**MiniGit Project Report**

**Overview**

**MiniGit** is a lightweight, Git-inspired version control system implemented in C++ to simulate core version control functionalities such as file tracking, committing, branching, merging, and viewing commit history through a command-line interface (CLI). Built from scratch without external version control libraries, MiniGit provides a local-only experience that helps users understand the underlying principles of version control systems like Git. The implementation leverages data structures and algorithms (DSA) such as hashing, directed acyclic graphs (DAGs), and hash tables, offering a practical exploration of software engineering concepts like modularity, file I/O, and change management.

**Data Structures**

The system employs the following data structures to manage version control operations, aligning with the project's DSA requirements:

* **Blob**: Stores file content with SHA-1 hashing for unique identification. Each blob is saved in the .minigit/objects/ directory, using its hash as the filename. This facilitates efficient content tracking and deduplication. **DSA: Hashing, I/O**
* **Commit Node**: Represents a commit with metadata (timestamp, message, author, parent(s) hashes) and references to file blobs. Commits form a DAG, enabling history traversal and merge operations. Implemented as a Commit class. **DSA: Linked List/DAG**
* **Branch Reference**: Maps branch names to commit hashes using a std::map<std::string, std::shared\_ptr<Branch>>, stored in .minigit/refs/. This allows quick branch lookups and updates. **DSA: HashMap**
* **Staging Area**: Tracks files to be committed using a std::map<std::string, std::shared\_ptr<Blob>>, providing fast access to staged files. **DSA: Hash Table**
* **Log History**: Enables traversal of commit history from HEAD backward, following parent pointers to reconstruct the commit DAG. **DSA: Linked List**

**Key Features**

MiniGit implements all required features, providing a robust simulation of Git's core functionality:

1. **Initialization (init)**: Creates a hidden .minigit/ directory with subdirectories for objects (.minigit/objects/) and references (.minigit/refs/), initializes the main branch, and sets up the HEAD pointer. It checks for existing repositories to prevent re-initialization, ensuring safe setup.
2. **Add Files (add)**: Stages files for the next commit by creating Blob objects with SHA-1 hashes of their content. Validates file existence to prevent errors and stores staged files in memory for efficient commit preparation.
3. **Commit (commit -m )**: Captures a snapshot of staged files, creating a Commit object with metadata (timestamp, message, parent hash) and blob references. Updates HEAD and the current branch, then clears the staging area. Commits are stored as files in .minigit/objects/ using their SHA-1 hash.
4. **View Log (log)**: Traverses the commit history from HEAD, displaying commit hashes, author, timestamp, and message in a Git-like format. Includes a safeguard to prevent infinite loops, ensuring reliable history navigation.
5. **Branching (branch)**: Creates a new branch pointing to the current commit, storing it in .minigit/refs/. Supports quick branch creation and management, mimicking Git's branching model.
6. **Checkout (checkout)**: Switches to a specified branch or commit hash, updating HEAD and restoring the working directory to match the target commit's files. Supports both branch and commit-based checkouts for flexibility.
7. **Merge (merge)**: Performs a three-way merge by finding the lowest common ancestor (LCA) of the current and target branch commits using a set-based approach. Compares file changes from the LCA to both branches, adopting non-conflicting changes and marking conflicts with standard Git-style conflict markers (<<<<<<< HEAD, =======, >>>>>>> <branch>). Creates a merge commit with two parents to record the merge.
8. **Diff Viewer (diff)**: Implements a basic line-by-line diff viewer to compare two commits, showing added, removed, or modified files in a Git-like format. Displays new/deleted files and changed lines, though it lacks advanced diff algorithms for optimal output.

**Design Decisions**

* **Object-Oriented Design**: MiniGit uses separate classes (Blob, Commit, Branch) for encapsulation and reusability, with smart pointers (std::shared\_ptr) to manage memory safely. The MiniGit class orchestrates all operations, ensuring a clear separation of concerns.
* **Persistence**: Stores blobs and commits as files in .minigit/objects/ using SHA-1 hashes for naming, and branches in .minigit/refs/. This mirrors Git's storage model, providing durability and efficient content addressing.
* **Hashing**: Relies on a utility function (utils::sha1\_hash, assumed to use OpenSSL or similar) for SHA-1 hashing, ensuring consistent and reliable content identification.
* **Merge Strategy**: Implements a three-way merge with LCA computation, using a set-based algorithm to find the LCA efficiently. Conflicts are marked in files for manual resolution, balancing functionality with simplicity.
* **Error Handling**: Includes validation for repository initialization, file existence, branch names, and commit hashes. Provides clear error, warning, and success messages via utility functions (utils::print\_error, utils::print\_warning, utils::print\_success) to enhance user feedback.
* **Modularity**: Separates file I/O, hashing, and formatting into a utils namespace, improving code maintainability and reusability.

**CLI Usability**

MiniGit's CLI mirrors Git's command structure, supporting init, add <filename>, commit -m <message>, log, branch <branch-name>, checkout <branch|hash>, merge <branch-name>, and diff <commit1> <commit2>. It provides informative feedback for successful operations and errors (e.g., "File does not exist", "Branch already exists"). However, it lacks a help command to list available commands, which could improve user experience, especially for new users.

**Limitations**

* **Diff Viewer**: The diff implementation shows basic line-by-line changes but does not use advanced algorithms (e.g., Myers' diff) for optimal diff output, limiting its precision for complex changes.
* **Conflict Resolution**: Merge conflicts require manual resolution by editing files with conflict markers, as automated resolution is not implemented.
* **Storage Efficiency**: Blobs are stored uncompressed, which may lead to larger disk usage for large files or repositories.
* **Edge Cases**: While error handling is robust, additional validation for special characters in file paths or branch names could further improve reliability.
* **Missing Advanced Features**: Features like rebasing, stashing, or garbage collection are not implemented, though these are beyond the project scope.

**Future Improvements**

* Enhance the diff viewer with a sophisticated algorithm (e.g., Myers' diff) to provide precise line-by-line differences, improving usability for developers.
* Implement automated conflict resolution options to simplify merge conflict handling.
* Add blob compression (e.g., using zlib) to reduce storage requirements.
* Introduce a help command to display available commands and usage, improving CLI accessibility.
* Support advanced Git features like rebase or stash to extend functionality.
* Implement garbage collection to remove unused objects, optimizing repository size.
* Add more robust validation for edge cases, such as invalid file paths or malformed commit objects.

**Conclusion**

MiniGit successfully implements a lightweight version control system that meets all core and advanced project requirements, including initialization, file staging, committing, branching, merging with LCA-based three-way merge, log viewing, and a basic diff viewer. Its object-oriented design, robust error handling, and Git-like CLI make it a practical tool for understanding version control internals. The use of DSA concepts like hashing, DAGs, and hash tables aligns with the project's educational goals, providing a clear demonstration of how Git works under the hood. While minor enhancements (e.g., advanced diff, help command) could further improve usability, MiniGit is a complete and functional system suitable for local version control tasks. A video demo and GitHub repository link will be provided to showcase the system's functionality and development history.