UCLA Computer Science 97 Midterm 2 Solutions

Spring 2020

(Instructor: Dr. Paul Eggert)

These solutions are intended to be as comprehensive and instructive as we can make them. They are not exemplars of the level of detail expected of you in the actual midterm.

1. **In a Files-11 type filesystem, it's the filesystem's job to make numbered backups, which it does under application control. On typical GNU/Linux hosts, the filesystem does not make backups; instead, applications like GNU Emacs can individually choose filenames like '`foo.~1~`', '`foo.~2~`', etc. if they want to make numbered backups.**

    a. **Discuss pros and cons of the two approaches.**

    Answer:
    Pros:
    — For Files-11, the application/user does not need to worry about handling backups. The filesystem will do it for them.
    — For GNU/Linux, the application is able to have a custom naming convention; i.e. you can tell which backups were made by what application solely based on file names.

    Cons:
    — All else being equal, the default Files-11 type filesystem's performance is bound to be lower than a file system that does not automatically create backup files.
    — Many applications (such as Git) have their own robust backup mechanism that may duplicate the features of Files-11. Even if the filesystem does provide a way for applications to disable the behavior, that would require extra specialized adaptations in the application code.
    — For GNU/Linux, a serious problem arises if two applications choose the same backup naming convention. Backups created by one application could be overwritten by another.

    b. **Suppose you are using Git on a filesystem like Files-11, or with an application like GNU Emacs. How should Git interact with the numbered backups? Briefly explain.**

    Answer: Git should ignore any of these backup files automatically created, perhaps by including them in a `.gitignore` file. Git has its own backup mechanism, so the numbered backups don't really help Git, but rather just gives Git more files to keep track of.

2. **What technique is used both by WAFL snapshots and by local-filesystem '`git clone`' in order to improve efficiency? And what resources are conserved via this technique? Briefly explain.**

   Answer: Both WAFL snapshots and `git clone` use a technique called copy-on-write, where a copy of a file is made not when the user requests it to be made, but only when the copy is modified. This technique conserves disk space for the case that two copies of the file exist in the file system, but they contain the same content.

3. **Suppose you meant to type '`cp ../myconfig .gitconfig`' but by mistake you typed '`cp ../myconfig .git/config`'. What's a good thing to do to recover if you did this on SEASnet? On your laptop? Explain any assumptions you make.**

   Answer: The incorrect operation might have overwritten the `.git/config` file, which is generally a file containing Git configurations managed by Git itself. Our recovery effort will focus on getting this file back somehow.

   SEASnet has a WAFL file system that periodically makes snapshots of the whole filesystem. Assuming that a snapshot was taken before we overwrote the file in question, we could access that snapshot as done in lecture and use it to restore the correct .git/config file.

   On an ordinary laptop, the correct answer depends highly on the configuration of the computer. If backups (or snapshots) are regularly taken, then it would be a trivial matter to go look for a backed-up version of the .git/config – though it would be unfortunate if this file did not exist in the latest snapshot or has been modified since then.

   If backups are not taken, the best course of action may well be to clone the repository anew (assuming it has been pushed to a remote), and copy over the pristine configuration file from the new repository to the overwritten file. You may have to redo some configurations which were present in the old file but not in the cloned configuration, though.

4. **Git's implementation would have been simpler and more reliable if commits lacked commit messages, with the idea that programmers should instead use the convention that to associate a message with your commit you append the message to a file named `COMMITLOG` in the root directory of your project, and make `COMMITLOG`'s changes part of your commit. Explain why this simplification of Git would have been a bad idea, and give an example of what could go wrong.**

Answer: There are many reasons why this is a bad idea. Here are some that we came up with:

- It introduces many chances for developer error. Requiring the developer to edit the COMMITLOG file is more work than supplying it via the command line, so from a design perspective, it would promote laziness by discouraging people from making commit messages. Furthermore, depending on such a convention to track project history is potentially unreliable as it is up to each individual developer to follow it properly.

- It makes otherwise conflict-free and painless `git merges` have merge conflicts in the `COMMITLOG` file. This would make it difficult to automate such merge operations by a machine.

- It is less space-efficient. Git commits (including messages) are compressed using gzip. Although blobs are also compressed, since Git may not use delta encoding for them in certain situations, this might lead to having multiple copies of essentially the same file in the Git object store.

- If a Git commit for some reason does not have a `COMMITLOG` file (*e.g.,* if the user is being sloppy), then the default output of `git log` would have virtually no information other than the hash, author/committer, and dates. While one could argue that the same would be true if the user is sloppy about commit messages, it has been shown that defaults dominate user behavior.

