Final Assignment For ACIT4420 Problem-solving with scripting

candidate number
129

05 December 2024

Oslo Metropolitan University

Contents

1 Introduction

In today's fast-paced world, effective solutions for managing everyday challenges are essential. This project tackles two distinct yet practical problems, combining innovation with functionality. The first part, Tarjan Planner, addresses the complexities of urban travel by optimizing routes in metropolitan areas where multiple destinations and diverse transportation modes must be considered. The second part focuses on simplifying file management through a Python script that automatically categorizes files by type and organizes them into designated folders. Together, these solutions aim to enhance efficiency in both travel planning and digital organization.

1.1 Part I: Tarjan, the Thoughtful Planner

The first part of the project centers around Tarjan, a fictional character navigating the busy streets of Seoul to visit ten relatives during the festival season. With time constraints and diverse transportation options like buses, trains, bicycles, and walking paths, Tarjan needs a planner that calculates the most efficient way to reach all destinations. The goal is to develop a Python-based solution that optimizes routes based on different criteria, including the shortest travel time, minimum cost, or least number of transfer time. This part incorporates Python libraries like networkx and matplotlib for graph-based visualization and optimization, creating an interactive tool that factors in geographic data, transport speeds, and costs.

1.2 Part II: File Organizer

The second part of the project tackles a common organizational issue, sorting files in a directory. This Python script categorizes files based on their types and automatically organizes them into designated folders. The solution emphasizes modularity, error handling, and metaprogramming to allow dynamic addition of file categories. It demonstrates file manipulation techniques, configuration management, and logging, providing users with a flexible, automated file sorting tool.

Overall, this project showcases core competencies in Python scripting, spanning route optimization in urban settings to automated file organization. By applying concepts such as modular programming, error handling, and metaprogramming. the overall project is designed to be both practical and versatile,reflecting the capabilities gained throughout the Problem-solving with scripting course. The Tarjan Planner, as a part of this project, specifically demonstrates efficient travel route optimization in a metropolitan setting.

2 Design Overview

The design of this project was guided by the need to address two distinct but practical challenges with efficient and scalable solutions. The Tarjan Planner was conceptualized to handle the complexity of urban route optimization, emphasizing user needs such as time efficiency, cost-effectiveness, and minimal transfer time. Meanwhile, the File Organizer was designed to simplify digital disorganization by creating an adaptable, modular tool capable of dynamically categorizing and organizing files. Both components were developed with a focus on usability, robustness, and leveraging Python's capabilities to tackle real-world problems.

2.1 Part I: The Thoughtful Planner

TarjanPlanner is a modular Python package designed to solve an optimal route-finding problem in an urban setting. It uses various transport modes and optimizes routes based on user selected criteria, including least cost, minimal transfer time, and shortest travel time. The application offers a graphical user interface (GUI), enabling flexible interaction for users.

- **Design Principles**: The design of TarjanPlanner follows three main principles:
  - **Modularity**: Each major function or feature is encapsulated within its own module, making the codebase easier to maintain and extend.
  - **Flexibility**: The architecture allows easy addition of transport modes, criteria, and user interfaces.
  - **User-Centric Design**: GUI offer ease of use, enabling users to interact with the application based on their preferences.
- **System Architecture**: The architecture of TarjanPlanner consists of several independent modules:
  - **Data Loader**: Manages data loading from external files.
  - **Distance Calculation**: Computes distances between coordinates.
  - **Route Optimizer**: Contains algorithms for finding the shortest path and suggesting transport modes based on user-selected criteria.
  - **Transport Mode Manager**: Stores and calculates transport-specific properties like cost, speed, and transfer times.
  - **Logger**: Tracks and logs key events and performance data.
  - **Error Handling**: Manages custom exceptions for robustness.

  The following diagram outlines how these components interact within the application:

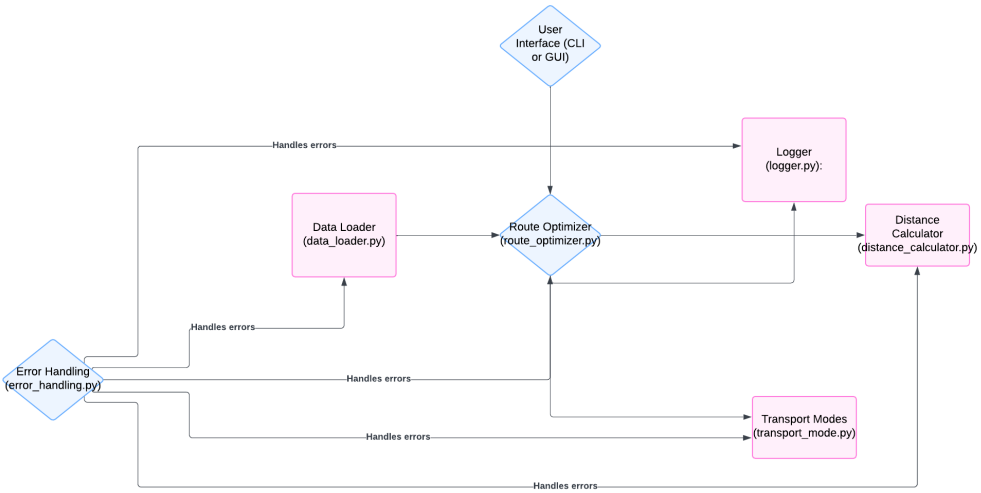

Figure 1: components interaction diagram

- **Data Flow**:

  1. **Data Loading:** TarjanPlanner begins by loading relative location data from a CSV file, which includes latitude and longitude for each relative. This data is loaded through the data_loader.py module and returned as a DataFrame to be used throughout the program.
  2. **Distance Calculation:** The distance_calculator.py module provides a function to compute geodesic distances between locations based on their coordinates. Using the geopy library, this module calculates distances in kilometers, which are then used in path calculations within route_optimizer.py.
  3. **Route Optimization:**
     - The route_optimizer.py module calculates an approximate shortest path using a greedy approach. It starts from an initial location and selects the nearest unvisited location at each step until all locations are visited.
     - Once the path is determined, the module offers additional functionality to adjust the route based on user selected criteria. The suggest_route_by_criteria function allows users to optimize for least cost, minimal transfers, or shortest travel time.
  4. **Transport Mode Management:**
     - Transport modes (train, bus, bicycle, walking) are defined in transport_mode.py, where each mode has specific properties, including speed, cost, and transfer time.

  - This module contains functions that calculate travel cost and time based on these properties, allowing for flexible adjustments to route optimization.

5. **Logging:** The logger.py module logs key actions and performance data, such as execution time for the route calculation. Logging helps monitor the application's performance and debug issues.
6. **Error Handling:** Custom exceptions are defined in error_handling.py to handle cases like data load errors and invalid transport modes. This improves robustness by preventing unexpected crashes and providing meaningful error messages to users.

- **User Interaction**:

  Users can launch the application using a graphical interface, where they can select optimization criteria through a Tkinter window. Upon selection, the GUI displays route suggestions along with a NetworkX graph representing the route.

- **Key Algorithms and Techniques**:
  - **Greedy Path Selection**: TarjanPlanner employs a greedy approach for determining the shortest path, which works well for the relatively small number of locations involved.
  - **Criteria-Based Route Adjustment**: After calculating the base shortest path, the route is adjusted based on the user's selected criteria, achieved by evaluating transport mode properties for each segment of the path.
  - **Decorator for Execution Time Logging**: A custom decorator in logger.py tracks and logs execution time for key functions, providing insights into performance.

- **Extensibility**: TarjanPlanner's modular design allows for easy extension in several areas:
  - **Adding New Transport Modes**: Simply add a new entry in TRANSPORT_MODES in transport_mode.py, specifying the mode's speed, cost, and transfer time.
  - **Additional Criteria**: New optimization criteria can be added to suggest_route_by_criteria in route_optimizer.py to enable further customization.
  - **Extended User Interfaces**: The existing GUI can be expanded to include a web-based interface or mobile compatibility in the future.

The design of TarjanPlanner prioritizes modularity, flexibility, and user-centric functionality, enabling both easy interaction and powerful route optimization. By combining structured data handling, flexible route calculations, and detailed logging, TarjanPlanner provides a practical and extensible solution to the optimal route-finding problem.

2.2 Part II: File Organizer

The File Organizer is designed as a modular Python package aimed at categorizing files within a directory based on file types, moving them to designated folders for efficient organization. The package is structured to facilitate ease of installation and usage, with components that handle configuration, sorting, and testing, ensuring flexibility and maintainability.

Package Structure

The File_Organizer_Project package follows a clear, organized layout that separates core functionality, configuration, and testing. Below is an overview of each main component:

1. **Package Directory** (file_organizer/)
    - The file_organizer/ directory is the core of the package, containing all primary modules necessary for file organization:
        - *__init__.py*: This file initializes the package, allowing it to be imported as a module.
        - *config.py*:A configuration module where users can customize sorting preferences, including adding new file categories without modifying core logic. This module dynamically adapts to user specifications, enhancing flexibility.
        - *generate_test_files.py*: Generates test files for quick verification and testing, enabling users to create sample files of different types within the target directory.
        - *main.py*: The main script that orchestrates the file organization process. This script leverages sorter.py and config.py to execute the sorting logic and
        - *sorter.py*: The sorter.py module implements the primary sorting mechanism by identifying file types using regular expressions and categorizing them based on extension patterns. This approach simplifies the addition of new categories, making the module extensible and capable of handling diverse file types.
        - *config.py:* The config.py module is designed to facilitate the dynamic categorization of files based on their extensions. It enables easy management and expansion of file categories without requiring modifications to the core logic of the file organization system. The module defines a flexible structure for adding new file categories and updating existing ones, ensuring adaptability as the system evolves.

