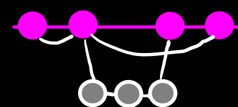


SANDWICHING $g_{n,k}$

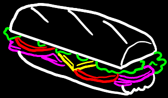
$g_{n,k}$ = number of Git graphs with n vertices, k of them being magenta

$$\boxed{\underbrace{\binom{n-k-1}{k-2}(k-1)!}_{\substack{\text{nb of Git graphs}\\ \text{where every magenta}\\ \text{vertex has indegree 2}}} \leq g_{n,k} \leq \underbrace{\binom{n-2}{k-2}(k-1)!}_{\substack{\text{nb of Git graphs}\\ \text{where every magenta}\\ \text{vertex has indegree 2}\\ \text{(except the 1st one)}\\ \text{but branches with 0}\\ \text{white vertex are allowed}}} \quad \text{if } k \leq \frac{n+1}{2}}$$





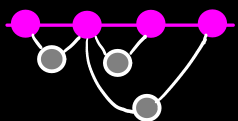
$$\boxed{\frac{(k-1)!}{(2k-n-1)!} \leq g_{n,k} \leq \binom{n-2}{k-2}\frac{(k-1)!}{(2k-n-1)!} \quad \text{if } k > \frac{n+1}{2}}$$

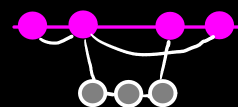


SANDWICHING $g_{n,k}$

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

$$\boxed{\underbrace{\binom{n-k-1}{k-2}(k-1)!}_{\text{nb of Git graphs where every magenta vertex has indegree 2}} \leq g_{n,k} \leq \underbrace{\binom{n-2}{k-2}(k-1)!}_{\text{nb of Git graphs where every magenta vertex has indegree 2 (except the 1st one) but branches with 0 white vertex are allowed}} \quad \text{if } k \leq \frac{n+1}{2}}$$





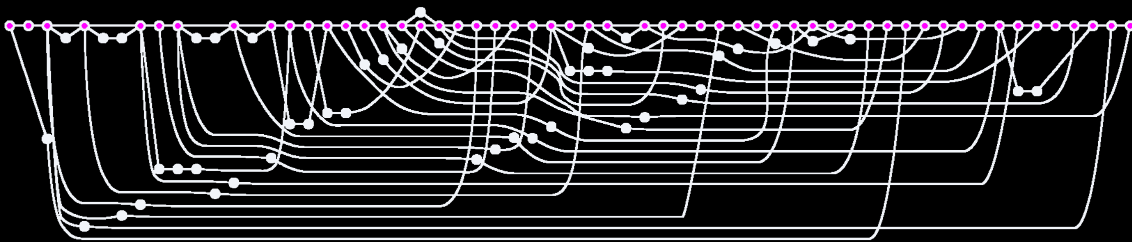
$$\boxed{\frac{(k-1)!}{(2k-n-1)!} \leq g_{n,k} \leq \binom{n-2}{k-2}\frac{(k-1)!}{(2k-n-1)!} \quad \text{if } k > \frac{n+1}{2}}$$

<u>Consequence:</u> $\sum_{n,k} g_{n,k} z^n u^k$ 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 $\frac{n}{2} + o(n)$

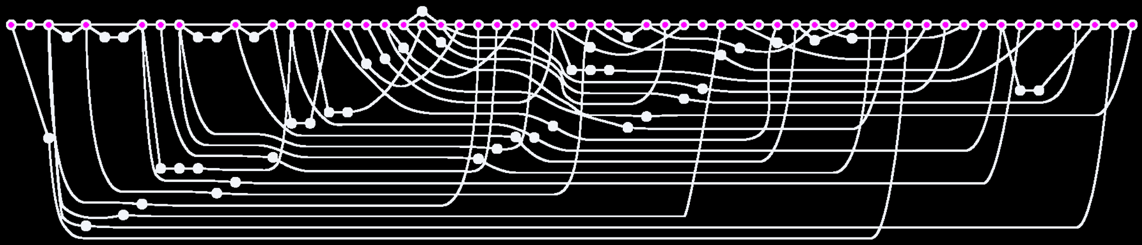


random Git graph of size 100

MOST GIT GRAPHS LOOK ALIKE

Theorem

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



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	$g_{n,k} = g_{n-1,k-1} + \sum_{\ell \geq 1} (k-1)\, g_{n-1-\ell,\, k-1}$
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 $\underbrace{\sum_{n,k \geq 0} g_{n,k} z^n u^k}_{G(z,u),}$

not analytic ✗

TRANSFORMING THE EQUATION

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

Usual trick:

Borel transform

Ordinary Generating Function $\underbrace{\sum_{n,k\geq 0} g_{n,k} z^n u^k}_{\substack{G(z,u),\\ \text{not analytic}}}$ ✗

Exponential Generating Function $\underbrace{\sum_{n,k\geq 0} \frac{g_{n,k}}{n!} z^n u^k}_{\substack{\text{analytic,}\\ \text{but no pretty equation}}}$ ✗

TRANSFORMING THE EQUATION

Recurrence	$g_{n,k} = g_{n-1,k-1} + \sum_{\ell \geq 1} (k-1)\, g_{n-1-\ell,\, k-1}$
Differential Equation	$G(z,u) = 1 + zu\, G(z,u) + \dfrac{z^2 u^2}{1-z} \dfrac{\partial G}{\partial u}(z,u)$

Usual trick:

Borel transform

Ordinary Generating Function $\underbrace{\sum_{n,k \geq 0} g_{n,k} z^n u^k}_{G(z,u),}$ not analytic ✗

Exponential Generating Function $\underbrace{\sum_{n,k \geq 0} \frac{g_{n,k}}{n!} z^n u^k}_{}$ analytic, but no pretty equation ✗

Borel transform on u

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

analytic ✓

and

Differential Equation for $\tilde{G}$
$\dfrac{\partial \tilde{G}}{\partial u} = z \tilde{G} + \dfrac{z^2 u}{1-z} \dfrac{\partial \tilde{G}}{\partial u}$

✓

TRANSFORMING THE EQUATION

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

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

analytic ✓

and

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

✓

TRANSFORMING THE EQUATION

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

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

analytic ✓

and

Differential Equation for $\tilde{G}$

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

this can be solved!