5.

a. Suppose changes to four files A, B, C, and D are in your Git index, but before you commit you decide that you want to split the first commit into two, with the changes to A and C being in the first commit and the changes to B and D being in the second commit. Briefly explain how you'd go about this.

Answer: Since the files are already in the Git index (staging area), we first need to unstage them with `git reset`. After this, we simply `git add` A and C and make the first commit, then `git add` B and D and make the second one. The following sequence of commands achieve this:

```
$ git reset A B C D
$ git add A C
$ git commit -m 'first commit'
$ git add B D
```

```
$ git commit -m 'second commit'
```

b. **Suppose after you've made both of the commits of (a), you change your mind and decide that you should have put the changes to A and B in the first commit, and the changes to C and D in the second. Briefly explain how you'd fix things up, hopefully before your boss sees your mistake.**

Answer: Since we don't want our boss to see our mistake, we should rewind history and "undo" the commits we just made. We can do this using an interactive rebase. One way is the following:

```
$ git rebase -i HEAD^
# Change the rebase TODO to say 'pick' for the first commit and 'fixup' for the
# second.
# Now the changes to A, B, C, D are all in the same commit, known as HEAD.
$ git reset HEAD^
# Now the changes to A, B, C, D are all unstaged.
$ git add A B
$ git commit -m 'first commit'
$ git add C D
$ git commit -m 'second commit'
```

This certainly works, but note that we used `git reset` right after `git rebase`: we don't even have to rebase in the first place! A better solution is this:

```
$ git reset HEAD^^
# Now the changes to A, B, C, D are all unstaged.
$ git add A B
$ git commit -m 'first commit'
$ git add C D
$ git commit -m 'second commit'
```

6. **In Git, a detached HEAD occurs when HEAD points directly at a commit, rather than at a branch tip.**

   a. **Explain how to get a detached HEAD.**

   Answer: You can get a detached HEAD using the `git checkout` command, and passing it any valid commit hash (i.e. not a branch name) as its single argument.

b. **Suppose HEAD is detached. How can you reattach it, without changing any commits or working files?**

Answer: You can make a branch at your current head:

```
$ git checkout -b new-branch
or
$ git switch -c new-branch
```

In fact, if you create a detached HEAD, the above is precisely what Git will tell you:

> `$ git checkout HEAD^`
> Note: switching to 'HEAD^'.
>
> You are in 'detached HEAD' state. You can look around, make experimental changes and commit them, and you can discard any commits you make in this state without impacting any branches by performing another checkout.
>
> If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-b` with the `checkout` command again. Example:
>
> ```
> git checkout -b <new-branch-name>
> ```

If you are using newer versions of Git, you may see this instead:

> If you want to create a new branch to retain commits you create, you may do so (now or later) by using `-c` with the `switch` command. Example:
>
> ```
> git switch -c <new-branch-name>
> ```
>
> Or undo this operation with:
>
> ```
> git switch -
> ```

c. **A "detached tail" occurs if the tail of a chain of commits is not in any branch. Why does Git warn you about detached heads but not detached tails?**

Answer: A detached tail can only occur if there is a detached head, so warning about either head or tail is enough. Additionally, there can be multiple detached

tails corresponding to a single detached head, so warning about detached heads is more concise.

7. **If ‘make’ lets you restart failed builds without doing everything from scratch, whereas a shell script doesn’t have that capability, why do people bother using the shell instead of using ‘make’ for everything?**

   Answer: A few possible answers here:

   - Generally, the shell is available on more platforms out-of-the-box, while ‘make’ may be something that people have to install, usually as part of the “software development” suite of packages.

   - Shell scripts, because they start from scratch every single time, tend to be more predictable in their operations. If for example a previous instance of the script created a partial output before it crashed, ‘make’ would happily treat that as finished while the shell script would recreate the output.

