FINAL REPORT FOR THE COURSE COMP1110 – GROUP D08

CHUN HEI BANH, SZE POK LIU, ZHAN HO JACOB SHING, YUBING YE, AND YIKUN ZHANG

Contents

1. Problem Definition and Modelling

1.1. **Problem Definition.** The University of Hong Kong (HKU) campus occupies a steeply terraced hill in which buildings span multiple levels and are interconnected by a network of outdoor paths, staircases, escalators, lifts, and covered walkways. For students and staff navigating this environment daily, identifying the most suitable route between two locations is non-trivial. A student rushing from the MTR station to a lecture theatre before a class begins, for example, must reason about path lengths, vertical changes in elevation, and their own walking pace simultaneously. Similarly, a person with a mobility impairment must identify routes that are entirely free of stairs and escalators — a constraint that standard general-purpose mapping applications do not reliably enforce at the granularity of individual campus segments.

The significance of the problem is beyond mere convenience. Arriving late to examinations or laboratory sessions has direct academic consequences, and the inability to find accessible routes may exclude users from participating fully in campus life. Despite this, no dedicated pedestrian navigation tool targeting HKU's specific topology exists. General-purpose solutions such as Google Maps operate at a coarser spatial granularity and do not model campus-internal infrastructure such as covered linkways or building-internal lifts as distinct, traversable segments.

C. Banh , BEng(Elite/AI&DataSc), 3036570233, kbch@connect.hku.hk
S. Liu , BEng(AI&DataSc), 3036589454, u3658945@connect.hku.hk
J. Shing , BEng(CompSc), 3036228892, jacobszh@connect.hku.hk
Y. Ye , BEng(CompSc), 3036291502, u3629150@connect.hku.hk
Y. Zhang , BEng(CompSc), 3036482333, pzdmmsd@connect.hku.hk
Date: 8 April 2026.

Several challenges compound the difficulty of the problem. First, the path network is heterogeneous: segments differ not only in length but in *type* (flat walkway, staircase, escalator, lift), and each type interacts differently with a pedestrian's speed and physical capability. Second, user requirements are multi-dimensional: a single query may simultaneously impose hard exclusion constraints ("no stairs"), soft preferences ("prefer lifts"), and a personalised speed parameter. Third, between any pair of adjacent locations there may exist multiple parallel segments of different types, forming a *multigraph* structure that complicates standard shortest-path reasoning. Finally, routes may pass through one or more mandatory intermediate waypoints, requiring the navigation problem to be solved as a sequence of coupled sub-problems.

1.2. **Problem Modelling.**

1.2.1. *Graph Representation.* The campus is modelled as a finite, undirected multigraph

$$G = (V, E), \tag{1}$$

where the vertex set V represents discrete locations of interest on campus (building entrances, floor landings, transit exits, etc.), and the edge multiset E represents traversable segments connecting pairs of locations. The multigraph formulation is necessary because two vertices may be connected by more than one edge of distinct types — for instance, two adjacent floors of a building may be linked by both a staircase and a lift, each forming a separate edge.

Each edge $e \in E$ carries two attributes: a type $\tau(e) \in \mathcal{T}$ and a base traversal time $c(e) \in \mathbb{Z}_{>0}$ measured in seconds. The type set is

$$\mathcal{T} = \{\, \textsc{Flat},\ \textsc{Stairs},\ \textsc{MinorStairs},\ \textsc{Escalator},\ \textsc{Lift} \,\}, \tag{2}$$

reflecting the physical modes of traversal present on campus, where MinorStairs denotes flights of at most 7 steps (including the top landing but excluding the ground-level step), and other types are self-explanatory.

1.2.2. *Speed-Adjusted Edge Costs.* Pedestrian speed varies between individuals and circumstances. To model this, a scalar *speed multiplier* $\sigma > 0$ is supplied by the user, where $\sigma = 1$ corresponds to an average walking pace, $\sigma > 1$ to a faster pace, and $\sigma < 1$ to a slower one.

The degree to which speed affects traversal time depends on the edge type. Mechanical conveyances such as lifts operate at a fixed speed independent of the passenger, while climbing a staircase is far more sensitive to physical exertion than walking on flat ground. This is captured by a type-dependent *speed exponent* $\alpha : \mathcal{T} \to \mathbb{R}_{\geq 0}$, defined as

$$\alpha(\tau) = \begin{cases} 1.0 & \tau = \textsc{Flat}, \\ 1.3 & \tau = \textsc{MinorStairs}, \\ 1.5 & \tau = \textsc{Stairs}, \\ 0.3 & \tau = \textsc{Escalator}, \\ 0.0 & \tau = \textsc{Lift}. \end{cases} \tag{3}$$

The *adjusted cost* of traversing an edge e at speed multiplier σ is then

$$\hat{c}(e, \sigma) = \frac{c(e)}{\sigma^{\alpha(\tau(e))}}. \tag{4}$$

For $\alpha(\tau) = 0$ (lifts), the adjusted cost equals the base cost regardless of σ. For $\alpha(\tau) = 1.5$ (stairs), the adjusted cost decreases more steeply as σ increases, reflecting the disproportionate benefit of a faster pace on elevation gain.

1.2.3. *Preference-Weighted Cost Function.* Users may specify a preference profile P, which partitions $\mathcal{T}$ into three disjoint subsets: $\mathcal{T}_{\text{avoid}}(P)$ (hard exclusions), $\mathcal{T}_{\text{prefer}}(P)$ (soft preferences), and $\mathcal{T}_{\text{neutral}}(P)$ (no preference). A fixed *preference discount factor* $\beta \in (0, 1)$ rewards the use of preferred segment types by reducing their effective cost. The pathfinding cost function $w : E \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is defined as

$$w(e, \sigma, P) = \begin{cases} \infty & \tau(e) \in \mathcal{T}_{\text{avoid}}(P), \\ \beta \cdot \hat{c}(e, \sigma) & \tau(e) \in \mathcal{T}_{\text{prefer}}(P), \\ \hat{c}(e, \sigma) & \text{otherwise}. \end{cases} \tag{5}$$

Edges with $w = \infty$ are effectively removed from the graph during the search, enforcing hard exclusion constraints absolutely.

1.2.4. *Route and Multi-Waypoint Formulation.* A *route* is a walk $\pi = (v_0, e_1, v_1, \ldots, e_k, v_k)$ in G. Its total weighted cost is

$$W(\pi) = \sum_{i=1}^{k} w(e_i, \sigma, P), \tag{6}$$

and its total displayed travel time is computed using the adjusted costs $\hat{c}$ (without the preference discount), so that the displayed time reflects true traversal duration rather than the artificial cost used to bias routing.

When the user specifies $m \geq 1$ intermediate waypoints in addition to origin and destination, the full journey is decomposed into $m + 1$ independent sub-problems. Given an ordered waypoint sequence $(w_0, w_1, \ldots, w_{m+1})$, the optimal route for each segment $[w_i, w_{i+1}]$ is found independently, and the full route is their concatenation.

