Enabling Routes as Context in Mobile Services

Agnė Brilingaitė, Christian S. Jensen, and Nora Zokaitė

April 19, 2005

TR-9

A DB Technical Report

Title	Enabling Routes as Context in Mobile Services
	Copyright © 2005 Agnė Brilingaitė, Christian S. Jensen, and Nora Zokaitė. All rights reserved.
Author(s)	Agnė Brilingaitė, Christian S. Jensen, and Nora Zokaitė
Publication History	Extended version of: Brilingaitė, A., C. S. Jensen, and N. Zokaitė, "Enabling Routes as Context in Mobile Services," in *Proceedings of the Twelfth ACM International Symposium on Advances in Geographic Information Systems*, Washington DC, USA, November 12–13, 2004, pp. 127–136.

Abstract

With the continuing advances in wireless communications, geo-positioning, and portable electronics, an infrastructure is emerging that enables the delivery of on-line, location-enabled services to very large numbers of mobile users. A typical usage situation for mobile services is one characterized by a small screen and no keyboard, and by the service being only a secondary focus of the user. Under such circumstances, it is particularly important to deliver the "right" information and service at the right time, with as little user interaction as possible. This may be achieved by making services context aware.

Mobile users frequently follow the same route to a destination as they did during previous trips to the destination, and the route and destination constitute important aspects of the context for a range of services. This paper presents key concepts underlying a software component that identifies and accumulates the routes of a user along with their usage patterns and that makes the routes available to services. Experiences from using the component on logs of GPS positions acquired from vehicles traveling within a real road network are reported.

1 Introduction

The global adoption rate of mobile phones is very large, and while mobile phones are currently being used mostly for voice communication, the volume of data communication is increasing. With technologies such as GPRS, 2.5G (EDGE), and 3G (CDMA, UMTS), the user can be always on at no extra cost, and bandwidth is increasing. Next, the advent of the global navigation satellite system Galileo as well as regulatory developments, such as the US E911 Mandate [13] and similar developments in Asia and Europe, contribute to the increasing availability of positioning capabilities. An infrastructure is thus emerging that supports a range of location-enabled on-line mobile services [23].

However, mobile services are delivered to devices that are typically without (qwerty) keyboards and that have only small screens. Further, the services may be expected to be delivered in situations where the user's main focus of attention is not the service, but rather that of, e.g., navigating safely in traffic. For these reasons, it is much more important than in a desktop computing situation that the user receives only the relevant information and service, with as little interaction with the system as possible. One approach to obtaining these qualities is to make the mobile services aware of the user's context.

The user's current location is one possible context, and the user's destination is another. Yet another is the route that takes the user from the current location to the destination. This paper's focus is on the latter.

Routes are interesting for two reasons. First, folklore as well as common sense has it that mobile users typically travel towards a destination (rather than moving around, aimlessly) and that a user often or typically follows the same route when going from one location to another. For example, a user typically travels on the same route from home to work. Second, routes are significant as context for a range of services. For example, a service that knows the route of a user may alert the user about road conditions, e.g., congestion, construction, and accidents, on the route ahead, while not bothering the user with conditions that do not relate to the user's route. As another example, routes may be used when a user requests the locations of "nearby" points of interest. More specifically, a service may suggest restaurants or gas stations to the user that are near to the user's route, rather than merely to the user's current location [16]. Information about routes followed by mobile users can also be used in tracking services to accomplish more efficient tracking of the users [9, 10].

This paper describes key techniques underlying a software component that builds routes for individual users based on traces of GPS coordinates. In the proposed system architecture, client-side devices perform information filtering and prepare information for sending to the server. The server side uses linear referencing for the capture of the underlying transportation infrastructure and for the capture of routes, which are sequences of road parts that connect start and end destination objects. Aggregated usage information for each route is also maintained. The component is implemented using Java, Oracle's PL/SQL, and Oracle

Spatial. Proof of concept experiments that use GPS logs obtained from vehicles traveling in the Aalborg area and use a road network for this area are reported.

