Git From Scratch

Name: Richa Sharma
Student ID: 23037468
Project Link

Introduction

Version control systems like Git are essential tools for developers to track changes to source code over time. In this project, I implemented core Git functionality from scratch in Python. Developing my own miniature version control system allowed me to deeply understand the algorithms and data structures powering Git. This report summarizes key techniques I employed, including hash-based storage, trees for commit history, simplified diffing for file comparisons, and reference tracking workflows. In the following sections, I will provide an overview of major algorithms and concepts used in my Python Git implementation called ualg and analyze their time and space complexities.

Command Line Interface and Workflow

In our project, we implemented different Git commands and sub-programs to manage various aspects of version control.

1 Initialization (`init` command)

The `init` command helps initialize an empty git repository. It creates the folders and files necessary to keep track of changes in the repository and perform different git operations. For instance when we initialize a demo repository using our program:

```
$ ./ualg init demo
```

We get the following tree structure in the repository.

```
demo
        .git
           branches
           config
           description
           HEAD
           objects
           refs
              heads
              tags
```

2 File Information (`cat-file` command)

The `cat-file` command, when used with a commit object, retrieves details about the commit. It provides essential metadata such as the associated tree, parent commit, author, committer, and commit message. For instance, when the cat-file command is called with the commit hash `8a842cf523edd570f32908679cc47007df7c30e8`,

```
./ualg cat-file commit 8a842cf523edd570f32908679cc47007df7c30e8
```

the returned information includes:

```
tree 1d5b10410d4c2e3a8702db78f133222c8c54c9d7
parent bac445df59168f25e3de2d6bfd3eb715ccb2ef0a
author 14Richa <onlinericha19@gmail.com> 1701192532 +0530
committer 14Richa <onlinericha19@gmail.com> 1701192532 +0530
```

```
added tag command
```

3 Object Hashing (`hash-object` command)

The `hash-object` command is used in generating unique identifiers for objects or files within the repository. This allows us to track changes effectively and verify the integrity of our data.

For instance, upon executing the command `./ualg hash-object -t blob README.md`, the system produces the following output:

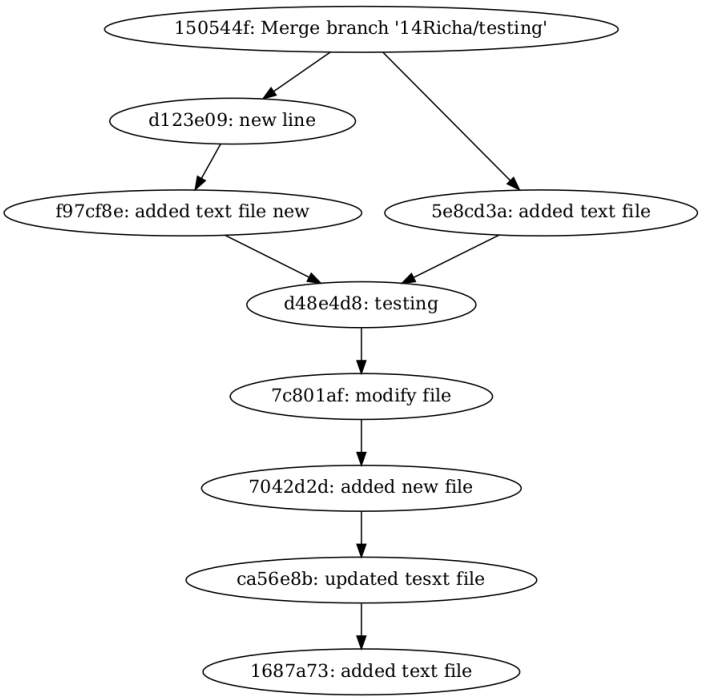
```
File path: README.md
SHA: 790127396e328869fdb69c1df4b7e233538682b5
```

This output is dependent on the content and structure of the file. The hash would change if the content of the file changes.

