Project 3 : Campus Navigation System

Course: CS20009.04 Data Structure
Name: Yijia Chen
Student Number: 24300240127
Date: December 9, 2025

---

1. Introduction

This project focuses on computing the shortest path between two points on a small map. The map is modeled as a graph, and shortest paths are computed using Dijkstra's algorithm. The core algorithms are implemented in C++, with a lightweight Node.js and browser-based interface for visualization.

The project aims to provide the following features:

1. Parse and structure real-world map data from CSV files.
2. Compute shortest paths using Dijkstra's algorithm.
3. Generate K-shortest alternatives using Yen's algorithm.
4. Provide an interactive web interface for visualization.

This report documents the system design, implementation details, algorithmic considerations, and the integration of data, logic, and GUI components.

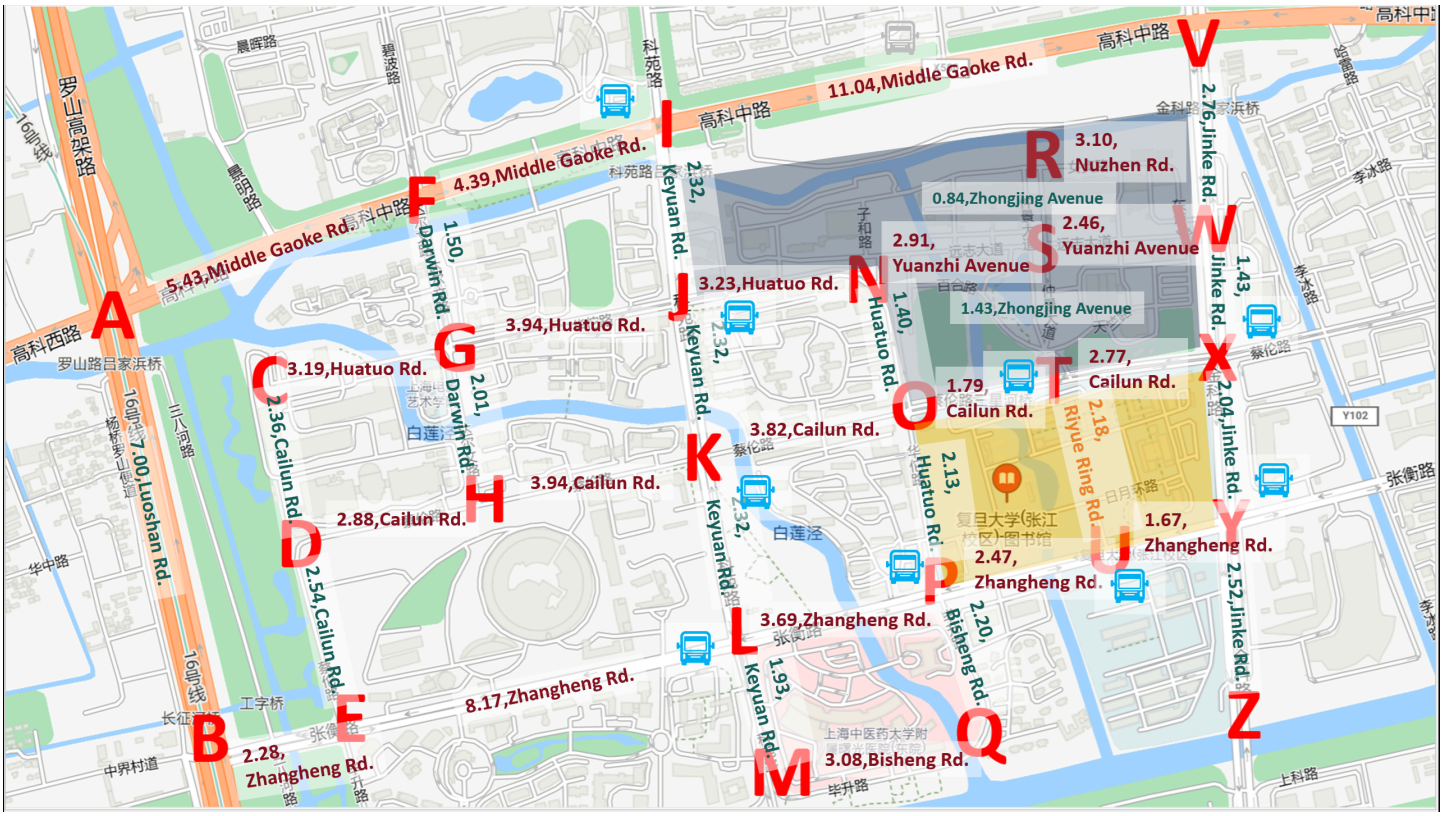
---

2. Basic Functionality Overview

2.1 Map Layout

In the given map, there are 26 points and 39 edges. Each point represents a location that a person may want to visit, and is labeled with a capital letter from A to Z. The edges correspond to the roads that connect two adjacent points in the map.

The distance between two adjacent points, in other words, the distance of a single "hop", is stored in `data/edge.txt`. The unit of distance is hundred meters. For example, there is a road connecting A and B, and the distance between A and B is 7.00 hundred meters. To facilitate user understanding of routing, each "hop" is labeled with its corresponding road name in `data/edgeNamed.csv`, and each point is marked with the names of the two intersecting roads at its respective crossroads in `data/pointNamed.csv`.



2.2 Bus System

Beyond basic pedestrian and road connectivity, the navigation system must also incorporate public transit options.

The map contains several *bus stops* 🚌. The information of bus stops in stored in `data/busStop.csv`. For implementation consistency and data normalization, the two stops Keyuan Road Huatuo Road and Huatuo Road Keyuan Road are combined as a single stop with all available buses.

Bus routes are recorded in `data/busEdge.csv`. Every record corresponds to the route between two adjacent bus stops. Multiple points are recorded in the same record if the bus goes through several points to arrive at the next stop.

For example, the entry `6,J,K,O,P` means that bus number `6` can travel from stop `J` to stop `P`, and it goes from point `J` to `K` to `O` to `P`, where there is no stop at `K` or `O`. Another example is `161,J,K`, which means that bus number `161` can travel from stop `J` directly to stop `K`, where `J` and `K` are adjacent points. Bus number `0` refers to Xuchuan Zhuanxian.

We assume bus routes follow the simplified road network exactly. In reality, bus `161` travels from Middle Gaoke Road Jingming Road to Bibo Road, turns right at Zuchongzhi Road, turns right at Keyuan Road, and finally arrives at Keyuan Road Cailun Road, which is much longer than `F->I->J` in our simplified map.

2.3 Restrictions on Campus

Most of the roads are accessible to everyone, but some of them are not. The map covers two campuses, Shanghai University of Traditional Chinese Medicine (SHUTCM) and Zhangjiang Campus of Fudan University (FDU). SHUTCM is located in the area enclosed by Middle Gaoke Road, Keyuan Road, Huatuo Road, Cailun Road and Jinke Road, which is colored dark cyan in the above picture. FDU, colored golden yellow, is surrounded by Cailun Road, Huatuo Road, Zhangheng Road, and Jinke Road.

These campuses are only open to students and staff, as well as the visitors that has applied for entry permission in advance. We assume that *private cars are not allowed to enter the campus*, so only routes that contain additional segments of walking or cycling on campus may contribute to a better plan.

We do not pay special attention to School of Pharmacy, Zhangjiang Campus of Fudan University, and Student Apartments of SHUTCM for the following reason. There is no road going across these two campuses, which does not lead to reduced accessibility for some people. On the other hand, there are roads across SHUTCM and FDU Zhangjiang Campus, which provides possibly shorter paths for those allowed to enter the campus.

The last column of `data/edgeNamed.csv` indicates the roads' accessibility. `0` means accessible to everyone, `1` means only accessible to people at FDU, `2` means only accessible to people at SHUTCM.

A few additional details require clarification.

Given a starting point and a termination point, the routings on foot or by bike are the same. As is mentioned before, however, the routes by car might be different, because the car is not allowed to enter the campus. Walking or cycling may achieve shorter distance by going across the campus.

From another aspective, we first consider all possible paths without premission to enter the campus, and then filter out the paths that contain additional segments of walking or cycling on campus. The former produces the same routing no matter the visitor chooses to walk, to cycle, or to drive. Similarly, routes with starting or termination points on campus requires entry permission.

2.4 Transportation Modes

A traveller can choose from the following ways to travel:

Means of Transport	Meters per Minute (Approx.)
Walking	50-100
Biking	200-300
Bus	300-500
Car	500-1000

For simplicity, we choose a *single speed value* instead of an interval for every means of transport in `src/PathFinder.h`.

2.5 Time

The user provides a start and end point, and the algorithm returns one or more possible routes. The time is calculated by the following formula:

$$\text{time} = \sum_{\text{periods in one travel plan}} \frac{\text{length of path}_i}{\text{speed}_i}.$$

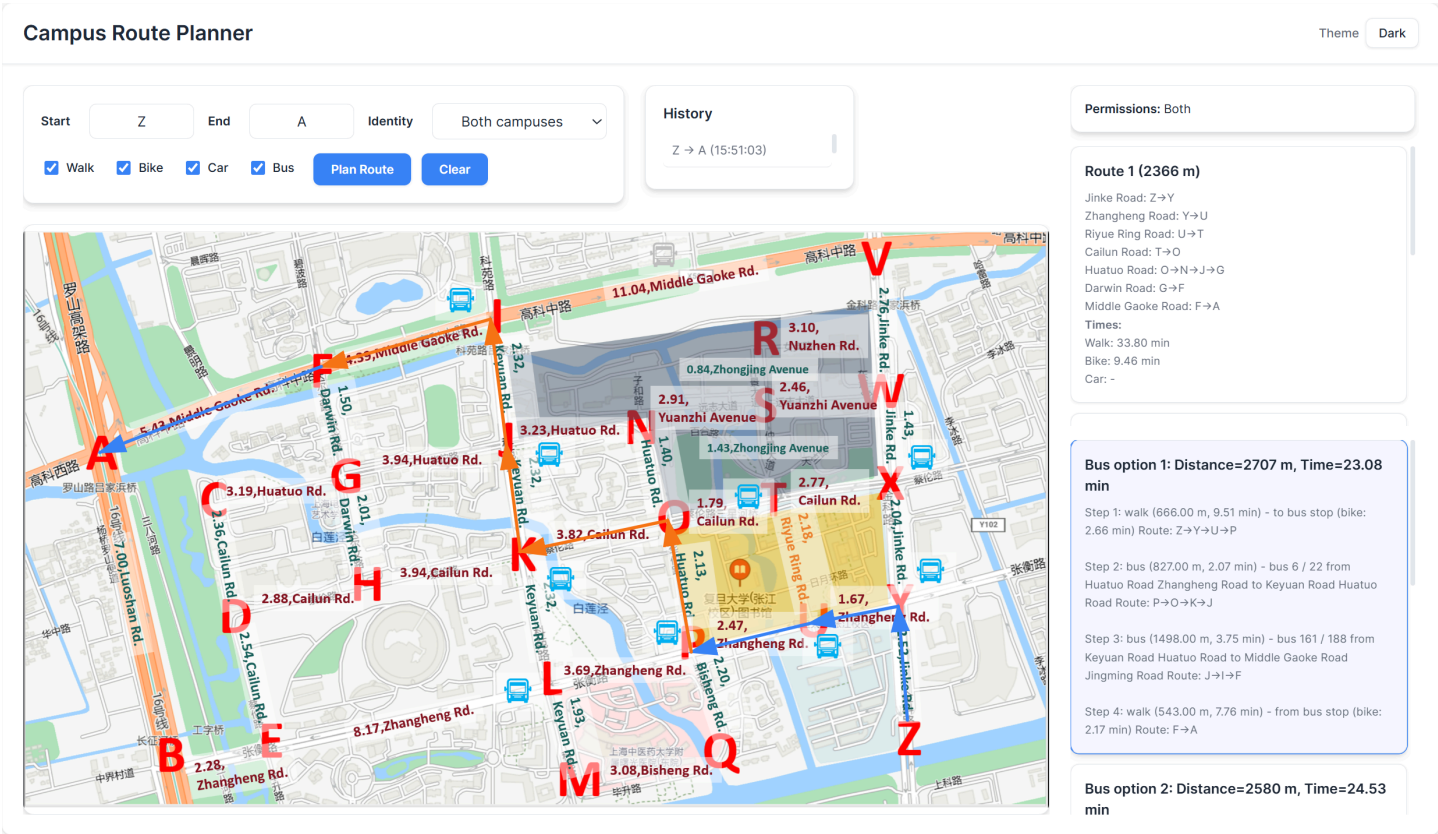
where length of path_i is the length of the i-th period of path, and speed_i is the speed of the i-th path. A "period" (or segment) here refers to a continuous section of the journey that uses the same mode of transportation; thus, a single route plan may consist of multiple periods if it involves switching between

different transport modes (e.g., walking to a bus stop, taking a bus, then walking again to the destination would constitute three separate periods).

2.6 Graphical User Interface Design

GUI is crucial for a user-friendly navigating system.

At the top of the webpage is the *input box*. The user can enter starting point, destination, means of transport, and identity. Beside it is the *query history*. Below the input box is a map, with the selected path drawn on it. The segments by bus is highlighted in orange, while other segments on foot, by bike, or by car are colored blue. On the right side, description of possible paths is listed and sorted by distance. The system displays the three shortest paths for walking, biking, and driving, as well as the three shortest bus routes.



---

3. System Architecture

3.1 Overall Structure

The navigation system follows a modular architecture with clear separation between data processing, algorithm implementation, and user interface. The system consists of three main components:

1. **Data Layer**: Responsible for parsing and storing map data from CSV files. It includes `src/Parser.h` and `src/Parser.cpp`.
2. **Algorithm Layer**: Implements graph algorithms for pathfinding. It mainly includes `src/PathFinder.h` and `src/PathFinder.cpp`.
3. **Presentation Layer**: Provides both command-line and web interfaces for user interaction. The related implementation is in `web/` directory, which interacts with `src/main.cpp`.

3.2 Data Flow

1. **Data Loading**: The `Parser` component reads map data from CSV files and constructs a `Graph` object. The data includes information about locations (points), roads (edges) with distance (weight), and attributes like accessibility and speed.
2. **Path Request**: User submits a route request with start/end points, preferred transportation mode, and entry permission to campuses.
3. **Algorithm Execution**: `PathFinder` executes the appropriate algorithm (Dijkstra or Yen's) to compute paths.
4. **Result Processing**: The system processes the results, calculating distances and times for different modes.
5. **Response Delivery**: Results are returned to the user via either command-line output or JSON API.