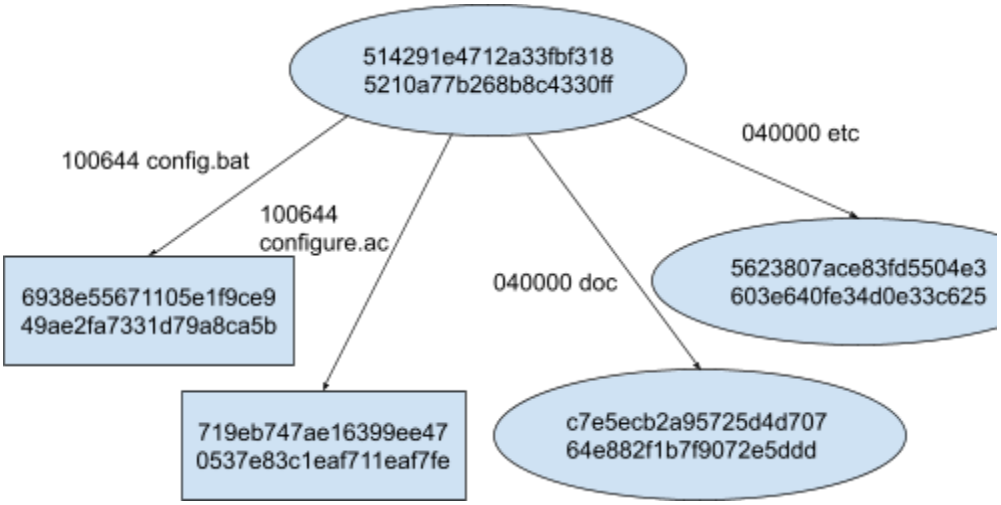
   - ‘make’ requires files to be created in order to figure out what work has been done. Sometimes, creating files may not be desirable for a script.

8. **Consider the following shell command and output:**

```
$ git cat-file -p ‘514291e4712a33fbf3185210a77b268b8c4330ff^{tree}’
100644 blob 6938e55671105e1f9ce949ae2fa7331d79a8ca5b  config.bat
100644 blob 719eb747ae16399ee470537e83c1eaf711eaf7fe  configure.ac
040000 tree c7e5ecb2a95725d4d70764e882f1b7f9072e5ddd  doc
040000 tree 5623807ace83fd5504e3603e640fe34d0e33c625  etc
```

   a. **Draw a diagram of the tree that this output represents, as best you can.**

Answer:



Ellipses are trees, boxes are blobs.

b. Do Git tree objects represent only trees? Or can they represent arbitrary graphs? Explain what kind of graphs can be represented by Git tree objects, and how Git constrains the graphs to be the kind you say.

Answer: The question is a bit confusingly worded. Git tree objects can only represent directed graphs (*i.e.,* graphs whose edges have directions/arrows), while *technically* "tree" only refers to undirected graphs, so Git tree objects cannot represent trees. The following extremely detailed answer is based on the assumption that a tree is an *undirected* graph.

Here are its most important insights (tl;dr):
— Git tree objects can represent any connected DAGs.
— Used normally, Git tree objects always represent connected graphs. Any Git object not connected to the tree cannot be a part of the tree in the first place.
— Git tree objects can never contain cycles. This is because each tree object is identified by a SHA-1 hash.

Let us define some more terminology found in graph theory:

— A *tree* is a *connected*, *acyclic, undirected* graph.
— A *directed tree* is a *directed* graph whose underlying undirected graph is a tree.
— A *DAG* is an *acyclic, directed* graph.
— A *connected DAG* is a *connected, acyclic,* and *directed* graph.

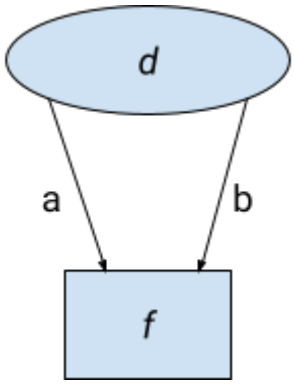
All connected DAGs are DAGs (though not vice versa), while all directed trees are connected DAGs (though not vice versa).

Git tree objects can represent any connected DAGs. We will first show that the graph they represent must be *connected* and *acyclic*. Then, we will show that they can represent graphs that are not directed trees.

Git constrains tree objects to be connected simply by definition: any object that is not connected with the tree object is *ipso facto* not part of the graph that the tree object represents.

Git tree objects can never contain cycles. A cycle could only occur if there was a path from the tree object back to itself, through other tree objects (blobs cannot point to anything). Since each tree object is identified by a SHA-1 hash, in order to create another tree object that contains this tree object we must know its hash. But one can't create a hash for an object that (directly or indirectly) *contains its own hash*, at least through normal means. (This is a chicken-and-egg problem, and we don't consider bruteforcing here.) This shows that cycles cannot exist.

