



A Non-Greedy Spatially Indexed Route-Finding Algorithm for Public Transit in Kathmandu Valley

Sohan Mehta^{1*}, Sandip Pandey², Santosh Kumar Yadav³, Shrisha Bhatta⁴, Er. Anup Shrestha⁵

¹ Dept. of Electronics and Computer Engineering, Thapathali Campus, Nepal

Email: sosohan404@gmail.com

² Dept. of Electronics and Computer Engineering, Thapathali Campus, Nepal

Email: sandipvai4456@gmail.com

³ Dept. of Electronics and Computer Engineering, Thapathali Campus, Nepal

Email: santoshy7284@gmail.com

⁴ Dept. of Electronics and Computer Engineering, Thapathali Campus, Nepal

Email: bhattashreeshal2@gmail.com

⁵ Asst. Professor, Dept. of Electronics and Computer Engineering, Thapathali Campus, Nepal

Email: anup@tcioe.edu.np

Original Article

Correspondence Author*

Received: Jul 09, 2026

Revised & Accepted: Aug 24, 2026

Copyright: Author(s) (2026)



This work is licensed under a Creative Commons Attribution-Non-Commercial
4.0 International License

ABSTRACT

Navigating informal public transit networks in developing urban regions presents unique challenges absent from systems served by established platforms such as Google Maps. In the Kathmandu Valley—encompassing Kathmandu, Lalitpur, and Bhaktapur—public transportation operates across heterogeneous route topologies including linear bidirectional, circular unidirectional (e.g., Ring Road), and hybrid “lollipop” configurations, none of which are indexed by mainstream navigation services. This paper presents a client-side, offline-first route finding



system that combines spatial grid indexing with a non-greedy, penalty-weighted search algorithm specifically engineered for fixed-route transit networks. The system partitions the geographical space into $500 \text{ m} \times 500 \text{ m}$ grid cells, enabling amortized $O(1)$ stop lookups, and employs a graduated radius search strategy to identify candidate boarding and a lighting stop while accounting for multi-lane road configurations. A weighted scoring function $S = 40T + t + 12W$ balances transfer penalties (T), total travel time (t), and walking distance (W) to rank route options. The algorithm handles both direct and single-transfer routes, with directional constraints on transfer candidate selection to prevent incorrect path selection on circular routes. Performance benchmarking on a physically collected dataset of 55 routes and over 600 GPS-mapped bus stops demonstrates direct route resolution in 0.15 ms and transfer route resolution in 5.37 ms, well within the 200 ms real-time threshold. Scalability experiments on a 5,000-stop network simulation confirm that performance scales sub-linearly with network size. The system is deployed as a cross-platform mobile application, serving an active user base in the Kathmandu Valley. This work contributes a domain-specific alternative to graph-based shortest-path algorithms for transit networks where fixed-route constraints, topology heterogeneity, and infrastructure data scarcity render classical approaches impractical.

Index Terms—Informal transit networks, non-greedy route optimization, public transportation, route finding algorithm, smart city navigation, spatial grid indexing

I. INTRODUCTION

Public transportation systems in developing urban areas often lack centralized scheduling, standardized stop infrastructure, or digital route databases. An example of this challenge is the Kathmandu Valley—the political, economic and cultural hub of Nepal made up of the districts of Kathmandu, Lalitpur and Bhaktapur. The Valley receives more than one million annual tourists [5] and experiences continuous internal migration for education and employment, making the Valley’s public transit network an essential, yet under documented piece of infrastructure.

In cities such as London, Tokyo, or Shenzhen, transit agencies publish General Transit Feed Specification (GTFS) data that is consumed by platforms like Google Maps and Citymapper, but there is no standard way in which Kathmandu’s bus network is digitally codified. The routes run by agencies like Mahanagar Yatayat, Sajha Yatayat and many private operators have different topological forms: linear bidirectional routes where buses turn around at terminals, circular unidirectional loops (like the Ring Road services), and hybrid “lollipop” routes where a linear stem terminates in a loop. None of the current navigation platforms provide route-by-route navigation for these services.

The issue of the lack of information creates several problems for both tourists and locals. For instance, 85% of foreign tourists interviewed in Nepal stated that they would benefit from a public



transport route finding application which provides information on fares and reviews [6]. Thus, without proper information on the public transport system, tourists end up getting unsuitable routes which increases the probability of them being ripped-off by dishonest drivers.

Existing approaches to public transit route finding are based on graph algorithms such as Dijkstra's algorithm [2], A* [3], or its variations. These solutions, although successful in generic road networks where drivers can freely choose their routes, are inadequate for modeling a fixed-route system. Building a graph representation requires encoding all possible combinations of routes, stops, and transfers, generating unnecessarily large graphs that are impractical to maintain for systems that do not provide their GTFS feed. Furthermore, shortest-path algorithms usually find an optimal path according to a single metric, and do not naturally favor connections with specific properties such as minimizing transfers or avoiding walking. This paper introduces a route finding system for informal fixed-route transit networks. The system makes the following contributions:

- 1) **Spatial Grid Indexing for Transit Stop Lookup:** A grid-based spatial index partitions the Kathmandu Valley into $500 \text{ m} \times 500 \text{ m}$ cells, reducing stop proximity queries from $O(N)$ linear scans to amortized $O(1)$ hash-map lookups, where N is the total number of stops in the network.
- 2) **Non-Greedy, Topology-Aware Route Search:** Instead of greedily picking the nearest stop, the algorithm evaluates every boarding and alighting candidate inside the search radius. This accounts for multi-lane roads, where two stops at almost the same location can serve opposite travel directions and therefore differ sharply in usefulness.
- 3) **Penalty-Weighted Scoring for Multi-Objective Optimization:** A composite scoring function that jointly minimizes travel time, walking distance, and transfer count, with weights calibrated to the travel preferences observed among transit users in the Kathmandu Valley.
- 4) **Heterogeneous Topology Support:** Native handling of linear, circular, and hybrid "lollipop" route geometries through virtual stop-sequence doubling and directional constraint enforcement on transfer candidate selection.