Theorem

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

How can it be exploited?

Different Perspective

Is there a combinatorial explanation for the formula

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

DIFFERENT PERSPECTIVE

Is there a combinatorial explanation for the formula

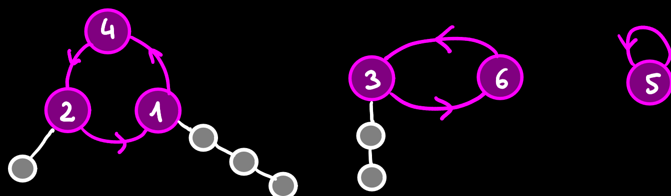
$$\tilde{G}(z,u) = \left(1 - \frac{z^2 u}{1-z}\right)^{-\frac{1-z}{z}} = \exp\left(\frac{1}{\frac{z}{1-z}} \ln\left(\frac{1}{1 - uz\frac{z}{1-z}}\right)\right) ?$$

Different Perspective

Definition

cyclarium = 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

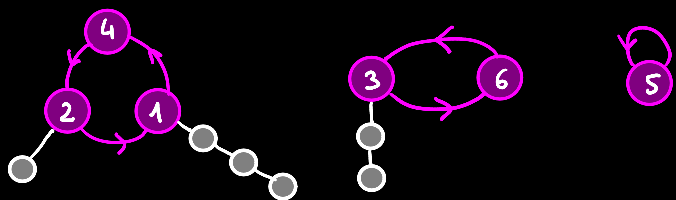
$$\tilde{G}(z,u) = \left(1 - \frac{z^2 u}{1-z}\right)^{-\frac{1-z}{z}} = \exp\left(\frac{1}{\frac{z}{1-z}} \ln\left(\frac{1}{1 - uz\frac{z}{1-z}}\right)\right) ?$$

DIFFERENT PERSPECTIVE

Definition

cyclarium = 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

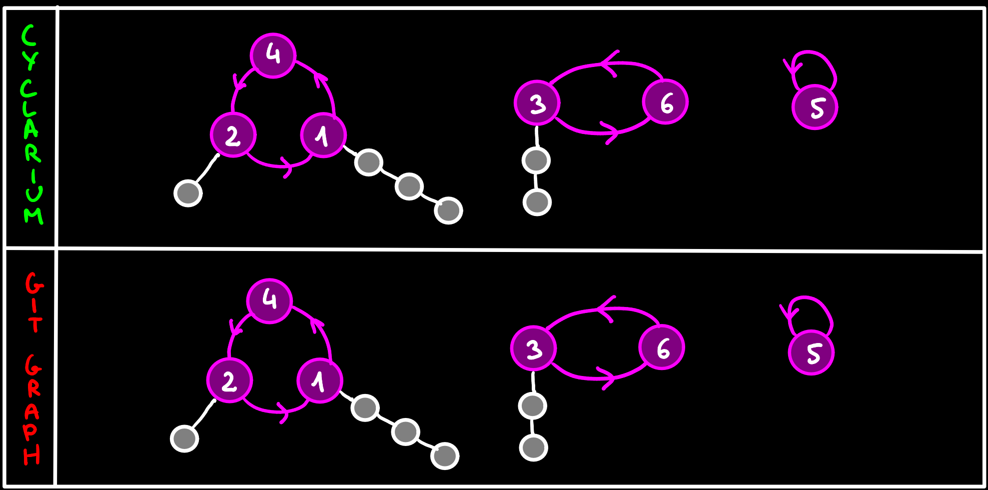
$$\tilde{G}(z,u) = \left(1 - \frac{z^2 u}{1-z}\right)^{-\frac{1-z}{z}} = \exp\left(\frac{1}{\frac{z}{1-z}} \ln\left(\frac{1}{1 - uz\frac{z}{1-z}}\right)\right) \; ?$$

It's the generating function of cyclariums!

BIJECTION

Proposition

There is a bijection from cycłariums to Git graphs:



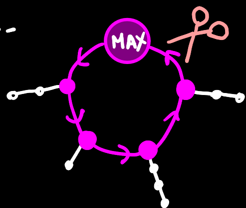
BIJECTION

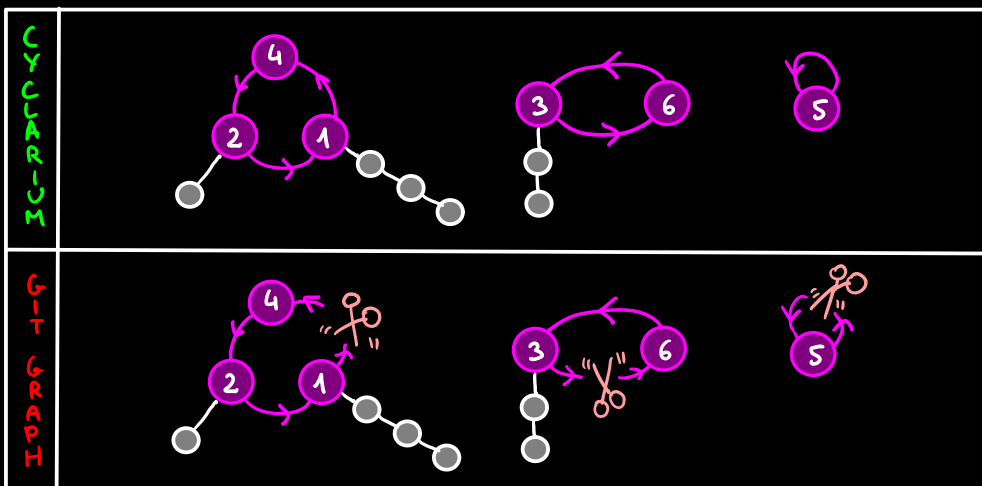
Proposition

There is a bijection from cyclariums to Git graphs:

STEPS

1.





