PARISSUBWAY

Amine Khelif KHELIF[1]

I. INTRODUCTION

The project for the Advanced C++ course is designed to challenge and evaluate the students' proficiency with the language. This project involves the development of a straightforward navigation system that enables users to find routes between two stations within the Parisian subway network, managed by RATP. It utilizes Dijkstra's algorithm to determine the shortest path between any two stations.

The project serves as a practical application of essential C++ concepts, including classes, inheritance, polymorphism, templates, and exception handling. Furthermore, it incorporates the use of the C++ Standard Library, emphasizing the importance of writing clean, efficient code. This project not only tests the students' technical skills but also their ability to apply these skills in solving real-world problems.

II. DEVELOPMENT

The project is structured into several progressive stages, each designed to build upon the previous. Details of these stages are available in the following GitHub repository: PARISSUBWAY.

My development process is organized into branches within the repository, each corresponding to a specific stage(see Fig 1) :

Fig. 1. Project Git Repo

A. Stage 1: Implementation of Parsing Classes

- **StationParser Class**: Parses the stations file and stores the data in a hashmap of Station objects.
- **ConnectionParser Class**: Parses the connections file and stores the data in a hashmap of Connection objects (See Fig 2).

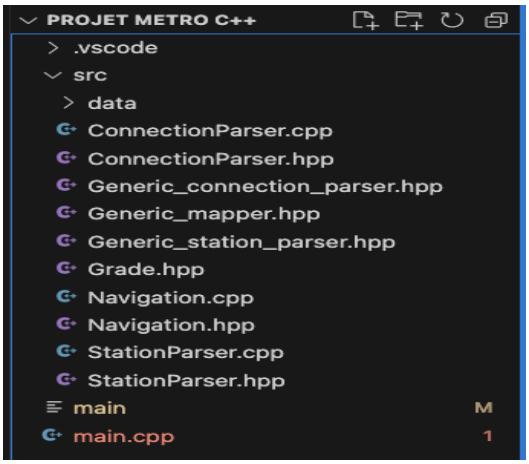

Fig. 2. Classes at the first stage

- **Navigation Class**: Implements Dijkstra's algorithm to determine the shortest path between two stations. It also includes methods to retrieve and display the path and the distance between stations, works only with Ids at this stage.(See Fig 3)

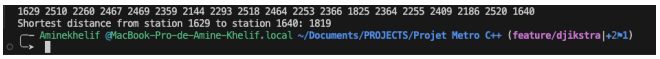

Fig. 3. Djikstra Algorithme inference with IDs

B. Stage 2: Integration of Station Search Features

Enhancement of the StationParser class to include methods such as `get_station_id_by_name_and_line`, `get_station_name_by_id`, and `get_station_by_id`. These methods allow users to search for a station by its name and line number, facilitating user interaction by enabling the return of the station ID. This ID is then utilized in the Navigation class to compute and display the shortest path and the corresponding distance using station names and line numbers, rather than just station IDs (See Fig 4).

C. Stage 3: Implementation of Interactive Mode

- **User Interaction Loop**: Engages users through a continuous interaction loop, allowing for seamless naviga-

