Fighting regressions with git bisect

Christian Couder
36 rue de l'Alma
92600 Asnières sur Seine
France
chriscool@tuxfamily.org

1 Abstract

"git bisect" enables software users and developers to easily find the commit that introduced a regression. We show why it is important to have good tools to fight regressions. We describe how “git bisect” works from the outside and the algorithms it uses inside. Then we explain how to take advantage of “git bisect” to improve current practices. And we discuss how “git bisect” could improve in the future.

2 Introduction to "git bisect"

Git is a Distributed Version Control system (DVCS) created by Linus Torvalds and maintained by Junio Hamano.

In Git like in many other Version Control Systems (VCS), the different states of the data that is managed by the system are called commits. And, as VCS are mostly used to manage software source code, sometimes "interesting" changes of behavior in the software are introduced in some commits.

In fact people are specially interested in commits that introduce a "bad" behavior, called a bug or a regression. They are interested in these commits because a commit (hopefully) contains a very small set of source code changes. And it's much easier to understand and properly fix a problem when you only need to check a very small set of changes, than when you don't know where look in the first place.

So to help people find commits that introduce a "bad" behavior, the "git bisect" set of commands was invented. And it follows of course that in "git bisect" parlance, commits where the "interesting behavior" is present are called "bad" commits, while other commits are called "good" commits. And a commit that introduce the behavior we are interested in is called a "first bad commit". Note that there could be more than one "first bad commit" in the commit space we are searching.

So "git bisect" is designed to help find a "first bad commit". And to be as efficient as possible, it tries to perform a binary search.

3 Fighting regressions overview

3.1 Regressions: a big problem

Regressions are a big problem in the software industry. But it's difficult to put some real numbers behind that claim.

There are some numbers about bugs in general, like a NIST study in 2002 [1] that said:

Software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product, according to a newly released study commissioned by the Department of Commerce's National Institute of Standards and Technology (NIST). At the national level, over half of the costs are borne by software users and the remainder by software developers/vendors. The study also found that, although all errors cannot be removed, more than a third of these costs, or an estimated $22.2 billion, could be eliminated by an improved testing infrastructure that enables earlier and more effective identification and removal of software defects. These are the savings associated with finding an increased percentage (but not 100 percent) of errors closer to the development stages in which they are introduced. Currently, over half of all errors are not found until "downstream" in the development process or during post-sale software use.

And then:

Software developers already spend approximately 80 percent of development costs on identifying and correcting defects, and yet few products of any type other than software are shipped with such high levels of errors.

Eventually the conclusion started with:

The path to higher software quality is significantly improved software testing.

There are other estimates saying that 80% of the cost related to software is about maintenance [2].

Though, according to Wikipedia [3]:

A common perception of maintenance is that it is merely fixing bugs. However, studies and surveys over the years have indicated that the majority, over 80%, of the maintenance effort is used for non-corrective actions (Pigosky 1997). This perception is perpetuated by users submitting problem reports that in reality are functionality enhancements to the system.

But we can guess that improving on existing software is very costly because you have to watch out for regressions. At least this would make the above studies consistent among themselves.

Of course some kind of software is developed, then used during some time without being improved on much, and then finally thrown away. In this case, of course, regressions may not be a big problem. But on the other hand, there is a lot of big software that is continually developed and maintained during years or even tens of years by a lot of people. And as there are often many people who depend (sometimes critically) on such software, regressions are a really big problem.

One such software is the linux kernel. And if we look at the linux kernel, we can see that a lot of time and effort is spent to fight regressions. The release cycle start with a 2 weeks long merge window. Then the first release candidate (rc) version is tagged. And after that about 7 or 8 more rc versions will appear with around one week between each of them, before the final release.

The time between the first rc release and the final release is supposed to be used to test rc versions and fight bugs and especially regressions. And this time is more than 80% of the release cycle time. But this is not the end of the fight yet, as of course it continues after the release.

And then this is what Ingo Molnar (a well known linux kernel developer) says about his use of git bisect:

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.

So regressions are fought all the time by developers, and indeed it is well known that bugs should be fixed as soon as possible, so as soon as they are found. That's why it is interesting to have good tools for this purpose.

3.2 Other tools to fight regressions

So what are the tools used to fight regressions? They are nearly the same as those used to fight regular bugs. The only specific tools are test suites and tools similar as "git bisect".

