

Research on Multi-Objective Urban Pedestrian Path Optimization Model: Integrating Time Efficiency and Greenery

Qihong Wu, Zhang Yang

Shanghai Qingpu World Foreign Language High School, Shanghai, China

Email: alexwu080928@gmail.com

How to cite this paper: Wu, Q.H. and Yang, Z. (2026) Research on Multi-Objective Urban Pedestrian Path Optimization Model: Integrating Time Efficiency and Greenery. *Journal of Transportation Technologies*, 16, 451-480.

<https://doi.org/10.4236/jtts.2026.164023>

Received: September 3, 2026

Accepted: October 26, 2026

Published: October 29, 2026

Copyright © 2026 by author(s) and Scientific Research Publishing Inc.

This work is licensed under the Creative Commons Attribution International License (CC BY 4.0).

<http://creativecommons.org/licenses/by/4.0/>



Open Access

Abstract

Currently, traditional pedestrian navigation path planning focuses solely on minimizing time or distance, neglecting pedestrians' needs for environmental comfort and travel experience. This project integrates time efficiency, path length, and greening index to propose an innovative path planning approach: a multi-objective weighted path planning model. This model abstracts urban road nodes and paths into a weighted graph $G = (V, E)$, under a weighted function. This study integrates time efficiency (segment length and walking speed), waiting time cost (traffic light waiting time), and environmental comfort (greening index based on an inverse distance decay model C_e) into the weights. This study designs three path planning strategies: "highest efficiency", "comfortable walk", and "balanced recommendation". The optimal walking path is solved using Dijkstra's shortest path algorithm. The results show that the model can effectively balance time efficiency and comfort, providing decision support for urban residents to achieve more comfortable, safe, and green walking paths.

Keywords

Graph, Shortest Path Algorithm, Multi-Objective Optimization, Greening Index, Urban Walking

1. Introduction

As the technology develops and the rate of urbanization accelerates, urban spatial architectures and people's travel behavior are changing deeply. In today's city, walking, as one of the most primitive means of transport, not only acts as a medium for transport, but also reflects citizens' perception of environmental quality,

living comfort, and spatial sense. However, most navigation systems' route planning still retains a single objective orientation: for motorized means of transport, road planning is concentrated on driving efficiency, and for pedestrian navigation, walking is mostly focused on the shortest time or shortest distance. This "efficiency-first" planning mode does not take pedestrians' feelings and desires into account in real walking. In reality, when people actually travel on foot, they often weigh speed against environmental comfort. For example, in the hot summer, a street with dense shade, even if it's a bit longer, might be more popular than a shortcut through the scorching sun. Pedestrians may prefer routes with good greenery, fresh air, less noise, or higher safety, rather than simply pursuing the shortest path [1]. Therefore, one-dimensional route planning methods can no longer fully reflect the real travel needs of people in urban spaces.

Therefore, this study decided to use a comprehensive cost function and Dijkstra's shortest path algorithm to achieve multi-dimensional path planning. Based on this, this paper develops a framework applicable to Shanghai, Beijing, and Guangzhou, while the reported case study focuses on Shanghai, and constructs a mathematical model that can quantify and integrate time costs and environmental benefits to achieve pedestrian path planning that is more in line with human-centered principles. The study uses OpenStreetMap (OSM) data, a publicly downloadable map dataset, to construct pedestrian networks for the three cities, and extracts and quantifies the geometric features of roads and environmental factors. Subsequently, through the weighted sum method, multi-dimensional indicators such as time costs and environmental benefits are incorporated into a unified optimization framework to achieve path optimization that balances efficiency and experience. The design of this model aims to provide urban residents with more comfortable, safe, and green pedestrian paths, while also providing data support and decision-making references for urban planners and transportation designers in promoting sustainable mobility and optimizing urban spatial layout.

2. Urban Pedestrian Network and Graph Model Construction

2.1. Definition of Graphical Model

In this study, the urban pedestrian network is abstractly modeled as a weighted directed graph $G = (V, E)$. Here, V represents the set of nodes. Corresponding to road intersections, street corners, or key locations where pedestrians can turn in real urban space; E representing the set of edges, used for describe the roads or walking paths connecting these nodes. Each edge $e = (u, v) \in E$ represents a segment of passageway from node u to node v , and it comes with several attributes reflecting the characteristics of the road. Besides its geometric length L_e , an edge also has one attribute C_e . The degree of roadside greening can be represented by this structured modeling approach C_e . Through this

method, complex urban pedestrian spaces can be quantified and expressed. This provides a clear data foundation for subsequent path optimization. The re-

sulting graph becomes the candidate network for the route computation when downloaded. Routes of interest to pedestrians are the focus of the study within the chosen city.

In order to obtain data, this paper used the Python OSMx library to download high accuracy data of sidewalks for the pedestrian networks in three cities of Shanghai, Beijing, and Guangzhou, from the OpenStreetMap (OSM) service. OSMx automatically downloads and downloads the node and segment information of the road in the city and converts the data to the graph format, maintaining the geometric properties, length, directionality, and other attributes of each of the roads. This method can not only greatly enhance the efficiency and accuracy of data construction, but it cannot be influenced by its subjective error when data collection is conducted manually. Thus, the resulting model inherits strong repeatability and scalability. The pedestrian network used in the reported experiment is extracted for the selected Shanghai study area, and it is completely extracted and converted into a structured directed graph with the help of the automatic processing of the OSMx library, which provides a solid foundation for the weight design, path analysis, and multi-objective optimization in the following parts of the work.

2.2. Quantification of Comfort Indicators (C_e)

To quantify the environmental comfort of pedestrian paths, this paper puts forward a greening index C_e that focuses on the greening aspect. This index takes into account the spatial proximity between the segments of the road and the surrounding green space objects (urban green spaces such as green spaces, grasslands, and woodlands) and corresponds to the environmental comfort of pedestrians when they are walking in the model.

In the data extraction step, all green space objects within the study area (*i.e.* Shanghai) were downloaded from the OpenStreetMap (OSM) data using the `featuresfrompoint()` function of the OSMx library. More specifically, all spatial objects of type `leisure = park`, `landuse = grass`, `landuse = forest`, and `natural = tree`, `natural = treerow` were downloaded, and their types were organized into a set of greening element types, which were stored in nodes:

$$K = \{\text{park, grass, forest, tree, tree_row}\} \quad (1)$$

In the geometric processing step, each edge e of the road was considered as a line segment between node u and node v . The location of the midpoint of this line segment was calculated as Me . The choice of the midpoint was based on the average of the spatial relationship between the road segment and its surrounding environment, and the deviation caused by the difference in road length. In the distance calculation step, for each type of greening object $k \in K$, the shortest distance between the midpoint Me of each road segment to representative points of green features was calculated:

$$d_k(e) = \min_{g \in G_k} \text{Distance}(M_e, g) \quad (2)$$

where, G_k is the set of greening elements of type k . To guarantee accuracy in distance calculations, all the spatial operations were conducted in the Web Mercator (EPSG:3857) coordinate system, and the resultant values were expressed in feet, which provided a unified dimension standard.

