Exploring the Data Structures and Algorithms behind Version Control Systems

(Student Proposed)

Reece Donovan

Final Year Project
B.Sc. Computer Science

Supervisor: Prof. Ken Brown
Second Reader: Dr. Klaas-Jan Stol

Department of Computer Science
University College Cork

April 2023

Abstract

Declaration of Originality

Acknowledgements

Contents

Chapter 1

Introduction

Version Control is a critical aspect of software development that helps developers keep track of changes made to a codebase. It enables software teams to work collaboratively, efficiently, and accurately on projects, reducing the likelihood of errors and conflicts. In essence, version control is the process of tracking and managing changes to files over time.

Version Control System (VCS) is a software tool that automates the version control process. It provides a centralised repository where developers can store their code, track changes, and collaborate with other team members. The Version Control System ensures that each team member has access to the latest version of the code and can work on it simultaneously.

The impact of **Version Control Systems** on software development has been immense. Before the advent of Version Control Systems, developers used to rely on manual processes to track changes, which was time-consuming and error-prone. With VCS, software teams can work together more efficiently, manage changes more effectively, and deliver better-quality software products.

Version Control Systems have also facilitated the rapid growth of **Continuous Integration** and **Continuous Delivery** (CI/CD), which have become essential systems in modern large scale software development. Overall, Version Control Systems have revolutionised how software is developed, making it easier, faster, and more reliable.

1.1 Motivation

Over the years, Version Control Systems have become essential for software developers, enabling them to collaborate and work more effectively. As the software development industry has evolved, so have Version Control Systems, leading to the creation of many different software solutions, each with unique features and strengths. However, these systems' core data structures and algorithms have remained relatively unchanged.

The performance of Version Control Systems can be crucial in determining the efficiency of software development projects. Slow or inefficient Version Control Systems can result in delays, errors, and even project failure. Therefore, it is important to explore alternative data structures and algorithms that could improve the performance of these systems.

1.2 Objective

Version Control Systems (VCS) have become essential for software developers to collaborate effectively and efficiently. However, many users may not fully comprehend the intricacies of the underlying concepts at the core of these systems.

This report aimed to address this knowledge gap by exploring the underlying data structures and algorithms used to power Version Control Systems and by evaluating potential trade-offs and benefits of alternative approaches. In order to provide insight into how these factors impact system performance, scalability, and overall effectiveness.

In addition, this report aimed to provide a comprehensive overview of the evolution of Version Control Systems, including an overview of the most popular version control solutions throughout the years, such as Source Code Control System (SCCS)—1972, Apache Subversion (SVN)—2000, and Git—2005.

Chapter 2

Background

2.1 What is a Version Control System

A Version Control System (VCS) saves modifications made by individual software developers, allowing them to track their work over time and making the process more accessible. Furthermore, it facilitates sharing data between nodes, where each node can be kept up to date with the latest versions, minimising the need to handle merge conflicts.

Version Control Systems provide several advantages, such as aiding in collaboration among programmers and improving efficiency when working on large groups of mixed products. In addition, VCSs offer an audit trail and easy branching functionality, simplifying team-based development. It also enables an understanding of who has worked on specific pieces or sections during a given period.

By ensuring all changes are appropriately tracked, VCSs promote transparency within teams while maintaining optimal performance levels through streamlined management processes. This helps avoid errors arising from inconsistencies between versions, such as mismatched documents or programming problems resulting from conflicting edit operations.

Significance of Version Control Systems

For most software projects, the source code is a valuable asset that must be protected. Additionally, for many software teams, the source code represents a repository of invaluable knowledge and understanding about the problem that developers have amassed and refined through meticulous effort.

Version control safeguards this precious resource by preventing catastrophes, casual degradation, human error, or unintended consequences from affecting its quality. Software developers working in teams continually write new source code and modify existing code, which is organised into file folders called ”file trees”. Developers may work on different tasks simultaneously, such as implementing new features or fixing unrelated bugs, by altering parts of the file tree.

Version control assists teams in addressing these challenges by tracking each change made by contributors, helping to prevent concurrent work from conflicting with one another. Consequently, any incompatibility can be detected and resolved without impeding team members' progress further down the line, ensuring that changes do not introduce bugs.

2.2 Evolution of Version Control Systems

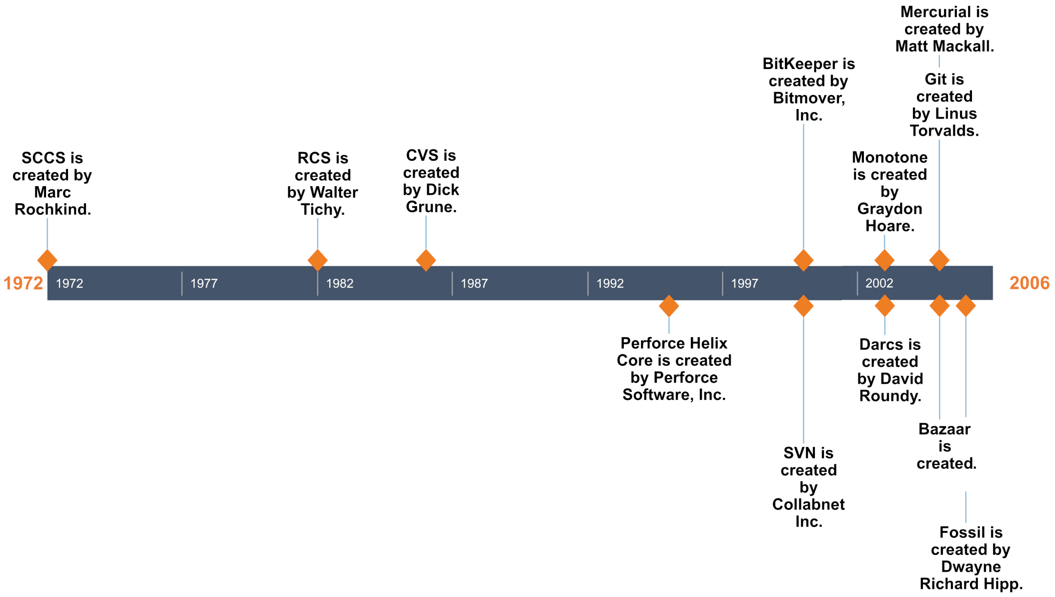
Mercurial is created by Matt Mackall.
BitKeeper is created by Bitmover, Inc.
Git is created by Linus Torvalds.
SCCS is created by Marc Rochkind.
RCS is created by Walter Tichy.
CVS is created by Dick Grune.
Monotone is created by Graydon Hoare.
1972
1972
1977
1982
1987
1992
1997
2002
2006
Perforce Helix Core is created by Perforce Software, Inc.
Darcs is created by David Roundy.
Bazaar is created.
SVN is created by Collabnet Inc.
Fossil is created by Dwayne Richard Hipp.

Figure 2.1: Timeline of the Creation of Version Control Systems [9]

2.2.1 Local Version Control Systems (LVCS)

Local Version Control Systems are systems designed to operate on a single machine, early VCSs were intended to track changes for individual files, and checked-out files could only be edited locally by one user at a time [9]. In addition, they were built on the assumption that all users would log into the same shared Unix host with their own accounts, which was not always possible.

The main benefit of LVCS is that it provides a basic level of version control functionality without requiring any network connectivity. However, this can also be seen as a drawback when you consider that with LVCS, there is no central repository to store the code, which means that each developer has their own copy. This can lead to issues with merging changes and tracking changes across multiple copies.