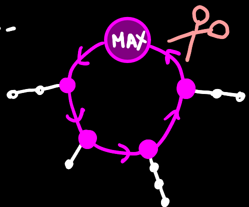
BIJECTION

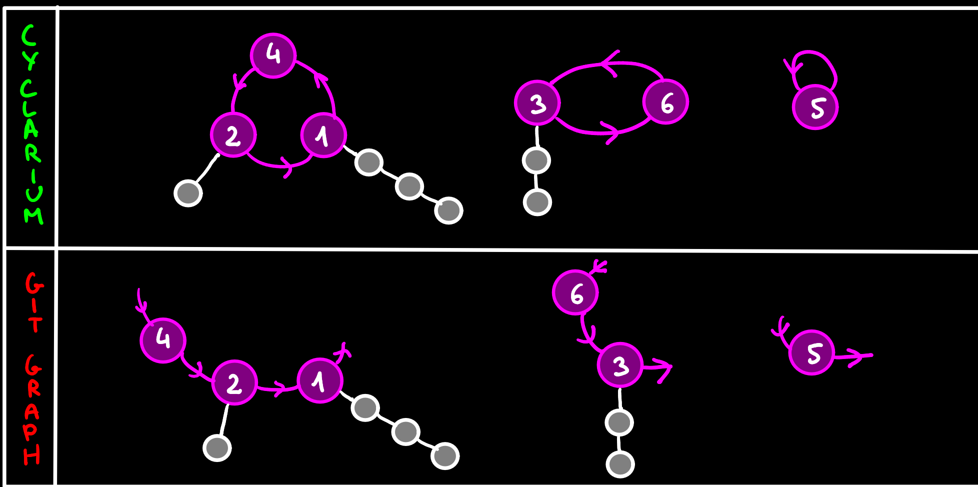
Proposition

There is a bijection from cycloriums to Git graphs:

STEPS

1.

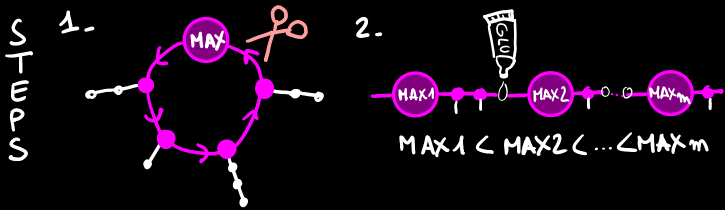


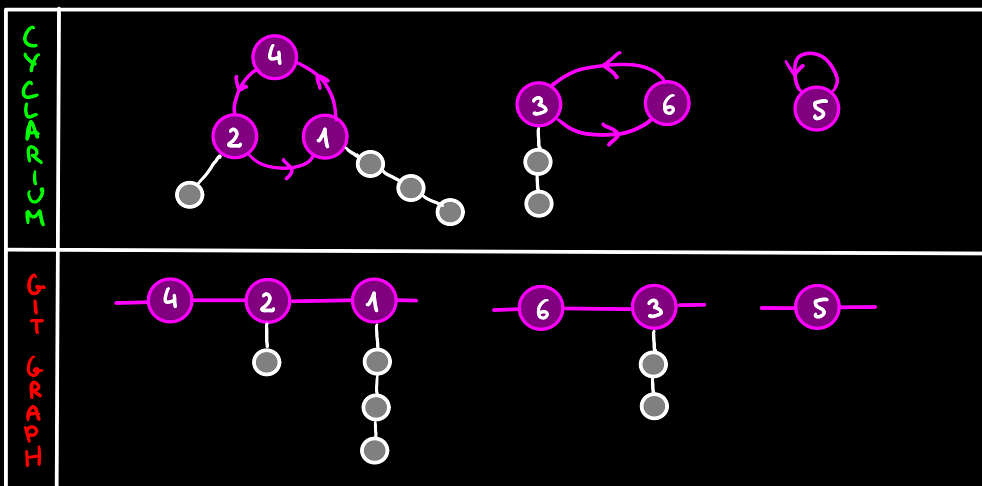


BIJECTION

Proposition

There is a bijection from cyclariums to Git graphs:





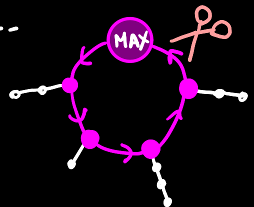
BIJECTION

Proposition

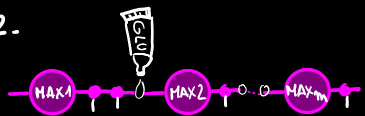
There is a bijection from cyclariums to Git graphs:

STEPS

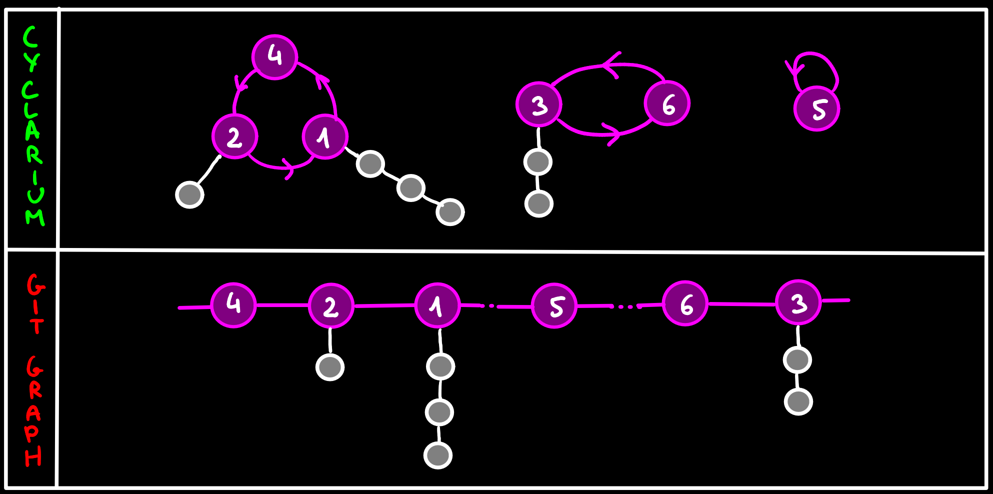
1.



2.



$MAX1 < MAX2 < \dots < MAXm$



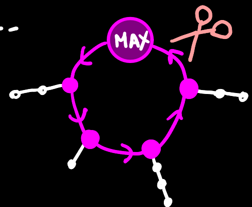
BIJECTION

Proposition

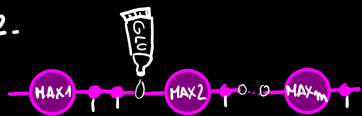
There is a bijection from cyclariums to Git graphs :

STEPS

1.



