Undo Trees and Git Simulation: An Application of Trees in Modeling Change History with Branching Operations

Philipp Hamara - 13524101
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: philipphamara420@gmail.com, 13524101@std.stei.itb.ac.id

***Abstract*— A vital component of modern software is the undo/redo system, which maintains consistency and prevents permanent mistakes. Standard undo/redo systems using a linear model often risk data loss when navigating between different branches of changes. Undo Trees address this issue by preserving the entire version history in a branching structure. A similar concept underlies version control systems like Git, which use tree-based branching to support collaboration and maintain code history. This paper presents a practical and visual implementation of both Undo Trees and a Git simulation, using concepts from Discrete Mathematics — specifically tree structures — to model branching histories in change-tracking systems.**

***Keywords*—Undo Trees, Git, Version Control System**

I. INTRODUCTION

Tracking change history in software is essential for maintaining consistency and enabling recovery of previous states. Undo and redo operations are key features that allow users to reverse or revisit changes. Most modern software implements undo/redo using a linear model based on a stack data structure. While efficient, this approach has limitations — particularly when a user makes edits to an earlier state, which can lead to overwriting alternate versions and losing data from diverging histories. These drawbacks have motivated the exploration of more flexible approaches to modeling change history.

One such approach is the concept of Undo Trees, introduced and popularized by the Vim text editor in 2010. Undo Trees use a nonlinear model where edits to past states create new branches instead of discarding alternative futures, thus preserving the entire change history. Despite its power, this branching undo system has not gained mainstream adoption due to its complexity and lack of intuitive behavior compared to the standard linear model.

A similar principle is also applicable in version control systems like Git, which manage project histories through branching and merging. In Git, branches allow multiple lines of development to coexist, enabling collaboration, experimentation, and rollback without data loss—features that are essential in modern software development workflows. Both systems fundamentally rely on tree-like structures to represent diverging and evolving states over time.

This paper presents a basic implementation of both an Undo Tree and a Git simulation, with interactive and visual features. The model is built upon the Discrete Mathematics concept of trees, showcasing a practical and instructive application of theoretical structures in software design. Through this work, the author aims to bridge theoretical understanding and functional implementation in the context of change tracking.

II. THEORETICAL BASIS

A. Graph

A graph is a structure that represents discrete objects (vertices) and the relationships between those objects (edges). Graph G is defined as G = (V, E), where V is a non-empty set of vertices, and E is a set of edges connecting pairs of vertices.

Based on the orientation of edges, graphs are classified into two types:

1) Undirected graph
   An undirected graph is a graph where the edges do not have a specific direction.
2) Directed graph
   A directed graph is a graph where each edge has a specific direction.

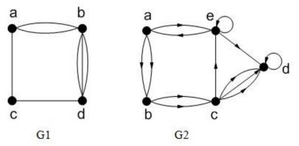


Fig. 1. Illustration of an undirected graph (G1) and a directed graph (G2). Source: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

There are a few graph terminologies that are important in this paper:

1) Path
A path in a graph is a sequence of vertices connected by edges, traversing from one vertex to another.

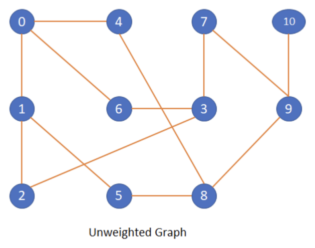


Fig. 2. 0, 6, 3, 7, 9, 10 path is the path from vertex 0 to 10
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

2) Cycle or Circuit
A cycle (or circuit) is a path that starts and ends at the same vertex.

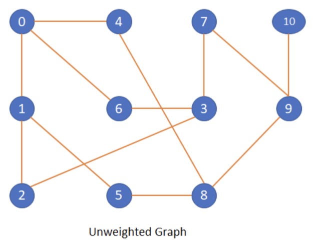


Fig. 3. 0, 4, 8, 5, 1, 0 path is a circuit
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

3) Connected
Two vertices v1 and v2 are said to be connected if there is a path from v1 to v2.

4) Connected graph
A connected graph is a type of graph in which every vertex can be reached from any other vertex through one or more paths.

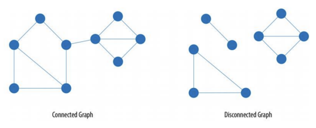


Fig. 4. Illustration of connected and unconnected graphs
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

B. Tree

A tree is an undirected graph that is connected and does not contain any circuits (cycles). A rooted tree is a tree in which one of its vertices is designated as the root, and its edges are assigned directions to form a directed graph.