---

4. Core Data Structures

4.1 Graph (in `src/Graph.h`)

The system represents the map using an adjacency-list graph structure for efficient neighbor lookups and traversal. It is implemented in the `Graph` class:

```
class Graph {
public:
    Graph();
    bool hasNode(char label) const;            // Check if node exists
    int nodeIndex(char label) const;
    char indexLabel(int idx) const;
    int nodeCount() const;
    void addEdge(char a, char b, double meters, const std::string &roadName, int access);
    const std::vector<Edge>& neighbors(int idx) const;  // Get adjacent edges
    double edgeDistance(int u, int v) const; // return meters or -1 if not connected

private:
    std::unordered_map<char,int> label2idx;     // Map node labels (A-Z) to indices
    std::vector<char> idx2label;                // Map indices back to labels
    std::vector<std::vector<Edge>> adj;         // Adjacency list
    int ensureNode(char label);                 // Helper to add nodes if missing
};
```

4.2 Edge (in `src/Graph.h`)

Each edge in the graph contains detailed information about the road connection:

```
struct Edge {
    int to;                    // Target node index
    double dist;               // Distance in meters
    std::string roadName;      // Road name for description
    int access;                // Access control: 0=public, 1=FDU, 2=SHUTCM
};
```

4.3 Data Bundle (in `src/Parser.h`)

The `DataBundle` struct encapsulates all parsed map data, providing a unified interface for the pathfinding components:

```
struct DataBundle {
    Graph graph;                              // Road network graph
    std::vector<BusRoute> busRoutes;          // Bus route information
    std::vector<char> busStops;               // List of bus stop nodes
    std::unordered_map<char,std::string> busStopNames;  // Bus stop names
    std::unordered_map<char, std::pair<std::string,std::string>> pointNames;  // Node names
};
```

The `BusRoute` struct stores bus route information:

```
struct BusRoute {
    std::string busNo;
    std::vector<char> seq; // sequence of points this bus goes through
};
```

4.4 Plan and PlanStep (in `src/PathFinder.h`)

These structures represent computed routes with transportation-specific details:

```
struct PlanStep {
    std::string mode;          // Transportation mode (walk, bike, car, bus)
    std::vector<char> path;    // Sequence of nodes for this step
    std::string note;          // Additional information (e.g., bus number)
    double distance;           // Distance in meters
    double time;               // Time in minutes
};

struct Plan {
    std::vector<PlanStep> steps;  // Sequence of steps for the entire route
    double totalDistance;         // Total distance in meters
    double totalTime;             // Total time in minutes
};
```

4.5 Speed Model (in `src/PathFinder.h`)

The speed assumptions are also defined:

```
// speeds in m/min
struct Speeds {
    double walk = 70.0;
    double bike = 250.0;
    double bus = 400.0;
    double car = 750.0;
};
```

---

5. Algorithm Implementation

Most of the core functions are implemented in `src/PathFinder.cpp`. Their declarations in `src/PathFinder.h` are included here for reference.

```
#pragma once
#include "Parser.h"
#include <vector>
#include <string>
#include <unordered_set>

struct PlanStep {
    std::string mode;          // Transportation mode (walk, bike, car, bus)
    std::vector<char> path;    // Sequence of nodes for this step
    std::string note;          // Additional information (e.g., bus number)
    double distance;           // Distance in meters
    double time;               // Time in minutes
};

struct Plan {
    std::vector<PlanStep> steps;  // Sequence of steps for the entire route
    double totalDistance;         // Total distance in meters
    double totalTime;             // Total time in minutes
};

// speeds in m/min
struct Speeds {
    double walk = 70.0;
    double bike = 250.0;
    double bus = 400.0;
    double car = 750.0;
};

class PathFinder {
public:
    PathFinder(const DataBundle &d);

    // compute shortest path distances and predecessors with access control
    // identityFlag: bit0 (1)=FDU allowed, bit1 (2)=SHUTCM allowed
    // for carMode: if true, cars are not allowed on campus edges (access 1 or 2)
    void dijkstra(int src, std::vector<double> &dist, std::vector<int> &prev,
                  int identityFlag, bool carMode=false) const;

    // variant of dijkstra that accepts banned edges/nodes (for Yen's k-shortest)
    void dijkstra(int src, std::vector<double> &dist, std::vector<int> &prev,
                  int identityFlag, bool carMode,
                  const std::unordered_set<unsigned long long> &bannedEdges,
                  const std::unordered_set<int> &bannedNodes) const;

    // Yen's algorithm for K shortest simple paths (returns list of paths as node-label vectors)
    std::vector<std::vector<char>> kShortestPaths(char s, char t, int K,
                                                  int identityFlag, bool carMode=false) const;
    std::vector<char> reconstructPath(const std::vector<int> &prev, int dst) const;
```

```
    // helpers
    Plan walkingPlan(char s, char t, int identityFlag) const;
    Plan vehiclePlanFromPath(const std::vector<char> &path, const std::string &mode) const; // bike
    Plan vehiclePlan(char s, char t, int identityFlag, const std::string &mode) const; // bike/car
    std::vector<Plan> busPlans(char s, char t, int identityFlag) const;
    double distanceBetweenNodes(char a, char b, int identityFlag) const;

    // format helpers
    // given a node sequence, produce a collapsed road-name description and total distance
    std::pair<double, std::vector<std::string>> describePathWithRoads(const std::vector<char> &path
    // helpers to compute distance/time/permissions for a given node path
    double pathDistance(const std::vector<char> &path) const;
    double timeForPath(const std::vector<char> &path, const std::string &mode) const;
    bool pathCarAllowed(const std::vector<char> &path) const;
    // compute combined car-to-public-node + walk-into-campus plan when car cannot reach destinatio
    Plan carThenWalkToCampus(char s, char t, int identityFlag) const;

    std::vector<Plan> generateCandidates(char s, char t) const;

private:
    const DataBundle &data;
    Speeds sp;
};
```

5.1 Dijkstra's Algorithm

The core shortest path computation uses Dijkstra's algorithm, implemented in the `PathFinder` class.

5.1.1 `PathFinder::dijkstra`

`PathFinder::dijkstra` is a wrapper for running Dijkstra under normal conditions, when no edges or nodes are temporarily blocked. It constructs two empty sets `bannedE` and `bannedN`, and calls the full Dijkstra function with empty limits.

```
void PathFinder::dijkstra(int src, std::vector<double> &dist, std::vector<int> &prev,
                          int identityFlag, bool carMode) const {
    // wrapper: no banned edges/nodes
    std::unordered_set<unsigned long long> bannedE;
    std::unordered_set<int> bannedN;
    dijkstra(src, dist, prev, identityFlag, carMode, bannedE, bannedN);
}
```

5.1.2 `edgeKey`

The static helper `edgeKey` turns a directed edge `(u, v)` into a unique 64-bit key. The upper 32 bits store `u`, and the lower 32 bits store `v`. It allows edges to be stored in `unordered_set<unsigned long long>` for fast lookup and banning, so that there is no need for custom hash functions. This is crucial for Yen's algorithm, where edges must be temporarily removed.

```
// Helper to combine u,v into a 64-bit key
static unsigned long long edgeKey(int u, int v){
    return ( (unsigned long long)u << 32 ) | (unsigned long long)(unsigned int)v;
}
```

5.1.3 `PathFinder::dijkstra`

The following is the full implementation of Dijkstra's algorithm. It takes into account access restrictions, banned edges, and banned nodes.

The algorithm maintains three data structures:

- `dist[v]` : shortest known distance from source to node `v`.
- `prev[v]` : predecessor pointer for path reconstruction.
- `pq` : priority queue. It stores pairs `(distance, node)` and is a min-heap thanks to `std::greater<P>` declaration.

```
void PathFinder::dijkstra(int src, std::vector<double> &dist, std::vector<int> &prev,
                          int identityFlag, bool carMode,
                          const std::unordered_set<unsigned long long> &bannedEdges,
                          const std::unordered_set<int> &bannedNodes) const {
    // Standard Dijkstra's algorithm with access checks.
    // identityFlag: bit0 = FDU allowed, bit1 = SHUTCM allowed.
    // carMode=true prevents traversing campus-only edges even if identity permits.

    int n = data.graph.nodeCount();
    using P = std::pair<double,int>;
    std::priority_queue<P, std::vector<P>, std::greater<P>> pq;
```

Standard initialization: everything is unreachable except the source.

```
    dist.assign(n, std::numeric_limits<double>::infinity());
    prev.assign(n, -1);
    dist[src]=0;
    pq.push({0,src});
```

The `while` loop repeats until the priority queue `pq` is empty.

```
    while(!pq.empty()){
        auto [d,u] = pq.top();
        pq.pop();
```

At this point, `d` is the candidate shortest distance to `u` discovered so far. `pq` may contain multiple entries for the same `u` (older, stale distances). The next `if` (*lazy deletion*) filters such stale entries.

When the algorithm relaxes an edge and finds a better `nd` for node `v`, it pushes `{nd, v}` onto the heap but does not remove older entries for `v`, because removing arbitrary entries from `std::priority_queue` is expensive. The lazy check avoids processing outdated entries. Ignoring stale entries maintains Dijkstra's invariants because an entry with larger distance cannot lead to shorter paths than the already-known `dist[]`.

```
        if (d > dist[u]) continue;
```

Banned node check: If node `u` is present in the `bannedNodes` set, the algorithm skips processing its outgoing edges entirely.

```
        // banned node check
        if (bannedNodes.find(u) != bannedNodes.end()) continue;
```

The algorithm then iterates over all the neighbors of `u`, check some restrictions, and perform the *relaxation*. Banned edge checks and access permission checks are handled in the similar way to banned node checks.

```
        // iterate neighbors
        for (auto &e: data.graph.neighbors(u)){

            // banned edge check
            if (bannedEdges.find(edgeKey(u,e.to)) != bannedEdges.end()) continue;
```

Access permission: Each edge carries an `access` integer that encodes whether the edge is public or restricted to certain identities. `identityFlag` is a bitmask from the caller: `bit0` indicates FDU permission, `bit1` indicates SHUTCM permission.

```
            // access permissions: 0 public, 1 FDU-only, 2 SHUTCM-only
            if (e.access == 1 && !(identityFlag & 1)) continue;
            if (e.access == 2 && !(identityFlag & 2)) continue;

            // car restriction: campus edges forbidden
            if (carMode && e.access != 0) continue;
```

`v` is the neighbor node index reached by edge `e`. `nd` is the tentative new distance to node `v` obtained by going from `source → ... → u` (distance `d`) and then along the edge `(u→v)` of length `e.dist`.

```
            int v = e.to;
            double nd = d + e.dist;
```

Relaxation: If `nd` (new distance through `u`) is strictly less than the currently known `dist[v]`, we have found a shorter path to `v`. So we update `dist[v]` and `prev[v]`, and push `v` into the priority queue so `v` will be processed later with its updated distance.

```
            if (nd < dist[v]) {
                dist[v] = nd;
                prev[v] = u;
                pq.push({nd, v});
            }
        }
    }
}
```

5.2 Yen's K-Shortest Paths Algorithm

The system implements Yen's algorithm to find K shortest simple paths between two nodes. This algorithm builds upon Dijkstra's algorithm by iteratively finding alternative paths. It generates alternative routes by identifying a deviation point (spur node), temporarily removing edges or nodes that would reproduce previous paths, and running Dijkstra again from the spur node.

The function returns a vector `A` of up to `K` shortest simple paths (each path is a `vector<char>` of node labels). `A` stores the final accepted paths in ascending order. `B` is a temporary collection of candidate paths (each as `(distance, path)`) discovered during spur searches.

If either the source or target label is not present in the graph, return an empty list `A`. Then map the character node labels to internal integer indices `si` and `ti` via `data.graph.nodeIndex`. The rest of the routine uses these indices when calling Dijkstra and computing distances.

```
std::vector<std::vector<char>> kShortestPaths(char s, char t, int K,
                                              int identityFlag, bool carMode=false) const;
    std::vector<std::vector<char>> A;                  // shortest paths
    std::vector<std::pair<double,std::vector<char>>> B; // candidate spur paths (dist, path)

    if (!data.graph.hasNode(s) || !data.graph.hasNode(t)) return A;
    int si = data.graph.nodeIndex(s), ti = data.graph.nodeIndex(t);
```

Firstly, compute the **(first) shortest path** from `si` to `ti` under the given `identityFlag` and `carMode` by calling the Dijkstra wrapper. If the destination is unreachable (`dist[ti] == +∞`), the function returns an empty `A`. Otherwise, `reconstructPath(prev, ti)` converts the `prev[]` predecessor array into a forward `vector<char>` of node labels (using `data.graph.indexLabel` inside `reconstructPath`), producing `p0`, and pushes it into `A` as the first shortest path `P1`.

```
    // first shortest path
    std::vector<double> dist; std::vector<int> prev;
    dijkstra(si, dist, prev, identityFlag, carMode);
    if (dist[ti] == std::numeric_limits<double>::infinity()) return A;
    auto p0 = reconstructPath(prev, ti);
    A.push_back(p0);
```

Outer loop `k` iterates to build `P2 ... PK`, which are the `K` shortest paths that we are looking for. For each new path, we examine the previously found path `A[k-1]`. The inner loop iterates every node in the previous path except the last. For the current index `i`, `spurNode` is the node at position `i` and `rootPath` is the prefix from the start up to and including `spurNode`. `bannedEdges` and `bannedNodes` are freshly created for each spur search and will record edges and nodes to be *temporarily removed* before running a spur Dijkstra from `spurNode`.

```
    for (int k = 1; k < K; ++k){
        // for each node in previous shortest path except last
        for (size_t i = 0; i + 1 < A[k-1].size(); ++i){
            char spurNode = A[k-1][i];
            // root path is nodes from 0..i
            std::vector<char> rootPath(A[k-1].begin(), A[k-1].begin()+i+1);
            // banned edges and nodes
            std::unordered_set<unsigned long long> bannedEdges;
            std::unordered_set<int> bannedNodes;
```

For every already-accepted path `p` in `A`, if `p` shares the same `rootPath` (i.e., its first `i+1` nodes equal `rootPath`), then the edge that follows the root in `p` (edge `p[i] -> p[i+1]`) is banned for the current spur search. Banning those edges prevents Dijkstra from reproducing exactly the same full paths that already exist in `A` while keeping the common root intact. This step is essential to ensure newly found spur paths differ from previously accepted paths at the point of deviation. Note that this loop only bans the exact next-edge from the root that leads to an already-found path; it does not ban other edges beyond the immediate next step.

```
            // remove the edges that would create previously found paths with same root
            for (auto &p: A){
                if (p.size() > i && std::equal(rootPath.begin(), rootPath.end(), p.begin())){
                    int u = data.graph.nodeIndex(p[i]);
                    int v = data.graph.nodeIndex(p[i+1]);
                    bannedEdges.insert(edgeKey(u,v));
                }
            }
```