1.2.5. *Multi-Candidate Route Generation.* Rather than returning a single optimal route, the system generates a small set of candidate routes by solving the shortest-path problem under several distinct preference profiles derived from the user's input (for example, one enforcing stair avoidance and another promoting lift usage). Duplicate routes, identified by equality of their ordered edge sequences, are discarded. The surviving candidates are then ranked and labelled according to secondary criteria such as total travel time and number of stops.

2. Surveying of Existing Solutions

As planned, six existing solutions for navigation, grouped into three categories, were surveyed and evaluated on different aspects. The solutions are first compared in an overall manner in Table 1, and then compared in a more detailed manner with focus on specific features in Table 2.

The three categories of solutions surveyed are: (1) general-purpose navigation applications, represented by Google Maps and Citymapper; (2) transit-specific applications, represented by MTR Mobile and HKBUS.app; and (3) static campus reference tools, represented by the HKU campus map and on-campus signage. The general-purpose navigation applications provide dynamic estimated times of arrival, multi-modal transfer handling, and city-scale route planning, but both perform poorly in multi-floor environments such as the HKU campus and operate exclusively online. The transit-specific applications are limited in scope to their respective transport modes: MTR Mobile supports only station-to-station queries within the MTR and light rail network, while HKBUS.app restricts users to at most one bus transfer and cannot locate the user or direct them to the nearest bus stop. Both transit apps also require a live network connection. The static campus reference tools — the HKU campus map and on-campus signage — provide a visual overview of building locations and nearby paths, but are non-interactive, offer no route direction or estimated arrival time, and are inherently unable to adapt to a user's starting point or destination.

The surveying exercise informed several concrete design decisions for this project. The station-based topology of transit apps, in which discrete named stops are connected by typed transport links, was adopted as the foundational data model, resulting in the node-and-edge multigraph representation described in subsection 1.2. The accessibility mode offered by Google Maps — which avoids stairs and promotes lift usage — motivated the *Accessible Route* preference toggle and the hard-exclusion mechanism in the cost function. The route suggestion display of Citymapper, which presents fare and expected time for each option in a compact list, informed the design of the route card display, where each candidate route is accompanied by a time estimate, stop count, and a salient label indicating its primary characteristic. Finally, the dependence of all surveyed online solutions on live network connectivity, contrasted with the offline availability of the HKU campus map PDF, motivated the decision to bundle a static copy of the campus dataset within the application, ensuring full functionality in offline or restricted-network environments.

3. Core Implementation

3.1. **System Architecture Overview.** The project is constructed with a layered and modular architecture, as illustrated in Figure 1. The project enforces principles of object-oriented programming throughout its design, with some utility functions implemented in a functional style to avoid boilerplate code and improve readability. The design allows a clear separation of data representation, data parsing, route-finding business logic, and state management of the user interface, which facilitates maintainability of the codebase.

Table 1: Overall comparison of the existing solutions.

Aspect / Solution	Target Users	Key Features	Missing Features	Assumptions
Google Maps	Any users, especially users who are trying to find a location that is on the street.	Provide a real-time instruction from the current location to the destination.	Poor performance in multiplicity floors, such as the HKU campus or a mall.	None.
Citymapper	Users who prefer to direct them from one place to another.	Provide suggestions displayed with fare and expected time in a list.	Poor translation due to the software being originally in English.	None.
MTR Mobile	Users who want to travel by MTR.	Provide the estimated time of the train and its start/end service times.	Only station-to-station, but not available to handle arbitrary locations.	Assume the user only takes MTR and travels from station to station.
HKBUS.app	Users who want to take a bus or a minibus.	Provide the estimated waiting time for the next bus.	Unable to locate where you are and direct you to the bus station.	Assume users only take the bus instead of other modes of transportation, such as the MTR.
HKU campus map	Users who may be interested in HKU or will go to HKU later.	Provide a map for building on campus with names labeled.	Static map and unable to show 3D structure.	None.
On-campus signage	Users on campus who are finding a building.	Show where the signs are and the nearby path and building on campus.	Static map, unable to direct the user to where they want.	Assume users will only travel within campus.

Table 2: Comparison of existing solutions on specific aspects.

Solution \ Aspect	Estimated Time of Arrival (ETA)	Fair Display	Transfer Handling	Accessibility Info	Offline Availability
Google Maps	Provided ETA with multiple methods, calculated dynamically with the current situation.	No fare display.	Able to handle any transfer, even if some distance needs to be walked.	Provided accessibility mode to avoid stairs and prioritize the lift.	No route planning function when offline.
Citymapper	Provided ETA with multiple methods.	Provided adult fares for multiple methods. Taxi fare is estimated by driving time.	Able to handle any transfer, even if some distance needs to be walked.	No accessibility info.	Unable to use it offline.
MTR Mobile	Provided ETA with multiple methods, estimated by maximum waiting time.	Adult fare is displayed, while others can be checked.	Able to handle any transfer, only within MTR stations or light rail.	Able to display accessibility facilities for multiple kinds of accessibility needs, such as those of the visually impaired.	No route planning function when offline.
HKBUS.app	ETA must be checked by checking different paths one by one.	Adult fare is displayed separately for different paths, and you need to sum them up to get a total.	At most one transfer is allowed. Able to handle at most 500m walking distance.	No accessibility info.	Unable to use it offline.
HKU campus map	No ETA provided.	No fare display.	No transfer handling.	No accessibility info.	The web version is unavailable offline, but the PDF version can be used offline.
On-campus signage	No ETA provided.	No fare display.	No transfer handling.	No accessibility info.	Not applicable.

As for the technical stack, the project is implemented in Python 3.13, with its terminal user interface built using the `Textual`[?textual] library.

FIGURE 1. System architecture diagram.

3.2. **Data Model.** The entities required by Equation 1 are represented by three classes: `Node` (v), `Edge` (e), and `CampusMap` (G). The core attributes and their relationships are illustrated in Figure 2. Each class comes with a set of methods for constructing instances, accessing attributes, and performing relevant operations. These classes together represent a undirected multigraph, which allows multiple edges of different edge types between the same pair of nodes.

