GraVE Documentation

Release

GraVE developers

Mar 31, 2018

Contents:

1 The GraVE API **1**
- 1.1 Basic Plotting . . . 1
- 1.2 Styling your plot . . . 1

2 Developer Guide **3**
- 2.1 Developer overview . . . 3
- 2.2 Working with *grave* source code . . . 5

3 Gallery **23**
- 3.1 A dead simple network . . . 23
- 3.2 GraVE Documentation . . . 24
- 3.3 GraVE Documentation . . . 25
- 3.4 GraVE Documentation . . . 26
- 3.5 Using another style . . . 27
- 3.6 GraVE Documentation . . . 28
- 3.7 Using a custom layout . . . 29
- 3.8 GraVE Documentation . . . 30
- 3.9 Different layouts . . . 30
- 3.10 Coloring the degrees of a node . . . 32
- 3.11 GraVE Documentation . . . 33
- 3.12 Labeled 2D Grid . . . 34
- 3.13 Interactively highlight nodes and edges . . . 36
- 3.14 Cities . . . 37

4 Notes from GraphXD sprints **41**
- 4.1 From discussions with Nelle Varoquaux, Aric Hagberg, and Dan Schult . . . 41
- 4.2 From discussion with large group of network practitioners . . . 41
- 4.3 initial target cases . . . 42

5 Indices and tables **43**

CHAPTER 1

The GraVE API

1.1 Basic Plotting

plot_network(graph[, layout, node_style, ...])	Plot network
style_merger(*funcs)	

1.1.1 grave.grave.plot_network

grave.grave.**plot_network**(*graph*, *layout='spring'*, *node_style=None*, *edge_style=None*, *node_label_style=None*, *edge_label_style=None*, ***, *ax*)

Plot network

Parameters **graph** (*networkx graph object*) –

Examples using grave.grave.plot_network

- *A dead simple network*
- sphx_glr_gallery_color_dominators.py
- *GraVE Documentation*

1.1.2 grave.grave.style_merger

1.2 Styling your plot

apply_style(style, item_iterable)	
generate_node_styles(network, node_style)	
generate_edge_styles(network, edge_style)	

1.2.1 grave.style.apply_style

grave.style.**apply_style**(*style*, *item_iterable*, *default*)

1.2.2 grave.style.generate_node_styles

grave.style.**generate_node_styles**(*network*, *node_style*)

1.2.3 grave.style.generate_edge_styles

grave.style.**generate_edge_styles**(*network*, *edge_style*)

CHAPTER 2

Developer Guide

2.1 Developer overview

1. If you are a first-time contributor:
   - Go to https://github.com/networkx/grave and click the "fork" button to create your own copy of the project.
   - Clone the project to your local computer:

     ```
     git clone git@github.com:your-username/grave.git
     ```

   - Add the upstream repository:

     ```
     git remote add upstream git@github.com:networkx/grave.git
     ```

   - Now, you have remote repositories named:
     - `upstream`, which refers to the `networkx/grave` repository
     - `origin`, which refers to your personal fork
2. Develop your contribution:
   - Pull the latest changes from upstream:

     ```
     git checkout master
     git pull upstream master
     ```

   - Create a branch for the feature you want to work on. Since the branch name will appear in the merge message, use a sensible name such as 'bugfix-for-issue-1480':

     ```
     git checkout -b bugfix-for-issue-1480
     ```

   - Commit locally as you progress (`git add` and `git commit`)
3. To submit your contribution:

- Push your changes back to your fork on GitHub:

```
git push origin bugfix-for-issue-1480
```

- Go to GitHub. The new branch will show up with a green Pull Request button—click it.
- If you want, post on the mailing list to explain your changes or to ask for review.

For a more detailed discussion, read these *detailed documents* on how to use Git with `grave` (http://grave.readthedocs.io/en/latest/developer/gitwash/index.html).

4. Review process:
   - Reviewers (the other developers and interested community members) will write inline and/or general comments on your Pull Request (PR) to help you improve its implementation, documentation, and style. Every single developer working on the project has their code reviewed, and we've come to see it as friendly conversation from which we all learn and the overall code quality benefits. Therefore, please don't let the review discourage you from contributing: its only aim is to improve the quality of project, not to criticize (we are, after all, very grateful for the time you're donating!).
   - To update your pull request, make your changes on your local repository and commit. As soon as those changes are pushed up (to the same branch as before) the pull request will update automatically.
   - Travis-CI, a continuous integration service, is triggered after each Pull Request update to build the code and run unit tests of your branch. The Travis tests must pass before your PR can be merged. If Travis fails, you can find out why by clicking on the "failed" icon (red cross) and inspecting the build and test log.
   - AppVeyor, is another continuous integration service, which we use. You will also need to make sure that the AppVeyor tests pass.

Note: If closing a bug, also add "Fixes #1480" where 1480 is the issue number.

2.1.1 Divergence between `upstream master` and your feature branch

Never merge the main branch into yours. If GitHub indicates that the branch of your Pull Request can no longer be merged automatically, rebase onto master:

```
git checkout master
git pull upstream master
git checkout bugfix-for-issue-1480
git rebase master
```

If any conflicts occur, fix the according files and continue:

```
git add conflict-file1 conflict-file2
git rebase --continue
```

However, you should only rebase your own branches and must generally not rebase any branch which you collaborate on with someone else.

Finally, you must push your rebased branch:

```
git push --force origin bugfix-for-issue-1480
```

(If you are curious, here's a further discussion on the dangers of rebasing. Also see this LWN article.)

2.1.2 Guidelines

- All code should have tests.
- All code should be documented, to the same standard as NumPy and SciPy.
- For new functionality, always add an example to the gallery.
- All changes are reviewed. Ask on the mailing list if you get no response to your pull request.

2.1.3 Stylistic Guidelines

- Set up your editor to remove trailing whitespace. Follow PEP08. Check code with *pyflakes* / *flake8*.
- Use the following import conventions:

```
import numpy as np
import scipy as sp
import matplotlib as mpl
import matplotlib.pyplot as plt
import networkx as nx
import grave as gve

cimport numpy as cnp # in Cython code
```

2.1.4 Pull request codes

When you submit a pull request to GitHub, GitHub will ask you for a summary. If your code is not ready to merge, but you want to get feedback, please consider using `WIP: experimental optimization` or similar for the title of your pull request. That way we will all know that it's not yet ready to merge and that you may be interested in more fundamental comments about design.

When you think the pull request is ready to merge, change the title (using the *Edit* button) to remove the `WIP:`.

2.1.5 Bugs

Please report bugs on GitHub.

2.2 Working with *grave* source code

Contents:

2.2.1 Introduction

These pages describe a git and github workflow for the grave project.

There are several different workflows here, for different ways of working with *grave*.

This is not a comprehensive git reference, it's just a workflow for our own project. It's tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

For general resources for learning git, see *git resources*.

2.2.2 Install git

Overview

Debian / Ubuntu	`sudo apt-get install git`
Fedora	`sudo dnf install git`
Windows	Download and install msysGit
OS X	Use the git-osx-installer

In detail

See the git page for the most recent information.

Have a look at the github install help pages available from github help

There are good instructions here: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

2.2.3 Following the latest source

These are the instructions if you just want to follow the latest *grave* source, but you don't need to do any development for now.

The steps are:

- *Install git*
- get local copy of the grave github git repository
- update local copy from time to time

Get the local copy of the code

From the command line:

```
git clone git://github.com/grave/grave.git
```

You now have a copy of the code tree in the new `grave` directory.

Updating the code

From time to time you may want to pull down the latest code. Do this with:

```
cd grave
git pull
```

The tree in `grave` will now have the latest changes from the initial repository.

2.2.4 Making a patch

You've discovered a bug or something else you want to change in grave .. — excellent!

You've worked out a way to fix it — even better!

