Fighting regressions with git bisect

Christian Couder
chriscool@tuxfamily.org

October 29,2009

About Git

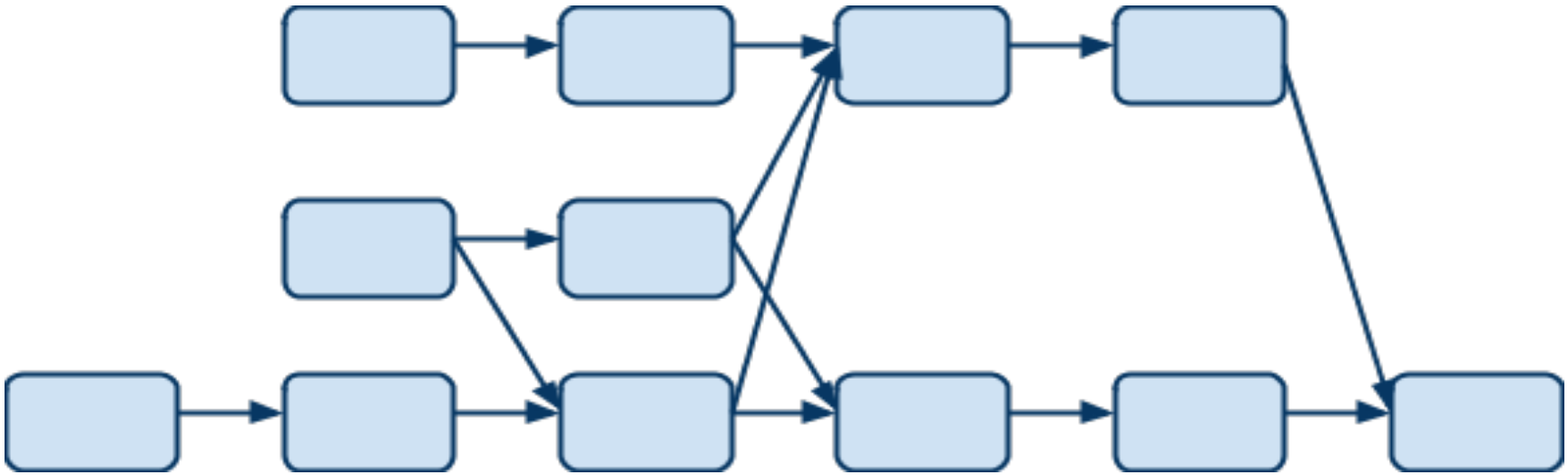
A Distributed Version Control system (DVCS):

- created by Linus Torvalds
- maintained by Junio Hamano

Basics:

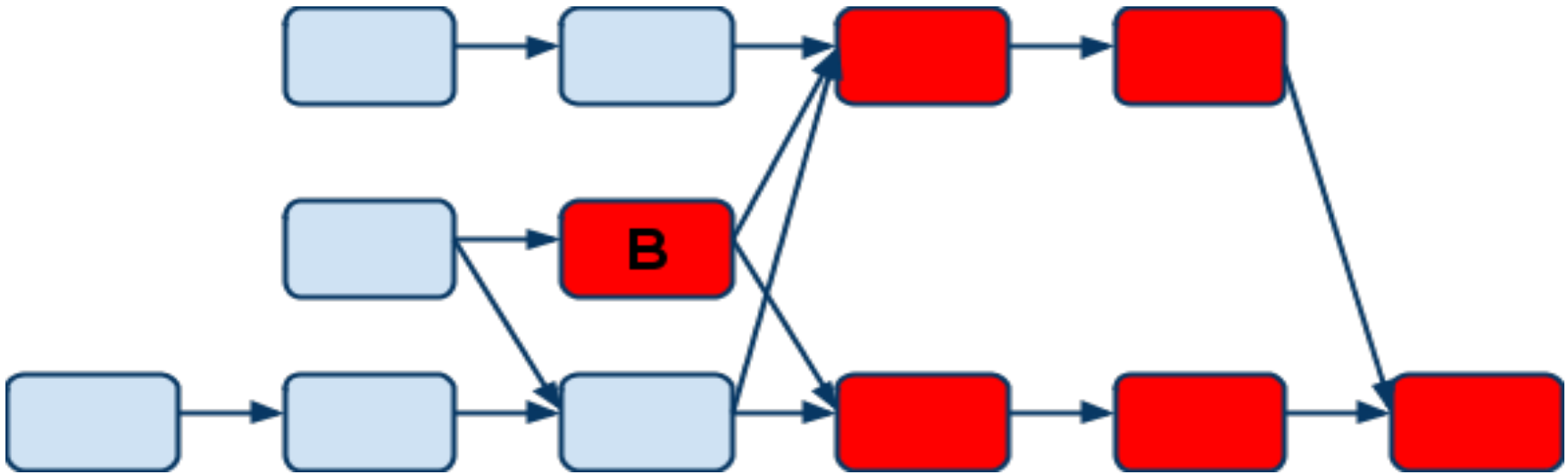
- commits are states of the managed data
- managed data is software source code
- so each commit corresponds to a software behavior

Commits in Git form a DAG (directed acyclic graph)



- DAG direction is from left to right
- older commits point to newer commits

First bad commit



- B introduces a bad behavior called "bug" or "regression"
- B is called a "first bad commit"
- red commits are called "bad"
- blue commits are called "good"

"git bisect"

Idea:

- help find a first bad commit
- use binary search for efficiency if possible

Benefits:

- checking the changes from only one commit is easy
- the commit gives extra information: commit message, author, ...

Regressions: a big problem

Related studies:

- 80% of development costs is identifying and correcting defects (NIST 2002),
- 80% of the lifetime cost of a piece of software goes to maintenance (Sun in Java code conventions),
- over 80%, of the maintenance effort is used for non-corrective actions (Pigosky 1997, cited by Wikipedia).

So either:

- at least one study is completely wrong,
- or there is an underlying fact.

We guess that regressions make it very difficult to improve on existing software.

Linux kernel example

Regression is an important problem because:

- big code base growing fast
- many different developers
- developed and maintained for many years
- many users depending on it

Development process:

- 2 weeks "merge window"
- 8 or 9 "rc" releases to fix bugs, especially regressions, around 1 week apart
- release 2.6.X
- stable releases 2.6.X.Y and distribution maintenance

Ingo Molnar about his "git bisect" use

I most actively use it during the merge window (when a lot of trees get merged upstream and when the influx of bugs is the highest) - and yes, there have been cases that i used it multiple times a day. My average is roughly once a day.

=> regressions are fought all the time

Indeed it is well known that is is more efficient (and less costly) to fix bugs as soon as possible.

Other tools to fight regressions

The NIST study found that more than a third of the costs "*could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects*".

Other tools:

- some are the same as for regular bugs
- test suites
- tools similar as git bisect

Test suites

Very useful

- to prevent regressions,
- to ensure an amount of functionality and testability.

