Lab 4: Version Control With Git and GitHub

DS-GA 1004, Spring 2016

Erin Carson and Nicholas Knight

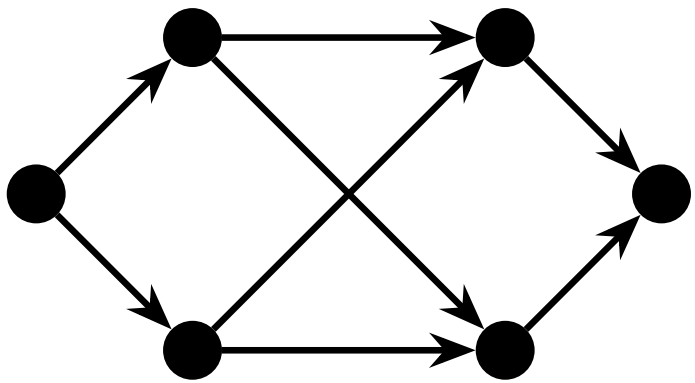
Dept. of Mathematics
Courant Institute of Mathematical Sciences
New York University

February 22, 2016

Version Control

Version control (a.k.a. *revision control*):

- A logical approach for tracking changes/revisions to a project
- Useful for coordinating multiple editors



The *version history diagram* — a rooted directed acyclic graph, (V, E, s)

- V, the set of versions
- $(u, v) \in E$ if u is revised by v
- s is the initial version (the root)

Git: Basic Terminology

Git provides version control for files.

- project = *repository*, a database of files
- version/revision = *commit*, a snapshot of the database
- history diagram grows along paths called *branches*

Useful Git references:

- Git Pro (book, 2014): `http://git-scm.com/book`
- Git Reference Manual: `http://git-scm.com/docs`
  - Same man-pages available via `git help foo` or `man git-foo`.

Task 0: Setup

1. Download and install Git: `http://git-scm.com/download`
   - Try `git --version` to see if you already have it

2. Configure Git (if you haven't already):

```
git config --global user.name "NAME"
git config --global user.email "EMAIL ADDRESS"
git config --global core.editor EDITOR
git config --global -l
```

3. Sign up for GitHub account: `http://github.com/`

Deliverable: None, but Tasks 1 and 2 depend on Task 0.

Creating a Local Repository

Create a new repository:

```
mkdir ds1004; cd ds1004
git init .
ls -alp
ls -alp .git
git status
```

By default the repository

- is empty (has no commits),
- is stored in `ds1004/.git`,
- includes one branch, named `master`

Adding a File to the Repository

- Each file in `ds1004` is *tracked* or *untracked* by the repository.
- If a file is tracked, then it is either *modified* or *unmodified*.

Add a file to the repository and make the initial commit:

```
echo "This is a test." > README; git status
git add README; git status
git commit
```

Enter a commit message: `1st commit (1st on master).`

```
git status
```

Check the log:

```
git log --decorate
```

`HEAD` indicates the commit most recently checked out.

The Git Index ("Staging Area")

The *index* is the list of all tracked files in the current branch;

- Kept in `.git/index`, viewable with `git ls-files`
- Serves as *staging area* of changes for next commit

```
echo "Making a change." >> README; git status
git add README; git status
echo "Making another change." >> README; git status
```

Note the unstaged changes: snapshot is out-of-date.

```
git add README; git status
```

To unstage this change (without affecting `README`):

```
git reset HEAD README; git status
```

Renaming and Deleting Files

Use `git mv` and `git rm` to rename and delete tracked files

```
touch TEST1 TEST2; ls
git add TEST1 TEST2; git status
git mv TEST2 TEST; git status
git rm --cached TEST1; git status
ls
```

Use to rename a tracked file

- `git mv` modifies both index and working directory
- `git rm` only modifies index
- `git rm` fails if uncommitted changes (override with `--cached`)

Avoid using POSIX utilities `mv` or `rm` on tracked files:

```
rm TEST; git status
git rm TEST; git status
```