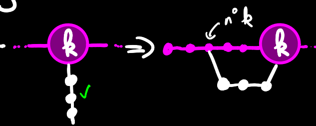
2.



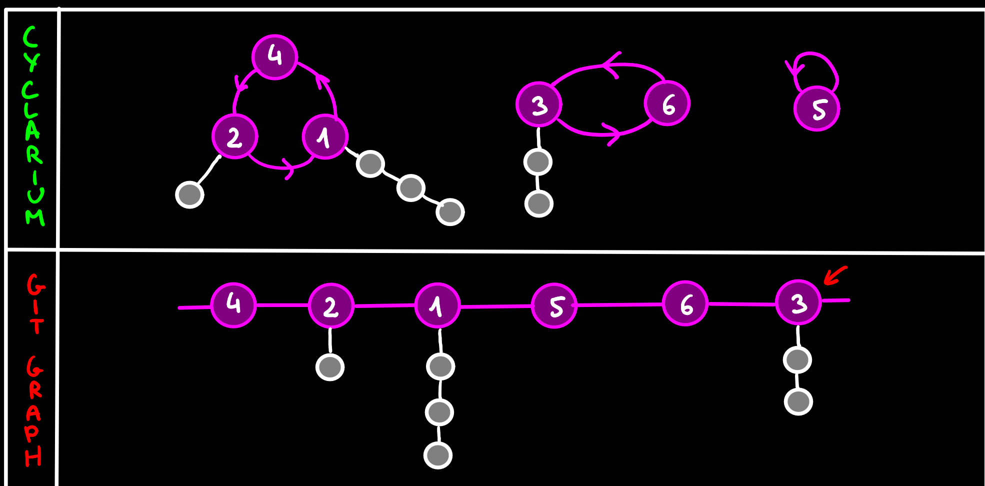
MAX1 < MAX2 < ... < MAXm

3. Right to left :

3a.



3b. Label removing + shift



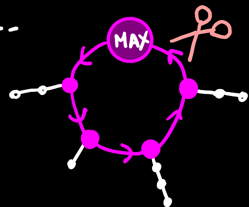
BIJECTION

Proposition

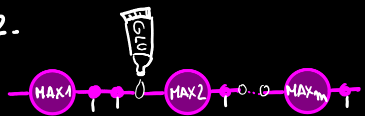
There is a bijection from cyclariums to Git graphs:

STEPS

1.



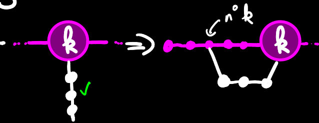
2.



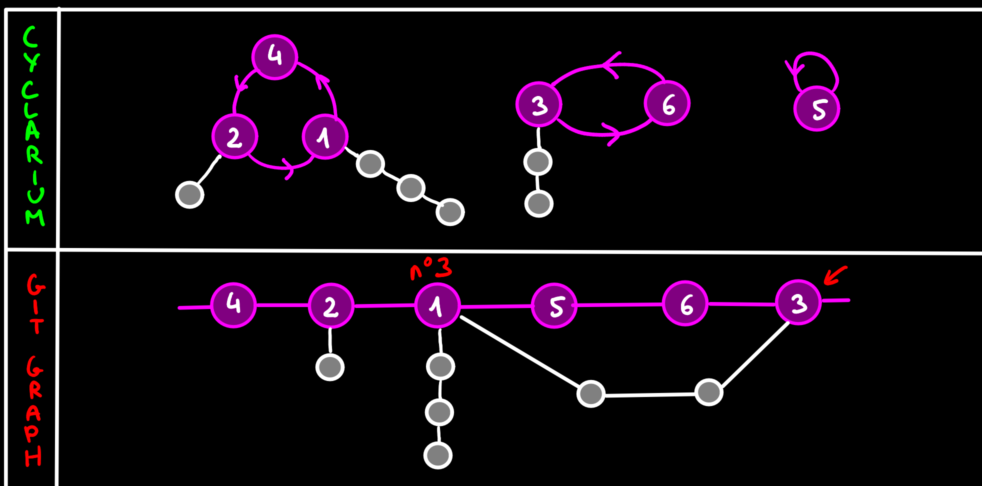
$MAX1 < MAX2 < \ldots < MAXm$

3. Right to left:

3a.



3b. Label removing + shift



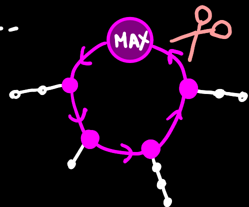
BIJECTION

Proposition

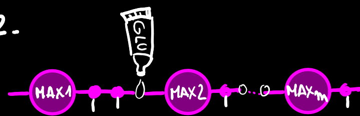
There is a bijection from cyclariums to Git graphs:

STEPS

1.



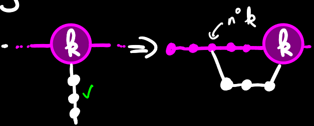
2.



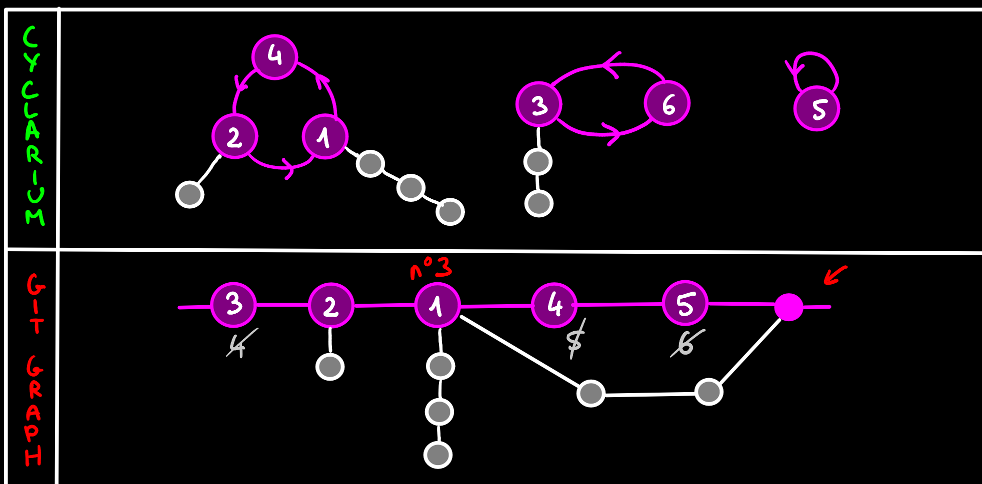
$MAX1 < MAX2 < \ldots < MAXm$

3. Right to left:

3a.



