Assignment 9. Change management spelunking and implementation

Useful pointers

- Scott Chacon, Pro Git, 2nd ed. (2014), §§2, 3, 7.1–7.8, 7.11, 10.1–10.5.

Laboratory: Git spelunking

Consider the Git repository and working files in the directory `~eggert/src/gnu/emacs-CS-35L` on the SEASnet GNU/Linux hosts. For this repository, answer the following questions, using Git commands and/or scripts of your own devising. For each question, record which commands you used; if you wrote and used a script, include a copy of the script.

1. How much disk space is used to represent the working files in this directory? How much is used to represent the Git repository? What file or files consume most of the repository space and why?
2. How many branches are local to the repository? How many are remote?
3. How many repositories are remote to this repository, and where are they?
4. What are the ten local branches most recently committed to? List their names in order of commit date, most recent first.
5. How many commits are in the master branch?
6. What percentage of the commits that are in any branch, are also in the master branch?
7. Which ten people authored the most master-branch commits that were committed in the year 2013 or later, and how many commits did each of them author during that period?
8. Use the `gitk` command to visualize the commit graph in this repository. If you are SSHing into SEASnet, you'll need to log in via `ssh -X` or (less securely) `ssh -Y`. Draw a diagram relating the following commits to each other, and explain what likely happened to cause their commit-graph neighborhood. You need not list every single intervening commit individually; you can simply use ellipses.

```
4ea37c2b8b0c5a68fde59770c3536195e0972217
977cd6cb28a37744966ec62f70cf62659f6f302a
625cee531623feddbe3174fad52c7db96ec60bb3
5490ccc5ebf39759dfd084bbd31f464701a3e775
0c06b93c1e467debd401eb0b3be4652fde14fa95
820739bbb572b30b6ce45756c9960e48dca859af
00e4e3e9d273a193620c3a4bb4914e555cb8e343
49cd561dc62ea6b3fbedab7aef0f020733f4cf09
abcb2e62dae6aa26308f7ac9efc89247f89cbe65
98ac36efe4ce4bd3a0bca76fc73ce6c7abaa4371
```

As usual, keep a log in the file `lab9.txt` of what you do in the lab so that you can reproduce the results later. This should not merely be a transcript of what you typed: it should be more like a true lab notebook, in which you briefly note down what you did and what happened. Your lab note should contain the abovementioned record of the scripts you used.

Homework: Topologically ordered commits

Given a Git repository with multiple branches, the commits can be thought of as having the structure of a rooted tree. In particular, starting from the head of each branch, i.e. the last commit of each branch, one can trace through the commits by following each commit's parent. Once this procedure is done for all the branches, we will get a commit graph, which is a rooted tree, with the root being the oldest commit which we assume all branches share.

Note that if a commit is a merge commit, it will have two parents. But one should always select the parent which gives the longest sequence of commits when tracing up to the root commit. The previous sentence will make sense after you investigate how a merge commit is created.

Follow these five steps to implement `topo_order_commits.py`.

1. Discover the `.git` directory. Inside a directory, when the script `topo_order_commits.py` (which doesn't have to reside in the same directory) is invoked, the script should first determine where the top level Git directory is. The top level Git directory is the one containing the `.git` directory. One can do this by looking for `.git` in the current directory, and if it doesn't exist search the parent directory, etc. Output a diagnostic '`Not inside a Git repository`' to standard error and exit with status 1 if `.git` cannot be found when the search went all the way to the `/` directory.
2. Get the list of local branch names. Figure out what the the different directories inside `.git` do, particularly the refs and objects directories. Read §10.2 Git Internals – Git Objects and §10.3 Git Internals – Git References. One can use the `zlib` library in Python

to decompress Git objects. To simplify this assignment, you can assume that the repository does not use packfiles (see §10.4 Git Internals – Packfiles), i.e., that all its objects are loose.