But inefficient

- when using them to check each commit backward,
- when testing each commit because of combinational explosion.

N configurations, M commits, T tests means:

N * M * T tests to perform

Starting a bisection and bounding it

2 ways to do it:

$ git bisect start
$ git bisect bad [COMMIT]
$ git bisect good [COMMIT...]

or

$ git bisect start BAD GOOD [GOOD...]

where COMMIT, BAD and GOOD can be resolved to a commit

Starting example

(toy example with the linux kernel)

$ git bisect start v2.6.27 v2.6.25
Bisecting: 10928 revisions left to test after this (roughly 14 steps)
[2ec65f8b89ea003c27ff7723525a2ee335a2b393] x86: clean up using max_low_pfn on 32-bit
$

=> the commit you should test has been checked out

Driving a bisection manually

1. test the current commit
2. tell "git bisect" whether it is good or bad, for example:

$ git bisect bad
Bisecting: 5480 revisions left to test after this (roughly 13 steps)
[66c0b394f08fd89236515c1c84485ea712a157be] KVM: kill file->f_count abuse in kvm

repeat step 1. and 2. until the first bad commit is found...

First bad commit found

$ git bisect bad
2ddcca36c8bcfa251724fe342c8327451988be0d is the first bad commit
commit 2ddcca36c8bcfa251724fe342c8327451988be0d
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date: Sat May 3 11:59:44 2008 -0700

Linux 2.6.26-rc1

:100644 100644 5cf8258195331a4dbdddff08b8d68642638eea57 4492984efc09ab72ff6219a7bc21fb6a957c4cd5 M Makefile

End of bisection

When the first bad commit is found:

- you can check it out and tinker with it, or
- you can use "git bisect reset", like that:

$ git bisect reset
Checking out files: 100% (21549/21549), done.
Previous HEAD position was 2ddcca3... Linux 2.6.26-rc1
Switched to branch 'master'

to go back to the branch you were in before you started bisecting

Driving a bisection automatically

At each bisection step a script or command will be launched to tell if the current commit is good or bad.

Syntax:

$ git bisect run COMMAND [ARG...]

Example to bisect a broken build:
$ git bisect run make

Automatic bisect example part 1

$ git bisect start v2.6.27 v2.6.25
Bisecting: 10928 revisions left to test after this (roughly 14 steps)
[2ec65f8b89ea003c27ff7723525a2ee335a2b393] x86: clean up using max_low_pfn on 32-bit
$
$ git bisect run grep '^SUBLEVEL = 25' Makefile
running grep ^SUBLEVEL = 25 Makefile
Bisecting: 5480 revisions left to test after this (roughly 13 steps)
[66c0b394f08fd89236515c1c84485ea712a157be] KVM: kill file->f_count abuse in kvm
running grep ^SUBLEVEL = 25 Makefile

Automatic bisect example part 2

SUBLEVEL = 25
Bisecting: 2740 revisions left to test after this (roughly 12 steps)
[671294719628f1671faefd4882764886f8ad08cb] V4L/DVB(7879): Adding cx18 Support for mxl5005s
...
...
running grep ^SUBLEVEL = 25 Makefile
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[2ddcca36c8bcfa251724fe342c8327451988be0d] Linux 2.6.26-rc1
running grep ^SUBLEVEL = 25 Makefile

Automatic bisect example part 3

2ddcca36c8bcfa251724fe342c8327451988be0d is the first bad commit
commit 2ddcca36c8bcfa251724fe342c8327451988be0d
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date: Sat May 3 11:59:44 2008 -0700

Linux 2.6.26-rc1

:100644 100644 5cf8258195331a4dbdddff08b8d68642638eea57 4492984efc09ab72ff6219a7bc21fb6a957c4cd5 M Makefile
bisect run success

Run script exit codes

0 => good
1-124 and 126-127 => bad
128-255 => "stop": bisection is stopped immediately
125 => "skip": mark commit as "untestable"

"stop" is useful to abort bisection in abnormal situations

"skip" means "git bisect" will choose another commit to be tested

Untestable commits

Manual bisection choice:

- "git bisect visualize" or "git bisect view": gitk or "git log" to help you find a better commit to test
- "git bisect skip"

Possible situation with skipped commits

Possible end of bisection

There are only 'skip'ped commits left to test.
The first bad commit could be any of:
15722f2fa328eaba97022898a305ffc8172db6b1
78e86cf3e850bd755bb71831f42e200626fbd1e0
e15b73ad3db9b48d7d1ade32f8cd23a751fe0ace
070eab2303024706f2924822bfec8b9847e4ac1b
We cannot bisect more!

Saving a log and replaying it

Saving:

$ git bisect log > bisect_log.txt

Replaying:

$ git bisect replay bisect_log.txt

Bisection algorithm

It gives the commit that will be tested.

So the goal is to find the best bisection commit.

The algorithm currently used

- is "truly stupid" (Linus Torvalds)
- but works quite well in practice

We suppose that there are no skip'ped commits.

Bisection algorithm, step 0

If a commit was just tested, then it can be marked as either:

- good, in this case we have one more good commits, or
- bad, in this case it becomes the bad commit, the previous bad commit is not considered as bad anymore.

The algorithm is not symmetric, it uses only one current bad commit and many good commits.