2.3 Package Installation

- **Setup Script** (setup.py) setup.py is the setup script that configures the package for installation. It includes metadata and dependencies, ensur-

ing that users can easily install the package via pip or similar package managers.

- **README** (README.md) The README file serves as a guide for users, providing an overview of the package, installation instructions, usage examples, and config- uration options. It ensures that users can set up and use the package with minimal setup and guidance.

3 Implementation Details

This project solves two different problems, each with its own approach. The Tarjan Planner focuses on finding the best travel routes in cities using smart algorithms and modular design. The File Organizer handles the challenge of sorting and organizing files automatically. The sections below explain the key techniques, logic, and tools used to build these solutions.

3.1 Part I: Tarjan, the Thoughtful Planner

- **Assumptions**:

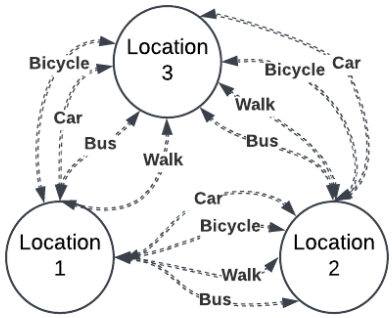

Figure 2: Each location has multiple transport options and is fully connected to all other locations

In implementing TarjanPlanner, we began by establishing clear assumptions that form the foundation of the project. These assumptions are directly tied to the problem's requirements and ensure that the objectives are met effectively:

  - **Transport Modes Are Universally Available**: Each location is assumed to have all transport modes (train, bus, bicycle, walking) readily accessible. This ensures the user has flexibility in choosing an optimization criterion (cost, transfers, or time) for every segment

of the route. This assumption simplifies the design while aligning with the problem's goal of offering tailored route recommendations.

- **Locations Are Modeled as a Fully Connected Graph**: To calculate distances dynamically, the locations are represented as a graph where every node (location) is connected to every other node. This simplifies the shortest path calculation while ensuring that all locations are considered without requiring predefined connections.
  figure 2 illustrates this concept, where Location 1, 2, and 3 are fully connected and have multiple transport options (e.g., train, bus, bicycle, and walking) available for traveling between them.
- **Static Properties of Transport Modes**: We have defined each transport mode with fixed characteristics, including speed (km/h), cost per kilometer, and transfer time. These properties are central to evaluating the optimal mode for each segment and allow the system to remain consistent and extensible.
- **A Greedy Approach for Shortest Path**: The greedy algorithm employed in this project, while not globally optimal for large datasets, is efficient and practical for the scope of this assignment. It demonstrates our understanding of algorithmic trade-offs: balancing simplicity, performance, and implementation feasibility.

- **Implementation Flow**: The implementation of TarjanPlanner builds upon the program's design to ensure practical functionality. Each stage is designed to handle specific tasks while keeping the overall program adaptable and efficient. This section explains how the program operates in practice, highlighting key decisions and the reasoning behind them.

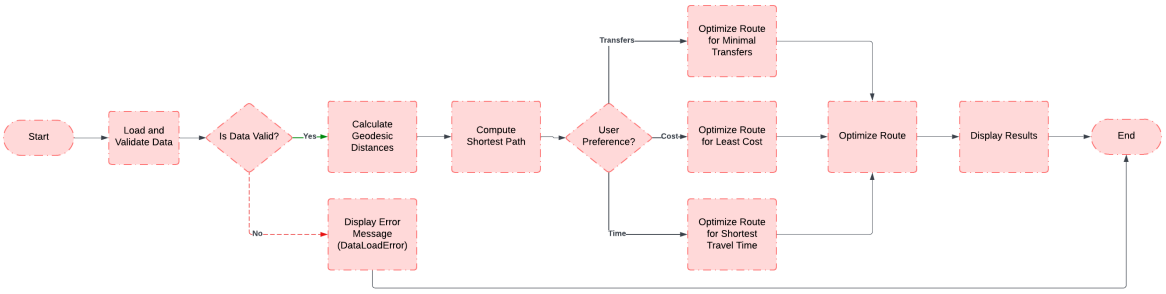

Figure 3: This visualization outlines the sequential steps of the program, from data validation and distance calculations to route optimization and result presentation.