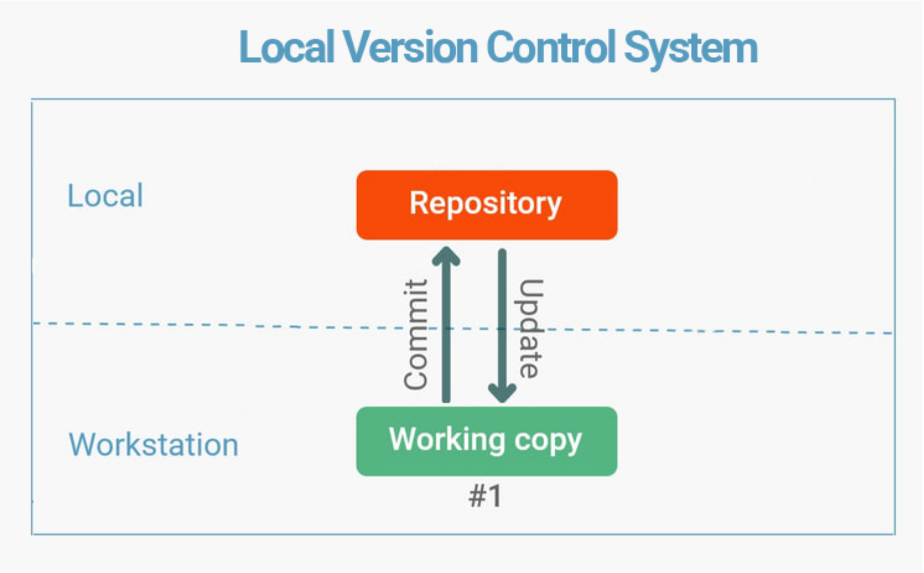

Figure 2.2: Local Version Control System (LVCS)

Source Code Control System (SCCS)

Source Code Control System was released in 1972 and is one of the first successful VCS tools [9]. It was written by Marc Rochkind at Bell Labs, who wanted to solve the problem of tracking file revisions. The tool made tracking down bugs introduced into a program significantly more manageable. SCCS is worth understanding at a basic level because it helped set up modern VCS tools that developers use today.

Architecture

Much like modern VCS tools, SCCS has a set of commands that allow developers to work with the versioning of files. The basic command functionality is:

- Check-in files to track their history.
- Check-out specific file versions for review.

- Check-out specific file versions for editing.
- Check-in new file versions with comments explaining the changes.
- Revert changes made to a checked-out file.
- Basic branching and merging of changes.
- Print a log of a file's version history.

A particular type of file called an `s-file` or a `history file` is created when a file is tracked by SCCS. This file is named with the original filename prefixed with an `s.` and is stored in a subdirectory called `SCCS`.

So a file called `test.txt` would get a history file created in the `./SCCS/` directory with the name of `s.test.txt`. When created, the `s-file` contains the original file contents, a header that contains the file's version number, and some other metadata. There are also checksums stored in the `s-file` that are used to verify the integrity of the file (i.e. to ensure that the file has not been tampered with). The `s-file` content is not encoded or compressed in any way, which is a clear difference from modern VCS tools.

Since the original file's content is now stored in the history file, it can be retrieved into the working directory for review, compilation, or editing. Further changes made to this new copy, such as line additions, modifications and removals, can be checked back into a revised version of the history file, which increments its revision number [9].

Subsequent SCCS check-ins only store only deltas or changes to a file instead of storing entire contents each time; when a check-in is made, subsequent revisions are added onto existing delta tables inside an amended history file (history files do not use compression). This decreases the size of these large histories since they are not using compression on their files, so they take up more space than just having one complete copy that you are tracking with no differences like Word docs etc.

As previously mentioned, SCCS uses the Delta method known as Interleaved Deltas, which allows constant-time checkout regardless of how old your checked-out revision is - i.e., older revisions do not take longer than newer ones.

It is important to note that all files are tracked and checked in separately; there is no way to check in changes on multiple files as a part of one atomic unit - like a commit in Git. Each tracked file has its corresponding history file, which stores revisions. Generally, this means that the version numbers of different files cannot match each other. However, matching revision numbers can be achieved by editing every file at once (even if not all of the changed areas have fundamental changes) and checking them all together. This will increment revision numbers for any modified content, so they will be consistent with their revisions—but it's not comparable to including lots of different pieces within a single commit like you would in Git. In SCCS, these make individual check-ins across separate history folders rather than one extensive report containing everything at once.

When a file is checked out for editing in SCCS, it is locked so that anyone else cannot edit it. This prevents changes from being overwritten by other users and limits development since only one user can edit the file simultaneously.

The software supports branches that store sequences of changes to specific files - these can then be merged back into the original versions or with copies of other branched versions of the same parent branch.

Revision Control System (RCS)

The Revision Control System was released in 1982 as an alternative to the closed-source Source Code Control System. Developed by Walter Tichy and written in C, RCS was released under the GNU General Public License, making it suitable for use in open-source projects.

> ”RCS manages revisions of text documents, in particular source programs, documentation, and test data. It automates the storing, retrieval, logging and identification of revisions.” – Walter Tichy [11]

Architecture

RCS shares many traits with its predecessor[9], including:

- Handling revisions on a file-by-file basis.
- Changes across multiple files can't be grouped together into an atomic commit.
- Tracked files are intended to be modified by one user at a time.
- No network functionality.
- Revisions for each tracked file are stored in a corresponding history file.
- Basic branching and merging of revisions within individual files.

Upon checking a file into RCS for the first time, a corresponding history file is created in the local `./RCS/` directory. The history file is postfixed with a `,v`, so a file named `test.txt` would be tracked by a file called `test.txt,v`[9].

RCS employs a reverse-delta scheme for storing file changes. When a file is checked in, the history file contains a complete snapshot of its contents. Subsequent modifications and check-ins result in RCS calculating a single delta—the difference between the new version of that specific revision and the previously recorded version - and saving it along with an older snapshot if necessary.

The reverse-delta scheme functions by checking out an earlier revision from the newest version and applying consecutive deltas until reaching the desired revision. Starting from the newest version allows quick checkout times, as the current revisions' snapshots are readily accessible.

However, when attempting to access older versions beyond just one recent update (e.g., 1 or 2 old versions), the process becomes considerably more complex. This is because these older versions' snapshots must be calculated against each other before they can be applied to the newest version.

Unlike RCS, SCCS maintains a consistent checkout time for any revision. Additionally, RCS history files do not store checksums, meaning file integrity cannot be guaranteed.

2.2.2 Centralized Version Control Systems (CVCS)

A Centralised Version Control System is a system that enables developers to work together on the same project by storing the primary copy of files in a central repository. This system keeps track of all files and saves information in the local repository. CVCS is called centralised because there is only one central server or repository.

The server maintains a complete record of issues, while clients only maintain a local copy of the shared documents. All developers make their modifications to the repository through checkout. However, only the last version of the files is retrieved from the server, meaning any modifications made will automatically be shared with other developers.

Users can modify in parallel with their local copy of shared documents and sync with the central server to release their contributions and make them visible to other collaborators. However, because centralised version control systems rely on one repository that includes the correct project version, they must restrict write access so that only trusted contributors can commit modifications.

CVCS has some challenges, such as if the central server is inaccessible, users cannot merge their work or save the released modifications. Also, if the central repository is corrupted, everything will be lost. Contributors must be the ones who have writing permissions to perform basic tasks, such as reverting modifications to a previous state, creating or merging branches, or releasing modifications with complete revision history. This limitation affects participation and authorship for new contributors.

So, the main drawbacks of using CVCS are that it requires a network connection to work on the source code, developers must order to contribute to a project, and a single point of failure is an issue when using one server.