4 Commit History (`log` command)

The `log` command generates a visual representation of the commit history using Graphviz, resulting in a PDF file. The picture shows how the project has changed over time. For example, when executed with the command `./ual log > log.dot`, a log.dot file is created. Subsequently, when we run `dot -O -Tpdf log.dot`, a PDF file containing the graph is generated.

Below is an example of a generated PDF file:



5 Listing Files (`ls-tree` command):

The `ls-tree` command generates a structured representation of files and directories within the given branch of the repository. For instance, when executed with the command `./ualg ls-tree main`, it returns:

```
100644 blob 1d2099eaff2ec396e5dd739598c89019ec7f3c43 README.md
40000 None 96feb223c5f044c84ec0fc6364d45ae382b6f5a4 commands
40000 None 3ae7018ee42f9f5928e0463439df2843e4b5dc29 models
100755 blob f71dd992f8fc6effb946712e98070ce0457da791 ualg
```

6 List References (`show-ref` command)

The `show-ref` command listed references within the repository along with their corresponding commit hashes. Executing the command `./ualg show-ref`, it returns:

```
ccaf4a50321ac260aefeac2768117e552b1c7e1c refs/heads/main
ccaf4a50321ac260aefeac2768117e552b1c7e1c refs/remotes/origin/main
```

Algorithms Underneath

In the remainder of this report, we will discuss the major algorithms powering my Python Git implementation in greater depth. Analyzing these algorithms is key both for understanding how Git functions, as well as documenting the techniques I employed within my project. I will discuss the motivation and context for each algorithm, explain the implementation details and analyze time and space complexity tradeoffs.In the remainder of this report, I will explore the major algorithms powering my Python Git implementation in greater depth. Analyzing these algorithms is key both for understanding how Git functions at a fundamental level, as well as documenting the techniques I employed within my project. I will discuss the motivation and context for each algorithm, explain the implementation details and highlight any custom optimizations, analyze time and space complexity tradeoffs, and outline where further improvements can be made.

1. **Recursive Search Algorithm** The function `repo_find` was designed to search for a Git repository within a specified directory or its parent directories. This recursive search algorithm aims to locate the presence of a .git directory, indicative of a Git repository, employing a greedy strategy.

```python
def repo_find(path=".", required=True):
    path = os.path.realpath(path)
    # check if the .git directory exists within the path
    if os.path.isdir(os.path.join(path, ".git")):
        return GitRepository(path)

    parent = os.path.realpath(os.path.join(path, ".."))

    if parent == path:
        if required:
            raise Exception("No git directory.")
        else:
            return None
    return repo_find(parent, required)
```

   Time Complexity: $O(N)$ - The function might traverse up to N directories, where N is the depth of the directory structure. In the worst case, it goes up to the root directory.
   Space Complexity: $O(N)$ - Due to the recursive nature of the function, there could be up to N recursive calls on the call stack, where N is the depth of the directory tree.

2. **Recursive Parsing Algorithm:** The `kvlm_parse` function, recursively parses a raw byte string into key-value pairs. It identifies keys and values delimited by spaces and newlines, constructing an ordered dictionary. The process involves extracting keys until spaces, locating values by line ends, and managing key collisions by appending values or creating lists. This recursive approach systematically generates a dictionary from the input byte string, ensuring a structured mapping of keys to their respective values.

```python
def kvlm_parse(raw, start_index=0, dictionary=None):
    if not dictionary:
        dictionary = collections.OrderedDict()

    space_index = raw.find(b' ', start_index)
    newline_index = raw.find(b'\n', start_index)

    if (space_index < 0) or (newline_index < space_index):
        assert newline_index == start_index
        dictionary[None] = raw[start_index+1:]
```

```
    return dictionary

key = raw[start_index:space_index]

end = start_index
while True:
    end = raw.find(b'\n', end+1)
    if raw[end+1] != ord(' '): break
value = raw[space_index+1:end].replace(b'\n ', b'\n')

if key in dictionary:
    if type(dictionary[key]) == list:
        dictionary[key].append(value)
    else:
        dictionary[key] = [ dictionary[key], value ]
else:
    dictionary[key]=value

return kvlm_parse(raw, start_index=end+1, dictionary=dictionary)
```