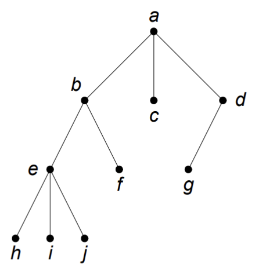


Fig. 5. Illustration of a rooted tree
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/24-Pohon-Bag2-2024.pdf

Rooted trees have several fundamental terminologies to describe their structure and relationships:

1) Child and Parent
A vertex v1 is a child of vertex v2 if there is a directed edge connecting v2 to v1. The originating vertex v2 is the parent of v1.
2) Path
A path is a sequence of vertices and edges from a starting vertex to a destination vertex.
3) Ancestor
A vertex v1 is an ancestor of vertex v2 if there is a path from v1 to v2. In other words, the ancestors of a vertex v2 are all vertices along the path from the root to v2.
4) Sibling
Vertices v1 and v2 are called siblings if they share the same parent.
5) Leaf
A leaf is a vertex that has no children, i.e., it has a degree of zero.

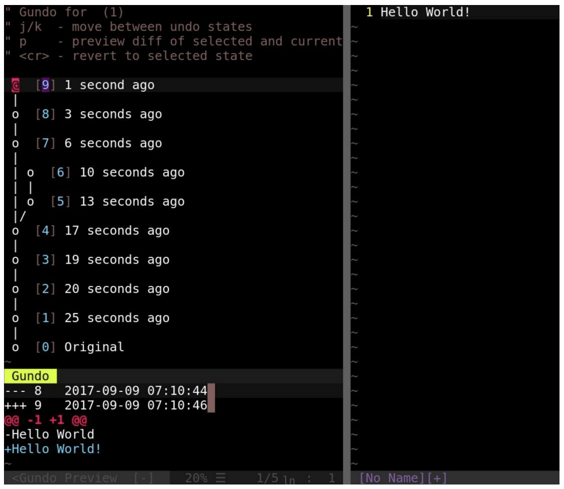
C. Undo Tree

The concept of the Undo Tree was introduced in Vim 7.0, which transitioned from using a simple stack-based undo model to a tree-based approach for tracking changes. In this model, the root node represents the original state of the file (i.e., with no edits). Each time a change is made, a new node is added as a child of the current state. This allows for the creation of a branching history that reflects multiple possible editing paths.

In a traditional linear undo system, if a user makes a new edit after undoing a change, the previous future states are discarded. This results in the loss of potentially important alternative versions. The Undo Tree overcomes this limitation by preserving those alternative futures as branches, allowing users to return to and continue from any point in the editing history.

The path from the root node to the current leaf node represents the sequence of changes leading to the current file state. Standard undo (‘u‘) and redo (‘<C-r>‘) commands traverse this path linearly, mimicking the behavior of conventional editors. However, if a user introduces a new change from an earlier node, Vim spawns a new branch rather than erasing future nodes, ensuring the entire history is retained.

Furthermore, Vim offers additional navigation commands such as ‘g-‘ and ‘g+‘, which move backward and forward through changes in the order they were made, regardless of their position in the tree. This provides users with a more flexible and powerful system for exploring and recovering past states.



```
" Gundo for  (1)
" j/k  - move between undo states
" p    - preview diff of selected and current
" <cr> - revert to selected state

@  [9] 1 second ago
|
o  [8] 3 seconds ago
|
o  [7] 6 seconds ago
|
| o  [6] 10 seconds ago
| |
| o  [5] 13 seconds ago
|/
o  [4] 17 seconds ago
|
o  [3] 19 seconds ago
|
o  [2] 20 seconds ago
|
o  [1] 25 seconds ago
|
o  [0] Original
~
Gundo
--- 8   2017-09-09 07:10:44
+++ 9   2017-09-09 07:10:46
@@ -1 +1 @@
-Hello World
+Hello World!
~
<Gundo Preview  [-]   20% ☰    1/5 ln  :  1
```

```
1 Hello World!
[No Name][+]
```

Fig. 6. Undo Tree visualization in Gundo.vim plugin
Source:
http://advancedweb.hu/never-lose-a-change-again-undo-branches-in-vim/

D. Git

Version Control Systems (VCS) are essential tools in software development, enabling individuals and teams to track, manage, and coordinate changes to code and other digital assets. Among the various VCS available, Git has emerged as the most widely used, known for its speed, flexibility, and support for distributed collaboration.