All nodes in the `rootPath` except the last `spurNode` are banned. This prevents the spur path from routing back through earlier nodes in the root. This enforces that the spur path uses a different continuation while the root prefix stays identical up to the spur node. The spur node itself is not banned so the spur Dijkstra can start there.

```
            // ban nodes in root path except spur node
            for (size_t j = 0; j < rootPath.size() - 1; ++j) bannedNodes.insert(data.graph.nodeInde
```

After that, run Dijkstra from the spur node index `spurIdx` on a temporarily pruned graph, passing `bannedEdges` and `bannedNodes`. The Dijkstra call respects identity and car mode as usual but will skip edges/nodes in the banned sets. If the target is unreachable under these constraints, the search continues to the next spur index. Otherwise, `reconstructPath(sprev, ti)` returns the node sequence from `spurNode` to `t` (since `reconstructPath` expects a `prev` mapping defined by the run whose source was `spurIdx`).

```
            // compute spur path from spurNode to target
            int spurIdx = data.graph.nodeIndex(spurNode);
            std::vector<double> sdist; std::vector<int> sprev;
            dijkstra(spurIdx, sdist, sprev, identityFlag, carMode, bannedEdges, bannedNodes);
            if (sdist[ti] == std::numeric_limits<double>::infinity()) continue;
            auto spurPart = reconstructPath(sprev, ti);
```

`total` is the full candidate path created by concatenating the `rootPath` and the spur segment from `spurNode` to the target, but the first node of `spurPart` is the spur node and already appears as the last element of `rootPath`, so `spurPart.begin()+1` avoids duplicating the spur node. The resulting total is a simple node label path `(start → ... → spur → ... → end)`.

```
            // combine root + spurPart (avoid double counting spur node)
            std::vector<char> total = rootPath;
            total.insert(total.end(), spurPart.begin()+1, spurPart.end());
```

Accumulate the geometric distance of the candidate path by summing `edgeDistance(u,v)` for each consecutive node pair in total. The resulting `td` is used as the path key (distance) and the `(td, total)` pair is appended to `B` as a candidate. Later the algorithm will pick the smallest `td` from `B`.

```
            // compute total distance
            double td = 0;
            for (size_t z = 0; z + 1 < total.size(); ++z){
                td += data.graph.edgeDistance(data.graph.nodeIndex(total[z]), data.graph.nodeIndex(
            }
            B.emplace_back(td, total);
        }
```

After iterating all possible spur nodes for the current `k`, the code checks if `B` contains any candidates. If none were generated, no further alternate paths exist and the outer loop breaks early. Otherwise it sorts `B` by candidate distance ascending, takes the shortest candidate `B.front()`, appends its path to `A` as the next accepted path, and removes that candidate from `B` so that it cannot be chosen again.

```
        if (B.empty()) break;
        // choose shortest candidate from B
        std::sort(B.begin(), B.end(), [](auto &a, auto &b){ return a.first < b.first; });
        A.push_back(B.front().second);
        B.erase(B.begin());
    }
    return A;
}
```

5.3 Algorithms for different modes

Theses functions call the Dijkstra search and Yen's algorithm to find paths for different modes of transportation.

5.3.1 Walking/Biking Plans

Walking and biking use the same path but different speeds:

```
Plan walkingPlan(char s, char t, int identityFlag) const;
Plan vehiclePlanFromPath(const std::vector<char> &path, const std::string &mode) const;
```

5.3.2 Car Plans

Car plans respect campus access restrictions and use higher speeds. This function tries to reuse the existing Dijkstra search for walking plans first.

```
Plan vehiclePlan(char s, char t, int identityFlag, const std::string &mode) const;
```

5.3.3 Bus Plans

This function generates transportation plans that combine walking and bus travel between two locations.

```
std::vector<Plan> busPlans(char s, char t, int identityFlag) const;
```

The function evaluates every bus route. For each pair of bus stops on a route, it checks whether the user can walk to the boarding stop, take the bus segment between them, and then walk from the final stop to the destination. Only reasonable options are kept, for example, routes that are excessively longer than walking are discarded. Each valid combination becomes a candidate plan composed of steps: walk to the stop, ride the bus, and walk to the destination.

Then, it considers two-bus transfers. It searches for transfer stops shared by two bus lines and builds plans that walk to the first bus, ride to the transfer point, continue on a second bus, and finally walk to the target. Again, unrealistic detours are filtered out.

We don't consider transfers that involve more than two bus lines.

After all single-bus and two-bus plans are generated, the function merges those that have identical paths but differ only in bus numbers, leaving a cleaner, deduplicated set.

---

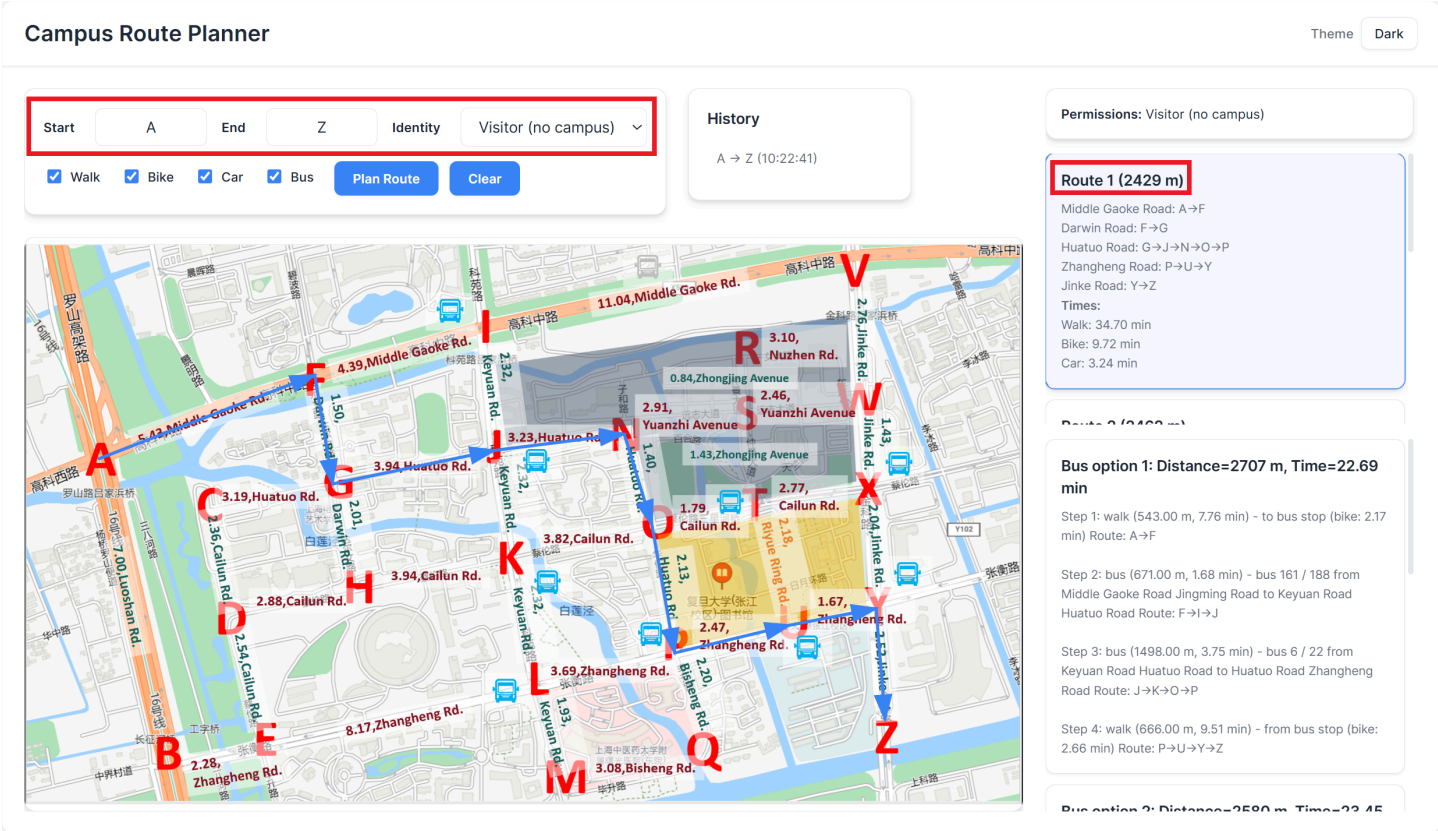
6. System Demonstration

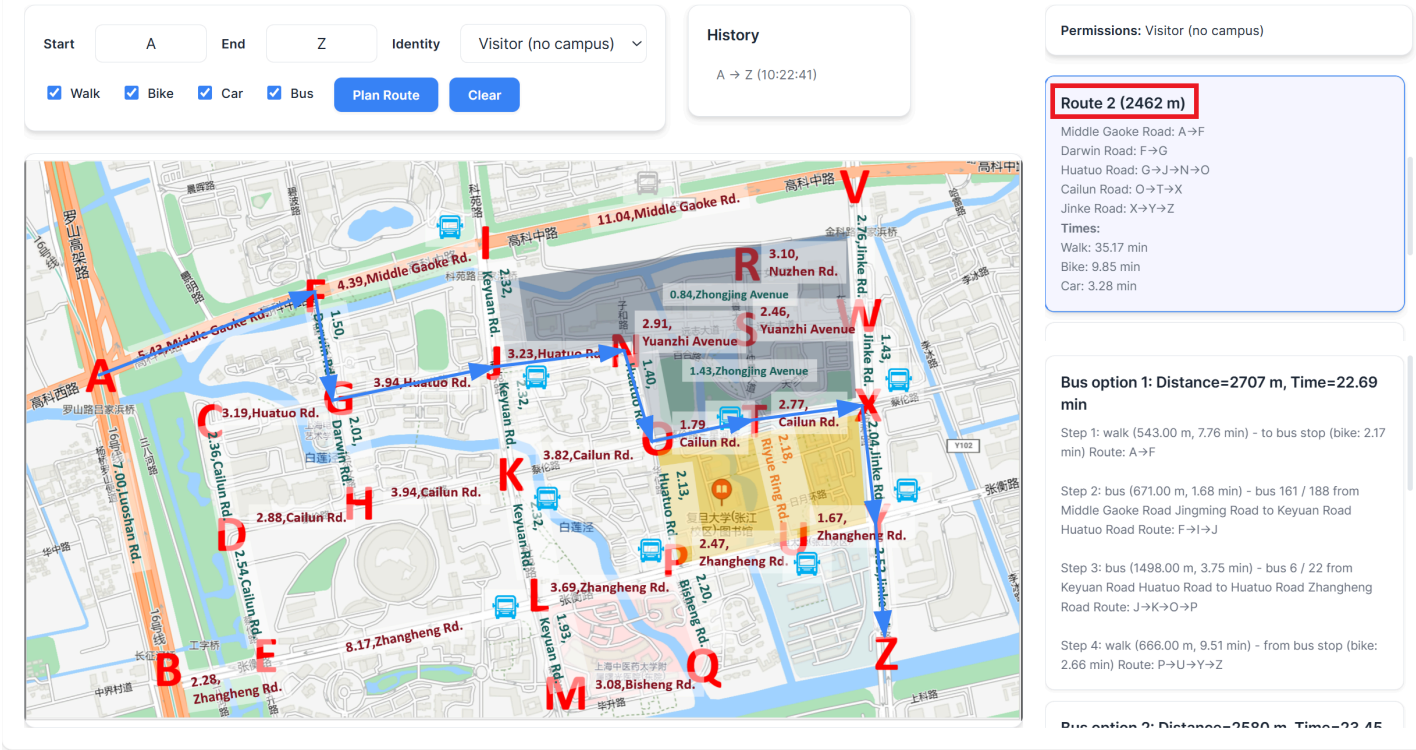
Here are some examples that help illustrate the algorithm, user interface, and some edge cases.

6.1 Example 1: Longest Journey `A~>Z`

The user requests the shortest path from A to Z. In the first case that we demonstrate, the user has no entry permission to any campus.

The following three screenshots show the three possible paths on foot, by bike, or by car.





The system also gives three possible paths by bus. Since the journey is quite long, all the three candidates integrate two segments of bus rides.