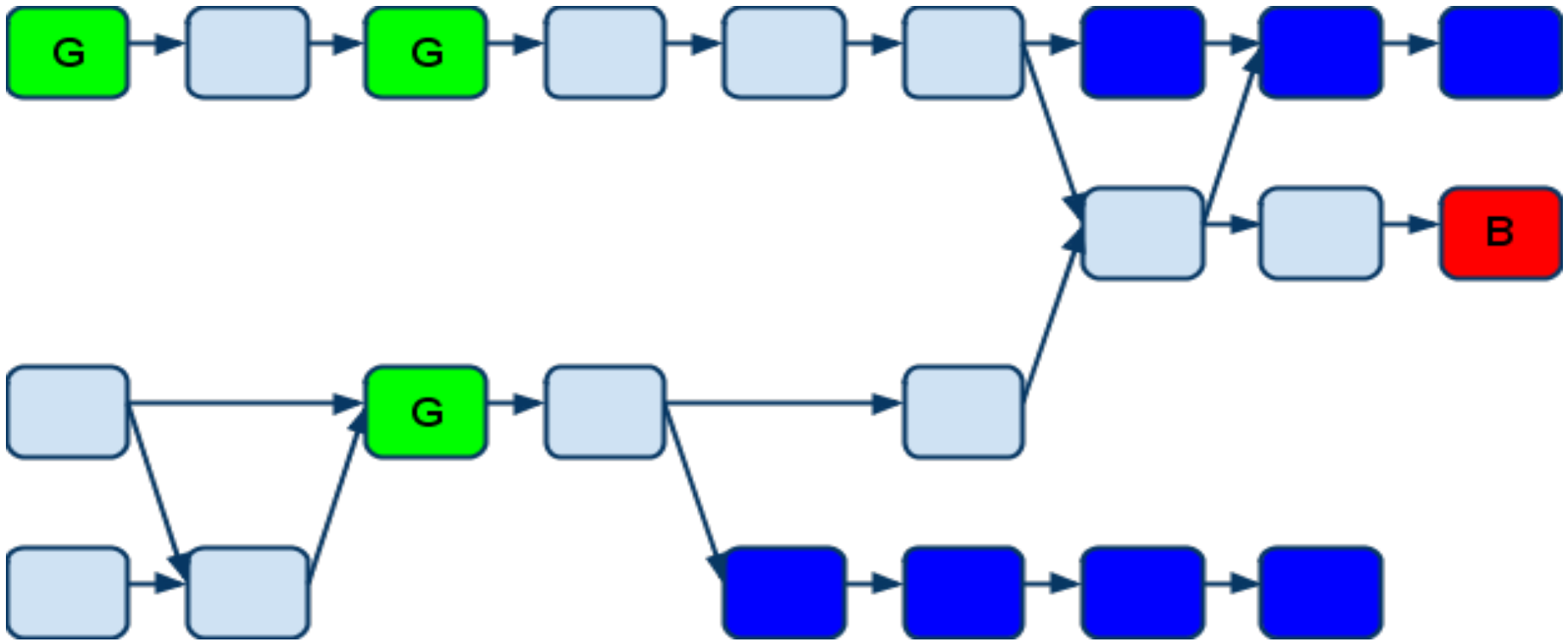
Bisection algorithm, step 1

We want a cleaned up commit graph with only "interesting" commits.

Keep only the commits that:

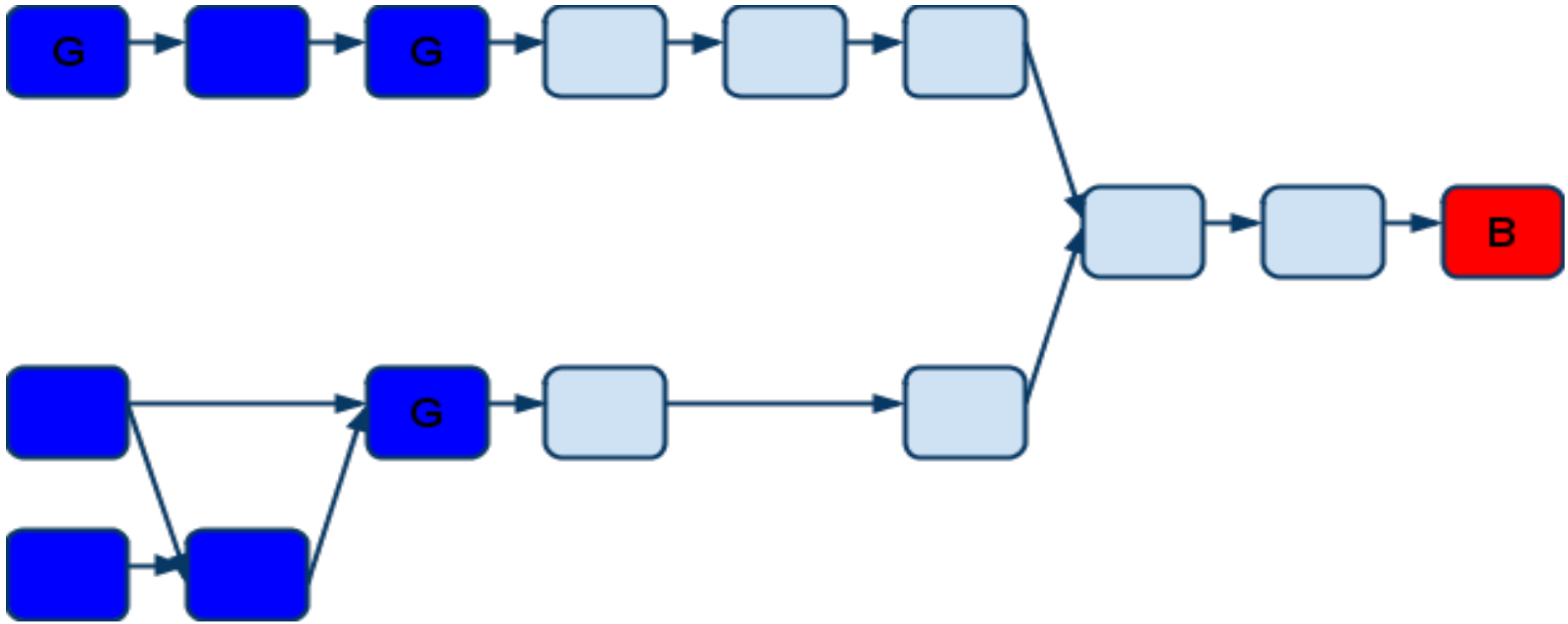
1. are ancestor of the "bad" commit (including the "bad" commit itself),
2. are not ancestor of a "good" commit, (excluding the "good" commits).

Bisection algorithm, step 1.1



1.1 Keep ancestors of the "bad" commit

Bisection algorithm, step 1.2



1.2 Keep commits that are not ancestor of a "good" commit, excluding good commits

Bisection algorithm, step 1

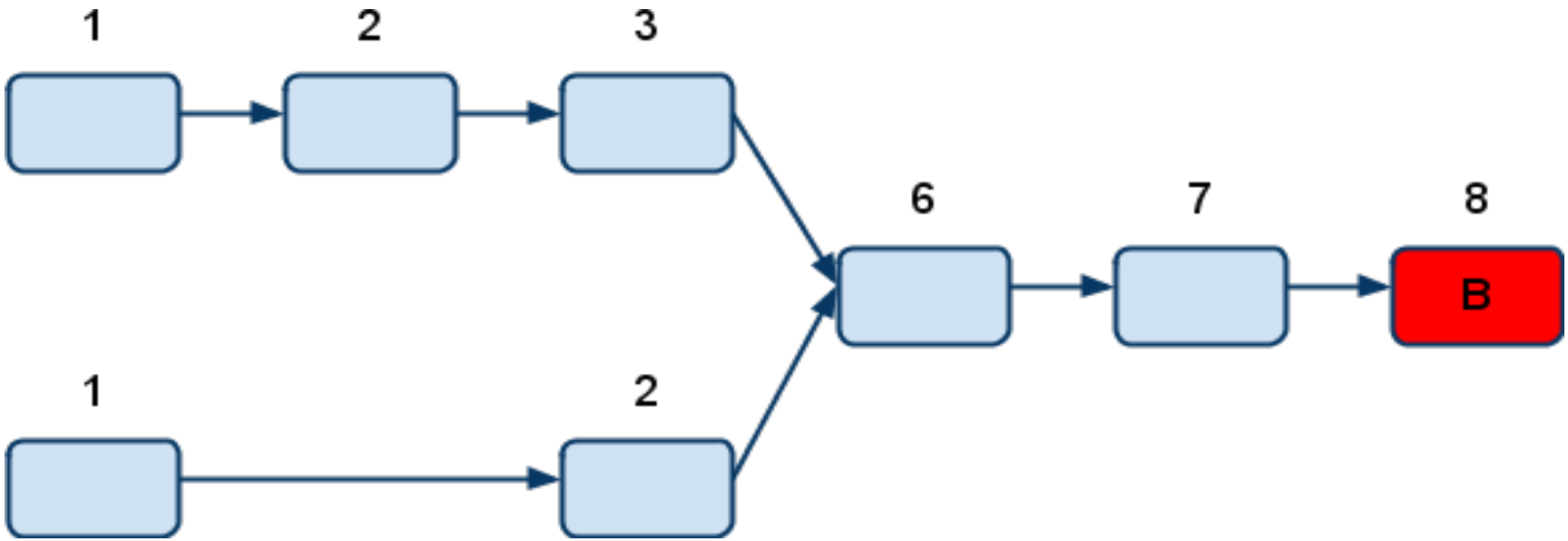
So we keep only ancestors of the bad commit that are not ancestors of the good commits.