Git was developed by Linus Torvalds in 2005 to manage the Linux kernel’s source code. As a distributed version control system, Git allows every contributor to maintain a complete local history of the project, supporting independent development and robust merging capabilities.

At the core of Git’s functionality lies a tree-based structure that represents the evolution of a project through commits. Each commit in Git is a snapshot of the project at a specific point in time and is identified by a unique hash. Commits are linked together to form a directed acyclic graph (DAG), where each node points to one or more parent nodes, depending on whether it results from a linear update or a merge operation.

Branches in Git are lightweight pointers to specific commits, allowing parallel lines of development. This enables developers to experiment with new features without affecting the main codebase. When a feature is complete, it can be merged back into the main branch, integrating the separate line of development. The process of merging often introduces commits with multiple parents, capturing the history of both branches in the DAG.

Fig. 7. Git Logo
Source:
https://git-scm.com/downloads/logos

The core features of Git to be simulated in this paper are:

- Commits as nodes in trees
- Branches
- Creating commits
- Checkout
- Merging

III. IMPLEMENTATION

This section outlines the development of two interactive simulations: a text-based Undo Tree system and a Git-like version control simulator. Both applications are implemented in Python and make extensive use of object-oriented programming (OOP) principles. The `tkinter` module is utilized to build graphical user interfaces (GUIs), allowing visual interaction with the underlying tree structures.

A. Text-Based Undo Trees

To demonstrate how an undo system can preserve branching histories, a simplified text editor was created. Each word input is treated as a new state. Whenever a user presses `Space` or `Enter`, a new node is appended to the tree, preserving the full editing history. If a change is made after an undo, a new branch is formed instead of overwriting prior states.

1) TreeNode Class: The core data structure is a `TreeNode` class, representing a single text state. Each node contains:

- The current state of the text.
- A reference to its parent (previous state).
- A list of children (future branches).

This structure reflects the tree concept in Discrete Mathematics, where nodes represent elements and edges represent the transitions between versions. The add_state() method is used to create and link a new node as a child of the current node, preserving the full history and allowing multiple possible futures to coexist.

```python
class TreeNode:
    def __init__(self, text_state, parent=
        None):
        self.text_state = text_state
        self.parent = parent
        self.children = []

    def add_state(self, new_text):
        new_node = TreeNode(new_text, parent=
            self)
        self.children.append(new_node)
        return new_node
```

2) UndoTreeApp Class: The UndoTreeApp class implements an interactive text editor that uses the TreeNode structure to model its undo and redo functionality as a tree. Unlike a conventional linear undo stack, this application allows branching at any point in history. Each time the user makes a new change, a new TreeNode is created as a child of the current node. The application preserves previous branches, allowing users to revisit and continue from any prior state.

At a high level, the class is composed of the following parts:

- **Initialization (__init__):**
  Sets up the GUI layout, initializes the root node of the tree, and connects widgets like the text editor, undo/redo buttons, and the listbox that visually represents the tree.

```python
def __init__(self, master):
    self.master = master
    self.master.title("Undo Tree Text Editor")
    self.master.configure(bg='#0f1117')
    self.master.geometry("1200x600")

    self.root_node = TreeNode("")
    self.current_node = self.root_node

    self.last_text = self.root_node.text_state

    # Stack for redo
    self.redo_stack = []

    # Manual 50-50 split
    self.editor_frame = tk.Frame(self.master, bg='#1e1e2e')
    self.editor_frame.place(relx=0, rely=0, relwidth=0.5, relheight=1)

    self.tree_frame = tk.Frame(self.master, bg='#0f1117')
    self.tree_frame.place(relx=0.5, rely=0, relwidth=0.5, relheight=1)

    # Editor widgets
    self.text_label = tk.Label(self.editor_frame, text="Current Text:", bg='#1e1e2e', fg='#f1f1f1', font=("Helvetica", 32))
    self.text_label.pack(anchor='w', padx=10, pady=(10, 0))

    self.text_display = tk.Text(self.editor_frame, height=10, bg='#2b2d42', fg='#ffffff', insertbackground='white', font=("Consolas", 32), bd=0, relief=tk.FLAT)
    self.text_display.pack(fill=tk.BOTH, expand=True, padx=10, pady=(0, 10))
    self.text_display.bind('<KeyRelease-Return>', self.on_text_change)
    self.text_display.bind('<KeyRelease-space>', self.on_text_change)
    self.text_display.bind('<KeyRelease>', self.track_last_text)
    self.text_display.bind('<Control-z>', self.ctrl_z_handler)
    self.text_display.bind('<Control-y>', self.ctrl_y_handler)

    button_frame = tk.Frame(self.editor_frame, bg='#1e1e2e')
    button_frame.pack(anchor='w', padx=10, pady=(0, 10))

    self.undo_button = tk.Button(button_frame, text="Undo", command=self.undo, bg='#4c566a', fg='white', font=("Helvetica", 30, "bold"), activebackground='#5e81ac')
    self.undo_button.pack(side=tk.LEFT, padx=(0, 10))

    self.redo_button = tk.Button(button_frame, text="Redo", command=self.redo, bg='#4c566a', fg='white', font=("Helvetica", 30, "bold"), activebackground='#88c0d0')
    self.redo_button.pack(side=tk.LEFT)

    # Tree view widgets
    self.tree_label = tk.Label(self.tree_frame, text="Undo Tree", bg='#0f1117', fg='#81a1c1', font=("Helvetica", 32, "bold"))
    self.tree_label.pack(pady=(10, 5))
    self.tree_listbox = tk.Listbox(self.tree_frame, bg='#1a1d23', fg='#d8dee9', selectbackground='#5e81ac', font=("Courier", 20), bd=0, relief=tk.FLAT)
    self.tree_listbox.pack(fill=tk.BOTH, expand=True, padx=10, pady=(0, 10))
    self.tree_listbox.bind('<<ListboxSelect>>', self.on_tree_select)

    self.update_text_display()
    self.update_tree_view()
```

- **Text State Management:**
  The on_text_change() method listens for edits and appends a new TreeNode if the content has changed. This creates a new branch in the undo tree.

```python
def on_text_change(self, event=None):
    new_text = self.text_display.get("1.0", tk.END).rstrip('\n')
    if new_text != self.last_text:
        self.current_node = self.current_node.add_state(new_text)
        self.last_text = new_text
        self.redo_stack.clear()
        self.update_tree_view()
```

- **Undo and Redo Operations**:
  The undo() method moves to the parent node, while

redo() restores a child node if it exists on the redo stack. These operations update both the text editor and tree view.

```python
def undo(self):
    if self.current_node.parent:
        self.redo_stack.append(self.current_node)
        self.current_node = self.current_node.parent
        self.update_text_display()
        self.update_tree_view()
    else:
        messagebox.showinfo("Undo", "Already at the root node.")

def redo(self):
    if self.redo_stack:
        redo_node = self.redo_stack.pop()
        if redo_node.parent == self.current_node:
            self.current_node = redo_node
            self.update_text_display()
        self.update_tree_view()
        else:
            messagebox.showinfo("Redo", "Redo path no longer valid.")
    else:
        messagebox.showinfo("Redo", "Nothing to redo.")
```

- **Tree View Update:**
  The update_tree_view() and build_tree_list() methods recursively display the tree structure using text-based indentation. The currently active node is annotated with ← CURRENT.

```python
def update_tree_view(self):
    self.tree_listbox.delete(0, tk.END)
    self.node_list = []
    self.build_tree_list(self.root_node, "", True)

def build_tree_list(self, node, prefix="", is_last=True):
    connector = "+-- " if is_last else "|-- "
    display = f"{prefix}{connector}{repr(node.text_state)}"
    if node == self.current_node:
        display += "  <-- CURRENT"
    self.tree_listbox.insert(tk.END, display)
    self.node_list.append(node)

    child_count = len(node.children)
    for i, child in enumerate(node.children):
        next_prefix = prefix + ("    " if is_last else "|   ")
        self.build_tree_list(child, next_prefix, i == child_count - 1)
```

