UROP 1803

SUTD Mapping Project

Ryan Pek[1], Lee Cheng Yong[1], Simon Perrault[2]
Singapore University of Technology and Design

Acad Year 2024 Sep (2430) to Acad Year 2025 Jan (2510)

[1]Student
[2]Supervisor

Contents

Abstract

This project presents a Python3 client based utility designed to assist users in navigating locations within the Singapore University of Technology and Design (SUTD) campus. Unlike existing solutions, our application offers dynamic path-finding capabilities tailored for new and visiting individuals. By leveraging built-in Python libraries and custom modules, the project emphasizes educational growth and minimizes external dependencies. Additionally, we conducted primary data collection to accurately map campus distances, ensuring reliability and precision in navigation. The project is open-sourced to encourage further development, data validation and integration by the community in SUTD.

1 Introduction

Navigating the Singapore University of Technology and Design (SUTD) campus can be challenging for new and visiting individuals. Existing solutions, such as the official Campus Map site and the Telegram-based Sam - SUTD Assistive Maps bot, provide general location descriptions but lack comprehensive path-finding functionalities.

To address this gap, the SUTD Mapping Project aims to develop a Python 3 utility that offers detailed navigation assistance within the campus. Python was chosen due to its accessibility and prevalence within the SUTD community, particularly in common computing modules.

From the project's inception, the team prioritized minimizing dependencies on external libraries. By utilizing Python's built-in libraries and developing custom modules, we enhanced the educational value of the project, fostering a deeper understanding of software development and algorithm design.

One significant challenge encountered was obtaining accurate distance data for campus locations since the data was not readily available. The team used surveying tools to gather actual measurements, allowing us to set the weights for representing the campus in graph format.

We intend to open source the project, enabling other developers and interested parties to utilize our dataset and tools in their own projects. Future developments worth considering include advanced features such as real-time navigation updates, integration with mobile platforms, and expanded data sets for more comprehensive coverage.

The source code and dataset we collected for the project can be found and cloned from Github, this report will detail the intended usage of the utility. Specifically, the individual floor data can be found in json format under here. The timeline and changes to the program can be viewed through the project's commit history.

A key learning objective we have set for this UROP apart from writing software is to experience firsthand data collection, building of software to parse the data and the complications involved in the process of doing so.

2 Methodology

2.1 Data Collection

Campus Layout SUTD comprises four main academic buildings (1, 2, 3, and 5) plus residential and recreational blocks. We set out to focus on the academic cluster (1, 2 and 5) due to its high undergraduate footfall.

Survey Procedure

- **Reference Maps**: Fire-safety boards and lift lobby floor plans were photographed for initial node placement.
- **Tools**: Laser range-finders and a surveyor's wheel captured linear distances; a lensatic compass recorded bearings.
- **Node Definition**: Traversable locations with signage, visual landmarks, or inter-building connections were instantiated as graph vertices. Specifically, we focused on identifying venues/rooms that make practical and realistic sense like lecture theatres, classrooms and offices.

2.1.1 Equipment Used

(a) (b) (c)

Figure 1: (a) Compass (b) Laser Rangefinder (c) Surveyor's wheel

2.1.2 Campus Layout

We initially attempted to source the campus layout via the Office of Campus Infrastructure and Facilities but was unable to get a response. Instead, we opted to make use of either the floor load plan that is available at every service lift, or the floor plan based on the fire safety board, whichever was more readable and accessible.

SUTD occupies a relatively isolated plot of land in the rough shape of a square, with two sides adjacent to the main road, one side to a canal and the last to a fenced residential area. The campus itself consists of 4 faculty buildings (Buildings 1,2,3,5)[3], 5 residential accommodations (Buildings 51,53,55,57,59), 1 recreational building (Building 61). Of which, we have decided to map the main faculty buildings since this cluster of buildings have the highest footfall and has the highest chance of having visitors unfamiliar to the University to visit. Making Buildings 1,2,3 and 5 the most suitable region to be mapped.

To plot out the nodes and edges on the maps, we had printed out the maps, followed by identifying locations of interest by hand. As a general rule of thumb, locations that are significant typically have names, are easily accessible, and places that lead to locations outside of the current map.

[3]There is no Building 4

2.1.3 Mapping details

Each location in the university can be represented as nodes in a graph, with the shortest traversable path and distance to the nearest neighbour of a location being the edge and weights respectively.

While it is possible to store every node on one graph, in practice this will make the data difficult to read and maintain. We have separated the nodes into clusters based on which building and which floor they exist in; with the exception of the First Floor, which will act as one large map. The decision to combine the first floor was to simplify the calculation required to connect the different portions of the first floor.

Using Python 3, we can use the built-in data structure of dictionary to represent this graph. In our implementation, each node will be a key in the dictionary key-value pair, while the neighbours are keys in a dictionary nested in the original node's value in it's key-value pair.

To store and load the data, we can make use of the built in json module, storing the data from each floor in its own json file. The design choice was to keep the data maintainable, since large scale changes often occur on a floor to floor basis, allowing us to access and modify the dataset more conveniently.

The json format also gives the dataset scalability, we have tried to keep to the standard json data types when storing data of the nodes. In theory this should allow the data set to be used outside of Python (and by extension native Javascrypt) so long as they can parse json and store the data types within json specifications.

An average map will contain roughly 30kb of data, consisting of the node, its neighbours and the related distances and bearing information, modifiers for the program to change the node's weight as well as meta data of the node for description purposes.

2.2 Interface

The program was designed to be used in CLI in order to focus more on the software architecture, algorithms and data collection.

As mentioned previously, we have also drafted a telegram bot for users to query a path from their current location to their desired location. The path returned will be a descriptive step by step instruction list, consisting of landmark descriptions, distance and direction from our dataset.

Telegram was chosen as a platform due to its widespread use in the SUTD community, hosting the general university chat, as well as most if not all of the official club chats. Visitors to the University will likely have an account in order to communicate with SUTD groups, hence the convenience in integrating this utility into a telegram bot.

2.3 Running the CLI Program

To make use of the utility on a terminal, clone the project from GitHub and follow the steps below:

```
git clone https://github.com/rpeky/SUTD_Map_project.git
cd SUTD_Map_project/SUTD_Map_project
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
python3 SUTD_Map_project.py
```

The dependency used in the main file is matplotlib to plot out the graphs to visually check the output. Either run in a pyton virtual environment or comment out the sections using matplotlib.

2.4 Custom Modules

We have written a set of custom modules to more easily access and log changes from our python dictionary containing the graph information of a floor to json format.

The following sections delineate the primary custom modules utilized in the project: `Graph.py`, `Json_OS_ProcessingFunctions.py`, and `Path_query.py`. Each module is discussed in detail below, highlighting the purpose, functionalities, strategic implementations, and interactions with other modules.

2.5 Graph Data Structure and Its Role in This Project

A graph is a fundamental abstraction for modelling pairwise relationships. Formally, following the terminology of *Introduction to Algorithms* [1], a graph is an ordered pair $G = (V, E)$ in which:

- V is a finite, non-empty set of **vertices** (also called *nodes*); and
- $E \subseteq V \times V$ is a set of **edges** that connect vertices.

An edge $(u, v) \in E$ indicates that vertices u and v share a relationship. When edge pairs are *unordered* ($\{u, v\}$), we obtain an **undirected** graph whose connections work both ways. When order matters, the structure is a **directed** graph (*digraph*), suitable for modelling one-way streets, precedence constraints, or data-flow dependencies.

Weighted graphs. If every edge carries an associated numeric weight $w\colon E \to \mathbb{R}_{\geq 0}$—for instance distance, travel-time, or cost—then G is said to be *weighted*. In our case, the weights are the real measured distances between any two identified vertices in the campus.

Why a graph for campus navigation? The SUTD Mapping Project treats each physically significant location (lecture-theatre entrance, lift lobby, bridge portal, ...) as a vertex, and each traversable corridor or stair flight as an edge. Edge weights represent surveyed walking distances, and potentially later adjusted by live sensor-feedback (crowd density, rain exposure) to reflect real-time conditions.

This choice of representation delivers three key benefits:

1. **Algorithmic clarity.** Classic graph algorithms— breadth-first search for reachability, Dijkstra for optimal routing, A* for heuristic search—are directly applicable without ad-hoc special-case code.
2. **Extensibility.** Adding a new floor or building reduces to inserting vertices and edges; the underlying logic remains unchanged.
3. **Interoperability.** The graph can be exported as JSON and consumed by external visualisation tools, simulation engines, or mobile clients without violating interface boundaries.

Intuitively, it is tempting to describe the graph in which the university campus as an undirected weighted graph. However on further thought, specifically distances that are physically the same such as staircase distances or elevator traversals can be weighted further to describe the perceived distance rather than absolute.

For example, traversal of staircases is a fixed 20 meters up and down, but the perceived distance going up will be higher due to the additional effort required to ascend the stairs than descend. We can modify the distance by multiplying upwards staircase traversals with an arbitrary multiplier like 1.25 to compensate for this discrepancy.

Consequently the relationship between vertices is represented as a symmetric directed graph. A digraph is a graph in which: $\forall$ vertices $u, v \in V$, $(u, v) \in E \implies (v, u) \in E$. In the implementation below, the `Graph.py` module encapsulates all operations on this data structure — vertex management, edge manipulation, and path-finding; providing a clean API for the rest of the system to build upon.

2.6 Module: Graph.py

2.6.1 Purpose and Overview

The `Graph.py` module is foundational to the SUTD Mapping Project, encapsulating all graph-related functionalities necessary for mapping and pathfinding within the campus. It defines the `Graph` class, which manages the creation, manipulation, and analysis of graph structures representing campus locations and their interconnections. This module is instrumental in enabling users to add or remove vertices (locations), establish or sever connections (paths), and compute optimal routes using various algorithms.

2.6.2 Class: Graph

Attributes

- `vertex_template`: A class-level dictionary template defining the default structure and attributes of each vertex in the graph. Attributes include neighbors, coordinates, visitation status, density metrics, shelter status, connection points, and more.
- `dd_graph`: A dictionary representing the graph, where each key is a vertex identifier, and the value is a dictionary of vertex attributes.
- `dd_lkup`, `dd_cplkup`, `dd_idlkup`: Dictionaries serving as lookup tables for directory mappings, connection points, and location IDs respectively.
- `access_clearance`: A list indicating the clearance levels required to access certain vertices.
- `area_file_tosave`, `localname`: Strings storing filenames and local identifiers for graph data persistence.

Initialization (__init__) Upon instantiation, the `Graph` class performs the following steps:

1. Ensures the existence of necessary directories using `Json_OS_ProcessingFunctions.check_folders_exist()`.
2. Initializes graph-related attributes and lookup tables.
3. Checks for the existence of the specified area file. If it exists, the graph is loaded from the master directory; otherwise, the graph generation tool is invoked to create a new graph.
4. Initiates the graph generation tool to facilitate user-driven graph modifications.

Destructor (__del__) The destructor ensures that upon object deletion:

1. The current graph state is saved to the working directory.
2. Lookup tables are updated to reflect any changes in the graph.
3. Dijkstra's algorithm described in *Numerische Mathematik* [2] is executed across all vertices to pre-compute shortest paths.
4. Final log messages are printed for debugging and confirmation.