3b. Label removing + shift



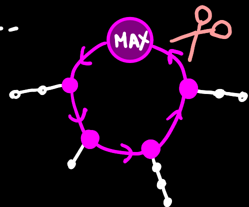
BIJECTION

Proposition

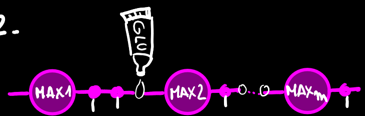
There is a bijection from cyclariums to Git graphs:

STEPS

1.



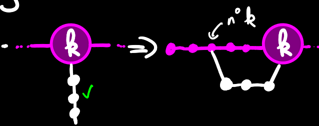
2.



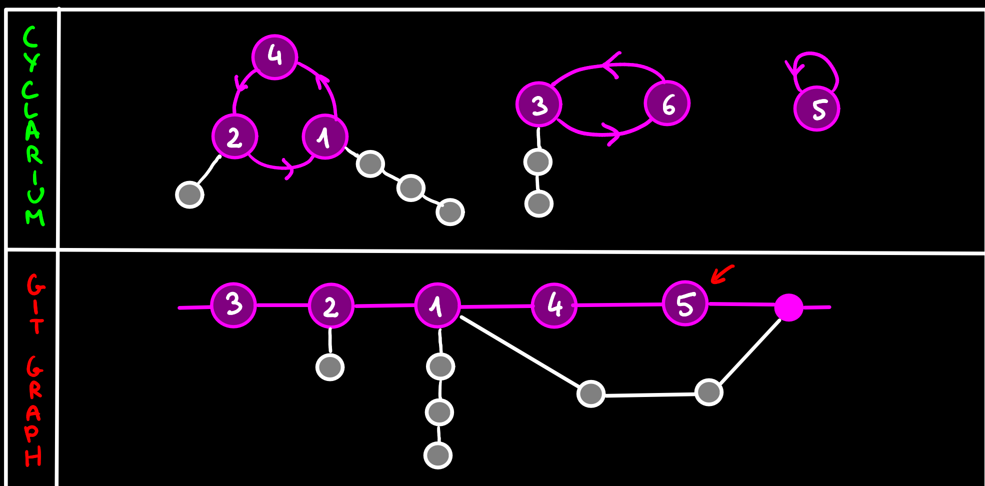
MAX1 < MAX2 < … < MAXm

3. Right to left:

3a.



3b. Label removing + shift



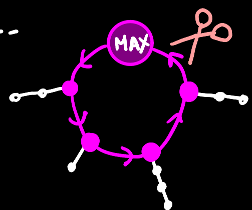
BIJECTION

Proposition

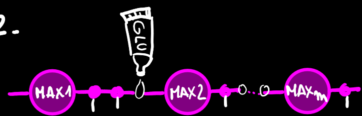
There is a bijection from cyclariums to Git graphs:

STEPS

1.



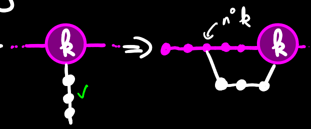
2.



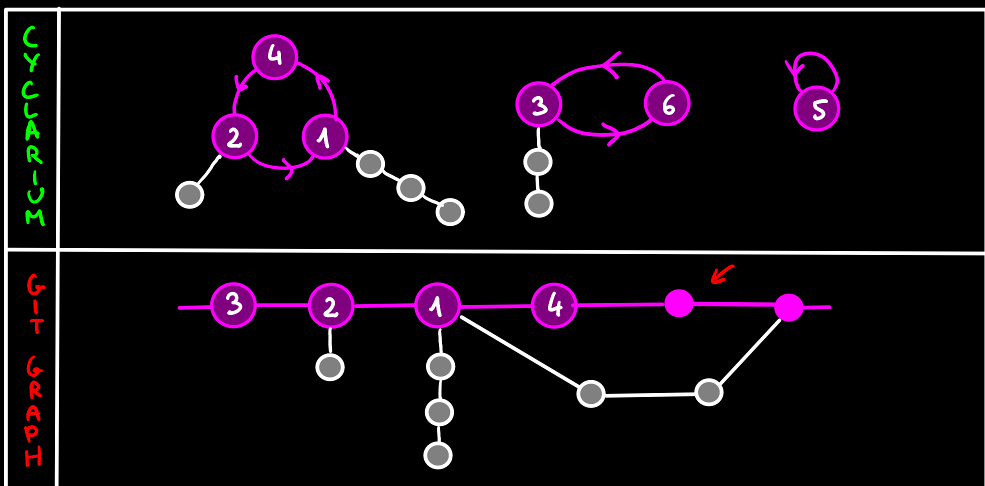
$MAX1 < MAX2 < \dots < MAXm$

3. Right to left:

3a.



3b. Label removing + shift



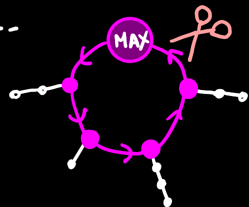
BIJECTION

Proposition

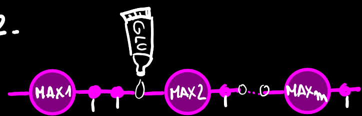
There is a bijection from cyclariums to Git graphs:

STEPS

1.



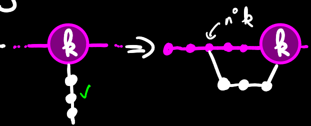
2.