- 5) **Offline-First, Client-Side Architecture:** All route computation executes on-device using locally cached data, eliminating server round-trip latency and enabling full functionality without internet connectivity.

The remainder of this paper is organized as follows. Section II surveys prior work on transit route optimization and intelligent transportation systems. Section III describes the system architecture and the underlying data model. Section IV develops the algorithm, covering the spatial index, the route search strategies, and the scoring function. Section V reports the performance evaluation. Section VI discusses limitations and directions for future work, and Section VII concludes the paper.

II. RELATED WORK

A. Transit Route Optimization Algorithms

The classical shortest-path algorithm from Dijkstra [1] is used in many applications related to transportation networks. Bozyiğit et al. [2] suggested the Modified Dijkstra's Algorithm for public transport, which incorporated transfer penalties to account for the inconvenience of changing between different modes of transportation. The results showed that compared to the original algorithm, their approach reduced transfers by 47% and the distance walked by 17% with an increase of only 8% in the overall distance of the route. They proved that it was not enough to consider the distance between stops as the sole factor in route planning. However, their algorithm still requires a graph to be constructed beforehand, which makes it impossible to use in networks that do not have appropriate infrastructure, for example, in many countries of the developing world.

Darip et al. [3] compared Breadth-First Search (BFS) against Depth-First Search (DFS) for shortest-path search. They concluded that although BFS does return a path of minimum length in an unweighted graph, it ignores the other factors a passenger cares about, such as the number of transfers or the amount of walking required. Their conclusion—that graph traversal algorithms need substantial modification before they are usable for transportation planning—is consistent with our own design rationale.



B. Intelligent Transportation Systems and Smart City Navigation

Akram et al. [4] studied the role of public transportation in sustainable urban development by comparing the cities of Manchester (UK) and Shenzhen (China). Their work underlined that smart transit information systems are a key enabler of sustainable mobility, and that their availability is a major factor in raising ridership and reducing dependence on private vehicles. This finding is directly relevant to our setting, since we aim to improve both the commuter and the tourist experience in Kathmandu through better transit information—a need that grows sharper under rapid urbanization.

Ksentini et al. [9] introduced +Tour, a system that leverages Multi-access Edge Computing (MEC) to recommend personalized tourism itineraries. Their approach optimizes the allocation of resources across Points of Interest (POIs) and reports an 11% gain in efficiency and a 40% gain in user experience. Their architecture is server-side and edge-assisted, and therefore distinct from our fully client-side design, but their work supports the case for embedding tourism context into the design of transportation systems.

Ding et al. [5] investigated intelligent tourism delivered over the mobile Internet, establishing that integrating real-time data into tourism applications markedly raises user satisfaction. Their framework for personalized travel planning informed our decision to combine attraction discovery with transit route finding in a single application, so that the user is assisted both in reaching a destination and in exploring the city.

C. Mobile Application Frameworks for Transit Systems

Santhosh et al. [7] evaluate Flutter as a cross-platform development framework for building responsive applications, which can target both Android and iOS platforms with a single code base. The authors report that Flutter's capacity to compile Dart code to near-native performance is sufficient to implement the computationally taxing route calculation locally, thus supporting the technical decision to implement the calculation on the device, rather than in the cloud.

Figueiredo and Rodrigues [8] describe a general approach to designing geotourism applications, with a focus on representing the domain model according to user-centered design principles. The authors' emphasis on developing personas and scenarios helped shape the feasibility study and the



identification of requirements. In particular, the consideration of the tourist user persona has guided the present study's choices in the design of the user interface.

D. Research Gap

Despite the recent progress in transit routing algorithms and smart city software development, there is a clear gap in solutions targeting the specific needs of informal transit systems of developing countries. Specifically, existing algorithms are built upon GTFS or similar data standards implying formally defined route topologies. There are no existing solutions that simultaneously address the challenges of (i) heterogeneous route topology (linear, circular, hybrid), (ii) complete lack of standardized data formats requiring crowdsourcing and subsequent complex data preprocessing, (iii) multi-lane roads where the stop position is not indicative of the direction, and (iv) necessity to function in offline mode due to unreliable internet infrastructure. We present an algorithmic framework targeting the specific needs of informal transit systems in developing countries.

III. SYSTEM ARCHITECTURE

A. Architectural Overview

The proposed system follows a client-heavy, offline-first architecture where all route computation executes on the mobile device. The system comprises four layers, as illustrated in Fig. 1.

Presentation Layer: Built with the Flutter framework [7], the UI provides three primary interfaces: Route Search (source/destination input, route option display, map visualization), Explore (location-based attraction discovery), and Transit Management.