That is we keep commits given by:

$ git rev-list BAD --not GOOD1 GOOD2...

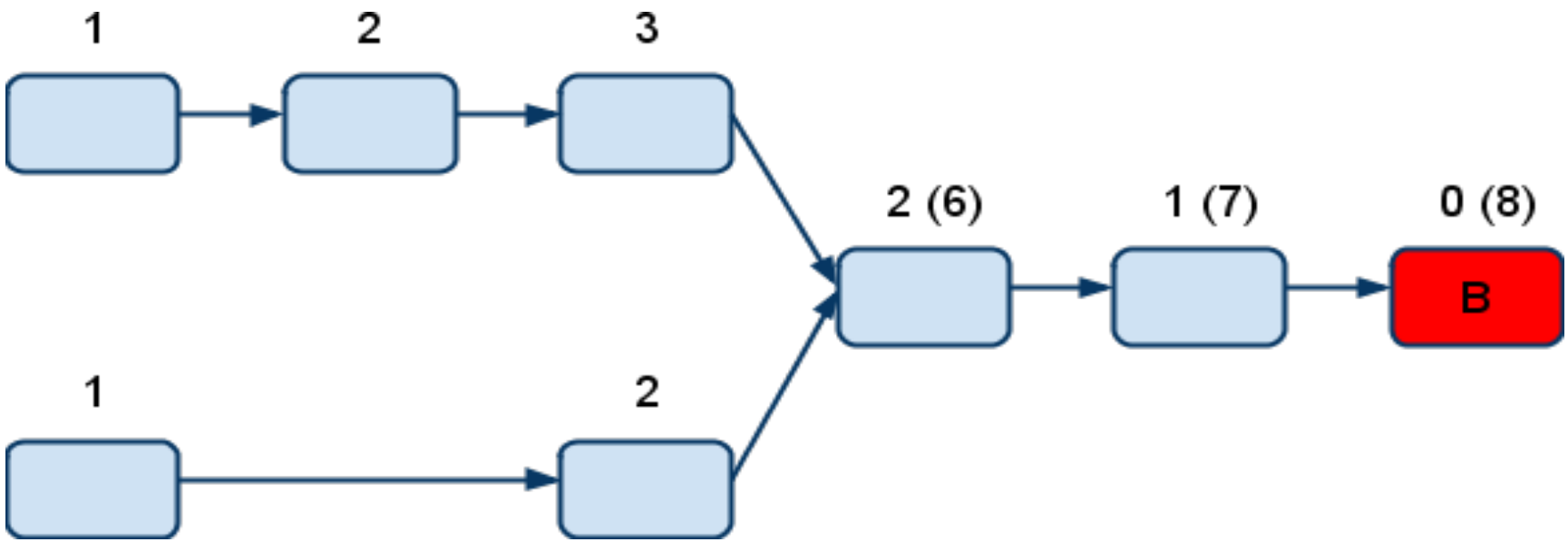
Bisection algorithm, step 2

Associate to each commit the number of ancestors it has plus one.