3. Build the commit tree. Each commit can be represented as an instance of the CommitNode class, which you can define as follows:

```
class CommitNode:
    def __init__(self, commit_hash, associated_branches):
        """
        :type commit_hash: str
        :param associated_branches: the branch names associated with the commit, set of str
        :type associated_branches: set
        """
        self.commit_hash = commit_hash
        self.associated_branches = associated_branches
        self.parents = set()
        self.children = set()

    def __str__(self):
        return f'commit_hash: {self.commit_hash}, associated_branches: {self.associated_branches}'

    def __repr__(self):
        return f'CommitNode<commit_hash: {self.commit_hash}, associated_branches: {self.associated_branches}>'
```

   The commit tree consists of all the commit nodes from all the branches. For a commit object with two parents, figure out which parent to use. Note that all branches should trace to a single root commit. If this assumption is violated, output a diagnostic '`multiple commit roots found`' to standard error, and exit with status 1.

4. Generate a topological ordering of the commits in the tree. A topological ordering for our case would be a total ordering of the commit nodes such that all the ancestral commit nodes come before the descendent commits, where the root commit node is the oldest ancestor. One way to generate a topological ordering is to use a depth-first search; see Topological sorting.

5. Given the ordering in the previous step, print the commit hashes in the reverse order. That is, the the root commit hash should be printed last. When jumping to a sibling branch of the tree, that is, if the next commit to be printed is not the parent of the current commit, insert a "sticky end" followed by an empty line before printing the next commit. The "sticky end" will be the commit hash of the parent of the current commit, with an equal sign "=" appended to it. This is to inform us how the fragments of the tree can be "glued" together to form the tree.

   Furthermore, if a commit corresponds to a branch head or heads, the branch names are listed after the commit in lexicographical order, separated by a white space.

For example, consider the following commits where c3 is a child of c1, and c5 is a child of c4:

```
c0 -> c1 -> c2 (branch-1)
       \
        c3 -> c4 (branch-2, branch-5)
               \
                c5 (branch-3)
```

A valid topological ordering will be (c0, c1, c2, c3, c4, c5), which should give the following output (assuming the commit hash for cX is hX):

```
h5 branch-3
h4 branch-2 branch-5
h3
h1=

h2 branch-1
h1
h0
```

A equally valid topological ordering will be (c0, c1, c3, c4, c5, c2), which should give the following output:

```
h2 branch-1
h1=

h5 branch-3
h4 branch-2 branch-5
h3
h1
h0
```

As a concrete example, given the repository in the flow-test-dir compressed tarball, the output of `topo_order_commits.py` could be as follows:

```
3c25412d2521b8f77778de839acb0350492c3634 b11 b12
5860da5c2f6181541566190b2f704da20e4753bc
b5d29544d6c69bc654588fbf22071bdaadbc0d23
```

```
777e41d9a9ba12b9a485300e16fa4405a4f633c0=

dd9515b11ebcab015a8a91b494097d06827df5a5 b1
55bb5940a549a475c768b503cb1b76a6f029d444
127169a28b68c1f09a6f28e289b7aafafe3105a3=

0e7b5db7a6ebfe3fa72598d0d7919d64bcfc2ab7 b9
777e41d9a9ba12b9a485300e16fa4405a4f633c0=

59961945fd47abdcf8d526154befb24810fa9a79 HEAD b3 b4 z
5fe5a724533a4970c19fa7bc0986c1f646a07e4c b2
831c585eaada1f7fb3b9f2751b8019dbccc59d94
127169a28b68c1f09a6f28e289b7aafafe3105a3 b8
f72697fe0dbf5455b61ab586346578710fbfb6fd
5e6f9ac703e879de3134990765d2e5dee7ee7ed1
777e41d9a9ba12b9a485300e16fa4405a4f633c0 b10 b13 master
6ad43a64e81fff1fd81de5d2fb6a7c99a71f60a9
a109cb0e9f18fd0f0106216fccb9c9719e5119db
```

Submit

Submit two files:

- The file `lab9.txt` as described in the lab.
- The file `topo_order_commits.py` as described in the homework.

The `.txt` file should be an ASCII text file, with no carriage returns.