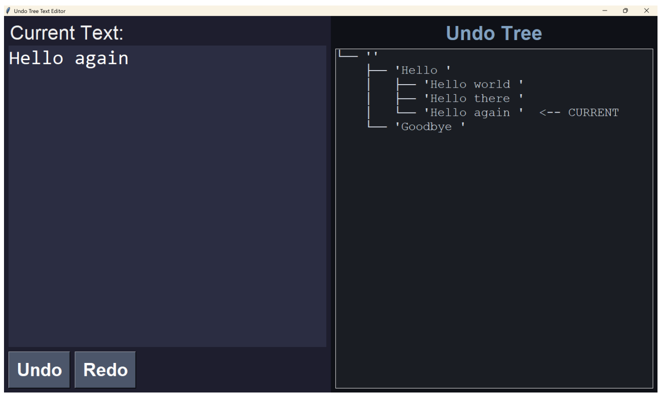
- **Tree Navigation:**
  The on_tree_select() function allows users to click on any past version in the tree and restore the corresponding text state.

```python
def on_tree_select(self, event):
    if not self.tree_listbox.curselection():
        return
    index = self.tree_listbox.curselection()[0]
    selected_node = self.node_list[index]
    self.current_node = selected_node
    self.update_text_display()
    self.update_tree_view()
```

3) Results: Using the program, users can interactively observe how each editing action contributes to the formation of a branching undo history.

The left pane of the application is the interactive text editor, where users can type and modify text. Every time the user presses the `Space` or `Enter` key, a snapshot of the current text is taken and stored as a new node in the undo tree. This mimics a lightweight commit mechanism, allowing users to observe how each action translates into a new state in the tree structure.

The right pane of the application displays this structure as an indented tree, where each line corresponds to a node (state), and the indentation reflects the node's depth in the tree. The active state is highlighted with a `<- CURRENT` annotation. This visual representation allows users to intuitively explore nonlinear editing histories and understand how different sequences of actions evolve into multiple branches, rather than overwriting each other.



```
Hello again
```

```
└── ''
    ├── 'Hello '
    │   ├── 'Hello world '
    │   ├── 'Hello there '
    │   └── 'Hello again '  <-- CURRENT
    └── 'Goodbye '
```

Fig. 8. Screenshot of the Undo Tree Text Editor program

Source:
Author's archive

As seen in Fig. 8., initially the tree is still empty except for the root node, the editing process begins with the text "`Hello `". From this base state, the user first writes "`Hello world `", creating the first branch. After undoing, a second variation "`Hello there `" is written, forming a sibling branch. Another two undos are performed, and this time the user writes "`Goodbye `", creating a second branch from the root node. Using the interactive and clickable tree view on the right side, a final variant "`Hello` again " is added from the "Hello branch. This shows how each edit from a previous state results in a new branch, preserving all alternative histories. The current state, marked by <- CURRENT, is at "Hello again ", illustrating the tree-like, non-linear structure of the undo history.

B. *Git Simulation*

To illustrate version control principles, a Git simulation was implemented. It allows users to perform basic Git operations such as commit creation, branching, checkout, and merging.

1) CommitNode Class: The CommitNode class models each Git commit. It contains:

- A unique commit ID (randomized string).
- A commit message.
- A reference to one or more parent commits.
- A list of children (next commits).
- Branch metadata for visualization.

This models Git's real-world commit DAG structure, where merge commits have multiple parents.

```python
# Generate a short random commit hash
def generate_commit_id():
    return ''.join(random.choices(string.
        hexdigits[:16], k=7))

class CommitNode:
    commit_counter = 0

    def __init__(self, message, parents=None,
     branch='main'):
        self.id = generate_commit_id()
        self.message = message
        self.parents = parents if parents
            else []
        self.children = []
        self.branch = branch
        self.order = CommitNode.
            commit_counter
        CommitNode.commit_counter += 1

    def add_child(self, message, branch=None)
    :
        new_commit = CommitNode(message,
            parents=[self], branch=branch or
            self.branch)
        self.children.append(new_commit)
        return new_commit
```

2) GitSimulationApp Class: The GitSimulationApp class provides an interactive interface to simulate core Git operations like making commits, switching branches, and merging. It mirrors Git's internal structure using a tree-like model where each commit is a node, and branches are visual paths of development.

The GitSimulationApp class simulates Git operations, including:

- **Commits:** Creating new commits as children of the current node.

```python
def commit(self):
    message = simpledialog.askstring("
        Commit Message", "Enter commit
        message:")
    if message:
        new_commit = self.head.add_child(
            message, branch=self.
            current_branch)
        self.branches[self.current_branch
            ].append(new_commit)
        self.head = new_commit
        self.update_view()
```

- **Branching:** Generating a new branch from the current commit.

```python
def create_branch(self):
    name = simpledialog.askstring("New
        Branch", "Enter new branch name:")
    if name and name not in self.branches
        :
        self.branches[name] = [self.head]
        self._log(f"Branch '{name}'
            created at commit {self.head.
            id}\n")
        self.update_view()
```

- **Checkout:** Moving the HEAD pointer to a different branch tip.