1. **Validating and Structuring Data**: The program begins by loading location data, which includes the names and coordinates (latitude and longitude) of each relative. Before any calculations, the data is

validated to ensure that it is complete and correctly formatted. If the CSV file is missing, incomplete, or contains incorrect data, the program stops running and raises a DataLoadError with a descriptive error message. Once validated, the data is structured into a Pandas DataFrame, making it easy to process in subsequent steps. This step provides a strong foundation for accurate and reliable calculations by ensuring that all data-related issues are addressed early in the program.

2. **Dynamic Distance Calculations**: After loading the data, the program calculates the distances between locations dynamically.The program uses the geopy library to compute geodesic distances, representing the shortest path between two points on the Earth's surface. This dynamic calculation eliminates the need for static, precomputed distance matrices. It also ensures the program can adapt to changes, such as adding new locations. By prioritizing accuracy and flexibility, the program provides realistic and dependable distance calculations that are essential for reliable route optimization.

3. **Shortest Path Calculation** With the distances in place, the program calculates the shortest path that visits all locations. A greedy algorithm is used for this purpose, where the nearest unvisited location is selected at each step. While more sophisticated algorithms could offer globally optimal solutions, the greedy algorithm is an efficient and effective choice for the scale of this program, where datasets are relatively small. This step ensures that the program delivers quick and practical results, forming the backbone of the route optimization process.

4. **Optimizing the Route Based on User Preferences** Once the shortest path is calculated, the program optimizes it further based on the user's preferences. The user can choose one of three criteria such as least cost, minimal transfer time, or shortest travel time. For each segment of the route, the program dynamically evaluates the transport mode properties (speed, cost, and transfer time) and selects the mode that best fits the chosen criterion. This approach makes the program highly adaptable, allowing it to provide personalized recommendations for different user needs. For instance, a cost-focused route might prioritize buses, while a time sensitive route could favor trains. This step highlights the program's flexibility and user-centric design.

5. **Presenting Results**: The final step is to present the optimized route to the user. The program provides a Graphical User Interface (GUI) that offers a visual representation of the route using a graph. Built with NetworkX and Tkinter, the GUI uses color-coded edges and labels to indicate transport modes and distances, making the output intuitive and easy to understand. This interface ensures that the program accommodates users who prefer a more visual and interactive experience.

The implementation of TarjanPlanner is designed to balance efficiency, flexibility, and user-friendliness. It dynamically calculates distances and tailors routes to user preferences. Results are presented in multiple formats to ensure practicality and accuracy. Each step is carefully structured to keep the program adaptable to future enhancements, such as adding new transport modes or optimization criteria. This thoughtful design demonstrates a clear understanding of the problem and its practical solutions.

3.2 Part II: File Organizer

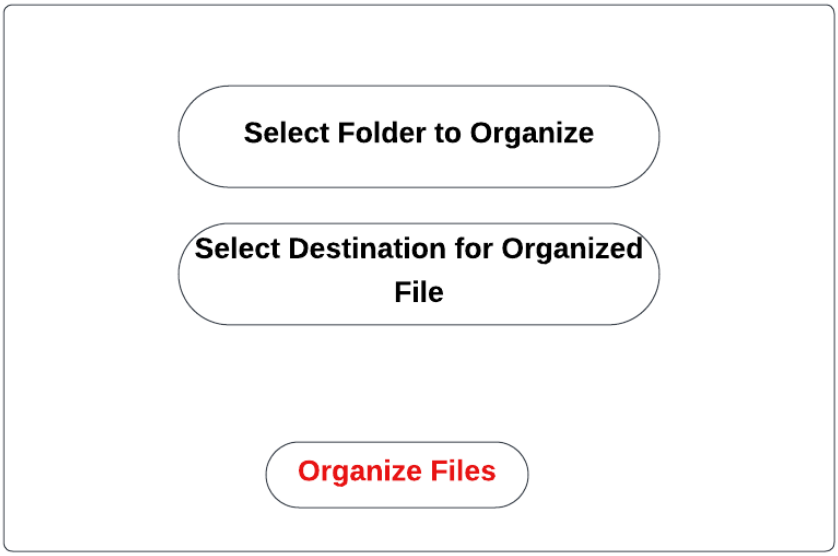

Figure 4: Prototype user interface for the file organizer

The File Organizer package is designed to efficiently categorize and sort files in a directory based on their types. Each component in the package contributes a specific functionality to the organization process, providing flexibility and adaptability for various file management needs. The following sections outline the core modules and their role in implementing the file organizer package

1. **Main Module** (main.py)
   - Acting as the entry point, main.py orchestrates the categorization workflow by invoking sorting and configuration logic within the package. This module manages the overall sequence, from identifying file types to relocating files to designated folders. By centralizing the execution flow here, the package maintains a clean and organized approach to file management, allowing main.py to serve as the only module directly called by the user.
2. **Sorting Logic** (sorter.py)
   - The *sorter.py* module implements the primary sorting mechanism, identifying file types using regular expressions and categorizing them based on extension patterns. Regular expressions are

used to detect specific file types, enabling accurate classification without needing to enumerate every possible extension. This pattern-matching approach simplifies the addition of new categories, making the module extensible and capable of handling diverse file types.