Git tree objects *can* represent non–directed trees. Consider a tree object with hash *d* with two blobs 'a' and 'b', both of them with the same content (and thus the same blob hash *f* (they will have different commit hashes if introduced in separate commits, though!)). The tree object *d* would look like the following:



While this is a DAG, as there is no cycle, it is not a directed tree, as the underlying undirected graph (*i.e.*, remove the arrows from the edges) has a cycle (*d*, *f*, *d*).

(Strictly speaking, Git tree objects in fact represent connected, acyclic, directed *multigraphs,* as multiple edges are allowed between the same pair of nodes. But we will omit that distinction for brevity.).

9. **Explain why merging *two* branches invariably involves looking at at least *three* commits.**

Answer: Motivation: Sometimes when you do `git merge`, you will see something like "falling back to three-way merge" when things didn't go as smoothly as they could. This is what the question is about.

In order to merge two branches, the version control system (VCS) must know what each branch has changed in order to merge things correctly. In the usual coworker analogy, if Alice changed some code on branch *a*, and Bob changed some code on branch *b*, a key part of the merging process would be to first understand what Alice and Bob each changed. To do so, one would have to look for the closest ancestor *c* to the tip of the two branches. Then the diff *c..a* is compared against the diff *c..b*, and an algorithm will be able to figure out what the end state would look like. Commits *a*, *b*, and *c* are the three commits the VCS must look at.

Also check out GNU diff3, which diffs three files and even has a merge mode!

10. **Will your projects' code be more concerned about throughput or about latency? Briefly explain why, and give a technique you'll likely use to address the more important of the two concerns.**

    Answer: This question, of course, is inherently open-ended and the answer will vary depending on your particular project. Most importantly, it is asking you to demonstrate an understanding of the difference between throughput and latency as performance metrics.

    In the context of networking applications, throughput measures how much data can be transmitted at its peak, while latency measures how long it takes a piece of data to be transmitted. The following are two sample answers advocating for the importance of one or the other:

    Answer 1: Our project's code will be more concerned about **throughput**. Our project involves an interactive map that plots walking routes optimized for safety. This calculated route can quickly become complex: for example, in urban areas with many streets or as the distance between the start and destination increases. Since we would like our app to be robust, it is critical for us to be able to send large amounts of data to the client if needed. One technique we are using to address this is compression. We encode routes as a series of points that we will linearly connect on the client side, and before sending, we further compress the data using the compression middleware from npm (utilizing gzip or Brotli). This effectively increases the total amount of data we can send.

    Answer 2: Our project's code will be more concerned about **latency**. Our project relies on a real-time messaging feature in order to connect prospective students with student

mentors. As such, it is critical for our app to be responsive, with the shortest delay possible between the sending and receiving of a message. One technique we are using to address this is asynchronous programming. We are relying on Google Firebase's real-time database: when a message is sent, the necessary overhead to package the data and send it over the Internet is performed concurrently with updating the sender's UI. Simultaneously, the receiver is asynchronously listening for new messages and can thus process any incoming data as quickly as possible.

11. **Suppose your project's application used HTTP/1. Explain why head-of-line (HOL) blocking could hurt its performance, and how switching to HTTP/2 could help your app avoid some (but not all) performance issues due to HOL blocking.**

    Answer: Suppose we have multiple server responses that we would like to send back to the client via a single connection.

    HTTP/1 mandates responses be strictly sent (and received) in sequential order. So HOL blocking occurs when the first response "holds up" the rest of the responses (due to its size). This is particularly harmful when the first response is for a non-critical resource (like a meme in GIF format) compared to other responses (say a CSS stylesheet).

    This bottleneck easily generalizes as a problem for virtually any application that makes use of multiple resources (*i.e.,* not the CS 97 course website). Any reasonable example of how your project would be affected by this would suffice.

    Switching to HTTP/2 would help tremendously in mitigating this problem, because unlike HTTP/1 it allows for multiple concurrent streams (responses) in a single TCP connection, a pattern called multiplexing. Now, different responses can be sent in parallel to the client – if one response ends up being disproportionately large or incurs larger transmission delays for whatever reason, it will probably not block many of the other packets. In practice, you wouldn't have to wait for one large file to be transmitted fully before others you request from the server.

    However, HTTP/2 would not help in avoiding *all* performance issues related to HOL blocking. HTTP/2 only solves the HOL blocking problem at the application-layer level. Although packets from multiple streams can be sent at the same time, HOL blocking can still occur at the TCP (transport layer) level, as TCP also mandates packets be sent/received in order. If a TCP packet is lost, remaining segments will not be delivered to the app until the lost segment is re-sent and delivered, since TCP guarantees reliable delivery.