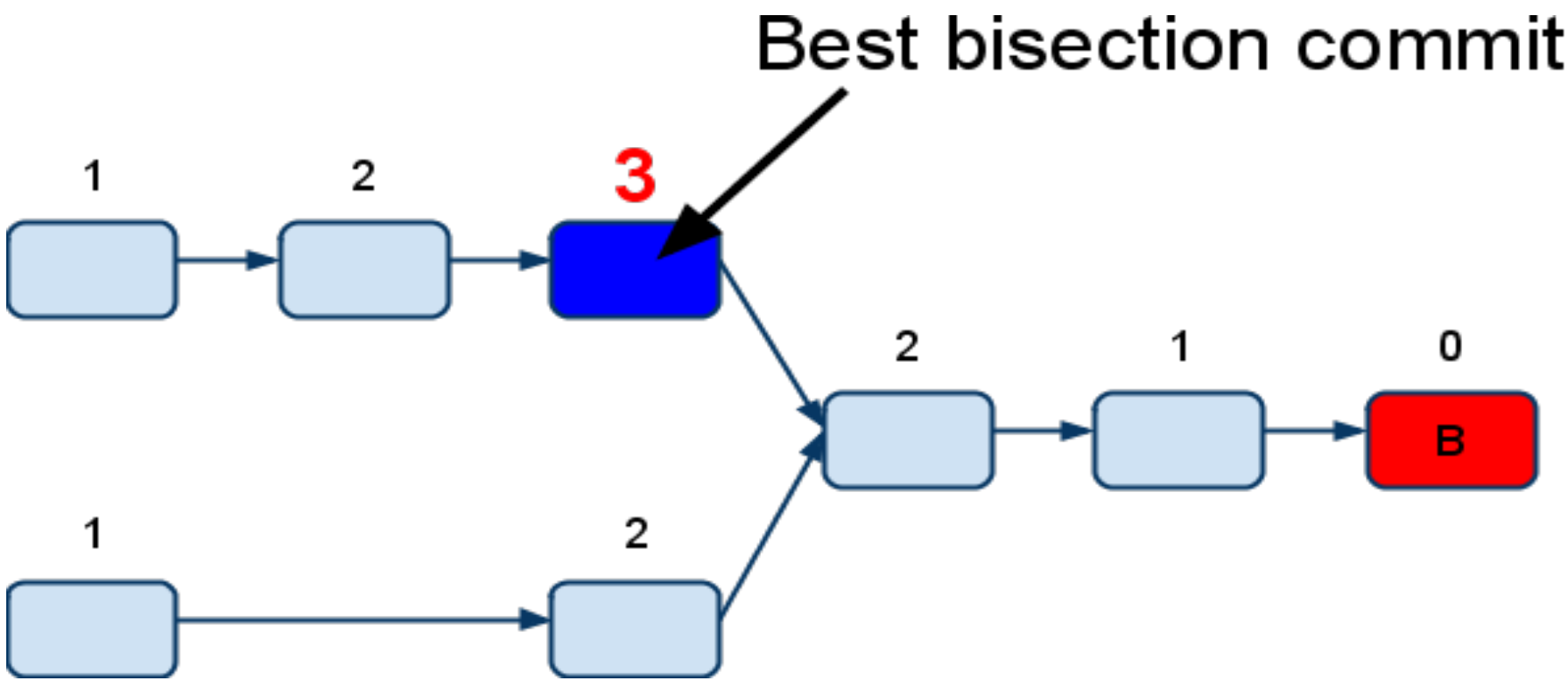
Bisection algorithm, step 3

Associate to each commit min(X, N - X), where X is the value associated in step 2, and N is the total number of commits.



Bisection algorithm, step 4

The best bisection commit is the commit with the highest associated value.



Bisection algorithm, shortcuts

We know N the number of commits in the graph from the beginning (after step 1).

So if we associate N/2 to any commit during step 2 or 3, then we know we can use this commit as the best bisection commit.

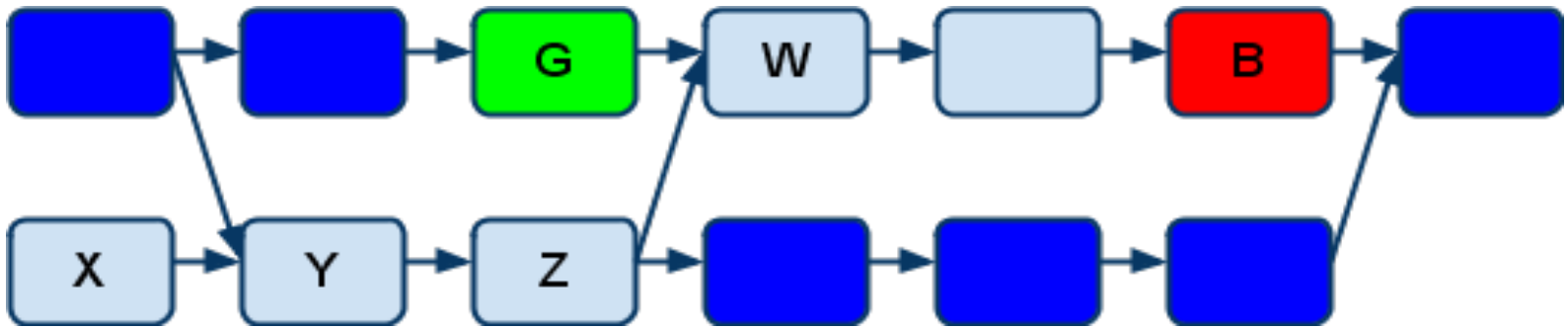
Bisection algorithm, debugging

To show values associated with commits:

$ git rev-list --bisect-all BAD --not GOOD1 GOOD2

For example:
e15b73ad3db9b48d7d1ade32f8cd23a751fe0ace (dist=3)
15722f2fa328eaba97022898a305ffc8172db6b1 (dist=2)
78e86cf3e850bd755bb71831f42e200626fbd1e0 (dist=2)
a1939d9a142de972094af4dde9a544e577ddef0e (dist=2)
070eab2303024706f2924822bfec8b9847e4ac1b (dist=1)
a3864d4f32a3bf5ed177ddef598490a08760b70d (dist=1)
a41baa717dd74f1180abf55e9341bc7a0bb9d556 (dist=1)
9e622a6dad403b71c40979743bb9d5be17b16bd6 (dist=0)

Bisection algorithm, pitfalls



Commits X, Y and Z are not removed from the graph we are bisecting on.

So you may have to test kernels with version 2.6.25 even if you are bisecting between v2.6.26 and v2.6.27!

Or you may be on a branch with only the Btrfs driver code!

Bisection algorithm, discussion

We want the commit X so that we get as much information as possible whether X happens to be good or bad.
So we want to maximize:

f(X) = min(info_if_good(X), info_if_bad(X))

where **info_if_good(X)** is the information we get if X is good and **info_if_bad(X)** is the information we get if X is bad.

Bisection algorithm, discussion

If commit X is good, then ancestors of X are good.

So we want to say:

info_if_good(X) = number_of_ancestors(X)

And this is true. (See step 1.2.)

(This suppose that commits have an equal chance of being good or bad.)

Bisection algorithm, discussion

And if commit X is bad,then descendants of X are bad, so we want to say:

info_if_bad(X) = number_of_descendants(X)

But this is WRONG!

We get more information than that, when a commit happens to be bad, because it replaces the previous bad commit in step 0, so commits that are not any more ancestors of the bad commit are discarded in step 1.1.

Bisection algorithm, discussion

During step 2, we compute:

info_if_good(X) = number_of_ancestors(X)

and during step 3 we compute:

info_if_bad(X) = N - number_of_ancestors(X)

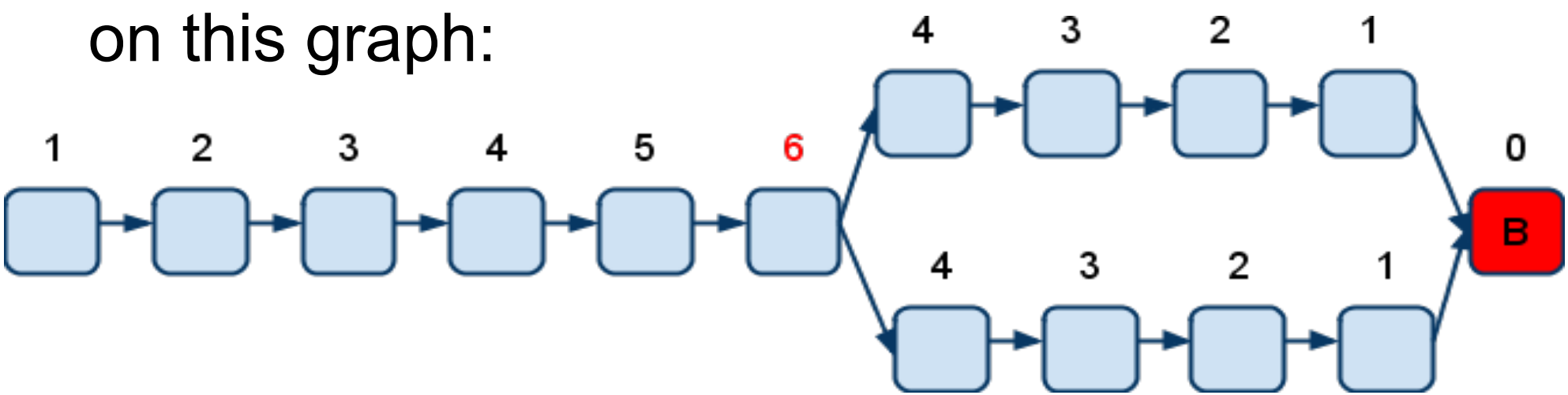
and this is right because in step 1.1 we discard commits that are not ancestors of the new bad commit.