You want to tell us about it — best of all!

The easiest way is to make a *patch* or set of patches. Here we explain how. Making a patch is the simplest and quickest, but if you're going to be doing anything more than simple quick things, please consider following the *Git for development* model instead.

Making patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/grave/grave.git
# make a branch for your patching
cd grave
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
# make the patch files
git format-patch -M -C master
```

Then, send the generated patch files to the grave mailing list — where we will thank you warmly.

In detail

1. Tell git who you are so it can label the commits you've made:

   ```
   git config --global user.email you@yourdomain.example.com
   git config --global user.name "Your Name Comes Here"
   ```

2. If you don't already have one, clone a copy of the grave repository:

   ```
   git clone git://github.com/grave/grave.git
   cd grave
   ```

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

   ```
   git branch the-fix-im-thinking-of
   git checkout the-fix-im-thinking-of
   ```

4. Do some edits, and commit them as you go:

   ```
   # hack, hack, hack
   # Tell git about any new files you've made
   git add somewhere/tests/test_my_bug.py
   # commit work in progress as you go
   ```

```
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag?.

5. When you have finished, check you have committed all your changes:

```
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

```
git format-patch -M -C master
```

You will now have several files named for the commits:

```
0001-BF-added-tests-for-Funny-bug.patch
0002-BF-added-fix-for-Funny-bug.patch
```

Send these files to the grave mailing list.

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

Moving from patching to development

If you find you have done some patches, and you have one or more feature branches, you will probably want to switch to development mode. You can do this with the repository you have.

Fork the grave repository on github — *Making your own copy (fork) of grave*. Then:

```
# checkout and refresh master branch from main repo
git checkout master
git pull origin master
# rename pointer to main repository to 'upstream'
git remote rename origin upstream
# point your repo to default read / write to your fork on github
git remote add origin git@github.com:your-user-name/grave.git
# push up any branches you've made and want to keep
git push origin the-fix-im-thinking-of
```

Then you can, if you want, follow the *Development workflow*.

2.2.5 Git for development

Contents:

Making your own copy (fork) of grave

You need to do this only once. The instructions here are very similar to the instructions at https://help.github.com/forking/ — please see that page for more detail. We're repeating some of it here just to give the specifics for the grave project, and to suggest some default names.

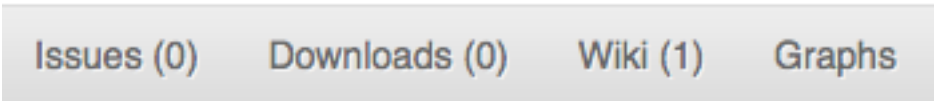
Set up and configure a github account

If you don't have a github account, go to the github page, and make one.

You then need to configure your account to allow write access — see the `Generating SSH keys` help on github help.

Create your own forked copy of grave

1. Log into your github account.
2. Go to the grave github home at grave github.
3. Click on the *fork* button:



Now, after a short pause, you should find yourself at the home page for your own forked copy of grave.

Set up your fork

First you follow the instructions for *Making your own copy (fork) of grave*.

Overview

```
git clone git@github.com:your-user-name/grave.git
cd grave
git remote add upstream git://github.com/grave/grave.git
```

In detail

Clone your fork

1. Clone your fork to the local computer with `git clone git@github.com:your-user-name/grave.git`
2. Investigate. Change directory to your new repo: `cd grave`. Then `git branch -a` to show you all branches. You'll get something like:

```
* master
remotes/origin/master
```

This tells you that you are currently on the `master` branch, and that you also have a `remote` connection to `origin/master`. What remote repository is `remote/origin`? Try `git remote -v` to see the URLs for the remote. They will point to your github fork.

Now you want to connect to the upstream grave github repository, so you can merge in changes from trunk.

Linking your repository to the upstream repo

```
cd grave
git remote add upstream git://github.com/grave/grave.git
```

`upstream` here is just the arbitrary name we're using to refer to the main grave repository at grave github.

Note that we've used `git://` for the URL rather than `git@`. The `git://` URL is read only. This means we that we can't accidentally (or deliberately) write to the upstream repo, and we are only going to use it to merge into our own code.

Just for your own satisfaction, show yourself that you now have a new 'remote', with `git remote -v show`, giving you something like:

```
upstream     git://github.com/grave/grave.git (fetch)
upstream     git://github.com/grave/grave.git (push)
origin       git@github.com:your-user-name/grave.git (fetch)
origin       git@github.com:your-user-name/grave.git (push)
```

Configure git

Overview

Your personal git configurations are saved in the `.gitconfig` file in your home directory.

Here is an example `.gitconfig` file:

```
[user]
        name = Your Name
        email = you@yourdomain.example.com

[alias]
        ci = commit -a
        co = checkout
        st = status
        stat = status
        br = branch
        wdiff = diff --color-words

[core]
        editor = vim

[merge]
        summary = true
```

You can edit this file directly or you can use the `git config --global` command:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
git config --global core.editor vim
git config --global merge.summary true
```

To set up on another computer, you can copy your `~/.gitconfig` file, or run the commands above.

In detail

user.name and user.email

It is good practice to tell git who you are, for labeling any changes you make to the code. The simplest way to do this is from the command line:

```
git config --global user.name "Your Name"
git config --global user.email you@yourdomain.example.com
```

This will write the settings into your git configuration file, which should now contain a user section with your name and email:

```
[user]
      name = Your Name
      email = you@yourdomain.example.com
```

Of course you'll need to replace `Your Name` and `you@yourdomain.example.com` with your actual name and email address.

Aliases

You might well benefit from some aliases to common commands.

For example, you might well want to be able to shorten `git checkout` to `git co`. Or you may want to alias `git diff --color-words` (which gives a nicely formatted output of the diff) to `git wdiff`

The following `git config --global` commands:

```
git config --global alias.ci "commit -a"
git config --global alias.co checkout
git config --global alias.st "status -a"
git config --global alias.stat "status -a"
git config --global alias.br branch
git config --global alias.wdiff "diff --color-words"
```

will create an `alias` section in your `.gitconfig` file with contents like this:

```
[alias]
        ci = commit -a
        co = checkout
        st = status -a
        stat = status -a
        br = branch
        wdiff = diff --color-words
```

Editor

You may also want to make sure that your editor of choice is used

```
git config --global core.editor vim
```

Merging

To enforce summaries when doing merges (`~/.gitconfig` file again):

```
[merge]
   log = true
```

Or from the command line:

```
git config --global merge.log true
```

Fancy log output

This is a very nice alias to get a fancy log output; it should go in the `alias` section of your `.gitconfig` file:

```
lg = log --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr)
↪%C(bold blue)[%an]%Creset' --abbrev-commit --date=relative
```

You use the alias with:

```
git lg
```

and it gives graph / text output something like this (but with color!):

```
* 6d8e1ee - (HEAD, origin/my-fancy-feature, my-fancy-feature) NF - a fancy file (45␣
↪minutes ago) [Matthew Brett]
*   d304a73 - (origin/placeholder, placeholder) Merge pull request #48 from hhuuggoo/
↪master (2 weeks ago) [Jonathan Terhorst]
|\
| * 4aff2a8 - fixed bug 35, and added a test in test_bugfixes (2 weeks ago) [Hugo]
|/
* a7ff2e5 - Added notes on discussion/proposal made during Data Array Summit. (2␣
↪weeks ago) [Corran Webster]
* 68f6752 - Initial implimentation of AxisIndexer - uses 'index_by' which needs to be␣
↪changed to a call on an Axes object - this is all very sketchy right now. (2 weeks␣
↪ago) [Corr
*   376adbd - Merge pull request #46 from terhorst/master (2 weeks ago) [Jonathan␣
↪Terhorst]
|\
| * b605216 - updated joshu example to current api (3 weeks ago) [Jonathan Terhorst]
| * 2e991e8 - add testing for outer ufunc (3 weeks ago) [Jonathan Terhorst]
| * 7beda5a - prevent axis from throwing an exception if testing equality with non-
↪axis object (3 weeks ago) [Jonathan Terhorst]
| * 65af65e - convert unit testing code to assertions (3 weeks ago) [Jonathan␣
↪Terhorst]
| *   956fbab - Merge remote-tracking branch 'upstream/master' (3 weeks ago)␣
↪[Jonathan Terhorst]
| |\
| |/
```

Thanks to Yury V. Zaytsev for posting it.