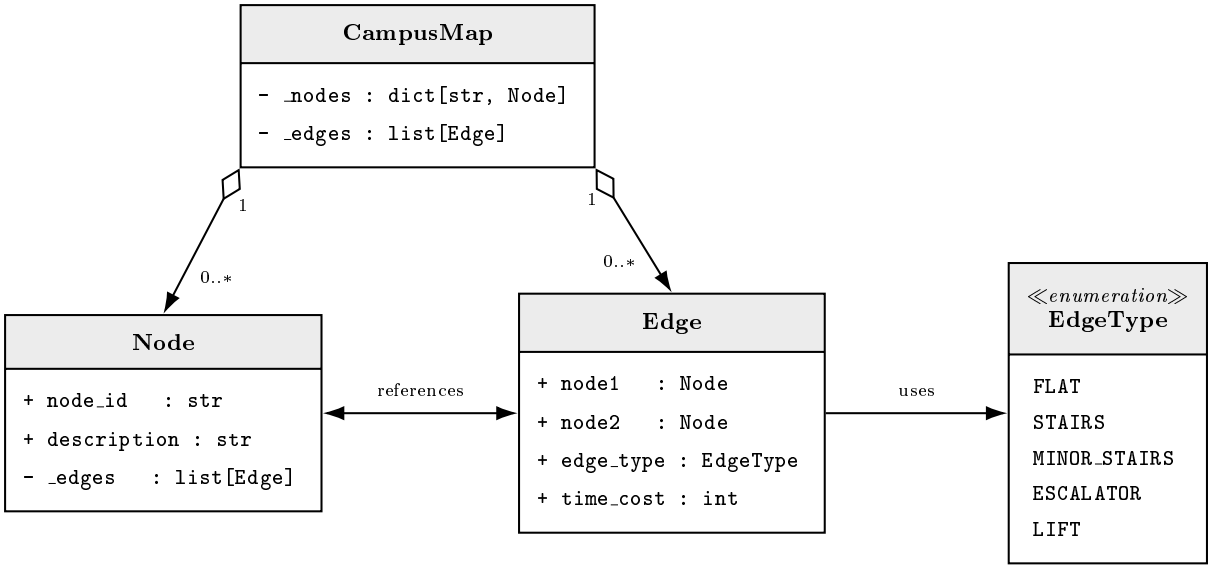

FIGURE 2. Class diagram of the data model.

3.3. **Data Collection, Representation, and Validation.** For implementing this project, the group has divided the campus into 4 zones for distributing labour. Each zone is assigned to a member, who is responsible for defining the nodes within the zone. After finalising the nodes, the member will then walk between the nodes and measure the time taken to traverse each edge (measured in seconds) and record the data on a shared Google Sheet document. One of the members is responsible for measuring the time taken between nodes that fall in different zones, and the data is recorded in the same Google Sheet document. To ensure consistency and reliability of the data, the group has agreed on a standardised method for measuring traversal times and an acceptable range of walking paces.

The data on Google Sheets is fetched automatically by a GitHub Action workflow daily. The workflow calls `fetch_csv_from_google_sheet()` defined in `campus_nav/utils.py`. This function converts the spreadsheet data into a CSV format. The two CSV files are `campus_nav/data/nodes.csv` and `campus_nav/data/edges.cs` Their schemas are defined in Listing 1 and Listing 2, respectively.

In particular, to make the nodes and edges be more consistent, they are defined in a way that all of the entries can be deterministically sorted. Nodes are sorted by their `node_id` in lexicographical order. For edges, in the normalisation process, the two incident nodes are compared, such that `node1` is always lexicographically sorted before `node2`. Then, the edges are sorted by the tuple (`node1`, `node2`, `edge_type`, `time_cost`) in lexicographical/numerical order. The serialisation codes enforce these sorting rules.

LISTING 1. CSV Schema for `nodes.csv`

```
version 1.1
@totalColumns 2

"node_id":          notEmpty unique
"node_description":
```

LISTING 2. CSV Schema for `edges.csv`

```
version 1.1
@totalColumns 4

"node_1_id":  notEmpty
"node_2_id":  notEmpty
"edge_type":  notEmpty is("FLAT", "STAIRS", "MINOR_STAIRS", "ESCALATOR", "LIFT")
"time_cost":  notEmpty positiveInteger
```

3.4. **Route Engine.** The route engine is the core of the navigation system. Given a campus map, an ordered list of waypoints, a set of user preferences, and a speed multiplier, it is responsible for computing a set of candidate routes for the user to choose from. The engine is implemented entirely in `campus_nav/main_app/route_engine.py` as a collection of pure functions, with no dependency on any UI component.

3.4.1. *Representing User Preferences.* User preferences are encapsulated in the `Preferences` dataclass, whose fields correspond directly to the options exposed in the configuration screen. Each field is a boolean flag, grouped into three pairs: avoid/prefer stairs, avoid/prefer escalators, and avoid/prefer lifts. A separate `accessible` flag acts as a master toggle that implicitly enables stair and escalator avoidance and lift preference.

3.4.2. *Computing Edge Costs.* The cost assigned to an edge during pathfinding implements the model described in the Problem Modelling section. First, the base traversal time $c(e)$ is adjusted for the user's walking speed using Equation 4, where σ is the speed multiplier and $\alpha(\tau)$ is the type-dependent speed exponent defined in Equation 3. This is implemented directly in `compute_adjusted_cost()` (Listing 3).

LISTING 3. Speed-adjusted edge cost computation

```
_SPEED_EXPONENTS = {
    EdgeType.FLAT: 1.0,
    EdgeType.STAIRS: 1.5,
    EdgeType.MINOR_STAIRS: 1.3,
    EdgeType.ESCALATOR: 0.3,
    EdgeType.LIFT: 0.0,
}

def compute_adjusted_cost(edge, speed_multiplier):
    exponent = _SPEED_EXPONENTS[edge.edge_type]
    return edge.time_cost / (speed_multiplier ** exponent)
```

The preference-weighted cost function $w(e, \sigma, P)$ from Equation 5 is then applied on top. Hard avoidance constraints are enforced by returning infinity for the forbidden edge types, which effectively removes them from consideration. Soft preferences reduce the cost of favoured edge types by a factor of $\beta = 0.5$, making them appear cheaper to the pathfinding algorithm and therefore more likely to be included in the result.

3.4.3. *Pathfinding with Dijkstra's Algorithm.* The `_dijkstra()` function finds the lowest-cost path between two nodes in the campus map. It implements *Dijkstra's algorithm*, a classic and well-known method for finding the shortest path in a graph with non-negative edge weights.

The algorithm works by maintaining a *priority queue*. It starts at the source node with a cost of zero, and repeatedly picks the unvisited node with the lowest known cost so far. For each such node, it checks all its neighbouring edges: if travelling via the current node would reach a neighbour at a lower cost than previously known, the neighbour's cost is updated and it is added to the priority queue. This process repeats until the destination node is reached. The path is then reconstructed by tracing back through a `prev` dictionary, which records how each node was reached.

Listing 4. Core loop of Dijkstra's algorithm

```
while pq:
    d, u = heapq.heappop(pq)    # pick lowest-cost node
    if d > dist[u]:
        continue                # skip if already processed
    if u == end_id:
        break                   # reached destination
    for edge in nodes[u].get_edges():
        v = edge.get_other_node_of(nodes[u]).node_id
        c = cost_fn(edge)
        if c >= _INF:
            continue            # skip forbidden edges
        if d + c < dist[v]:
            dist[v] = d + c
            prev[v] = (u, edge)
            heapq.heappush(pq, (d + c, v))
```

The cost function is passed in as an argument (`cost_fn`), which allows the same algorithm to be reused for different routing strategies without any modification.