Test suites are very nice. But when they are used alone, they are supposed to be used so that all the tests are checked after each commit. This means that they are not very efficient, because many tests are run for no interesting result, and they suffer from combinational explosion.

In fact the problem is that big software often has many different configuration options and that each test case should pass for each configuration after each commit. So if you have for each release: N configurations, M commits and T test cases, you should perform:

```
N * M * T tests
```

where N, M and T are all growing with the size your software.

So very soon it will not be possible to completely test everything.

And if some bugs slip through your test suite, then you can add a test to your test suite. But if you want to use your new improved test suite to find where the bug slipped in, then you will either have to emulate a bisection process or you will perhaps bluntly test each commit backward starting from the "bad" commit you have which may be very wasteful.

4 “git bisect” overview

4.1 Starting a bisection

The first "git bisect" subcommand to use is "git bisect start" to start the search. Then bounds must be set to limit the commit space. This is done usually by giving one "bad" and at least one "good" commit. They can be passed in the initial call to "git bisect start" like this:

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

or they can be set using:

```
$ git bisect bad [COMMIT]
```

and:

```
$ git bisect good [COMMIT...]
```

where BAD, GOOD and COMMIT are all names that can be resolved to a commit.

Then "git bisect" will checkout a commit of its choosing and ask the user to test it, like this:

```
$ 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
```

Note that the example that we will use is really a toy example, we will be looking for the first commit that has a version like "2.6.26-something", that is the commit that has a "SUBLEVEL = 26" line in the top level

Makefile. This is a toy example because there are better ways to find this commit with git than using "git bisect" (for example "git blame" or "git log -S<string>").

4.2 Driving a bisection manually

At this point there are basically 2 ways to drive the search. It can be driven manually by the user or it can be driven automatically by a script or a command.

If the user is driving it, then at each step of the search, the user will have to test the current commit and say if it is "good" or "bad" using the "git bisect good" or "git bisect bad" commands respectively that have been described above. 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
```

And after a few more steps like that, "git bisect" will eventually find a first bad commit:

```
$ 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
```

At this point we can see what the commit does, check it out (if it's not already checked out) or tinker with it, for example:

```
$ git show HEAD
commit 2ddcca36c8bcfa251724fe342c8327451988be0d
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Sat May 3 11:59:44 2008 -0700

    Linux 2.6.26-rc1

diff --git a/Makefile b/Makefile
index 5cf8258..4492984 100644
--- a/Makefile
+++ b/Makefile
@@ -1,7 +1,7 @@
 VERSION = 2
 PATCHLEVEL = 6
-SUBLEVEL = 25
-EXTRAVERSION =
+SUBLEVEL = 26
+EXTRAVERSION = -rc1
 NAME = Funky Weasel is Jiggy wit it

 # *DOCUMENTATION*
```

And when we are finished we can use "git bisect reset" to go back to the branch we were in before we started bisecting:

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

4.3 Driving a bisection automatically

The other way to drive the bisection process is to tell "git bisect" to launch a script or command at each bisection step to know if the current commit is "good" or "bad". To do that, we use the "git bisect run" command. For example:

```
$ 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
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
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
```

In this example, we passed "grep '^SUBLEVEL = 25' Makefile" as parameter to "git bisect run". This means that at each step, the grep command we passed will be launched. And if it exits with code 0 (that means success) then git bisect will mark the current state as "good". If it exits with code 1 (or any code between 1 and 127 included, except the special code 125), then the current state will be marked as "bad".

Exit code between 128 and 255 are special to "git bisect run". They make it stop immediately the bisection process. This is useful for example if the command passed takes too long to complete, because you can kill it with a signal and it will stop the bisection process.

It can also be useful in scripts passed to "git bisect run" to "exit 255" if some very abnormal situation is detected.

4.4 Avoiding untestable commits

Sometimes it happens that the current state cannot be tested, for example if it does not compile because there was a bug preventing it at that time. This is what the special exit code 125 is for. It tells "git bisect run" that the current commit should be marked as untestable and that another one should be chosen and checked out.

If the bisection process is driven manually, you can use "git bisect skip" to do the same thing. (In fact the special exit code 125 makes "git bisect run" use "git bisect skip" in the background.)

Or if you want more control, you can inspect the current state using for example "git bisect visualize". It will launch gitk (or "git log" if the DISPLAY environment variable is not set) to help you find a better bisection point.

Either way, if you have a string of untestable commits, it might happen that the regression you are looking for has been introduced by one of these untestable commits. In this case it's not possible to tell for sure which commit introduced the regression.

So if you used "git bisect skip" (or the run script exited with special code 125) you could get a result like this:

```
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!
```

4.5 Saving a log and replaying it

If you want to show other people your bisection process, you can get a log using for example:

```
$ git bisect log > bisect_log.txt
```

And it is possible to replay it using:

```
$ git bisect replay bisect_log.txt
```

5 “git bisect” details

5.1 Bisection algorithm

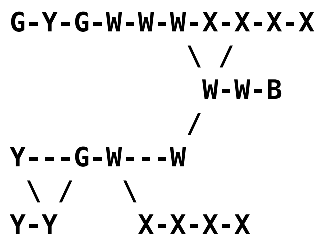
As the Git commits form a directed acyclic graph (DAG), finding the best bisection commit to test at each step is not so simple. Anyway Linus found and implemented a "truly stupid" algorithm, later improved by Junio Hamano, that works quite well.

So the algorithm used by "git bisect" to find the best bisection commit when there are no skipped commits is the following:

1) keep only the commits that:
   a) are ancestor of the "bad" commit (including the "bad" commit itself),
   b) are not ancestor of a "good" commit (excluding the "good" commits).

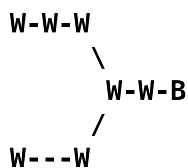
This means that we get rid of the uninteresting commits in the DAG.

For example if we start with a graph like this:




```
-> time goes this way ->
```

where B is the "bad" commit, "G" are "good" commits and W, X, and Y are other commits, we will get the following graph after this first step:

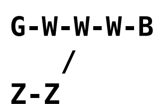


So only the W and B commits will be kept. Because commits X and Y will have been removed by rules a) and b) respectively, and because commits G are removed by rule b) too.

Note for git users, that it is equivalent as keeping only the commit given by:

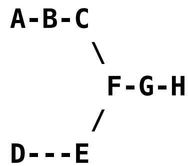
```
git rev-list BAD --not GOOD1 GOOD2...
```

Also note that we don't require the commits that are kept to be descendants of a “good” commit. So in the following example, commits W and Z will be kept:

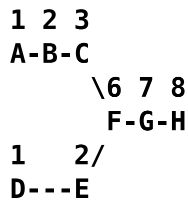


2) starting from the "good" ends of the graph, associate to each commit the number of ancestors it has plus one

For example with the following graph where H is the "bad" commit and A and D are some parents of some "good" commits:

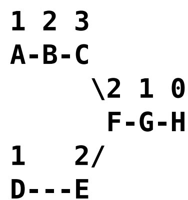


this will give:



3) associate to each commit: min(X, N – X), where X is the value associated to the commit in step 2) and N is the total number of commits in the graph

In the above example we have N = 8, so this will give:



4) the best bisection point is the commit with the highest associated number

So in the above example the best bisection point is commit C.

5) note that some shortcuts are implemented to speed up the algorithm

As we know N from the beginning, we know that min(X, N - X) can't be greater than N/2. So during steps 2) and 3), if we would associate N/2 to a commit, then we know this is the best bisection point. So in this case we can just stop processing any other commit and return the current commit.

5.2 Bisection algorithm debugging

For any commit graph, you can see the number associated with each commit using "git rev-list –bisect-all".

For example, for the above graph, a command like:

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

would output something like:

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

5.3 Bisection algorithm discussed

First let's define "best bisection point". We will say that a commit X is a best bisection point or a best bisection commit if knowing its state ("good" or "bad") gives as much information as possible whether the state of the commit happens to be "good" or "bad".

This means that the best bisection commits are the commits where the following function is maximum:

```
f(X) = min(information_if_good(X), information_if_bad(X))
```

where information_if_good(X) is the information we get if X is good and information_if_bad(X) is the information we get if X is bad.

Now we will suppose that there is only one "first bad commit". This means that all its descendants are "bad" and all the other commits are "good". And we will suppose that all commits have an equal probability of being good or bad, or of being the first bad commit, so knowing the state of c commits gives always the same amount of information wherever these c commits are on the graph and whatever c is. (So we suppose that these commits being for example on a branch or near a good or a bad commit does not give more or less information).

Let's also suppose that we have a cleaned up graph like one after step 1) in the bisection algorithm above. This means that we can measure the information we get in terms of number of commit we can remove from the graph..

And let's take a commit X in the graph.

If X is found to be "good", then we know that its ancestors are all "good", so we want to say that:

```
information_if_good(X) = number_of_ancestors(X)  (TRUE)
```

And this is true because at step 1) b) we remove the ancestors of the "good" commits.

If X is found to be "bad", then we know that its descendants are all "bad", so we want to say that:

```
information_if_bad(X) = number_of_descendants(X)  (WRONG)
```

But this is wrong because at step 1) a) we keep only the ancestors of the bad commit. So we get more information when a commit is marked as "bad", because we also know that the ancestors of the previous "bad" commit that are not ancestors of the new "bad" commit are not the first bad commit. We don't know if they are good or bad, but we know that they are not the first bad commit because they are not ancestor of the new "bad" commit.

So when a commit is marked as "bad" we know we can remove all the commits in the graph except those that are ancestors of the new "bad" commit. This means that:

```
information_if_bad(X) = N - number_of_ancestors(X)  (TRUE)
```

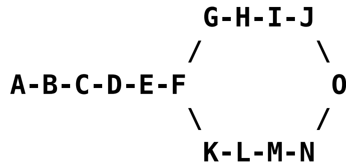
where N is the number of commits in the (cleaned up) graph.

So in the end this means that to find the best bisection commits we should maximize the function:

f(X) = min(number_of_ancestors(X), N - number_of_ancestors(X))

And this is nice because at step 2) we compute number_of_ancestors(X) and so at step 3) we compute f(X).

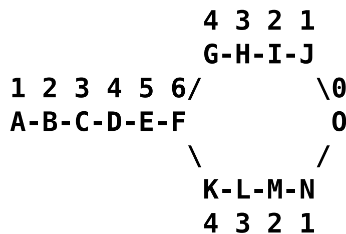
Let's take the following graph as an example:



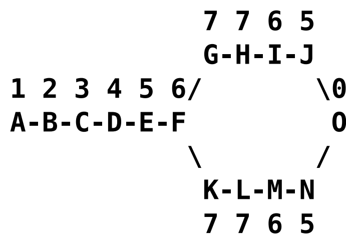
If we compute the following non optimal function on it:

g(X) = min(number_of_ancestors(X), number_of_descendants(X))

we get:



but with the algorithm used by git bisect we get:



So we chose G, H, K or L as the best bisection point, which is better than F. Because if for example L is bad, then we will know not only that L, M and N are bad but also that G, H, I and J are not the first bad commit (since we suppose that there is only one first bad commit and it must be an ancestor of L).

So the current algorithm seems to be the best possible given what we initially supposed.

5.4 Skip algorithm

When some commits have been skipped (using "git bisect skip"), then the bisection algorithm is the same for step 1) to 3). But then we use roughly the following steps:

6) sort the commit by decreasing associated value

7) if the first commit has not been skipped, we can return it and stop here

8) otherwise filter out all the skipped commits in the sorted list

9) use a pseudo random number generator (PRNG) to generate a random number between 0 and 1

10) multiply this random number with its square root to bias it toward 0

11) multiply the result by the number of commits in the filtered list to get an index into this list

12) return the commit at the computed index

5.5 Skip algorithm discussed

After step 7) (in the skip algorithm), we could check if the second commit has been skipped and return it if it is not the case. And in fact that was the algorithm we used from when "git bisect skip" was developed in git version 1.5.4 (released on February 1st 2008) until git version 1.6.4 (released July 29th 2009).

But Ingo Molnar and H. Peter Anvin (another well known linux kernel developer) both complained that sometimes the best bisection points all happened to be in an area where all the commits are untestable. And in this case the user was asked to test many untestable commits, which could be very inefficient.

Indeed untestable commits are often untestable because a breakage was introduced at one time, and that breakage was fixed only after many other commits were introduced.

This breakage is of course most of the time unrelated to the breakage we are trying to locate in the commit graph. But it prevents us to know if the interesting "bad behavior" is present or not.

So it is a fact that commits near an untestable commit have a high probability of being untestable themselves. And the best bisection commits are often found together too (due to the bisection algorithm).

This is why it is a bad idea to just chose the next best unskipped bisection commit when the first one has been skipped.

We found that most commits on the graph may give quite a lot of information when they are tested. And the commits that will not on average give a lot of information are the one near the good and bad commits.

So using a PRNG with a bias to favor commits away from the good and bad commits looked like a good choice.

One obvious improvement to this algorithm would be to look for a commit that has an associated value near the one of the best bisection commit, and that is on another branch, before using the PRNG. Because if such a commit exists, then it is not very likely to be untestable too, so it will probably give more information than a nearly randomly chosen one.

5.6 Checking merge bases

There is another tweak in the bisection algorithm that has not been described in the "bisection algorithm" above.

We supposed in the previous examples that the "good" commits were ancestors of the "bad" commit. But this is not a requirement of "git bisect".

Of course the "bad" commit cannot be an ancestor of a "good" commit, because the ancestors of the good commits are supposed to be "good". And all the "good" commits must be related to the bad commit. They cannot be on a branch that has no link with the branch of the "bad" commit. But it is possible for a good commit to be related to a bad commit and yet not be neither one of its ancestor nor one of its descendants.

For example, there can be a "main" branch, and a "dev" branch that was forked of the main branch at a commit named "D" like this:

```
A-B-C-D-E-F-G  <--main
       \
        H-I-J  <--dev
```

The commit "D" is called a "merge base" for branch "main" and "dev" because it's the best common ancestor for these branches for a merge.

Now let's suppose that commit J is bad and commit G is good and that we apply the bisection algorithm like it has been previously described.

As described in step 1) b) of the bisection algorithm, we remove all the ancestors of the good commits because they are supposed to be good too.

So we would be left with only:

```
H-I-J
```

But what happens if the first bad commit is "B" and if it has been fixed in the "main" branch by commit "F"?

The result of such a bisection would be that we would find that H is the first bad commit, when in fact it's B. So that would be wrong!

And yes it's can happen in practice that people working on one branch are not aware that people working on another branch fixed a bug! It could also happen that F fixed more than one bug or that it is a revert of some big development effort that was not ready to be released.

In fact development teams often maintain both a development branch and a maintenance branch, and it would be quite easy for them if "git bisect" just worked when they want to bisect a regression on the development branch that is not on the maintenance branch. They should be able to start bisecting using:

```
$ git bisect start dev main
```

To enable that additional nice feature, when a bisection is started and when some good commits are not ancestors of the bad commit, we first compute the merge bases between the bad and the good commits and we chose these merge bases as the first commits that will be checked out and tested.

If it happens that one merge base is bad, then the bisection process is stopped with a message like:

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

where BBBBBB is the sha1 hash of the bad merge base and [GGGGGG,...] is a comma separated list of the sha1 of the good commits.

If some of the merge bases are skipped, then the bisection process continues, but the following message is printed for each skipped merge base:

```
Warning: the merge base between BBBBBB and [GGGGGG,...] must be skipped.
So we cannot be sure the first bad commit is between MMMMMM and BBBBBB.
We continue anyway.
```

where BBBBBB is the sha1 hash of the bad commit, MMMMMM is the sha1 hash of the merge base that is skipped and [GGGGGG,...] is a comma separated list of the sha1 of the good commits.

So if there is no bad merge base, the bisection process continues as usual after this step.

6 Best bisecting practices

6.1 Using test suites and git bisect together

If you both have a test suite and use git bisect, then it becomes less important to check that all tests pass after each commit. Though of course it is probably a good idea to have some checks to avoid breaking too many things because it could make bisecting other bugs more difficult.

You can focus your efforts to check at a few points (for example rc and beta releases) that all the T test cases pass for all the N configurations. And when some tests don't pass you can use "git bisect" (or better "git bisect run"). So you should perform roughly:

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

where c is the number of rounds of test (so a small constant) and b is the ratio of bug per commit (hopefully a small constant too).

So of course it's much better as it's O(N * T) vs O(N * T * M) if you would test everything after each commit.

This means that test suites are good to prevent some bugs from being committed and they are also quite good to tell you that you have some bugs. But they are not so good to tell you where some bugs have been introduced. To tell you that efficiently, git bisect is needed.

The other nice thing with test suites, is that when you have one, you already know how to test for bad behavior. So you can use this knowledge to create a new test case for "git bisect" when it appears that there is a regression. So it will be easier to bisect the bug and fix it. And then you can add the test case you just created to your test suite.

So if you know how to create test cases and how to bisect, you will be subject to a virtuous circle:

```
more tests => easier to create tests => easier to bisect => more tests
```

So test suites and "git bisect" are complementary tools that are very powerful and efficient when used together.

6.2 Bisecting build failures

You can very easily automatically bisect broken builds using something like:

```
$ git bisect start BAD GOOD
$ git bisect run make
```

6.3 Passing sh -c "some commands" to "git bisect run"

For example:

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

On the other hand if you do this often, then it can be worth having scripts to avoid too much typing.

6.4 Finding performance regressions

Here is an example script that comes slightly modified from a real world script used by Junio Hamano [4].

This script can be passed to "git bisect run" to find the commit that introduced a performance regression:

```
#!/bin/sh

# Build errors are not what I am interested in.
make my_app || exit 255

# We are checking if it stops in a reasonable amount of time, so
# let it run in the background...

./my_app >log 2>&1 &

# ... and grab its process ID.
pid=$!

# ... and then wait for sufficiently long.
sleep $NORMAL_TIME

# ... and then see if the process is still there.
if kill -0 $pid
then
      # It is still running -- that is bad.
      kill $pid; sleep 1; kill $pid;
      exit 1
else
      # It has already finished (the $pid process was no more),
        # and we are happy.
      exit 0
fi
```

6.5 Following general best practices

It is obviously a good idea not to have commits with changes that knowingly break things, even if some other commits later fix the breakage.

It is also a good idea when using any VCS to have only one small logical change in each commit.

The smaller the changes in your commit, the most effective "git bisect" will be. And you will probably need "git bisect" less in the first place, as small changes are easier to review even if they are only reviewed by the commiter.

Another good idea is to have good commit messages. They can be very helpful to understand why some changes were made.

These general best practices are very helpful if you bisect often.

6.6 Avoiding bug prone merges

First merges by themselves can introduce some regressions even when the merge needs no source code conflict resolution. This is because a semantic change can happen in one branch while the other branch is not aware of it.

For example one branch can change the semantic of a function while the other branch add more calls to the same function.

This is made much worse if many files have to be fixed to resolve conflicts. That's why such merges are called "evil merges". They can make regressions very difficult to track down. It can even be misleading to know the first bad commit if it happens to be such a merge, because people might think that the bug comes from bad conflict resolution when it comes from a semantic change in one branch.

Anyway "git rebase" can be used to linearize history. This can be used either to avoid merging in the first place. Or it can be used to bisect on a linear history instead of the non linear one, as this should give more information in case of a semantic change in one branch.

Merges can be also made simpler by using smaller branches or by using many topic branches instead of only long version related branches.

And testing can be done more often in special integration branches like linux-next for the linux kernel.

6.7 Adapting your work-flow

A special work-flow to process regressions can give great results.

Here is an example of a work-flow used by Andreas Ericsson:

- write, in the test suite, a test script that exposes the regression
- use "git bisect run" to find the commit that introduced it
- fix the bug that is often made obvious by the previous step
- commit both the fix and the test script (and if needed more tests)

And here is what Andreas said about this work-flow [5]:

To give some hard figures, we used to have an average report-to-fix cycle of 142.6 hours (according to our somewhat weird bug-tracker which just measures wall-clock time). Since we moved to git, we've lowered that to 16.2 hours. Primarily because we can stay on top of the bug fixing now, and because everyone's jockeying to get to fix bugs (we're quite proud of how lazy we are to let git find the bugs for us). Each new release results in ~40% fewer bugs (almost certainly due to how we now feel about writing tests).

Clearly this work-flow uses the virtuous circle between test suites and "git bisect". In fact it makes it the standard procedure to deal with regression.

In other messages Andreas says that they also use the "best practices" described above: small logical commits, topic branches, no evil merge,... These practices all improve the bisectability of the commit graph, by making it easier and more useful to bisect.

So a good work-flow should be designed around the above points. That is making bisecting easier, more useful and standard.

6.8 Involving QA people and if possible end users

One nice about "git bisect" is that it is not only a developer tool. It can effectively be used by QA people or even end users (if they have access to the source code or if they can get access to all the builds).

There was a discussion at one point on the linux kernel mailing list of whether it was ok to always ask end user to bisect, and very good points were made to support the point of view that it is ok.

For example David Miller wrote [6]:

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.

This means that it is often "cheaper" if QA people or end users can do it.

What is interesting too is that end users that are reporting bugs (or QA people that reproduced a bug) have access to the environment where the bug happens. So they can often more easily reproduce a regression. And if they can bisect, then more information will be extracted from the environment where the bug happens, which means that it will be easier to understand and then fix the bug.

For open source projects it can be a good way to get more useful contributions from end users, and to introduce them to QA and development activities.

6.9 Using complex scripts

In some cases like for kernel development it can be worth developing complex scripts to be able to fully automate bisecting.

Here is what Ingo Molnar says about that [7]:

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)

6.10 Combining test suites, git bisect and other systems together

We have seen that test suites an git bisect are very powerful when used together. It can be even more powerful if you can combine them with other systems.

For example some test suites could be run automatically at night with some unusual (or even random) configurations. And if a regression is found by a test suite, then "git bisect" can be automatically launched, and its result can be emailed to the author of the first bad commit found by "git bisect", and perhaps other people too. And a new entry in the bug tracking system could be automatically created too.

7 The future of bisecting

7.1 “git replace”

We saw earlier that "git bisect skip" is now using a PRNG to try to avoid areas in the commit graph where commits are untestable. The problem is that sometimes the first bad commit will be in an untestable area.

To simplify the discussion we will suppose that the untestable area is a simple string of commits and that it was created by a breakage introduced by one commit (let's call it BBC for bisect breaking commit) and later fixed by another one (let's call it BFC for bisect fixing commit).

For example:

```
...-Y-BBC-X1-X2-X3-X4-X5-X6-BFC-Z-...
```

where we know that Y is good and BFC is bad, and where BBC and X1 to X6 are untestable.

In this case if you are bisecting manually, what you can do is create a special branch that starts just before the BBC. The first commit in this branch should be the BBC with the BFC squashed into it. And the other commits in the branch should be the commits between BBC and BFC rebased on the first commit of the branch and then the commit after BFC also rebased on.

For example:

```
      (BBC+BFC)-X1'-X2'-X3'-X4'-X5'-X6'-Z'
     /
...-Y-BBC-X1-X2-X3-X4-X5-X6-BFC-Z-...
```

where commits quoted with ' have been rebased.

You can easily create such a branch with Git using interactive rebase.

For example using:

```
$ git rebase -i Y Z
```

and then moving BFC after BBC and squashing it.

After that you can start bisecting as usual in the new branch and you should eventually find the first bad commit.

For example:

```
$ git bisect start Z' Y
```

If you are using "git bisect run", you can use the same manual fix up as above, and then start another "git bisect run" in the special branch. Or as the "git bisect" man page says, the script passed to "git bisect run" can apply a patch before it compiles and test the software [8]. The patch should turn a current untestable commits into a testable one. So the testing will result in "good" or "bad" and "git bisect" will be able to find the first bad commit. And the script should not forget to remove the patch once the testing is done before exiting from the script.

(Note that instead of a patch you can use "git cherry-pick BFC" to apply the fix, and in this case you should use "git reset --hard HEAD^" to revert the cherry-pick after testing and before returning from the script.)

But the above ways to work around untestable areas are a little bit clunky. Using special branches is nice because these branches can be shared by developers like usual branches, but the risk is that people will get many such branches. And it disrupts the normal "git bisect" work-flow. So, if you want to use "git bisect run" completely automatically, you have to add special code in your script to restart bisection in the special branches.

Anyway one can notice in the above special branch example that the Z' and Z commits should point to the same source code state (the same "tree" in git parlance). That's because Z' result from applying the same changes as Z just in a slightly different order.

So if we could just "replace" Z by Z' when we bisect, then we would not need to add anything to a script. It would just work for anyone in the project sharing the special branches and the replacements.

With the example above that would give:

```
      (BBC+BFC)-X1'-X2'-X3'-X4'-X5'-X6'-Z'-...
     /
...-Y-BBC-X1-X2-X3-X4-X5-X6-BFC-Z
```

That's why the "git replace" command was created. Technically it stores replacements "refs" in the "refs/replace/" hierarchy. These "refs" are like branches (that are stored in "refs/heads/") or tags (that are stored in "refs/tags"), and that means that they can automatically be shared like branches or tags among developers.

"git replace" is a very powerful mechanism. It can be used to fix commits in already released history, for example to change the commit message or the author. And it can also be used instead of git "grafts" to link a repository with another old repository.

In fact it's this last feature that "sold" it to the git community, so it is now in the "master" branch of git's git repository and it should be released in git 1.6.5 in October or November 2009.

One problem with "git replace" is that currently it stores all the replacements refs in "refs/replace/", but it would be perhaps better if the replacement refs that are useful only for bisecting would be in "refs/replace/bisect/". This way the replacement refs could be used only for bisecting, while other refs directly in "refs/replace/" would be used nearly all the time.

7.2 Bisecting sporadic bugs

Another possible improvement to "git bisect" would be to optionally add some redundancy to the tests performed so that it would be more reliable when tracking sporadic bugs.

This has been requested by some kernel developers because some bugs called sporadic bugs do not appear in all the kernel builds because they are very dependent on the compiler output.

The idea is that every 3 test for example, "git bisect" could ask the user to test a commit that has already been found to be "good" or "bad" (because one of its descendants or one of its ancestors has been found to be "good" or "bad" respectively). If it happens that a commit has been previously incorrectly classified then the bisection can be aborted early, hopefully before too many mistakes have been made. Then the user will have to look at what happened and then restart the bisection using a fixed bisect log.

There is already a project called BBChop created by Ealdwulf Wuffinga on Github that does something like that using Bayesian Search Theory [9]:

BBChop is like 'git bisect' (or equivalent), but works when your bug is intermittent. That is, it works in the presence of false negatives (when a version happens to work this time even though it contains the bug). It assumes that there are no false positives (in principle, the same approach would work, but adding it may be non-trivial).

But BBChop is independent of any VCS and it would be easier for Git users to have something integrated in Git.

8 Conclusion

We have seen that regressions are an important problem, and that “git bisect” has nice features that complement very well practices and other tools, especially test suites, that are generally used to fight regressions. But it might be needed to change some work-flows and (bad) habits to get the most out of it.

Some improvements to the algorithms inside “git bisect” are possible and some new features could help in some cases, but overall “git bisect” works already very well, is used a lot, and is already very useful. To back up that last claim, let's give the final word to Ingo Molnar when he was asked by the author how much time does he think "git bisect" saves him when he uses it:

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 days spent sorting out patches, creating what in essence were standalone commits that i guessed to be related to that bug.

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.

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.

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
;-)

9 Acknowledgements

Many thanks to Junio Hamano for his help in reviewing this paper, for reviewing the patches I sent to the git mailing list, for discussing some ideas and helping me improve them, for improving "git bisect" a lot and for his awesome work in maintaining and developing Git.

Many thanks to Ingo Molnar for giving me very useful information that appears in this paper, for commenting on this paper, for his suggestions to improve "git bisect" and for evangelizing "git bisect" on the linux kernel mailing lists.

Many thanks to Linus Torvalds for inventing, developing and evangelizing "git bisect", Git and Linux.

Many thanks to the many other great people who helped one way or another when I worked on git, especially to Andreas Ericsson, Johannes Schindelin, H. Peter Anvin, Daniel Barkalow, Bill Lear, John Hawley, Shawn O. Pierce, Jeff King, Sam Vilain, Jon Seymour.

Many thanks to the Linux-Kongress program committee for choosing the author to given a talk and for publishing this paper.

References

[1] http://www.nist.gov/public_affairs/releases/n02-10.htm

[2] http://java.sun.com/docs/codeconv/html/CodeConventions.doc.html#16712

[3] http://en.wikipedia.org/wiki/Software_maintenance

[4] http://article.gmane.org/gmane.comp.version-control.git/45195/

[5] http://lwn.net/Articles/317154/

[6] http://lwn.net/Articles/277872/

[7] http://article.gmane.org/gmane.linux.scsi/36652/

[8] http://www.kernel.org/pub/software/scm/git/docs/git-bisect.html

[9] http://github.com/Ealdwulf/bbchop