- The sorting logic also utilizes metaprogramming, specifically decorator functions, to allow dynamic updates to the file type categories. This method enables users to register new file categories as needed, supporting custom configurations without altering core code.

3. **Configuration Management** (config.py)
   - *config.py* offers a flexible configuration interface that enables users to define file categories and their destination folders. By centralizing configurations, this module simplifies the categorization process, allowing users to adjust directory paths or add new categories without modifying the sorting logic itself.
   - A decorator pattern is used within config.py to dynamically manage new file types and destination mappings. This technique leverages Python's metaprogramming capabilities to register new categories as decorators, allowing the program to adapt to new requirements on the fly. This design choice minimizes code adjustments and improves the package's adaptability to different environments.
4. **Error Handling and Logging** (logger.py)
   - Robust error handling is integrated into each module, particularly in *main.py* and s*orter.py*, to manage issues like permission errors, missing directories, or invalid file types. When errors occur, user-friendly messages are logged and displayed, helping users understand and resolve issues without needing to trace the error origin manually.
   - The *logger.py* module captures and logs each sorting operation, recording successful file movements and any encountered errors. This log information aids troubleshooting, provides an audit trail, and helps users track the organization process over time. By offering clear, organized logging, *logger.py* ensures that users can easily monitor the file organization's success and troubleshoot as needed.

Through its structured, modular design, the File Organizer package offers a customizable, user-friendly approach to file management, effectively handling diverse file types with minimal configuration. By leveraging regular expressions, metaprogramming, and robust error handling, the package delivers a scalable solution for automated file categorization.

4 Testing and Evaluation

The testing and evaluation process for this project focuses on validating the core functionalities of its two distinct parts, the file organizer and the route optimizer. Unit tests were implemented to ensure that the primary functions of each component operate as expected. These tests provide a foundational level of assurance but are designed to validate only the core features and do not cover exhaustive edge cases or performance testing. This section describes the testing approach for each part, emphasizing the critical aspects validated and insights gained.

4.1 Part I: Tarjan, the Thoughtful Planner

The route optimizer is the computational backbone of this project, responsible for calculating distances, finding the shortest path, and optimizing routes based on user preferences. The testing focused on multiple core functions to confirm reliability and correctness.

The calculate_distance function, located in the distance calculator module, computes geodesic distances between two geographical points. Test cases provided specific latitude and longitude values to validate the calculated distances against expected real-world values. For example, the distance between two test coordinates was validated to be 4.58 kilometers. Edge cases, such as scenarios where two locations had identical coordinates, were also tested to ensure the function handled them appropriately.

The find_shortest_path function in the route optimizer module was tested for its ability to generate a route that visits all specified locations using a greedy algorithm. The tests confirmed that the generated path included all locations and that the total distance matched the geodesic distances computed earlier. The function effectively produced valid routes for small datasets.

The suggest_route_by_criteria function was tested with all three available criteria: least cost, minimal transfers, and shortest travel time. Each test checked whether the route was adjusted correctly based on user selection. For instance, when the cost criterion was chosen, the function consistently selected the most economical transport mode. The tests demonstrated the function's adaptability and correctness in assigning transport modes based on user preferences.

The transport mode calculations for cost and travel time were validated through tests in the transport mode module. Test cases for a fixed distance of 10 kilometers ensured that the calculations reflected the properties of each mode, such as speed, cost per kilometer, and transfer time. The results matched predefined expectations, confirming the accuracy of these calculations.

Additionally, the log execution time decorator in the logger module was tested to ensure that it successfully measured and logged the execution time of wrapped functions. A sample function was wrapped with the decorator, and its runtime was logged accurately, confirming that the decorator worked as intended.

The unit tests for the route optimizer validated its core functionalities, confirming that it performs as intended under standard conditions. However, further enhancements are needed, such as expanding edge case coverage, introducing integration testing to validate seamless module interactions, and conducting performance testing to evaluate scalability with larger datasets.

4.2 Part II: File Organizer

The file organizer's functionality is spread across three main components: file sorting, configuration management, and the GUI interface. Tests were written for each of these areas to validate the core functionalities and ensure reliable behavior.