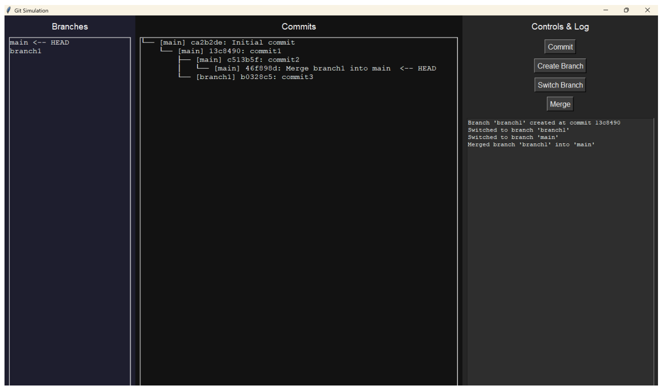
```python
def switch_branch(self):
    name = simpledialog.askstring("Switch
     Branch", "Enter branch name:")
    if name in self.branches:
        self.current_branch = name
        self.head = self.branches[name
            ][-1]
        self._log(f"Switched to branch '{
            name}'\n")
        self.update_view()
    else:
        messagebox.showerror("Error", "
            Branch does not exist.")
```

- **Merging:** Creating a new commit node with multiple parents.

```python
def merge_branch(self):
    name = simpledialog.askstring("Merge
        Branch", "Enter branch to merge:")
    if name in self.branches and name !=
        self.current_branch:
        other_head = self.branches[name
            ][-1]
        merge_commit = CommitNode(f"Merge
             {name} into {self.
            current_branch}", parents=[
            self.head, other_head], branch
            =self.current_branch)
        self.head.children.append(
            merge_commit)
        other_head.children.append(
            merge_commit)
        self.branches[self.current_branch
            ].append(merge_commit)
        self.head = merge_commit
        self.update_view()
        self._log(f"Merged branch '{name
            }' into '{self.current_branch
            }'\n")
    else:
        messagebox.showerror("Error", "
            Invalid branch.")
```

3) Results: The Git simulation enables users to interact with a simplified Git-like model where they can create branches, make commits, and perform merges. Each commit is represented as a node containing a truncated hash, a commit message, and the branch it belongs to. The relationships between commits—including parent-child links and merges—are maintained to reflect the underlying tree structure of Git.

- **Left Pane – Branches:** This panel lists all existing branches. The currently active branch is highlighted, and users can click on any branch to check it out. This visually reinforces Git's ability to manage multiple lines of development.
- **Middle Pane – Commit History:** This section displays the commit tree. Each commit is represented by its short hash, message, and the branch it belongs to. Indentation reflects parent-child relationships. Merge commits, which have multiple parents, are also displayed here, providing a clear visual of how branches diverge and converge.
- **Right Pane – Command Panel:** The rightmost panel contains interactive buttons and input fields to perform operations such as creating a commit, switching branches, and merging. Users can type commit messages or new branch names, then press buttons to execute Git operations. The interface immediately reflects these actions in the branch and commit views.



```
main <-- HEAD
branch1
```

```
└── [main] ca2b2de: Initial commit
    └── [main] 13c8490: commit1
        ├── [main] c513b5f: commit2
        │   └── [main] 46f898d: Merge branch1 into main  <-- HEAD
        └── [branch1] b0328c5: commit3
```

```
Branch 'branch1' created at commit 13c8490
Switched to branch 'branch1'
Switched to branch 'main'
Merged branch 'branch1' into 'main'
```

Fig. 9. Screenshot of Git Simulation program
Source:
Author's archive

As seen in Fig. 9., the Git simulation begins with the default `main` branch and an `Initial commit`. Two consecutive commits, `Commit 1` and `Commit 2`, are made on this main branch. Afterward, a new branch named `branch1` is created and checked out. A new commit, `Commit 3`, is added on `branch1`, branching off from `Commit 2`.

The user then checks out the `main` branch again and performs a merge with `branch1`. This creates a merge commit on the main branch with two parents: one from the end of `main` and one from the latest commit in `branch1`. The structure is clearly displayed in the middle pane (Commit History), showing the divergence and convergence of branches.