Development workflow

You already have your own forked copy of the grave repository, by following *Making your own copy (fork) of grave*. You have *Set up your fork*. You have configured git by following *Configure git*. Now you are ready for some real work.

Workflow summary

In what follows we'll refer to the upstream grave `master` branch, as "trunk".

- Don't use your `master` branch for anything. Consider deleting it.
- When you are starting a new set of changes, fetch any changes from trunk, and start a new *feature branch* from that.
- Make a new branch for each separable set of changes — "one task, one branch" (ipython git workflow).
- Name your branch for the purpose of the changes - e.g. `bugfix-for-issue-14` or `refactor-database-code`.
- If you can possibly avoid it, avoid merging trunk or any other branches into your feature branch while you are working.
- If you do find yourself merging from trunk, consider *Rebasing on trunk*
- Ask on the grave mailing list if you get stuck.
- Ask for code review!

This way of working helps to keep work well organized, with readable history. This in turn makes it easier for project maintainers (that might be you) to see what you've done, and why you did it.

See linux git workflow and ipython git workflow for some explanation.

Consider deleting your master branch

It may sound strange, but deleting your own `master` branch can help reduce confusion about which branch you are on. See deleting master on github for details.

Update the mirror of trunk

First make sure you have done *Linking your repository to the upstream repo*.

From time to time you should fetch the upstream (trunk) changes from github:

```
git fetch upstream
```

This will pull down any commits you don't have, and set the remote branches to point to the right commit. For example, 'trunk' is the branch referred to by (remote/branchname) `upstream/master` - and if there have been commits since you last checked, `upstream/master` will change after you do the fetch.

Make a new feature branch

When you are ready to make some changes to the code, you should start a new branch. Branches that are for a collection of related edits are often called 'feature branches'.

Making an new branch for each set of related changes will make it easier for someone reviewing your branch to see what you are doing.

Choose an informative name for the branch to remind yourself and the rest of us what the changes in the branch are for. For example `add-ability-to-fly`, or `buxfix-for-issue-42`.

```
# Update the mirror of trunk
git fetch upstream
# Make new feature branch starting at current trunk
git branch my-new-feature upstream/master
git checkout my-new-feature
```

Generally, you will want to keep your feature branches on your public github fork of grave. To do this, you git push this new branch up to your github repo. Generally (if you followed the instructions in these pages, and by default), git will have a link to your github repo, called `origin`. You push up to your own repo on github with:

```
git push origin my-new-feature
```

In git >= 1.7 you can ensure that the link is correctly set by using the `--set-upstream` option:

```
git push --set-upstream origin my-new-feature
```

From now on git will know that `my-new-feature` is related to the `my-new-feature` branch in the github repo.

The editing workflow

Overview

```
# hack hack
git add my_new_file
git commit -am 'NF - some message'
git push
```

In more detail

1. Make some changes
2. See which files have changed with `git status` (see git status). You'll see a listing like this one:

```
# On branch ny-new-feature
# Changed but not updated:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#  modified:   README
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#  INSTALL
no changes added to commit (use "git add" and/or "git commit -a")
```

3. Check what the actual changes are with `git diff` (git diff).
4. Add any new files to version control `git add new_file_name` (see git add).

5. To commit all modified files into the local copy of your repo,, do `git commit -am 'A commit message'`. Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line. The `a` flag — you can just take on faith — or see why the -a flag? — and the helpful use-case description in the tangled working copy problem. The git commit manual page might also be useful.
6. To push the changes up to your forked repo on github, do a `git push` (see git push).

Ask for your changes to be reviewed or merged

When you are ready to ask for someone to review your code and consider a merge:

1. Go to the URL of your forked repo, say `https://github.com/your-user-name/grave`.
2. Use the 'Switch Branches' dropdown menu near the top left of the page to select the branch with your changes:

3. Click on the 'Pull request' button:

Enter a title for the set of changes, and some explanation of what you've done. Say if there is anything you'd like particular attention for - like a complicated change or some code you are not happy with.

If you don't think your request is ready to be merged, just say so in your pull request message. This is still a good way of getting some preliminary code review.

Some other things you might want to do

Delete a branch on github

```
git checkout master
# delete branch locally
git branch -D my-unwanted-branch
# delete branch on github
git push origin :my-unwanted-branch
```

Note the colon `:` before `my-unwanted-branch`. See also: https://help.github.com/articles/pushing-to-a-remote/#deleting-a-remote-branch-or-tag

Several people sharing a single repository

If you want to work on some stuff with other people, where you are all committing into the same repository, or even the same branch, then just share it via github.

First fork grave into your account, as from *Making your own copy (fork) of grave*.

Then, go to your forked repository github page, say `https://github.com/your-user-name/grave`

Click on the 'Admin' button, and add anyone else to the repo as a collaborator:

Now all those people can do:

```
git clone git@githhub.com:your-user-name/grave.git
```

Remember that links starting with `git@` use the ssh protocol and are read-write; links starting with `git://` are read-only.

Your collaborators can then commit directly into that repo with the usual:

```
git commit -am 'ENH - much better code'
git push origin master # pushes directly into your repo
```

Explore your repository

To see a graphical representation of the repository branches and commits:

```
gitk --all
```

To see a linear list of commits for this branch:

```
git log
```

You can also look at the network graph visualizer for your github repo.

Finally the *Fancy log output* `lg` alias will give you a reasonable text-based graph of the repository.

Rebasing on trunk

Let's say you thought of some work you'd like to do. You *Update the mirror of trunk* and *Make a new feature branch* called `cool-feature`. At this stage trunk is at some commit, let's call it E. Now you make some new commits on your `cool-feature` branch, let's call them A, B, C. Maybe your changes take a while, or you come back to them after a while. In the meantime, trunk has progressed from commit E to commit (say) G:

```
      A---B---C cool-feature
     /
D---E---F---G trunk
```

At this stage you consider merging trunk into your feature branch, and you remember that this here page sternly advises you not to do that, because the history will get messy. Most of the time you can just ask for a review, and not worry that trunk has got a little ahead. But sometimes, the changes in trunk might affect your changes, and you need to harmonize them. In this situation you may prefer to do a rebase.

rebase takes your changes (A, B, C) and replays them as if they had been made to the current state of `trunk`. In other words, in this case, it takes the changes represented by A, B, C and replays them on top of G. After the rebase, your history will look like this:

```
              A'--B'--C' cool-feature
             /
D---E---F---G trunk
```

See rebase without tears for more detail.

To do a rebase on trunk:

```
# Update the mirror of trunk
git fetch upstream
# go to the feature branch
git checkout cool-feature
# make a backup in case you mess up
git branch tmp cool-feature
# rebase cool-feature onto trunk
git rebase --onto upstream/master upstream/master cool-feature
```

In this situation, where you are already on branch `cool-feature`, the last command can be written more succinctly as:

```
git rebase upstream/master
```

When all looks good you can delete your backup branch:

```
git branch -D tmp
```

If it doesn't look good you may need to have a look at *Recovering from mess-ups*.

If you have made changes to files that have also changed in trunk, this may generate merge conflicts that you need to resolve - see the git rebase man page for some instructions at the end of the "Description" section. There is some related help on merging in the git user manual - see resolving a merge.

