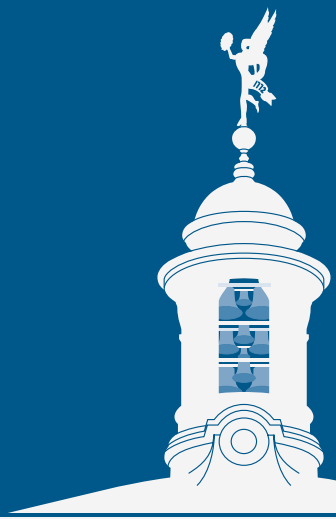
WWU
MÜNSTER

Git for Mathematical Writing

A tool for proper version control and collaborative writing

Jannes Bantje

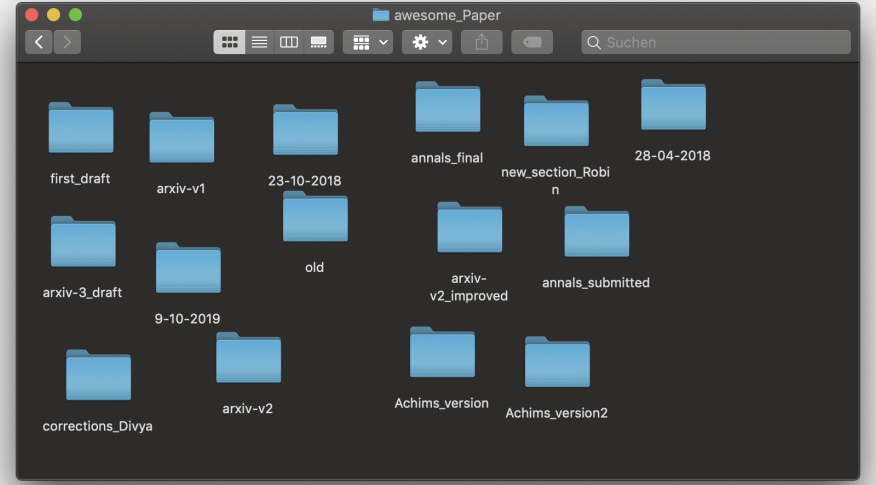
October 9, 2019



living.knowledge

MM
Mathematics
Münster
Cluster of Excellence

The Problem


awesome_Paper
Suchen
first_draft
arxiv-v1
23-10-2018
annals_final
new_section_Robin
28-04-2018
arxiv-3_draft
9-10-2019
old
arxiv-v2_improved
annals_submitted
corrections_Divya
arxiv-v2
Achims_version
Achims_version2

Concrete Problems

The following tasks are hard/virtually impossible/actually impossible, with this type of “version control”:

- Recover an idea, that was abandoned earlier
- Review changes of a file to find the cause of that cryptic LaTeX error, that did not occur with earlier versions
- Review what changes have been made during `<insert arbitrary amount of time>`
- When was this line of code written? (and by whom?)
- Compare v1 of your arXiv article to v2
- Revert changing all the φ’s to ϕ
- Merge the changes of your coauthor with yours
- …

(The file history features of Sciebo or Dropbox certainly help, but they do *not* solve those problems)

The Solution:

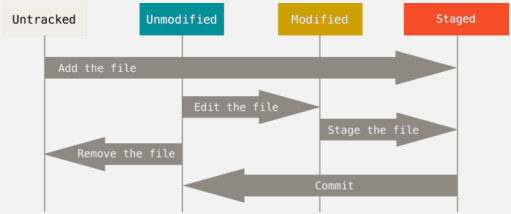
git

- Originally developed in 2005 by Linus Torvalds, the creator of Linux, and therefore open-source
- Torvalds needed a version control system for the Linux kernel
- “I’m an egoistical bastard, and I name all my project after myself, first ‘Linux’ now ‘git’”
- Facilitated by web services like GitHub and GitLab, git has become the de facto standard for version control in software development
- Microsoft migrated the Windows source code to Git in 2017

http://xkcd.com/1597/

First Steps – the basic commands

- Initialise a new repository: `git init`
- Get basic information about the status, that the repository is in: `git status`
- New and modified files have to be added to the so called **staging area**:
  `git add <filename>`
  Wildcards are supported: `git add *.tex` adds all `.tex`-files
- All staged changes are bundled into a **commit**, which you might read as “version”
- A commit necessarily also has to have a commit message:
  `git commit -m " <commit message> "`



lifecycle of a file, taken from [CS14]

Let’s see that in action …

The `.gitignore` file

There are files, that should *not* be tracked by git as they would unnecessarily clutter your repository:

- The files that are produced by compiling the source code, e.g. PDFs in the case of LaTeX
- Intermediate and log files (for example `.aux`-files)
- hidden files automatically generated by your file browser, ...

Git can be told to ignore specific files by adding a `.gitignore`-file to your repository, that specifies patterns like so:

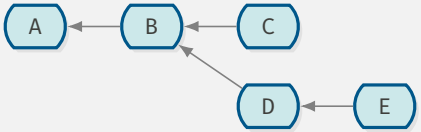
```
## Core latex/pdflatex auxiliary files:
*.aux
*.log
*.out
*.toc
# Exclude PDFs
*.pdf
# Exception: PDFs in 'img'
!**/img/*.pdf
```

A closer look on commits

A commit consists of the following data:

- an unique identifier (a *hash*)
- a commit message
- name and email address of the committer
- date and time of the commit
- the actual changes …
- …relative to the **parent commit**(s)

Taking commits as vertices and the parent relationship as edges, we get a **directed acyclic graph** representing our repository:



The hash of a commit can be used to `git checkout <hash>` a given commit (and thereby create graphs like the one above).