2.6.3 Core Functionalities

Graph Generation and Modification

- `add_vertex()`: Facilitates the addition of new vertices to the graph, ensuring unique identifiers and initializing default attributes.
- `remove_vertex()`: Enables the removal of existing vertices, along with their associated edges, maintaining graph integrity.
- `neighbour_tool(vertex_ID)`: Manages the addition of neighbors to a specified vertex, allowing for dynamic graph expansion.
- `modify_display_existing_vertex()`: Provides an interface for modifying attributes of existing vertices, such as density, shelter status, and connection points.
- `add_neighbour(vertex_ID, neighbour_ID)`: Establishes bidirectional connections between vertices, calculating distance and heading based on coordinates or user input.

Pathfinding Algorithms

- `Dijkstra_modified(startpoint)`: Implements Dijkstra's algorithm to compute the shortest paths from a given starting vertex to all other vertices within the graph.
- `Dijkstra_all()`: Executes Dijkstra's algorithm for every vertex in the graph, precomputing all-pairs shortest paths.
- `pathfind_long_rundijk_supermap()`: Utilizes a supermap—a comprehensive graph amalgamating all individual maps—to perform extensive pathfinding across the entire campus.
- `A_star(source, destination)`, `bfs(source, dest)`, `Ant_colony(source)`, `genetic_search(source)`: Placeholder methods indicating planned implementations of alternative pathfinding algorithms to enhance search efficiency and flexibility.

Utility and Helper Methods

- `query(query_type, prompt, options, quit_option, confirm_selected_option)`: A versatile method facilitating user interactions for selecting options, inputting text, or specifying ranges, with built-in validation and confirmation mechanisms.
- `distance_heading_to_dx_dy(distance, heading)` and `dx_dy_to_distance_heading(dx, dy)`: Static methods converting between polar (distance and heading) and Cartesian (dx, dy) coordinates to manage vertex positioning.
- `convertloc_todd(vtx)`, `verify_endpoint_samegraph(endpoint)`: Methods for translating vertex identifiers and verifying graph consistency.
- `set_coordinates(vertex_ID)`, `set_room_ID(vertex)`, `set_Average_travel_time(vertex)`: Methods for updating vertex attributes based on user input or algorithmic calculations.

2.6.4 Graph Persistence and Data Management

The `Graph` class leverages the `Json_OS_ProcessingFunctions` module to handle data persistence:

- `store_solution_Master()`: Saves the current graph state to the master directory for long-term storage.
- `store_solution_Working()`: Saves the graph state to the working directory for ongoing modifications.
- `check_area_file_exist(area_file)`: Verifies the existence of a graph file within the master directory.

2.6.5 Error Handling and Validation

Robust error handling mechanisms are integrated throughout the `Graph` class to ensure reliability:

- Input validations for user selections, ensuring indices and inputs are within valid ranges.
- Confirmation prompts for critical actions such as adding or removing vertices and modifying attributes.
- Graceful exits and returns to previous menus upon invalid inputs or user-initiated quits.

2.6.6 Strategic Implementations

Modular Design The `Graph` class is meticulously designed to encapsulate all graph-related operations, promoting reusability and ease of maintenance. By segregating functionalities into distinct methods, the class facilitates targeted updates and scalability.

User-Centric Interface Interactive methods like `add_vertex()`, `remove_vertex()`, and `modify_display_existing_vertex()` prioritize user engagement, providing intuitive prompts and feedback to guide users through graph modifications seamlessly.

Efficiency in Pathfinding Implementing Dijkstra's algorithm and planning for additional algorithms (A*, BFS, Ant Colony Optimization) demonstrates a commitment to optimizing pathfinding performance. The supermap approach further enhances efficiency by consolidating all maps into a singular graph structure, reducing computational overhead during cross-map navigation.

Data Integrity and Consistency By maintaining and updating comprehensive lookup tables (`dd_lkup`, `dd_cplkup`, `dd_idlkup`) and ensuring synchronized updates across graph modifications, the module upholds data integrity, preventing inconsistencies and facilitating accurate pathfinding results.

Extensibility The modular architecture and clear method delineations allow for effortless integration of future enhancements, such as real-time data adjustments based on environmental factors (e.g., density changes during lunch hours) and the incorporation of advanced pathfinding heuristics.

2.6.7 Integration with Other Modules

The `Graph.py` module is tightly integrated with the `Json_OS_ProcessingFunctions.py` module:

- `Json_OS_ProcessingFunctions.py`: Utilized for loading and saving graph data, managing directories, and handling logging operations.
- `Path_query.py`: Interfaces with `Graph.py` to perform pathfinding operations based on user inputs.

This integration ensures that graph-related functionalities operate on up-to-date and accurate data, providing reliable navigation assistance to users.

2.6.8 Conclusion

The `Graph.py` module is a pivotal component of the SUTD Mapping Project, seamlessly integrating graph management, user interaction, and pathfinding algorithms to provide a robust navigation tool. Its modular design and reliance on custom processing functions and data structures ensure both flexibility and scalability, laying a solid foundation for future enhancements and feature additions.

2.7 Module: Json_OS_ProcessingFunctions.py

2.7.1 Purpose and Overview

The `Json_OS_ProcessingFunctions.py` module serves as a utility library for managing JSON file operations, directory validations, and logging mechanisms. It abstracts the complexities of file I/O and system interactions, providing streamlined functions that facilitate data persistence, retrieval, and maintenance across the project.

2.7.2 Core Functionalities

Directory Management

- `check_folders_exist()`: Ensures the presence of essential directories (`Master`, `Working`, `LookUp`, `Misc`, `Log`) within the project's root. If a directory is absent, it is created, and the action is logged.

File Operations

- `check_file_exist(filename, folder_idx)`: Verifies the existence of a specified file within a designated directory, returning a boolean result.
- `save_file_json(tosave, filename, folder_idx)`: Saves a Python dictionary or list to a JSON file within the specified directory, formatting the output for readability.
- `load_file_json(filename, folder_idx)`: Loads and returns the contents of a JSON file as a Python object.
- `pullup_vertices(filename, folder_idx)`: Retrieves and returns a list of vertex identifiers from a specified JSON file.

Logging Mechanism

- `generate_logfile(logmsg)`: Appends timestamped log messages to a `logs.log` file within the `Log` directory, facilitating tracking of system actions and debugging.
- `clear_logfile()`: Clears the contents of the `logs.log` file, resetting the log history.

Lookup Table Reconstruction

- `rebuild_lookupdir()`: Reconstructs the `Lookup_directory.json` file by scanning all graph files in the `Master` directory, ensuring accurate mappings between vertices and their corresponding files.
- `rebuild_lookupcon()`: Rebuilds the `Lookup_connections.json` file by identifying vertices marked as connection points across all graph files, maintaining up-to-date connection mappings.
- `rebuild_locationID()`: Regenerates the `Lookup_locationID.json` file by extracting room IDs from graph files, ensuring accurate associations between room codes and vertex identifiers.

Supermap Generation

- `generate_supermap()`: Aggregates all individual graph files within the `Master` directory into a comprehensive `.supermap.json` file, enabling large-scale pathfinding operations across the entire campus.

2.7.3 Strategic Implementations

Abstraction and Re-usability By encapsulating common file and directory operations within dedicated functions, the module promotes code reuse and reduces redundancy. This abstraction allows other modules (e.g., `Graph.py` and `Path_query.py`) to perform file-related tasks without delving into the intricacies of file handling.

Data Integrity and Consistency The module ensures data integrity by:

- Validating the existence of necessary directories and files before performing operations.
- Maintaining synchronized and up-to-date lookup tables through reconstruction functions.
- Formatting JSON outputs consistently, facilitating interoperability and ease of data manipulation.

Logging for Debugging and Auditing The integrated logging mechanism records critical actions and events, providing a valuable trail for debugging and auditing purposes. Timestamped logs offer insights into the system's operational history, aiding in troubleshooting and performance analysis.

Scalability through Supermap Generation The ability to generate a supermap underscores the module's scalability, enabling the system to handle extensive graphs encompassing the entire campus. This strategic implementation ensures that pathfinding algorithms can operate efficiently on large datasets, supporting comprehensive navigation functionalities.

Error Handling and Recovery Functions like `check_folders_exist()` and `check_file_exist()` incorporate error checking to prevent runtime issues caused by missing directories or files. By proactively managing such scenarios, the module enhances the overall robustness of the software utility.

2.7.4 Integration with Other Modules

The `Json_OS_ProcessingFunctions.py` module is tightly integrated with both the `Graph.py` and `Path_query.py` modules:

- `Graph.py`: Utilizes file loading and saving functions to manage graph data persistence.
- `Path_query.py`: Relies on lookup tables and supermap data for accurate pathfinding operations.

This integration ensures that data management operations are consistent and efficient across the entire software architecture.

2.7.5 Conclusion

The `Json_OS_ProcessingFunctions.py` module is a vital utility library within the SUTD Mapping Project, providing essential functionalities for data management, directory validation, and logging. Its strategic design emphasizes abstraction, reusability, and data integrity, ensuring seamless integration with other modules and facilitating robust and scalable software development.

2.8 Module: Path_query.py

2.8.1 Purpose and Overview

The `Path_query.py` module is a critical component of the SUTD Mapping Project, responsible for handling user interactions related to pathfinding within the SUTD campus. It defines the `Query` class, which facilitates the selection of start and end locations, translates user inputs into internal vertex identifiers, and computes the shortest paths using graph-based algorithms. This module leverages the functionalities provided by the `Graph.py` and `Json_OS_ProcessingFunctions.py` modules to deliver an efficient and user-friendly navigation experience.

2.8.2 Class: Query

Attributes

- `dd_locationid`: A dictionary loaded from `Lookup_locationID.json`, mapping user-entered location IDs to internal vertex names.
- `dd_masterlookup`: A dictionary loaded from `Lookup_directory.json`, associating internal vertex names with specific map files.

Initialization (__init__) Upon instantiation, the `Query` class performs the following actions:

1. Loads essential lookup tables:
   - `Lookup_locationID.json`: Maps location IDs to internal vertex identifiers.
   - `Lookup_directory.json`: Associates internal vertex names with corresponding map files.
2. Prints the loaded data for debugging purposes.
3. Displays a welcome message to the user.
4. Initiates the pathfinding process by invoking the method `pathfind_long_rundijk_supermap()`.

Destructor (__del__) The destructor ensures that a message is printed when an instance of the `Query` class is deleted, aiding in debugging and resource management:

```
def __del__(self):
print('test del query class')
```

2.8.3 Core Functionalities

User Interaction Methods

- `welcome_message()`: Displays an introductory message to the user.
- `display_options_startpoint()`: Presents users with options to select their starting or destination locations, such as entering a room ID directly or selecting from a predefined location list.
- `locationlist()`: Guides the user through a multi-step selection process, including building selection, floor selection, and specific location selection within the chosen floor.
- `inputroomID()`: Allows users to input a specific room ID, validating the input against the `Lookup_locationID.json` file and translating the valid ID to an internal vertex name.
- `startloc()`: Initiates the selection of the starting location by invoking `display_options_startpoint()`.
- `endloc()`: Initiates the selection of the destination location similarly.

Pathfinding Algorithms

- `dijkstra(sp)`: Implements a modified Dijkstra's algorithm to compute the shortest path from a starting vertex `sp` within a single map, returning the distance and a list containing the Path to reach the destination vertex.
- `pathfind_long_rundijk_supermap()`: Extends pathfinding across multiple maps using a supermap. It loads the comprehensive `.supermap.json` file, executes Dijkstra's algorithm from each vertex to every other vertex, and stores the computed shortest paths and distances in a solution dictionary.
- `twosidepfind(sloc, eloc)`, `pathfind_long_assumeleastmaps(sloc, eloc)`: Placeholder methods for planned implementations for more advanced pathfinding strategies.
- `bfs_supermap(source, dest)`, `genetic_supermap(source)`, `antcolony(source, dest)`: Placeholder methods for alternative pathfinding algorithms (e.g., Breadth-First Search, Genetic Algorithms, Ant Colony Optimization) to enhance search flexibility and efficiency.

