BASH GRADER

CS 108 PROJECT

by

SANCHITA CHAURASIA

Roll No. 23B1048

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY

Mumbai - 400076, India

April, 2024

Abstract

Bash Grader is a versatile tool designed to streamline the management of student records while incorporating custom version control functionalities tailored for educational settings. This Bash-scripted solution offers a user-friendly interface, enabling educators and administrators to efficiently manage student data, analyze performance metrics, and collaborate on record updates.

The system's core functionality includes record management, statistical analysis, version control, performance optimization, and error handling. Leveraging Bash scripting, the system provides robust utilities for handling student records, generating graphical plots, and implementing Git-like version control operations tailored to educational workflows.

Customizations such as color formatting, and autocorrect enhance user experience, while performance optimization techniques ensure swift operation even with large datasets. Error handling mechanisms provide resilience against unexpected scenarios, ensuring system reliability and user satisfaction.

Directory Structure

```
submission.sh
|- bash_colors.sh
|- basic.sh
|- graph.sh
|   |- box_graph.py  box_plot.png
|   |- density_graph.py  density_plot.png
|   |- histogram_graph.py  histogram_plot.png
|   |- marks_graph.py  distribution_plot.png
|   |- scatter_graph.py  scatter_plot.png
|- git.sh
|- latex.sh
|- stats.sh
|- utils.sh
```

Contents

Chapter 1

Introduction

Welcome to the detailed report outlining the development and implementation of a ***student record management system*** with custom version control capabilities, powered by Bash scripting. This project addresses the need for efficient management of student records and integrates version control functionalities tailored to the **educational domain's requirements**.

1.1 Problem Statement Overview

The project consists of several interconnected components, each serving a specific role in achieving the overall objectives:

- **Bash Scripting:** The core of the project, Bash scripting, powers the implementation of essential functions, including record management, statistical analysis, version control, performance optimization, error handling, user interaction, and integration with external tools.
- **Custom Version Control System:** Implemented in Bash scripting, the custom version control system enables efficient tracking of changes, collaboration,

version history management, and error handling tailored to the educational context.

- **Statistical Analysis and Visualization:** Bash scripts facilitate statistical calculations and graphical visualization of student data, providing insights into student performance trends and aiding decision-making.
- **Main Control Script (submission.sh):** Serving as the central orchestrator, submission.sh coordinates the execution of various functions, manages user interactions, ensures seamless integration among different components, and handles error scenarios, contributing to the system's overall efficiency, coherence, and reliability.

1.2 Project Objectives

The project aims to develop a comprehensive **student record management system** with integrated version control capabilities using Bash scripting. This system is designed to fulfill the following objectives:

1. **Record Management:** Efficiently manage student records, including marks for assessments, assignments, and examinations, using Bash scripting for data processing and manipulation.
2. **Statistical Analysis and Visualization:** Generate statistical metrics and visualize data trends to aid educators in assessing learning outcomes and identifying areas for improvement.

3. **Custom Version Control:** Implement a custom version control system inspired by Git to track changes, manage version history, and facilitate collaboration on record updates.

4. **Usability and Reliability:** Ensure the system is user-friendly, scalable, and reliable, with intuitive interfaces, robust error handling, and clear documentation to support educators and administrators.

5. **Performance Optimization:** Optimize system performance to ensure swift and responsive operation, even with large datasets, through efficient algorithms and Bash scripting techniques.

6. **Error Handling:** Implement robust error handling mechanisms to detect and gracefully handle exceptional scenarios, enhancing the system's reliability and user experience.

7. **User Interaction:** Provide intuitive user interfaces and clear documentation to facilitate user interaction and ensure ease of use for educators and administrators.

8. **Scalability:** Design the system to handle growing datasets and accommodate future enhancements, maintaining performance and reliability as the user base expands.

9. **Integration with External Tools:** Integrate with external tools and scripts, such as Python scripts for generating graphical plots, to enhance the system's functionality and versatility.

1.3 Overview of Project Components

The project consists of the following components:

- **submission.sh:** The main control script that orchestrates various operations related to managing student records and Git operations.
- **basic.sh:** Contains basic utility functions such as combine all csv files into main,csv, upload files and total.
- **utils.sh:** Contains utility functions for displaying student details and updating student marks.
- **git.sh:** Contains functions for Git operations like initializing a repository, committing changes, staging files, viewing commit history, branching, merging and checking out commits.
- **stats.sh:** Contains functions for calculating statistics like mean, median, mode, range, variance, standard deviation, and percentiles for student marks.
- **graph.sh:** Contains functions for generating different types of graphs (marks, density, scatter, histogram, box) based on student marks.
- **main.csv:** The main CSV file containing student records.
- **marks_graph.py:** Python script for generating a line plot of marks distribution.
- **density_graph.py:** Python script for generating a density plot of marks distribution.
- **scatter_graph.py:** Python script for generating a scatter plot of marks distribution.