Service Layer: Contains the core computational components:

- *Route Finder Service:* Implements the spatial indexing, route search, scoring, and deduplication algorithms detailed in Section IV.
- *Spatial Index:* Maintains the grid-based spatial index for $O(1)$ stop proximity queries.

Data Layer: Transit data—routes, stops, and operators—is retrieved from Google Fire store and cached on the device using Shared Preferences. A version flag held in the Fire store `app_config` collection signals that route data has changed; when the remote flag differs from the



locally stored value, the cache is invalidated and refreshed. Every subsequent launch is therefore served from local storage, and a network read occurs only when the underlying data has actually been updated.

External Layer: Google Maps provides map rendering and geocoding. Device positioning is obtained through the Geolocator plugin. The Open Source Routing Machine (OSRM) generates the walking and driving polylines drawn on the map. Additional external APIs supply traffic data and supplementary geocoding services.

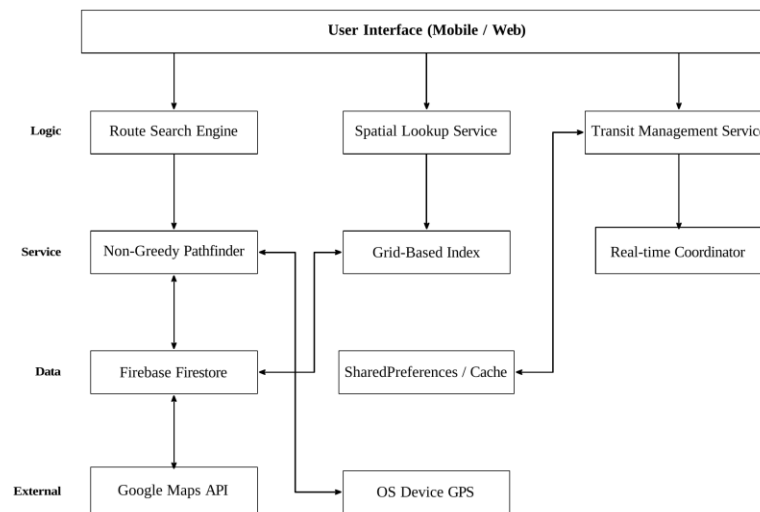


Fig. 1. Proposed system architecture. The Logic Layer communicates with the Service Layer, which interfaces with the Data Layer and External APIs.

B. Data Model

The transit network is represented by three core entities:

Definition 1 (Bus Stop). A bus stop s is a tuple $s = (id, name, \phi, \lambda, area)$, where id is a descriptive string identifier, $name$ is the human-readable label, (ϕ, λ) are the latitude and longitude coordinates, and $area$ is a string tag for text-based search indexing.

Definition 2 (Bus Route). A bus route r is a tuple $r = (id, name, S, \beta, \gamma)$, where id is the route identifier, $name$ is the display name, $S = [s_0, s_1, \dots, s_{n-1}]$ is an ordered sequence of bus stops defining the route path, $\beta \in \{true, false\}$ is a reversibility flag, and γ denotes the topological classification.

The system handles three primary route topologies from the Kathmandu Valley network:



- 1) **Linear (Bidirectional):** Standard paths ($A \leftrightarrow B$) where $\beta = true$. The return journey is inferred by reversing the stop sequence: $S^R = [s_{n-1}, s_{n-2}, \dots, s_0]$.
- 2) **Circular (Unidirectional):** Complex loops (e.g., Ring Road) where $\beta = false$. These are stored as separate entries (e.g., Clockwise/Anticlockwise) with strict directionality to reflect physical lane separation.
- 3) **Hybrid (Lollipop):** Routes starting with a linear stem and terminating in a circular loop. The system identifies these by matching start/end stop parity while detecting differing intermediate stop sequences, ensuring correct distance calculation for loop entry and wrap-around logic.

Definition 3 (Bus Stop Reference). A bus stop reference ρ is a tuple $\rho = (s, r, i, \delta)$, where s is a bus stop, r is the route it belongs to, i is the index of s in the route's stop sequence, and $\delta \in \{forward, reverse\}$ indicates the traversal direction. This structure enables the spatial index to return not only nearby stops but also the route and directional context for each stop.

C. Data Synchronization and Caching

The system employs a two-tiered caching strategy:

Tier 1 (Persistent Data Cache): Route and stop data is pulled from Firestore on first use and persisted locally. On subsequent launches, the locally stored version number is compared against the remote flag (`app_config/cache_trigger`). If $v_{remote} \neq v_{local}$, the local cache is discarded and fresh data is fetched. This keeps Firestore read operations—and therefore operating cost—to a minimum while guaranteeing that the cached data remains current.

Tier 2 (In-Memory Polyline Cache): Decoded polyline geometries are cached in memory with a 24-hour time-to-live (TTL). Cache keys are constructed from the start/end coordinates and transport mode:

$$k = ([\varphi_a \cdot 10^5], [\lambda_a \cdot 10^5], [\varphi_b \cdot 10^5], [\lambda_b \cdot 10^5], m) \quad (1)$$

where (φ_a, λ_a) and (φ_b, λ_b) are the segment endpoints and m is the mode (walking/driving). This eliminates redundant geometry decoding when users revisit or compare route options.