3.4.4. *Multi-Waypoint Support.* When the user specifies intermediate waypoints, the route is decomposed into segments as described in the Problem Modelling section. The function `_find_route()` iterates over each consecutive pair of waypoints and runs Dijkstra's algorithm independently for each segment. The resulting steps are concatenated to form the complete route, and the total adjusted cost is computed as the sum of the adjusted costs of all steps.

3.4.5. *Multi-Strategy Candidate Generation.* Rather than returning a single route, `find_candidate_routes()` runs the pathfinding algorithm under four different preference configurations to produce a diverse set of options:

(1) The user's own preferences, as configured.
(2) A variant with stair avoidance forced on, encouraging the use of lifts and escalators.
(3) A variant with lift preference forced on, favouring lift-equipped paths.
(4) A *fewest-stops* variant, where all passable edges are assigned a uniform cost of 1 regardless of traversal time, so that Dijkstra minimises the number of hops rather than total time.

After running all four strategies, any duplicate routes — identified by comparing their full ordered sequences of edges — are discarded. The surviving unique routes are then labelled based on their properties: the route with the lowest total time receives the *Fastest* label; the route with the fewest steps receives the *Fewest Stops* label; and routes are additionally tagged to indicate whether they contain stairs, lifts, or escalators.

3.5. **User Interface.** The user interface is implemented using the `Textual` library, which provides a framework for building terminal-based applications with a modern, responsive design. The app launches into an `AppSelector`, where the user can choose to launch the navigation app or the data visualisation app. The navigation app is where the main functionality of the project is implemented, and it is orchestrated by several screens, each focusing on a specific task. The data visualisation app is only a helper for development and **is not** part of this project. The flow of the user interface is illustrated in Figure 3.

Note that the data visualisation app requires a graphical environment to run and will fail if launched in a headless terminal (e.g., via SSH on HKU's Ubuntu servers).

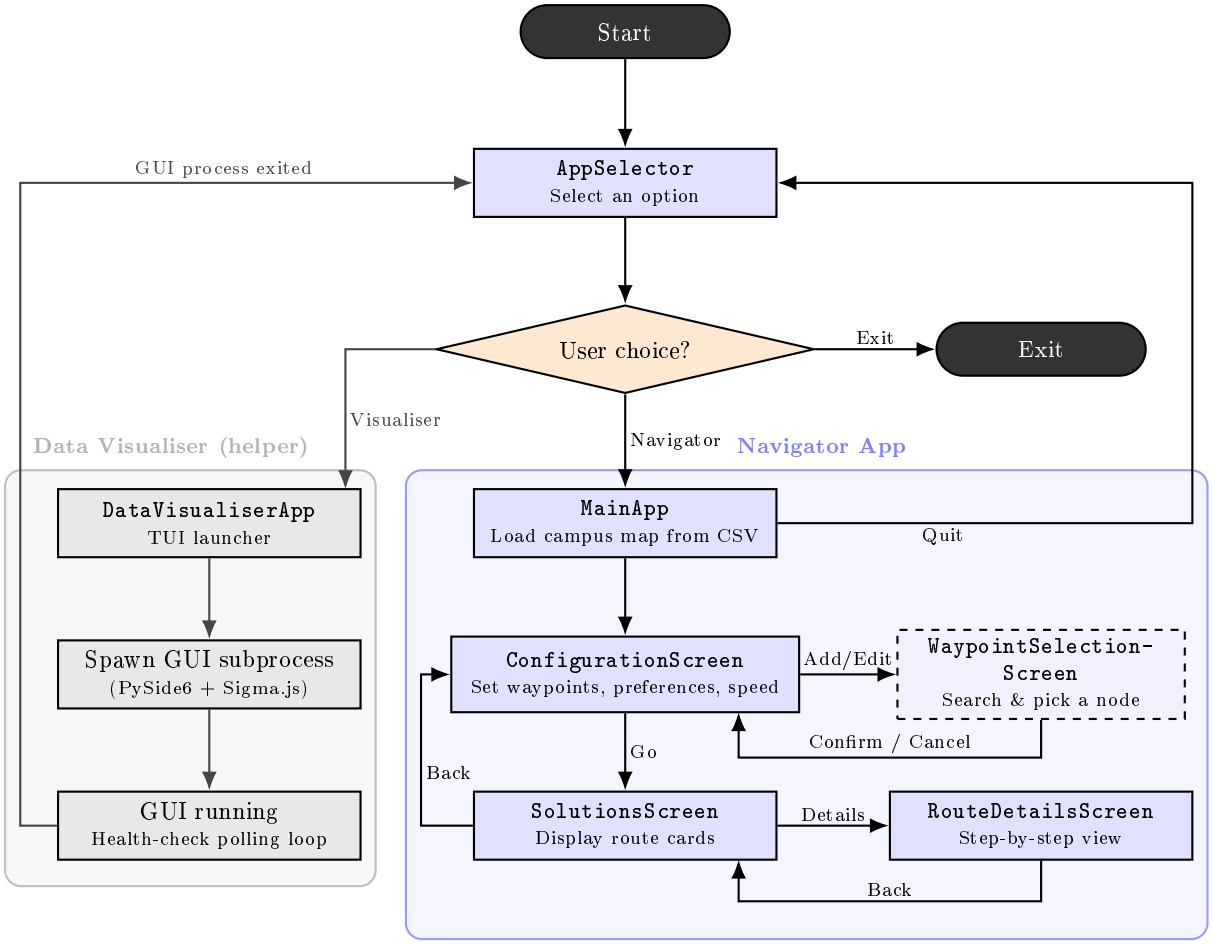

FIGURE 3. User interface flow diagram.

4. SCENARIO STUDY

For the sake of evaluating the performance of the product, several scenarios were designed to test the product's functionality and the quality of the results it produces. These scenarios, together with the sample input, output, and evaluation of the results, are presented in this section. All of the scenarios are tested on the same network graph.

4.1. **Scenario 1 – Last-minute Arrival at Campus.** This scenario simulates a situation where a student arrives at the campus last minute just before the start of a test.

4.1.1. *Scenario Assumptions.* It is assumed that the student arrives at the MTR A2 exit at 10:55 am, and needs to get to the classroom at 1/F of the Meng Wah Complex before 11:00 am.

4.1.2. *Purpose of the Scenario.* The purpose of this scenario is to evaluate the system's ability to provide accurate and efficient navigation solutions under time pressure.

4.1.3. *Inputs.* The following parameters are input into the system (all other parameters are left as default):

- **Starting Node**: `UStr_Upper_MTR_Exit`
- **Destination Node**: `MengWahComplex_1F`
- **Speed Multiplier**: 1.3 (to simulate the student rushing to the classroom with a faster pace)

4.1.4. *Outputs.* The system generated three solutions for the scenario, which are presented in Figure 4.