The paper is structured as follows. The system architecture and the route recording component is described in Section 2. Data structures necessary for the capture of routes are given in Section 3, and key algorithms used by the component are covered in Section 4. Insights from an experimental validation are reported in Section 5. Finally, Section 6 covers related work, and Section 7 summarizes and offers directions for future work. Four appendices present algorithms used in the techniques presented in the main body of the paper.

2 System Architecture

Following an overview of the client and server sides, this section describes how the two sides collaborate during route recording.

2.1 Client and Server Sides

We assume that a client device has a GPS receiver, a data connection to the server, and the computing and storage capabilities of a typical modern mobile phone. A current example is a Nokia 3650 with a GPRS connection and an Emtac Bluetooth GPS. GPS receivers transmit NMEA sentences [11, 20], which include location/time/date information, but also additional information that is less important for our purposes.

Client devices store four data blocks, which are described in Figure 1 in XML format. The first block contains personal information about each user. The second block records each user's destination objects. Each object has global/local IDs, a location given by a circular area, and a description. The description is a name that is meaningful to the user, e.g., "home" or "work." The third block captures the destination objects of routes. The fourth block of data records the usage times of each route. The time is approximated to week days, hours, and quarters of an hour.

The user inputs personal information and names for destination objects when this is requested by the client.

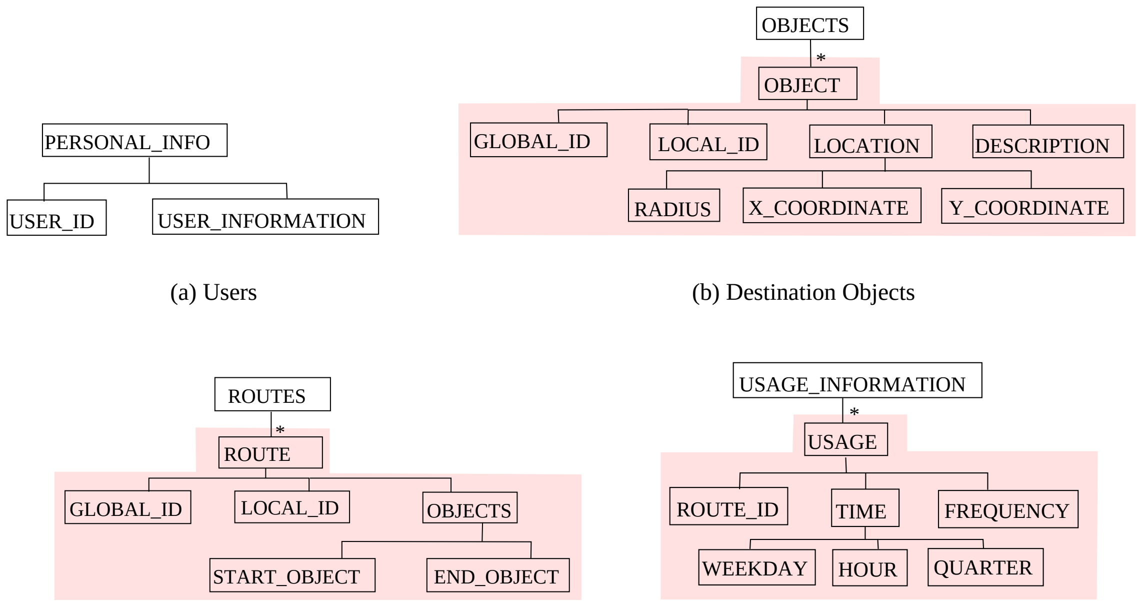
The server side uses the Oracle Application Server. The server records and analyzes the information received from the clients. Everything about each route, i.e., its constituent road-network parts and its usage, as well as each user's personal information are stored on the server. This is done to avoid information loss—users who switch to a new device can obtain all relevant information from the server. While not discussed further in this paper, we believe that encryption may be employed to counter privacy concerns.