IV. ALGORITHM DESIGN

This section presents the core algorithmic contribution: a spatially-indexed, penalty-weighted search strategy. The system’s operational parameters, derived from field data in the Kathmandu Valley, are summarized in Table I.

TABLE I
ALGORITHM SEARCH AND SPEED CONSTANTS

Constant	Value	Purpose
$D_{\text{walk,max}}$	600 m	Max walk to source/dest stop
$D_{\text{trans,max}}$	400 m	Max walk between transfers
R_{min}	100 m	Initial radial search distance
R_{max}	1500 m	Maximum expanded search
v_{bus}	20 km/h	Avg. bus speed with stops
v_{walk}	5 km/h	Avg. human walking speed

A. Spatial Grid Index

Given a transit network with N total stops across all routes, the naive way to answer the query “which stops lie near the position $q = (\phi_q, \lambda_q)$?” is to compute the Haversine distance between the query point and every stop, which costs $O(N)$ time per query. For the Kathmandu Valley network ($N \approx 5,000$) this remains computationally feasible, but it is wasteful for real-time interaction on a mobile device.

We use a grid-based spatial index that partitions the geographic space into rectangular cells of approximately $500 \text{ m} \times 500 \text{ m}$ at the latitude of Kathmandu ($\phi \approx 27.7^\circ\text{N}$). Each stop is assigned to the cell that contains its coordinates:

$$\text{CellKey}(s) = (\lfloor \phi_s / 0.005 \rfloor, \lfloor \lambda_s / 0.005 \rfloor) \quad (2)$$

The constant 0.005 degrees corresponds to approximately 555 m in latitude and $555 \cos(\phi) \approx 491 \text{ m}$ in longitude at $\phi = 27.7^\circ\text{N}$, yielding approximately $500 \text{ m} \times 490 \text{ m}$ cells.

The index is constructed as a hash map $G : \text{String} \rightarrow \text{List}(\rho)$, mapping cell keys to lists of bus stop references. For routes with $\beta = \text{true}$, both forward and reverse references are indexed, effectively doubling the entries for bidirectional routes. Construction complexity is $O(N)$.

Proximity Query. Given a query point q and search radius R (in km), the query identifies the cell containing q and searches all cells within a radius of $\lceil R/0.5 \rceil$ cells:



$$\text{Nearby}(q, R) = \cup \{p \in G[i, j] : d_H(q, p.s) \leq R\} \quad (3)$$

over all i, j with $|i - i_q| \leq c_R$ and $|j - j_q| \leq c_R$

where $(i_q, j_q) = \text{CellKey}(q)$, $c_R = \lceil R/0.5 \rceil$, and d_H is the Haversine distance function:

$$d_H(p_1, p_2) = 2R_E \cdot \arctan2(\sqrt{a}, \sqrt{1-a}) \quad (4)$$

$$a = \sin^2(\Delta\phi / 2) + \cos \phi_1 \cdot \cos \phi_2 \cdot \sin^2(\Delta\lambda / 2) \quad (5)$$

where $\Delta\phi = \phi_2 - \phi_1$ and $\Delta\lambda = \lambda_2 - \lambda_1$ are the differences in latitude and longitude between points p_1 and p_2 respectively, and $R_E = 6,371$ km is the mean Earth radius. The amortized query complexity is $O(c^2 \bar{k})$ where $\bar{k}$ is the average number of stops per cell, which for practical search radii ($R \leq 1.5$ km) is effectively $O(1)$ relative to the total network size N .

B. Graduated Radius Search

Rather than searching a fixed large radius—which would return excessive candidates and waste computation—the system employs an adaptive search strategy:

$$\mathbf{R} = [0.1, 0.3, 0.5, 0.8, 1.2, 1.5] \text{ km} \quad (6)$$

The search iterates over these radii in ascending order and terminates as soon as at least three candidate stops have been found *and* the current radius is at least 0.5 km. The 0.5 km floor exists to mitigate the *highway problem*: on a multi-lane divided road, the physically closest stop may lie on the opposite carriageway and therefore serve the wrong travel direction. Enforcing a minimum search area guarantees that stops on both sides of a wide road enter the candidate set.

C. Short-Distance Bypass

Before invoking the route search, the system checks the straight-line distance between source and destination:

$$d_H(\text{source}, \text{destination}) \leq 0.8 \text{ km} \Rightarrow \text{Walk-only route} \quad (7)$$

Under this threshold, any bus route would involve walking to a stop, riding 1–2 stops, and walking again, which is likely to take longer than simply walking the whole way. The bypass returns an immediate walk-only result, avoiding nonsensical suggestions. An additional UI-level check rejects any query for which $d_H < 0.3$ km, returning an inline message instead.



D. Direct Route Search

The direct route algorithm identifies single-bus connections between a source and destination. Algorithm 1 presents the pseudocode.

1) Circular Route Handling:

Circular routes (e.g., Ring Road) pose a particular difficulty: the stop sequence $S = [s_0, s_1, \dots, s_{n-1}]$ closes on itself so that $s_0 = s_{n-1}$. A naive linear scan starting from the boarding index would miss every journey that crosses the sequence boundary. We resolve this through *virtual doubling*: the sequence is extended to $S' = [s_0, s_1, \dots, s_{n-1}, s_1, s_2, \dots, s_{n-1}]$, allowing the search to traverse the wrap-around point. For direct trips, both the original boarding index and the wrapped index ($i + |S| - 1$) are evaluated, so that the shorter and the longer direction around the loop are both scored.