Utility and Helper Methods

- `convertloc_todd(vtx)`: Translates a vertex name to its corresponding data dictionary by identifying the map it belongs to and loading the specific map data file.
- `translate_internalnameforoutput(raw_data)`: Formats internal location names into a user-friendly format by replacing underscores with spaces and capitalizing words, while preserving acronyms.

Pathfinding Orchestration

- `path_find()`: Orchestrates the overall pathfinding process by prompting the user to select starting and destination locations, determining if both locations reside within the same map, executing the appropriate pathfinding algorithm, and displaying the shortest path and distance to the user.

2.8.4 Error Handling and Validation

Robust error handling mechanisms are integrated throughout the `Query` class to ensure reliability:

- Validates user inputs against predefined options and existing data entries.
- Handles exceptions gracefully, providing informative feedback to users.
- Ensures that invalid or out-of-range inputs do not cause the application to crash.

2.8.5 Data Management

The module relies heavily on JSON files for data management:

- `Lookup_locationID.json`: Maps user-entered room IDs to internal identifiers.
- `Lookup_directory.json`: Associates internal identifiers with specific map data files.
- `.supermap.json`: A comprehensive graph representation of the entire campus used for advanced pathfinding.

By organizing data in this manner, the `Query` class efficiently accesses and manipulates necessary information to perform its functions.

2.8.6 Strategic Implementations

Modular Design The `Query` class is meticulously designed to encapsulate all pathfinding-related operations, promoting reusability and ease of maintenance. By segregating functionalities into distinct methods, the class facilitates targeted updates and scalability.

User-Centric Interface Interactive methods like `display_options_startpoint()`, `locationlist()`, and `inputroomID()` prioritize user engagement, providing intuitive prompts and feedback to guide users through the pathfinding process seamlessly.

Efficiency in Pathfinding Implementing Dijkstra's algorithm and planning for the integration of a supermap demonstrates a commitment to optimizing pathfinding performance. The supermap approach enhances efficiency by consolidating all maps into a singular graph structure, reducing computational overhead during cross-map navigation.

Data Integrity and Consistency By maintaining and updating comprehensive lookup tables (`dd_locationid`, `dd_masterlookup`) and ensuring synchronized updates across pathfinding operations, the module upholds data integrity, preventing inconsistencies and facilitating accurate navigation results.

Extensibility The modular architecture and clear method delineations allow for effortless integration of future enhancements, such as real-time data adjustments based on environmental factors (e.g., density changes during lunch hours) and the incorporation of advanced pathfinding heuristics.

2.8.7 Integration with Other Modules

The `Path_query.py` module is tightly integrated with both the `Graph.py` and `Json_OS_ProcessingFunctions.py` modules:

- `Graph.py`: Utilized for graph-based operations such as loading maps, managing vertices, and executing pathfinding algorithms.
- `Json_OS_ProcessingFunctions.py`: Handles data persistence, loading, and saving of JSON files, ensuring seamless data management.

This integration ensures that pathfinding functionalities operate on up-to-date and accurate data, providing reliable navigation assistance to users.

2.8.8 Conclusion

The `Path_query.py` module is a pivotal component of the SUTD Mapping Project, seamlessly integrating user interaction, data processing, and pathfinding algorithms to provide a robust navigation tool. Its modular design and reliance on custom processing functions and data structures ensure both flexibility and scalability, laying a solid foundation for future enhancements and feature additions.

2.9 Inter-Module Interactions

The program hinges on the effective interaction between the custom modules:

- `Graph.py` relies on `Json_OS_ProcessingFunctions.py` for loading and saving graph data, ensuring persistent and accurate graph representations.
- `Path_query.py` utilizes both `Graph.py` and `Json_OS_ProcessingFunctions.py` to facilitate user interactions, manage data, and execute pathfinding algorithms based on user inputs.

This inter connectivity ensures that each module can perform its designated functions efficiently while maintaining data consistency and integrity across the entire software ecosystem.

2.10 Strategic Overview

The strategic design of the software modules in the SUTD Mapping Project emphasizes:

- **Modularity**: Each module encapsulates specific functionalities, promoting code reuse, and simplifying maintenance.
- **Scalability**: The architecture supports the expansion of features and the integration of additional pathfinding algorithms without significant restructuring.
- **User-Centric Design**: Interactive methods and intuitive prompts ensure that users can navigate and utilize the mapping tool with ease.
- **Data Integrity**: Comprehensive error handling and consistent data management practices uphold the accuracy and reliability of the mapping and pathfinding functionalities.
- **Extensibility**: The clear delineation of responsibilities among modules allows for the effortless incorporation of future enhancements, such as real-time data adjustments and advanced heuristics.

2.11 Recommendations for Enhancements

To further enhance the robustness and user experience of the SUTD Mapping Project, the following recommendations are proposed:

- **UML Diagrams and Flowcharts**: Develop UML diagrams for each module to visually represent class structures, method interactions, and data flows. These visuals can complement the textual explanations and aid in better understanding the system architecture.
- **Performance Metrics**: Include performance metrics or benchmarks demonstrating the efficiency of pathfinding algorithms or the responsiveness of graph modifications. For instance, report on the time complexity of Dijkstra's algorithm implementation or the scalability of the supermap approach.
- **User Feedback and Testing**: Incorporate results from user testing sessions or feedback surveys that highlight the usability and effectiveness of the mapping and pathfinding functionalities. This empirical evidence can showcase the practical utility and user satisfaction levels.
- **Future Enhancements**: Outline potential future enhancements, such as the implementation of real-time data adjustments based on environmental factors (e.g., density changes during lunch hours), integration with mobile platforms, or the incorporation of machine learning algorithms for predictive pathfinding.
- **Mapping Completion**: Outline potential future enhancements, such as the implementation of real-time data adjustments based on environmental factors (e.g., density changes during lunch hours), integration with mobile platforms, or the incorporation of machine learning algorithms for predictive pathfinding.

By implementing these recommendations, the report will provide a comprehensive and insightful overview of the software modules, their functionalities, and their strategic roles within the SUTD Mapping Project.

2.12 External Modules Used

While we initially wrote a url query module to access telegram where users can interface to use this utility, it would be more effective to make use of the Telegram bot API to write the communication layer between the utility and the telegram bot acting as the frontend.

2.13 Hardware

We have made use of a few Raspberry Pis in the project for different roles. One Pi was used as the host for the telegram bot, supplying the responses after calculating the responses. We have configured systemd on this Pi to keep running the telegram module so long as there is internet access to the Pi.

The rest of the Pis were planned to act as nodes for our sensors to send live data to, before aggregating and cleaning the live data to update the dataset (such as closing open air routes when it is raining, or increase the weights of edges with high traffic flow)

We had planned to fabricate simple devices consisting of an Espressif ESP32 connected to sensors to detect the flow of people, and the presence of rain. Based on physical survey of the site, these two factors would affect[4] the pathing solution the greatest. (to show which points we identified as choke points, to work on writing the communication/transmission of sensor data from the ESP32, to the relay ESP32 to how the pi receives the data). Due to time constraints from the academic terms we were unable to fully write code, fabricate or test this idea.

For the measurement devices, we have made use of Laser Rangefinders to measure the linear distances between nodes, a surveyors wheel for distances not within line of sight, and a lensatic compass to determine the bearing in the direction we are measuring towards. These physical tool allow us to detemrine the weights and metadata of the nodes, giving us the necessary information to calculate the shortest paths, as well as provide context clues such as which direction to turn towards based on the relative difference between angles of nodes.

2.14 Software

We used Python 3 for the backbone of the project, the reason being that Python 3 is taught in the common terms 1 and 3 in SUTD, as well as one of the most popular and known programming languages, making it an accessible language for the project contributors to work on. In addition, Python as a diverse ecosystem of packages, both built in and native which further development can take advantage of in the future. We have kept the utility as native as possible to reduce the reliance on dependencies in any projects working on top of ours.

On the Raspberry Pi 5, it runs on RPI OS, a Debian based operating system for the Raspberry Pi. We have taken advantage of systemd to run our telegram bot service, since it will execute the service on startup, and the service has been configured to constantly restart an attempt to establish a connection and run the service whenever it is down.

[4] Detection of the flow of people will help modify the distance in choke points, simulating the lag time in a crowd, while routes such as the connection point on level 5 between buildings 1 and 2 is not sheltered and un-passable in rain.

3 Results

3.1 CLI tool

We have completed the CLI tool for users to interact with, giving users the ability to add data to the graph, query it, and validate map and supermap information. Below are screen captures of the tool in action, including the interface and output.