Recovering from mess-ups

Sometimes, you mess up merges or rebases. Luckily, in git it is relatively straightforward to recover from such mistakes.

If you mess up during a rebase:

```
git rebase --abort
```

If you notice you messed up after the rebase:

```
# reset branch back to the saved point
git reset --hard tmp
```

If you forgot to make a backup branch:

```
# look at the reflog of the branch
git reflog show cool-feature

8630830 cool-feature@{0}: commit: BUG: io: close file handles immediately
278dd2a cool-feature@{1}: rebase finished: refs/heads/my-feature-branch onto␣
↪11ee694744f2552d
26aa21a cool-feature@{2}: commit: BUG: lib: make seek_gzip_factory not leak gzip obj
...

# reset the branch to where it was before the botched rebase
git reset --hard cool-feature@{2}
```

Rewriting commit history

Note: Do this only for your own feature branches.

There's an embarrassing typo in a commit you made? Or perhaps the you made several false starts you would like the posterity not to see.

This can be done via *interactive rebasing*.

Suppose that the commit history looks like this:

```
git log --oneline
eadc391 Fix some remaining bugs
a815645 Modify it so that it works
2dec1ac Fix a few bugs + disable
13d7934 First implementation
6ad92e5 * masked is now an instance of a new object, MaskedConstant
29001ed Add pre-nep for a copule of structured_array_extensions.
...
```

and `6ad92e5` is the last commit in the `cool-feature` branch. Suppose we want to make the following changes:

- Rewrite the commit message for `13d7934` to something more sensible.
- Combine the commits `2dec1ac`, `a815645`, `eadc391` into a single one.

We do as follows:

```
# make a backup of the current state
git branch tmp HEAD
# interactive rebase
git rebase -i 6ad92e5
```

This will open an editor with the following text in it:

```
pick 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
pick a815645 Modify it so that it works
pick eadc391 Fix some remaining bugs
```

```
# Rebase 6ad92e5..eadc391 onto 6ad92e5
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

To achieve what we want, we will make the following changes to it:

```
r 13d7934 First implementation
pick 2dec1ac Fix a few bugs + disable
f a815645 Modify it so that it works
f eadc391 Fix some remaining bugs
```

This means that (i) we want to edit the commit message for `13d7934`, and (ii) collapse the last three commits into one. Now we save and quit the editor.

Git will then immediately bring up an editor for editing the commit message. After revising it, we get the output:

```
[detached HEAD 721fc64] FOO: First implementation
 2 files changed, 199 insertions(+), 66 deletions(-)
[detached HEAD 0f22701] Fix a few bugs + disable
 1 files changed, 79 insertions(+), 61 deletions(-)
Successfully rebased and updated refs/heads/my-feature-branch.
```

and the history looks now like this:

```
0f22701 Fix a few bugs + disable
721fc64 ENH: Sophisticated feature
6ad92e5 * masked is now an instance of a new object, MaskedConstant
```

If it went wrong, recovery is again possible as explained *above*.

Maintainer workflow

This page is for maintainers — those of us who merge our own or other peoples' changes into the upstream repository.

Being as how you're a maintainer, you are completely on top of the basic stuff in *Development workflow*.

The instructions in *Linking your repository to the upstream repo* add a remote that has read-only access to the upstream repo. Being a maintainer, you've got read-write access.

It's good to have your upstream remote have a scary name, to remind you that it's a read-write remote:

```
git remote add upstream-rw git@github.com:grave/grave.git
git fetch upstream-rw
```

Integrating changes

Let's say you have some changes that need to go into trunk (`upstream-rw/master`).

The changes are in some branch that you are currently on. For example, you are looking at someone's changes like this:

```
git remote add someone git://github.com/someone/grave.git
git fetch someone
git branch cool-feature --track someone/cool-feature
git checkout cool-feature
```

So now you are on the branch with the changes to be incorporated upstream. The rest of this section assumes you are on this branch.

A few commits

If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
git fetch upstream-rw
# rebase
git rebase upstream-rw/master
```

Remember that, if you do a rebase, and push that, you'll have to close any github pull requests manually, because github will not be able to detect the changes have already been merged.

A long series of commits

If there are a longer series of related commits, consider a merge instead:

```
git fetch upstream-rw
git merge --no-ff upstream-rw/master
```

The merge will be detected by github, and should close any related pull requests automatically.

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

Check the history

Now, in either case, you should check that the history is sensible and you have the right commits:

```
git log --oneline --graph
git log -p upstream-rw/master..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/master`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

Push to trunk

```
git push upstream-rw my-new-feature:master
```

This pushes the `my-new-feature` branch in this repository to the `master` branch in the `upstream-rw` repository.

2.2.6 git resources

Tutorials and summaries

- github help has an excellent series of how-to guides.
- The pro git book is a good in-depth book on git.
- A git cheat sheet is a page giving summaries of common commands.
- The git user manual
- The git tutorial
- git ready — a nice series of tutorials
- git magic — extended introduction with intermediate detail
- The git parable is an easy read explaining the concepts behind git.
- git foundation expands on the git parable.
- Fernando Perez' git page — Fernando's git page — many links and tips
- A good but technical page on git concepts
- git svn crash course: git for those of us used to subversion

Advanced git workflow

There are many ways of working with git; here are some posts on the rules of thumb that other projects have come up with:

- Linus Torvalds on git management
- Linus Torvalds on linux git workflow . Summary; use the git tools to make the history of your edits as clean as possible; merge from upstream edits as little as possible in branches where you are doing active development.

Manual pages online

You can get these on your own machine with (e.g) `git help push` or (same thing) `git push --help`, but, for convenience, here are the online manual pages for some common commands:

- git add
- git branch
- git checkout
- git clone
- git commit
- git config
- git diff
- git log

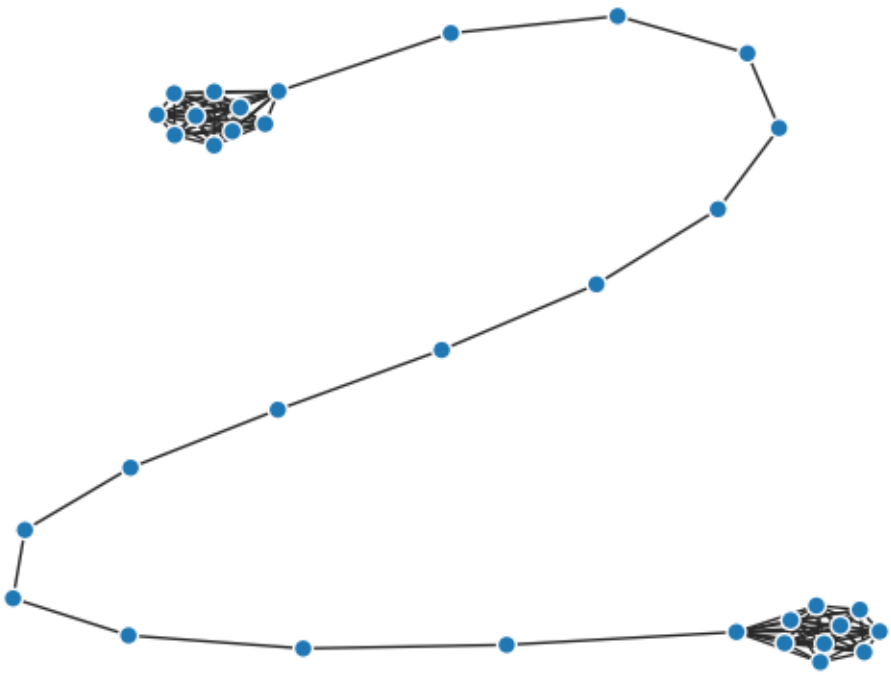
- git pull
- git push
- git remote
- git status

CHAPTER 3

Gallery

3.1 A dead simple network

The simplest way to plot a graphe ever. And yet it looks cool!