4.1.5. *Discussions.* For this scenario, one of the main concerns is the time taken to reach the destination. As the student is in a rush, it would be reasonable to expect that the student would walk faster than usual. Existing pedestrian navigation solutions typically assume a normal walking pace and do not account for variations. They tend to estimate the arrival time as the user walks using past data. While this approach may work for later sections of the trail, it is prone to inaccuracies at the beginning. In

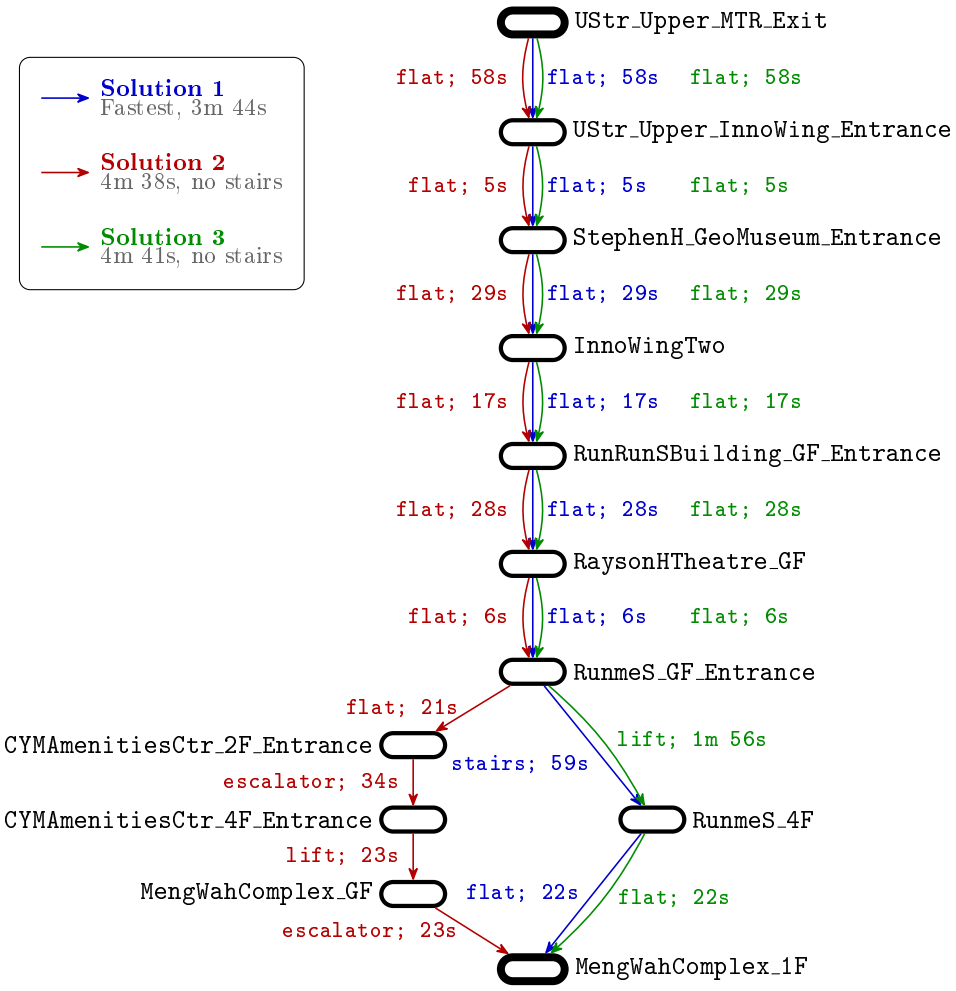

FIGURE 4. Output solutions for Scenario 1.

contrast, our project accounts for a customisable speed multiplier, which allows the user to adjust the estimated time based on their current pace.

4.2. **Scenario 2 – Accessible Route Navigation.** This scenario simulates a user with mobility impairment who needs to navigate across campus while avoiding all stairs and escalators.

4.2.1. *Scenario Assumptions.* It is assumed that the user arrives at the MTR C1 exit and needs to reach the K.K. Leung Building ground floor entrance. The user cannot use stairs or escalators due to mobility constraints and therefore requires a fully lift-accessible route. The user's walking pace is assumed to be slower than average.

4.2.2. *Purpose of the Scenario.* The purpose of this scenario is to evaluate the system's ability to generate accessible routes that strictly exclude stairs and escalators while favouring lift-equipped segments. This tests whether the *Accessible Route* preference correctly propagates the appropriate constraints throughout the route-finding process.

4.2.3. *Inputs.* The following parameters are input into the system (all other parameters are left as default):

- **Starting Node**: `MTR_C1_Exit`
- **Destination Node**: `KKLeung_Building_GF`
- **Accessible Route**: enabled (implicitly sets *Avoid Stairs*, *Avoid Escalators*, and *Prefer Lifts*)
- **Speed Multiplier**: 0.7 (to simulate a slower walking pace)

4.2.4. *Outputs.* The system generated two solutions for the scenario, which are presented in Figure 5.