The organize_files function, located in the sorter module, was tested to confirm its ability to categorize and move files based on their extensions. The tests simulated a realistic directory structure by creating source and destination folders in a temporary environment. Files such as image.jpg, document.pdf, and spreadsheet.xlsx were placed in the source directory, and their movement into corresponding folders like Images, Documents, and Spreadsheets in the destination directory was verified. Custom file types, such as Audio files, were dynamically added to the configuration, demonstrating the function's flexibility in handling new categories.

The move_file function, a core utility within the sorting process, was tested independently to validate its ability to move individual files. Tests confirmed that files were correctly placed in their target directories and that errors, such as attempting to move non-existent files, were handled gracefully without creating invalid directories or throwing exceptions.

The add_file_type function in the config module was tested to ensure that new categories and associated file extensions could be dynamically added. Tests validated that when new categories, such as Audio, were introduced with extensions like .mp3 and .wav, the system correctly updated its configuration and recognized these files during subsequent organization tasks. Duplicate categories or redundant extensions were also handled appropriately.

The GUI component, implemented in the FileOrganizerApp class, was tested to verify its interaction with the underlying sorting logic. Mock inputs were used to simulate user directory selection, ensuring that the source and destination directory variables were correctly updated. The

GUI-triggered sorting process was tested to confirm that it displayed success messages upon completion, validating the interface's usability and functionality.

The testing process confirmed that the file organizer reliably categorizes and moves files. Its dynamic configuration capabilities and intuitive GUI enhance usability, making it robust and scalable for various use cases.Future efforts should focus on expanding test coverage to include edge cases and performance scenarios, ensuring the system remains robust and scalable.

5 Results and Discussion

The results and discussion section focuses on analyzing the outcomes of the two main components of this project: the route optimizer in TarjanPlanner and the file organizer. This section highlights the key findings, explores the implications of the results, and evaluates the effectiveness of each component in addressing the project goals.

5.1 Part I: Tarjan, the Thoughtful Planner

The Route Optimizer in TarjanPlanner calculates the shortest path between a series of locations, with the flexibility to optimize the journey based on user-selected criteria such as minimal cost, minimal transfers, or shortest travel time. Importantly, the shortest path remains constant across all criteria, ensuring that all locations are visited in the same sequence. The variations arise from how transport modes are dynamically selected based on the user's preference. Below, we document the results for each criterion and discuss their implications.

Figure 5 shows the relative locations and distances between all nodes, representing the different locations to be visited. Each edge is labeled with the distance between respective nodes, which serves as the basis for calculating the shortest path. This visual representation helps illustrate the geographic layout of the locations, providing insight into how the Route Optimizer determines the most efficient path. By using this graph, we can understand how the optimizer evaluates distances to select the best route depending on user criteria.

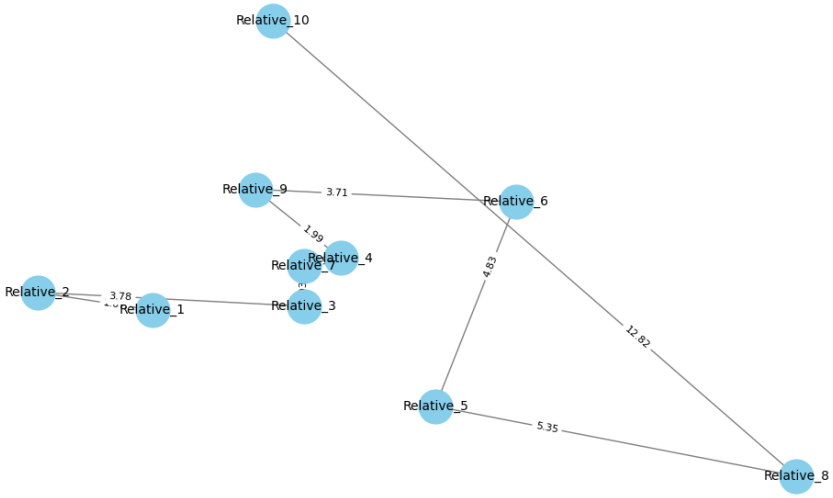

Figure 5: Relative Locations and Distances for Route Optimization

5.1.1 Optimized for Cost

Figure 6: Route optimized for minimal cost, selecting bicycles for all segments to maintain zero cost.

The cost-optimized route avoids paid transport modes like trains and prioritizes cost-free options. By selecting bicycles instead of walking, the optimizer significantly reduces travel time while maintaining zero cost.