- **histogram_graph.py:** Python script for generating a histogram of marks distribution.
- **box_graph.py:** Python script for generating a box plot of marks distribution.

Chapter 2

Chapter 2: Detailed Functionality Overview

For better understanding, it is recommended to go through the files mentioned along with this report. The comments are self explanatory. The following overview has been categorized into script file followed by functions it contains and so on.

2.1 basic.sh:

combine_csv_files()

Usage: bash submission.sh combine

- This is a function that combines multiple CSV files into one file named `main.csv`.
- It first declares two associative arrays `students` and `present`, and an array `columns` to store column names.

- It checks if `main.csv` exists and if it contains a 'total' column. If it does, it sets `total_exists` to true, otherwise false.
- It then loops over each CSV file in the current directory. For each file, it extracts the column name from the file name and adds it to the `columns` array.
- A `for loop` then starts which interates through all .csv files except main.csv.
  - It then initializes the `present` array to false for each student.
  - It reads the CSV file line by line. For each line, it checks if the student is already in the `students` array. If not, it adds them and fills in "a" for any missing marks. It then adds the marks from the current file to the student's data and marks the student as present in this file.
  - After reading the file, it adds "a" for any students not present in this file.
- After processing all files, it writes the column names and student data to `main.csv`.
- If the 'total' column existed in the original `main.csv` file, it adds it back.

upload_csv_file()

Usage: bash submission.sh upload "file_path"

- This function uploads a CSV file to the current directory.
- It first checks if the file path is not empty and file exists. It stores the file path in a variable `file_path` and extracts the `file_name` from the file path.

- It then checks if a file with the same name already exists in the current directory. If it does, it asks the user if they want to overwrite the existing file. If the user answers yes(Y or y), it overwrites the file. If the file does not exist, it simply copies the file to the current directory.

add_total_column()

Usage: bash submission.sh total

- This function adds a 'total' column to the `main.csv` file.
- It first checks if `main.csv` contains a 'total' column. If it does, it updates the totals. If it doesn't, it adds a 'total' column and calculates the totals.
- The totals are calculated by summing the marks for each student using `awk`. If a mark is "a" or empty, it is counted as 0. The updated file is written to a temporary file, which is then renamed to `main.csv`.

2.2 utils.sh

update_student_marks

Usage: bash submission.sh update

- The function starts by checking if `main.csv` file exists and if it contains a "total" column. If it does, `total_exists` is set to true, otherwise false.

- An infinite loop is started to continuously ask the user for a student's roll number.
- The function searches for the entered roll number at the start of a line in the `main.csv` file.
- If no matches are found, `agrep` is used to find similar roll numbers in the `main.csv` file. If there are any suggestions, they are displayed to the user and the loop continues to the next iteration.
- If more than one match is found, the user is informed that multiple matches were found and the matches are displayed. The loop then continues to the next iteration.
- If exactly one match is found, the user is asked to confirm that they want to update the marks for the student. If the user enters 'n' or 'N', the loop continues to the next iteration.
- The user is asked to choose an update mode: Interactive mode or Name mode.
- If the user chooses Interactive mode, the function reads the first line of the `main.csv` file, which contains the column names, into an array. For each column that is not the roll number, name, or total column, the user is asked to enter new marks. The old marks are replaced with the new marks in the `main.csv` file and the corresponding CSV file (if it exists).
- If the user chooses Name mode, they are asked to enter the name of the column to update. The old marks are replaced with the new marks in the `main.csv` file and the corresponding CSV file (if it exists).
- If the total column exists, the `add_total_column` function is called to update the total column.

- The loop then continues to the next iteration, asking the user for another student's roll number.