```
import networkx as nx
import matplotlib.pyplot as plt
from grave import plot_network

# Generating a networkx graph
graph = nx.barbell_graph(10, 14)

fig, ax = plt.subplots()
plot_network(graph, ax=ax)
plt.show()
```

Total running time of the script: (0 minutes 0.063 seconds)

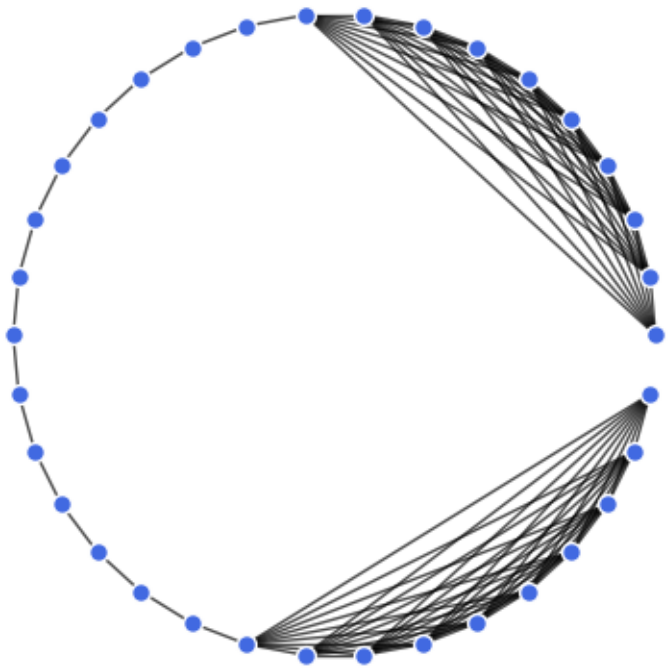
3.2 GraVE Documentation

```
import networkx as nx

toy_network = nx.barbell_graph(100, 10)


for node, node_attributes in toy_network.nodes(data=True):
    node['style'] = {'color': 'blue'}
```

```
plot_the_graph(toy_network,
               node_style=lambda attrs: attrs['style'])
```

Total running time of the script: (0 minutes 0.000 seconds)

3.3 GraVE Documentation

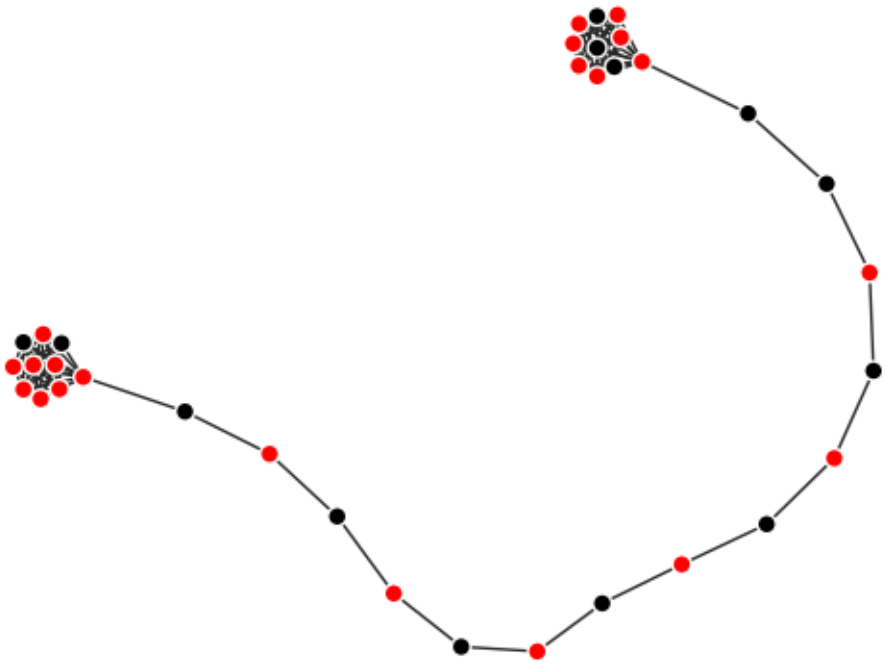


```
import networkx as nx
import matplotlib.pyplot as plt


toy_network = nx.barbell_graph(10, 14)


node_options = {
    'node_color': 'royalblue',
    'node_size': 50,
    'edgecolors': 'white',
}

edge_options = {
    'line_color': 'grey',
    'alpha': 0.7,
```

```
}

pos = nx.circular_layout(toy_network)
nx.draw_networkx_nodes(toy_network, pos, **node_options)
nx.draw_networkx_edges(toy_network, pos, **edge_options)
plt.axes().set_aspect('equal')
plt.axis('off')
```

Total running time of the script: (0 minutes 0.027 seconds)

3.4 GraVE Documentation



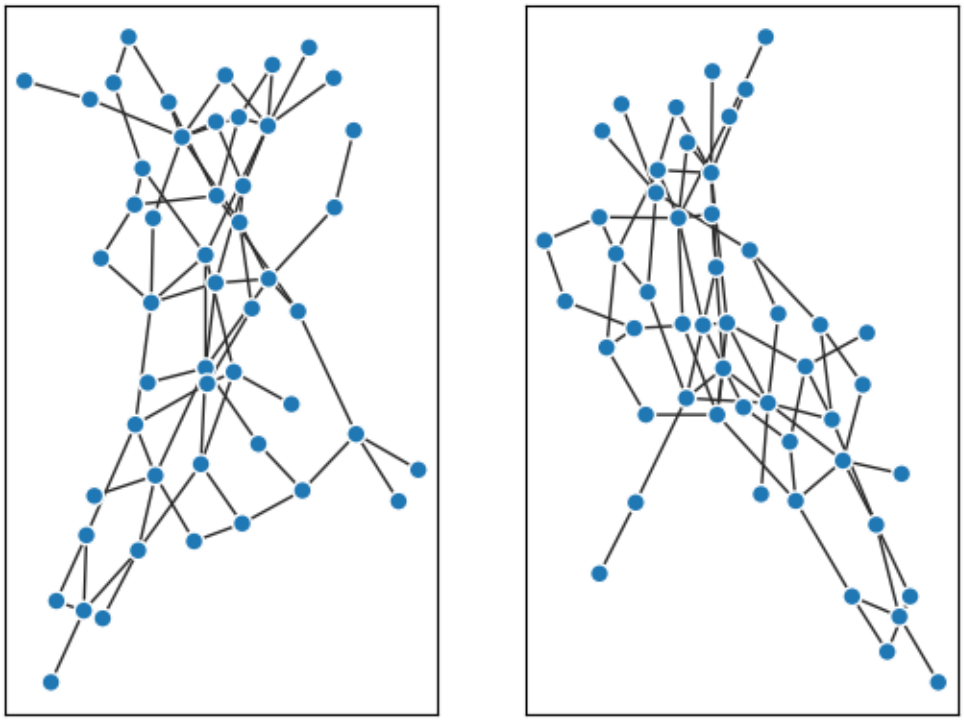
```
import networkx as nx
from networkx.algorithms.approximation.dominating_set import min_weighted_dominating_
↪set
import matplotlib.pyplot as plt

from grave import plot_network, use_attributes

toy_network = nx.barbell_graph(10, 14)
dom_set = min_weighted_dominating_set(toy_network)

for node, node_attrs in toy_network.nodes(data=True):
```

```
    if node in dom_set:
        node_attrs['color'] = 'red'
    else:
        node_attrs['color'] = 'black'
    node_attrs['size'] = 50

fig, ax = plt.subplots()
plot_network(toy_network,
             node_style=use_attributes())
plt.show()
```

Total running time of the script: (0 minutes 0.182 seconds)

3.5 Using another style

In this example, we show how to use another default Matplotlib style.