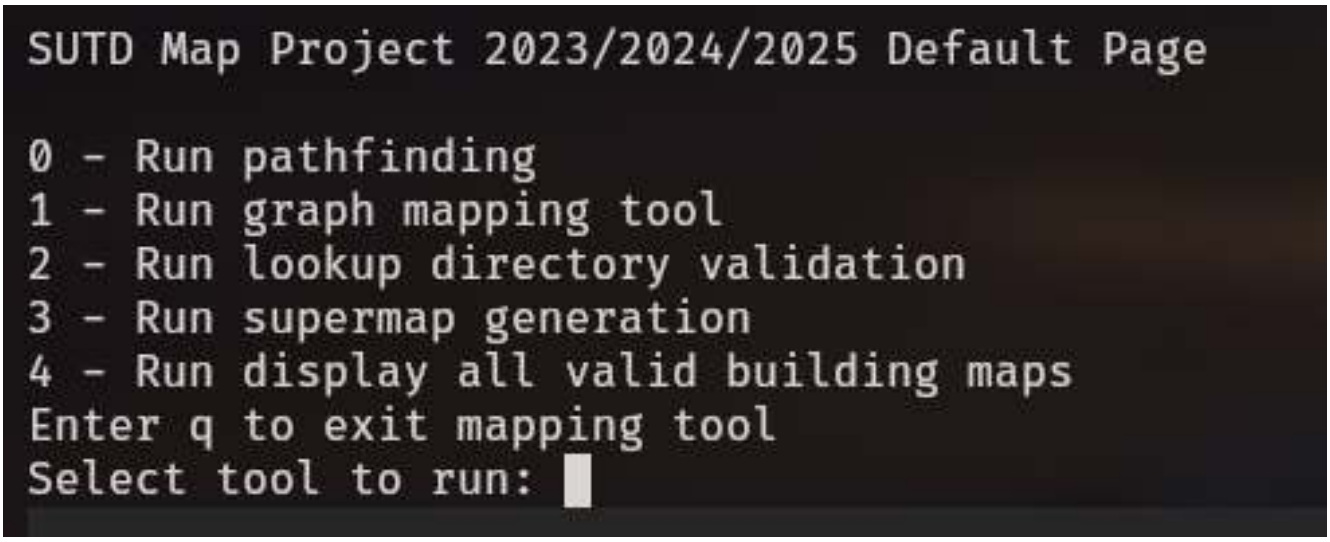

Figure 2: CLI welcome output

Figure 3: Main Menu Option (0) - Running Pathfinding on every vertex

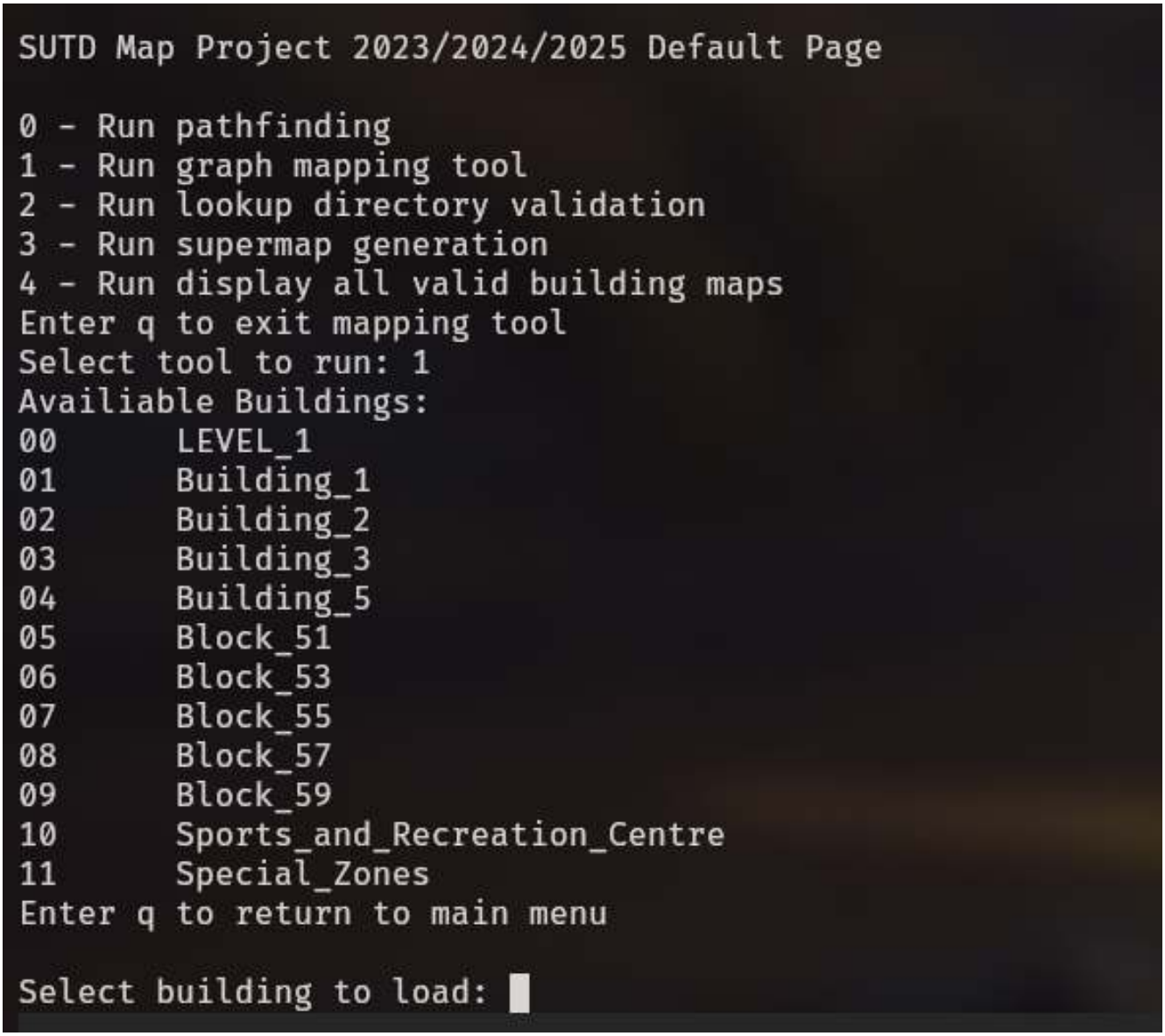

Figure 4: Main Menu Option (1) - Interface to select graph data to modify

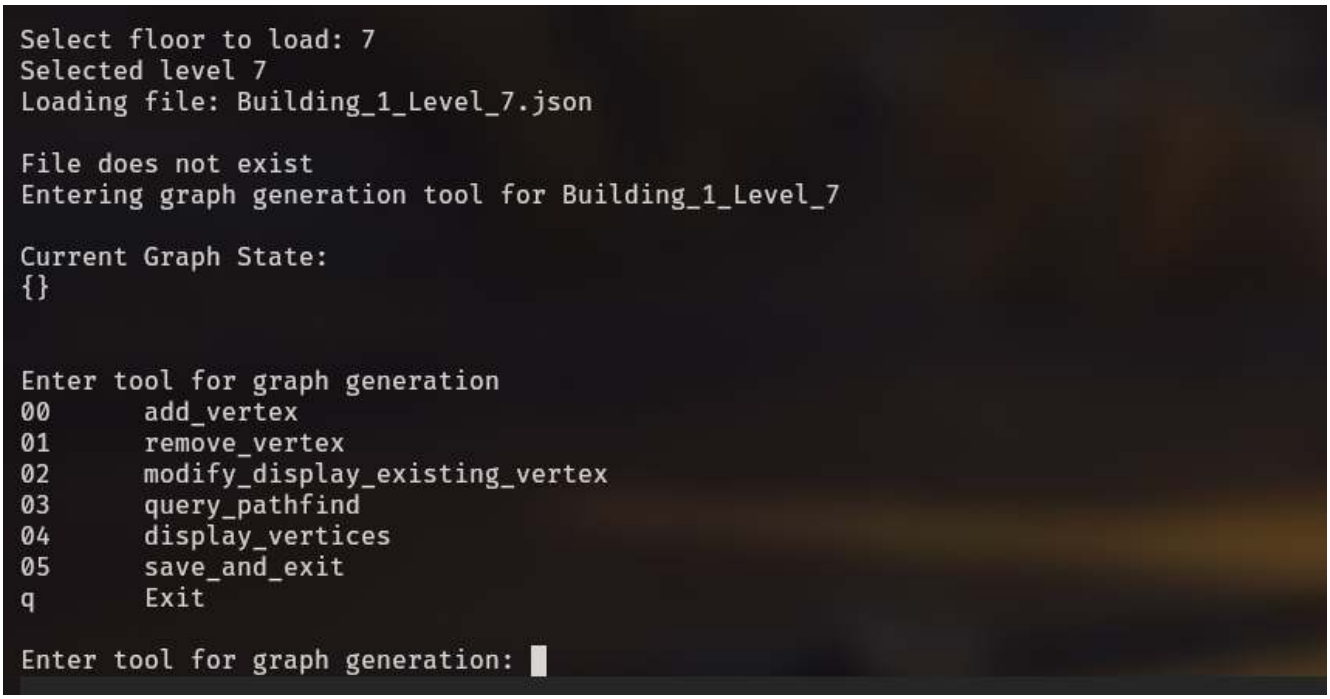

Figure 5: Main Menu Option (1) - Loading of empty graph (unmapped area)

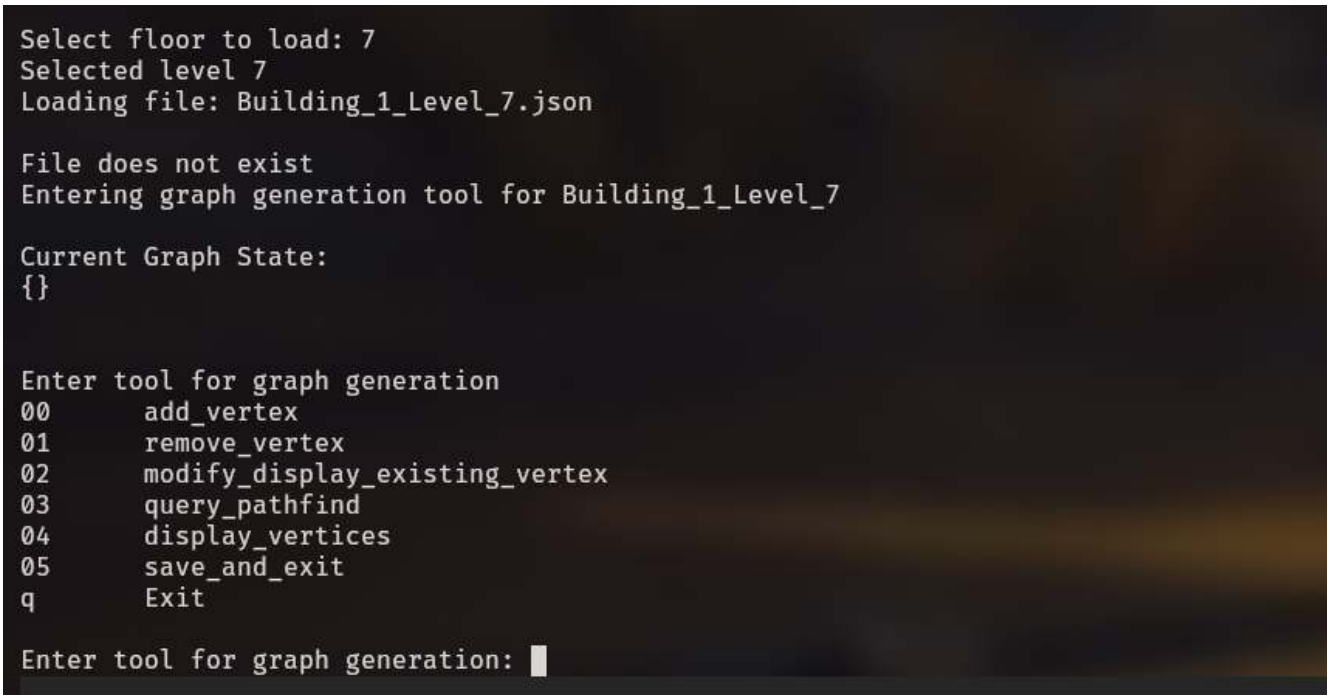

Figure 6: Main Menu Option (1) - Loading of graph instance (previously mapped area)

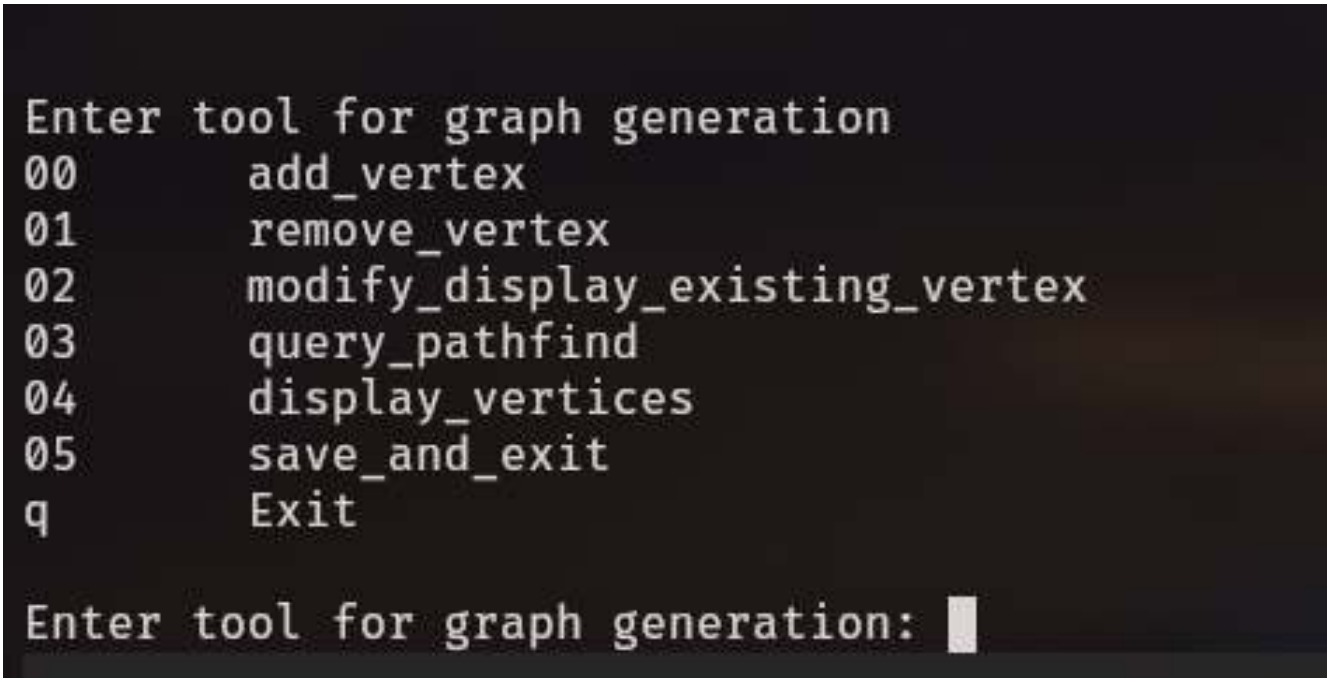

Figure 7: Main Menu Option (1) Graph data - Graph.py modification selection interface