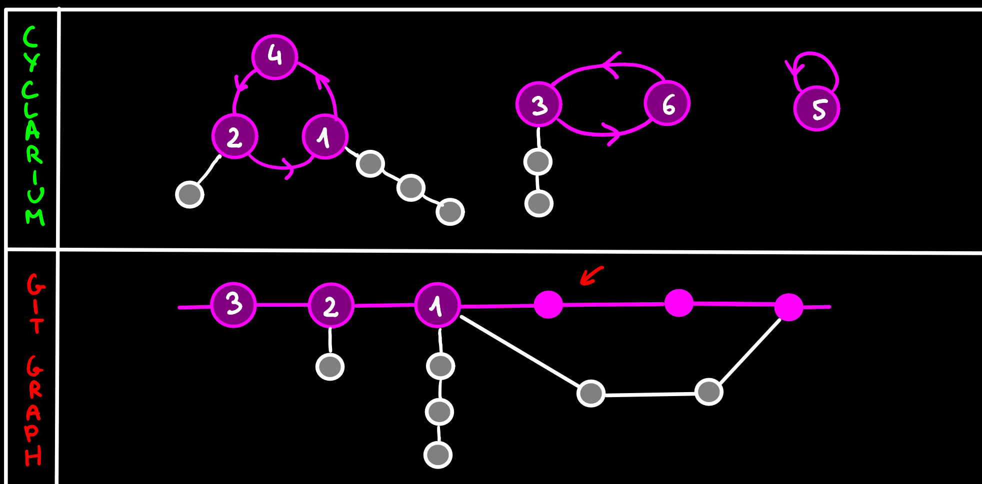
MAX1 < MAX2 < … < MAXm

3. Right to left:

3a.



3b. Label removing + shift



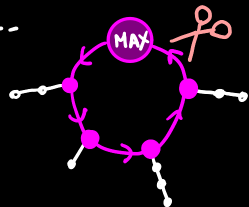
BIJECTION

Proposition

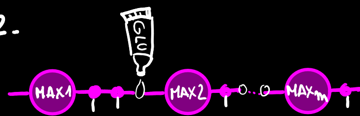
There is a bijection from cyclariums to Git graphs:

STEPS

1.



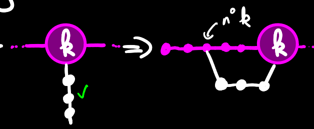
2.



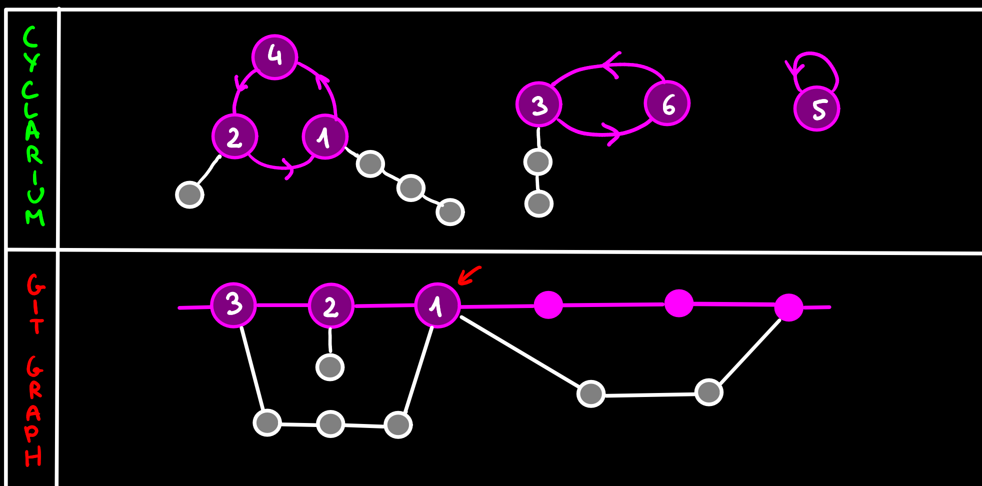
$MAX1 < MAX2 < \ldots < MAXm$

3. Right to left:

3a.



3b. Label removing + shift



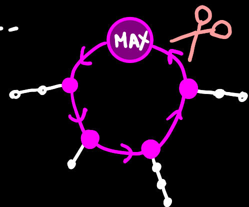
BIJECTION

Proposition

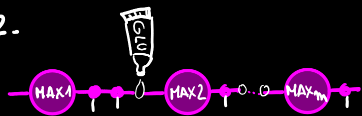
There is a bijection from cyclariums to Git graphs:

STEPS

1.



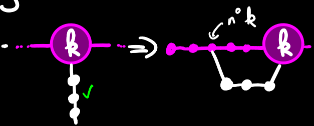
2.



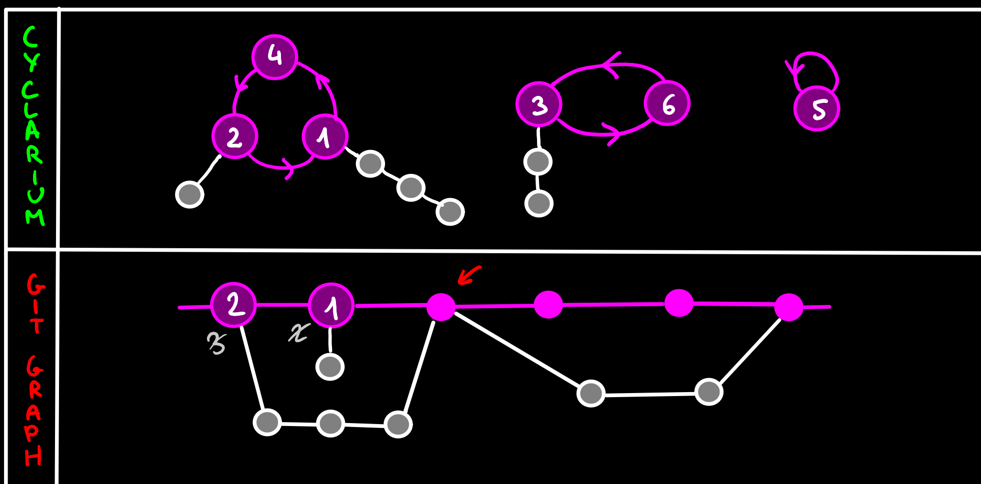
$\mathrm{MAX1} < \mathrm{MAX2} < \ldots < \mathrm{MAXm}$

3. Right to left:

3a.