Campus Route Planner

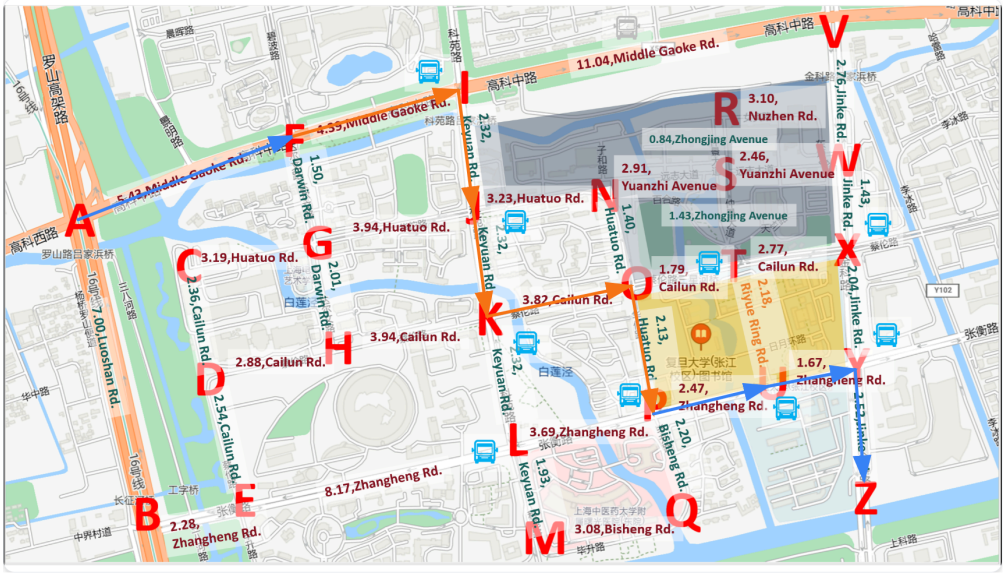
Theme Dark

Start A End Z Identity Visitor (no campus)

Walk Bike Car Bus Plan Route Clear

History

A → Z (10:22:41)



Permissions: Visitor (no campus)

Route 3 (2549 m)

Middle Gaoke Road: A→F
Darwin Road: F→G→H
Cailun Road: H→K→O
Huatuo Road: O→P
Zhangheng Road: P→U→Y
Jinke Road: Y→Z
Times:
Walk: 36.41 min
Bike: 10.20 min
Car: 3.40 min

Bus option 1: Distance=2707 m, Time=22.69 min

Step 1: walk (543.00 m, 7.76 min) - to bus stop (bike: 2.17 min) Route: A→F

Step 2: bus (671.00 m, 1.68 min) - bus 161 / 188 from Middle Gaoke Road Jingming Road to Keyuan Road Huatuo Road Route: F→I→J

Step 3: bus (1498.00 m, 3.75 min) - bus 6 / 22 from Keyuan Road Huatuo Road to Huatuo Road Zhangheng Road Route: J→K→O→P

Step 4: walk (666.00 m, 9.51 min) - from bus stop (bike: 2.66 min) Route: P→U→Y→Z

Campus Route Planner

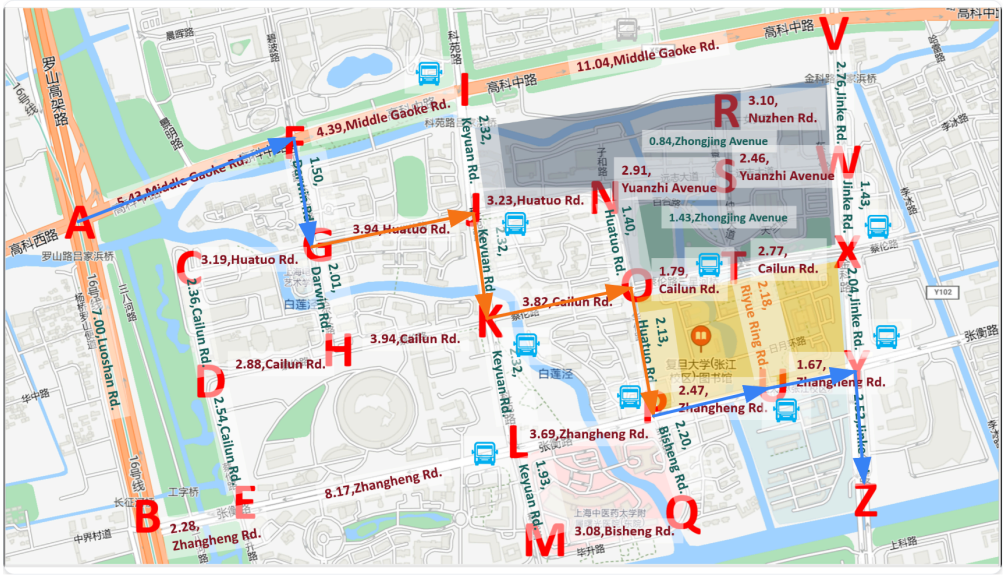
Theme Dark

Start A End Z Identity Visitor (no campus)

Walk Bike Car Bus Plan Route Clear

History

A → Z (10:22:41)



Permissions: Visitor (no campus)

Route 3 (2549 m)

Middle Gaoke Road: A→F
Darwin Road: F→G→H
Cailun Road: H→K→O
Huatuo Road: O→P
Zhangheng Road: P→U→Y
Jinke Road: Y→Z
Times:
Walk: 36.41 min
Bike: 10.20 min
Car: 3.40 min

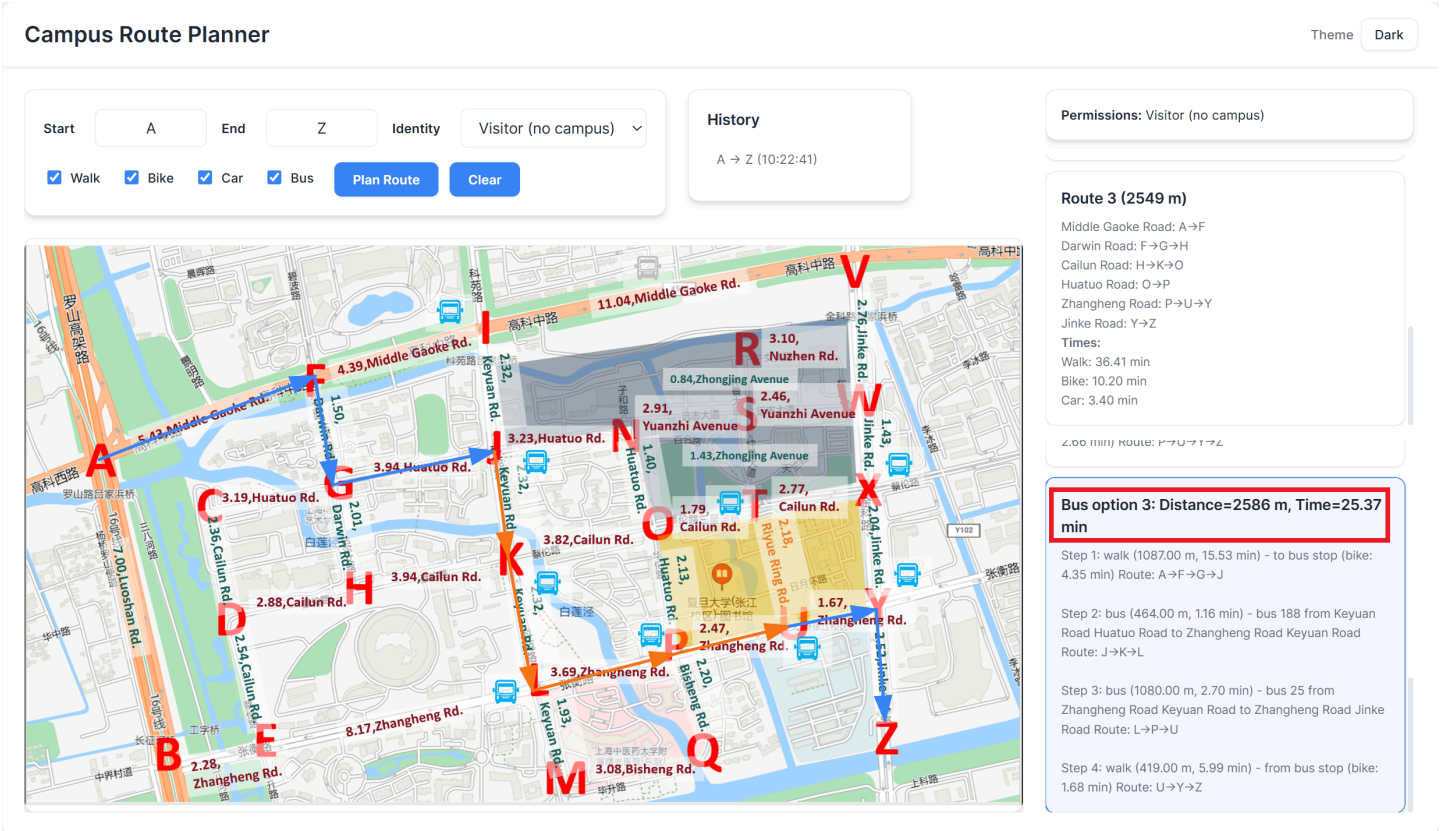
Bus option 2: Distance=2580 m, Time=23.45 min

Step 1: walk (693.00 m, 9.90 min) - to bus stop (bike: 2.77 min) Route: A→F→G

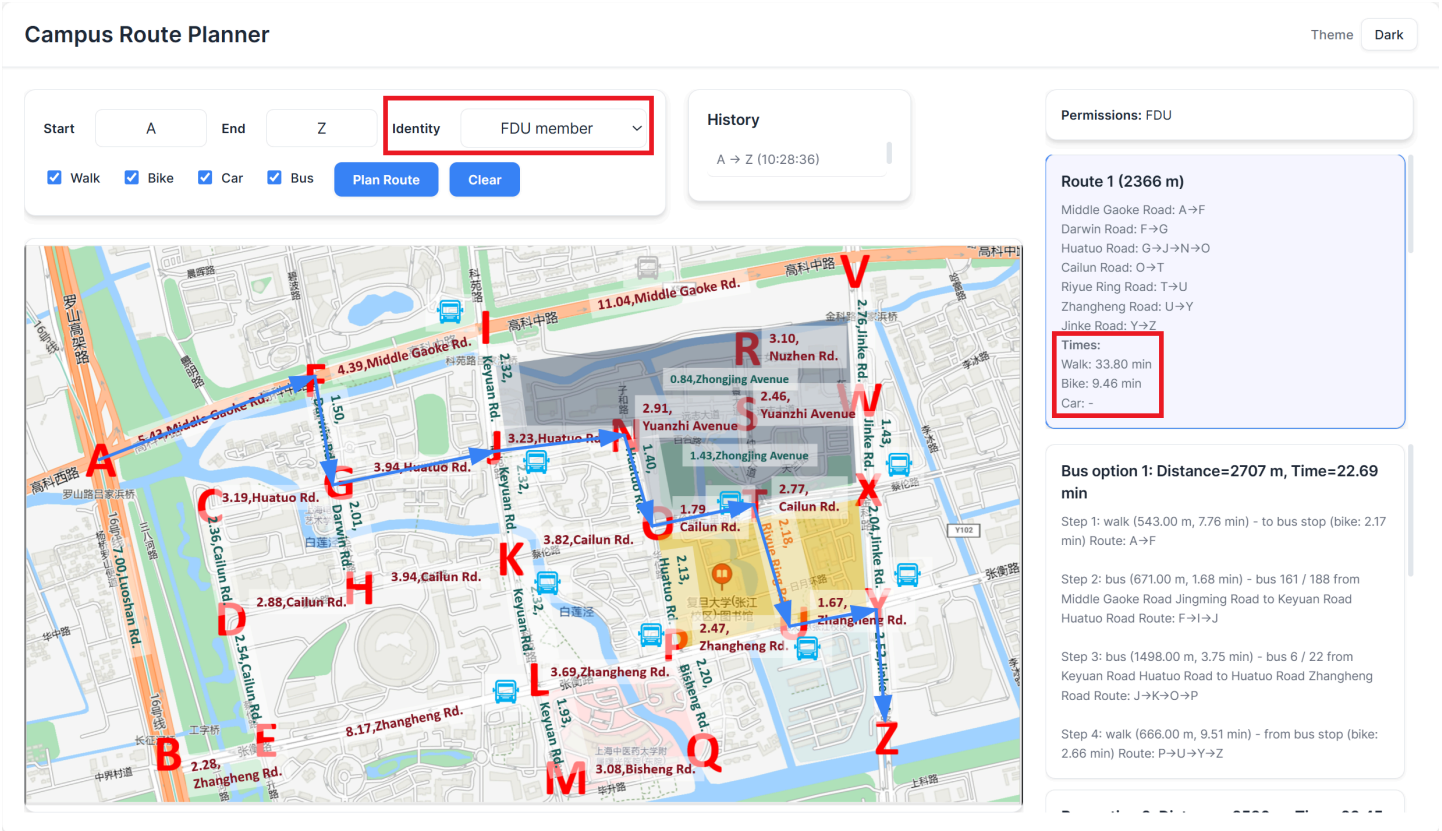
Step 2: bus (394.00 m, 0.98 min) - bus 112 from Huatuo Road Darwin Road to Keyuan Road Huatuo Road Route: G→J

Step 3: bus (1221.00 m, 3.05 min) - bus 6 / 22 from Keyuan Road Huatuo Road to Huatuo Road Zhangheng Road Route: J→K→O→P

Step 4: walk (666.00 m, 9.51 min) - from bus stop (bike: 2.66 min) Route: P→U→Y→Z



If the user is allowed to enter FDU, there is a better path. Note that cars are not allowed on campus, so the route by car does not make sense and its time is omitted.



6.2 Example 2: One Step across FDU `U~>T`

These six screenshots show the possible paths when the user is denied access to all campuses.

Start U End T Identity Visitor (no campus)

Walk Bike Car Bus Plan Route Clear

History

U → T (14:21:46)

Permissions: Visitor (no campus)

Route 1 (639 m)

Zhangheng Road: U→P
Huatuo Road: P→O
Cailun Road: O→T
Times:
Walk: 9.13 min
Bike: 2.56 min
Car: 0.85 min

Route 2 (648 m)

Zhangheng Road: U→Y
Jinke Road: Y→X
Cailun Road: X→T

Bus option 1: Distance=639 m, Time=2.21 min

Step 1: bus (247.00 m, 0.62 min) - bus 188 / 25 from Zhangheng Road Jinke Road to Huatuo Road Zhangheng Road Route: U→P

Step 2: bus (639.00 m, 1.60 min) - bus 58 from Huatuo Road Zhangheng Road to Cailun Road Huatuo Road Route: P→O→T

Bus option 2: Distance=639 m, Time=4.51 min

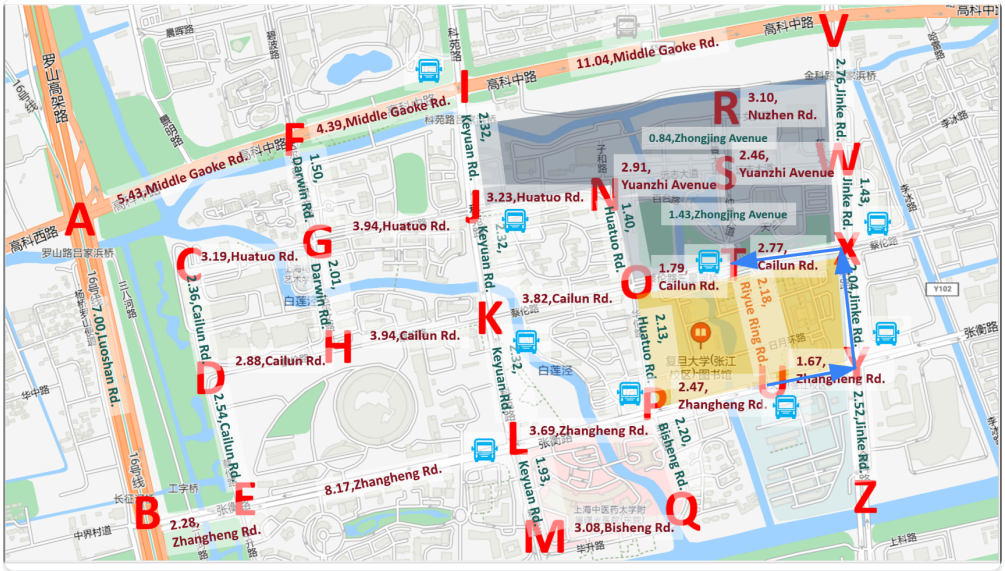
Step 1: walk (247.00 m, 3.53 min) - to bus stop (bike: 0.99 min) Route: U→P