```
import networkx as nx
import matplotlib.pyplot as plt
from grave import plot_network

network = nx.binomial_graph(50, .05)
```

```
fig, ax_mat = plt.subplots(ncols=2)

plot_network(network, ax=ax_mat[0])
ax_mat[0].set_axis_on()
with plt.style.context(('ggplot')):
    plot_network(network, ax=ax_mat[1])

ax_mat[1].set_axis_on()
for ax in ax_mat:
    ax.set_axis_on()
    ax.tick_params(which='both',
                   bottom=False,
                   top=False,
                   left=False,
                   right=False,
                   labelbottom=False,
                   labelleft=False)
plt.show()
```

Total running time of the script: (0 minutes 0.117 seconds)

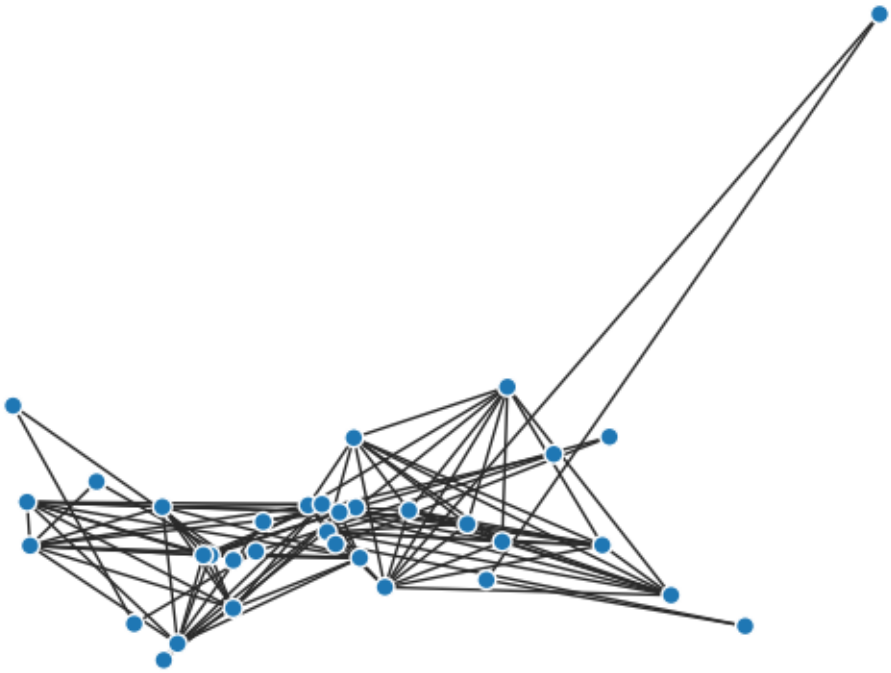
3.6 GraVE Documentation

```
import networkx as nx


toy_network = nx.barbell_graph(10, 14)

node_options = {
    'node_color': 'royalblue',
    'node_size': 50,
    'edgecolors': 'white',
}

edge_options = {
    'line_color': 'grey',
    'alpha': 0.7,
}

def protein_style(node_attributes):
    if node_attributes.get('type', '') == 'protein':
        return {'color': 'blue'}
    else:
        return {'color': 'red'}


plot_the_graph(toy_network,
               layout='spring',
               node_style=protein_style,
               edge_stlye=edge_options,
               node_labels=None,
               edge_labels=None,
               extra_artists=None)
```

Total running time of the script: (0 minutes 0.000 seconds)

3.7 Using a custom layout

The default layouts available through GraVE may not be sufficient for ones need. Hence, GraVE also support custom layouts.



```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
from grave import grave


graph = nx.barbell_graph(10, 14)
nx.set_node_attributes(graph, dict(graph.degree()), 'degree')


def random_constrained_layout(networkx):
    """
    Let's build my own layout. It's going to be random, except for a handful
    of points!
    """
    n_nodes = len(graph.nodes.data())
    random_state = np.random.RandomState(seed=0)
    xy = random_state.randn(n_nodes, 2)
    xy[0] = [0, 0]
    xy[10] = [+3, 8]
```

```
    return {k: xy[k] for k in graph.nodes.keys()}


fig, ax = plt.subplots()
grave.plot_network(graph, ax=ax, layout=random_constrained_layout)
```

Total running time of the script: (0 minutes 0.032 seconds)

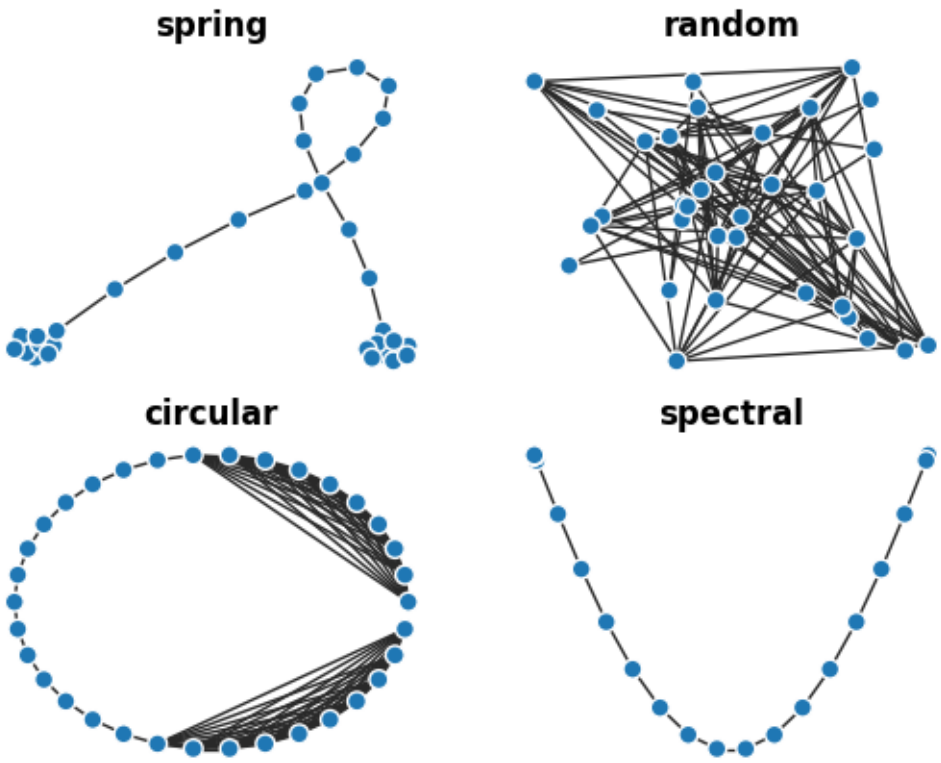
3.8 GraVE Documentation