3b. Label removing + shift



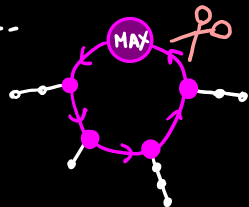
BIJECTION

Proposition

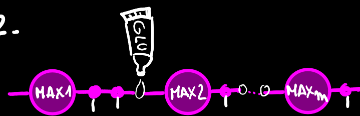
There is a bijection from cyclariums to Git graphs:

STEPS

1.



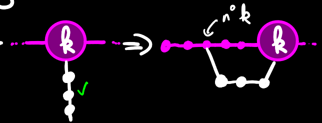
2.



$MAX1 < MAX2 < \dots < MAXm$

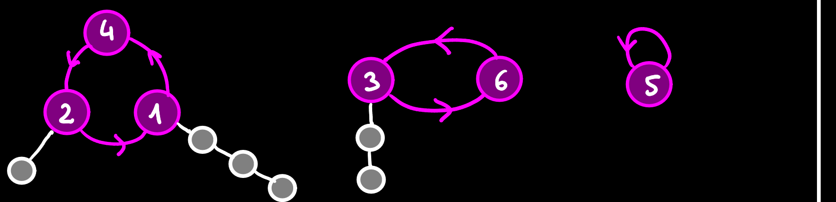
3. Right to left:

3a.

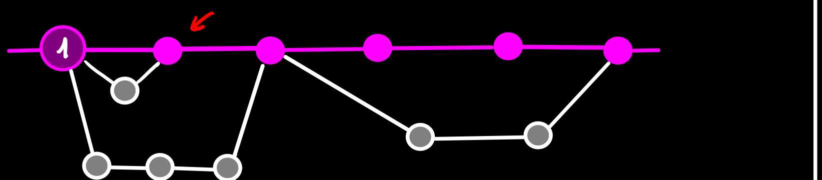


3b. Label removing + shift

CYCLARIUM



GIT GRAPH



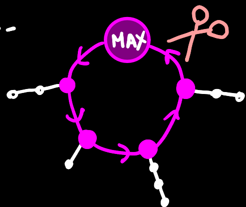
BIJECTION

Proposition

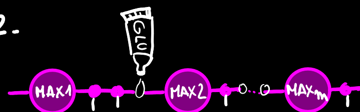
There is a bijection from cyclariums to Git graphs:

STEPS

1.



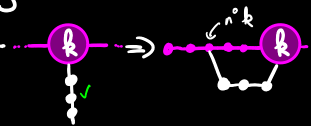
2.



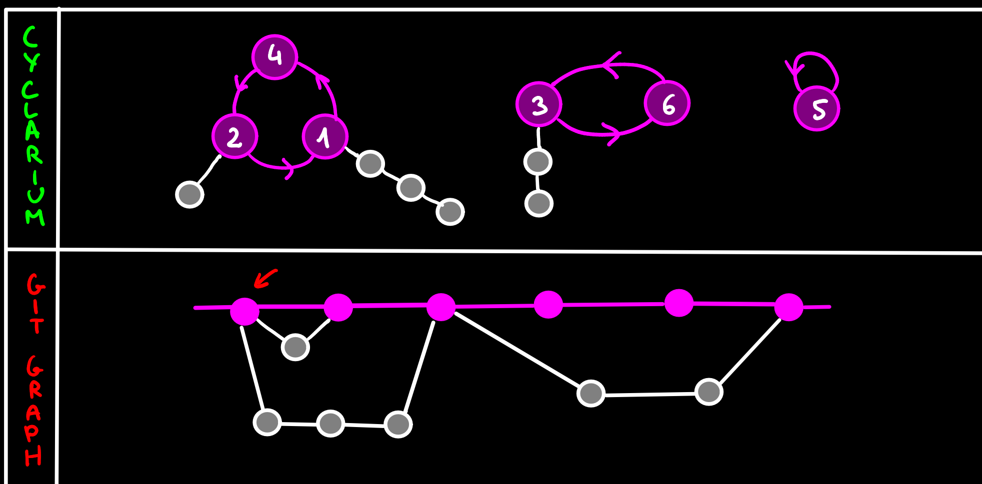
MAX1 < MAX2 < ... < MAXm

3. Right to left:

3a.



3b. Label removing + shift



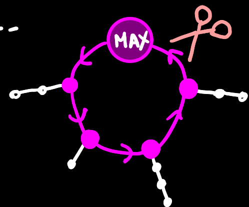
BIJECTION

Proposition

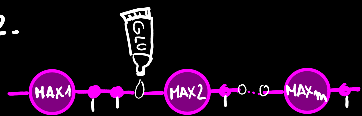
There is a bijection from cyclariums to Git graphs:

STEPS

1.



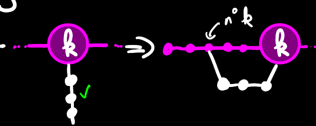
2.



MAX1 < MAX2 < … < MAXm

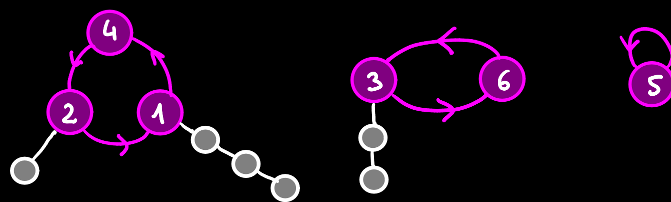
3. Right to left:

3a.