In view of the different degrees of influence of different types of greening on pedestrian comfort, a weighting coefficient w_k was introduced in this paper, representing the various kinds of greening elements, and the original cost of greening of the road was defined as follows:

$$C_e = \min_{k \in K} (w_k * d_k(e)) \quad (3)$$

They applied weighting of 1.0, 1.1, 0.9, 1.3, 1.5, and 1.2 for parks, garden areas, forest and wooded areas, grass, trees and tree rows respectively. The weighted nearest-green distance is transformed into a clamped greenery penalty in the implementation. This index captures the distance of the efficient environment under the influence of various types of greening, considering a variety of factors. In cases where no types of greening are mentioned in the above equation around the road, a certain reasonable greening cost was assigned to the road with the help of empirical inference of the road type, and this ensured the computability of the entire model.

The equation captures the idea that “the closer the distance, the better the comfort”. When a road segment is near green space, $dk(e)$ would be smaller, thus resulting in a lower C_e value; however, the value of the index of distant roads is infinity. A lower and an upper bound are imposed to avoid the problem of negative weights or heavy weights caused by the extreme distance, ensuring that the comfort index would have a positive contribution in the path cost function.

Overall, this approach can accurately express the characteristics of green space, which allow the model to comprehensively consider the environmental experience of the pedestrian in the path planning process, and it also provides a calculable greening index that can be used in the multi-objective optimization.

3. Multi-Objective Weighted Cost Function

In this paper, the total cost of a walking path is taken as the sum of the weights of all edges in the path, which is further used to comprehensively describe the “overall travel cost” experienced by pedestrians on the entire path. A path P containing some directed edges, has a total cost which can be written as:

$$C(P) = \sum_{e \in P} W(e) \quad (4)$$

Here, $W(e)$ is the aggregate weight of all edges e in the path, and it is used to summarize the overall performance of the examined road part in terms of its time efficiency, traffic disturbance and environmental experience. More explicitly, the weight of the edge is a weighted combination of three members: time cost, signal interference cost and greening comfort index. In order to more fully describe the characteristics of the path, the composition and calculation methods of their three

weights will be detailed separately as follows.

3.1. Decomposition of Cost Elements

(1) Time Cost $T(e)$

Time cost is used to measure the expected time a pedestrian needs to cross road segment e . Its value is proportional to the geometric length L_e of the road segment and the average walking speed v_{walk} , and can be calculated using the following formula:

$$T(e) = \frac{L_e}{v_{\text{walk}}} \quad (5)$$

where L_e is the segment length (in meters), and $v_{\text{walk}} = 1.3$ m/s is the set constant average walking speed. This value is derived from empirical statistics of urban pedestrian walking speed and can accurately estimate travel time at different street scales.

(2) Signal Interruption Cost $T(s)$

Signal interruption costs reflect the time delays pedestrians experience while waiting at intersections due to traffic lights, and are a significant factor affecting walking efficiency. The formula for calculating this cost is:

$$T(s) = \max(S_{\text{OSM}}(s), S_{\text{infer}}(s), S_{\text{length}}(s)) \quad (6)$$

The traffic light interruption cost $T(s)$ measures the additional waiting impact on pedestrians caused by traffic light delays. This paper considers the traffic light nodes marked in the OpenStreetMap data, S_{OSM} , and counts the number of relevant traffic lights along the roadside. The program combines road connection structure and road grade characteristics, S_{infer} , based on data analysis and empirical inference from OpenStreetMap, to make reasonable inferences about the distribution of traffic lights.

For all signalized nodes we apply a constant time delay of 40 seconds. The routing cost on edges is then determined by the signalization of the two neighboring nodes. In case of no signal information in OSM we guess the signal location using both node degree and road type. Also, the program considers the length of the road to induce the traffic lights number, S_{length} . Last, considering it as a cost, so the program finds the maximum of the three.

By introducing multiple measurement and data extraction methods, this paper improves the feasibility of the model in real urban road networks when facing incomplete data, while making full use of the actual number of traffic lights, thus making the traffic light delay cost more closely reflect the real pedestrian experience [2].

(3) Greening Index $G(e)$

The greening benefit, defined in section 2.2, is a greening index used to describe the inverse impact of the roadside green environment on the walking experience. A lower value indicates a closer proximity of the road to green coverage and a more pleasant walking environment. To align with time costs, this paper converts the

greening index into a time-equivalent quantity, dividing C_e by a constant walking speed v_{walk} , to obtain the comfort benefit over time:

$$G(e) = \frac{C_e}{v_{\text{walk}}} \quad (7)$$

3.2. Comprehensive Cost Function $W(e)$

To ensure that all indicators are weighted and calculated on a uniform scale, this paper performs linear normalization on time cost $T(e)$, signal interruption cost $T(s)$, and greening benefit $G(e)$, limiting their numerical range to the interval $[0, 1]$. The specific calculations are as follows:

$$T''(e) = \frac{T(e) - T(e)_{\min}}{T(e)_{\max} - T(e)_{\min}} \quad (8)$$

$$T''(s) = \frac{T(s) - T(s)_{\min}}{T(s)_{\max} - T(s)_{\min}} \quad (9)$$

$$G'(e) = \frac{G(e) - G(e)_{\min}}{G(e)_{\max} - G(e)_{\min}} \quad (10)$$

where $T(e)$, $T(s)$, $G(e)$ and their corresponding maximum values are the minimum and maximum observed values of each indicator in the scope of the study, respectively. All three indicators are dimensionless and numerically comparable after normalization, so that the reason for estimating the total weight is unbiased because of the difference in dimensions and numerical sizes.

Regarding the model construction, the total cost function $W(e)$ is applied to formulate a weighted sum model to ensure the unified quantification and comparability of the factor in different dimensions. In other words, time cost, signal interruption cost and greening index are “reward items”, and the smaller value is more advantages. All these factors are mathematically represented in direct form with positive weights, in order to fully reflect their positive impact. Hence, the path optimization is more focused on those road sections with strong greening, small relative distance, and low time consumption.

The formula is defined as follows:

$$W(e) = \alpha T'(e) + \beta T'(s) + \gamma G'(e) \quad (11)$$

wherein,

$T'(e)$: represents the time cost of road segment e , considering road length and average walking speed;

$T'(s)$: represents the signal interruption cost of the road segment, which is the time delay caused by waiting for traffic signals at that road segment or intersection;

$G'(e)$: represents the greening benefit of the road segment, with lower values meaning a better environment;

α, β, γ : are the weighting coefficients for time, signal, and greening factors, respectively, which can be customized according to user needs or research focus to

reflect the balance of preferences in different scenarios.

This function will ensure the efficiency and stability of the shortest path algorithms such as Dijkstra's algorithm in the computation process. Based on this processing, the model can adapt flexibly to the runtime characteristics of different optimization algorithms while considering both the mathematical solvability and the practicality of ration.

In summary, this overall cost function will quantitative merger of efficiency (time cost and signal cost) and experience (greening cost and comforting cost) and provide unified and flexible weighting model for the following path optimization. The generated goal path will be more reasonable with the actual decisions of urban pedestrians.

3.3. Optimization Mode and Weight Allocation

To realize the travel preference and actual demands of different kinds of pedestrians, the model is set up for three flexibility optimization modes. By setting and adjusting parameters of weights α , β and γ and so on, the user will be able to select either "efficiency priority" and "comfort priority", in order to generate walking routes with better personal preferences. The three optimization modes and their corresponding weight allocations are summarized in **Table 1**.

