Farouk Youssef - Ziad Eliwa - Mostafa Abdelwahed - Ali Mohammed

Project - Version Control System

CSCE 2211 - Applied Data Structures

2025

1 Introduction & Motivation

As the needs of developers increased and the huge breakout in code bases that humans produce, a system for tracking changes and organizing the developers workflow. Version control is the software engineering practice of controlling, organizing, and tracking different versions in history of computer files; primarily source code text files, but generally any type of file. Git is the world's leading distributed version control system. Git was developed in 2005 by the Finnish software engineer Linus Trovalds, programmer of Linux Operating System Kernel and author of Git. According to the narrative told by Linus Trovalds, Git was not intended to be a software product, but a side tool that he created for himself in five days to ease his workflow. Now, Git is used by 90% of developers in the world, followed by CVS and Mercurial. Version control systems (VCS) are either centralized or distributed. Centralized VCS are systems that only work locally for one developer, while Distributed VCS connects to a remote repository for multiple developers to work on.For Applied Data Structures Course, we implemented a centralized version control system following Git-style implementation for reference.

2 Project Description

`jit` is our Git-Like VCS intended for organizing developers workflow. `jit` uses the same file and folder structure of Git where files are called *Blobs* (Binary Large Objects), directories are represented as trees, and commits are a snapshot of the working directory specified by the user.[1]

Blobs:

Blobs are used to store file contents such as text, and binary representation of images, videos and other formats. Two files with different names with same content will be mapped into the same blob by the `ObjectStore`.

Tree structure example:

```
tree 4
tree dummy2 444a7207
blob test1.txt a704acaa
blob test2.txt 7d185ae7
blob test4.txt 47ab0cd5
```

Commit structure example:

```
commit 4
author pharaok
timestamp 2025-12-05,19:24:39
message First Commit
tree 87991f17
```

The following commands were implemented:

- `jit init` for initializing the repository in the current directory. No `jit` commands can be run without existence of `.jit` folder.
- `jit add <file/dirname>` adds the specified file or directory to the staging area.
- `jit commit -m <commit-msg>` creates commit from the files that have been added to the staging area.
- `jit branch <branch-name>` creates a branch.

[1]This project is intended for learning purposes only. All rights reserved to the git foundation.

- `jit merge <branch-1> <branch-2>` merges two branches.
- `jit log` shows the commit history of the working repository.
- `jit diff <file-name1> <file-name2>` show the difference between two files.
- `jit diff <commit-hash>` shows the difference between the current working directory and the specified hash.
- `jit status` shows the status of tracked and untracked files in the staging area of the working repository.
- `jit checkout <commit-hash/branch-name>` switched the head pointer to the specified hash or branch name.

Supplementary structures and helpers were introduced for efficient management of software flow:

- `.jit` folder for storage of VCS metadata and repository internals.
- `Object Store` for storage and retrieval of logs, commits, blobs and trees.
- `Index` for tracking changes in current working directory and storing it in the staging area.
- `Refs` for tracking branches.
- `HEAD` pointer to track the current working commit.
- `Murmur3_32` hash function for effective file serialization hashing.

3 Data structures & Algorithms Used

3.1 Merkle Tree (Hash Tree)

A hash tree or Merkle tree is a tree in which every "leaf" node is labelled with the cryptographic hash of a data block, and every node that is not a leaf (called a branch, inner node, or inode) is labelled with the cryptographic hash of the labels of its child nodes. A hash tree allows efficient and secure verification of the contents of a large data structure.

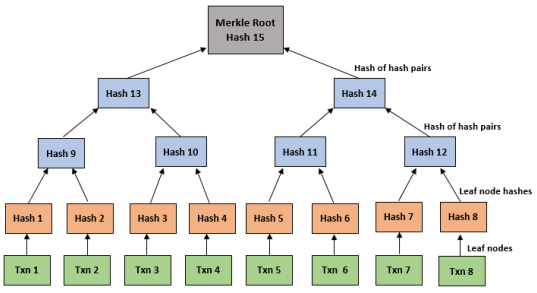

Figure 1: Demonstration of Merkle Tree

`jit` uses Merkle trees as the main data strucure that represents working directories and files where nodes are either subtrees or blobs. Commits stores Merkle trees as their corresponding directory.