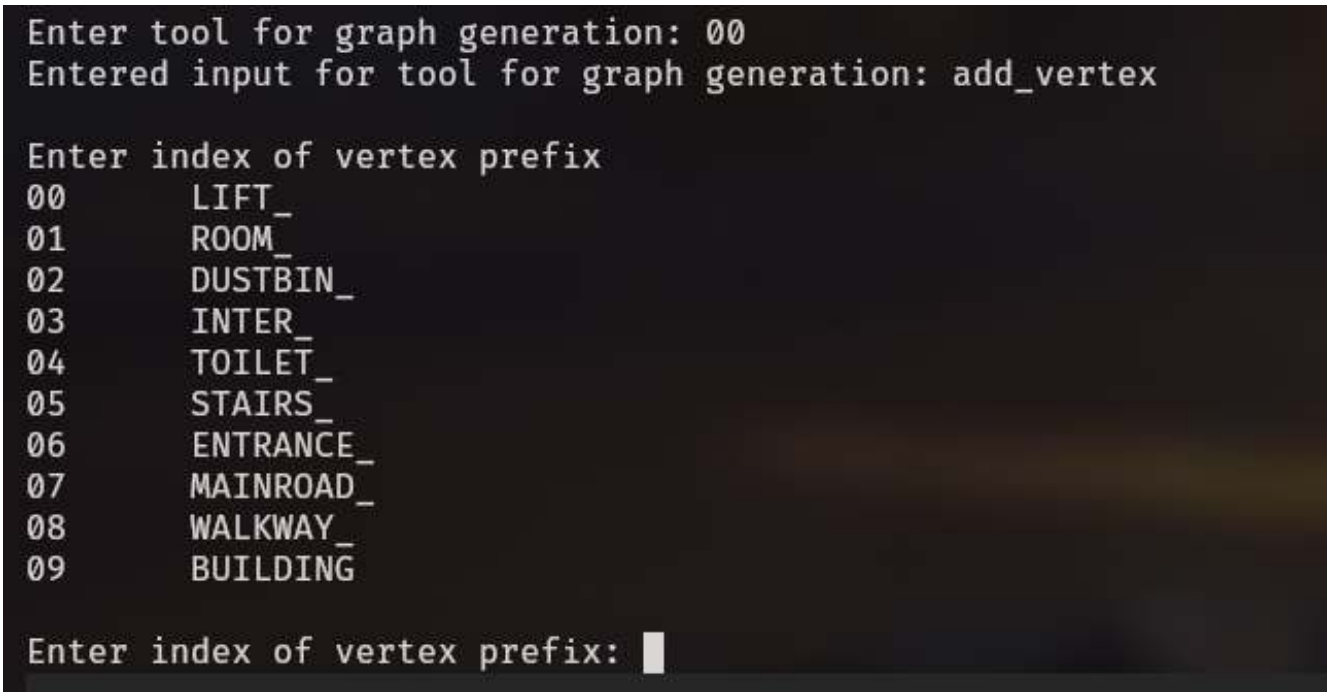

Figure 8: Main Menu Option (1) Graph data - Interface to add new vertex

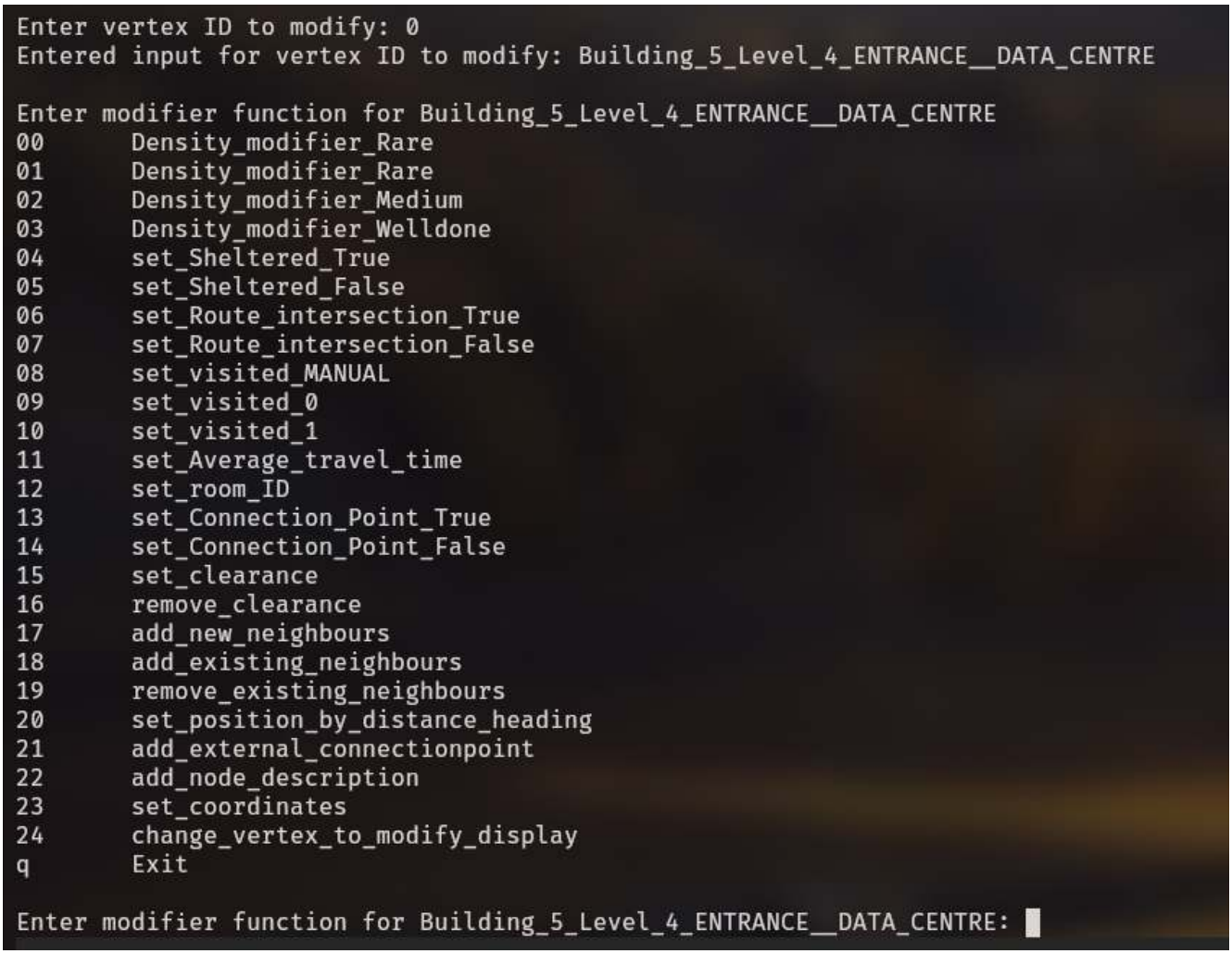

Figure 9: Main Menu Option (1) Graph data - Graph vertex modification selection interface

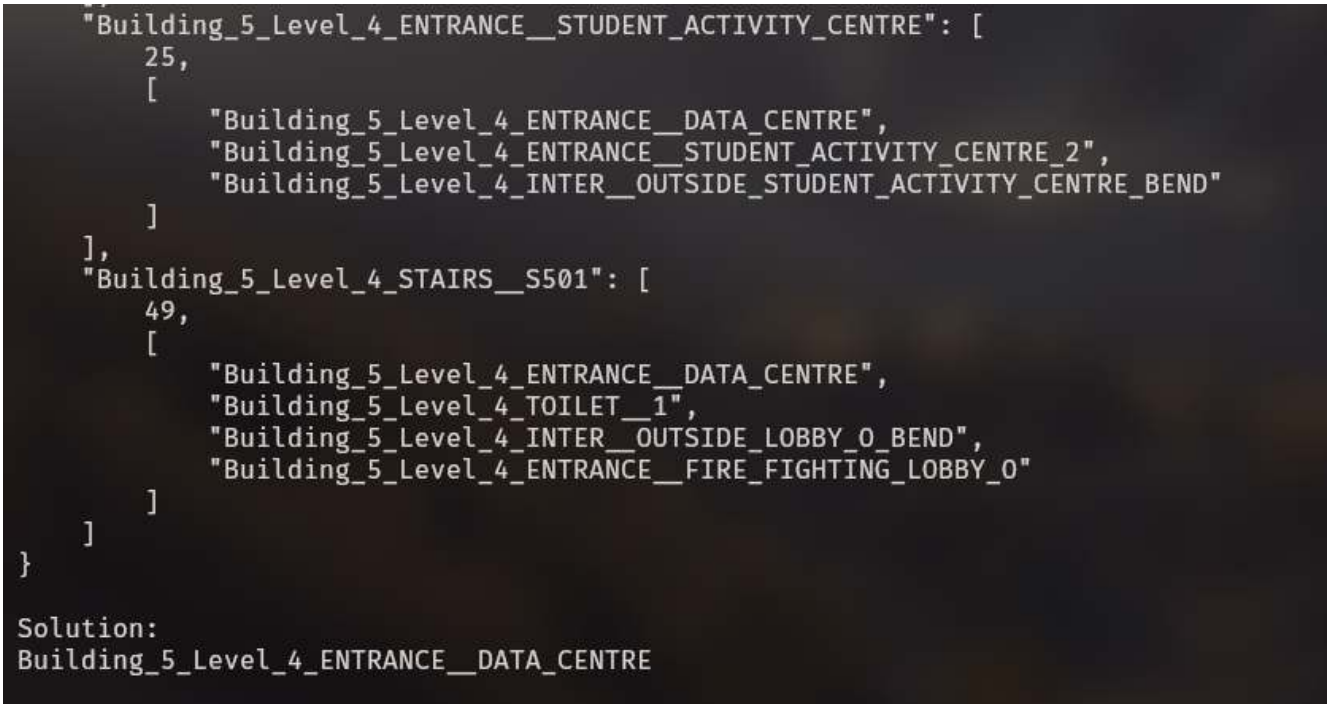

Figure 10: Main Menu Option (1) Graph data - Instance map pathfinding solution output