```
import networkx as nx


toy_network = nx.barbell_graph(10, 14)

node_options = {
    'node_color': 'royalblue',
    'node_size': 50,
    'edgecolors': 'white',
}

edge_options = {
    'line_color': 'grey',
    'alpha': 0.7,
}

for node, node_attributes in toy_network.nodes(data=True):
    node_attributes['distance'] = my_compute(toy_network, node)


def protein_style(node_attributes):
    if node_attributes.get('type', '') == 'protein':
        return {'color': 'blue'}
    else:
        return {'color': 'red'}


plot_the_graph(toy_network,
               layout='spring',
               node_style=protein_style,
               edge_stlye=edge_options)
```

Total running time of the script: (0 minutes 0.000 seconds)

3.9 Different layouts

GraVE supports different layouts by default.

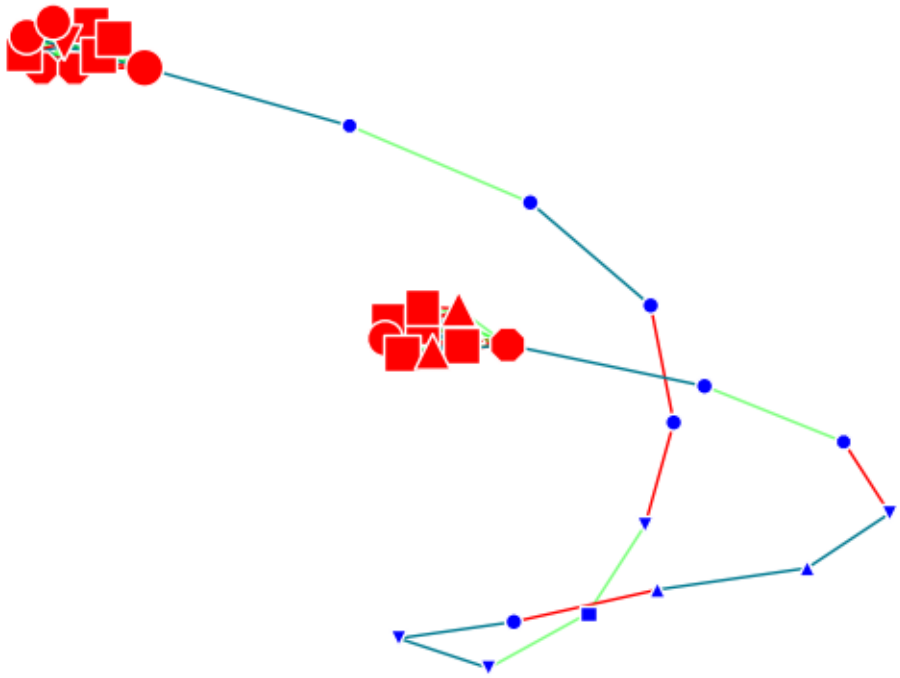



```
import networkx as nx
import matplotlib.pyplot as plt
from grave import grave


graph = nx.barbell_graph(10, 14)
nx.set_node_attributes(graph, dict(graph.degree()), 'degree')

fig, axes = plt.subplots(nrows=2, ncols=2)

grave.plot_network(graph, ax=axes[0, 0], layout="spring")
axes[0, 0].set_title("spring", fontweight="bold")

grave.plot_network(graph, ax=axes[1, 0], layout="circular")
axes[1, 0].set_title("circular", fontweight="bold")

grave.plot_network(graph, ax=axes[0, 1], layout="random")
axes[0, 1].set_title("random", fontweight="bold")

grave.plot_network(graph, ax=axes[1, 1], layout="spectral")
axes[1, 1].set_title("spectral", fontweight="bold")

plt.show()
```

Total running time of the script: (0 minutes 0.212 seconds)

3.10 Coloring the degrees of a node

Test

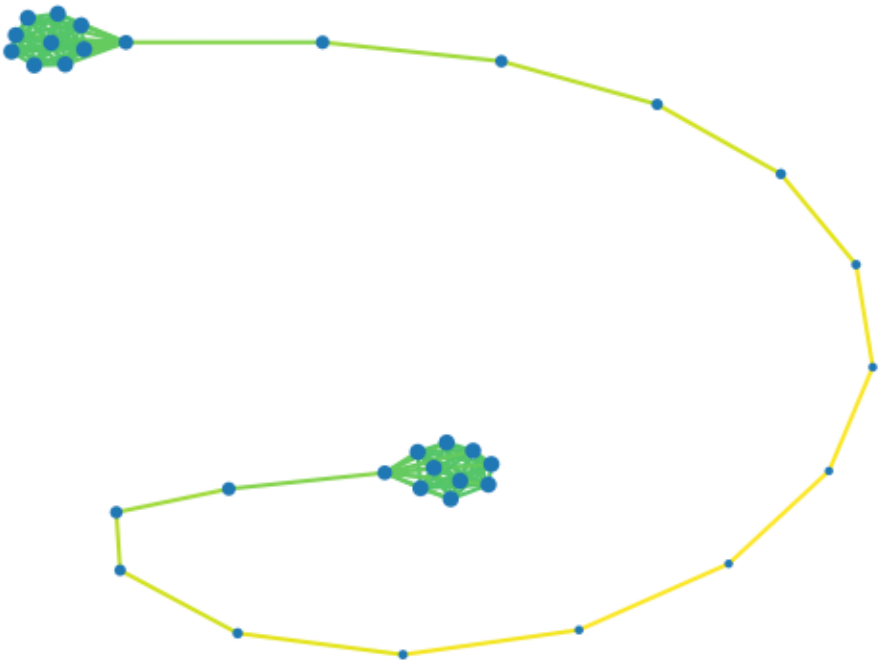


```
import networkx as nx
import matplotlib.pyplot as plt
import random
from grave import plot_network


graph = nx.barbell_graph(10, 14)

nx.set_node_attributes(graph, dict(graph.degree()), 'degree')


def degree_colorer(node_attributes):
    deg = node_attributes['degree']
    shape = random.choice(['s', 'o', '^', 'v', '8'])
    if deg > 5:
        return {'color': 'r', 'size': 20*deg, 'shape': shape}
    return {'color': 'b', 'size': 20*deg, 'shape': shape}


def pathological_edge_style(edge_attrs):
    return {'color': random.choice(['r', (0, 1, 0, .5), 'xkcd:ocean'])}
```

```
fig, ax = plt.subplots()
plot_network(graph, ax=ax, node_style=degree_colorer,
             edge_style=pathological_edge_style)
plt.show()
```

Total running time of the script: (0 minutes 0.050 seconds)

3.11 GraVE Documentation



```
import networkx as nx
from networkx.algorithms.centrality import closeness_centrality
import matplotlib.pyplot as plt

from grave import plot_network, use_attributes

toy_network = nx.barbell_graph(10, 14)
toy_centrality = closeness_centrality(toy_network)
max_centrality = max(toy_centrality.values())

for u, v, edge_attributes in toy_network.edges.data():
    c = (toy_centrality[u] +
         toy_centrality[v]) / 2
```

```
    color_idx = (c / max_centrality)
    cmap = plt.get_cmap()
    edge_attributes['color'] = cmap(color_idx)
    edge_attributes['width'] = 2

for node, node_attributes in toy_network.nodes.data():
    node_attributes['size'] = (1 - (toy_centrality[node] /
                                    max_centrality) + .1) * 100


def edge_style(edge_attributes):
    return {'linewidth': edge_attributes.get('weight', 1)}

fig, ax = plt.subplots()
plot_network(toy_network,
             layout='spring',
             node_style=use_attributes(),
             edge_style=use_attributes('color'))
plt.show()
```

Total running time of the script: (0 minutes 0.154 seconds)

3.12 Labeled 2D Grid

This example shows both labels and custom layout.