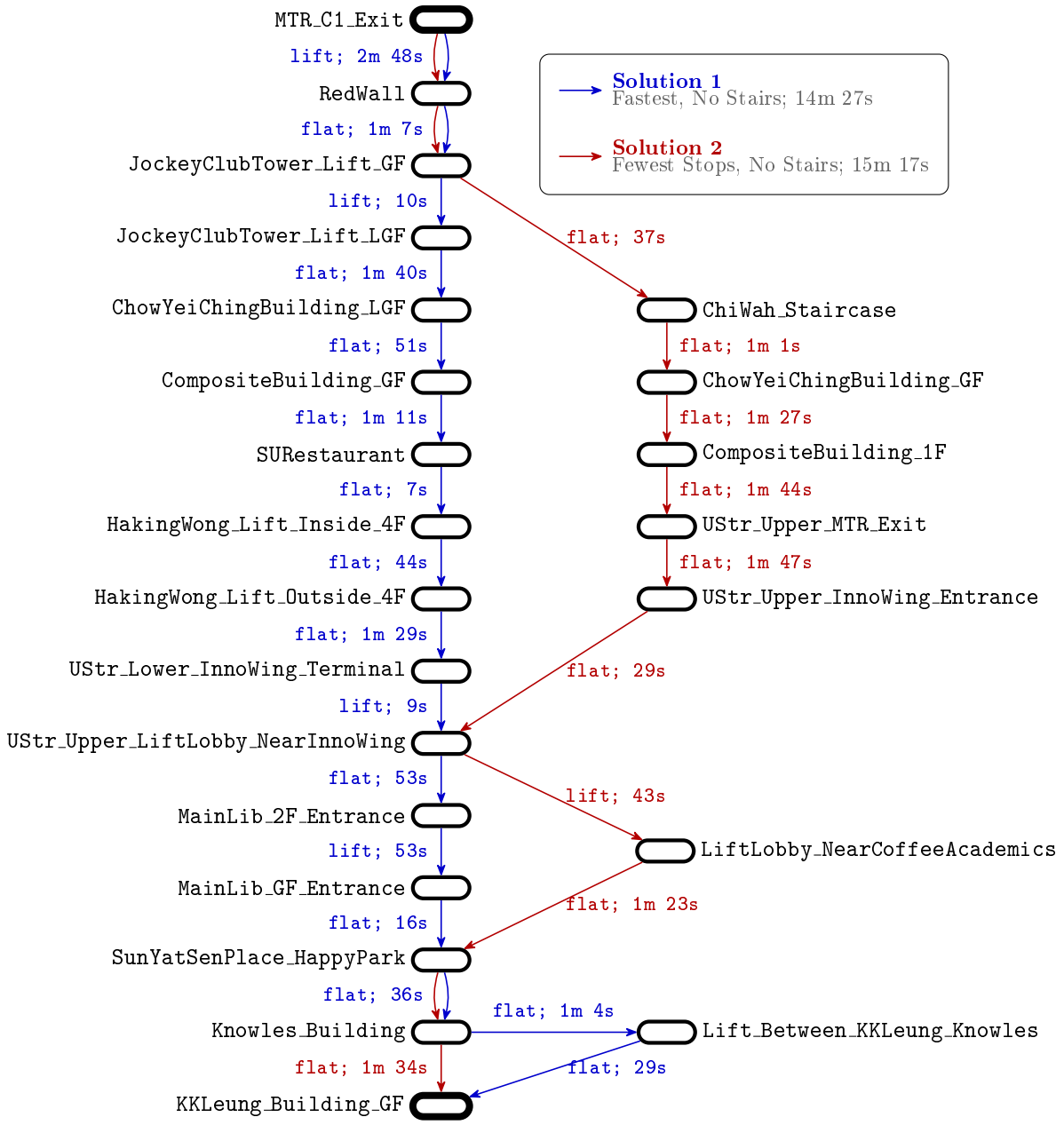

FIGURE 5. Output solutions for Scenario 2.

4.2.5. *Discussions.* For this scenario, the main concern is ensuring a barrier-free path for a user with mobility constraints. Since the user must avoid all stairs and escalators, it is reasonable to assume they may at a slower pace than average. Existing pedestrian navigation solutions often prioritise the shortest geometric distance and frequently overlook specific accessibility barriers for users with slower walking speeds. In contrast, our project includes an "Accessible Route" preference that strictly avoid escalator-required route and utilises a customisable speed multiplier to provide an accurate time estimate tailored to the user's actual pace.

4.3. **Scenario 3 – Multi-Stop Navigation.** This scenario simulates a student who needs to visit an intermediate location on the way to their final destination, requiring the system to plan a route through multiple specified waypoints.

4.3.1. *Scenario Assumptions.* It is assumed that the student starts at the Composite Building ground floor and must pass through the Main Library ground floor entrance before proceeding to the Tang Chi Ngong Building as the final destination. No special mobility preferences are set.

4.3.2. *Purpose of the Scenario.* The purpose of this scenario is to evaluate the system's multi-waypoint route planning capability. Specifically, it tests whether the system correctly chains route segments across multiple stops and produces solutions that respect all intermediate waypoints in the correct order.

4.3.3. *Inputs.* The following parameters are input into the system (all other parameters are left as default):

- **Starting Node**: `CompositeBuilding_GF`
- **Via (Intermediate Stop)**: `MainLib_GF_Entrance`
- **Destination Node**: `TangChiNgong_Building`

4.3.4. *Outputs.* The system generated three solutions for the scenario, which are presented in Figure 6.

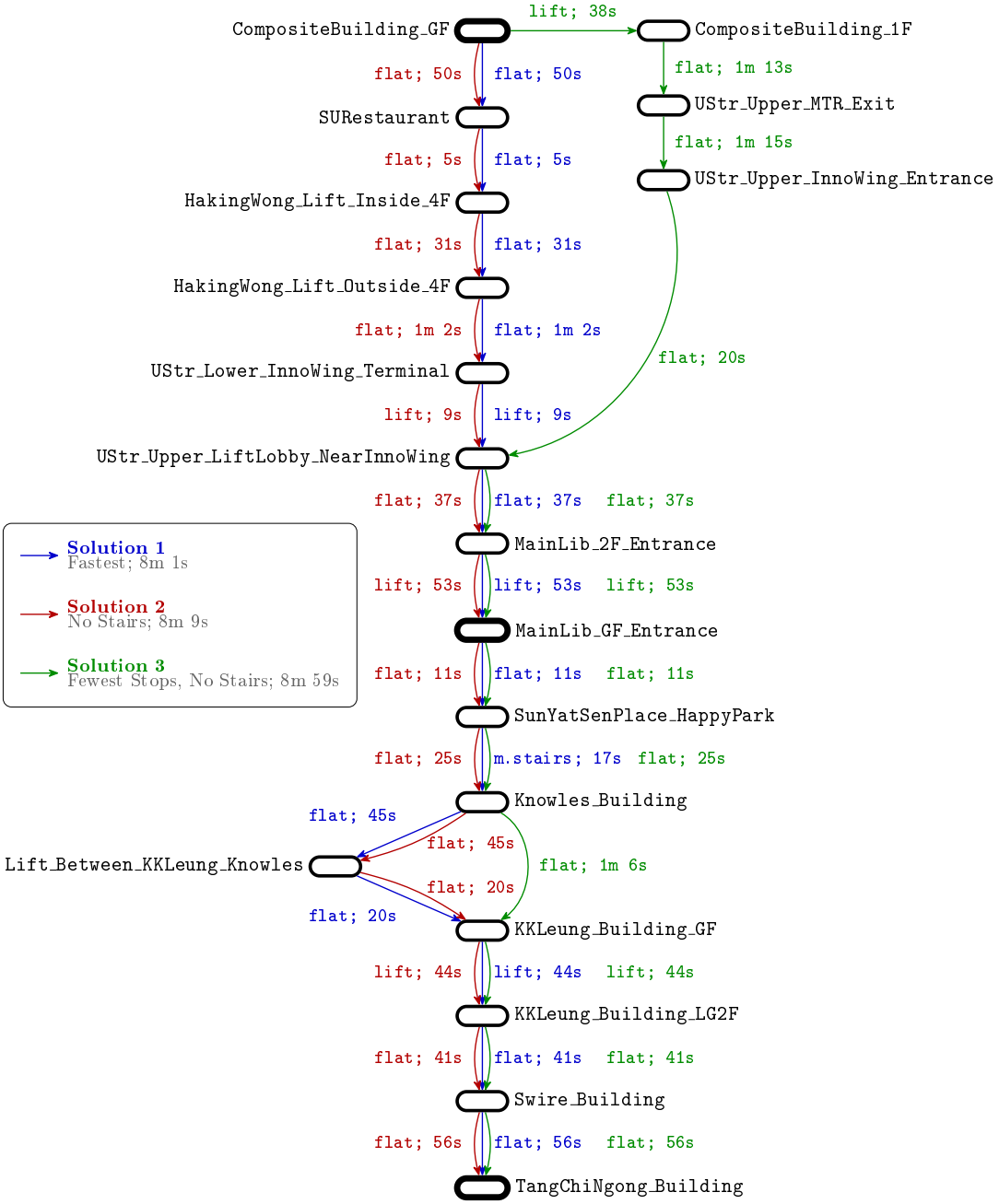

FIGURE 6. Output solutions for Scenario 3.