Topologically Ordered Commits
version 1.2 (Last Updated 17:42 Dec 2, 2019)

Given a git repository, the commits can be thought of as having the structure of a directed acyclic graph (DAG) with the commits being the vertices. In particular, one can create a directed edge from each child commit to each of its parent commits. Alternatively, one can create a directed edge from each parent to each of its children. Note that if a commit is a merge commit, it will have two parents. In that case, one has to consider both parents.

Please follow these 5 steps and implement `topo_order_commits.py` using Python 3.8.0, the version of /usr/local/cs/bin/python3 on SEASnet.

1. Discover the .git directory

Inside a directory, when the script `topo_order_commits.py` (which doesn't have to reside in the same directory) is invoked, the script should first determine where the top level git directory is. The top level git directory is the one containing the .git directory. One can do this by looking for .git in the current directory, and if it doesn't exist search the parent directory, etc. Output a diagnostic 'Not inside a Git repository' to standard error and exit with status 1 if .git cannot be found when the search went all the way to the / directory.

2. Get the list of local branch names

Figure out what the different directories inside .git do, particularly the refs and objects directories. Beware of branch names with forward slashes. Read §10.2 Git Internals – Git Objects and §10.3 Git Internals – Git References. One can use the zlib library in Python to decompress Git objects.
To simplify this assignment, you can assume that the repository does not use packfiles (see §10.4 Git Internals – Packfiles), i.e., that all its objects are loose.

3. Build the commit graph

Each commit can be represented as an instance of the `CommitNode` class, which you can define as follows, and which you are also free to modify:

```
class CommitNode:
    def __init__(self, commit_hash):
        """
        :type commit_hash: str
        """
        self.commit_hash = commit_hash
        self.parents = set()
        self.children = set()
```

The commit graph consists of all the commit nodes from all the branches. Each commit node might have multiple parents and children.

In particular, for each branch, perform a depth-first search traversal starting from the branch head, i.e. the commit pointed to by the branch, to establish the parent-child relationships between the commit nodes. The traversal should trace through the parents, and for every possible pair of parent and child, add the child hash to the parent node's children, and add the parent hash to the child node's parents. The leaf nodes for each branch will be the root commits for that branch, where the leaf nodes are the nodes without any parents. Let **`root_commits`** be the union of all the leaf nodes across all the branches.

4. Generate a topological ordering of the commits in the graph
A topological ordering for our case would be a total ordering of the commit nodes such that all the descendent commit nodes are strictly smaller than the ancestral commits, where the nodes in **`root_commits`** are the oldest ancestors. One way to generate a topological ordering is to use a depth-first search; see Topological sorting.

In particular, given the way we do topological sorting in the discussion section slides, for each node **`s`** in **`root_commits`**, if that **`s`** has not been visited, then run DFS with **`s`** as the starting point and trace through the children, i.e. **`s.children`**.

5. Print the commit hashes in the order generated by step 4, from the smallest to the largest
If the next commit to be printed is not the parent of the current commit, insert a "sticky end" followed by an empty line before printing the next commit. The "sticky end" will contain the commit hashes of the parents of the current commit, with an equal sign "=" appended to the last hash. These hashes of the parents, if any, can be printed in any order separated by a whitespace. If there are no parents, just print an equal sign "=". This is to inform us how the fragments can be "glued" together.

On the other hand, if an empty line has just been printed, before printing the first commit C in a new segment, print a "sticky start" line starting with an equal sign "=", followed by the hashes of the children of C, if any, on the same line in any order and separated by a whitespace, so that we know which commits led to commit C (remember that children are smaller in the topological sort so they are printed first).

Furthermore, if a commit corresponds to a branch head or heads, the branch names should be listed after the commit in lexicographical order, separated by a white space.