2) Non-Greedy Stop Selection:

A critical design element is the rejection of greedy nearest-stop selection. On multi-lane roadways, the stop nearest to the user may serve the direction opposite to the one the user intends to travel. The user might have to walk 200 m to reach the correct stop while a stop serving the opposite direction lies only 50 m away. Naively choosing the nearest stop would therefore return a route travelling away from the intended destination.

Our algorithm evaluates *all* possible boarding stops within the search radius, computing for each candidate the full journey cost: walking to the stop, the bus ride, and walking from the alighting stop. In the example above, the 200 m walk to the correctly oriented stop is preferred over the closer 50 m walk to the wrongly oriented one, avoiding roughly 30 minutes wasted on a bus travelling in the wrong direction.



Algorithm 1 Direct Route Search

Require: Source q_s , Destination q_d , Nearby source stops N_s , Nearby dest stops N_d

Ensure: List of RouteOption candidates

```

1: results  $\leftarrow \emptyset$ 
2: for each  $\rho_s \in N_s$  do
    3:  $r \leftarrow \rho_s.route$ ;  $\delta \leftarrow \rho_s.direction$ 
    4:  $S \leftarrow$  ordered stops of  $r$  in direction  $\delta$ 
    5: if  $r$  is circular then
        6:  $n \leftarrow |S|$  {Save base length before doubling}
        7:  $S \leftarrow S \parallel S[1:]$  {Virtual doubling for wrap-around}
        8: starts  $\leftarrow [\rho_s.i, \rho_s.i + n - 1]$ 
    9: else
        10: starts  $\leftarrow [\rho_s.i]$ 
    11: end if
    12: for each  $b \in$  starts do
        13:  $a^* \leftarrow \arg \min_{a>b} d_H(S[a], q_d)$  s.t.  $d_H \leq D_{walk,max}$ 
        14: if  $a^*$  exists then
            15: Compute walking segments, bus segment, total time
            16: score  $\leftarrow \text{Score}(0, t_{total}, w_{total})$ 
            17: results.add(RouteOption(...))
        18: end if
    19: end for
20: end for
21: return results

```

3) Trivial Ride Filtering:

Routes where the bus segment is shorter than the combined walking segments are filtered:

$$|S_{segment}| \leq 2 \wedge d_{bus} < (d_{walk,start} + d_{walk,end}) \Rightarrow \text{reject} \quad (8)$$

This prevents suggestions such as “walk 500 m to a stop, ride 1 stop for 300 m, walk 400 m to destination.”

E. Transfer Route Search

When no direct route connects the starting point and the destination, the system searches for single-transfer journeys served by two buses. The transfer search is invoked only after the direct search returns no results, so this additional computation is performed only when it is actually required.

1) Transfer Candidate Identification:

Transfer candidates are identified by locating stops on Route 1 (after the boarding point) that lie within 400 m walking distance of stops on Route 2 (before the alighting point). The directional constraint imposed on Route 2 is critical:



Directional Constraint: On Route 2, only stops with index $j < i_{\text{dest}}$ are admitted as transfer boarding points, where i_{dest} is the sequence index of the alighting stop on Route 2 nearest to destination q_d . This forces Bus 2 to travel *forward* from the transfer stop to the destination. Without the constraint, on a circular route such as Ring Road the algorithm could select a transfer point that sends the passenger the long way around the entire loop.

2) Cost-Based Candidate Ranking:

Transfer candidates are ranked not by transfer walking distance alone, but by estimated total journey cost:

$$C_{\text{transfer}} = d_{\text{bus1}}(i_{\text{board}}, i_{\text{transfer}}) + d_{\text{bus2}}(j_{\text{transfer}}, j_{\text{alight}}) + d_{\text{walk,transfer}} \quad (9)$$

where i_{board} is the boarding stop index on Route 1, i_{transfer} is the exit stop index on Route 1, j_{transfer} is the boarding stop index on Route 2, j_{alight} is the alighting stop index on Route 2, $d_{\text{walk,transfer}}$ is the walking distance between the two transfer stops, and $d_{\text{bus1}}(\cdot, \cdot)$, $d_{\text{bus2}}(\cdot, \cdot)$ are cumulative segment distances along each route. This prevents a 0 m-walk transfer with a 20-stop detour from outranking a 200 m-walk transfer with a 3-stop direct ride.

Algorithm 2 Transfer Route Search

Require: Source q_s , Destination q_d , N_s , N_d , All routes R

Ensure: List of RouteOption candidates

```

1: results  $\leftarrow \emptyset$ ; seen  $\leftarrow \emptyset$ 
2: for each  $\rho_s \in N_s$  do
3:   for each  $\rho_d \in N_d$  do
4:     if  $\rho_s.\text{route} = \rho_d.\text{route} \wedge \rho_s.\delta = \rho_d.\delta$  then
5:       continue {Same route/direction = direct, not transfer}
6:     end if
7:     key  $\leftarrow (\rho_s.\text{route.id}, \rho_s.\delta, \rho_d.\text{route.id}, \rho_d.\delta)$ 
8:     if key  $\in$  seen then
9:       continue
10:    end if
11:    seen.add(key)
12:    C  $\leftarrow$  FindTransferCandidates( $\rho_s, \rho_d, R$ )
13:    for each candidate  $c$  in top-5 of C do
14:      Build 5-segment path: walk  $\rightarrow$  bus1  $\rightarrow$  transfer walk  $\rightarrow$  bus2  $\rightarrow$  walk
15:       $t_{\text{transfer}} \leftarrow d_{\text{walk}}/v_{\text{walk}} + 5$  min buffer
16:      score  $\leftarrow$  Score(1,  $t_{\text{total}}$ ,  $w_{\text{total}}$ )
17:      results.add(RouteOption(. . .))
18:    end for
19:  end for
20: end for
21: return results