4.3.5. *Discussions.* For this scenario, the primary focus is the system's ability to plan a cohesive route through multiple intermediate waypoints. As students often need to visit locations like the library before heading to a classroom, the navigation must account for specific stops in a precise order. Existing pedestrian navigation solutions typically treat multi-stop navigation as a series of independent, point-to-point trips, which can lead to inefficient routing at the transition nodes. In contrast, our project features a multi-waypoint planning capability that correctly chains route segments across all stops, ensuring the system produces solutions that respect the user's specific sequence of destinations while maintaining overall efficiency.

4.4. **Scenario 4 – Stair-Preferring Navigation at Elevated Speed.** This scenario simulates a physically active user who explicitly prefers routes that include staircases and walks at a faster-than-normal pace.

4.4.1. *Scenario Assumptions.* It is assumed that the user starts at the MTR Upper University Street exit and needs to reach the Meng Wah Complex 1/F. The user chooses to use stairs wherever possible, for example to incorporate physical activity into their commute, and walks at a pace 1.5 times faster than average.

4.4.2. *Purpose of the Scenario.* The purpose of this scenario is twofold. First, it evaluates whether the *Prefer Stairs* option successfully biases the route engine towards stair-inclusive segments. Second, it assesses whether an elevated speed multiplier correctly reduces estimated travel times, particularly for stair segments which are modelled with a higher speed exponent than flat segments.

4.4.3. *Inputs.* The following parameters are input into the system (all other parameters are left as default):

- **Starting Node**: `UStr_Upper_MTR_Exit`
- **Destination Node**: `MengWahComplex_1F`
- **Prefer Stairs**: enabled
- **Speed Multiplier**: 1.5 (to simulate a faster walking pace)

4.4.4. *Outputs.* The system generated three solutions for the scenario, which are presented in Figure 7.

4.4.5. *Discussions.* For this scenario, the main concern is catering to a user who explicitly prefers stairs and maintains a fast pace. As the user incorporates physical activity into their commute, it is reasonable to expect they would walk significantly faster than usual. Existing pedestrian navigation solutions typically assume a preference for flat terrain and a normal pace. They tend to default to the easiest route and neglect specific requirements from users. In contrast, our project provides a "Prefer Stairs" option and an elevated speed multiplier to accurately reflect travel times for active users.

However, as displayed on the output solutions, the system has included two other routes that contain no stairs. While this is an intended behaviour as the "Prefer" preference profile only biases the route search algorithm rather than enforcing a hard constraint, it may be confusing for users who expect solutions with stairs. For future improvement, a visual prompt that reminds users of the possibility of non-stair routes may be added to the output page to clarify the situation.

4.5. **Overall Findings.** Across all four scenarios, the system demonstrated a consistent ability to accommodate user-specific preferences and constraints that are seldom addressed by general-purpose navigation applications. The preference system proved effective in producing routes tailored to diverse user needs. The speed multiplier further distinguished the system from conventional navigation tools by enabling personalised travel time estimates that reflect individual walking pace rather than relying on population-level averages. The multi-waypoint capability was likewise validated, with the system correctly chaining route segments through ordered intermediate stops while preserving overall route coherence.

Nevertheless, certain limitations were identified during testing. In more complex routing configurations, the system occasionally produced routes containing repeated segments, resulting in unnecessary detours. This issue arises from the segment-wise chaining strategy employed for multi-waypoint routes, where locally optimal sub-routes may share edges when concatenated. Addressing this limitation remains an area for future improvement, and would bring the system's behaviour closer to that expected of a production-grade pedestrian navigation tool.

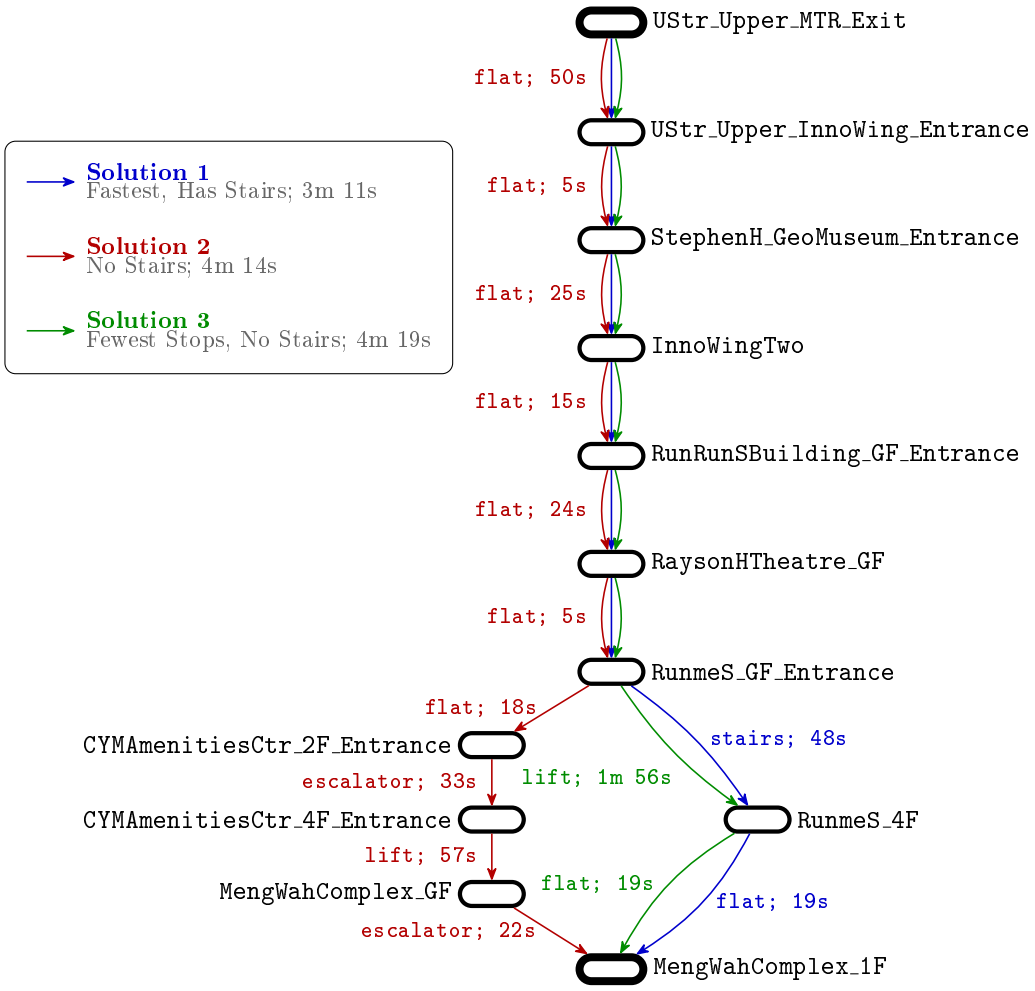

FIGURE 7. Output solutions for Scenario 4.