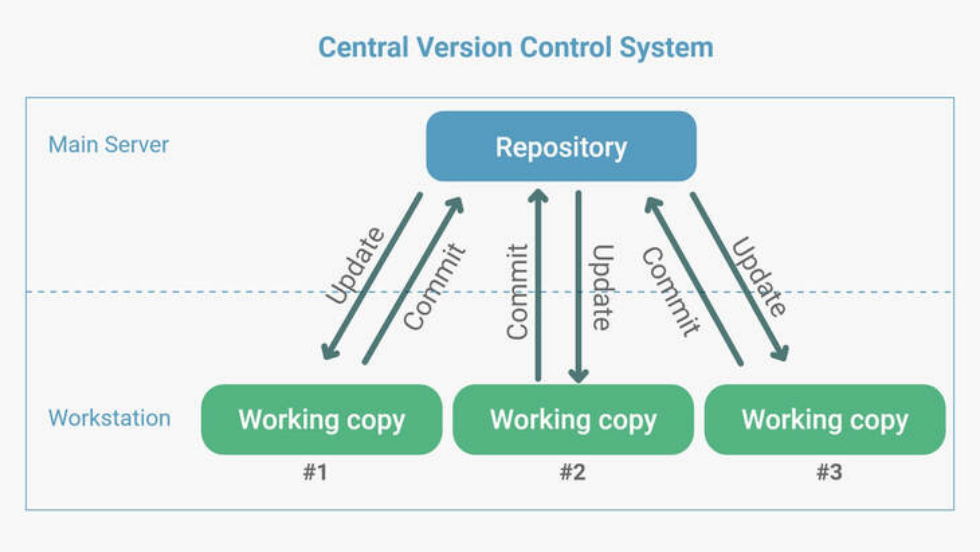

Figure 2.3: Centralized Version Control System (CVCS)

Concurrent Versions System (CVS)

Dick Grune developed the Concurrent Versions System in 1986 to introduce a networking element to version control. Written in `C`, CVS became the first widely used VCS tool that enabled multiple users to work on the same project simultaneously from different locations. This innovation began the second generation of VCS tools and facilitated collaboration among geographically dispersed development teams.

Architecture

CVS functions as a frontend for RCS, providing a set of commands to interact with files in a project while utilising the RCS history file format and commands behind the scenes. CVS allowed multiple developers to check out and work on duplicate files for the first time in history by employing a centralised repository model.

When a project is imported into CVS, each file is converted into a `,v` history file and stored in a central directory referred to as a `module`[9]. Generally, the repository resides on a remote server accessible via a local network or the Internet.

A developer checks out a copy of the module, which is copied to a working directory on their local machine. No files are locked during this process, allowing an unlimited number of developers to check out the module simultaneously. In addition, developers can modify their checked-out files and commit changes as needed.

When a developer commits a change, other developers must update their working copies through a (usually) automated merge process before committing their changes. Occasionally, merge conflicts require manual resolution before a commit can be made. Therefore, CVS also offers the capability to create and merge branches.

Perforce Helix Core

Perforce Helix Core is a proprietary VCS developed, owned, and maintained by Perforce Software Inc., Written in `C` and `C++`; it was initially released in 1995. Although primarily designed for a centralised model, it also offers a distributed model option. Helix Core is commonly used by large companies managing substantial content with large binary files, such as in the video game development industry. Although typically cost-prohibitive for smaller projects, Perforce provides a free version for teams of up to five developers.

Architecture

Perforce Helix Core utilises a server/client model. The server, a process called `p4d`, listens for incoming client connections on a designated port, typically port `1666`. The client, a process called `p4`, is available in both command-line and GUI variants. Users run the `p4` client to connect to the server and issue commands. Support for various programming language APIs, including Python and Java, enables automated issuance and processing of Helix Core commands via scripts. Integrations are also available for IDEs like Eclipse and Visual Studio, allowing users to work with version control within those tools.

The Helix Core Server manages repositories called depots, which store files in directory trees similar to Apache Subversion (SVN). Clients can check out sets of files and directories from the depots into local copies called `workspaces`. The atomic unit used to group and track changes in Helix Core depots is called the `changelist`, which is analogous to Git commits. In addition, Helix Core implements two similar forms of branching: `branches` and `streams`. While branches represent separate lines of development history, a stream is a branch with added functionality that Helix Core uses to provide recommendations on best merging practices throughout the development process.

When a file is added for tracking, Helix Core classifies it using a `file type` label. The two most commonly used file types are text and binary. For binary files, the entire file content is stored each time the file is saved. This is a common VCS tactic for handling binary files, which are not amenable to the normal merge process, as manual conflict resolution is typically impossible.

Only the deltas (changes between revisions) are stored for text files. Text file history and deltas are stored using the Revision Control System (RCS) format, which tracks each file in a corresponding `,v` file in the server depot. This is similar to Concurrent Versions System (CVS), which also leverages RCS file formats for preserving revision history. Files are often compressed using gzip when added to the depot and decompressed when synced back to the workspace.

Apache Subversion (SVN)

Subversion was created in 2000 by CollabNet Inc. and is now maintained by the Apache Software Foundation. Written in `C`, it was designed to offer a more robust centralised solution than Concurrent Versions System (CVS)[9].

Architecture

Similar to Concurrent Versions System (CVS), Subversion employs a centralised repository model, requiring remote users to have a working network connection to commit their changes to the central repository. However, in contrast to CVS, where a commit operation could fail midway due to a network outage and leave the repository in a corrupted and inconsistent state, Subversion ensures the repository remains consistent. Additionally, a Subversion commit or revision can include multiple files and directories, enabling users to track sets of related changes together as a grouped unit, as opposed to previous storage models that tracked changes separately for each file.

Subversion utilises a storage model called `FSFS (File System atop the File System)` for tracked files. The model's name originates from its database structure, which mirrors the operating system's file and directory structure. However, subversion's unique feature is its filesystem design, which tracks not only files and directories but also different versions of these files and directories as they change over time. This creates a filesystem with an added time dimension. Moreover, Subversion treats folders as first-class citizens, allowing empty folders to be committed, unlike other systems such as Git, where empty folders go unnoticed.

When a Subversion repository is created, a (nearly) empty database of files and folders is established [9]. Next, a directory called `db/revs` is created to store all revision tracking information for the committed files. Each commit, which may include changes to multiple files, is stored in a new file in the `revs` directory, named with a sequential numeric identifier starting with `1`. When a file is committed for the first time, its entire content is stored. Subsequent commits of the same file store only the changes, also known as `diffs` or `deltas`, to save space. Additionally, `deltas` are compressed using `lz4` or `zlib` compression algorithms to reduce their size further.

By default, this approach is only employed to a certain extent. Although storing file deltas instead of the entire file each time saves storage space, it adds time to checkout and commit operations, as all the deltas must be combined to recreate the file's current state. For this reason, Subversion stores up to 1023 deltas per file before saving a new complete copy of the file, striking a balance between storage and speed.

2.2.3 Distributed Version Control Systems (DVCS)

Distributed Version Control Systems (DVCS) were developed to address the limitations of Centralized Version Control Systems (CVCS), enabling more effective branching and merging, seamless local VCS operations, and improved collaboration among developers. Due to these limitations associated with Centralized Version Control Systems (CVCS), Distributed Version Control Systems have become widely adopted in Open Source Software (OSS) projects.

DVCS is designed to function in two ways: it maintains file histories locally on each device and can synchronise local user modifications with servers when necessary, enabling the sharing of these modifications with others. In DVCS, developers can work independently or collaboratively on the same project, as they have access to all necessary repositories. In addition, any repository can be cloned from another, meaning no repository holds greater importance than others.