2.2 Route Recording Functionality

We cover the interaction between client and server first and then cover route recording on the client and server sides.

2.2.1 Client and Server Interaction

The user activates and deactivates the process of route recording. When active, the client device filters and buffers location/time information obtained from the GPS receiver. This information is eventually transmitted to the server along with information about the user and the user's destination objects. The transmission frequency depends on the route length, the technical abilities of the client device, and the connection quality. When it has the necessary information, the server performs route construction, records the usage time, and assigns an ID to the route. The result is stored in the database and is also sent to the client.



(a) Users (b) Destination Objects

(c) Routes (d) Usages

Figure 1: Client-Side Data

The data sent to the server by the client has three parts: user, object, and standard information. The data format depends on which data is already available.

User information. If the user is already registered, this data block includes an ID. For new users, a user description is included. Thus, we have **[userId]** or **[undefined: description]** in this block.

Object information. Routes start and end at destination objects. The destination objects of a new route can have been used already to define the start or end of other routes, in which case the server can itself identify the objects according to their GPS coordinates. If both objects are known, this data block is empty, **[,]**. If one object is undefined, the data block contains a start description, **[undefined: description,]**, or an end description, **[,undefined: description]**. If the start and end objects are yet to be defined, the block has descriptions for both of them: **[undefined: description, undefined: description]**.

Standard information. Date, time, and GPS location information are always included. This block includes three elements, **[date, time, GPS]**.

When the server sends data to a client, it always returns the ID for a newly recorded route. If any of the route parameters are undefined, the client assumes that the data stream from the server will include the missing information. The server generates IDs for users and the users' destination objects. These IDs are returned to the client.

The server also returns a center location for a newly recorded destination object if the center location of the object differs from the first/last GPS coordinate pair in the GPS stream after location approximation. The server returns a radius together with the center location only if the server selects a radius that differs from the default value.

Thus, the format of the data from the server is **[userId, startObjectId, endObjectId, routeId, (xStart, yStart; radiusStart), (xEnd, yEnd; radiusEnd)]**, where **routeId** is the only parameter that is always included. The client receives the data stream from the server, analyzes it, and records its data.

2.2.2 Client-Side Route Recording

The client takes part in the route recording by preparing the data stream, described in the previous section, to be sent to the server. The blocks of user and object information in the data stream are constructed using data stored locally (see Section 2.1). The standard data block is constructed by analyzing the information from the GPS receiver.

The order of the steps for route recording on the client device is presented in Figure 2. When the