5.1.2 Shortest Travel Time

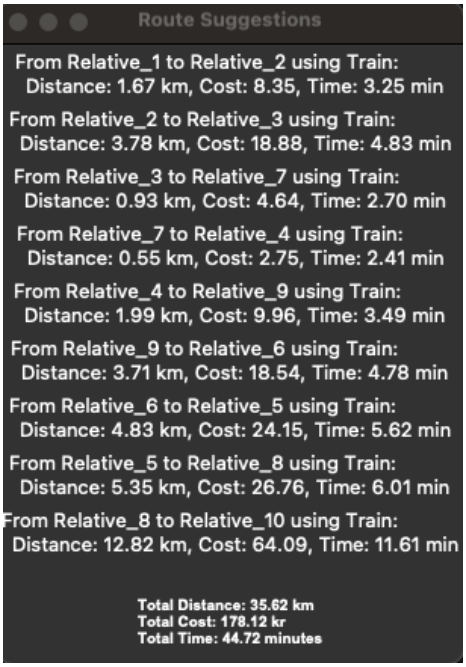

Figure 7: Route optimized for shortest travel time, selecting trains for all segments to minimize total travel duration.

When optimizing for time, the system consistently chooses trains, which are the fastest transport mode available. The results highlight the effectiveness of the time optimization criterion, as the total travel time is significantly reduced compared to bicycling or walking. This criterion is ideal for users with strict time constraints and demonstrates the program's ability to balance cost and time trade-offs based on user preferences.

5.1.3 Minimal Transfers Optimization

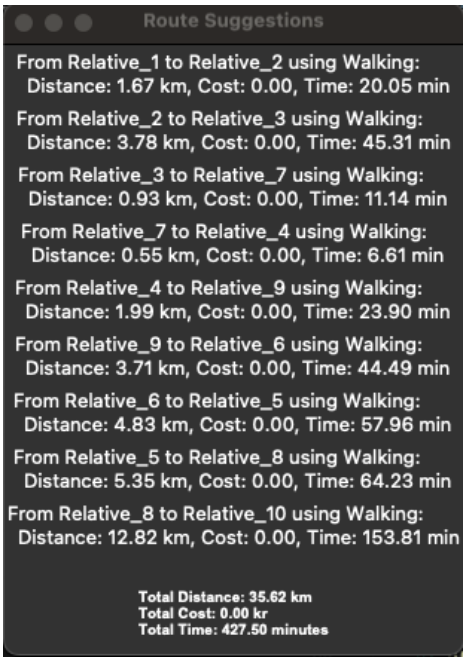

Figure 8: Route optimized for minimal transfers, consistently using walking to avoid mode changes

For minimal transfer time, the optimizer prioritizes walking for all segments to avoid the need for switching between transport modes. This criterion is especially useful for users who value simplicity and minimal disruption during their journey. While the travel time is longer than that of bicycles or trains, walking ensures there are no transfer points, providing a straightforward and uninterrupted route. This outcome showcases the system's adaptability to user preferences.

The results above highlights the flexibility of the Route Optimizer in meeting diverse user specific preferences while maintaining the same shortest path across all criteria. The optimizer effectively adapts transport modes to align with user priorities, demonstrating a strong ability to balance cost, time, and convenience.

The shortest path, calculated using a greedy algorithm, remains constant across all optimization criteria. This ensures that all locations are visited in the most efficient sequence. Variations arise only in the transport modes assigned to each segment. The consistency in the path provides a reliable baseline, allowing users to focus solely on their preferences without worrying about deviations in the route structure. The program provides

clear and detailed metrics for each optimization criterion, including total distance, total cost, and total time. This transparency enables users to make informed decisions based on their specific priorities. Additionally, the GUI visualization enhances usability by presenting the shortest path with annotated distances and the selected transport modes for each segment, making it intuitive and accessible to users with varying technical backgrounds.

The program demonstrates robust performance for small to medium-sized datasets. It efficiently handles scenarios where all transport modes are uniformly available, ensuring logical and user-friendly outcomes. However, for enhanced practicality in real-world applications, future iterations could incorporate additional considerations, such as transport mode unavailability, localized limitations, or external factors like traffic conditions.

The Route Optimizer results demonstrate its ability to provide practical and customizable solutions for diverse user needs. The updated logic for tie-breaking based on secondary criteria, such as travel time, ensures logical decision-making even when transport modes have identical costs. By maintaining a consistent shortest path and allowing dynamic variations in transport modes, the program effectively balances efficiency and flexibility. These attributes, combined with its robust design and user-centric features, establish a strong foundation for future enhancements in personalized route planning.

5.2 Part II: File Organizer

The File Organizer component effectively categorizes and organizes files into their appropriate directories based on file extensions. During testing, it demonstrated high accuracy and flexibility, with capabilities to dynamically adjust to new file types and deliver a smooth user experience. Below is a detailed evaluation of the system's results and its performance in various scenarios.

The File Organizer successfully classified and moved files to their designated directories during testing. For instance, files such as 'image.jpg', 'document.pdf', and 'spreadsheet.xlsx' were accurately moved into the Images, Documents, and Spreadsheets directories, respectively. The system also supported the dynamic addition of new file categories. When a new category, such as Audio, was introduced with extensions like '.mp3' and '.wav', these files were correctly categorized and sorted into their designated folders.