Example 1
Consider the following commits where c3 is a child of c1, and c5 is a child of c4:

```
c0 -> c1 -> c2 (branch-1)
       \
        c3 -> c4 (branch-2, branch-5)
               \
                c5 (branch-3)
```

A valid topological ordering from the smallest to the largest will be (c5, c4, c3, c2, c1, c0), which should give the following output (assuming the commit hash for cX is hX, and the triple backticks are not part of the output but to indicate the start and end of the output lines):

```
h5 branch-3
h4 branch-2 branch-5
h3
h1=

=
h2 branch-1
h1
h0
```

An equally valid topological ordering from the smallest to the largest will be (c2, c5, c4, c3, c1, c0), which should give the following output:

```
h2 branch-1
h1=

=
h5 branch-3
h4 branch-2 branch-5
h3
h1
h0
```

Example 2

For the following commits where c6 is a merge commit with parents c2 and c4

```
c0 -> c1 -> c2 -> c6 (branch-1)
     \     /
      c3 -> c4
```

A topological ordering is (c6, c2, c4, c3, c1, c0), and the corresponding output should be:

```
h6 branch-1
h2
h1=

=h6
h4
h3
h1
h0
```

A concrete example

Given the repository in the flow-test-dir compressed tarball, the output of `topo_order_commits.py` could be as follows:

```
3c25412d2521b8f77778de839acb0350492c3634 b11 b12
5860da5c2f6181541566190b2f704da20e4753bc
b5d29544d6c69bc654588fbf22071bdaadbc0d23
777e41d9a9ba12b9a485300e16fa4405a4f633c0=

=
59961945fd47abdcf8d526154befb24810fa9a79 b3 b4 z
5fe5a724533a4970c19fa7bc0986c1f646a07e4c b2
831c585eaada1f7fb3b9f2751b8019dbccc59d94
127169a28b68c1f09a6f28e289b7aafafe3105a3=

=
dd9515b11ebcab015a8a91b494097d06827df5a5 b1
55bb5940a549a475c768b503cb1b76a6f029d444
127169a28b68c1f09a6f28e289b7aafafe3105a3 b8
```

```
f72697fe0dbf5455b61ab586346578710fbfb6fd
5e6f9ac703e879de3134990765d2e5dee7ee7ed1
777e41d9a9ba12b9a485300e16fa4405a4f633c0=

=
0e7b5db7a6ebfe3fa72598d0d7919d64bcfc2ab7 b9
777e41d9a9ba12b9a485300e16fa4405a4f633c0 b10 b13 master
6ad43a64e81fff1fd81de5d2fb6a7c99a71f60a9
a109cb0e9f18fd0f0106216fccb9c9719e5119db
```

Implementation Notes

1. Do not invoke any git commands in any way. For example, do not use Python subprocess to call git.

2. The output of `topo_order_commits.py` should be deterministic. For a given Git repository, even though there might be multiple possible ways to perform DFS, multiple valid topological orderings and multiple valid outputs, different invocations of `topo_order_commits.py` should produce identical outputs. For example, since the iteration order on a set **s** in Python is not deterministic, one way to ensure determinism is to call `sorted(list(s))`.