2.3 git_basic.sh

init_remote_repo:

- This function initializes a new repository at the specified path.
- If a repository already exists (indicated by the presence of a .git_repo_path file), it asks the user if they want to initialize a new repository.
- If the user confirms, it asks if they want to delete the old repository. If the user confirms, it deletes the old repository; otherwise, it renames the old repository.
- If the new repository directory already exists, it prints an error message and returns 1.
- Finally, it creates the new repository directory and saves its path for future reference.

commit_changes:

- This function commits changes to the repository.
- It first checks if a .git_repo_path file exists. If not, it prints an error message and returns.
- It then reads the remote directory path from the .git_repo_path file and generates a random 16-digit hash value for the commit.

- It creates a new directory for the commit and copies all files (excluding the repository folder) to the commit directory.
- It then adds the commit hash and message to the log file.
- If a .last_commit file exists, it prints the files modified since the last commit.
- Finally, it saves the hash of the current commit as the last commit.

checkout_commit:

- This function checks out to a specific commit.
- It first checks if a .git_repo_path file exists. If not, it prints an error message and returns.
- It then reads the remote directory path from the .git_repo_path file and finds the commit folder that starts with the provided hash.
- If a matching commit folder is found, it copies its contents to the checkouted directory.
- If no matching commit folder is found, it prints an error message.

commit_graph:

- This function displays the commit graph.
- It first checks if a .git_repo_path file exists. If not, it prints an error message and returns.

- It then reads the remote directory path from the .git_repo_path file and checks if a .git_log file exists.
- If not, it prints an error message and returns.
- Finally, it reads the log file and prints each commit hash and message.

Chapter 3

Customization

bash_colors.sh

- This script defines color codes for printing text in different colors and styles.
- It has been sourced in other scripts to print color-coded messages.

Autocorrect Functions in utils.sh [display update]

- **Description:** These functions provide suggestions for possible corrections for mistyped names or roll numbers while using the display or update functions.
- **Code Snippets:**

```
matches=$(grep -i "^$roll_number" main.csv)

# If no matches were found
if [ -z "$matches" ]; then
```

```
# Use agrep to find similar roll numbers in the main.csv file
# The -2 option allows up to 2 errors for a match (insertions,
deletions or substitutions)
suggestions=$(agrep -2 "$roll_number" main.csv | cut -d',' -f1,2)
```

git_custom.sh

- **git_init** This function initializes a new repository at the specified path. If a repository already exists, it asks the user if they want to delete or rename the old repository.
- **git_checkout** This function checks out to a specific branch. It finds the branch folder and copies its contents to the checkouted directory.
- **git_commit** This function commits changes to the repository. It generates a random 16-digit hash value for the commit, creates a new directory for the commit, copies all files to the commit directory, and logs the commit hash and message. If a .last_commit file exists, it also prints the files modified since the last commit.
- **git_add** This function adds files to the index. It copies the specified files to the index directory.
- **git_branch** This function manages branches. It can create, delete, and list branches. When creating a new branch, it copies the logs of the current branch to a new file in the branches directory. When deleting a branch, it checks if the branch has unmerged changes. When listing branches, it lists all files in the branches directory.

- **git_log** This function displays the commit history of the current branch. It reads the repository path from the .git_repo_path file, checks that no arguments were passed, finds the current branch by reading the HEAD file, and then reads and prints the logs of the current branch.

- **git_merge** This function merges changes from another branch or commit into the current branch. It reads the repository path from the .git_repo_path file, checks if there are any commits, checks the arguments to determine the commit message, counts the commit number, finds the current branch by reading the HEAD file, checks if the branch or commit exists, gets the last commit number of the current branch, checks if the branch or commit exists in the log file of the current branch, checks for errors, copies the files from the branch or commit to the working directory and index, checks if a fast-forward merge is possible, and then changes the log file.

- **git_show** This function shows the contents of a file from a specific commit or the index. It reads the repository path from the .git_repo_path file, checks the arguments to determine the commit and filename, counts the commit number, and then shows the file from the specified commit or the index.

- **git_status** This function shows the status of each file in the working directory, index, and repository. It reads the repository path from the .git_repo_path file, creates a temporary file to store filenames, finds the current branch by reading the HEAD file, finds the last committed repository, checks each file in the working directory, index, and repository, displays the status of each file, and then removes the temporary file.

- **git_graph** This function generates a graph of the commit history. It reads the remote directory path from the .git_repo file, checks if the log file exists in the remote directory, and then reads the log file and generates the graph. Each commit is displayed with its hash and message.

graph.sh

Each function generates a different type of graph using a Python script and a CSV data file. The generated graph is saved as a PNG file.