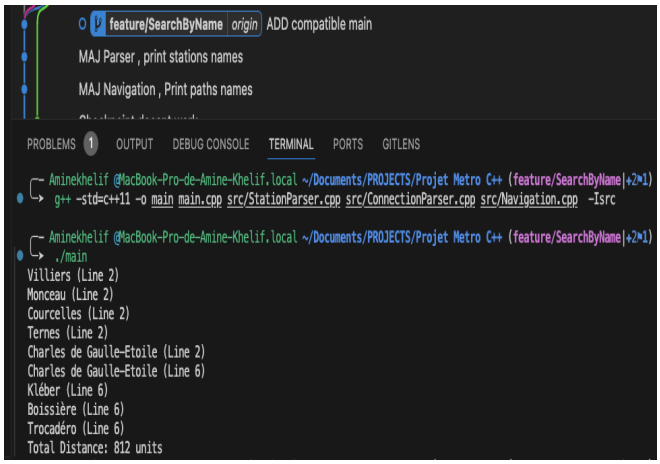

Fig. 4. Search by Name Feature

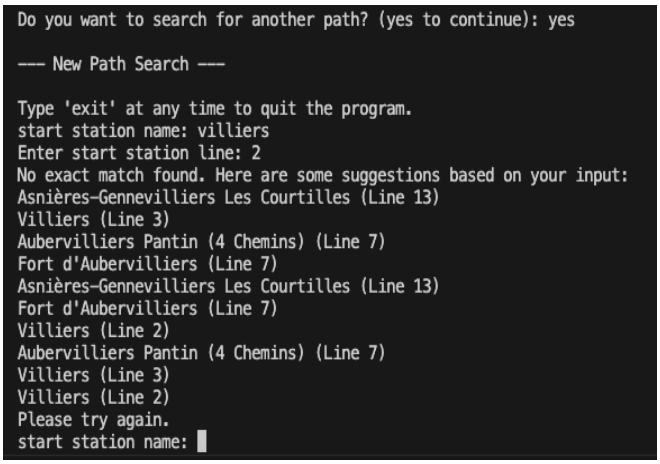

Fig. 5. Errors in Station Names

tion within the application. Users can easily exit the application by typing "exit."

- **Simplified Navigation**: Enables users to initiate searches by simply entering the station name and line number for both departure and arrival stations, thereby discovering the shortest path between them.
- **Decision-Making**: After completing a search, users are given the option to either conduct another search or exit the program, ensuring a user-friendly experience that accommodates both continuous interaction and simple termination.
- **User-Friendly Displays**: Outputs are designed to be clear and well-structured to aid comprehension. Each station is displayed with its name and line number, making it straightforward for users to understand the recommended path and navigate through the stations to reach their destination.
- **Organized Interface**: Each section of the output is clearly titled and separated, enhancing the readability and overall user experience, making it easy to follow and interact with the application.
- **Error Handling**: Implemented error handling mechanisms to manage invalid user inputs, a function that provides stations propositions when the user enters an incorrect station name. These features ensure that users receive informative feedback and can easily correct their inputs (See Fig 5).

D. Stage 4: Integration of Navigation and Data Parsing

- **MetroNetworkParser Class**: I have merged the functionalities of the StationParser and ConnectionParser into a unified class, the MetroNetworkParser, which inherits from `Generic_mapper`. This class is now responsible for parsing both station and connection data, efficiently storing this information, and providing seamless access to it. This integration has significantly streamlined the codebase, enhancing maintainability and reducing complexity.
- **Navigation Class Update**: I updated the Navigation class to utilize the MetroNetworkParser. Upon instantiation, the Navigation class now receives a pointer to the MetroNetworkParser, granting it direct access to the necessary network data for its operations. This modification emphasizes the Navigation class's dependence on the MetroNetworkParser.

E. Ownership and Composition Relationship

I designed the MetroNetworkParser to have a clear ownership over the Navigation class. It manages the lifecycle of the Navigation class, including its creation and deletion. This ownership ensures that all data handling responsibilities are centralized within the MetroNetworkParser, simplifying the system architecture and enhancing application robustness.

F. Operational Dependency

The Navigation class is operationally dependent on the MetroNetworkParser. This dependency is critical for enabling the Navigation class to perform its primary function—calculating the shortest paths between stations. By leveraging the comprehensive and well-structured data provided by the MetroNetworkParser, the Navigation class can focus exclusively on its algorithmic computations, optimizing both performance and accuracy.

G. Stage 5: Exception Handling and Error Reporting

- **Exception Handling**: Implemented robust exception handling mechanisms to ensure the program's stability and reliability. The application now effectively handles various exceptions, such as file parsing errors, invalid user inputs, and memory allocation failures, providing users with informative error messages and preventing program crashes.