The program demonstrated robust error-handling capabilities. Cases such as missing files, unsupported extensions, or inaccessible directories were managed gracefully, with appropriate error messages provided to guide the user. This ensured that users could resolve issues effectively without

experiencing disruptions, contributing to the program's reliability and robustness.

The File Organizer showcased its ability to efficiently classify files under a variety of conditions. Its strong performance can be attributed to several key features, including classification accuracy, dynamic configuration, and robust error handling. The ability to dynamically configure the program to handle new file types allowed users to add new categories and associated extensions seamlessly, making the File Organizer scalable and adaptable to diverse file management needs.

The graphical user interface (GUI) allowed users to interact with the system visually by selecting source and destination directories, making it accessible even to non-technical users. The system provided clear success messages and feedback after file organization tasks were completed, adding transparency and confidence in its operations.

The program performed efficiently for small to medium datasets. However, its performance in scenarios with large numbers of files or deeply nested directory structures remains unexplored. Future evaluations should focus on scalability to ensure that the program handles larger datasets without compromising performance.The addition of parallel processing or threading could improve efficiency when organizing thousands of files.

6 Future Work

Building on the strong foundation of this project, several enhancements can be explored to improve both components. For the TarjanPlanner, future iterations could incorporate advanced algorithms like Dijkstra's or A* to optimize routes more effectively, particularly for larger datasets or complex urban networks. Integrating real-time data such as traffic conditions, public transport schedules, and weather updates would add a dynamic element, making the tool more applicable in practical, real-world scenarios. These additions would enhance the accuracy and relevance of the route suggestions, ensuring that the tool remains robust under varying conditions.

Expanding the user interface for TarjanPlanner could significantly enhance its accessibility and usability. A mobile or web-based version of the tool would allow users to access the planner on the go, catering to modern convenience. Features such as offline functionality, saving frequently used routes, and incorporating interactive maps would make the tool more user-friendly and adaptable. These developments would align with a user-centric approach, ensuring that the planner serves a broader audience with diverse needs and preferences.

Similarly, for the File Organizer, future enhancements could include advanced sorting capabilities. Allowing users to sort files not only by type

but also by attributes such as size, creation date, or metadata would provide greater control over file management. Additionally, optimizing performance through multithreading or parallel processing could make the tool more efficient, especially when managing large numbers of files or deeply nested directory structures. These updates would ensure that the tool scales effectively to handle more complex file organization tasks.

The File Organizer's utility can be extended by integrating cloud storage solutions such as Google Drive, OneDrive, or Dropbox. This would enable cross-platform file management. Incorporating machine learning algorithms could also allow intelligent categorization based on user behavior, making the tool more adaptable and proactive in predicting organizational needs.These enhancements would not only increase the flexibility of the tool but also its appeal to both technical and non-technical users, positioning it as a comprehensive solution for modern file management challenges.

7 Conclusion

This project successfully addresses two significant and distinct challenges, optimizing urban travel routes and organizing digital files efficiently. The two components, TarjanPlanner and File Organizer, each tackle a unique problem, demonstrating the versatility and power of Python scripting in practical applications.

The TarjanPlanner component provides a solution for planning complex routes involving multiple destinations and varied transportation modes. By allowing users to optimize based on criteria such as minimal cost, minimal transfers, or shortest travel time, TarjanPlanner offers flexible and efficient route planning. The use of a greedy algorithm and dynamic evaluation of transport options ensures practicality, while the interactive Graphical User Interface (GUI) enhances usability, making the tool accessible to a broad audience.

The File Organizer component simplifies file management by categorizing files based on extensions and automating their organization into designated folders. Its modular design allows for dynamic configuration, enabling easy adaptation to new file types and categories. The implementation of regular expressions for file type detection, combined with robust error-handling capabilities, ensures reliable performance. The GUI further improves accessibility, allowing non-technical users to interact with the tool intuitively.

Testing and Evaluation confirmed that both components fulfill their intended functions and provide strong adaptability to various scenarios. Although the current versions are effective for standard use cases, there

are opportunities for future enhancements. For TarjanPlanner, scalability testing and considerations for real-world factors like traffic conditions could make the solution more robust. For the File Organizer, optimizations for handling larger datasets and adding more advanced sorting features would further enhance its utility.

In conclusion, the TarjanPlanner and File Organizer components collectively demonstrate how Python scripting can be leveraged to solve diverse real-world problems. Through modularity, user-centric design, and flexibility, the project provides comprehensive and practical solutions to route optimization and file management challenges, establishing a strong foundation for future development and customization.