References

Humans are bad at processing information like

`68506005947951925 3b8ae12ba51feafcd107f57`

Definition

References are pointers to commits.

There are a two different flavours:

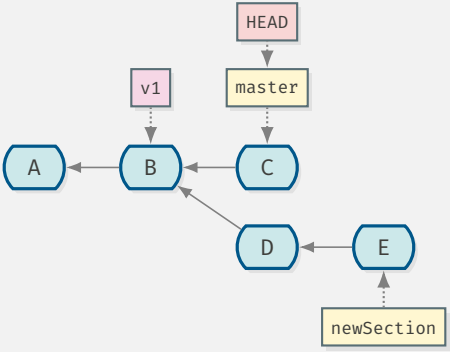
Branch a reference like “`master`” or “`new_section`” to some commit in your repository. They come in a local and remote sub-flavour, as we will see in a moment.
Created by `git branch` and updated by `git commit` (and several others).

Tag a reference that never moves, useful for marking specific versions, for example “`arXiv_v1`”.
Created by `git tag <name>`

References make human-friendly commands like `git checkout new_section` comprehensible to git.

Illustrated Example

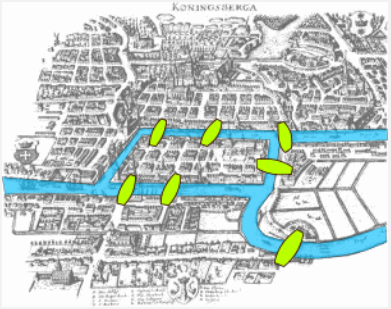
The references can be seen as labels of our graph: `git checkout master`



Note: Git will refuse to delete the branch `newSection` at this point.

Graph Theory and Reachability

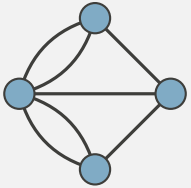
A graph essentially records **places to go and ways to get there**.



References make commits reachable, [Liv17]

A commit (=node in the graph) will – eventually – be deleted by git's garbage collection, if it is not reachable from any of the references!

- git generally refuses to perform actions that would render any commits unreachable.
- (This is extremely helpful to keep in mind, when performing more advanced tasks in git)

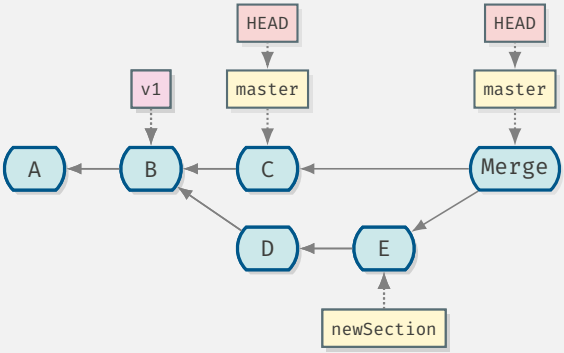


Solved Problems (so far)

- Recover an idea, that was abandoned earlier ✓
- Review changes of a file to find the cause of that cryptic LaTeX error, that did not occur with earlier versions ✓
- Review what changes have been made during `<insert arbitrary amount of time>` ✓
- When was this line of code written? (and by whom?) — use `git blame` ✓
- Compare v1 of your arXiv article to v2 ✓
- Revert changing all the φ's to ϕ — use `git revert` ✓
- Merge the changes of your coauthor with yours

Merging

Let's look at our earlier example (which does not incorporate a coauthor just yet):



- To merge `newSection` into `master`: `git merge newSection` (`master` is checked out)
- Now, `newSection` can safely be deleted, since every commit is still reachable from `master`.

Communication with other Repositories

- Your local repo has a list of **remote** repositories (usually just one, canonically named `origin`)
- remote repos are often hosted on GitHub, GitLab, etc. (which provide a sleek web interface *around* that repo)
- Local branches track corresponding ones on the remote, for example the local branch `master` tracks the remote branch `origin/master`
- there are 3 main commands for exchanging commits:

  `fetch` downloads all changes, that occurred on the remote
  `pull` downloads all changes and tries to apply them to the local branches
  `push` uploads local changes and advances branches on the remote

Let's see that in action …

Using Git for Managing TEX-files

Only commit the source files of your document!

Do not commit `.pdf` files and all the intermediate files (like `.toc`, `.aux`, …) as they unnecessarily clutter the repository. Use a `.gitignore`-file to exclude them.

(For the same reason you would not commit compiled programs, for example the `.class`-file produced by the Java compiler)

One sentence per line of code!

Git “thinks” in lines of code, in particular diffs are displayed as “this line was changed to that one”. Having a whole paragraph of text in one line of the code makes those diffs hard to read.

Installation guide

Git itself is available for every operating system:

- Linux: prepackaged with most distributions (⇒ installed on all university computers)
- macOS: run “`git`” on the terminal and follow the installation instructions
- Windows: Installer from `https://gitforwindows.org/` (comes with an extra terminal) or via the Windows Subsystem for Linux.

Recommendable GUI clients:

- Fork, `https://git-fork.com/` (macOS, Windows)
- GitKraken, `https://www.gitkraken.com/git-client` (Linux, macOS, Windows)
- SourceTree, `https://www.sourcetreeapp.com/` (macOS, Windows)

Bibliography I

[CS14] Scott Chacon and Ben Straub. *Pro Git book*. Apress, 2014. URL: https://git-scm.com/book/en/v2 (visited on 04/13/2016).

[Liv17] Sam Livingston-Gray. *Think Like (a) Git*. 2017. URL: http://think-like-a-git.net/ (visited on 10/09/2019).