Table 1. Weighting comparison of the three travel options.

Mode	Optimization Objective	α (Time)	β (Signal)	γ (Greening)
A	Time Priority	0.85	0.10	0.05
B	Green Priority	0.05	0.05	0.90
C	Balanced	0.50	0.25	0.25

4. Path Optimization Solution and Result Analysis

4.1. Shortest Path Algorithm

In the present study, the walking path optimization problem is transformed to a weighted shortest path problem. That is, given the starting point v_{start} to the ending point v_{end} in a weighted directed graph $G = (V, E)$, the goal is to find a path P in G such that its cumulative sum of edge weights is the lowest, and the mathematical definition is as follows:

$$P^* = \arg \min_P \sum_{e \in P} W(e) \quad (12)$$

Here, $W(e)$ is the aggregate cost of all edges e in the path, and it is used to summarize the overall performance of the examined road part in terms of Its time efficiency, signal disturbance and environmentally comfort value. The goal of the optimization is to get a path that minimizes the "overall travel experience index" so that it achieves the optimal balancing of time and environmentally greening index.

The graph G can be interpreted as a weighted directed graph without negative weight edges. It means that the problem can be solved easily without needing more complexity algorithms (e.g. Bellman-Fordon). Hence, this paper uses the classical Dijkstra's shortest path algorithm to solve the problem and the NetworkX library can automatically calculation the algorithm.

Dijkstra's algorithm is a shortest path search algorithm based on a greedy approach. Its core idea is:

- Start from the starting node $vstart$ and then expand to adjacent nodes;
- At each step, select the unvisited node with the smallest existing "accumulated cost" and note it as the shortest path node;
- Then assign the weights to all edges adjacent to this node;
- Repeat this process until the destination node $vend$ is visited and the shortest path with the smallest cost is found [3] [4].

At the implementation level, the algorithm's input includes:

- Weighted graph G —The pedestrian network of three city extracted and constructed by OSMnx;
- Start and end nodes—Corresponding to user-specified geographic coordinates or points of interest;
- Custom weight function $W(e)$ —Edge weights calculated according to the comprehensive cost model.

The output of the algorithm is the minimum-cost path P^* , which is the path that offers the best user experience under the weight system. This minimum-cost path can be retrieved by calling NetworkOxygen's `nx.shortestpath()` function to retrieve the sequence of path nodes, or be displayed on a map using OSMnx's visualization functionality. Moreover, the added benefits of calculating the maximum path are to ensure the overall performance of the algorithm is computationally efficient and stable while fully exploiting the multi-dimensional weight features established in the model to achieve a high-precision, interpretable, and practical pedestrian path planning.

4.2. Experimental Results and Comparative Analysis

Take two locations in the city (e.g., from the Bund to Nanjing Road Pedestrian Street) as an example for route planning. The comparative results of the three optimization modes are summarized in Table 2.

Table 2. Comparison of the three travel options.

Optimized mode	Distance (m)	Estimated time (min)	Number of traffic light	Green Score (lower = greener)	Traits of routines
A (Efficiency)	4390	71.2	12	2392	The path is the shortest and most linear, but it may pass through busy sections of road.

Continued

B (Comfort)	5600	86.5	22	1721	The distance is slightly longer, but it detours through green spaces, parks, or quiet streets, resulting in a low greening index.
C (Equilibrium)	5330	77.0	13	2269	The approach strikes a balance between time and comfort, resulting in a compromise.

As shown in **Table 2**, the results clearly demonstrate the effectiveness of the model: Mode A minimizes time, while Mode B “sacrifices” some time in exchange for a lower greening index. This proves that the multi-objective cost function can successfully guide the shortest path algorithm to find a non-shortest path that matches user preferences.

The following figures mainly illustrate the feasibility and accuracy of the program. **Figures 1-3** show the routes generated under the efficiency-priority, comfort-priority, and balanced modes, respectively. **Figure 4** shows the shortest route generated by Gaode Maps for comparison with the model results. **Figure 5** provides a direct comparison of all three optimized routes.

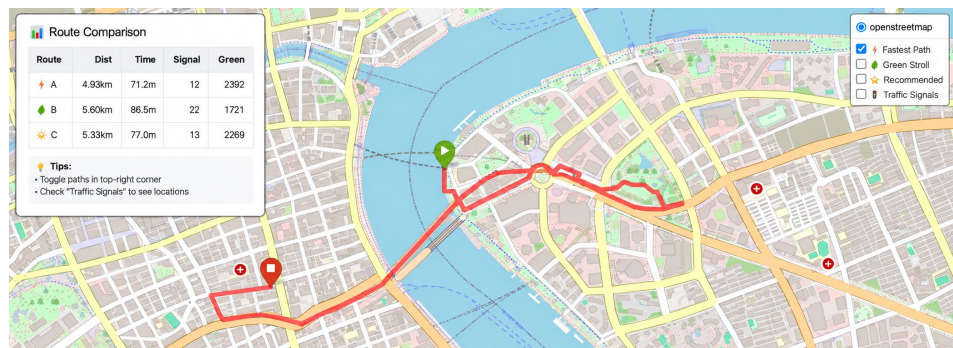


Figure 1. Route planning map of optimization method A from the Bund to Nanjing Road Pedestrian Street.

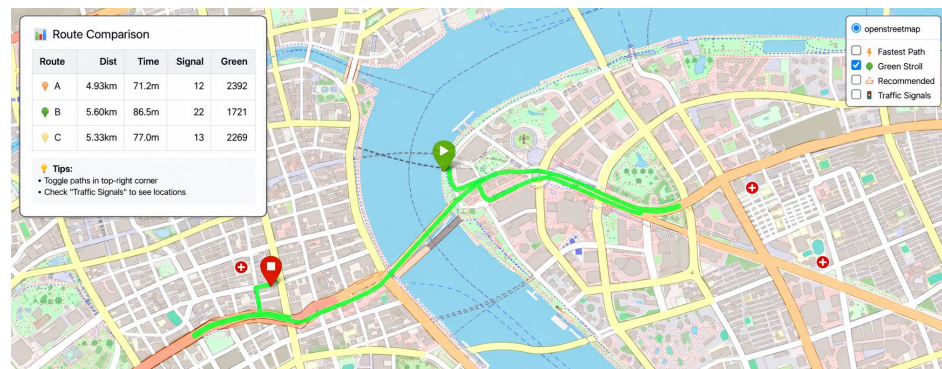


Figure 2. Route planning map of optimization method B from the Bund to Nanjing Road Pedestrian Street.

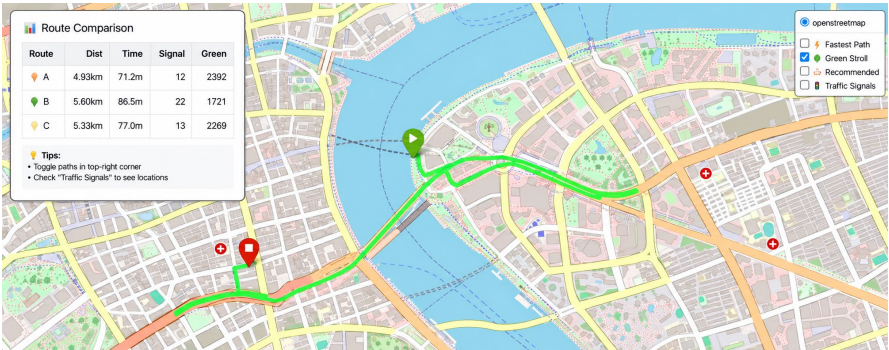


Figure 3. Route planning map of optimization method C from the Bund to Nanjing Road Pedestrian Street.