```



3) Transfer Time Buffer:

A fixed 5-minute buffer is added to all transfer walking times to model real-world transfer behavior:

$$t_{\text{transfer}} = (d_{\text{walk,transfer}} / v_{\text{walk}}) \times 60 + 5 \text{ minutes} \quad (10)$$

This accounts for disembarking, navigating to the connecting stop, and waiting for the second vehicle.

F. Scoring and Ranking

Route options are ranked by a weighted cost function designed to minimize total journey “friction”:

$$S = 40 \cdot T + t + 12 \cdot W \quad (11)$$

where T is the number of transfers, t is the total travel time in minutes, and W is the total walking distance in kilometers. A lower score indicates a more desirable route.

1) Weight Rationale:

- **Transfer Penalty (40 points):** Transfers add complexity and risk—waiting time, missed connections, and directional confusion at unfamiliar stops. The 40-point penalty keeps a direct route preferred unless the transfer alternative saves at least 40 minutes of adjusted time. The value 40 was calibrated by iteratively testing candidate weights against routes in the Kathmandu network whose optimal itinerary was already known.
- **Walking Penalty (12 points/km):** One kilometer of walking adds 12 points, roughly equivalent to 12 minutes of travel time (at a walking speed of 5 km/h, 1 km $\approx$ 12 minutes). This keeps the penalty proportionate and avoids over-penalizing the short walks that often make the overall route faster: a 200 m walk saving 5 minutes of bus time correctly scores better than a zero-walk option with a longer ride.
- **Travel Time (1 point/minute):** This term forms the baseline cost unit—one minute spent in any travel mode contributes exactly one point.



G. Time Estimation

Travel time is estimated using average speed constants calibrated to Kathmandu Valley traffic conditions:

Walking time:

$$t_{\text{walk}} = (d_{\text{walk}} / 5.0) \times 60 \text{ minutes} \quad (12)$$

Bus travel time:

$$t_{\text{bus}} = (d_{\text{bus}} / 20.0) \times 60 + n_{\text{stops}} \text{ minutes} \quad (13)$$

where n_{stops} is the number of intermediate stops, each adding approximately 1 minute to account for deceleration, passenger boarding/alighting, and acceleration. The average bus speed of 20 km/h reflects the stop-and-go nature of urban bus transit in Kathmandu, accounting for traffic congestion, narrow roads, and frequent stops.

H. Route Deduplication

The search may produce multiple route options from the same bus service (e.g., from different nearby boarding stops, or from clockwise/anticlockwise variants of Ring Road). A two-stage deduplication process reduces redundancy:

Stage 1 (Name Normalization): Route names are normalized by stripping directional indicators (“Clockwise,” “Anticlockwise,” “CW,” “CCW”). For each normalized name, only the option with the lowest score is retained.

Stage 2 (Segment Deduplication): Route options are further deduplicated by comparing their segment descriptions. Options with identical segment sequences are collapsed, retaining only the lowest-scored variant.

I. Concurrent Execution

Route computation is offloaded to a background isolate via Dart’s `compute()` function to prevent UI thread blocking. This ensures that the application remains responsive during route search, even for transfer routes that may evaluate thousands of candidate combinations.



V. RESULTS AND EVALUATION

The proposed system was evaluated using a physically collected set of transit routes (55 bus routes and more than 600 bus stops). The set was gathered in the field and manually entered into the database since no official GTFS feed was available for the Kathmandu valley (Kathmandu, Lalitpur, and Bhaktapur districts). In addition, the scalability of the proposed solution was characterized on a synthetic set of 200 routes and 5,000 stops.

A. Data Collection

The routing engine is built on a primary dataset assembled through field surveys and operator consultations, covering both the formal and the informal transit corridors of the Kathmandu Valley.

1) Transportation Data:

- 55 distinct public bus routes collected with sequenced stops and structured fare information.
- Precise GPS coordinates for 600+ bus stops accurately mapped across the Kathmandu Valley and indexed for routing.

Dataset 1: General Stop Spatial Coordinates Excerpt

Description: Extracted sample from the primary database comprising 600+ bus stops across the valley.

Stop ID	Area	Location (Lat, Lng)	Name
sumeru hospital	Dhapakhel	27.644909, 85.325996	Sumeru Hospital
gems school	Dhapakhel	27.643312, 85.327342	GEMS School
Tewa	Dhapakhel	27.639439, 85.328093	Tewa
pashupati temple dhapakhel	Dhapakhel	27.630763, 85.329321	Pashupati Temple (Dhapakhel)
tinkune dhapakhel	Dhapakhel	27.628059, 85.327946	Tinkune Dhapakhel
Nagdaha	Dhapakhel	27.625322, 85.333479	Nagdaha

Dataset 2: Directional Routing (Circular Path) Excerpt

Description: Dataset subset handling directionally constrained routes (e.g., clockwise loops) preventing invalid algorithmic backtracking.