Time Complexity Worst-Case: $O(N^2)$, where N is the length of text. The function iterates over text multiple times due to the find and replace operations within the loop and recursive calls. In the worst case, this can lead to quadratic complexity.
Space Complexity Worst-Case: $O(N)$, where N is the length of raw. The space complexity is linear with the size of the input, as it stores parts of the input in the dictionary and also uses space for recursive function calls.

3. **Merge Sort Algorithm:** The `tree_serialize` function uses the merge sort algorithm to sort a list of items within a tree-like structure (tree_object). Once the items are sorted, it converts them into a binary format. This conversion process involves taking each item's mode, path, and sha data and combining them into a structured binary representation. The result is a binary string that represents the sorted items from the tree, with the help of the merge sort algorithm for efficient sorting.

```
def tree_serialize(tree_object):
    def merge_sort(arr):
        if len(arr) <= 1:
            return arr

        mid = len(arr) // 2
        left = arr[:mid]
        right = arr[mid:]

        left = merge_sort(left)
        right = merge_sort(right)

        return merge(left, right)

    def merge(left, right):
        result = []
        i = j = 0

        while i < len(left) and j < len(right):
            if tree_leaf_sort_key(left[i]) < tree_leaf_sort_key(right[j]):
                result.append(left[i])
                i += 1
            else:
                result.append(right[j])
                j += 1

        result.extend(left[i:])
        result.extend(right[j:])
```

```
        return result

    tree_object.items = merge_sort(tree_object.items)

    serialized_tree = b''
    for item in tree_object.items:
        serialized_tree += item.mode
        serialized_tree += b' '
        serialized_tree += item.path.encode("utf8")
        serialized_tree += b'\x00'
        sha_int = int(item.sha, 16)
        serialized_tree += sha_int.to_bytes(20, byteorder="big")

    return serialized_tree
```

Time Complexity The function's time complexity is dominated by the merge sort algorithm, which has an average and worst-case time complexity of O(N log N), where "N" is the number of items in the tree.
Space Complexity The function uses additional space for the merge sort algorithm, which requires O(N) space due to its recursive calls and the creation of temporary subarrays.

4. **Markov Chain Algorithm:**

   The code includes two functions, create_transition_matrix and markov_hash, for generating a Markov Chain-based hash from input data.

   `create_transition_matrix` constructs a transition matrix by analyzing the input data's byte transitions, normalizing the matrix to represent probabilities, and handling uniform distributions when necessary.

   `markov_hash` uses the transition matrix to create a hash by performing a random walk through the data, selecting states according to the matrix's probabilities, and converting the resulting states into a hexadecimal string.

```
def create_transition_matrix(data, num_states=256):
    matrix = np.zeros((num_states, num_states))
    for i in range(len(data) - 1):
        current_state = data[i] % num_states
        next_state = data[i + 1] % num_states
        matrix[current_state][next_state] += 1

    # Normalize the matrix and handle rows that sum to 0
    for row in matrix:
        total = np.sum(row)
        if total > 0:
            row[:] = [x / total for x in row]
        else:
            row[:] = [1 / num_states for _ in row] # Uniform distribution

    return matrix

def markov_hash(data, hash_length=8, num_states=256): # Generate a hash using a
    Markov Chain.
    if not data:
        return ""

    # Convert data to byte array if it's a string
    if isinstance(data, str):
        data = data.encode()

    hash_result = []
    transition_matrix = create_transition_matrix(data, num_states)
    print(transition_matrix)
```

```
    for _ in range(hash_length):
        state = np.random.randint(num_states)
        for byte in data:
            next_state = np.random.choice(num_states, p=transition_matrix[state])
            state = next_state

        hash_result.append(state)

    return ''.join(format(x, '02x') for x in hash_result)
```