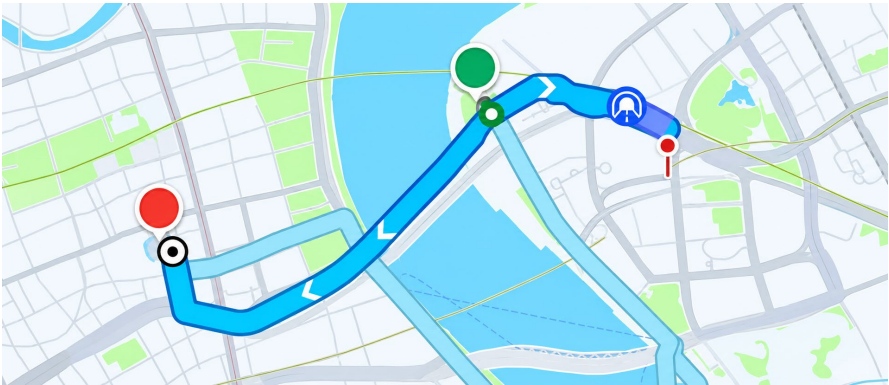


Figure 4. Shortest route from the Bund to Nanjing Road Pedestrian Street in Gaode Map.

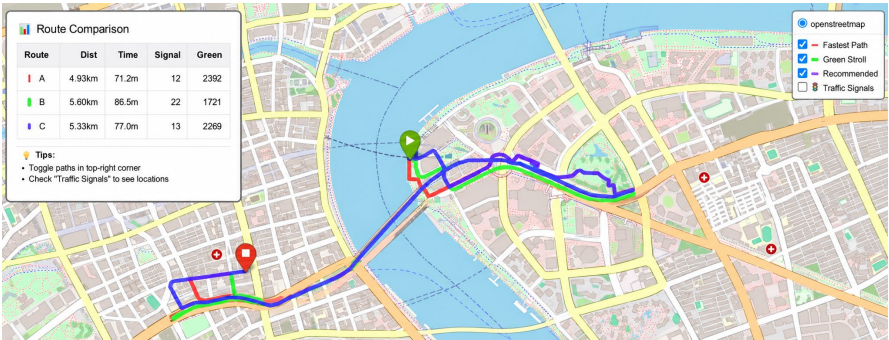


Figure 5. Route planning map of three possible routes from the Bund to Nanjing Road Pedestrian Street.

5. Conclusion and Outlook

This paper successfully constructs an urban pedestrian path optimization model that combines time efficiency and environmental comfort, and implements its computation and application process based on graph theory and the shortest path algorithm. The core innovations of the model are: firstly, by quantifying the environmental experience through a greening index, path planning not only focuses on time costs but also fully reflects pedestrians' subjective feelings about the spatial environment; secondly, through a flexible weighted design of a multi-objective

cost function, personalized adjustments can be made between efficiency and comfort according to different user needs, thus providing pedestrians with more humanized travel solutions.

Although the model has achieved good results at the current stage, there are still several directions for future research:

1. Real-time data integration

The models currently used are mostly based on static road and environment information. The future studies can seek to use the dynamic information, like live weather information, air pollution monitoring, road congestion information, and so forth to make the path planning able to react to the environmental changes in real time, thus making the model more real-time and more practical in the common life traveled situations.

2. A more refined environmental benefit model

The current greening comfort index is mostly based on spatial proximity between roads and green space. For instance, the factors such as the sidewalk width, road slope, noise pollution, and roadway lighting conditions could be taken into account to formulate a more nuanced environment advantage function, thus more fully reflecting the pedestrian experience when walking in practice [5].

3. Considerations for people with disabilities and special needs

In the future, the models can also incorporate constraints such as the route optimization of the wheelchair users, the users with mobility impairments, and the elderly. For instance, some factors like road slope, road surface smoothness, and accessibility facility distribution of roads could be taken into account, in order to create the only “people-centered” approach and make the urban pedestrian environment more inclusive.

6. Multi-Objective Pareto Front Analysis

The model also currently employs a weighted summation method to fully integrate multiple-dimensional indicators into one objective function. In the future, studies can seek to explore the multi-objective optimization approach to create Pareto fronts, thus offering the users a collection of all the possible optimal trade-offs between time efficiency and environmental comfort, not only avoiding the decision-making limitation imposed by the single weight bias, but also giving the users a set of more personal choices, so that the route planning becomes more transparent in practice and easily interpretable.

In conclusion, the model presented in this paper suggests a very useful approach and refers for the future practice of integrating time efficiency and greening index and also serves the basis of the future smart, personalized, and inclusive urban pedestrian navigation system intelligent design.

Author Contributions

Wu Qihong: Writing—original draft, Methodology, Software, Formal analysis, Data curation, Visualization, Investigation, Conceptualization. Yang Zhang: Writ-

ing—review and editing, Supervision, Validation, Project administration. All authors have read and agreed to the published version of the manuscript.

Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

References

- [1] World Health Organization (2025) Promoting Walking and Cycling: A Toolkit of Policy Options. <https://www.who.int/publications/i/item/9789240109902/>
- [2] OpenStreetMap & OSMnx Documentation (2025) Street Network Modeling and Metrics. <https://osmnx.readthedocs.io/>
- [3] NetworkX Documentation (2025) Shortest Path Algorithms. https://networkx.org/documentation/stable/reference/algorithms/shortest_paths.html
- [4] Dijkstra, E.W. (1959) A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, **1**, 269-271. <https://doi.org/10.1007/BF01386390>
- [5] Li, X., Zhang, C., Li, W., Ricard, R., Meng, Q. and Zhang, W. (2015) Assessing Street-level Urban Greenery Using Google Street View and a Modified Green View Index. *Urban Forestry & Urban Greening*, **14**, 675-685. <https://doi.org/10.1016/j.ufug.2015.06.006>