Bisection algorithm, discussion

For example let's compute:

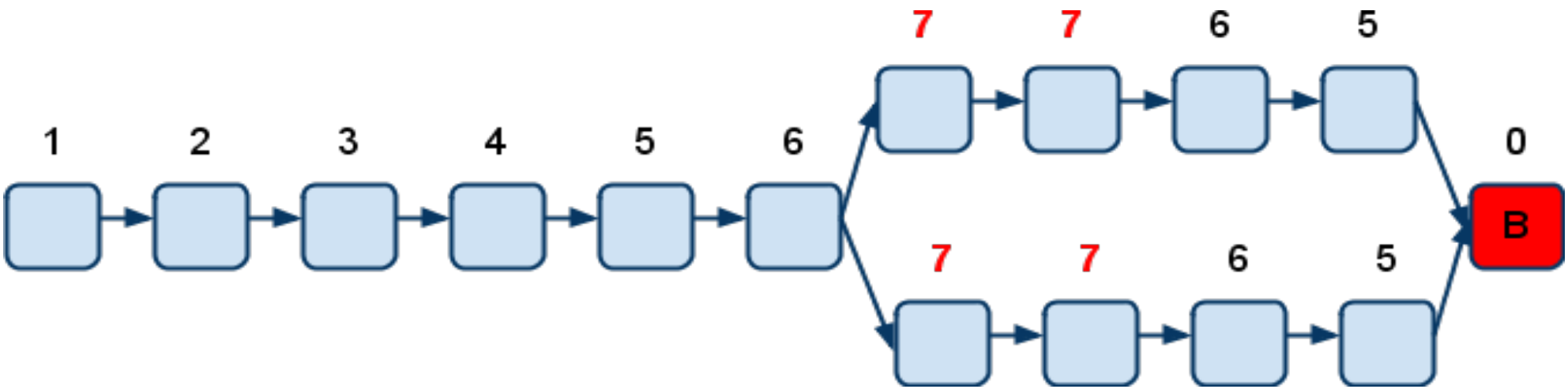
min(nb_ancestors(X), nb_descendants (X))

on this graph:



Bisection algorithm, discussion

With the current algorithm we get:



which is better!

Skip algorithm

When there are skip'ped commits step 0 to 3 are the same as the bisection algorithm. So there is a value associated with each commit. But no shortcut is taken.

What we do (after step 3) is first:

- sort commits by decreasing associated value
- if the first commit has not been skip'ped we return it and stop there
- otherwise we filter out all the skip'ped commits from the sorted list

Skip algorithm

Then we use a pseudo random number generator (PRNG) to generate a random integer R in [0, RND_MAX[.

And we return the commit that is at index:

$$\mathbf{i = N * (R / RND\_MAX)^{3/2}}$$

where N is the number of commits in the sorted list.

We use the power 3/2 to favor commits near the beginning of the list.

Skip algorithm, discussion

Previously we just returned the first non skip'ped commit.
The new algorithm is used only since version 1.6.4 (July 2009).

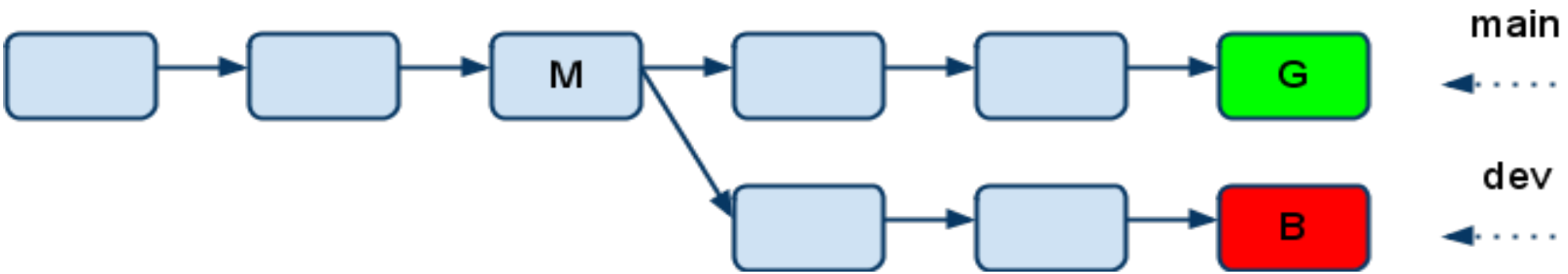
Reasons:

- Untestable commits are often created by a breakage.
- There can be very long strings of untestable commits.
- Commits with the highest associated values are also often near each other.

Checking merge bases

It is not a requirement that good commits be ancestors of the bad commit.

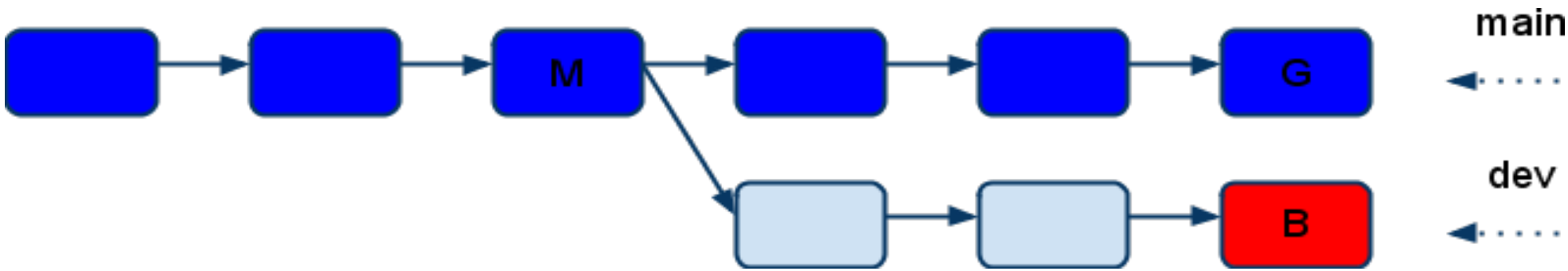
For example:



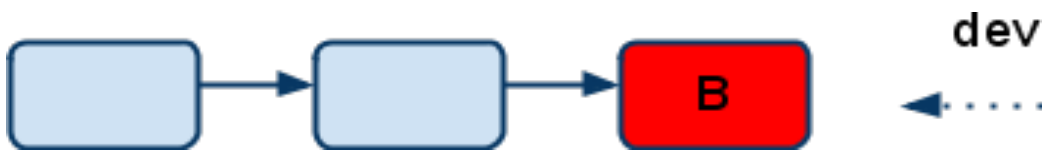
M is a "merge base" for branches "main" and "dev"

Checking merge bases

If we apply the bisection algorithm, we must remove all ancestors of the good commits.