Start U End T Identity Visitor (no campus)

Walk Bike Car Bus Plan Route Clear

History

U → T (14:21:46)



Permissions: Visitor (no campus)

Route 2 (648 m)

Zhangheng Road: U→Y
Jinke Road: Y→X
Cailun Road: X→T
Times:
Walk: 9.26 min
Bike: 2.59 min
Car: 0.86 min

Route 3 (1409 m)

Zhangheng Road: U→P→L
Keyuan Road: L→K

Bus option 1: Distance=639 m, Time=2.21 min

Step 1: bus (247.00 m, 0.62 min) - bus 188 / 25 from Zhangheng Road Jinke Road to Huatuo Road Zhangheng Road Route: U→P

Step 2: bus (639.00 m, 1.60 min) - bus 58 from Huatuo Road Zhangheng Road to Cailun Road Huatuo Road Route: P→O→T

Bus option 2: Distance=639 m, Time=4.51 min

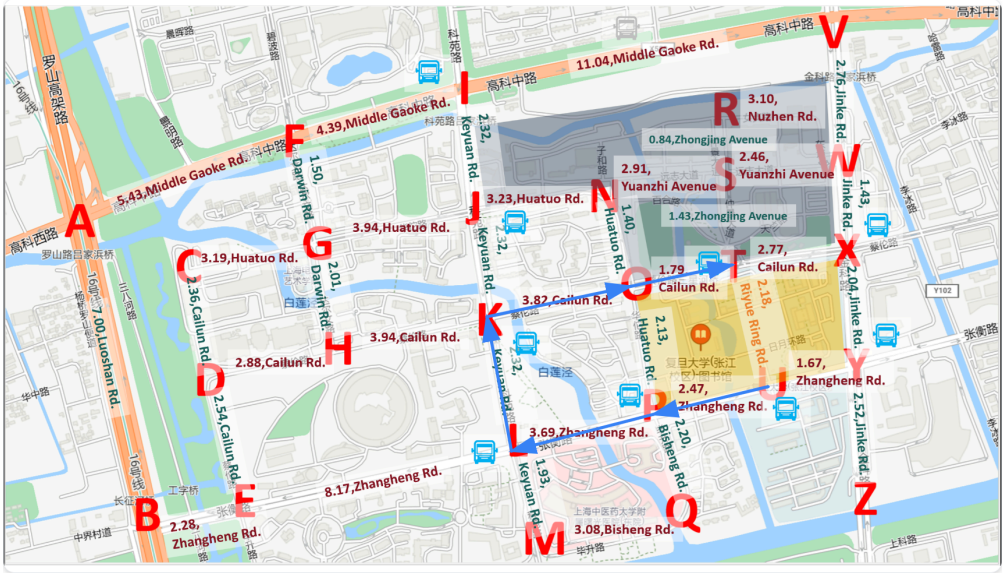
Step 1: walk (247.00 m, 3.53 min) - to bus stop (bike: 0.99 min) Route: U→P

Start U End T Identity Visitor (no campus)

Walk Bike Car Bus Plan Route Clear

History

U → T (14:21:46)



Permissions: Visitor (no campus)

Cailun Road: X→T
Times:
Walk: 9.26 min
Bike: 2.59 min
Car: 0.86 min

Route 3 (1409 m)

Zhangheng Road: U→P→L
Keyuan Road: L→K
Cailun Road: K→O→T
Times:
Walk: 20.13 min
Bike: 5.64 min
Car: 1.88 min

Bus option 1: Distance=639 m, Time=2.21 min

Step 1: bus (247.00 m, 0.62 min) - bus 188 / 25 from Zhangheng Road Jinke Road to Huatuo Road Zhangheng Road Route: U→P

Step 2: bus (639.00 m, 1.60 min) - bus 58 from Huatuo Road Zhangheng Road to Cailun Road Huatuo Road Route: P→O→T

Bus option 2: Distance=639 m, Time=4.51 min

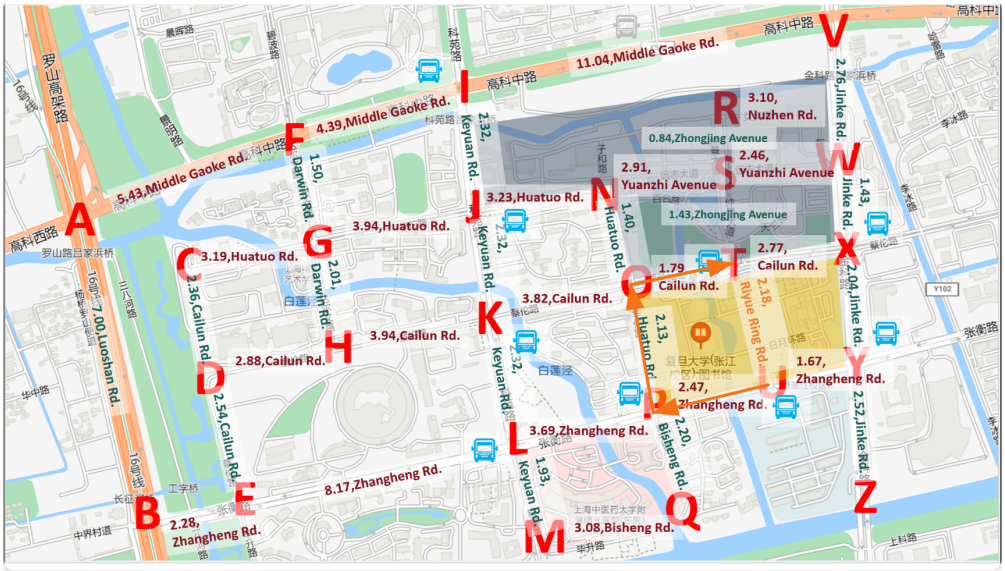
Step 1: walk (247.00 m, 3.53 min) - to bus stop (bike: 0.99 min) Route: U→P

Start U End T Identity Visitor (no campus)

Walk Bike Car Bus Plan Route Clear

History

U → T (14:21:46)



Permissions: Visitor (no campus)

Cailun Road: X→T
Times:
Walk: 9.26 min
Bike: 2.59 min
Car: 0.86 min

Route 3 (1409 m)

Zhangheng Road: U→P→L
Keyuan Road: L→K
Cailun Road: K→O→T
Times:
Walk: 20.13 min
Bike: 5.64 min
Car: 1.88 min

Bus option 1: Distance=639 m, Time=2.21 min

Step 1: bus (247.00 m, 0.62 min) - bus 188 / 25 from Zhangheng Road Jinke Road to Huatuo Road Zhangheng Road Route: U→P

Step 2: bus (639.00 m, 1.60 min) - bus 58 from Huatuo Road Zhangheng Road to Cailun Road Huatuo Road Route: P→O→T

Bus option 2: Distance=639 m, Time=4.51 min

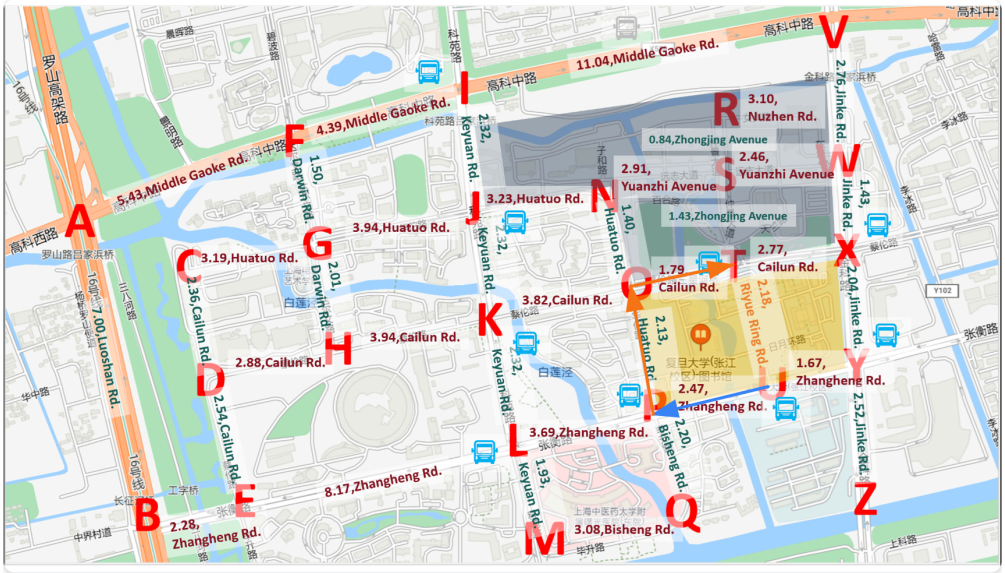
Step 1: walk (247.00 m, 3.53 min) - to bus stop (bike: 0.99 min) Route: U→P

Start U End T Identity Visitor (no campus)

Walk Bike Car Bus Plan Route Clear

History

U → T (14:21:46)



Permissions: Visitor (no campus)

Cailun Road: X→T
Times:
Walk: 9.26 min
Bike: 2.59 min
Car: 0.86 min

Route 3 (1409 m)

Zhangheng Road: U→P→L
Keyuan Road: L→K
Cailun Road: K→O→T
Times:
Walk: 20.13 min
Bike: 5.64 min
Car: 1.88 min

Bus option 2: Distance=639 m, Time=4.51 min

Step 1: walk (247.00 m, 3.53 min) - to bus stop (bike: 0.99 min) Route: U→P

Step 2: bus (392.00 m, 0.98 min) - bus 58 from Huatuo Road Zhangheng Road to Cailun Road Huatuo Road Route: P→O→T

Bus option 3: Distance=639 m, Time=6.22 min

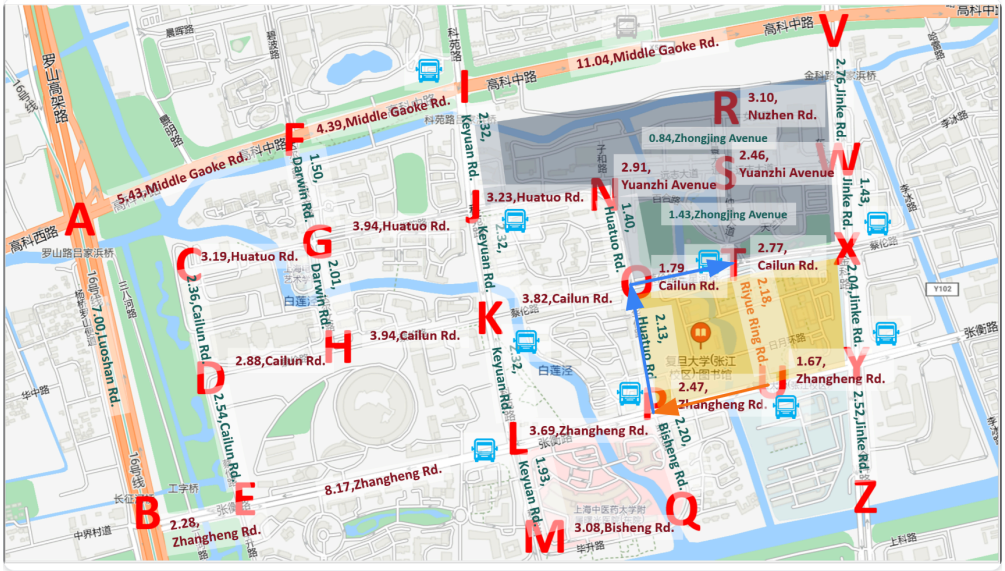
Step 1: bus (247.00 m, 0.62 min) - bus 188 / 25 from

Start U End T Identity Visitor (no campus)

Walk Bike Car Bus Plan Route Clear

History

U → T (14:21:46)



Permissions: Visitor (no campus)

Cailun Road: X→T
Times:
Walk: 9.26 min
Bike: 2.59 min
Car: 0.86 min

Route 3 (1409 m)

Zhangheng Road: U→P→L
Keyuan Road: L→K
Cailun Road: K→O→T
Times:
Walk: 20.13 min
Bike: 5.64 min
Car: 1.88 min

0.99 min) Route: U→P

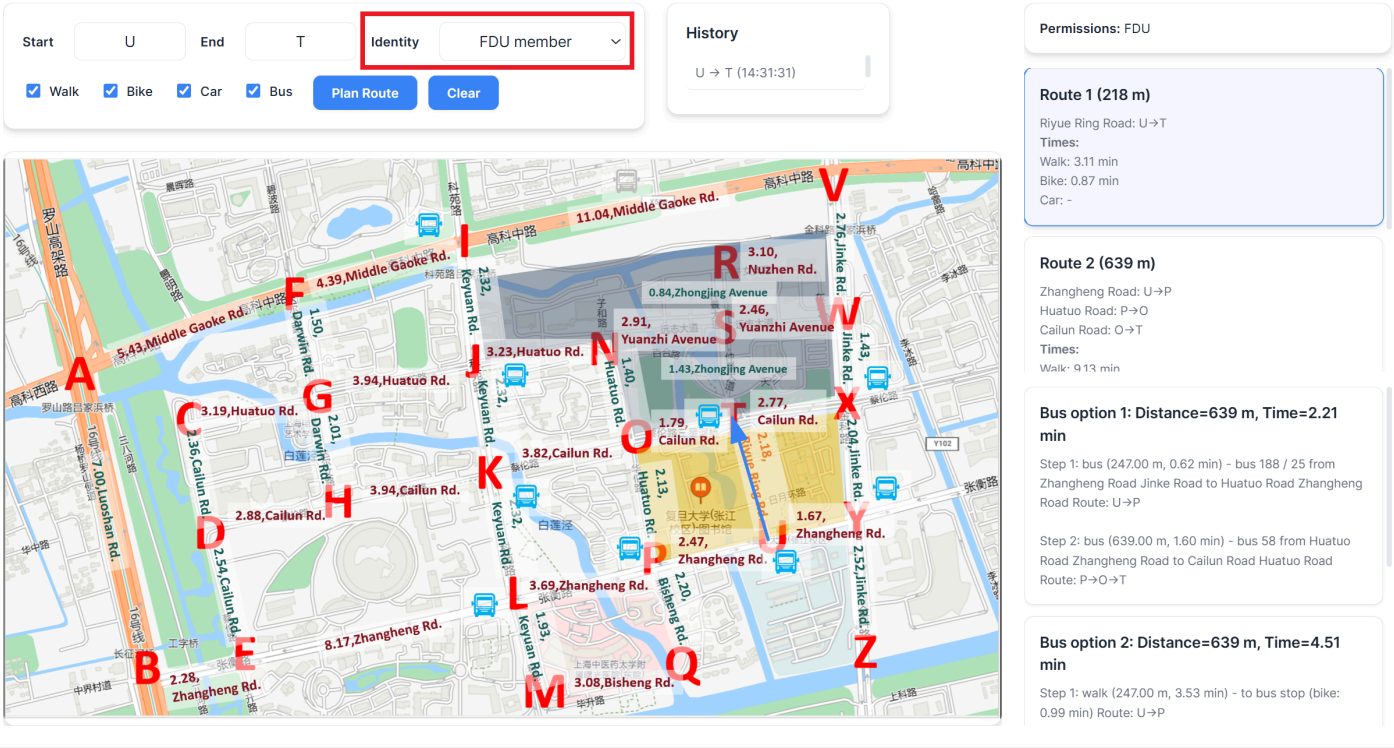
Step 2: bus (392.00 m, 0.98 min) - bus 58 from Huatuo Road Zhangheng Road to Cailun Road Huatuo Road Route: P→O→T