- **generate_marks_graph:** This function generates a marks graph using a Python script. It defines the Python script, data file, and plot file. If no arguments are provided, it plots it for all available column names. If arguments are provided, it calls the Python script with the provided arguments as column names.
- **generate_density_graph:** This function generates a density graph using a Python script. It defines the Python script, data file, and plot file. If no arguments are provided, it plots it for all available column names. If arguments are provided, it calls the Python script with the provided arguments as column names.
- **generate_scatter_graph:** This function generates a scatter graph using a Python script. It defines the Python script, data file, and plot file. If no arguments are provided, it plots it for all available column names. If arguments are provided, it calls the Python script with the provided arguments as column names.
- **generate_histogram_graph:** This function generates a histogram graph using a Python script. It defines the Python script, data file, and plot file.

If no arguments are provided, it plots it for all available column names. If arguments are provided, it calls the Python script with the provided arguments as column names.

- **generate_box_graph:** This function generates a box graph using a Python script. It defines the Python script, data file, and plot file. If no arguments are provided, it plots it for all available column names. If arguments are provided, it calls the Python script with the provided arguments as column names.

stats.sh

- **calculate_mean:** This function calculates the mean of a specified column in a CSV file. If no column is specified, it calculates the mean for all columns except "Roll_Number" and "Name". It uses awk to sum up the values and divide by the count.
- **calculate_median:** This function calculates the median of a specified column in a CSV file. If no column is specified, it calculates the median for all columns except "Roll_Number" and "Name". It uses awk to sort the values and find the middle one.
- **calculate_mode:** This function calculates the mode of a specified column in a CSV file. It uses awk to count the occurrences of each value and find the one with the maximum count.
- **calculate_range:** This function calculates the range of a specified column in a CSV file. It uses awk to find the maximum and minimum values and calculate the difference.

- **calculate_variance:** This function calculates the variance of a specified column in a CSV file. It uses awk to calculate the mean, subtract each value from the mean, square the result, and average these squared differences.
- **calculate_stddev:** This function calculates the standard deviation of a specified column in a CSV file. If no column is specified, it calculates the standard deviation for all columns except "Roll_Number" and "Name". It uses awk to calculate the variance and then takes the square root.
- **calculate_percentiles:** This function calculates specified percentiles of all columns in a CSV file, except "Roll_Number", "Name", and "total". It calls the calculate_percentile function for each column and percentile.
- **calculate_percentile:** This function calculates a specific percentile of a specified column in a CSV file. It uses awk to sort the values and find the one at the index that corresponds to the percentile.

Performance Optimization

- **Description:** This section outlines techniques implemented for optimizing the performance of the student record management system.
- **Techniques Implemented:**
  1. Use of associative arrays for efficient data storage and retrieval.
  2. Minimization of file I/O operations to reduce processing time.
  3. Utilization of optimized algorithms for data manipulation and analysis.
- **Impact on Project:** These optimization techniques significantly enhance the system's performance, resulting in faster execution and improved scalability.

For example, the combine function takes **only** 0.002 **seconds to run for combining 3 files with over 150 records each.**

Chapter 4

Conclusion

The development and implementation of the student record management system have culminated in a comprehensive solution tailored to the educational domain's requirements. Through the integration of Bash scripting, the system offers a robust set of functionalities for efficient record management and custom version control.

- **Project Achievements:** The system successfully addresses the project objectives, providing educators with a powerful tool for managing student records, analyzing performance metrics, and facilitating collaboration through version control capabilities.

- **Lessons Learned:** Throughout the development process, valuable insights were gained into the challenges and complexities of building a versatile and reliable system for educational purposes.

  For example, when you run a script with `bash -x`, it prints each command before executing it. This can be helpful for debugging because it allows you to see exactly what commands are being executed and can help identify any issues with the script's logic or execution flow. Lessons learned include the importance of robust error handling, the significance of performance optimization, and the benefits of clear documentation and user-friendly interfaces.

- **Future Enhancements:** While the current system meets the immediate needs of educators, there are opportunities for future enhancements to further improve functionality and usability. Potential areas for enhancement include the integration of additional statistical analysis tools, support for more advanced visualization techniques, and the implementation of user authentication and access control mechanisms for enhanced security.

In conclusion, the student record management system represents a significant achievement in the intersection of educational technology and software development. By leveraging the power of Bash scripting and custom version control, the system empowers educators to efficiently manage student records, analyze performance data, and collaborate on educational initiatives. As the system evolves and grows, it holds the potential to make a positive impact on teaching and learning in educational institutions worldwide.