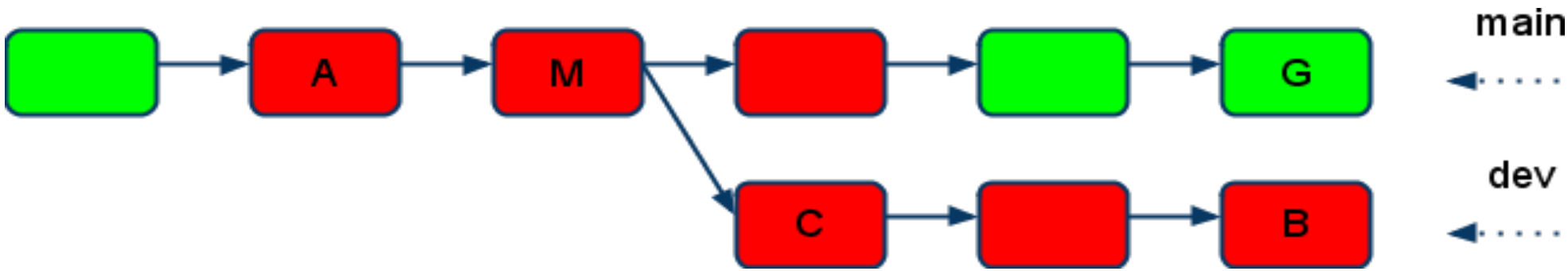


So we get only:



Checking merge bases

But what if the bug was fixed in the "main" branch?



We would find C as the first bad commit instead of A, so we would be wrong!

Checking merge bases

Solution:

- compute merge bases if a good commit is not ancestor of the bad commit
- ask the user to test merge bases first
- if a merge base is not good, stop!

For example:

The merge base BBBBBB is bad.
This means the bug has been fixed between BBBBBB and [GGGGGG,...].

Best practices: "git bisect run" tips

Bisect broken builds:
$ git bisect run make

Bisect broken test suite:

$ git bisect run make test

Bisect run with "sh -c":
$ git bisect run sh -c "make || exit 125; ./my_app | grep 'good output'"

Best practices: performance regressions

```
#!/bin/sh

make my_app || exit 255 # Stop if build fails
./my_app >log 2>&1 & # Launch app in background
pid=$! # Grab its process ID
sleep $NORMAL_TIME # Wait for sufficiently long

if kill -0 $pid # See if app is still there
then # It is still running -- that is bad.
    kill $pid; sleep 1; kill $pid; exit 1
else # It has already finished, we are happy.
    exit 0
fi
```

Best practices: complex scripts

It can be worth it. For example Ingo Molnar wrote:

i have a fully automated bootup-hang bisection script. It is based on "git-bisect run". I run the script, it builds and boots kernels fully automatically, and when the bootup fails (the script notices that via the serial log, which it continuously watches - or via a timeout, if the system does not come up within 10 minutes it's a "bad" kernel), the script raises my attention via a beep and i power cycle the test box. (yeah, i should make use of a managed power outlet to 100% automate it)

Best practices: general best practices

These practices are useful without "git bisect", but they are even more useful when using "git bisect":

- no commits that break things, even if other commits later fix the breakage,
- only one small logical change in each commit,
- small commits for easy review,
- good commits messages.

Best practices: no bug prone merge

Merges by themselves can introduce regressions when there is no conflict to resolve.

Example:

- semantic of a function change in one branch,
- a call to the function is added in another branch.

This is made much worse when there are many changes, either needed to fix conflicts or for any other reason, in a merge. When changes are not related to the branches, they are called "evil merges" and should be avoided.

Best practices: no bug prone merge

So what can be done:

- "git rebase" can linearize history, instead of merging, or to bisect a merge
- use shorter branches, or many topic branches
- use integration branches, like "linux-next", to prepare merges and to test

Best practices: test suites and git bisect

Using "git bisect run", it's very easy to find the commit that broke a test suite.

Virtuous cycle:

more tests in the test suite
=> easy to write one more test
=> easy to use "git bisect run"
=> test written for use with "git bisect run"
=> test added to the test suite
=> more tests in the test suite

Best practices: test suites and git bisect

With T test cases, testing for all N configurations only a few c times between each release, and then bisecting all bugs found means:

c * N * T + b * M * log2(M) tests

where b is the number of bug per commits and M the number of commits.

So we get O(N * T) versus O(M * N * T) if testing everything after each commit.

Best practices: adapting your work-flow

Test suites and "git bisect" are very powerful and efficient when used together.

For example, work-flow used by Andreas Ericsson:

- write, in the test suite, a test to catch a regression
- use "git bisect run" to find the first bad commit
- fix the bug
- commit both the fix and the test script (and if needed more tests)

Best practices: adapting your work-flow

Results reported by Andreas from using Git and adopting this work-flow after one year:

- report-to-fix cycle went from 142.6 hours (wall-clock time) to 16.2 hours,
- each new release results in ~40% fewer bugs ("*almost certainly due to how we now feel about writing tests*").

Best practices: adapting your work-flow

Like the work-flow used by Andreas, a good work-flow should be designed around:

- using general best pratices (small logical commits, good commit messages, topic branches, no evil merge, ...),
- taking advantage of the virtuous cycle between a test suite and "git bisect".

Best practices: involving non developers

No need to be a developer to use "git bisect".

During heated discussions on linux-kernel mailing list around April 2008, David Miller wrote:

What people don't get is that this is a situation where the "end node principle" applies. When you have limited resources (here: developers) you don't push the bulk of the burden upon them. Instead you push things out to the resource you have a lot of, the end nodes (here: users), so that the situation actually scales.

Best practices: involving non developers

Reasons:

- It can be "cheaper" if QA people or end users can do it.
- People reporting a bug have access to the environment where the bug happens, and "git bisect" automatically extract relevant information from this environment.
- For open source project, this is a good way to get new contributors.

Best practices: pluging in other tools

Test suites and "git bisect" can be combined with more tools.

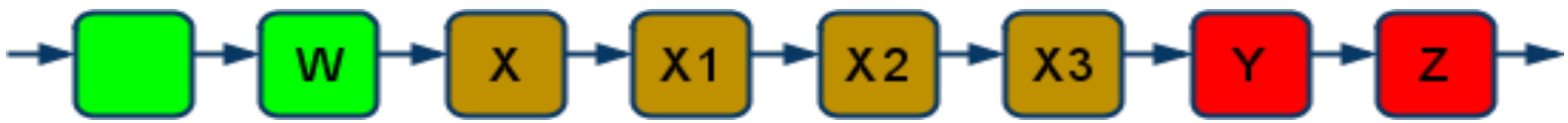
For example after "git bisect", it's possible to automatically:

- send an email to the people involved in the first bad commit, and/or
- create an entry in the bug tracking system.