Stop ID	Area	Location (Lat, Lng)	Name
acem clockwise	Kathmandu	27.687870, 85.289564	Advanced College of Eng.
balaju clockwise	Kathmandu	27.727269, 85.304755	Balaju
balkhu clockwise	Kathmandu	27.684833, 85.296500	Balkhu



balkumari bridge clockwise	Lalitpur	27.673271, 85.342208	Balkumari Pool
chabahil clockwise	Kathmandu	27.716852, 85.346615	Chabahil

The results show that the proposed system sustains consistent sub-millisecond search times on the 600-stop dataset, confirming the efficiency of the non-greedy, spatially indexed methodology.

B. Algorithmic Performance Analysis (Scale Analysis)

To verify system scalability, the algorithm was benchmarked on both a small-scale set (100 stops) and an equivalent large-scale simulation comprising 200 routes and 5,000 stops. Table II presents the empirical results and theoretical complexity ranks.

TABLE II
ALGORITHMIC PERFORMANCE BENCHMARKS (SCALE ANALYSIS)

Phase	Small (100 Stops)	Large (5000 Stops)	Complexity Rank
Spatial Query	170.0 μ s	131.0 μ s	$O(1)$ Average
Direct Search	150.3 μ s	150.1 μ s	$O(K \cdot L)$
Transfer Logic	385.0 μ s	5370.0 μ s	$O(K^2 \cdot L^2)$
Total Compute	0.705 ms	5.651 ms	N/A

The key insight is that all complexities are parameterized by K (number of nearby candidate stops) and L (average route length in stops), both of which are bounded constants independent of the total network size N . This ensures that route search performance does not degrade as the network grows.

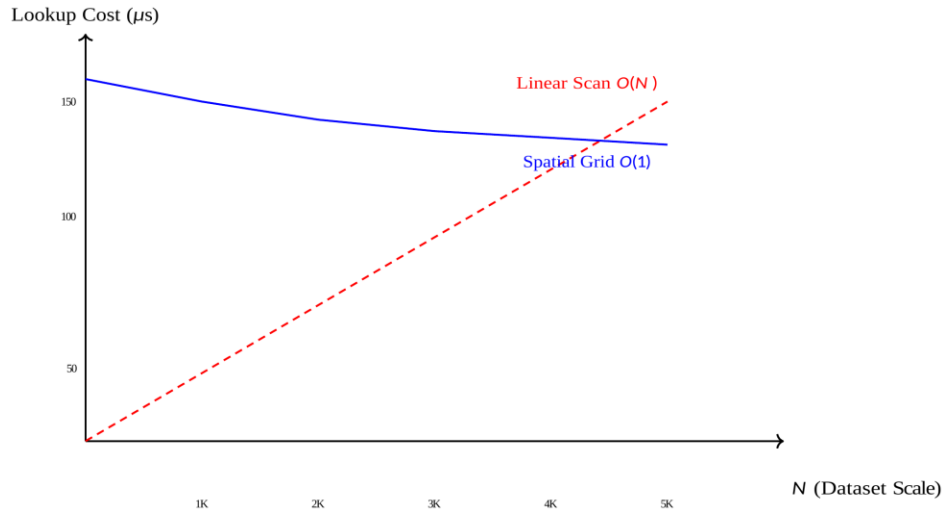


Fig. 2. Theoretical $O(N)$ linear scan vs. measured $O(1)$ spatial grid lookup cost. Only the endpoints (100-stop and 5,000-stop measurements) are empirical; the curves are illustrative of asymptotic behavior.

C. Comparative Analysis

Table III compares the proposed approach against established graph-based algorithms on key algorithmic design dimensions. No runtime comparison against external implementations is included, as the algorithms differ fundamentally in data representation and were not benchmarked on the same hardware and dataset.

TABLE III
ALGORITHMIC FEATURE COMPARISON

Algorithm	Graph Required	Topology Aware	Transfer Penalty	Offline First
Dijkstra [1]	Yes	No	No	No
Modified Dijkstra [2]	Yes	Partial	Yes	No
BFS [3]	Yes	No	No	No
Proposed (ours)	No	Yes	Yes (40 pts)	Yes

The proposed system requires no pre-constructed graph, supporting full offline deployment without GTFS data. Unlike graph traversal algorithms, it natively handles heterogeneous route topologies (linear, circular, hybrid) and provides simultaneous multi-objective optimization through the penalty-weighted scoring function. The architecture is thus uniquely suited to informal transit networks where structured data feeds are unavailable.



D. System Implementation Analysis

The proposed routing framework was implemented as a production-grade mobile application for the Kathmandu Valley. Practical evaluation focused on realistic use cases, including the Kathmandu-to-Lubu corridor, whose hub requires a transfer for most journeys. The algorithm resolved the transfer requirements of these journeys correctly, balancing waiting and walking penalties through the multi-objective scoring function.

A further operational feature is the 0.8 km short-distance bypass. Below this threshold, the system returns walking directions directly, preventing the suggestion of redundant transit segments that would only lengthen the total journey time. A driver telemetry module that broadcasts the current GPS position of the vehicle is also included, providing the technical foundation for integrating dynamic arrival predictions into the routing engine in future work.