Appendix

☐ Reliable Green Path Planning System - Network Optimized Edition

- ☒ Supports direct coordinate input (bypassing geocoding)
- ☒ Preset coordinates for common locations
- ☒ Network retry and timeout settings
- ☒ Precise road-snapping (using detailed geometry) ""

```
import os
```

```
# Proxy Settings (Replace with your proxy port) os.environ['HTTP_PROXY'] =
'http://127.0.0.1:7890' os.environ['HTTPS_PROXY'] = 'http://127.0.0.1:7890'
```

```
import subprocess import sys
import time import math
```

```
def install_package(package):
    subprocess.check_call([sys.executable, "-m", "pip", "install", package])
```

```
try:
    import osmnx as ox import networkx as nx import folium
    from folium import plugins
    from collections import defaultdict
```

```
print(" ☒ Libraries loaded successfully") except ImportError:
    print("Installing necessary libraries...") for pkg in ["osmnx", "folium"]:
        install_package(pkg) import osmnx as ox import networkx as nx import folium
    from folium import plugins
    from collections import defaultdict
```

```
# === Global Parameters === WALK_SPEED = 1.3
SIGNAL_DELAY = 40
TRAFFIC_LIGHT_THRESHOLD = 30
# ☐ Network Settings
ox.settings.timeout = 300 # Increase timeout to 300s ox.settings.use_cache =
True # Enable cache
```

```
# ☒ Preset Locations (Bypassing geocoding) PRESET_LOCATIONS = {
    "Shanghai": {
        "Jing'an Temple": (31.2245, 121.4452),
        "People's Square": (31.2307, 121.4751),
        "The Bund": (31.2403, 121.4900),
```

```

"Nanjing Road": (31.2349, 121.4810),
"Lujiazui": (31.2395, 121.4995),
"Xintiandi": (31.2196, 121.4748),
"Fudan University": (31.2987, 121.5033),
"Tongji University": (31.2818, 121.5047),
"NECC": (31.1479, 121.3105),
"Hongqiao Railway Station": (31.1949, 121.3203),
"Shanghai Railway Station": (31.2493, 121.4557),
"Yu Garden": (31.2278, 121.4922),
},
"Beijing": {
"Tiananmen": (39.9054, 116.3976),
"Wangfujing": (39.9146, 116.4100),
"Sanlitun": (39.9376, 116.4580),
"Guomao": (39.9084, 116.4584),
"Bird's Nest": (40.0026, 116.3891),
"Forbidden City": (39.9163, 116.3972),
"Summer Palace": (39.9999, 116.2752),
"Peking University": (39.9920, 116.3059),
"Tsinghua University": (40.0031, 116.3261),
},
"Guangzhou": {
"Canton Tower": (23.1052, 113.3191),
"Beijing Road": (23.1291, 113.2644),
"Teemall": (23.1362, 113.3236),
"Shangxiajiu": (23.1193, 113.2501),
}
}

def haversine_distance(lat1, lon1, lat2, lon2):
    """High-performance distance calculation (meters)"""
    R = 6371000
    phi1, phi2 = math.radians(lat1), math.radians(lat2)
    dphi = math.radians(lat2 - lat1)
    dlambda = math.radians(lon2 - lon1)
    a = math.sin(dphi / 2) ** 2 + math.cos(phi1) * math.cos(phi2) * \
        math.sin(dlambda / 2) ** 2
    return R * 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))

def get_location_input(city_name, prompt_text):
    """
    ✓ Smart location input: supports names or coordinates

```

```

"""
print(f"\n{prompt_text}")

if city_name in PRESET_LOCATIONS:
    print(f"\n□ Available locations in {city_name} (type name directly:)") locations
    = list(PRESET_LOCATIONS[city_name].keys())
    for i in range(0, len(locations), 3): row = locations[i:i + 3]
    print(f"{' | '.join(row)}")
    print(f"\n Or enter another location for online search")
    print(f" Or enter coordinates (Format: Lat,Lon e.g., 31.2245,121.4452)")
user_input = input(f"\nPlease enter location: ").strip()
# Check for coordinate format if ',' in user_input:
try:
    lat, lon = map(float, user_input.split(','))
    return (lat, lon), f'Custom({lat:.4f},{lon:.4f})' except:
    print(f"✗ Coordinate format error. Format should be: Latitude,Longitude")
    return None, None

# Check for preset location names
if city_name in PRESET_LOCATIONS and user_input in PRESET_LOCATIONS[city_name]:
    coords = PRESET_LOCATIONS[city_name][user_input]
    print(f"✅ Selected: {user_input} ({coords[0]:.4f}, {coords[1]:.4f})")
    return coords, user_input

# Attempt online geocoding
print(f"\n□ Searching address online: {user_input}") try:
    full_address = f'{user_input}, {city_name}, China' coords = safe_geocode(full_address)
    if coords:
        coords = (coords[0], coords[1])
        print(f"✅ Coordinates found: ({coords[0]:.4f}, {coords[1]:.4f})") return
coords, user_input
except Exception as e: print(f"✗ Online search failed: {e}")
print(f"✗ Location not recognized") return None, None

def safe_geocode(address, retry=3): """
Safe geocoding with retries """
for attempt in range(retry): try:
    result = ox.geocode(address) return result
except AttributeError: try:
    result = ox.geocoder.geocode(address) return result
except: pass

```

```
except Exception as e: if attempt < retry - 1:
    print(f" Retrying {attempt + 1}/{retry}..." ) time.sleep(2)
else:
    raise e return None

def safe_nearest_nodes(G, x, y): try:
    return ox.nearest_nodes(G, x, y) except AttributeError:
    return ox.distance.nearest_nodes(G, x, y)

def download_traffic_signals(center_lat, center_lon, radius): """Download real
traffic signal data"""
    print(f"\n□ Downloading traffic signal data..." ) traffic_lights = []

    try:
        features = ox.features_from_point( (center_lat, center_lon), tags={'highway':
'traffic_signals'}, dist=radius
        )

        if not features.empty:
            for idx, row in features.iterrows():
                geom = row['geometry']

                if geom.geom_type == 'Point': lat, lon = geom.y, geom.x traffic_lights.ap-
pend((lat, lon))

            print(f"✅ Found {len(traffic_lights)} traffic signals") else:
                print(f"□ No traffic signals found in OSM for this area. Using inference
method.") return traffic_lights
    except Exception as e:
        print(f"□ Failed to download signal data: {e}") return []

def download_green_spaces(center_lat, center_lon, radius): """Download real
greenery data"""
    print(f"\n□ Downloading greenery data..." ) green_features = []

    green_tags = {
        'leisure': ['park', 'garden', 'nature_reserve'],
        'landuse': ['grass', 'forest', 'meadow', 'recreation_ground'], 'natural': ['wood',
'tree', 'tree_row']
    }
```

```

try:
    for tag_key, tag_values in green_tags.items():
        for tag_value in tag_values:
            try:
                features = ox.features_from_point( (center_lat, center_lon), tags={tag_key:
tag_value}, dist=radius
                )

            if not features.empty:
                for idx, row in features.iterrows():
                    geom = row['geometry']

                    if geom.geom_type == 'Point':
                        lat, lon = geom.y, geom.x
                    elif geom.geom_type in ['Polygon', 'MultiPolygon']:
                        centroid = geom.centroid
                        lat, lon = centroid.y, centroid.x
                    elif geom.geom_type == 'LineString':
                        coords = list(geom.coords)
                        mid_idx = len(coords) // 2
                        lon, lat = coords[mid_idx]
                    else:
                        continue

                    name = row.get('name', f'{tag_key}={tag_value}')
                    green_features.append((lat, lon, tag_value, name))

            except Exception as e:
                continue
            print(f"✅ Found {len(green_features)} greenery points")
            type_count = defaultdict(int)
            for _, _, gtype, _ in green_features:
                type_count[gtype] += 1

            if type_count:
                print(f"\nGreenery Type Distribution:")
                for gtype, count in sorted(type_count.items(), key=lambda x: -x[1]):
                    print(f"• {gtype}: {count}")
            return green_features
        except Exception as e:
            print(f"❌ Failed to download greenery data: {e}")
            return []

def calculate_green_score_from_features(edge_lat, edge_lon, green_features):
    """Calculate road green score based on real features"""
    if not green_features:
        return 100.0

    min_distance = float('inf')