3. Only use the modules in the Python Standard Library. In fact, os, sys, zlib are the only libraries you need.

Updates

17:42 Dec 2, 2019
Step 5 now specifies that the parents/children in the sticky ends/starts can be printed in any order and separated by a whitespace.

```
Laboratory: Git spelunking

NOTE: Per the spec, this is a log of what I did in the lab so that I can reproduce the
results later and I've briefly noted down what I did and what happened. Trivial commands may
or may not be
explained.

# The commands for the following questions were run on lnxsrv03.

############################## Question 1
# Note the du command gives us the amount of space the file is using on disk, not the size of
the file, as desired for this question.

How much disk space is used to represent the working files in this directory?

162496 bytes

cd ~eggert/src/gnu/emacs-CS-35L
du --exclude='.git*' -s

Output of previous command:
162496 .

How much is used to represent the Git repository?

358316 bytes

du .git -s

Output of previous command:
358316 .git

What file or files consume most of the repository space and why?

The regular files consuming most of the repository space are:
- .git/objects/info/commit-graph because this file is a supplemental data structure that
accelerates commit graph walks (which can become slow as the commit count grows). This
indicates the repository
has a large number of commits, which we confirm using the command git rev-list to be 143910.
- .git/objects/pack/pack-24e56b1749b9320c560213d9045fa6cee42b4174.pack because this is the
packfile, a single binary file that Git packs objects into in order to save space and be more
efficient. The
packfile contains the contents of all the objects (such as blobs, trees, commits, and tags)
packed into it (which explains why it's so large), but it is still smaller than the original
size of the
objects. Git achieves this by looking for files that are named and sized similarly, and
stores just the deltas from one version of the file to the next.
- .git/objects/pack/pack-24e56b1749b9320c560213d9045fa6cee42b4174.idx because this is an
index that contains offsets into the packfile so you can quickly seek to a specific object.
Due to the large
number of commit objects and the fact this includes other types of objects as well, it makes
sense that this index consumes a sizable amount of disk space (though less than a tenth of
the space
consumed by the packfile itself).

du .git

Output of previous command:
56      .git/info
72      .git/hooks
4       .git/branches
4       .git/refs/heads/feature
4       .git/refs/heads/features
4       .git/refs/heads/fix
```

```
4       .git/refs/heads/heads
4       .git/refs/heads/old-branches
4       .git/refs/heads/other-branches
4       .git/refs/heads/scratch/joaot
4       .git/refs/heads/scratch/np
4       .git/refs/heads/scratch/ns
4       .git/refs/heads/scratch/tzz
20      .git/refs/heads/scratch
48      .git/refs/heads
4       .git/refs/tags
4       .git/refs/remotes/origin/feature
4       .git/refs/remotes/origin/scratch/joaot
4       .git/refs/remotes/origin/scratch/ns
12      .git/refs/remotes/origin/scratch
20      .git/refs/remotes/origin
24      .git/refs/remotes
80      .git/refs
342032 .git/objects/pack
7892    .git/objects/info
349928 .git/objects
24      .git/logs/refs/remotes/origin/feature
8       .git/logs/refs/remotes/origin/scratch/joaot
8       .git/logs/refs/remotes/origin/scratch/ns
40      .git/logs/refs/remotes/origin/scratch
76      .git/logs/refs/remotes/origin
80      .git/logs/refs/remotes
96      .git/logs/refs/heads/feature
8       .git/logs/refs/heads/features
56      .git/logs/refs/heads/fix
8       .git/logs/refs/heads/heads
100     .git/logs/refs/heads/old-branches
80      .git/logs/refs/heads/other-branches
8       .git/logs/refs/heads/scratch/joaot
8       .git/logs/refs/heads/scratch/np
12      .git/logs/refs/heads/scratch/ns
16      .git/logs/refs/heads/scratch/tzz
312     .git/logs/refs/heads/scratch
756     .git/logs/refs/heads
840     .git/logs/refs
848     .git/logs
358316 .git

ls -la .git/objects/info

Output of previous command:
total 7896
drwxr-xr-x 2 eggert csfac    4096 Nov 25 16:39 .
drwxr-xr-x 4 eggert csfac    4096 Aug 15 11:02 ..
-rw-r--r-- 1 eggert csfac 8060384 Nov 25 16:39 commit-graph
-rw-r--r-- 1 eggert csfac      54 Nov 25 16:39 packs

du .git/objects/info/commit-graph

Output of previous command:
7888    .git/objects/info/commit-graph

git rev-list --all --count