Bus option 3: Distance=639 m, Time=6.22 min

Step 1: bus (247.00 m, 0.62 min) - bus 188 / 25 from Zhangheng Road Jinke Road to Huatuo Road Zhangheng Road Route: U→P

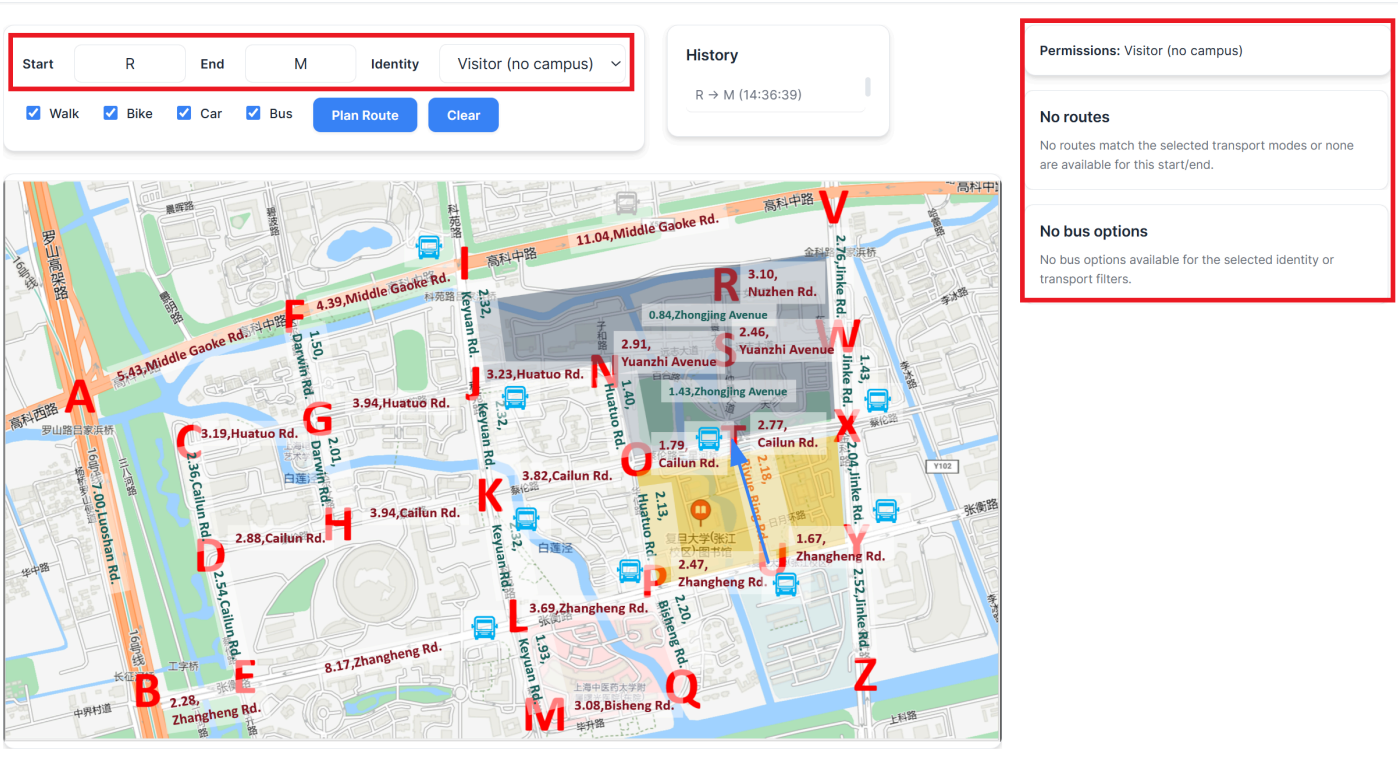
Step 2: walk (392.00 m, 5.60 min) - from bus stop (bike: 1.57 min) Route: P→O→T

When the user is allowed to enter FDU, the system provides a better path.

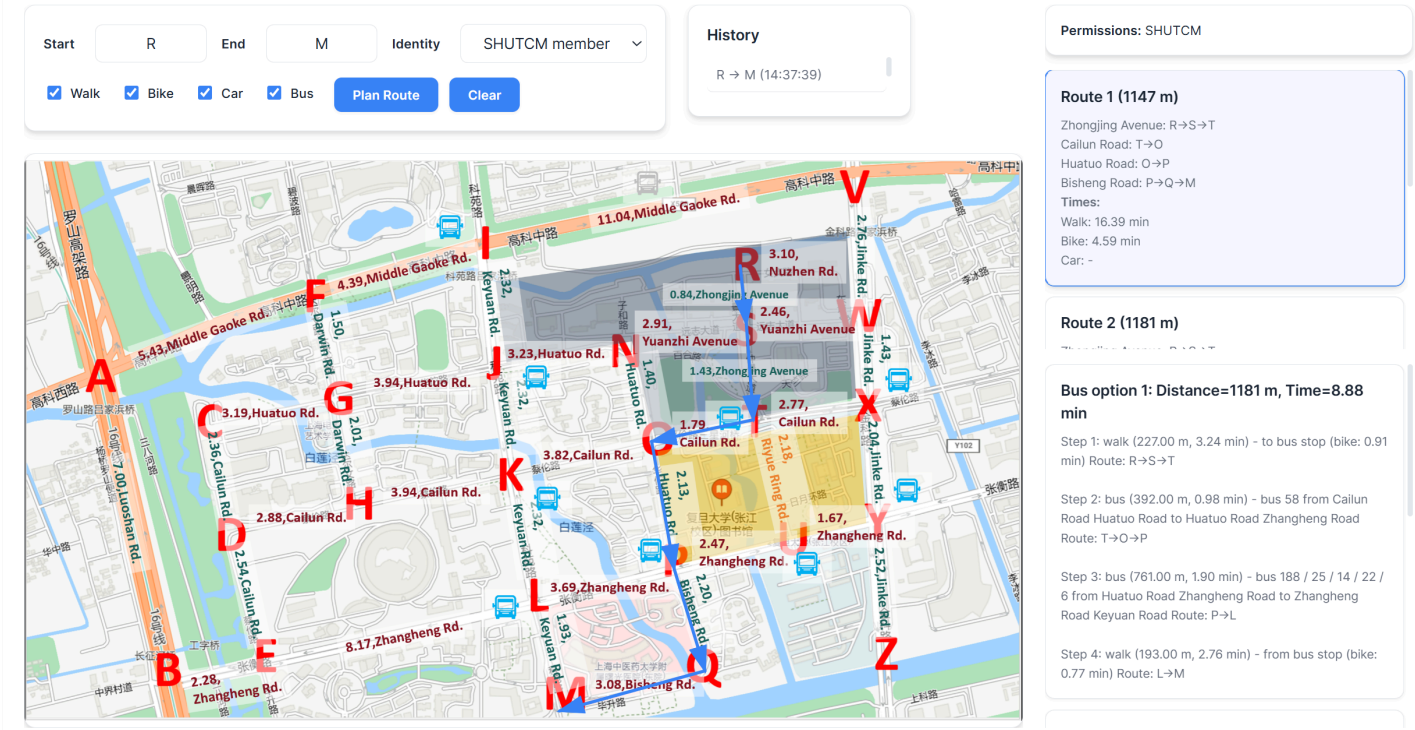


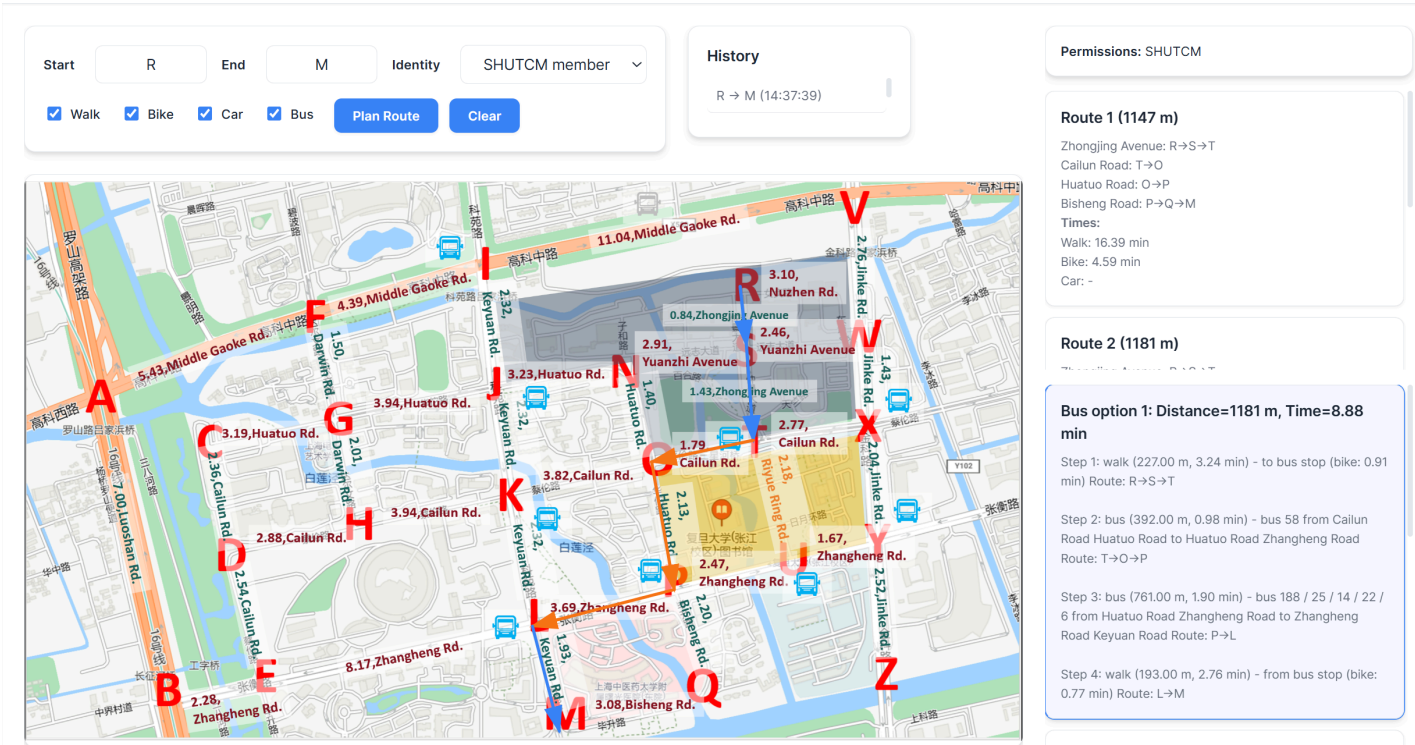
6.3 Example 3: Starting Point on Campus `R~>M`

There is no path with permission to SHUTCM.



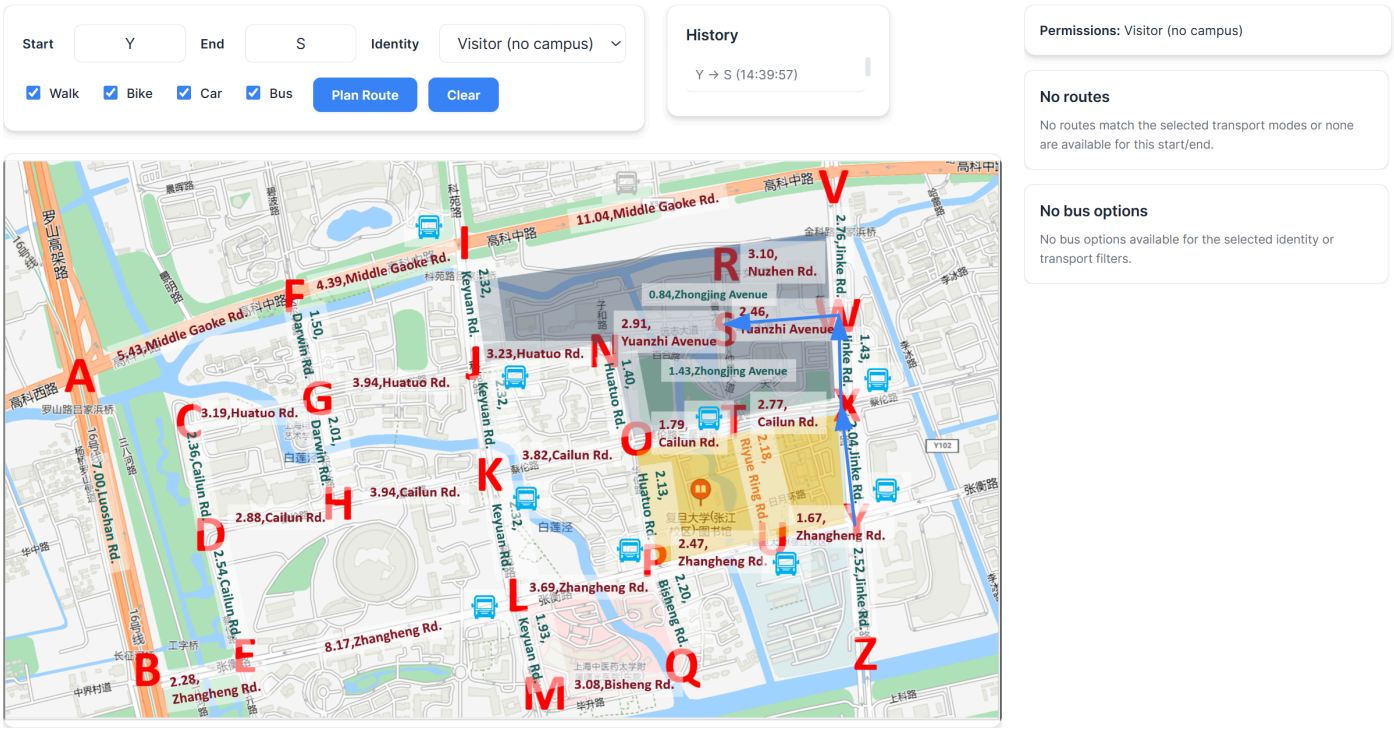
After the user is allowed to enter SHUTCM, the system provides some paths. The following are two of them.