Referencing Commits

Commit name: 40-character string over $\{0, \ldots, 9\} \cup \{a, \ldots, f\}$

- Encodes 160-bit SHA-1 message digest in hexadecimal

In commands, a reference `ref` to a commit can be

- leading 4+ characters of commit name (to uniquely identify it)
- `HEAD`: automatic reference to commit most recently checked out
- branch name: the tip of that branch

```
git show
git show ref
git show HEAD
```

Syntax for backtracing the history diagram, starting from `ref`:

- `ref^`n = `ref`'s n-th parent, numbered by merge order
- `ref^0` = `ref`, $n = 1$ default (when n omitted)
- `ref~`n = `ref^^`$\cdots$`^` (n times)
- Operations can be composed (left-associative)

Recovering Previous Commits

Check out commit `ref`'s version of `file` with `git checkout ref file`:

```
echo "A mistake." >> README; cat README
git checkout HEAD README; cat README
```

Warning: `git checkout` overwrites your (uncommited) changes!

To check out commit `ref`, use `git checkout ref`:

```
echo "Yet another change." >> README; git add README
git commit -m "2nd commit (2nd on master)."
git checkout HEAD~; git status
```

Use `git checkout` to get out of detached `HEAD` mode:

```
git checkout master
```

Using `HEAD` wouldn't help, so we used our previous branch name

Further Reading: Committing and Recovery

See the documentation to learn about the following features:

- `git init --bare`
- `git rm -f`
- `git reset --hard`
- `git commit --amend`
- `git tag`
- `git revert`

Creating Branches

Branching means having independent paths of development:

- Branches allow for 'sandboxed' experimentation with project.
- Common uses: developing/testing new features and bug fixes.
- Branches instrumental to Git's *distributed* nature.

Create a new branch, `testing`, that originates from `HEAD`:

```
git branch
git branch testing
git branch
git log --decorate --oneline --branches
```

Growing Branches

Check out `testing` and grow the branch by one commit:

```
git checkout testing
git log --decorate --oneline --branches
touch TEST; git add TEST
git commit -m "3rd commit (1st on testing)."
git log --decorate --oneline --branches
```

Merging

Merging combines two or more branches' tips.

- Hardest part of version control, big source of hard-to-find bugs.
- Manual conflict resolution: human compares and combines by hand.
- Automatic conflict resolution: Git has powerful heuristics to decide "best" merge, falling back on manual approach upon failure.

We will use the following command to draw the history diagram:

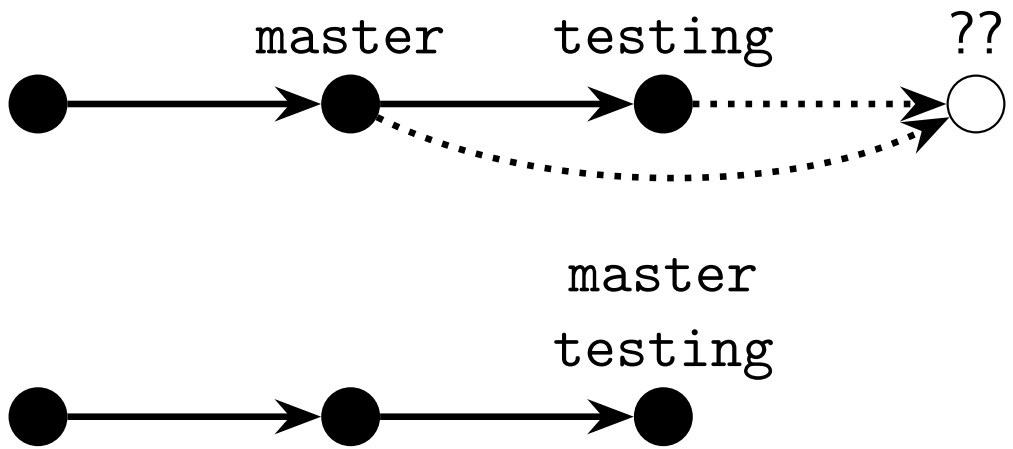
```
git config alias.graph \
     'log --decorate --oneline --branches --graph'
git graph
```

Merge Ex. 1: Fast-Forward Merge

Git will avoid a *true merge* if there's a path between tips:

```
git checkout master
git graph
git merge testing
git graph
```

Fast-forward `master`: not a true merge since no new commit.