Output of previous command:
143910

ls -la .git/objects/pack

Output of previous command:
total 342036
```

```
drwxr-xr-x 2 eggert csfac      4096 Nov 25 16:39 .
drwxr-xr-x 4 eggert csfac      4096 Aug 15 11:02 ..
-r--r--r-- 1 eggert csfac  24393552 Nov 25 16:39 pack-
24e56b1749b9320c560213d9045fa6cee42b4174.idx
-r--r--r-- 1 eggert csfac 325142809 Nov 25 19:00 pack-
24e56b1749b9320c560213d9045fa6cee42b4174.pack

du .git/objects/pack -a

Output of previous command:
23876  .git/objects/pack/pack-24e56b1749b9320c560213d9045fa6cee42b4174.idx
318152 .git/objects/pack/pack-24e56b1749b9320c560213d9045fa6cee42b4174.pack
342032 .git/objects/pack

############################### Question 2
How many branches are local to the repository?

176 branches

git branch | wc -l

Output of previous command:
176

How many are remote?

176 branches

git remote

Output of previous command:
origin

git branch -r

# Exclude first line with shows what HEAD of origin points to.
git branch -r | tail -n +2 | wc -l

Output of previous command:
176

############################### Question 3
How many repositories are remote to this repository, and where are they?

1 repository. It's fetch and push URL's are https://git.savannah.gnu.org/git/emacs.git.

git remote -v

Output of previous command:
origin https://git.savannah.gnu.org/git/emacs.git (fetch)
origin https://git.savannah.gnu.org/git/emacs.git (push)

############################### Question 4
What are the ten local branches most recently committed to? List their names in order of
commit date, most recent first.

master
scratch/joaot/make-completion-at-point-function
feature/windows-with-utils
scratch/completion-api
scratch/a-modest-completion-redesign-proposal
scratch/fido-mode
feature/gnus-select2
feature/extend_face_id
scratch/jit-lock-antiblink-cleaned-up
```

```
emacs-26

git for-each-ref --sort=-committerdate refs/heads/ --format='%(committerdate:short) %
(refname:short)' --count=10

Output of previous command:
2019-11-25 master
2019-11-20 scratch/joaot/make-completion-at-point-function
2019-11-18 feature/windows-with-utils
2019-11-16 scratch/completion-api
2019-11-10 scratch/a-modest-completion-redesign-proposal
2019-11-05 scratch/fido-mode
2019-11-03 feature/gnus-select2
2019-10-14 feature/extend_face_id
2019-10-10 scratch/jit-lock-antiblink-cleaned-up
2019-10-07 emacs-26

git for-each-ref --sort=-committerdate refs/heads/ --format='%(refname:short)' --count=10

Output of previous command:
master
scratch/joaot/make-completion-at-point-function
feature/windows-with-utils
scratch/completion-api
scratch/a-modest-completion-redesign-proposal
scratch/fido-mode
feature/gnus-select2
feature/extend_face_id
scratch/jit-lock-antiblink-cleaned-up
emacs-26

############################## Question 5
How many commits are in the master branch?

139583 commits

git rev-list --count master

Output of previous command:
139583

############################## Question 6
What percentage of the commits that are in any branch, are also in the master branch?

(number of commits in master branch)/(total number of commits across all branches) = 139583 /
143910 = 0.9699 = ~0.97

This is approximately 97%.

git rev-list --count master

Output of previous command:
139583

git rev-list --all --count

Output of previous command:
143910

############################## Question 7
Which ten people authored the most master-branch commits that were committed in the year 2013
or later, and how many commits did each of them author during that period?

3691  Eli Zaretskii
3647  Glenn Morris
```

```
3605  Paul Eggert
1806  Lars Ingebrigtsen
1784  Stefan Monnier
1571  Michael Albinus
619   Dmitry Gutov
576   Noam Postavsky
471   Alan Mackenzie
469   Juri Linkov

# current branch is master
git shortlog -n -s --since="2013-01-01T00:00:00-08:00" | head -n 10

Output of previous command:
  3691 Eli Zaretskii
  3647 Glenn Morris
  3605 Paul Eggert
  1806 Lars Ingebrigtsen
  1784 Stefan Monnier
  1571 Michael Albinus
   619 Dmitry Gutov
   576 Noam Postavsky
   471 Alan Mackenzie
   469 Juri Linkov

############################## Question 8
Use the gitk command to visualize the commit graph in this repository. If you are SSHing into
SEASnet, you'll need to log in via ssh -X or (less securely) ssh -Y. Draw a diagram relating
the following
commits to each other, and explain what likely happened to cause their commit-graph
neighborhood. You need not list every single intervening commit individually; you can simply
use ellipses.