E. Deployment and User Base

The application has been available since March 2026, serving tourists, students, and daily commuters in the Kathmandu Valley.

VI. DISCUSSION

A. Strengths

The proposed system demonstrates several architectural and algorithmic strengths. The offline-first design eliminates dependency on network connectivity, which is frequently intermittent in the Kathmandu Valley. The spatial indexing approach provides consistent sub-millisecond stop lookups regardless of network scale. The non-greedy search strategy correctly handles the multi-lane road configurations prevalent in the Valley's road network. The penalty-weighted scoring function produces route rankings that align with user preferences without requiring explicit user input.

B. Limitations

Route Coverage: The present dataset covers only the major operators (Mahanagar Yatayat, Sajha Yatayat, and selected private operators) and excludes micro-buses, tempos, and local carriers that operate at a smaller scale and serve peripheral localities. Extending coverage requires further manual field collection, since no official transit data is published for these services.



Static Time Estimation: The existing system uses fixed speed constants of 20 km/h for buses and 5 km/h for walking, and does not account for real-time traffic congestion or time-of-day variation. During peak hours, actual bus speeds in central Kathmandu can fall to 5–10 km/h, which introduces inaccuracy into the time estimates.

Single Transfer Limit: The algorithm admits at most one transfer per journey. This covers the majority of practical trips in the Kathmandu Valley, but journeys requiring two or more transfers are not returned. The restriction was deliberate: multi-transfer itineraries in the Valley are unreliable in practice, because vehicle frequencies on the relevant services are irregular.

Absence of Real-Time Vehicle Tracking: Although the system has a driver tracking component (for registered drivers who broadcast their GPS locations), the route finding algorithm itself does not make use of vehicle positions or predicted arrival times.

C. Novelty and Real-World Impact

The proposed system is the first operational public transport route finding system for the Kathmandu Valley. Its value is twofold. First, the algorithmic framework—spatial grid indexing combined with a non-greedy, topology-aware search—transfers to any informal transit network whose routes follow fixed sequences of stops but which lacks the digital infrastructure that graph-based approaches presuppose.

Second, the system removes a major information barrier to public transport use in the region, contributing directly to broader smart city and sustainable mobility objectives.

VII. CONCLUSION

This paper describes the design, implementation, and deployment of a spatially-indexed, non-greedy route finding system for informal public transport networks in developing regions. The system was evaluated on the Kathmandu Valley transit network, where no digital routing system existed before this work.

The technical contributions of this work include (1) a grid-based spatial index with amortized $O(1)$ stop lookups, (2) a non-greedy algorithm that handles multi-lane roads and heterogeneous route topologies (linear, circular, and mixed), (3) a penalty-based scoring function ($S = 40T + t +$



12W) that enables multi-objective optimization, and (4) directional constraints on transfer candidates to avoid erroneous circular route recommendations.

The system was evaluated on a 55-route, 600+-stop dataset collected in the field, achieving direct route resolution in 0.15 ms and transfer route resolution in 5.37 ms. Scalability was confirmed on a 5,000-stop simulation, whose total compute time of 5.65 ms remains well within the 200 ms real-time threshold. The system is deployed on the Google Play Store, the Apple App Store, and the web, serving the Kathmandu Valley.

Future work will (1) extend coverage to a greater diversity of services, including micro-buses, tempos, and local vehicles, (2) learn time estimates from real-time traffic data, (3) ingest live GPS positions from buses into the route recommendations, and (4) evaluate the generalizability of the algorithmic contributions to other informal transit networks in South Asia.

REFERENCES

- [1] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.
- [2] A. Bozyiğit, G. Alankuş, and E. Nasiboğlu, "Public transport route planning: Modified Dijkstra's algorithm," in *Proc. Int. Conf. Comput. Sci. Eng. (UBMK)*, Antalya, Turkey, 2017, pp. 1–6.
- [3] M. Darip *et al.*, "Comparative analysis of BFS and DFS algorithms for shortest path search in complex environments," *J. Inf. Technol. Comput. Sci.*, 2023.
- [4] N. Akram, W. Chen, K. Tariq, H. Linjun, and L. Zhaosheng, "Role of public transportation in reducing traffic congestion, enhancing connectivity, and promoting sustainable urban development in Manchester and Shenzhen," *Int. J. Soc. Sci. Humanities Invention*, vol. 12, pp. 8401–8408, 2025.
- [5] D. Ding, S. Wang, and X. Chang, "Intelligent tourism: Innovative applications of mobile technology in personalized travel planning," *Int. J. Interactive Mobile Technol. (iJIM)*, vol. 18, pp. 17–30, 2024.
- [6] I. Nur *et al.*, "An overview of public transportation services as an attraction of tourism industry in Malaysia," in *Proc. Int. Conf.*, 2019.
- [7] A. Santhosh, A. Tiji, A. R. N. Thomas, A. Anoop, and T. Varghese, "Cross-platform innovation: The rise and impact of Flutter in modern app development," *Int. Res. J. Adv. Eng. Manag. (IRJAEM)*, vol. 2, pp. 3560–3569, 2024.
- [8] S. Figueiredo and R. Rodrigues, "Designing a geo-tourism app: A principled approach," *arXiv preprint arXiv:2504.13876*, 2025.
- [9] A. Ksentini, L. Blanco, and D. Aversa, "+Tour: Recommending personalized itineraries for smart tourism."