Merge Ex. 2: True Merge (Automatic Resolution Succeeds)

```
git checkout testing; echo "Words words words" >> TEST
git commit -a -m "4th commit (2nd on testing)."
git checkout master; git add TEST1
git commit -m "5th commit (3rd on master)."; git graph
```

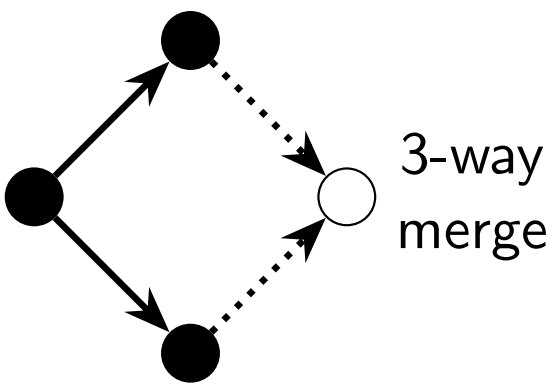
The paths have diverged (can't fast-forward), and a *true merge* is required:

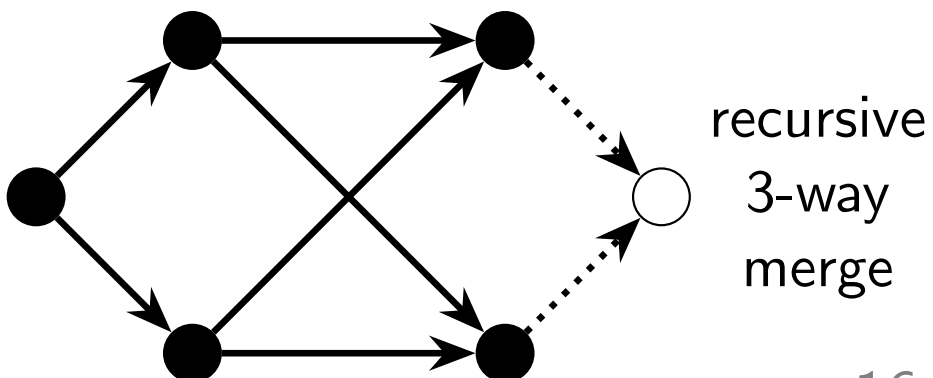
- `TEST` has words in `testing` but is empty in `master`,
- `TEST1` is tracked by `master` but not by `testing`.

Fortunately, Git's *3-way merge* can resolve this scenario automatically:

```
git merge testing -m "6th commit (4th on master)."; git graph
```

Observe the new commit — this was a true merge.





Merge Ex. 3: True Merge (Automatic Resolution Fails)

Automatic merging often fails due to merge conflicts.

```
echo "master change" >> TEST
git commit -a -m "7th commit (5th on master)."
git checkout testing
echo "testing change" >> TEST
git commit -a -m "8th commit (3rd on testing)."
git merge master
git status
git diff --diff-filter=U
```

Git doesn't know how best to merge these changes:

- the same line was changed in two different ways.

Merge Ex. 3: True Merge (Manual Resolution)

Edit `TEST` using your preferred text editor (e.g., `vim`):

```
vim TEST
```

Git replaces the conflicted lines by patterns like:

```
<<<<<<< HEAD
line(s) from the 'base' commit (HEAD, the tip of testing)
=======
line(s) from the 'incoming' commit (the tip of master)
>>>>>>> master
```

Resolve the merge as you see fit (!!); save and quit.

```
git status;
git add TEST; git status
git commit -m "9th commit (4th on testing)."; git status
git graph
```

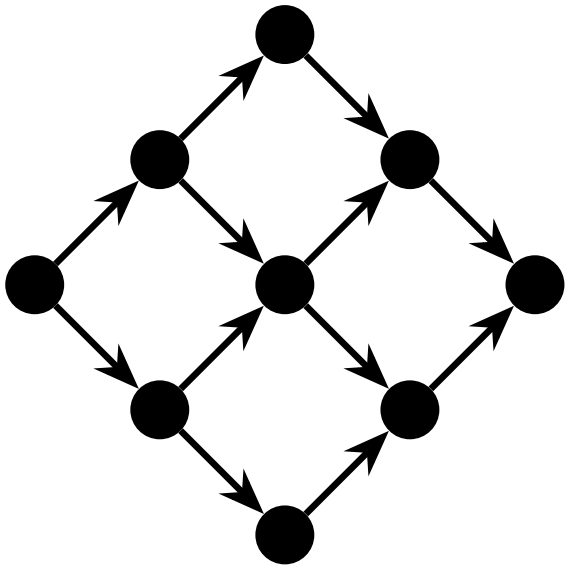
Further Reading: Branching and Merging

See the documentation to learn about the following features:

- `git checkout -b`
- `git branch -m`
- `git branch -d`
- `git branch --track`
- `git merge --no-ff`
- `git merge -s`
- `git mergetool`
- `git rebase`
- `git rebase --onto`

Task 1: Branching and Merging

Create a repository with the following history diagram:



Deliverable:

- Save your history diagram to a file (`NetID` is your NetID):

```
echo "NetID" > NetID.log
git log --graph --oneline --branches >> NetID.log
```

- You will submit this file as part of Task 2.

Remote Repositories and Branches

A Git project can be distributed across many repositories.

- Each repository stores a subset of the commits.
- The repository on your machine is your *local*.
- The repositories on other machines are your *remotes*.
- Synchronization is accomplished via *tracking branches*.

Each repository has its own branches:

- *remote branches* track branches on remote repositories
- *local branches* are the other branches

Migrating to GitHub

Go to GitHub and create a new repo; locate and copy the remote URL.

```
git remote add origin remote-URL
git remote -v
```

The default remote is called `origin` by convention, but any name is OK.

Mirror local on `origin` (requires your GitHub credentials):

```
git push --mirror origin
git branch -a -vv
```

Note how `git push --mirror` has set up remote tracking branches.

Synchronizing with Remotes

Make a local change and update remote (we are currently on `testing`):

```
echo "rewrite" > TEST; git status
git commit -a -m "10th commit (5th on testing)."; git status
git push origin testing:testing
```

To update local with updates from remote (if any):

```
git fetch origin
git merge origin/testing
```

Save typing by configuring upstream branches:

```
git branch -u origin/testing testing
git branch -u origin/master master
git branch -a -vv
```

Now you can do:

```
git fetch
git merge
git push
```

Cloning from GitHub

Use `git clone` to make a copy of an existing repository:

```
cd ..; mkdir ds1004_clone;
git clone remote-URL ds1004_clone
cd ds1004_clone; ls
git status
git remote -v
git branch -a -vv
git log --decorate --oneline --branches --graph
```

By default `git clone`

- copies all the commits and stores them locally,
- creates a remote (called `origin`),
- configures remote tracking branches,
- configures upstream branches.

Further Reading: Working With Remotes

See the documentation to learn about the following features:

- `git pull`
- `git pull --rebase`
- `git clone --bare`
- `git clone --mirror`

Task 2: GitHub Collaboration (Slide 1 of 2)

Details on forking and pull requests:
`http://guides.github.com/activities/forking/`

1. **Group up in teams of three or more.**
2. Select teammate A to host your repo.
3. Fork the class repo — `https://github.com/ds1004-sp16/ds1004` via Teammate A's GitHub console; add teammates as collaborators.
4. Clone the fork onto your local machine.
5. Add your `NetID.log` file in the top-level directory and commit.

Wait here until all teammates have caught up.

6. Push to remote. (You may have to fetch, merge, and retry.)

Wait here until all teammates have caught up.

Task 2: GitHub Collaboration (Slide 2 of 2)

7. Fetch and merge, so you now have all teammates' log-files.
8. Copy the contents of your log-file into each of the other log-files, above or below the existing contents, depending on whether your NetID is before or after theirs in alphabetical ordering. Commit.

Wait here until all teammates have caught up.

9. Push to remote. (You may have to fetch, merge, and retry.)

Wait here until all teammates have caught up.

10. All log-files should now have identical contents — verify with `diff`. Fix any issues before proceeding, making sure to update remote.
11. **Deliverable**: Submit a pull request (to the class repo) based on the last push made to your fork, using Teammate A's GitHub console. The pull request message must list all teammates' NetIDs. Individually submit your fork's URL via NYU Classes Assignments.