4ea37c2b8b0c5a68fde59770c3536195e0972217
977cd6cb28a37744966ec62f70cf62659f6f302a
625cee531623feddbe3174fad52c7db96ec60bb3
5490ccc5ebf39759dfd084bbd31f464701a3e775
0c06b93c1e467debd401eb0b3be4652fde14fa95
820739bbb572b30b6ce45756c9960e48dca859af
00e4e3e9d273a193620c3a4bb4914e555cb8e343
49cd561dc62ea6b3fbedab7aef0f020733f4cf09
abcb2e62dae6aa26308f7ac9efc89247f89cbe65
98ac36efe4ce4bd3a0bca76fc73ce6c7abaa4371

# Diagram:

# NOTE: 977cd6cb28 is a child of 4ea37c2b8b and 625cee5316 is a child of 4ea37c2b8b, as in
the format of the diagram in the spec of Homework: Topologically ordered commits.


4ea37c2b8b -> 977cd6cb28 -> ... -> ... -> 0c06b93c1e -> ... -> ... -> ... -> ... ->
abcb2e62da -> ... -> 98ac36efe4 -> ...
    \                                    /                                              /
/
     \                              ... -> ... -> 820739bbb5 -> 00e4e3e9d2 -> ... ->
49cd561dc6 -> ... -> ...
      \                            /
   625cee5316 -> ... -> 5490ccc5eb
                                   \
                                    ...

# Explain what likely happened to cause their commit-graph neighborhood:
# - Note in the diagram that a commit with two children (where both have no other parents)
likely indicates a branch has been created, and a commit with two parents likely indicates it
was a merge
# commit
# - By that reasoning, the following sets of commits explain what likely happened to cause
the commit-graph neighborhood:
# - branch created: 4ea37c2b8b, 5490ccc5eb
# - merge commit: 0c06b93c1e, abcb2e62da, 98ac36efe4

ssh -X stewart@lnxsrv03.seas.ucla.edu
cd ~eggert/src/gnu/emacs-CS-35L
gitk --all
# Type a unique prefix of the SHA-1 commit hash in the commit hash field in gitk, then click
"Goto:" for each commit and scroll
```

```
import os, sys, zlib

class CommitNode:
    def __init__(self, commit_hash):
        """
        :type commit_hash: str
        """
        self.commit_hash = commit_hash
        self.associated_branches = set()
        self.parents = set()
        self.children = set()
        commit_obj_str = self.get_commit_obj_str_from_hash(commit_hash)
        commit_obj_lines = commit_obj_str.split("\n")
        for line in commit_obj_lines:
            line_words = line.split(" ")
            if line_words[0] == "parent":
                self.parents.add(line_words[1])
    def get_commit_obj_str_from_hash(self, commit_hash):
        top_level_git_dir = get_top_level_git_dir()
        commit_obj_file = open(os.path.join(top_level_git_dir, ".git/objects",
commit_hash[0:2], commit_hash[2:40]), "rb")
        return zlib.decompress(commit_obj_file.read()).decode()
    def __str__(self):
        return f'commit_hash: {self.commit_hash}, parents: {self.parents}, children:
{self.children}, associated_branches: {self.associated_branches}'

def get_top_level_git_dir():
    # Discover the .git directory
    top_level_git_dir = os.getcwd()
    while os.path.isdir(os.path.join(top_level_git_dir, ".git")) == False:
        if top_level_git_dir == "/":
            sys.stderr.write("Not inside a Git repository\n")
            sys.exit(1)
        top_level_git_dir = os.path.dirname(top_level_git_dir)
    return top_level_git_dir

def get_local_branch_names(top_level_git_dir):
    # Get the list of local branch names
    local_branch_names = list()
    local_branch_dir = os.path.join(top_level_git_dir, ".git/refs/heads")
    for root, dirs, files in os.walk(local_branch_dir):
        if root == os.path.join(top_level_git_dir, ".git/refs/heads"):
            local_branch_names.extend(files)
        else:
            for file in files:
                local_branch_names.append(os.path.join(os.path.relpath(root,
local_branch_dir), file))
    return local_branch_names

def build_commit_graph(top_level_git_dir, local_branch_names):
    # Build the commit graph
    commit_graph = {}
    root_commits_set = set()
    for branch in local_branch_names:
        # Get commit hash of branch head
        f = open(os.path.join(top_level_git_dir, ".git/refs/heads", branch), "r")
        branch_head_commit_hash = f.read()
        # Do DFS starting from the branch head using a stack
        vertex = CommitNode(branch_head_commit_hash[0:40])
        if vertex.commit_hash in commit_graph:
            commit_graph[vertex.commit_hash].associated_branches.add(branch)
        else:
            vertex.associated_branches.add(branch)
        stack = [(vertex, [vertex])]
        visited = set()
        while stack:
            v, path = stack.pop()
            visited.add(v)
            if len(v.parents) == 0:
                root_commits_set.add(v.commit_hash)
            if not v.commit_hash in commit_graph:
                commit_graph[v.commit_hash] = v
            else:
                for child in sorted(list(v.children)):
                    commit_graph[v.commit_hash].children.add(child)
            for parent in sorted(list(v.parents)):
                if parent not in visited:
                    parent_obj = CommitNode(parent)
                    parent_obj.children.add(v.commit_hash)
                    stack.append((parent_obj, path + [parent_obj]))
    # The output should be deterministic
    root_commits = sorted(list(root_commits_set))
    return commit_graph, root_commits

def topo_sort_commits(commit_graph, root_commits):
    # Generate a topological ordering of the commits in the graph
    # Use list implementation of queue
    result = []
    visited = set()
    # For each node s in root_commits, if that s has not been visited, then run DFS with s as
the starting point and trace through the children
    for commit in root_commits:
        if commit not in visited:
            s = commit_graph[commit]
            dfs_stack = [s]
            aux_stack = []
            while dfs_stack:
                v = dfs_stack.pop()
                visited.add(v.commit_hash)
                # Default to True for when node has no children
                children_processed = True
                for child_commit_hash in sorted(list(v.children)):
                    if child_commit_hash not in visited:
                        children_processed = False
                if children_processed == False:
                    aux_stack.append(v)
                else:
                    if v not in result:
                        result.append(v)
                # The output should be deterministic
                children_sorted = sorted(list(v.children))
                for child_commit_hash in children_sorted:
                    if child_commit_hash not in visited:
                        dfs_stack.append(commit_graph[child_commit_hash])
                # Check if we finished processing the children for previously visited nodes
                finished_proc_node = True
                while aux_stack and finished_proc_node == True:
                    node = aux_stack.pop()
                    for child_commit_hash in sorted(list(node.children)):
                        if child_commit_hash not in visited:
                            finished_proc_node = False
                    if finished_proc_node == False:
                        aux_stack.append(node)
                    else:
                        if node not in result:
                            result.append(node)
    return result

def print_sticky_start(commit_obj):
    # The output should be deterministic
    if len(commit_obj.children) == 0:
        print("=")
    else:
        children_sorted = sorted(list(commit_obj.children))
        print("=" + " ".join(children_sorted))

def print_topo_order_commits(result):
    for i in range(len(result)):
        if len(result[i].associated_branches) != 0:
            # The output should be deterministic
            branches_sorted = sorted(list(result[i].associated_branches))
            print(result[i].commit_hash + " " + " ".join(branches_sorted))
        else:
            print(result[i].commit_hash)
        if i != len(result)-1:
            if result[i + 1].commit_hash not in result[i].parents:
                # Insert a "sticky end" followed by an empty line
                # The output should be deterministic
                if len(result[i].parents) == 0:
                    print("=\n")
                    print_sticky_start(result[i + 1])
                else:
                    parents_sorted = sorted(list(result[i].parents))
                    print(" ".join(parents_sorted) + "=\n")
                    print_sticky_start(result[i + 1])

# Keep the function signature,
# but replace its body with your implementation
def topo_order_commits():
    top_level_git_dir = get_top_level_git_dir()
    local_branch_names = get_local_branch_names(top_level_git_dir)
    commit_graph, root_commits= build_commit_graph(top_level_git_dir, local_branch_names)
    result = topo_sort_commits(commit_graph, root_commits)
    print_topo_order_commits(result)
    #raise NotImplementedError


if __name__ == '__main__':
    topo_order_commits()
```