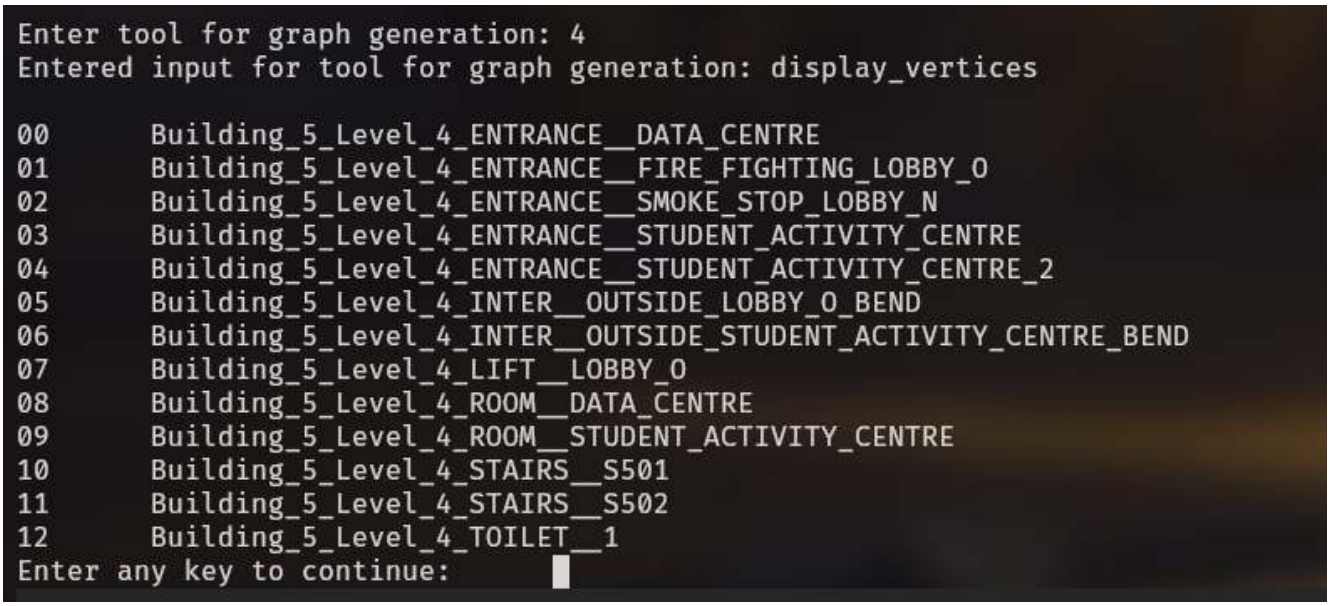

Figure 11: Main Menu Option (1) Graph data - Instance map vertices

Figure 12: Main Menu Option (2) - Lookup data validation - validating compiled vertices against maps

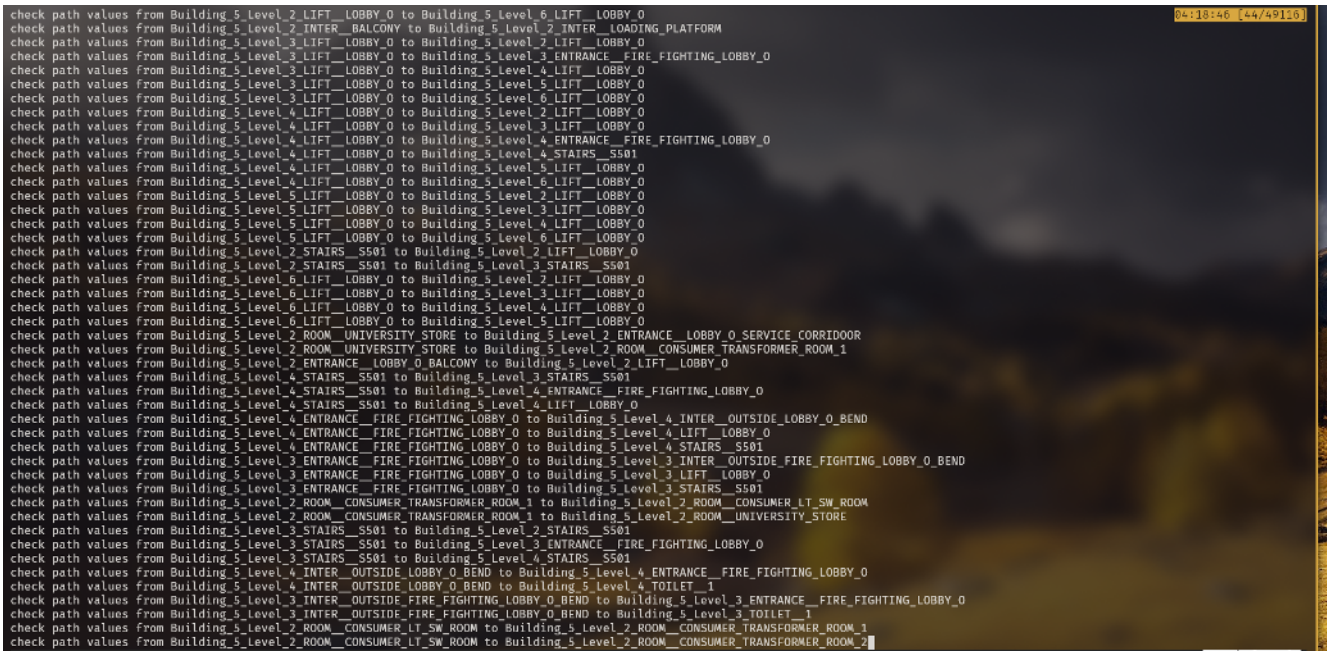

Figure 13: Main Menu Option (3) - Generating the supermap - running Dijkstra's on every vertex

Figure 14: Main Menu Option (4) - Visualisation of mapped vertices through matplotlib

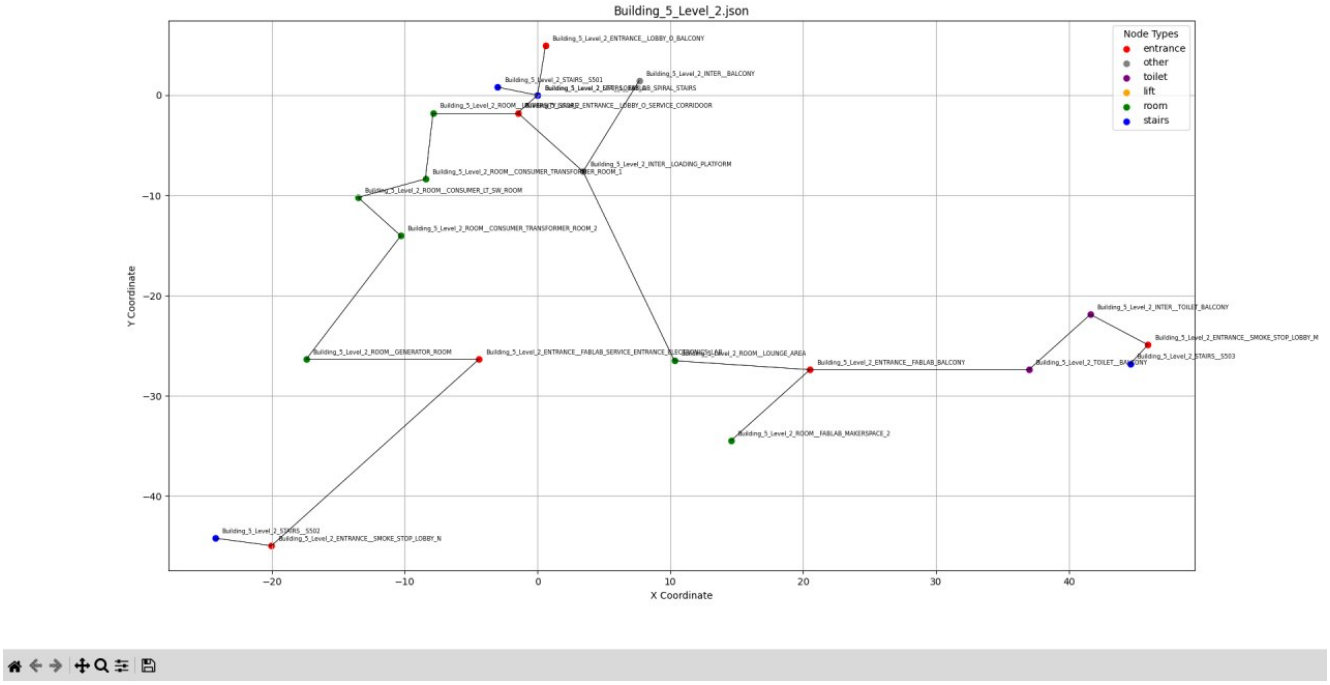

Figure 15: Visualisation of Building 5 Level 2 (Complete)

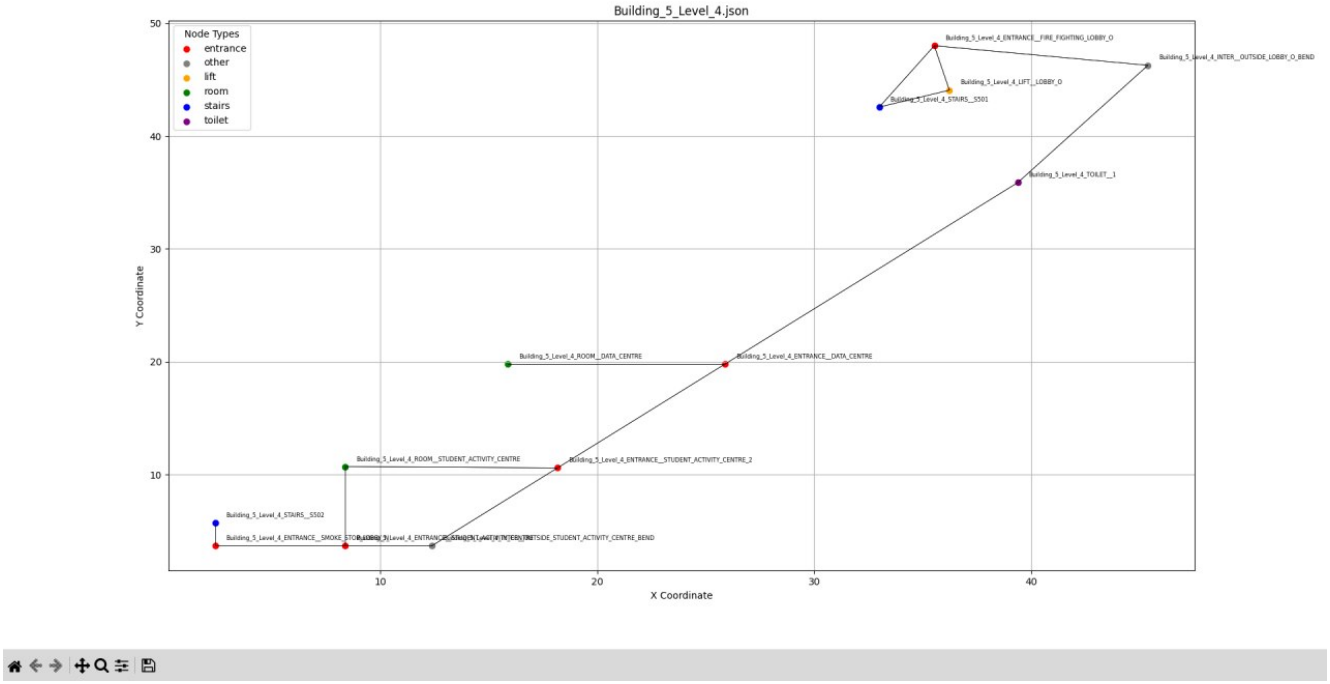

Figure 16: Visualisation of Building 5 Level 4 (Complete)