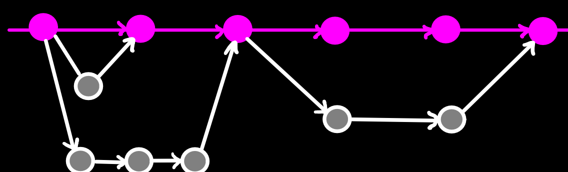


3b. Label removing + shift

CYCLARIUM



GIT GRAPH



BIJECTION

Proposition

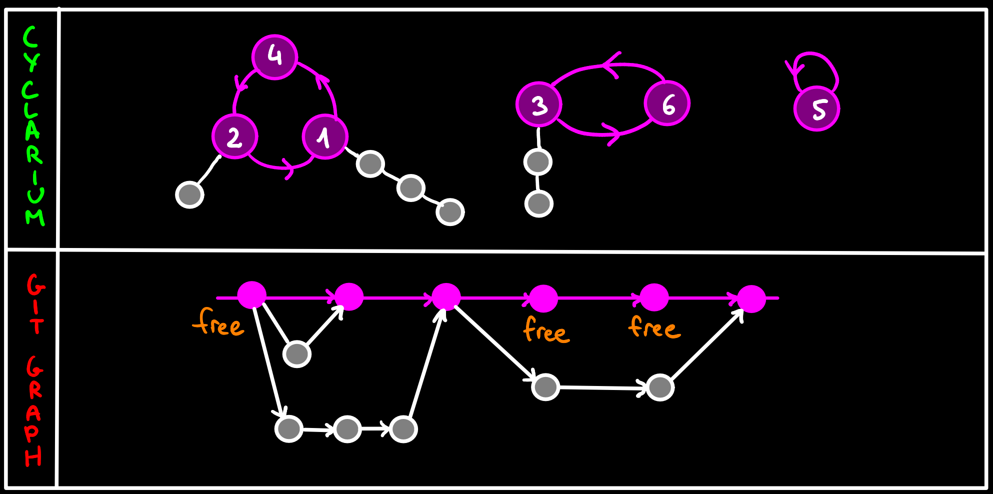
There is a bijection from cyclariums to Git graphs:
sending

vertices $\longrightarrow$ vertices

magenta vertices $\longrightarrow$ magenta vertices

cycles $\longrightarrow$ free vertices
i.e magenta vertices of indegree ≤ 1

cycle lengths $\longrightarrow$ gaps between free vertices



A NICE FORMULA

Proposition

There is a bijection from cyclariums to Git graphs: sending

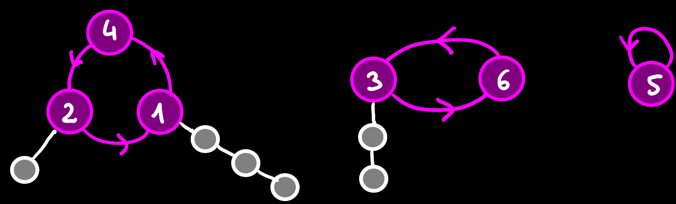
vertices ⟶ vertices

magenta vertices ⟶ magenta vertices

cycles ⟶ free vertices
i.e magenta vertices of indegree ≤ 1

cycle lengths ⟶ gaps between free vertices

CYCLARIUM



Corollary

$$g_{n,k} = \sum_{f=1}^{k-1} \begin{bmatrix} k \\ f \end{bmatrix} \binom{n-k-1}{k-f-1} \qquad (k < n)$$

where $g_{n,k}$ = number of Git graphs counted by vertices & magenta vertices

and $\begin{bmatrix} \cdot \\ \cdot \end{bmatrix}$ = (unsigned) Stirling number of 1st kind

RANDOM MODEL

"Boltzmann model" (exponential in u, ordinary in z)

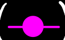
Fix $z > 0^*$ and $u > 0^*$.

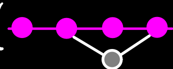
We wish to draw a Git graph γ with a weight proportional to $z^{\#\text{vertices in } \gamma} \dfrac{u^{\#\text{magenta vertices in } \gamma}}{(\#\text{magenta vertices in } \gamma)!}$

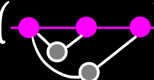
(Size is not fixed)

Examples

$\mathbb{P}($  $) \propto 1$

$\mathbb{P}($  $) \propto z u$

$\mathbb{P}($  $) \propto z^5 \dfrac{u^4}{24}$

$\mathbb{P}($  $) \propto z^5 \dfrac{u^3}{6}$

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

RANDOM MODEL

"Boltzmann model" (exponential in u, ordinary in z)

Fix $z > 0$* and $u > 0$*.

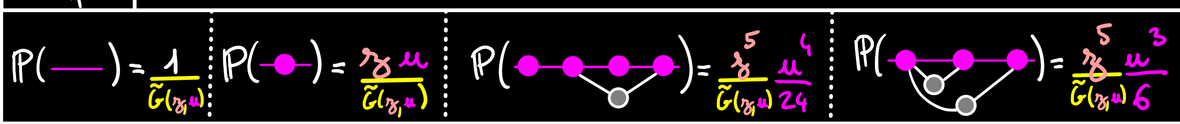
We wish to draw a Git graph γ with a weight ~~proportional~~ equal to

$$\frac{z^{\#\text{vertices in } \gamma}}{\tilde{G}(z,u)} \; \frac{u^{\#\text{magenta vertices in } \gamma}}{(\#\text{magenta vertices in } \gamma)!}$$

(Size is not fixed)

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

Examples



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

RANDOM MODEL

Proposition

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

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

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

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

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

RANDOM MODEL

Proposition

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

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

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

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

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

<u>Consequence</u>: A random generator for Git graphs with $\approx n$ vertices and $\simeq k$ magenta vertices $\left(k \leq \frac{n}{2}\right)$

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

2- Tune z so that $z = \rho_u - \dfrac{1-\rho_u}{n}$

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

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 ?

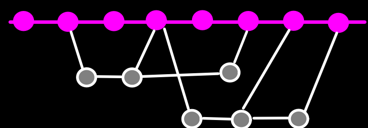
PERSPECTIVES ABOUT RANDOM GIT GRAPHS

→ Asymptotic behaviour

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

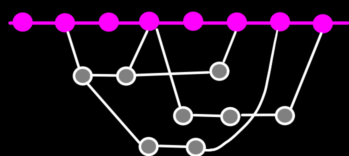
→ Other random models ▷ Collaboration with Clement + Maréchal

Phoenix graphs



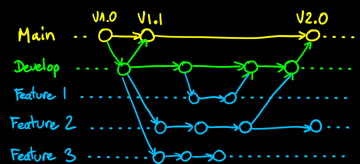
merged branches can be reborn

Fork Anywhere graphs



branches can be born anywhere but must be merged into main

More involved workflows



LINKS BETWEEN GIT GRAPHS & GIT BISECT

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

LINKS BETWEEN GIT GRAPHS & GIT BISECT

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

<u>But also</u>: - 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!