In response to the need for more advanced versioning of software artefacts, several Distributed Version Control Systems, such as Mercurial, and Git, emerged in the software field. Many OSS projects have widely adopted these tools. DVCS operations are significantly faster than those in CVCS, as they are performed locally, while CVCS operations require remote connections. Some experts believe that distributed systems will eventually replace centralised ones due to their suitability for more extensive projects involving independent developers seeking full functionality, even without a network connection. In addition, distributed systems offer advantages such as saving earlier drafts of work without publicly releasing or sharing them with others.

Collaboration between team members and the ability for individual developers to serve as servers or clients are essential features of version control systems, allowing developers to work on source code without being connected to a central or remote repository.

However, DVCS introduces some challenges, including the lack of a coherent version numbering system due to the absence of a centralised versioning server. Instead, DVCS relies on hash modifications or a unique GUID. This lack of a central server also complicates system backup.

The two most common criticisms of DVCS are the unavailability of pessimistic locks and weak support for binary files. Despite these disadvantages, the transition from centralised to decentralised version control systems continues, as developers can work offline and efficiently work incrementally. This flexibility enables developers to assume multiple roles, such as developing new tasks or fixing errors, leading to exploratory coding and greater freedom in their workflow while maintaining control over their code's release schedule.

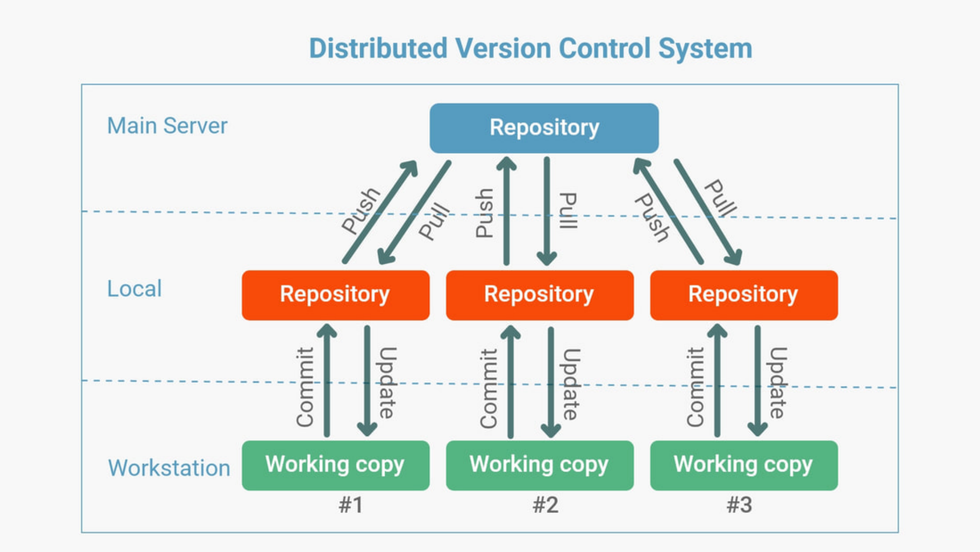

Figure 2.4: Distributed Version Control System (DVCS)

Git

Git, created in 2005 by Linus Torvalds (also the creator of Linux), is primarily written in C with some shell scripts. Git was initially developed for the Linux codebase and has since become the most popular VCS in use today due to its features, flexibility, and speed. Torvalds explains that Git is a robust set of tools with many options, and its usage is often determined by what works best for collaboration rather than technical limitations.

> ”You can do a lot of things with Git, and many of the rules of what you should do are not so much technical limitations but are about what works well when working together with other people. So Git is a very powerful set of tools, and that can not only be overwhelming at first, it also means that you can often do the same (or similar) things different ways, and they all ’work.’” – Linus Torvalds [1]

Git repositories are commonly hosted on local servers and cloud services, forming the backbone of a broad set of DevOps tools available from popular service providers, including GitHub, BitBucket, GitLab, and many others [9].

Architecture

As a Distributed Version Control Systems (DVCS), Git ensures that no repository copy needs to be designated as the centralised copy—instead, all copies are created equal. This contrasts with second-generation VCS, which relies on a centralised copy for users to check in and out.

This design allows developers and coding partners to share changes directly with each other before merging their changes into an official branch, fostering a flexible distributed workflow for team collaboration.

Moreover, developers can commit changes to their local copy of the repository without other repositories knowing about it. This enables commits without a network or internet connection, allowing developers to work offline until they are ready to push their changes to other repositories for review, testing, or deployment.

When a file is added for tracking with Git, it is compressed using the `zlib` compression algorithm and hashed using a `SHA-1` hash function [9]. This generates a unique hash value corresponding to the file content, which Git stores in an object database located in the hidden `.git/objects` folder. These files, called `Git blobs`, are created each time a new file (or a changed version of an existing file) is added to the repository.

Git uses a staging index that functions as a temporary space for changes being readied for a commit. When changes are set to be committed, their compressed data is referenced in a unique index file, appearing as a tree object. `Trees` in Git link blob objects to actual file names, file permissions, and connections to other trees, signifying the status of a specific collection of files and directories. After all associated changes have been staged for commit, the index tree can be committed to the repository, generating a commit object within the Git object database [9].

A commit denotes the primary tree for a specific revision, and the commit author, email address, date, and a descriptive commit message. Each commit also retains a reference to its preceding commit/commits, constructing a record of the project's evolution.

Git objects, including blobs, trees, and commits, are all compressed, hashed, and saved in the object database based on their respective hash values. These standalone objects avoid using diffs for space conservation, making Git highly efficient since the complete content of every file revision is readily available as an individual object.

Nonetheless, particular operations, such as pushing commits to a remote repository, storing an excessive number of objects, or manually executing Git's garbage collection command, may prompt Git to reorganise objects into pack files. This packing procedure compresses inverse diffs to remove duplicate content and minimise size. This leads to the creation of `.pack` files containing the object data, each paired with a corresponding `.idx` (index) file that references the packed objects and their positions within the pack file. These pack files are transmitted across the network when branches are pushed to or fetched from remote repositories. When pulling or retrieving branches, the pack files are decompressed to generate loose objects in the object repository.

Mercurial

Mercurial, created in 2005 by Matt Mackall and written in `Python`, initially aimed to host the codebase for Linux, but Git was chosen instead [9]. As the second most popular distributed VCS after Git, Mercurial is used far less frequently.

Architecture

Comparable to Git, Mercurial is a Distributed Version Control Systems (DVCS) that enables multiple developers to work on seperate copies of the same project. Although Mercurial utilises many similar technologies as Git, such as compression and `SHA-1` hashing, it does so in distinct ways.

When a new file is tracked in Mercurial, a corresponding `revlog` (revision log) file is generated for it in the hidden `.hg/store/data/` directory. `Revlog` files can be viewed as advanced versions of the history files employed by older VCSs such as Concurrent Versions System (CVS), Revision Control System (RCS), and Source Code Control System (SCCS).

Unlike Git, which generates a new blob for each version of every staged file, Mercurial merely creates a new entry in the revlog for that file. To conserve space, each new entry contains only the delta (modifications) from the preceding version. Once a certain number of deltas is achieved, a complete snapshot of the file is stored again, minimising lookup time when applying numerous deltas to reconstruct a specific file revision.

These file revlogs are named to correspond with the files they monitor but are postfixed with `.i` and `.d` extensions. The `.d` files hold the compressed `delta` content, while the `.i` files function as `indexes` to locate distinct revisions within the `.d` files swiftly. For small files with few revisions, both the `indexes` and `content` are stored in `.i` files. Revlog file entries are compressed for efficiency and hashed for identification, with the hash values referred to as `nodeids`.