3.2 Hash Maps

A hash map is an associative data structure that stores key–value pairs and supports efficient lookup, insertion, and deletion. It uses a hash function to map keys to indices in an underlying array, ensuring average-case constant-time operations. Collisions—when multiple keys hash to the same index—are typically handled using chaining or open addressing. Hash maps are widely used for fast retrieval tasks and provide a flexible alternative to tree-based dictionaries.
`jit` widely uses hash maps in reconstructing directories, hashing files and the staging area in the `Index`.

3.3 Doubly Ended Queue

A doubly ended queue is a linear data structure that allows insertion and removal of elements at both the front and the back in constant time. It is commonly implemented using a circular buffer or a doubly linked list. Deques generalize both stacks and queues, supporting operations like push/pop at either end efficiently. They are useful for sliding-window problems, task scheduling, and algorithms requiring flexible endpoint access.
`jit` uses doubly ended queue in some operations of branches merging.

3.4 Arrays (Vector)

A vector is a dynamic array structure that provides contiguous memory storage and allows random access in constant time. Unlike static arrays, vectors automatically resize when capacity is exceeded, typically by allocating a larger block and copying existing elements. They support efficient end insertions and are widely used due to their cache-friendly memory layout, fast access, and flexible size management.
`jit` uses vectors in implementing other data structures and efficient storage.

3.5 Myers' Diff Algorithm

Myers' Diff Algorithm, introduced by Eugene W. Myers in 1986, is one of the most efficient methods for computing the shortest edit script (SES) between two sequences. The algorithm determines the minimal number of insertions and deletions required to transform one sequence into another.

The algorithm represents the problem as finding a path through an edit graph, where each vertex (i, j) corresponds to prefixes of the two input sequences. A diagonal move represents a match, whereas horizontal and vertical moves represent deletions and insertions, respectively. The goal is to find a path from $(0, 0)$ to (N, M) using the smallest number of non-diagonal steps.

Myers observed that for a given edit distance D, one can compute the *furthest-reaching* point on each diagonal of the edit graph. By iteratively increasing D, the algorithm explores all possible paths of length D until a complete path is found. This approach yields an optimal edit script.

Myers' algorithm runs in $O(ND)$ time, where N is the combined length of the sequences and D is the edit distance of the optimal solution. With a divide-and-conquer strategy, it can also be implemented in linear space.
`jit` uses Myers' Diff algorithm in the comparison between file and commits as implemented in `jit diff <file-name1> <file-name2>` and `jit diff <commit-hash>` commands.

4 Limitations

Limitations of the project is subjected into three main things: (1) networking and remote repository, (2) merge conflicts handling, and (3) security concerns for not using a cryptographic hash. Merge conflicts that are introduced in the projects are split into two parts: we have no differences, then we merge the file, there exist a conflict markers with difference between files are written into the file and left for the user to handle himself.

5 Post-Project Learnings

The main learnings that has resulted from working in this project.

- System Design Experience and exposure to real world software.
- Efficient storage and retrieval in disks.
- CLI software development.
- File serialization and deserialization.

6 Further Work

Further work could be directed to the production of distributed VCS and adding features to `push` and `pull`. Exposure to computer networking, distributed systems architecture and smart protocols are perquisites for further development. In addition, more development in merging algorithms and scenarios for handling merge conflicts. Also, automatic merging scenarios handling can be introduced for efficient user experience.

7 Conclusion

In this project, we implemented `jit`, a Git-like version control system, to gain practical experience with version control concepts, data structures, and algorithms. By building components such as blobs, trees, commits, and the staging area, we explored efficient storage, retrieval, and file comparison using Merkle trees, hash maps, deques, vectors, and Myers' Diff Algorithm. While limitations exist in distributed functionality and merge handling, the project provided valuable insights into system design, CLI development, and the inner workings of modern VCS tools.

8 References

[1] Git documentation: https://git-scm.com/docs
[2] Free Code Camp: How Git works under the hood? https://www.freecodecamp.org/news/git-under-the-hood/

9 Appendix A: Source Code

The source code can be found on the following github repository: github.com/ziad-eliwa/Git-Clone-Version-Control-System.