H. Stage 6: Code Cleanup and Documentation

- **Documentation**: Provided comprehensive documentation for the entire codebase, including class descriptions, method summaries, and inline comments. This documentation serves as a valuable resource for developers,

facilitating code understanding, maintenance, and future enhancements.

- **Final Code Review**: with the flags -Wall -Wextra -Werror -pedantic -pedantic-errors -O3 to ensure that the code is clean, efficient, and free of errors. This rigorous code review process guarantees that the application adheres to the highest coding standards and best practices, delivering a robust and reliable solution.

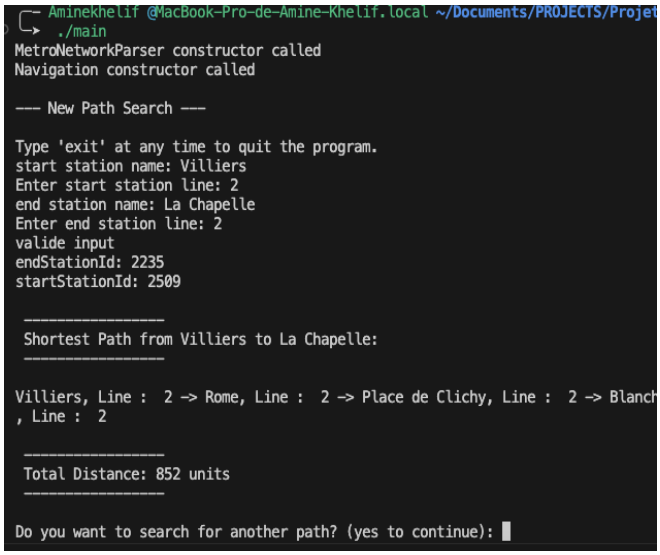

Fig. 6. final Project Inference Without Station Misspelling

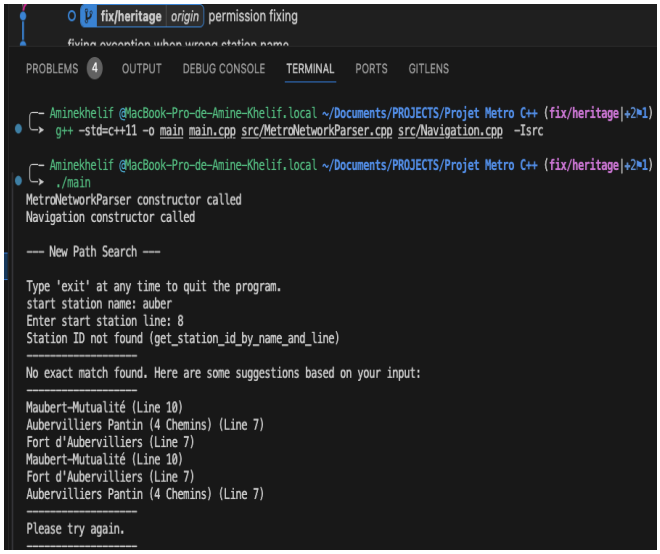

Fig. 7. final Project Inference With Station Misspelling

III. CONCLUSIONS

in developing a complex, real-world software application using the C++ programming language. Through the implementation of a practical navigation system for the Parisian subway, I not only refined my technical skills in C++ but also gained valuable insights into the intricacies of algorithm design, data management.

The use of Dijkstra's algorithm to calculate the shortest path between stations illustrated the power of classical algorithms in modern applications, while the progressive stages of development highlighted the importance of structured and incremental software development.

Furthermore, the project underscored the significance of clean coding practices, robust error handling, and thorough documentation. These elements are crucial for maintaining and scaling software efficiently, which I implemented diligently to ensure the program's reliability and user-friendliness.

In conclusion, this Advanced C++ course project not only tested my abilities to apply complex programming concepts but also prepared me for future challenges in software development. It was a definitive step in bridging the gap between theoretical knowledge and practical application