Chapter 3

Design

This chapter examines various data structures and algorithms with key aspects that may make them suitable for incorporation into a `Version Control System`. We also assess what metrics are relevant to facilitate the comparison of these data structures and algorithms.

3.1 Data Structures

Data structures are objects that can be used to store, organize, and manipulate large amounts of data. They play a crucial role in computer science and are fundamental building blocks of many software systems, including `Version Control Systems`. The choice of data structure is critical to the overall performance and scalability of a `Version Control System`.

In order to evaluate the suitability of a data structure for implementation in a `Version Control System`, it is necessary to consider several core aspects. Firstly, `operational efficiency`, which refers to the time and space complexity of the basic operations performed by the data structure, is a crucial consideration. The efficiency of these operations can have a significant impact on the overall performance of the `Version Control System`.

Another important aspect is the data structure's `structural specificity`, which encompasses how data is stored and organized within the structure. This is critical because the structural specifics can affect the ease of implementation and the ability to efficiently perform operations such as `insertions`, `deletions`, and `updates`.

Lastly, the `implementation details` must be considered, including the ease of implementation and compatibility with the programming language. A data structure that is straightforward to implement and easy to maintain/iterate upon will result in a more streamlined and efficient `Version Control System`.

3.1.1 Linked List

A `Linked List` is a linear collection of data elements, called nodes, each pointing to the next node by means of a pointer. The `Linked List` is the most sought-after data structure when it comes to handling dynamic data elements [5].

There are two main types of `Linked Lists`: Singly-linked lists (SLL) and Doubly-linked lists (DLL), but we will only be considering the Doubly-linked lists (DLL) when we reach the Implementation chapter.

Singly-linked lists (SLL)

- `SLL` nodes contain two fields: `data` field and `next` pointer field.
- Traversal of a `SLL` can be done using the `next` pointer field only. Meaning, the `SLL` can be traversed in only one direction, from the first node to the last node.
- The `SLL` occupies less memory than a `DLL` because it does not contain a `prev` pointer field.
- `SLL` is preferred over `DLL` when it comes to the execution of stack and queue operations.
- `SLL` is also preferred over `DLL` to save memory when a searching operation is not required.

Table 3.1: Efficiency Analysis of Singly-Linked List Operations

Operation	Worst Case	Average Case
Access	$O(n)$	$O(n)$
Search	$O(n)$	$O(n)$
Insert (at Head)	$O(1)$	$O(1)$
Delete (at Head)	$O(1)$	$O(1)$
Insert (at Current)	$O(1)$	$O(1)$
Delete (at Current)	$O(1)$	$O(1)$
Insert (at Tail)	$O(n)$	$O(n)$
Delete (at Tail)	$O(n)$	$O(n)$

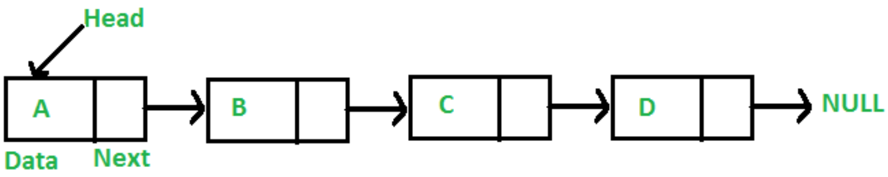

Figure 3.1: Singly Linked List (SLL) [12]

Doubly-linked lists (DLL)

- `DLL` nodes contain three fields: `data` field, `prev` pointer field and `next` pointer field.
- Traversal of a `DLL` can be done using the `next` pointer field or the `prev` pointer field. Meaning, the `DLL` can be traversed in both directions, from the first node to the last node and vice versa.
- The `DLL` occupies more memory than a `SLL` because it contains a `prev` pointer field.

Table 3.2: Efficiency Analysis of Doubly-Linked List Operations

Operation	Worst Case	Average Case
Access	$O(n)$	$O(n)$
Search	$O(n)$	$O(n)$
Insert (at Head)	$O(1)$	$O(1)$
Delete (at Head)	$O(1)$	$O(1)$
Insert (at Current)	$O(1)$	$O(1)$
Delete (at Current)	$O(1)$	$O(1)$
Insert (at Tail)	$O(1)$	$O(1)$
Delete (at Tail)	$O(1)$	$O(1)$

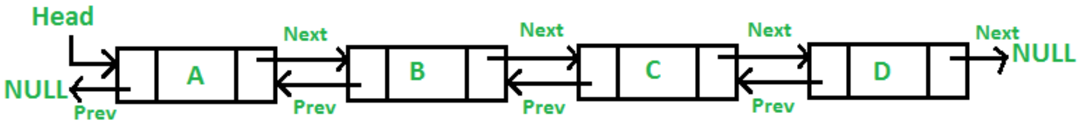

Figure 3.2: Doubly Linked List (DLL) [12]

Advantages

- Efficient insertion and deletion operations at the beginning, end, and middle of the list.
- Easy traversal in both directions, allowing for simpler and more flexible algorithms.
- Dynamic size, allowing the list to grow and shrink as needed during runtime.
- No need for contiguous memory allocation, which allows for better memory utilisation.
- Simplified implementation compared to more complex data structures.

Disadvantages

- Higher memory overhead due to the storage of two pointers for each node.
- Slower random access compared to arrays or hash tables, as elements must be traversed sequentially.
- No inherent support for efficient searching, leading to linear search times.

Summary

A `Doubly Linked List` could be used as the core data structure, but it has some limitations. The sequential nature of the data structure makes it easy to maintain a linear history of changes and revert to previous versions. However, the lack of efficient searching and random access capabilities can slow down operations when dealing with large repositories or complex branching scenarios.

3.1.2 Binary Tree

A `Binary Tree` is a hierarchical data structure in which each node has at most two child nodes, arranged in a way that the value of the node to the left is less than or equal to the parent node and the value of the node to the right is greater than or equal to the parent node. This ordering property ensures efficient search, insertion, and deletion operations. There are several types of binary trees, such as `binary search trees`, `AVL trees`, and `red-black trees`, each with different balancing mechanisms to maintain tree height and performance.

Each node in a `Binary Tree` contains the following elements:

1. Data: The data stored in the node.

2. Left child: A pointer to the left child node.

3. Right child: A pointer to the right child node.

Table 3.3: Efficiency Analysis of Binary Tree Operations

Operation	Worst Case	Average Case
Search	$O(n)$	$O(\log n)$
Insert	$O(n)$	$O(\log n)$
Delete	$O(n)$	$O(\log n)$

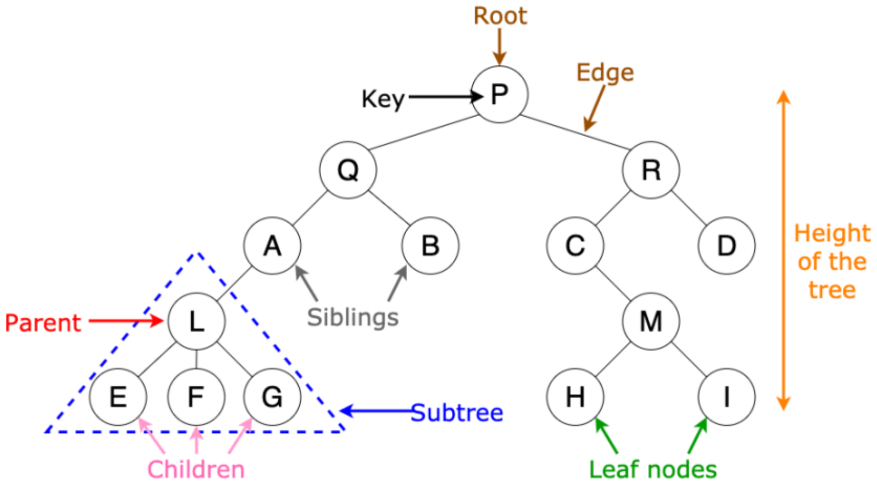

Figure 3.3: Binary Tree [2]

Advantages

- Efficient search, insertion, and deletion operations when the tree is balanced.
- Hierarchical structure allows for natural representation of hierarchical relationships or data with partial order.
- Can be easily traversed in various orders (e.g., inorder, preorder, and postorder) to suit different needs.
- No need for contiguous memory allocation, which allows for better memory utilisation.
- Provides the foundation for more advanced tree structures, like B-trees or trie, that can be used for advanced indexing or searching.