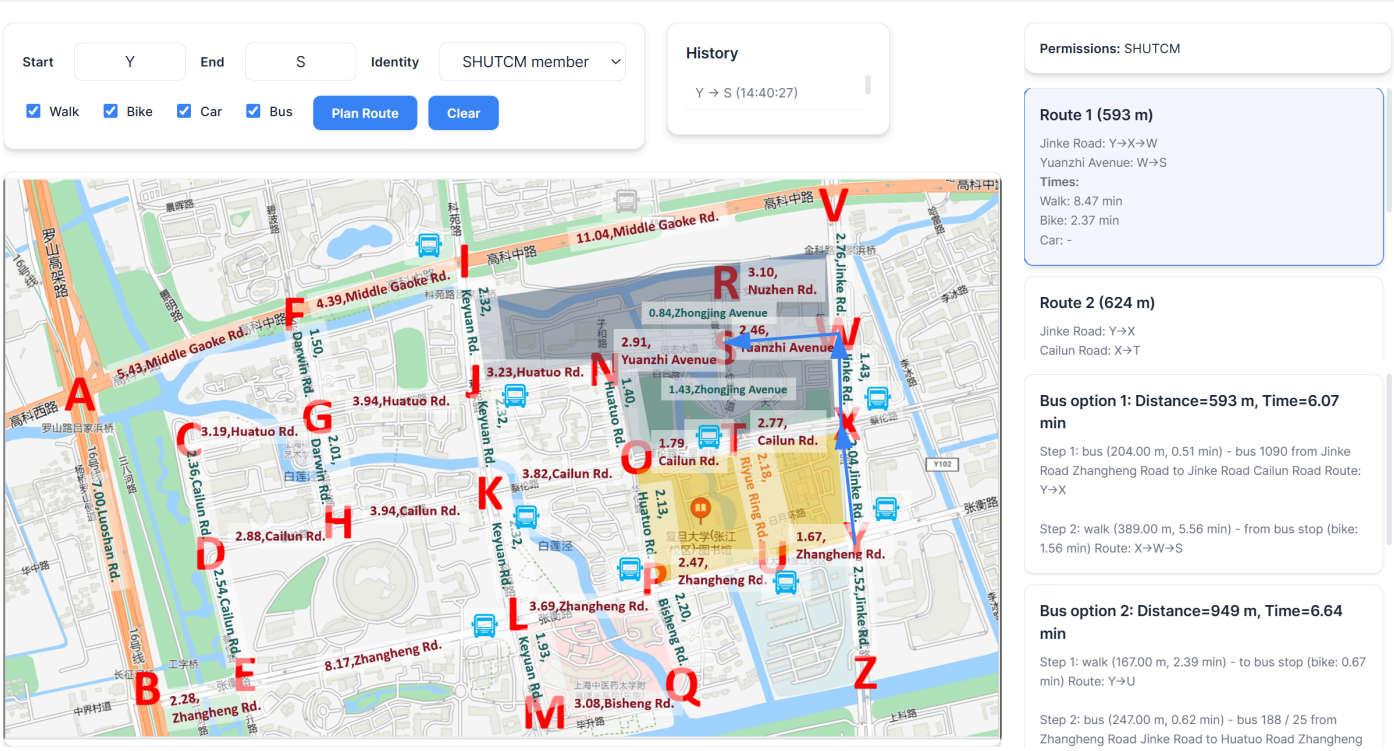


6.4 Example 4: Destination on Campus `Y~>S`

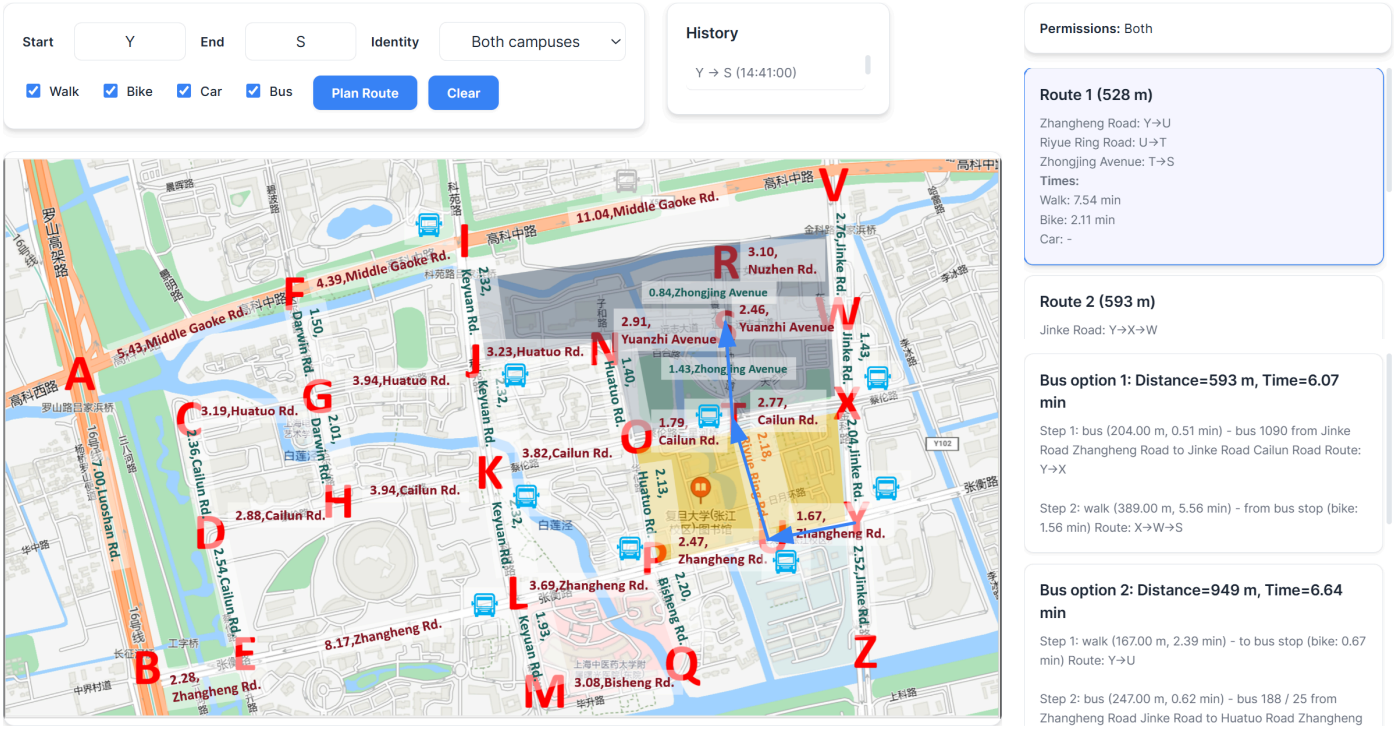
This is similar to example 3. Without permission:



With permission to SHUTCM only:

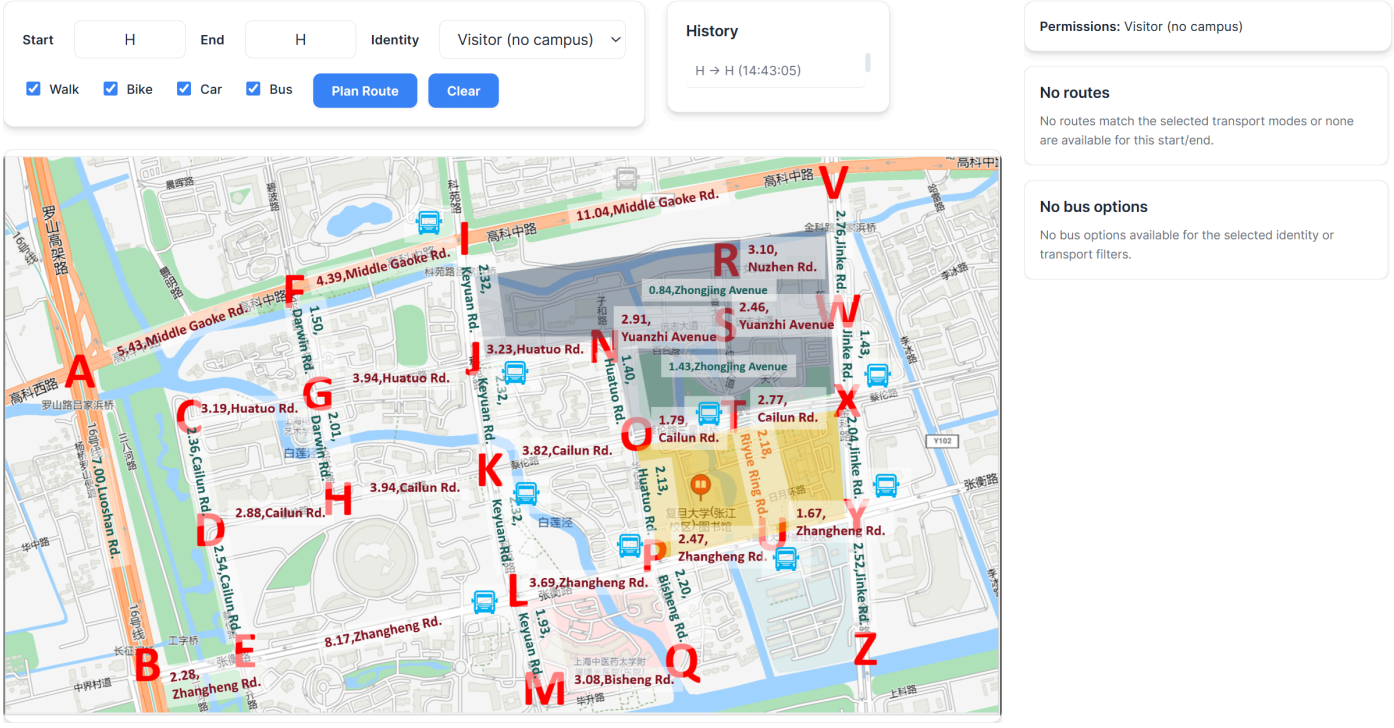


With permission to both campuses:



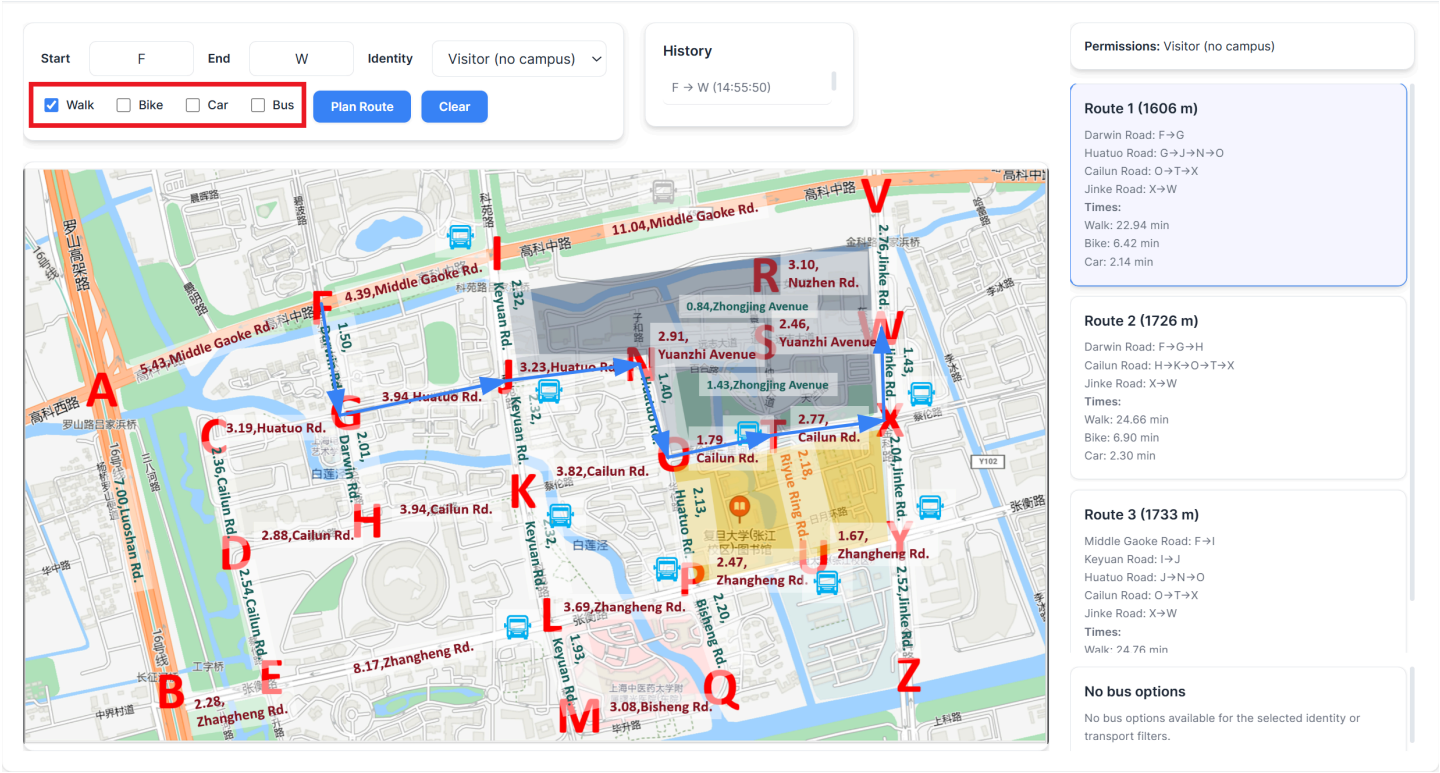
6.5 Example 5: Same Starting and Terminating Points `H~>H`

When the starting and terminating points are the same, the system omits the 0-length path and returns nothing.