```

```
for g_lat, g_lon, g_type, _ in green_features:
    dist = haversine_distance(edge_lat, edge_lon, g_lat, g_lon)

    weight_factors = { 'park': 1.0,
                        'garden': 1.1,
                        'forest': 0.9,
                        'wood': 0.9,
                        'grass': 1.3,
                        'tree': 1.5,
                        'tree_row': 1.2,
                        }
    factor = weight_factors.get(g_type, 1.0)

    weighted_dist = dist * factor
    min_distance = min(min_distance, weighted_dist)

    if min_distance < 10: return 5.0
    elif min_distance < 50:
        return 10.0 + (min_distance - 10) * 0.5
    elif min_distance < 150:
        return 30.0 + (min_distance - 50) * 0.7
    else:
        return min(100.0 + (min_distance - 150) * 0.3, 500.0)

def infer_green_score_from_road_type(road_type, road_name): """Infer green
score from road type"""
    type_scores = { 'residential': 30,
                    'service': 40,
                    'footway': 25,
                    'path': 20,
                    'pedestrian': 25,
                    'primary': 80,
                    'secondary': 70,
                    'tertiary': 60,
                    'trunk': 90,
                    'motorway': 100,
                    }
    base_score = type_scores.get(road_type, 50)
    if road_name:
        name_lower = str(road_name).lower()
        if any(word in name_lower for word in ['park', 'garden', 'green', 'park', 'green
area']): base_score *= 0.5
        elif any(word in name_lower for word in ['forest', 'tree', 'woods']): base_score
        *= 0.6
```

```

return base_score

def add_traffic_lights_to_nodes(G, traffic_lights): """Match real signals to road
network nodes""" print(f"\n□ Matching traffic signals to road nodes...")

for node in G.nodes():
    G.nodes[node]['has_traffic_light'] = False

if not traffic_lights:
    print(f"□ No real signal data. Using inference method.") return

matched_count = 0

for tl_lat, tl_lon in traffic_lights: min_dist = float('inf') nearest_node = None

for node in G.nodes():
    node_lat = G.nodes[node]['y'] node_lon = G.nodes[node]['x']

    dist = haversine_distance(tl_lat, tl_lon, node_lat, node_lon)

    if dist < min_dist and dist < TRAFFIC_LIGHT_THRESHOLD: min_dist = dist
    nearest_node = node

    if nearest_node: G.nodes[nearest_node]['has_traffic_light'] = True
    matched_count += 1

print(f"✅ Successfully matched {matched_count} signals to nodes")

total_nodes_with_lights = sum(1 for node in G.nodes() if
G.nodes[node]['has_traffic_light']) print(f" Nodes with traffic lights: {to-
tal_nodes_with_lights}")

def infer_traffic_lights_from_road_type(G): """Fallback: Infer signals from
road type""" print(f"\n□ Using road type to infer traffic signals...")

import random random.seed(42)

for node in G.nodes(): degree = G.degree(node)
edges = list(G.edges(node, data=True))

```

```

if not edges: continue

road_types = []
for u, v, data in edges:
    road_type = data.get('highway', 'residential') if isinstance(road_type, list):
    road_type = road_type[0] road_types.append(road_type)

major_roads = ['primary', 'secondary', 'tertiary', 'trunk'] has_major_road =
any(rt in major_roads for rt in road_types)

if degree >= 4 and has_major_road: probability = 0.6
elif degree >= 4: probability = 0.3
elif has_major_road: probability = 0.15
else:
    probability = 0.02

G.nodes[node]['has_traffic_light'] = random.random() < probability

total_lights = sum(1 for node in G.nodes() if G.nodes[node]['has_traf-
fic_light']) print(f"✅ Inferred {total_lights} potential traffic signal locations")

def add_green_attributes(G, green_features=None, traffic_lights=None):
    """Add green attributes to road network"""
    print(f"\n□ Calculating road greenery scores...") use_real_data = green_fea-
tures and len(green_features) > 0
    if use_real_data:
        print(f" Using real greenery data") else:
        print(f" Using road type inference") processed = 0
    for u, v, key, data in G.edges(keys=True, data=True): try:
        u_node = G.nodes[u] v_node = G.nodes[v]
        edge_lat = (u_node['y'] + v_node['y']) / 2 edge_lon = (u_node['x'] + v_node['x'])
        / 2
    except:
        edge_lat, edge_lon = 0, 0

    if use_real_data and edge_lat != 0:
        green_score = calculate_green_score_from_features( edge_lat, edge_lon,
green_features
        )
    else:
        road_type = data.get('highway', 'residential') if isinstance(road_type, list):
        road_type = road_type[0]

```

```

road_name = data.get('name', '')
green_score = infer_green_score_from_road_type(road_type, road_name)
data['green_score'] = green_score
processed += 1
if processed % 500 == 0:
    print(f"Processed {processed} roads...", end='\r')
    print(f"\n✅ Finished greenery calculation for {processed} roads") if traf-
fic_lights and len(traffic_lights) > 0:
        add_traffic_lights_to_nodes(G, traffic_lights) else:
            infer_traffic_lights_from_road_type(G)
green_scores = [data.get('green_score', 100) for _, _, data in
G.edges(data=True)] if green_scores:
    print(f"\n□ Statistics:") print(f"Greenery Scores:")
    print(f" Average: {sum(green_scores) / len(green_scores):.1f}") print(f" Range:
{min(green_scores):.1f} ~ {max(green_scores):.1f}")

```

```

def normalize(value, min_val, max_val): if max_val == min_val:
return 0.5
return max(0, min(1, (value - min_val) / (max_val - min_val)))

```

```

def count_traffic_lights_on_route(G, route): """

```

```

    ✅ Improved signal counting
    """

```

```

total_signals = 0

```

```

# Method 1: Count marked signals marked_signals = 0

```

```

for node in route:

```

```

    if G.nodes[node].get('has_traffic_light', False): marked_signals += 1

```

```

# Method 2: Inferred based on intersections and road type inferred_signals = 0

```

```

for node in route[1:-1]: degree = G.degree(node)

```

```

edges = list(G.edges(node, data=True)) road_types = []

```

```

for u, v, data in edges:

```

```

    road_type = data.get('highway', 'residential') if isinstance(road_type, list):

```

```

    road_type = road_type[0] road_types.append(road_type)

```

```

major_roads = ['primary', 'secondary', 'tertiary', 'trunk'] has_major_road =
any(rt in major_roads for rt in road_types)

```

```

if degree >= 4 and has_major_road: inferred_signals += 1

```

```

elif degree >= 3 and has_major_road: inferred_signals += 0.7

# Method 3: Estimated based on length total_length = 0
major_road_length = 0

for u, v in zip(route[:-1], route[1:]): edge_data = G[u][v][0]
length = edge_data.get('length', 0) total_length += length

road_type = edge_data.get('highway', 'residential') if isinstance(road_type, list):
road_type = road_type[0]

if road_type in ['primary', 'secondary', 'tertiary', 'trunk']: major_road_length +=
length
length_based_signals = major_road_length / 500 total_signals = max(
marked_signals,
int(inferred_signals), int(length_based_signals)
)

if total_signals < 5 and total_length > 3000:
total_signals = max(total_signals, int(total_length / 1000 * 2))
return total_signals

def calculate_route_with_mode(G, start_node, end_node, mode='C'): """Calcu-
late optimal path for specific mode"""
time_vals, green_vals = [], []

for _, _, data in G.edges(data=True): length = data.get('length', 100) green =
data.get('green_score', 50)

time_vals.append(length / WALK_SPEED) green_vals.append(green)

t_min, t_max = min(time_vals), max(time_vals) s_min, s_max = 0, SIG-
NAL_DELAY
g_min, g_max = min(green_vals), max(green_vals)

weights = {
'A': (0.85, 0.10, 0.05),
'B': (0.05, 0.05, 0.90),
'C': (0.50, 0.25, 0.25)
}
alpha, beta, gamma = weights[mode]