Disadvantages

- Unbalanced trees can lead to degraded performance.
- Requires more memory overhead compared to linear data structures due to the storage of pointers for each node.
- Less suitable for representing non-hierarchical or unordered data.
- Can be more complex to implement and maintain compared to simpler data structures.

Summary

A `Binary Tree` could be used as the core data structure, but it may not be the most suitable choice due to its hierarchical nature. While it can efficiently store and manage version history when dealing with linear or partially ordered data, version control systems often require support for complex branching and merging scenarios, which may not be well-suited for a binary tree.

Additionally, the need for balancing mechanisms to maintain tree height and performance can add complexity to the implementation and maintenance of the system. `Directed Acyclic Graphs` or other more advanced data structures might be more appropriate for handling complex functionality.

3.1.3 Hash Table

A `Hash Table` is a data structure that uses a hash function to map keys to their corresponding values. It is an efficient way to implement an associative array, where keys are used to look up values.

The basic idea behind a `Hash Table` is to use a hash function to map a key to an index in an array, called a bucket, where the corresponding value can be found or stored. The process of mapping a key to an index is called `hashing`.

Each element in a `Hash Table` consists of:

1. Key: This is the value used to look up a corresponding element in the `Hash Table`.
2. Value: This is the value associated with the key that is stored in the `Hash Table`.

When a new element is added to a `Hash Table`, the key is passed through a `hash` function which produces an `index` (also called a hash value or bucket) where the element is stored. When a value is to be retrieved, the key is passed through the same hash function, and the resulting `index` is used to look up the corresponding value in the `Hash Table`.

Table 3.4: Efficiency Analysis of Hash Table Operations

Operation	Worst Case	Average Case
Search	$O(n)$	$\Theta(1)$
Insert	$O(n)$	$\Theta(1)$
Delete	$O(n)$	$\Theta(1)$

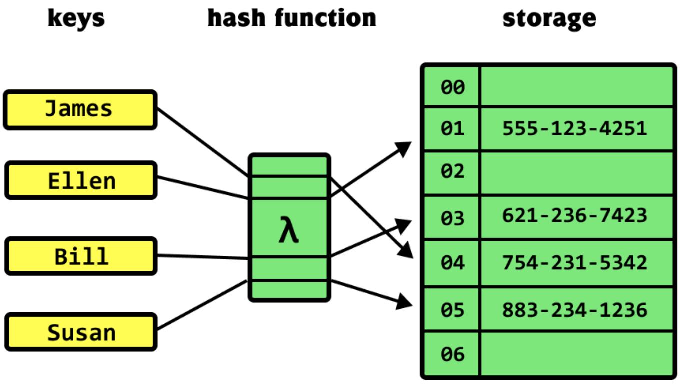

Figure 3.4: Hash Table [8]

Advantages

- Fast average-case performance for search, insertion, and deletion operations.
- Supports efficient key-based lookups and direct access to values.
- Can be easily resized to accommodate a growing number of key-value pairs, maintaining constant-time complexity.
- Suitable for storing unordered or non-hierarchical data.
- Provides a foundation for more advanced data structures, like distributed hash tables or bloom filters, used in various applications.

Disadvantages

- Requires a good hash function to ensure uniform key distribution and avoid performance degradation due to collisions.
- Higher memory overhead compared to linear data structures, as the underlying array needs to be larger than the number of stored key-value pairs to maintain performance.
- No inherent support for ordered traversal or range queries, as keys are not stored in a sorted manner.

Summary

While `Hash Tables` can provide fast and efficient key-based lookups, they may not be the most suitable data structure for the core of a `Version Control System`. The lack of inherent support for ordered traversal or range queries can make it difficult to efficiently handle complex branching and merging scenarios that are common in version control systems.

`Directed Acyclic Graphs` or other more advanced data structures are often better suited for handling complex branching and merging operations, as they can provide more efficient support for ordered traversal and range queries. Additionally, Version Control Systems typically need to maintain relationships between revisions, which is not a natural fit for the unordered nature of hash tables.

In summary, although `Hash Tables` can provide fast key-based lookups and efficient performance in certain scenarios, they may not be the most suitable or scalable choice for the core data structure of a version control system due to their unordered nature and lack of support for ordered traversal or range queries.

3.1.4 Directed Acyclic Graph (DAG)

A `Directed Acyclic Graph (DAG)` is a data structure that consists of nodes connected by directed edges, forming a graph with no cycles. This means that it is impossible to start at a node and follow a sequence of directed edges that leads back to the same node. DAGs are particularly useful for representing dependencies, partial orders, or processes that have a specific sequence of events. For example, in the context of a version control system, a DAG can be used to represent the history of changes made to a project, with nodes representing commits and edges representing parent-child relationships between commits.

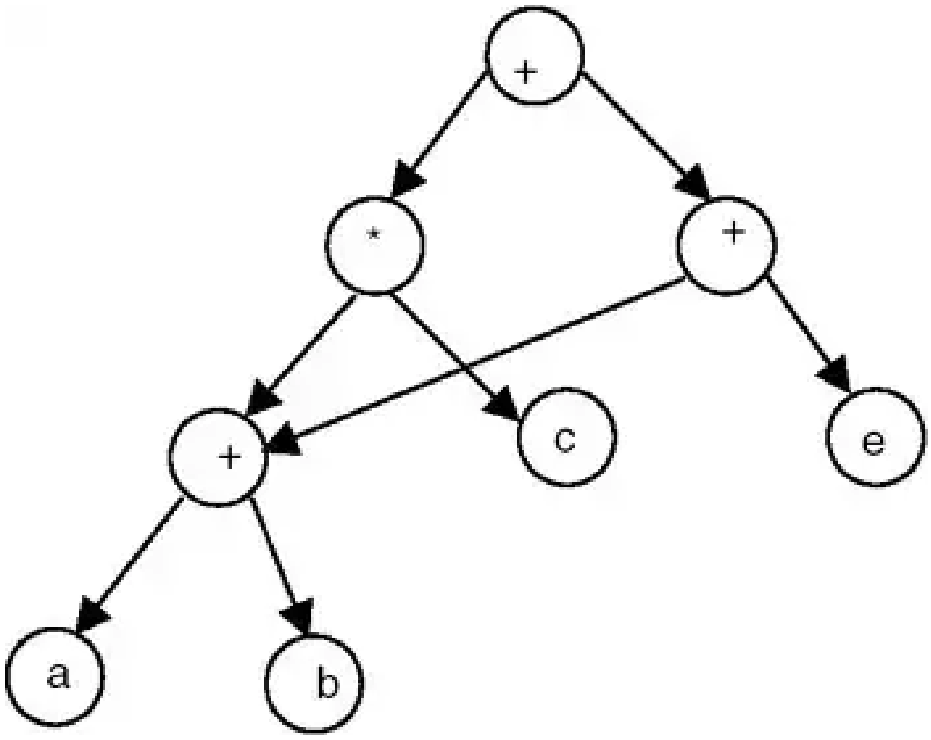

Figure 3.5: Directed Acyclic Graph (DAG) [10]

Advantages

- Efficiently represents complex branching and merging scenarios common in Version Control Systems.
- Can naturally model dependencies, partial orders, or processes with a specific sequence of events.
- Enables efficient algorithms for topological sorting, which can be useful for ordering commits or resolving dependencies.
- No need for contiguous memory allocation, which allows for better memory utilisation.
- Provides a more expressive and flexible data structure compared to linear or hierarchical structures, making it suitable for various applications.