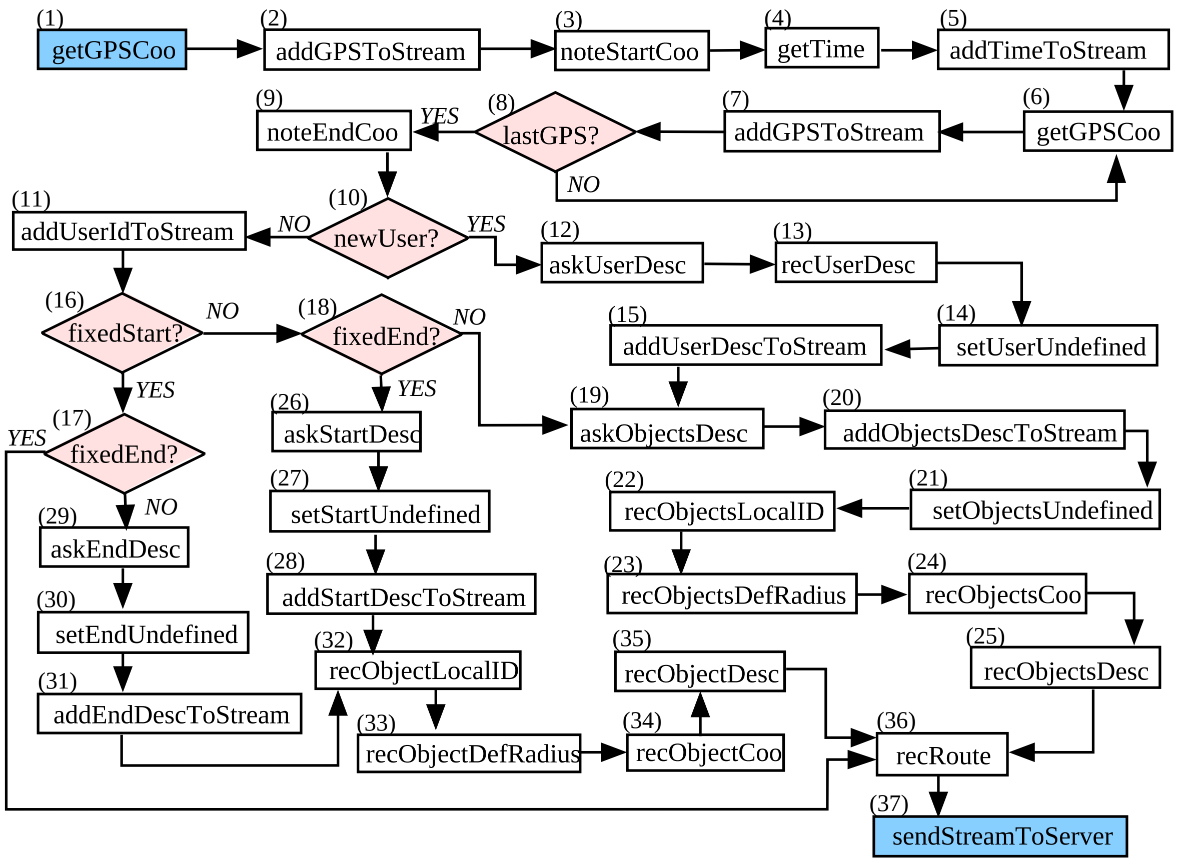


Figure 2: Client-Side Route Recording

user activates route recording, the client starts obtaining GPS information from the GPS receiver. Having received the first pair of coordinates, the client records the time to be associated with the usage of the route being recorded (1–5 in Figure 2). The client keeps extracting coordinates from the GPS stream until recording is deactivated (6–8). Upon deactivation, the end of the route is noted (9) for further analysis. The result is the standard information block for the data stream to be sent to the server.

If the user is already registered in the system, the user's ID is added to the stream (11 in Figure 2); otherwise, the client requests a user description. The device records the description locally, sets the user as undefined in the data stream, and adds the description to the data stream (12–15).

The last task is to build the destination object block. If the start and end objects are undefined (16,18) or the user is new, the device obtains descriptions of the destination objects (19). The objects are set as undefined in the stream and their descriptions are added to the data stream (20–21). The device records descriptions, default radiuses, and locations locally together with the local ID (22–25). If only one object is undefined, the same steps are done for only one object. If both objects are defined, the block is empty.

When all three data blocks have been constructed, the route is recorded (36) locally using the local parameters and leaving the global parameters undefined. The stream is finally sent to the server (37).

2.2.3 Server-Side Route Recording

The server performs the main route recording—that of transforming the data from a client into a route given by a sequence of road network parts. Also, an ID is generated for a route; and any data received from the client that describes destination objects and the user is recorded.