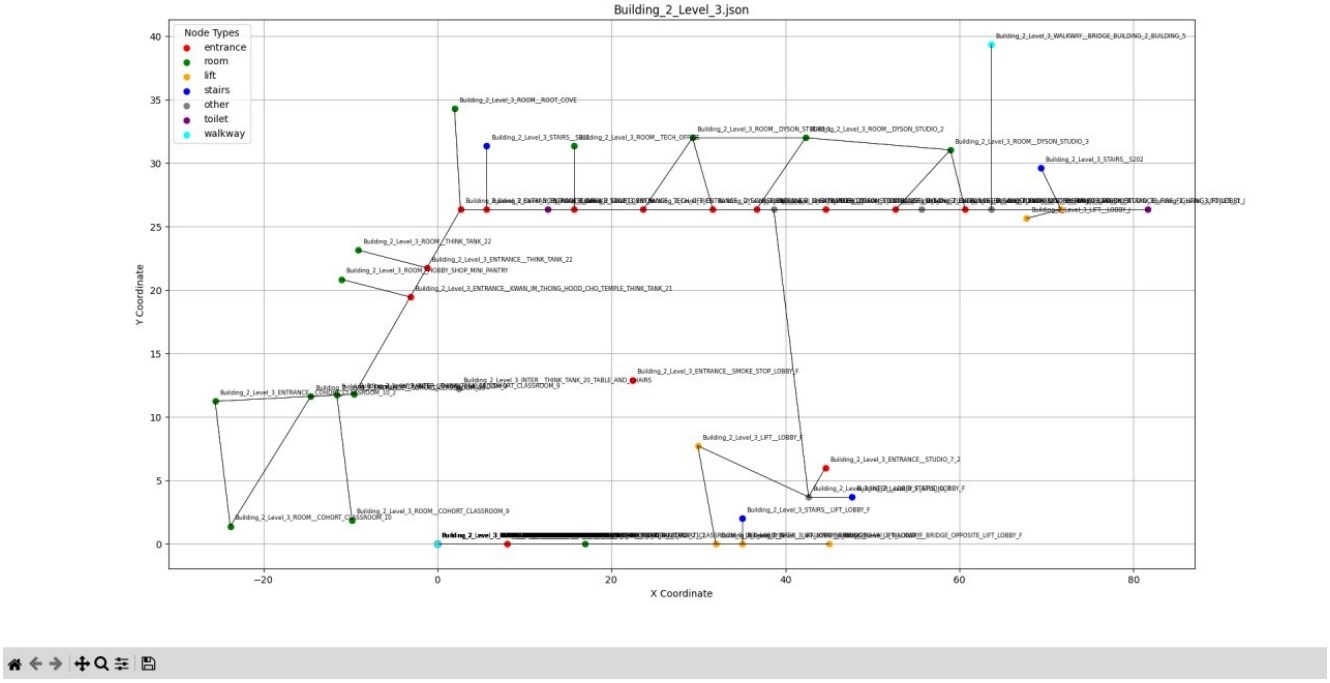

Figure 17: Visualisation of Building 2 Level 3 (Incomplete)

3.2 Visual Validation of Mapping Accuracy

To verify the geometric accuracy of the graph data, the final visualisation of the campus generated by `Graph.py` was rendered and overlaid on top of a satellite photograph of the SUTD campus pulled from Google Maps.

Key qualitative observations:

- **Building alignment** — Vertices were aligned with known landmarks such as the balcony intersection and the FabLab balcony entrance.
- **Scaling** — The image was rotated to compensate from magnetic north to true north, and scaled to fit the satellite image.

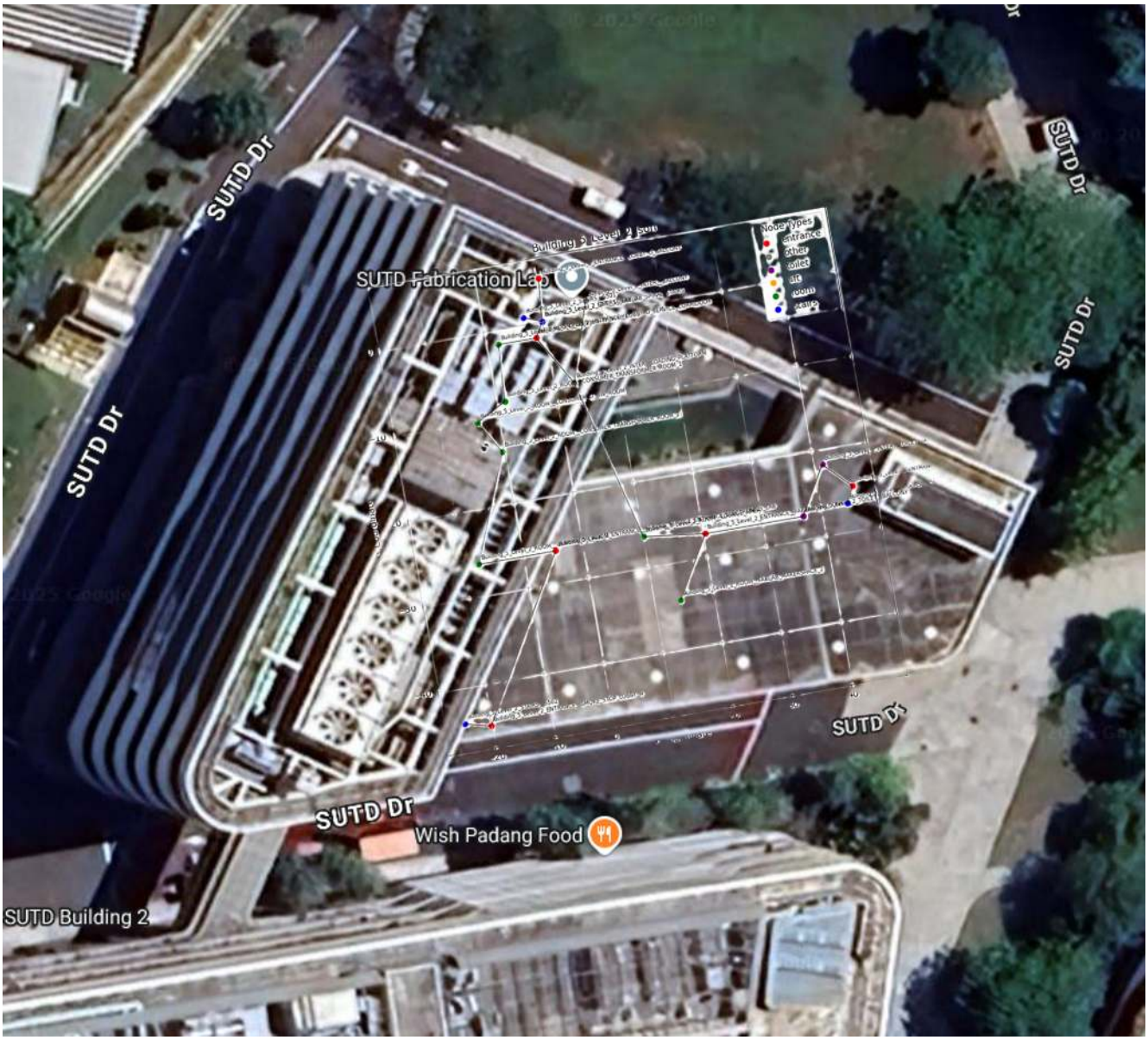

Figure 18: Composite Image

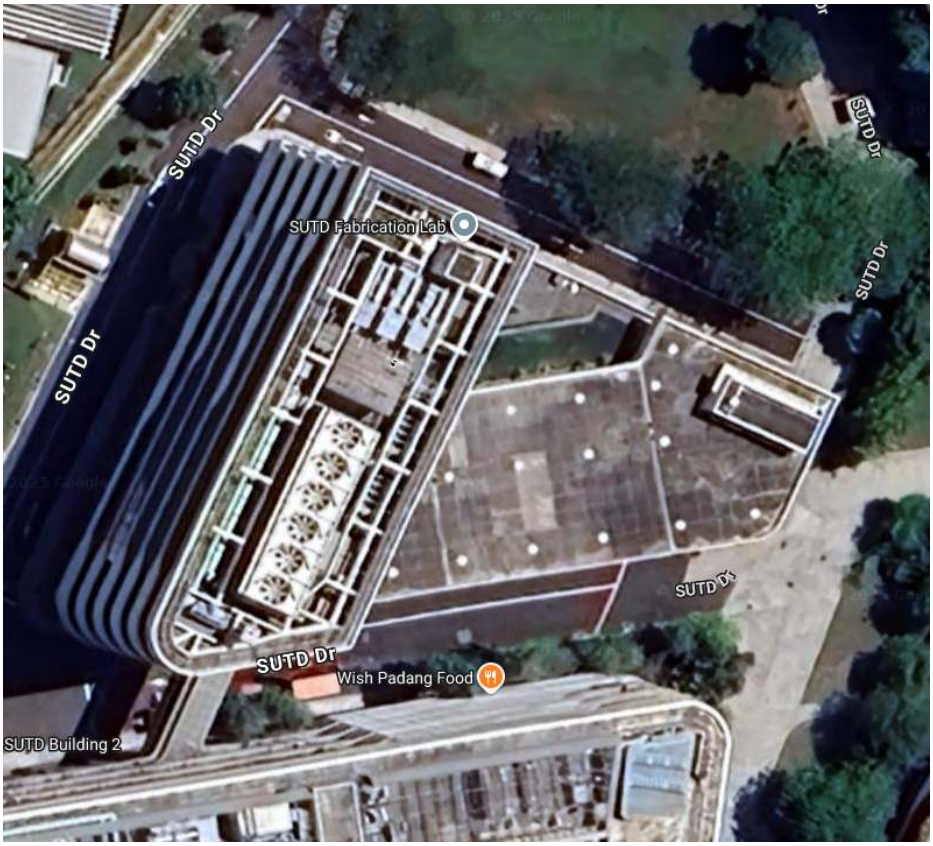

Figure 19: Satellite image of Building 5

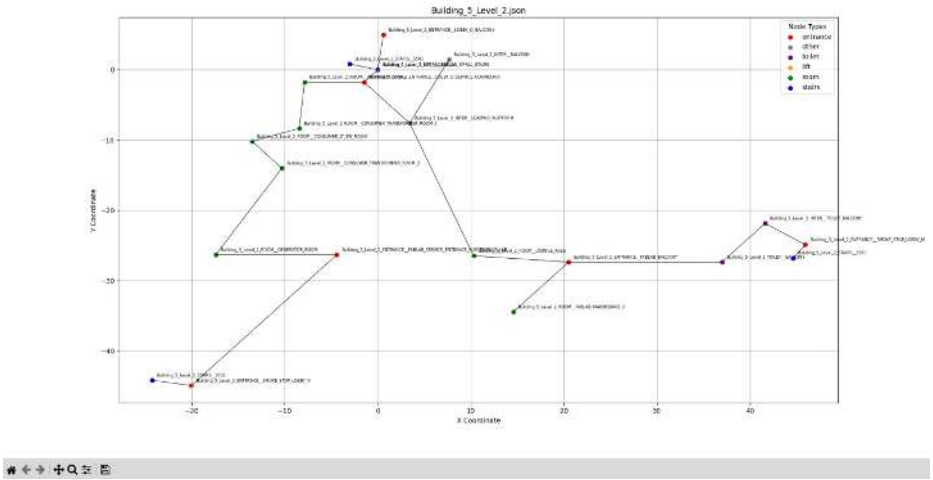

Figure 20: Matplotlib visualisation

4 Conclusion / Learning Points / Reflections

1. **Data Collection Difficulties**. We initially thought that the floor plans and layouts were easily accessible, but encountered difficulties in communicating and getting responses from the Office of Infrastructure. To overcome this, we ended up conducting field work to measure the angles and distance between vertices.