Disadvantages

- Can be more complex to implement and maintain compared to simpler data structures.
- Requires more memory overhead compared to linear data structures due to the storage of multiple pointers for each node.
- Finding the shortest or most efficient path between nodes can be computationally expensive in large graphs.
- No inherent support for fast key-based lookups or direct access to values, as nodes are not indexed by a specific key.

Summary

`Directed Acyclic Graphs` are particularly well-suited for use as the core data structure in a version control system. Their ability to efficiently represent complex branching and merging scenarios allows for more expressive and flexible management of project history. Additionally, the natural modelling of dependencies and partial orders enables efficient algorithms for tasks such as topological sorting and dependency resolution, which are common in version control systems. Overall, the advantages of DAGs in representing complex relationships and dependencies make them a suitable and scalable choice for the core data structure of a version control system.

3.2 Algorithms

3.2.1 Searching

`Search algorithms` are computational methods to locate specific items, solve problems, or explore and analyse data structures. They are essential tools in computer science, artificial intelligence, and various other domains, as they facilitate efficient processing and management of large datasets. By systematically searching through data or traversing complex structures, these algorithms identify target elements, optimal solutions, or specific patterns.

Linear Search

`Linear Search` is a simple and straightforward search algorithm that works on unsorted and sorted lists. It sequentially checks each element in the list until the desired element is found or the end of the list is reached [7]. The algorithm compares each element in the list with the target value, starting from the first element and moving through the list one element at a time. Linear Search has an average case time complexity of $O(n)$ and a worst case time complexity of $O(n)$, where n is the number of elements in the list.

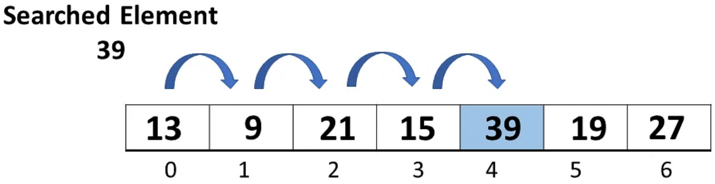

Figure 3.6: Linear Search Algorithm [6]

Advantages

- Simple and easy to implement.
- Works on unsorted and sorted lists.
- No additional memory requirements or data structure modifications needed.
- Performs well for small data sets or when the target element is near the beginning of the list.

- Can be used as a building block for more advanced search algorithms.

Disadvantages

- Inefficient for large data sets, as it has to check each element in the list.
- Slower than other search algorithms for sorted lists, such as binary search.
- Does not take advantage of any existing order in the list.
- May not be suitable for real-time or performance-critical applications.

Binary Search

`Binary Search` is an efficient search algorithm that works on sorted lists [4]. It repeatedly divides the list in half, comparing the middle element with the target value. If the middle element is equal to the target, the search is successful. If the target value is less than the middle element, the search continues in the left half of the list, otherwise in the right half. This process is repeated until the target value is found or the remaining list becomes empty. Binary Search has an average case time complexity of $O(\log n)$ and a worst case time complexity of $O(\log n)$, where n is the number of elements in the list.

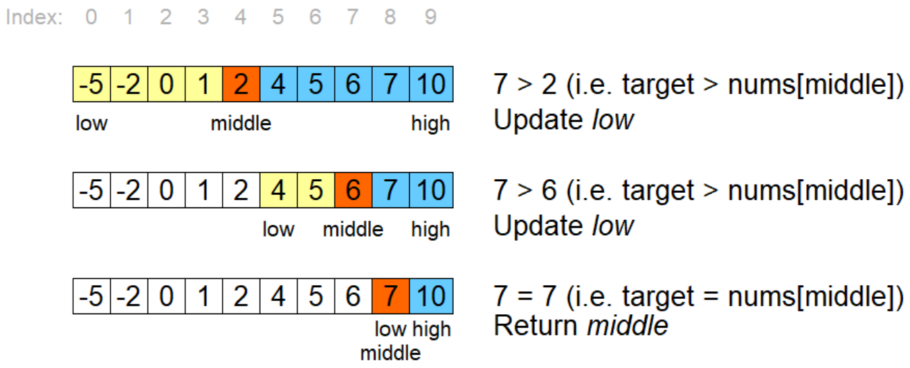

Figure 3.7: Binary Search Algorithm [3]

Advantages

- Efficient search algorithm for sorted lists, with a time complexity of $O(\log n)$.
- Takes advantage of the existing order in the list.

- Requires fewer comparisons than linear search for large data sets.
- Can be implemented iteratively or recursively.
- Provides a foundation for more advanced search algorithms, like interpolation search.

Disadvantages

- Requires the list to be sorted in ascending order beforehand.
- Not suitable for unsorted lists or lists with frequently changing data.
- Can be more complex to implement compared to linear search.
- Does not work well with large lists stored on slow access media (e.g., hard drives) due to multiple random access operations.

Depth First Search (DFS)

`Depth First Search` is a graph traversal algorithm that explores as far as possible along a branch before backtracking. It can be implemented using recursion or an explicit stack data structure. Starting from a source node, DFS visits a node and marks it as visited. Then, it recursively explores each unvisited neighbour of the current node, treating the neighbor as the new current node. The algorithm continues this process until all reachable nodes have been visited. DFS has an average case time complexity of $O(V + E)$ and a worst case time complexity of $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph.

Advantages

- Can be used to find connected components, cycles, and paths in a graph.
- Effective for searching large, sparsely connected graphs.
- Can be implemented using recursion or an explicit stack data structure.
- Visits nodes in a linear and natural order.
- Can be adapted for other graph traversal problems, like topological sorting.

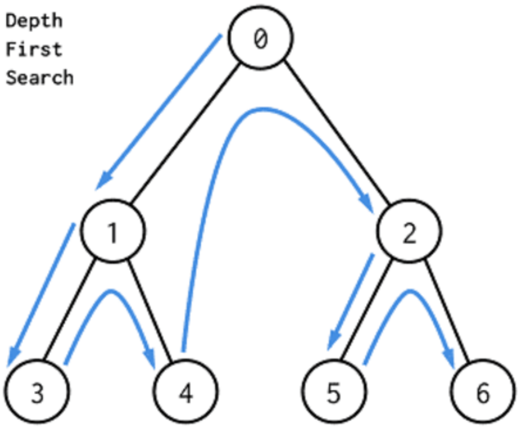

Figure 3.8: Depth First Search Algorithm [13]

Disadvantages

- May consume a large amount of memory for deep graphs when implemented recursively.
- Can get stuck in cycles if not properly implemented with visited node tracking.
- Does not always find the shortest path in weighted graphs.
- May be less intuitive than Breadth First Search for certain problems.

Breadth First Search (BFS)

`Breadth First Search` is a graph traversal algorithm that visits all nodes at the same level before moving on to the next level. It can be implemented using a queue data structure. Starting from a source node, BFS visits and marks the node as visited. Then, it enqueues all unvisited neighbors of the current node. Finally, the algorithm dequeues the next node from the front of the queue and repeats the process until the queue is empty, ensuring that all reachable nodes have been visited. BFS has an average case time complexity of $O(V + E)$ and a worst case time complexity of $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph.