```

```

    for u, v, key, data in G.edges(keys=True, data=True): length = data.get('length',
100)
    green = data.get('green_score', 50)

    u_signal = SIGNAL_DELAY if G.nodes[u].get('has_traffic_light', False) else 0
    v_signal = SIGNAL_DELAY if G.nodes[v].get('has_traffic_light', False) else 0
    avg_signal = (u_signal + v_signal) / 2

    t_norm = normalize(length / WALK_SPEED, t_min, t_max) s_norm = normal-
ize(avg_signal, s_min, s_max)
    g_norm = normalize(green, g_min, g_max)

    cost = max(alpha * t_norm + beta * s_norm + gamma * g_norm, 0.001)
    data[f'cost_{mode}'] = cost
    route = nx.shortest_path(G, start_node, end_node, weight=f'cost_{mode}') to-
tal_length = 0
    total_green = 0
    total_time = 0

    for u, v in zip(route[:-1], route[1:]):
        edge_data = G[u][v][0]
        length = edge_data.get('length', 0) green = edge_data.get('green_score', 0)

        total_length += length total_green += green
        total_time += length / WALK_SPEED

    total_signals = count_traffic_lights_on_route(G, route) total_time += to-
tal_signals * SIGNAL_DELAY

    return {
        'route': route, 'length': total_length,
        'signals': total_signals, 'green': total_green, 'time': total_time, 'nodes': len(route)
    }

def create_advanced_route_map(G, G_detailed, results, start_node, end_node,
traffic_lights=None, output_file='route_map.html'):
    """Create advanced interactive map with detailed geometry""" print(f'\n□ Gen-
erating advanced interactive map...")

    start_lat = G.nodes[start_node]['y'] start_lon = G.nodes[start_node]['x']

    m = folium.Map( location=[start_lat, start_lon], zoom_start=14, tiles='Open-
```

```

StreetMap'
)

route_configs = { 'A': {
'color': '#FF4444',
'weight': 6,
'opacity': 0.8, 'dash_array': None, 'name': ' ⚡ Fastest Path', 'icon': ' ⚡ ',
}, 'B': {
'color': '#44FF44',
'weight': 6,
'opacity': 0.8,
'dash_array': '10, 5', 'name': ' □ Green Stroll', 'icon': ' □ ',
},
'C': {
'color': '#4444FF',
'weight': 6,
'opacity': 0.8,
'dash_array': '5, 5, 2, 5', 'name': ' □ Recommended', 'icon': ' □ ',
}
}

for mode, result in results.items(): route = result['route']
config = route_configs[mode]

feature_group = folium.FeatureGroup(name=config['name'], show=True)
route_coords = []
for i in range(len(route) - 1): u, v = route[i], route[i + 1]

u_lat, u_lon = G.nodes[u]['y'], G.nodes[u]['x']
v_lat, v_lon = G.nodes[v]['y'], G.nodes[v]['x']

try:
u_detailed = safe_nearest_nodes(G_detailed, u_lon, u_lat) v_detailed =
safe_nearest_nodes(G_detailed, v_lon, v_lat)

if nx.has_path(G_detailed, u_detailed, v_detailed):
detailed_path = nx.shortest_path(G_detailed, u_detailed, v_detailed,
weight='length')

for j in range(len(detailed_path) - 1):
du, dv = detailed_path[j], detailed_path[j + 1]

edge_data = None

```

```

if G_detailed.has_edge(du, dv): edge_data = G_detailed[du][dv][0]
elif G_detailed.has_edge(dv, du): edge_data = G_detailed[dv][du][0]

if edge_data and 'geometry' in edge_data: geom = edge_data['geometry']
coords = [(lat, lon) for lon, lat in geom.coords]
if G_detailed.has_edge(dv, du) and not G_detailed.has_edge(du, dv):

else:

else:
if j == 0:
route_coords.append([G_detailed.nodes[du]['y'], G_detailed.nodes[du]['x']])
route_coords.append([G_detailed.nodes[dv]['y'], G_detailed.nodes[dv]['x']])
if i == 0:
route_coords.append([u_lat, u_lon]) route_coords.append([v_lat, v_lon])

except Exception as e: if i == 0:
route_coords.append([u_lat, u_lon]) route_coords.append([v_lat, v_lon])

route_line = folium.PolyLine( route_coords, color=config['color'],
weight=config['weight'], opacity=config['opacity'], dash_array=config['dash_ar-
ray'], popup=folium.Popup(
f"<div style='font-size:14px; min-width:200px>" f"<b>{con-
fig['name']}</b><br>" f"<b>Distance:</b> {result['length']:.0f}m<br>"
f"<b>Time:</b> {result['time'] / 60:.1f} min<br>" f"<b>Signals:</b> {result['sig-
nals']} □<br>" f"<b>Green Score:</b> {result['green']:.0f}<br>" f"</div>",
max_width=300
),
tooltip=f"{config['icon']} {config['name']}"
)

route_line.add_to(feature_group) feature_group.add_to(m)

if traffic_lights and len(traffic_lights) > 0:
traffic_light_group = folium.FeatureGroup(name='□ Traffic Signals',
show=False)

for tl_lat, tl_lon in traffic_lights: folium.CircleMarker(
[tl_lat, tl_lon], radius=4, color='red', fill=True, fillColor='yellow',
fillOpacity=0.8, popup='□ Traffic Signal', tooltip='Traffic Signal'
).add_to(traffic_light_group) traffic_light_group.add_to(m)
folium.Marker( [start_lat, start_lon], popup='<b>Start</b>',
icon=folium.Icon(color='green', icon='play', prefix='fa'), tooltip='Start'

```

```

).add_to(m)

end_lat = G.nodes[end_node]['y'] end_lon = G.nodes[end_node]['x'] fo-
lium.Marker(
    [end_lat, end_lon], popup='<b>End</b>',
    icon=folium.Icon(color='red', icon='stop', prefix='fa'), tooltip='End'
).add_to(m)

folium.LayerControl( position='topright', collapsed=False
).add_to(m)

comparison_html = f"""
<div style="position: fixed; top: 10px;
left: 60px; width: 320px;
background-color: white; border: 2px solid grey; border-radius: 8px;
z-index: 9999; padding: 15px;
box-shadow: 0 4px 6px rgba(0,0,0,0.1);">

<h4 style="margin: 0 0 10px 0; color: #333;">□ Route Comparison</h4>

<table style="width: 100%; border-collapse: collapse; font-size: 12px;">
<tr style="background-color: #f0f0f0; font-weight: bold;">
<th style="padding: 8px; text-align: left; border: 1px solid #ddd;">Route</th>
<th style="padding: 8px; text-align: right; border: 1px solid #ddd;">Dist</th>
<th style="padding: 8px; text-align: right; border: 1px solid #ddd;">Time</th>
<th style="padding: 8px; text-align: right; border: 1px solid #ddd;">Signal</th>
<th style="padding: 8px; text-align: right; border: 1px solid #ddd;">Green</th>
</tr>
"""

for mode in ['A', 'B', 'C']: result = results[mode]
config = route_configs[mode]

comparison_html += f"""
<tr>
<td style="padding: 8px; border: 1px solid #ddd;">
<span style="color: {config['color']}; font-weight: bold;">{con-
fig['icon']}</span>
{mode}
</td>
<td style="padding: 8px; text-align: right; border: 1px solid #ddd;">
{result['length'] / 1000:.2f}km
</td>