```
import networkx as nx
import matplotlib.pyplot as plt
import random
from grave import plot_network, style_merger



def degree_colorer(node_attributes):
    deg = node_attributes['degree']
    shape = 'o' #random.choice(['s', 'o', '^', 'v', '8'])
    return {'color': 'b', 'size': 20*deg, 'shape': shape}

def font_styler(attributes):
    return {'font_size': 8,
            'font_weight': .5,
            'font_color': 'k'}

def tiny_font_styler(attributes):
    return {'font_size': 4,
            'font_weight': .5,
            'font_color': 'r'}

def pathological_edge_style(edge_attrs):
    return {'color': random.choice(['r', (0, 1, 0, .5), 'xkcd:ocean'])}
```

```
network = nx.grid_2d_graph(4, 6)

nx.set_node_attributes(network, dict(network.degree()), 'degree')

fig, ax = plt.subplots()
plot_network(network, ax=ax, layout=lambda G: {node: node for node in G},
             node_style=degree_colorer,
             edge_style=pathological_edge_style,
             node_label_style=font_styler,
             edge_label_style=tiny_font_styler)

plt.show()
```

Total running time of the script: (0 minutes 0.078 seconds)

3.13 Interactively highlight nodes and edges

Run this with an interactive matplotlib backend!

Clicking on a node will hi-light it and it's edges

```
import networkx as nx
import matplotlib.pyplot as plt
from grave import plot_network
from grave.style import use_attributes


def hilighter(event):
    # if we did not hit a node, bail
    if not hasattr(event, 'nodes') or not event.nodes:
        return

    # pull out the graph,
    graph = event.artist.graph

    # clear any non-default color on nodes
    for node, attributes in graph.nodes.data():
        attributes.pop('color', None)

    for u, v, attributes in graph.edges.data():
        attributes.pop('width', None)

    for node in event.nodes:
        graph.nodes[node]['color'] = 'C1'

        for edge_attribute in graph[node].values():
            edge_attribute['width'] = 3

    # update the screen
    event.artist.stale = True
    event.artist.figure.canvas.draw_idle()


graph = nx.barbell_graph(10, 14)

fig, ax = plt.subplots()
```

```
art = plot_network(graph, ax=ax, node_style=use_attributes(),
                   edge_style=use_attributes())

art.set_picker(10)
ax.set_title('Click on the nodes!')
fig.canvas.mpl_connect('pick_event', hilighter)
plt.show()
```

Total running time of the script: (0 minutes 0.000 seconds)

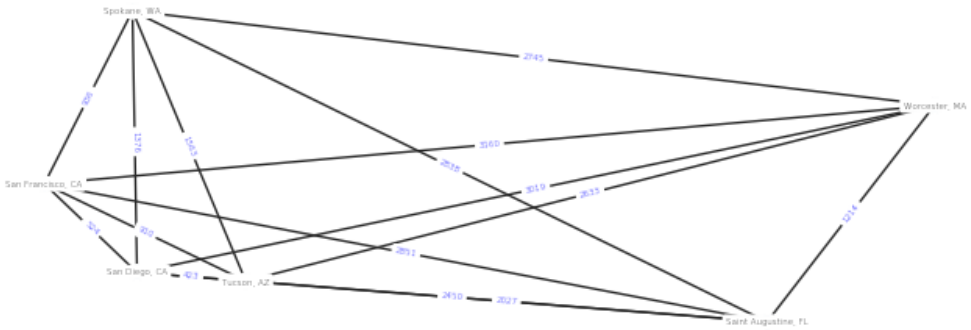
3.14 Cities

miles_graph() returns an undirected graph over the 128 US cities from the datafile *miles_dat.txt*. The cities each have location and population data. The edges are labeled with the distance between the two cities.

This example is described in Section 1.1 in Knuth's book (see[1] and[2]).

[1] Donald E. Knuth, "The Stanford GraphBase: A Platform for Combinatorial Computing", ACM Press, New York, 1993.
[2] http://www-cs-faculty.stanford.edu/~knuth/sgb.html

3.14.1 References.



Out:

```
Loaded miles_dat.txt containing 128 cities.
digraph has 128 nodes with 8128 edges
Subgraph has 6 nodes with 15 edges
['San Diego, CA', 'San Francisco, CA', 'Worcester, MA', 'Spokane, WA', 'Tucson, AZ',
↪'Saint Augustine, FL']
```

```
# Based on example from NetworkX

import re
import sys

import matplotlib.pyplot as plt
import networkx as nx
import grave


def miles_graph():
    """ Return the cites example graph in miles_dat.txt
        from the Stanford GraphBase.
    """
    # open file miles_dat.txt.gz (or miles_dat.txt)
    import gzip
    fh = gzip.open('knuth_miles.txt.gz', 'r')

    G = nx.Graph()
    G.position = {}
    G.population = {}

    cities = []
    for line in fh.readlines():
        line = line.decode()
        if line.startswith("*"):  # skip comments
            continue

        numfind = re.compile("^\d+")

        if numfind.match(line):  # this line is distances
            dist = line.split()
            for d in dist:
                G.add_edge(city, cities[i], weight=int(d))
                i = i + 1
        else:  # this line is a city, position, population
            i = 1
            (city, coordpop) = line.split("[")
            cities.insert(0, city)
            (coord, pop) = coordpop.split("]")
            (y, x) = coord.split(",")

            G.add_node(city)
            # assign position - flip x axis for matplotlib, shift origin
            G.position[city] = (-int(x) + 7500, int(y) - 3000)
            G.population[city] = float(pop) / 1000.0
    return G


if __name__ == '__main__':

    G = miles_graph()

    print("Loaded miles_dat.txt containing 128 cities.")
    print("digraph has %d nodes with %d edges"
          % (nx.number_of_nodes(G), nx.number_of_edges(G)))
    cities = ['San Diego, CA',
              'San Francisco, CA',
```

```
            'Saint Augustine, FL',
            'Spokane, WA',
            'Worcester, MA',
            'Tucson, AZ']

    # make subgraph of cities
    H = G.subgraph(cities)
    print("Subgraph has %d nodes with %d edges" % (len(H), H.size()))
    print(H.nodes)

    # draw with grave
    plt.figure(figsize=(8, 8))
    # create attribute for label
    nx.set_edge_attributes(H,
                           {e: G.edges[e]['weight'] for e in H.edges},
                           'label')

    # create stylers
    def transfer_G_layout(network):
        return {n: G.position[n] for n in network}

    def elabel_base_style(attr):
        return {'font_size': 4,
                'font_weight': .1,
                'font_family': 'sans-serif',
                'font_color': 'b',
                'rotate': True,  # TODO: make rotation less granular
                }

    elabel_style = grave.style_merger(grave.use_attributes('label'),
                                      elabel_base_style)

    grave.plot_network(H, transfer_G_layout,
                       node_style=dict(node_size=20),
                       edge_label_style=elabel_style,
                       node_label_style={})

    # scale the axes equally
    plt.xlim(-5000, 500)
    plt.ylim(-2000, 3500)

    plt.show()
```

Total running time of the script: (0 minutes 0.081 seconds)

CHAPTER 4

Notes from GraphXD sprints

Also see the example folder for concrete code

4.1 From discussions with Nelle Varoquaux, Aric Hagberg, and Dan Schult

- restrict to 'small' networks (few hundred to thousand)
- there are many choices in plotting mapping graph attributes -> visual properties
  - (x, y) layout
  - colors
  - labels
  - edges style
  - interactivity (hover / pick)
  - edge routing
- want to be able to update the plot in response to updating the network
- want to make it easily extensible
- performance?

4.2 From discussion with large group of network practitioners

- need "seaborn for networks" - heursitics for edge style
- also want "seaborn for network statistics"
- look at igraph - have lots of layout engines

- people tend to use different plots for data exploration vs publication
- functions for different on graph size
- input to layout engine should be flexible
- provide way to go to the bottom!

4.2.1 ball and edge

- per-vertex/edges labels
- per-vertex/edges annotation box (maybe?)
- per-vertex/edge artists / subplots - excited about pie charts - do this as roll-over / tool-tip
- mark sub-graphs
- per-node style
- per-edge style - directed edges - un-directed edges - multi-edges - arbitrary path

4.2.2 adjacency matrix view

- show adjacency matrix
- show anything at all in the gutters (all 8 places)

4.2.3 trees

- dendograms

4.2.4 sankey / flow

- this may be better interface to the existing sankey functionality in Matplotlib

4.3 initial target cases

- re-make jarod's slides - small (25 nodes)
- a 1000 node random graph - no labels, transparency, color per edge
- A tree of some sort
- splicing graph
- neural networks
- some flow network

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Index

A

apply_style() (in module grave.style), 2

G

generate_edge_styles() (in module grave.style), 2
generate_node_styles() (in module grave.style), 2

P

plot_network() (in module grave.grave), 1