IV. CONCLUSION

Through the implementation of both an Undo Tree and a Git simulation, this paper demonstrates the practical application of tree structures—one of the core concepts in Discrete Mathematics—in solving real-world problems. The undo tree model provides a nonlinear and lossless editing history, while the Git simulation helps in visualizing how branching, merging, and commit tracking work under the hood in modern software development tools.

However, these implementations also reveal important limitations. One of the main concerns is space efficiency—since every edit or commit is stored as a new node, the memory usage grows rapidly with the number of operations, which may become impractical in larger applications.

From a usability and robustness perspective, error handling in the current implementations remains basic. Undo operations may become unintuitive in complex trees, and merging logic does not resolve content-level conflicts like real Git systems do.

Despite these shortcomings, this work successfully bridges theory and practice. It shows how a fundamental structure like a tree can be repurposed for real software problems, reinforcing the importance of Discrete Mathematics in computer science.

V. SUGGESTION

For future developments based on this work, the author suggests several directions of exploration to deepen both the theoretical and practical understanding of Undo Trees and Version Control Systems:

- **Memory and Performance Analysis:** Future research can include an in-depth comparison of memory usage and time complexity between linear undo stacks and tree-based undo models. This would provide valuable insight into their trade-offs in real-world applications, particularly in performance-critical software.
- **More Complete Git/VCS Simulation:** The current simulation covers only basic features such as commits, branches, and merges. Expanding this to include other Git concepts would provide a more comprehensive learning tool and better reflect actual usage.
- **Full-Scale Integration with Editors or IDEs:** Another valuable direction would be integrating the undo tree system or Git simulation into a full-scale text editor or IDE plugin. This would allow users to experience the practical implications of tree-based change tracking in everyday software development environments.

By pursuing these directions, students and developers can not only gain a stronger grasp of discrete mathematics applications but also build more powerful tools for real-world collaborative software development.

APPENDIX

The GitHub repository for this paper can be accessed at https://github.com/philipphqiwu/Makalah-IF1220-UndoTrees-GitSimulation.

ACKNOWLEDGMENT

The author expresses his heartfelt gratitude to Dr. Ir. Rinaldi, M.T., his lecturer for Discrete Mathematics at Bandung Institute of Technology, for his comprehensive teaching which laid down the groundwork for understanding the concepts applied in this paper.

The author would also like to thank his friends and family for all the support they have shown throughout his journey as a student at ITB.

Lastly, the author would like to show his appreciation to the youtuber Tom Scott for getting him into computer science and as the source of inspiration for the topic of this paper. Mainly inspired from a small part of his video "The Worst Typo I Ever Made": https://www.youtube.com/watch?v=X6NJkWbM1xk.

REFERENCES

[1] Munir, Rinaldi. 2024. “Graf (Bagian 1)”. *Department of Informatics, Institut Teknologi Bandung, 2024-2025*, [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf. [Accessed: Jun. 18, 2025].

[2] Munir, Rinaldi. 2024. “Pohon (Bagian 1)”. *Department of Informatics, Institut Teknologi Bandung, 2024-2025*, [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/23-Pohon-Bag1-2024.pdf. [Accessed: Jun. 18, 2025].

[3] Munir, Rinaldi. 2024. “Pohon (Bagian 2)”. *Department of Informatics, Institut Teknologi Bandung, 2024-2025*, [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/24-Pohon-Bag2-2024.pdf. [Accessed: Jun. 18, 2025].

[4] Csákvári, Dávid. "Never lose a change again – Undo branches in Vim," *Advanced Web*, [Online]. Available: https://advancedweb.hu/never-lose-a-change-again-undo-branches-in-vim/. [Accessed: Jun. 18, 2025].

[5] Worsley, Summer. What is Git? - The Complete Guide to Git," *Datacamp*, [Online]. Available: https://www.datacamp.com/blog/all-about-git. [Accessed: Jun. 19, 2025].

[6] Sharma, Rajat. “Getting Started with Trees in Python – A Beginner’s Guide,” *Medium*, [Online]. Available: https://medium.com/pythoneers/getting-started-with-trees-in-python-a-beginners-guide-4e68818e7c05. [Accessed: Jun. 19, 2025].

STATEMENT

Hereby, I declare that this paper I have written is my own work, not an adaptation or translation of someone else’s paper, and not a product of plagiarism.

Bandung, 20 July 2025

[illegible]

Philipp Hamara
13524101