The future of bisecting: "git replace"

Sometimes the first bad commit will be in an untestable area of the graph.

For example:

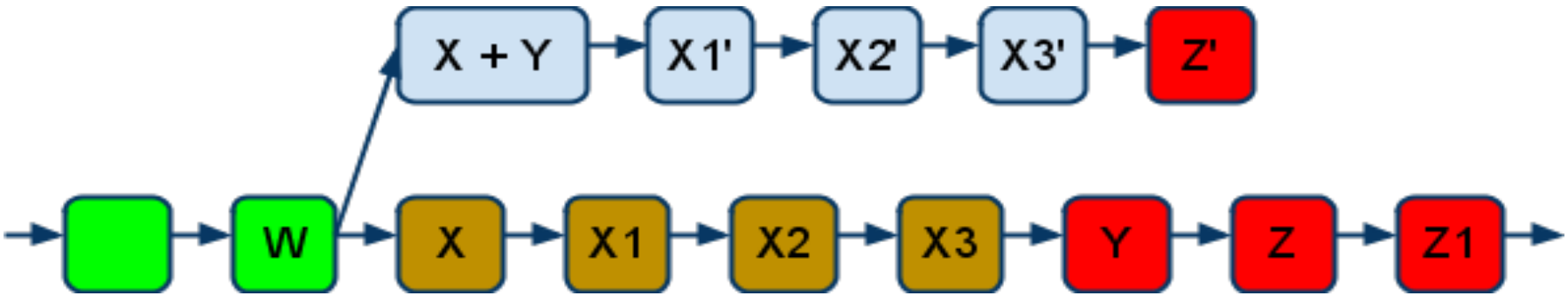


Commit X introduced a breakage, later fixed by commit Y.

The future of bisecting: "git replace"

Possible solutions to bisect anyway:

- apply a patch before testing and remove it after (can be done using "git cherry-pick"), or
- create a fixed up branch (can be done with "git rebase -i"), for example:



The future of bisecting: "git replace"

Fixed up branches are nice because:

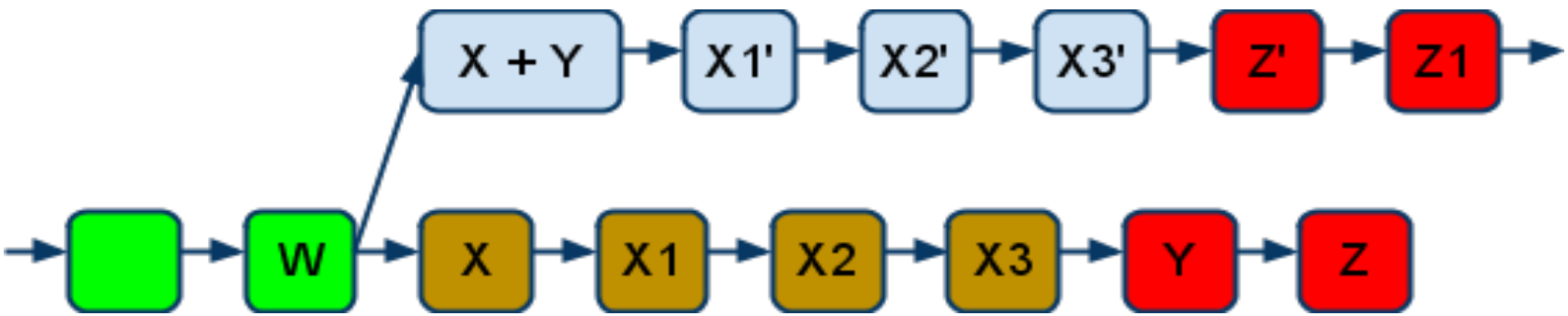
- they can be shared by users,
- regular git commands can be used on them,
- they avoid problems with patches that don't apply well.

But they are not so nice because:

- the bisection process is still disrupted,
- these branches clutter the branch name space.

The future of bisecting: "git replace"

The idea is that we will replace Z with Z' so that we bisect from the beginning using the fixed up branch.



$ git replace Z Z'

The future of bisecting: "git replace"

"git replace" uses replace refs stored in "refs/replace/" hierarchy, so:

- replace refs can be shared like branches and tags (refs),
- they don't clutter branch name space.

"git replace" is new in git version 1.6.5 (released in October 2009).

It was "sold" as an improvement over grafts.

The future of bisecting: sporadic bugs

Some bugs can depend on the compiler output and small changes unrelated to the bug can make it appear or disappear.

So "git bisect" is currently very unreliable to fight sporadic bugs.

The idea is to optionally add redundant tests when bisecting, for example 1 test out of 3 could be redundant. And if a redundant test fails, we hopefully will abort early.
There is an independent project called BBChop doing something like that based on Bayesian Search Theory.

Conclusion

- Regressions are an important problem;
- "git bisect" nicely complements best practices to fight them, especially general best practices and test suites;
- it may be worth it to adopt a special work-flow;
- "git bisect" could be improved in some cases, but
- it already works very well, is used a lot and is very useful.

Conclusion

Ingo Molnar when asked how much time it saves him:

a _lot_.

About ten years ago did i do my first 'bisection' of a Linux patch queue. That was prior the Git (and even prior the BitKeeper) days. I literally [spent days] sorting out patches, creating what in essence were standalone commits that i guessed to be related to that bug.

Conclusion

Ingo Molnar (continued):

It was a tool of absolute last resort. I'd rather spend days looking at printk output than do a manual 'patch bisection'.

With Git bisect it's a breeze: in the best case i can get a ~15 step kernel bisection done in 20-30 minutes, in an automated way. Even with manual help or when bisecting multiple, overlapping bugs, it's rarely more than an hour.

Conclusion

Ingo Molnar (continued):

In fact it's invaluable because there are bugs i would never even _try_ to debug if it wasn't for git bisect. In the past there were bug patterns that were immediately hopeless for me to debug - at best i could send the crash/bug signature to lkml and hope that someone else can think of something.

Conclusion

Ingo Molnar (continued):

And even if a bisection fails today it tells us something valuable about the bug: that it's non-deterministic - timing or kernel image layout dependent.

So git bisect is unconditional goodness - and feel free to quote that.

Many thanks to:

- Junio Hamano (comments, help, discussions, reviews, improvements),
- Ingo Molnar (comments, suggestions, evangelizing),
- Linus Torvalds (inventing, developing, evangelizing),
- many other great people in the Git and Linux communities, especially: Andreas Ericsson, Johannes Schindelin, H. Peter Anvin, Daniel Barkalow, Bill Lear, John Hawley, ...
- Linux-Kongress program committee.

Questions ?