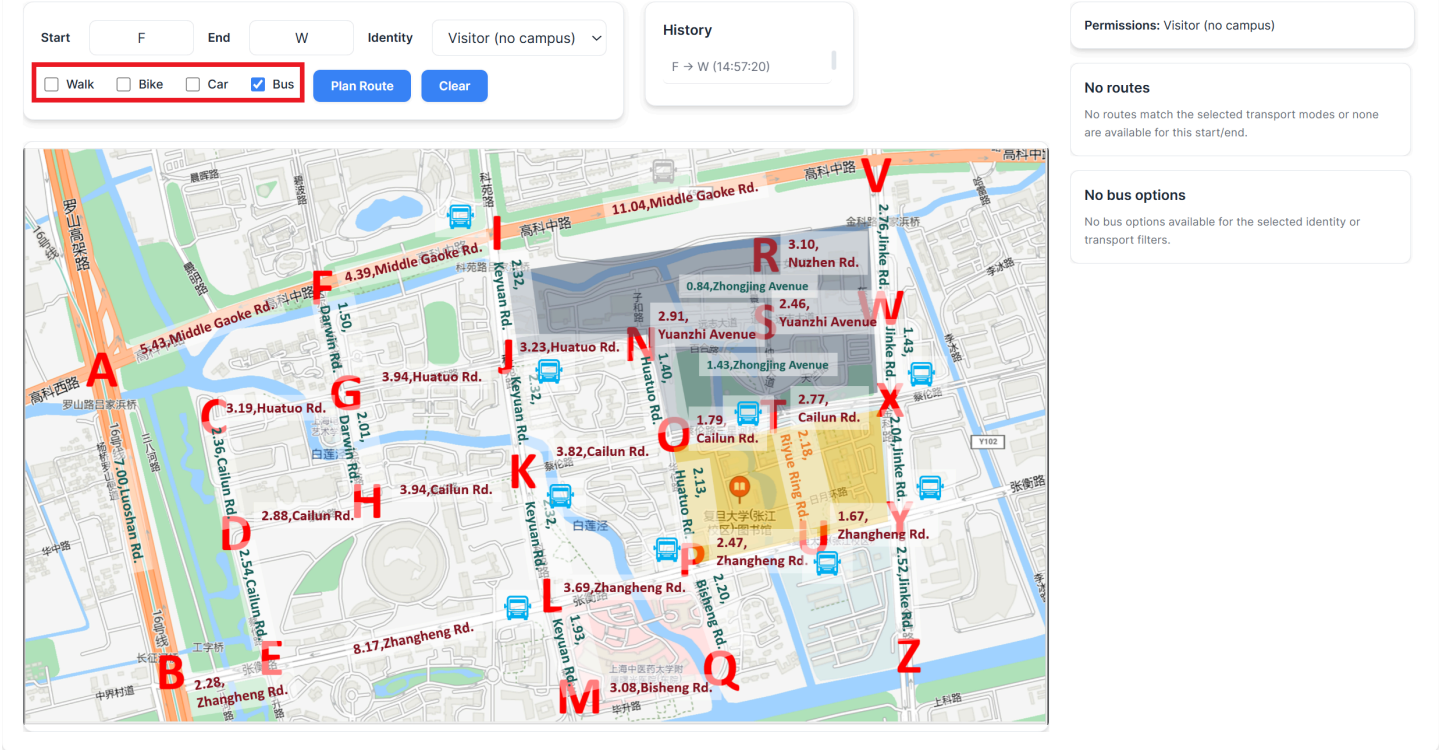


6.6 Example 6: Transport Filters Applied

The user can choose the means of transport that he prefers. In this example, the user prefers walking, so the system provides routes that only involves walking.

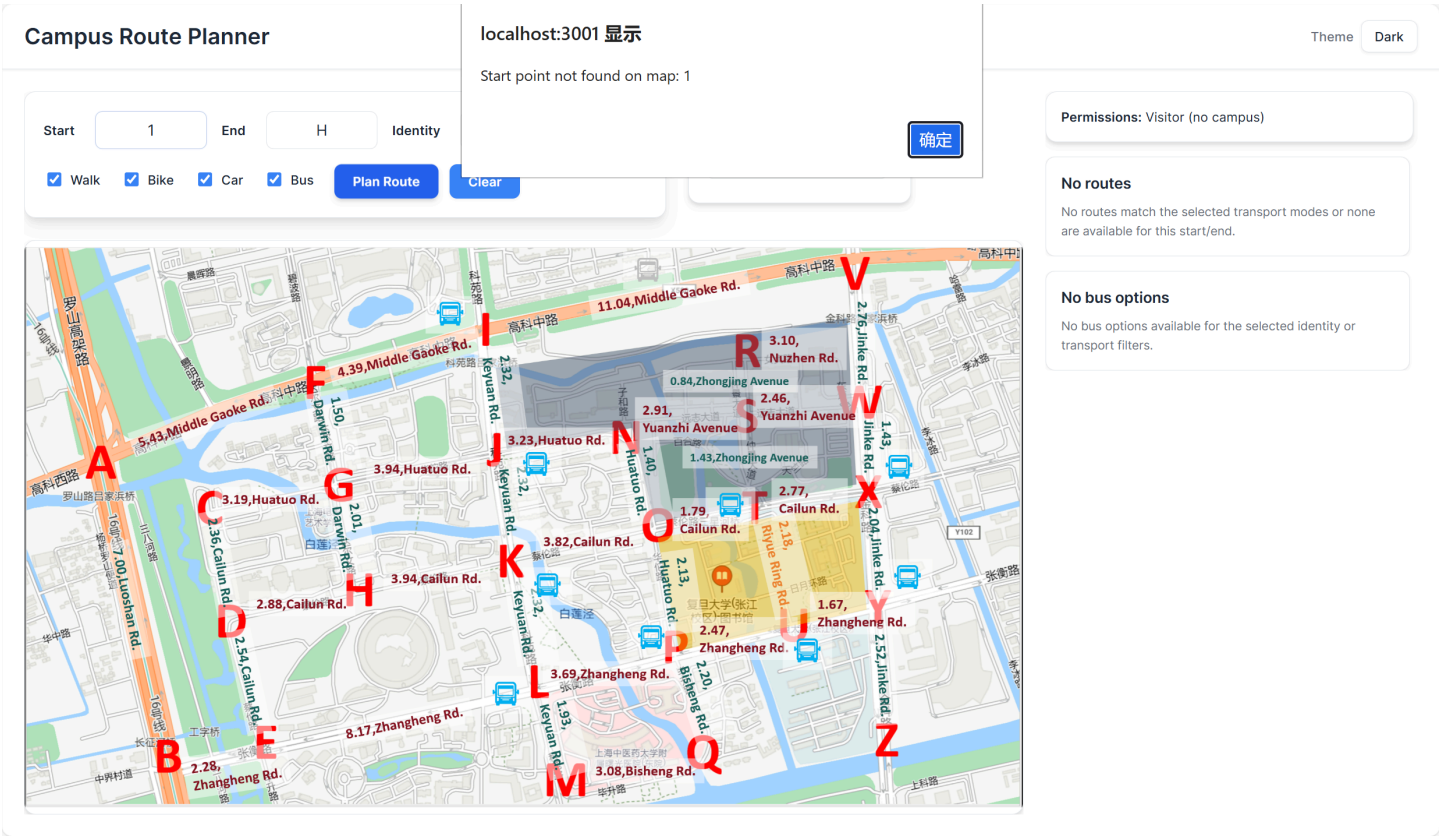


WHen the user only chooses bus, no route is returned because he must walk to the bus stop and then walk back to the destination.

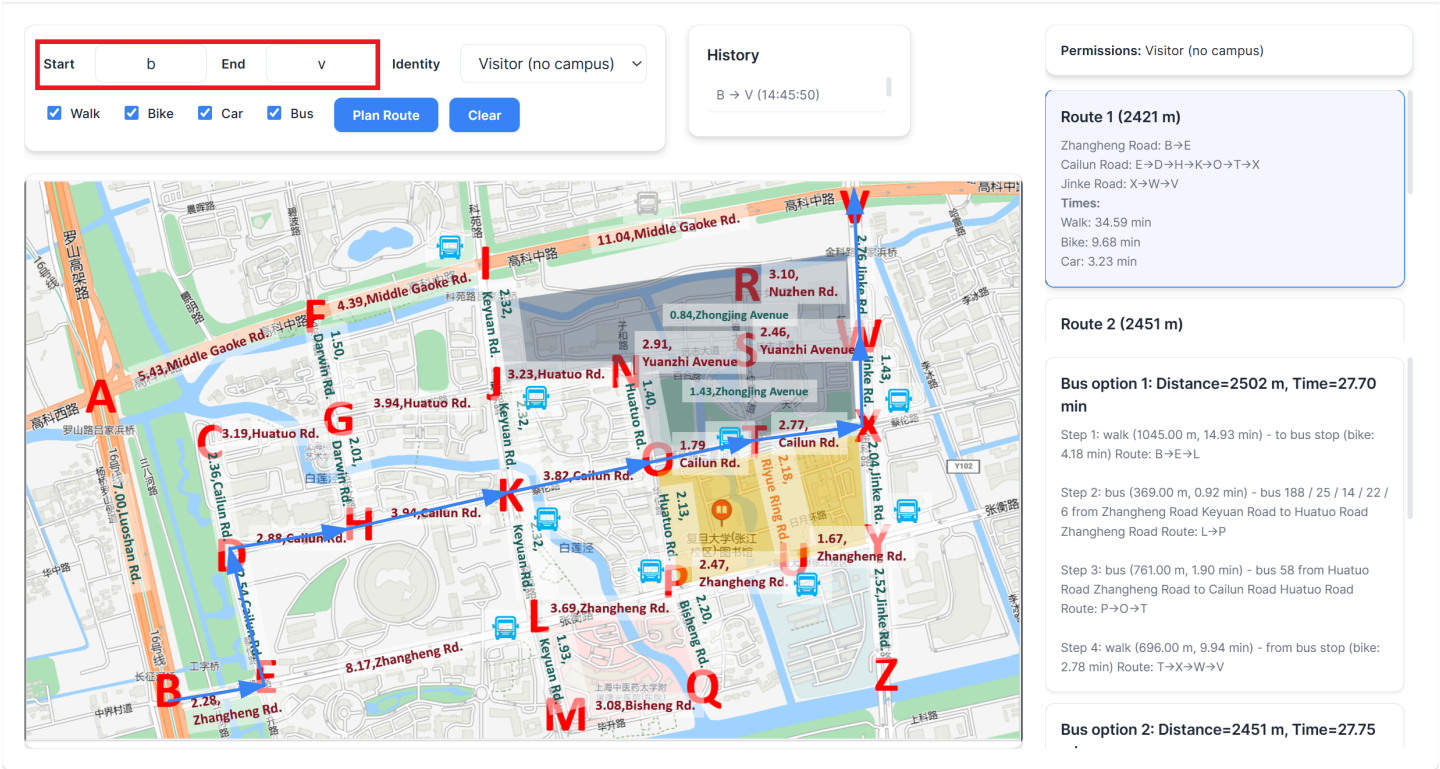


6.7 Example 7: Illegal Requests

If the user inputs unexpected starting or terminating points, the system returns an error message.



Fortunately, the system can accept lower-case letters and interpret them as the corresponding upper-case letters.



7. Discussion

This project is a comprehensive solution for navigation on a small map. However, it has some limitations.

We did not consider the following special scenarios:

- driving a car outside the campus, and then walking inside the campus
- crossing the roads on foot
- waiting for the green light at crossroads
- time for waiting for a bus or transferring to another bus
- bus service hours limits
- single lanes
- traffic jams or accidents

Ignoring these situations does not impact the overall design of this navigating system, but they are quite common in real world.

Since the map is small, it is unnecessary to consider highways or metros. If the map is larger, on the other hand, we can consider adding these features.

In addition, the efficiency of the algorithm is not optimal. The system calculates the desired path with Dijkstra's algorithm and Yen's K-shortest path algorithm after receiving user input. Dijkstra's algorithm takes $O(E \lg V)$ time, and Yen's algorithm takes $O(KVE \lg V)$ time, where V is the number of vertices, and E is the number of edges. The implementation is simple and saves space, but requires large amounts of time searching the graph every time. This delay does not seem much in the small map, but may be a problem in large real-life maps. We might adopt all-pairs shortest paths to solve the problem.

---

8. Conclusion

This project successfully implements a comprehensive navigation system that:

1. Computes shortest paths using Dijkstra's algorithm
2. Finds K shortest paths using Yen's algorithm
3. Supports multiple transportation modes
4. Handles access restrictions
5. Processes real-world map data

---

9. References

1. Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1(1), 269-271.
2. Yen, J. Y. (1971). Finding the K shortest loopless paths in a network. Management Science, 17(11), 712-715.

3. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms (4th ed.). MIT press.
4. Express.js Documentation: https://expressjs.com/
5. Node.js Documentation: https://nodejs.org/en/docs/

---

10. Appendices

10.1 File Structure

```
├── build/              # CMake build directory
├── data/               # Map data files
├── image/              # Map images
├── src/                # C++ source code
│   ├── Graph.cpp
│   ├── Graph.h
│   ├── Parser.cpp
│   ├── Parser.h
│   ├── PathFinder.cpp
│   ├── PathFinder.h
│   └── main.cpp
├── web/                # Web frontend
│   ├── public/
│   │   ├── app.js
│   │   ├── index.html
│   │   └── styles.css
│   ├── server.js
│   └── README.md       # Instruction to run the web server
├── CMakeLists.txt      # CMake configuration
└── Project3.md         # Report
```

10.2 Build Instructions

1. Install CMake and a C++ compiler
2. Create a build directory: `mkdir build && cd build`
3. Run CMake: `cmake ..`
4. Build the project: `cmake --build .`

10.3 Running the Web Server

1. Navigate to the web directory: `cd web`
2. Install dependencies: `npm install`
3. Start the server: `npm start`
4. Open a browser and navigate to `http://localhost:3001`

10.4 Project Specification

Given a small map.
Input: starting point and termination (two locations in the map).
Output: the shortest path from start point to termination and the corresponding routing in detail. At least, you should provide the information about the distance. You can also provide multiple alternative routes with the same distance or slightly longer distances as options. The more you take into account, the more grades you will get.

Document is very important, and you need to explain what you have done, how you implement it and why.

Enjoy it & Good luck!!

Grading

(1) Algorithm and implemented code (60%).
(2) Efficiency of the algorithm (20%).
(3) Document (20%)

---

11. Acknowledgments

This project was completed as part of the CS20009.04 Data Structure course. Special thanks to the course instructors and teaching assistants for their guidance and support. Copilot and trae provided valuable assistance throughout the development process. ChatGPT also assisted me in correcting spelling, improving word choice and sentence clarity, and refining the overall structure of this essay.