5. Evaluation, Limitations, and Possible Improvements

5.1. **Evaluation.** The system was evaluated through a structured set of ten test cases, organised into three categories: basic path planning (TC-01 to TC-03), preferences and configuration (TC-04 to TC-07), and boundary and exception handling (TC-08 to TC-10). The test cases cover a representative range of user interactions, including normal shortest-path queries, multi-waypoint routes, the accessible route mode, the speed multiplier, extreme input values, and conflicting or unsatisfiable preference configurations. All ten test cases passed, with actual outputs matching expected behaviour in each case. The full test case descriptions, inputs, expected outputs, actual results, and screenshots are documented in the project's `README.md`.

5.2. **Limitations.** Despite the passing test results, several limitations of the current system were identified.

5.2.1. *Errors in Traversal Time Measurement.* The base traversal times stored in `edges.csv` were collected by group members physically walking each segment and measuring the elapsed time. This process introduces unavoidable human variability: different members walked at different natural paces, and even a single member's pace varies between measurements depending on fatigue, crowd density at the time of measurement, and the precision of the timing method used. As a result, the base traversal times may not be internally consistent across the dataset, and any route cost computed from them carries an inherent margin of error.

5.2.2. *Insufficient Data Variation for Route Diversity.* The multi-strategy candidate generation was designed to return up to four distinct routes per query by running the pathfinding algorithm under different preference profiles. In practice, however, the campus graph is relatively sparse in some areas, meaning that many pairs of nodes have only one or two distinct paths between them regardless of the preference profile applied. When this occurs, the deduplication step discards the redundant candidates,

and fewer routes are returned than expected. Users are therefore sometimes presented with only one or two options rather than a full set of alternatives, which limits the usefulness of the multi-candidate design in those parts of the campus.

5.2.3. *No Support for Real-Time Infrastructure Status.* The system has no mechanism to account for infrastructure that is temporarily out of service. Lifts and escalators on the HKU campus are periodically shut down for maintenance, and certain paths may be closed during construction or special events. Because the graph is a static snapshot loaded from a CSV file, such closures cannot be reflected at runtime. A user who has been routed through a lift that is currently under maintenance will only discover the problem upon arrival, which is precisely the kind of friction the system was intended to eliminate.

5.2.4. *Route Repetition in Multi-Waypoint Queries.* As noted in the Scenario Study section, the multi-waypoint routing strategy decomposes the journey into independent segment-wise sub-problems. Because each segment is optimised independently, the optimal path for one segment may share edges with the optimal path for an adjacent segment, resulting in the user being directed to retrace steps they have already walked. This is a structural limitation of the greedy chaining approach and cannot be resolved without a more sophisticated global optimisation strategy.

5.2.5. *No Consideration of Time-of-Day or Crowd Conditions.* The traversal times in the dataset reflect measurements taken at a particular time and under particular crowd conditions. In reality, heavily trafficked corridors during peak hours (such as just before or after lectures) may take significantly longer to traverse than the recorded values suggest. The system currently has no way to model time-dependent edge weights.

5.3. **Possible Improvements.** Several directions for future improvement follow naturally from the limitations above.

5.3.1. *Standardised Data Collection Protocol.* A more rigorous data collection protocol — for example, requiring each segment to be walked by at least two members independently, with the recorded time taken as the average — would reduce inter-member variability and improve the internal consistency of the base traversal times. Alternatively, a calibration pass in which a single reference member re-walks a subset of edges measured by others could be used to normalise the dataset post-hoc.

5.3.2. *Global Multi-Waypoint Optimisation.* The segment-wise chaining approach could be replaced or supplemented by a post-processing step that detects and removes repeated edge sequences from the concatenated route. For a small number of waypoints (as constrained by the current maximum of five), an exhaustive check of all permutations of intermediate waypoints would also be computationally tractable, allowing the system to find the globally optimal visit order rather than assuming the user-specified order is optimal.

5.3.3. *Time-Dependent Edge Weights.* If traversal time measurements were collected at multiple times of day (e.g., peak and off-peak), the graph model could be extended to store multiple time-dependent weights per edge. The route engine could then select the appropriate weight based on the current time, producing more accurate estimates during busy periods.

References

Appendix A. Revised Task Assignment

The workload assignment planned in the project's plan has been revised as the project progressed. As of the end of the project, the workload distribution by each member is declared as follows:

The contributions declared by **C. Banh** are:

- Survey existing navigation solutions and compare them in a table format.
- Problem modelling for our solution: definition of the type of nodes, measuring method.
- Data collection: collect data in between other team members' assigned zones.
- Provide suggestion and recommendation of displaying UI and useful modes in the program.
- Data cleaning: cleaning up messy data in the Google Sheet.

The contributions declared by **S. Liu** are:

- Be responsible for data collection of a specific region, recorded the time and distance data between several different destinations.
- Data recording: settled and arranged a part of recorded data, provided resources for coding and modelling.
- Checking the program by verifying the accuracy through the comparison with reality, provide related possible solutions and potential improvement for different cases.

The contributions declared by **J. Shing** are:

- **Problem Modelling**: Came up with the rough idea of modelling the campus with the common approach of a graph. Afterwards, finalised more details on the graph representation (e.g., undirected/directed graph, simple graph/multigraph, attributes that edges and nodes should have, etc.) and implemented the fundamental data classes in Python.
- **Core Implementation**: Was responsible for implementing the core functionality of the Navigator app, including the design of UI/UX and the code of the pathfinding engine. Apart from coding, "implementation" also includes deciding on the technical stack, project architecture, development workflow, etc.
- **Data Collection**: Responsible for collecting traversal time data for several buildings, including Meng Wah Complex, Chong Yuet Ming Amenities Centre, Chong Yuet Ming Physics/Chemistry Building, etc.
- **Report Writing**: For the final report, was responsible for writing the sections on problem definition and implementation details.
- **Typesetting**: While other parts of the report were written by other members, they were written on a shared Google Docs file. To maintain a consistent style and format across the report, and provide a more professional presentation, was responsible for typesetting the entire report in LaTeX.

The contributions declared by **Y. Ye** are:

- Collaborated on organising a list of tasks for accomplishing the project.
- Assisted with problem modelling for the solution.
- Contributed to data collection.
- Made a draft of the project timeline.
- Wrote `README.md`, including sections on "How to Use", "Purposes of Each File", and "Sample Test Cases".
- Executed test cases and evaluated the results.

The contributions declared by **Y. Zhang** are:

- Choose key places in the campus to be the nodes in our model.
- Partially collected real-world walking times between campus buildings.
- Analysed the data requirements and theoretically verified the logical feasibility of our journey-generation approach before coding.
- Conducted edge-case testing on the Python app.
- Authored the `README.md`, detailing the setup, test cases, and some of the purposes of code files in the repository.

The group has reviewed the above declared contributions and unanimously agrees that they are a fair and accurate representation of each member's contributions to the project.