2. **Measurements**. We were initially too strict with the entry distance, truncating the distance into integers with the assumption that the loss in accuracy is compensated with a gain in measuring speed. This conflicted with our distance validation and automatic edge setting functions, which uses trigonometry to derive the distance and bearing of additional intersecting vertices. We changed the entry distance type from integer to float to reduce the discrepancy of the output of our automatic edge weights which also returned a float from the calculation.

3. **Field Work**. When carrying out the data collection, we split the data collection into two stages. The first stage was the manual determination of vertices to map and data entry into the program. This task involved collecting publicly accessible maps from the fire escape plans in the smoke stops, and floor load bearing diagram in the service lift area; followed by manually visiting all accessible locations and marking them out on paper, creating a primitive hand drawn graph we used as reference for data entry. The next stage was the collection of distance and bearings of each connected vertex on to build the set of edges in the graph.

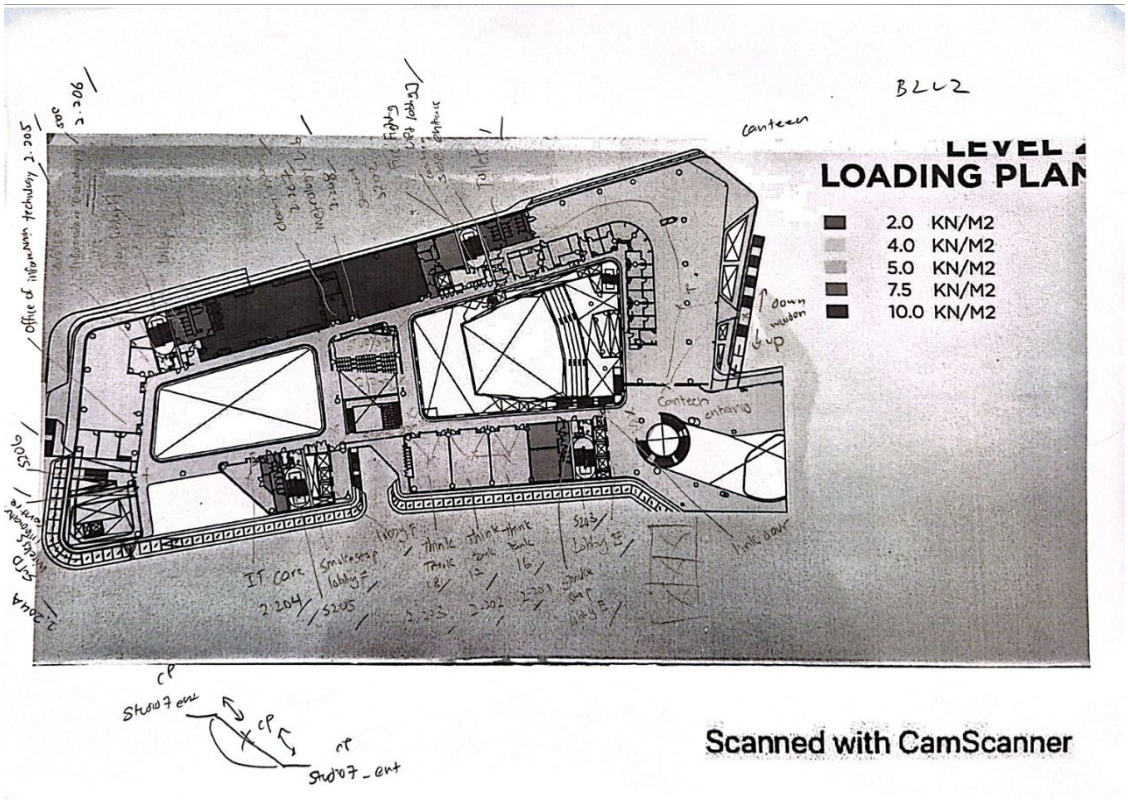

Figure 21: Hand plot of Building 2 Level 2 vertices, their identification and edges

4. **Delays**. For two terms in a row, we encountered a drop in data collection efficiency approaching the second half of each term (between week 6 - 14) due to the ramp up of syllabus content. More time could have been allocated in the beginning of each term to do the data collection instead of attempting to spread it across the term which resulted in incomplete data collection in our dataset.

5. **Scale of the first floor**. Using our model of segregating the map into sections based on buildings and floors, we have unintentionally neglected the entirety of the first floor of the campus which connects all the buildings while also having its own venues. We currently have no good solution for segregating the first floor without keeping the separation of zones intuitive to find or a standardised way to connect partitioned first floor segments. Mapping the first floor as one map comes with its own difficulties in the form of its openness, which introduces a high degree of connectivity and reduction of distinct landmarks that we were unable to reasonably come up with vertices to create a satisfactory graph. Future research can be put into coming up with a process or algorithm to tackle this limitation.

6. **Space usage I**. We attempted a naive speed up in the form of pre-computing all the solutions for every vertex through a union of all the independent maps into one super map and ran Dijkstra's from every vertex and saved that result in `.processed_dijk_supermap.json` outlined from `Path_query.py`. With 5 full graphs and an additional 2 incomplete graphs with no edges, the compiled solution takes 38.8 Mb of space. Running Dijkstra's on each unique vertex against every other vertex, there is an intuition that there will potentially be repeated solutions such as the reciprocal or sub-solutions that are solutions for other vertex pairs. Future research can be put into analysing these patterns and coming up with ways to compact the solutions since storing this naive approach is expensive.

7. **Space usage II**

n	number of vertices ($\|V\|$)
m	number of directed edges / arcs ($\|E\|$)
τ	number of average traversals to reach destination
$\tau(s, d)$	number of traversals to reach destination d from source s
b_{id}	bytes to store a destination–vertex ID
b_{dist}	bytes to store the distance of solution
b_{instance}	bytes to a single calculated solution
b_{file}	bytes to store file on disk

Examining individual solutions An individual solution (as described below as all destination solutions for a single source vertex in the graph) consists of selecting a source vertex, running out modified Dijkstra's to get a distance and a full path list. The average space taken in storing one instance will be

$$b_{\text{instance}} = b_{\text{dist}} + \tau \times b_{\text{id}}$$

where the specific space can be derived if the length of path is known.

```
{
    "Source Vertex": {
        "First Destination Vertex": [
            Distance,
            List of path in order
        ],
        "Next Destination Vertex": [
            Distance,
            List of path in order
        ],
        ...,
        "Last Destination Vertex": [
            Distance,
            List of path in order
        ]
    },
    "Next Source Vertex": {
```

```
        ...
        },
        ...
        "Last Source Vertex": {
        ...
        }
    }
```

Examining storing all solutions As mentioned above, storing a single instance will cost $b_{\text{instance}} = b_{\text{dist}} + \tau \times b_{\text{id}}$. Since we are evaluating n *solutions* for n *vertices*, the total entries in `.processed_dijk_supermap.json` will be n^2, resulting in a specific total size of:

$$b_{\text{file}} = b_{\text{dist}} \times n^2 + b_{\text{id}} \times \sum_{s=1}^{n} \sum_{d=1}^{n} \tau(s, d)$$

Lower Bound Case The lower bound of storing all solutions (and consequently the smallest file size required) occurs when Graph G is a complete graph which results in any $\tau(s, d) = 1$ since any source vertex is connected to every destination vertex and can be visualised as a complete symmetric digraph. We are able to reduce the total space required down to:

$$b_{\text{file}} = b_{\text{dist}} \times n^2 + b_{\text{id}} \times \sum_{s=1}^{n} \sum_{d=1}^{n} (\tau(s, d) = 1) = n^2 \times (b_{\text{dist}} + b_{\text{id}}) = \Theta(n^2)$$

We opted to use Θ instead of Ω here as there is no graph that contains a path with a shorter shortest path than a complete graph.

Upper Bound Case The upper bound of storing all solutions (and consequently the largest file size required) occurs when every combination of source s and destination d in Graph G are connected by a long shortest path. An extreme case of longest possible shortest path is the Hamiltonian path problem that has a depth of n-1. Using Theorem 2.3 from *Distance in graphs, Czechoslovak Mathematical Journal, Vol. 26 (1976)* [3], $n(n-1) \leq d(p) \leq \frac{n^3+5n-6}{6}$, providing an upper bound in the form of $\dfrac{n^3-n}{3}$ corrected using $\dfrac{n^3-n}{6}/\binom{n}{2}$ to account for the directed nature of the edges/arcs in our graph, giving us a cubic upper bound as shown:

$$b_{\text{file}} = b_{\text{dist}} \times n^2 + b_{\text{id}} \times \tfrac{n^3-n}{3} = \Theta(n^3)$$

Average Case On average the number of traversals required to reach destination d from source s is given by:

$$\tau = \tfrac{1}{n^2} \sum_{s=1}^{n} \sum_{d=1}^{n} \tau(s, d)$$

Rewriting this, we get:

$$n^2\tau = \sum_{s=1}^{n} \sum_{d=1}^{n} \tau(s, d)$$

Which lets us simplify the specific total size into an average total size of:

$$b_{\text{file}} = b_{\text{dist}} \times n^2 + b_{\text{id}} \times \sum_{s=1}^{n} \sum_{d=1}^{n} \tau(s, d) \implies b_{\text{file}} = n^2(b_{\text{dist}} + b_{\text{id}} \times \tau)$$

This average total size fits between the lower and upper bound determined above, but due to the way data was collected tends towards $O(n^3)$ as it is physically not possible to fulfill the conditions to be a complete graph — any source vertex is unable to provide a direct path to every destination vertex in the graph without first colliding with a nearer vertex.

$$\Omega(n^2) \leq \Theta(b_{\text{file}}) \leq O(n^3)$$

5 Acknowledgments

I have to admit that the usage of edges and arcs in the report is rather loose as the program treats each edge as a directed edge or arc despite each arc having a distinct magnetic bearing. Treating the graph as an undirected graph allowed me to more easily visualise the shape of the graph in my head when approaching the analysis.

I would like to acknowledge the use of the University Library, Google and ChatGPT o3 model in searching for resources and providing explanations for me to substantiate the analysis and evaluations made in the report.

I would also like to acknowledge my friends who helped me along the way in data collection, debugging, and especially the time spent on data collection.

Finally I would like to thank Prof Simon for his patience and effort in supervising the project.

Name	**Role**
Dr Simon Perrault	Supervisor, Advisor
Ryan Pek Yi Ze	Report, Software Development, Software Testing, Data Collection
Lee Cheng Yong	Software Development, Software Testing, Data Collection
Pathomphu Kanapornthada	Software Testing, Data Collection
Ming Liang Koh	Data Collection
Mark Giam	Data Collection
Amith Reddy	Data Collection
Ng Qun Yu	Data Collection
Samuel Lee	Data Collection

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 4 edition, 2022. Fourth Edition (CLRS v4).

[2] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[3] Roger C. Entringer, Douglas E. Jackson, and Daniel Snyder. Distance in graphs. *Czechoslovak Mathematical Journal*, 26(2):444–452, 1976.