```

```

<td style="padding: 8px; text-align: right; border: 1px solid #ddd;">
{result['time'] / 60:.1f}m
</td>
<td style="padding: 8px; text-align: right; border: 1px solid #ddd;">
{result['signals']}
</td>
<td style="padding: 8px; text-align: right; border: 1px solid #ddd;">
{result['green']:.0f}
</td>
</tr> """

comparison_html += """
</table>

<div style="margin-top: 10px; padding: 8px; background-color: #f9f9f9; border-radius: 4px; font-size: 11px;">
<b>□ Tips:</b><br>
• Toggle paths in top-right corner<br>
• Check "Traffic Signals" to see locations
</div>
</div> """

m.get_root().html.add_child(folium.Element(comparison_html)) m.save(output_file)
abs_path = os.path.abspath(output_file)

print(f"✅ Map saved: {abs_path}")
return abs_path

def analyze_route_differences(results): """Analyze route differences"""
print("\n" + "=" * 70)
print("□ Detailed Route Analysis") print("=" * 70)

routes_unique = len(set(tuple(r['route']) for r in results.values()))
print(f"\nGenerated {routes_unique} unique routes")

if results['A']['route'] != results['B']['route']:
diff_dist = results['B']['length'] - results['A']['length']
diff_green = results['A']['green'] - results['B']['green']
diff_time = (results['B']['time'] - results['A']['time']) / 60
diff_signals = results['B']['signals'] - results['A']['signals']

print(f"\n□ Fastest vs Green Stroll:")

```

```

    print(f" Distance Diff: +{diff_dist:.0f}m ({diff_dist / results['A']['length'] *
100:.1f}%)") print(f" Time Diff: +{diff_time:.1f} minutes")
    print(f" Signal Diff: {diff_signals:+d} □")
    print(f" Greenery Improvement: {diff_green:.0f} pts ({diff_green / re-
sults['A']['green'] * 100:.1f}%)")

    if diff_time < 5 and diff_green > 0:
        print(f"\n□ Recommendation: Green Stroll only takes {diff_time:.1f} more
minutes for much better greenery!")
    else:
        print(f"\n□ Fastest and Green Stroll modes chose the same route")

def main():
    print("\n" + "=" * 70)
    print("□ Smart Green Route Planner (Network Optimized)") print("=" * 70)
    print("\n□ Preset coordinates supported, no online search needed!")
    print("\nSelect city:")
    print(" 1. Shanghai")
    print(" 2. Beijing")
    print(" 3. Guangzhou")

    city_choice = input("\nChoice (1-3): ").strip() city_map = {
    '1': 'Shanghai',
    '2': 'Beijing',
    '3': 'Guangzhou'
    }
    city_name = city_map.get(city_choice, 'Shanghai') while True:
        print("\n" + "-" * 70)

        start_coords, start_name = get_location_input(city_name, "[START]") if not
start_coords:
            print("❌ Invalid start point")
            continue

        end_coords, end_name = get_location_input(city_name, "[END]") if not
end_coords:
            print("❌ Invalid end point")
            continue

        try:
            start_lat, start_lon = start_coords end_lat, end_lon = end_coords

```

```

print(f'\n✅ START:    {start_name}    ({start_lat:.4f},    {start_lon:.4f})")
print(f'\n✅ END: {end_name} ({end_lat:.4f}, {end_lon:.4f})")

distance = haversine_distance(start_lat, start_lon, end_lat, end_lon)
print(f'\nStraight-line Distance: {distance:.0f} m ({distance / 1000:.2f} km)")

if distance < 300:
    print("❑ Distance too short, try choosing further locations") continue
if distance > 15000:
    print("❑ Distance very long, download might take some time") continue

center_lat = (start_lat + end_lat) / 2 center_lon = (start_lon + end_lon) / 2
download_dist = min(distance * 0.8 + 1500, 6000)

print(f'\n❑ Downloading network (Radius {download_dist:.0f}m)...") print(f"
This might take 30-60 seconds, please wait...")
start_time = time.time()

print(f" Processing road network data...") G_simple = ox.graph_from_point(
(center_lat, center_lon), dist=download_dist, network_type='all',
simplify=True
)

G_detailed = ox.graph_from_point( (center_lat, center_lon), dist=down-
load_dist, network_type='all', simplify=False
)

G = G_simple

print(f'\n✅ Network downloaded (Time: {time.time() - start_time:.1f}s)")
print(f' {len(G.nodes)} nodes, {len(G.edges)} edges")

traffic_lights = download_traffic_signals(center_lat, center_lon, down-
load_dist) green_features = download_green_spaces(center_lat, center_lon,
download_dist)

start_node = safe_nearest_nodes(G, start_lon, start_lat) end_node = safe_near-
est_nodes(G, end_lon, end_lat)

if not nx.has_path(G, start_node, end_node): print("❌ Start and end are not
connected") continue
add_green_attributes(G, green_features, traffic_lights) print("\n" + "=" * 70)
print("❑ Route Planning Results")

```

```

print("=" * 70)

results = {} mode_names = {
'A': ' ⚡ Fastest Path',
'B': ' □ Green Stroll', 'C': ' □ Recommended'
}

for mode in ['A', 'B', 'C']:
    result = calculate_route_with_mode(G, start_node, end_node, mode) re-
sults[mode] = result

    print(f"\n{mode_names[mode]}:")
    print(f" Distance: {result['length']:.0f} m ({result['length'] / 1000:.2f} km)")
print(f" Estimated Time: {result['time'] / 60:.1f} minutes")
    print(f" Signals: {result['signals']} □")
    print(f" Green Score: {result['green']:.0f} (lower is greener)")
    analyze_route_differences(results)

    map_file = create_advanced_route_map(G, G_detailed, results, start_node,
end_node, traffic_lights)

    try:
import webbrowser
    print(f"\n □ Opening map in browser...") webbrowser.open('file://' + map_file)
    except:
    print(f"\n □ Please open manually: {map_file}")

    print("\n" + "=" * 70) print(" ✅ Planning complete!") print("=" * 70)

    print("\n1=Continue 2=Exit")
    if input("Choice: ").strip() == '2': return

    except Exception as e: print(f"\n ❌ Error: {e}") import traceback trace-
back.print_exc() print("\n1=Retry 2=Exit")
    if input("Choice: ").strip() == '2': return

if __name__ == "__main__": try:
main()
except KeyboardInterrupt: print("\n\n □ User interrupted")
except Exception as e: print(f"\n\n ❌ Program error: {e}")

```