Advantages

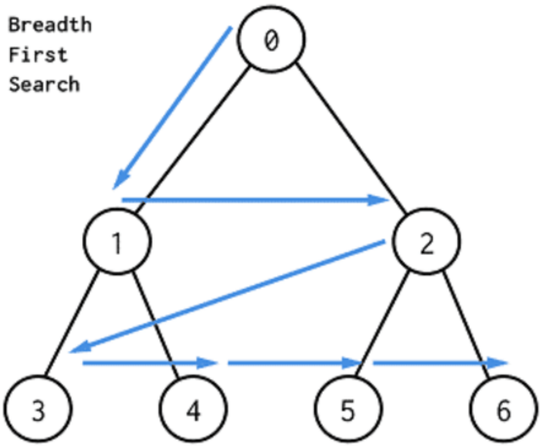

Figure 3.9: Breadth First Search Algorithm [13]

- Can be used to find the shortest path in unweighted graphs or determine the level of each node from the source node.
- Effective for searching large, densely connected graphs.
- Can be implemented using a queue data structure, avoiding recursion.
- Visits nodes in a level-wise order, which can be more intuitive for certain problems.
- Can be adapted for other graph traversal problems, like bipartite graph checking.

Disadvantages

- Can consume a large amount of memory for densely connected graphs due to the use of a queue.
- Not well-suited for finding all paths or connected components in a graph.
- Does not work efficiently for searching large, sparsely connected graphs.
- Can be slower than `Depth First Search` for certain problems, such as finding cycles or connected components.

3.2.2 Diffing Algorithms

`Diffing algorithms`, also known as difference algorithms or delta algorithms, are specialised computational methods designed to identify and highlight the differences between two sets of data. These algorithms compare input sequences, such as text, code, or binary data, to efficiently determine the changes needed to transform one sequence into the other. By detecting additions, deletions, and modifications, diffing algorithms enable the tracking and management of revisions in a concise and comprehensible manner.

Myers Diff Algorithm

The `Myers diffing algorithm`, developed by Eugene W. Myers, is an efficient algorithm for comparing two sequences (such as lines in text files) and finding the shortest edit script that transforms one sequence into the other. The algorithm is based on the concept of an edit graph, where each node represents an element in the sequences, and the edges represent insertions, deletions, or matches. The algorithm finds the shortest path through this graph, also known as the `Longest Common Subsequence (LCS)` path, which represents the minimal set of edits required to transform one sequence into the other.

The time and space complexity of the `Myers Diff` algorithm is $O(ND)$, where N is the sum of the lengths of both inputs, and D is the size of the minimum edit script that converts one input to the other. When the number of differences is small, various optimisations can be applied to improve complexity up to $O(N \log N + D^2)$ time and $O(N)$ space.

Advantages

- Efficiently finds the shortest edit script, reducing the number of changes needed to transform one sequence into the other.
- Can handle large input sequences, as it has a linear space complexity.
- Can be adapted for various applications, such as comparing text files, source code, or DNA sequences.
- The algorithm is well-established and widely used, making it a reliable choice for many diffing tasks.

Disadvantages

- Can produce suboptimal diffs for specific cases, such as when comparing sequences with many common elements but in a different order.

- The algorithm's implementation can be complex and difficult to understand for those unfamiliar with graph-based algorithms.

- May not be the most efficient algorithm for all diffing tasks, as other algorithms may produce better results in specific cases.

- The quality of the output can be sensitive to the choice of the underlying `Longest Common Subsequence` algorithm.

Patience Diff Algorithm

The `Patience Diffing algorithm`, developed by Bram Cohen, is a diffing algorithm that aims to produce more human-readable diffs by focusing on finding the `Longest Increasing Subsequence (LIS)` in the two sequences. The algorithm first identifies unique matching lines (or elements) between the sequences and sorts them by their position in the first sequence. Then, it finds the `LIS` in the sorted list of positions from the second sequence. The algorithm recursively applies this process to the unmatched portions of the sequences until the entire diff is produced.

The time complexity of the `Patience Diff` algorithm is $O(N \log N)$, where N is the sum of the lengths of both inputs.

Advantages

- Produces more human-readable diffs, as it focuses on preserving the order of common elements.

- Can handle large input sequences, as it has a linearithmic time complexity.

- Can be used as a standalone algorithm or in combination with other diffing algorithms, such as the Myers algorithm, to improve the quality of the output.

- The algorithm is relatively simple and straightforward to implement compared to some other diffing algorithms.

- Works well for cases where the sequences have many common elements but in a different order.

Disadvantages

- May produce suboptimal diffs for specific cases, such as when comparing sequences with many non-unique elements.
- The algorithm's performance depends on the choice of the underlying `LIS` algorithm.
- May not be the most efficient algorithm for all diffing tasks, as other algorithms may produce better results in specific cases.
- Not as widely used or well-established as some other diffing algorithms, such as the `Myers algorithm`.

Chapter 4

Implementation

Chapter 5

Evaluation

Chapter 6

Conclusion

List of Figures

List of Tables

Bibliography

[1] J. Cloer. "10 years of git: An interview with git creator linus torvalds." (Aug. 2019), [Online]. Available: https://www.linux.com/news/10-years-git-interview-git-creator-linus-torvalds/.

[2] C. McMahon. "Understanding binary search trees." (Jul. 2020), [Online]. Available: https://dev.to/christinamcmahon/understanding-binary-search-trees-4d90.

[3] J. P. "How to do a binary search in python." (Nov. 2020), [Online]. Available: https://learncodingfast.com/binary-search-in-python/.

[4] O. Popovi. "Binary search in python." (Jun. 2020), [Online]. Available: https://stackabuse.com/binary-search-in-python/.

[5] A. S. Ravikiran. "Linked list in a data structure: All you need to know." (Nov. 2022), [Online]. Available: https://www.simplilearn.com/tutorials/data-structure-tutorial/linked-list-in-data-structure.

[6] A. S. Ravikiran. "What is linear search algorithm?" (Feb. 2023), [Online]. Available: https://www.simplilearn.com/tutorials/data-structure-tutorial/linear-search-algorithm.

[7] Shaalaa. "Discuss linear search algorithm." (), [Online]. Available: https://www.shaalaa.com/question-bank-solutions/discuss-linear-search-algorithm-algorithm-for-searching-techniques_229584.

[8] K. Stemmler. "Hash tables: What, why, how to use them." (Jan. 2022), [Online]. Available: https://khalilstemmler.com/blogs/data-structures-algorithms/hash-tables/.

[9] J. Stopak. "Version control systems: A technical guide to vcs internals." (Nov. 2019), [Online]. Available: https://initialcommit.com/blog/Technical-Guide-VCS-Internals.

[10] H. Surti. "Advanced data structures part 1: Directed acyclic graph (dag)." (Apr. 2016), [Online]. Available: https://medium.com/@hamzasurti/advanced-data-structures-part-1-directed-acyclic-graph-dag-c1d1145b5e5a.

[11] W. F. Tichy, “Rcs - a system for version control,” *Software: Practice and Experience*, vol. 15, no. 7, pp. 637–654, 1985. DOI: 10.1002/spe.4380150703.

[12] R. Vaghani. “Difference between singly linked list and doubly linked list.” (Jan. 2023), [Online]. Available: https://www.geeksforgeeks.org/difference-between-singly-linked-list-and-doubly-linked-list/.

[13] D. Zaltsman. “Difference between depth first search and breadth first search.” (Mar. 2020), [Online]. Available: https://dev.to/danimal92/difference-between-depth-first-search-and-breadth-first-search-6om.