The server-side route recording is presented in Figure 3. Having obtained data from the client, the server

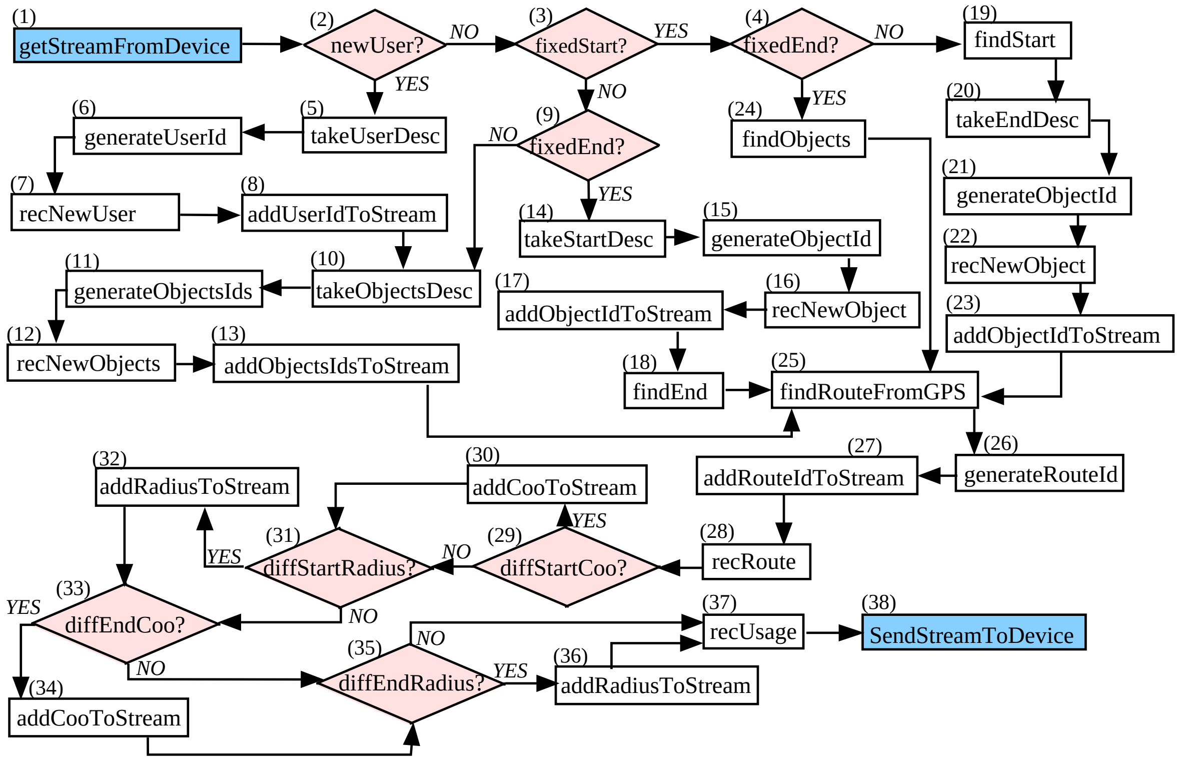


Figure 3: Server-Side Route Recording

checks if the user is new. If so, the server obtains the user's description from the stream, assigns an ID to the user, stores this information, and includes the user's ID in the stream for the client.

Next, the server considers the destination objects. If both destination objects are undefined (which is the case if the user is new) the server extracts destination object information from the stream (10), generates IDs (11), records the new objects (12), and adds the IDs to the stream for the client (13). If only one object is undefined, the steps are done for one object. If the start is undefined (3, 9), data about it is prepared (14, 15) and recorded (16, 17). Then the end object is identified using knowledge about the user's objects (18). Similar steps are taken if only the end object is undefined. If both objects are defined, they are identified using stored data (24).

Finally, the server analyses the third part of the stream that includes the standard data. The server detects the route from the GPS information (25), generates an ID for the route (26), adds this ID to the data stream for the client (27), and records the route in the database (28). The server also adds center coordinates of destination objects (30, 34) and/or their radiuses (32, 35) if the coordinates differ (29, 33) from the first/last GPS point in the GPS stream, and/or if the radiuses are not the default values (31, 35). Then the server records the first usage time of the route (37). The constructed stream is sent to the client to end the route recording (38).

3 Road Networks and Routes

We proceed to define the key data structures used for the capture of routes.

We project the real road network into 2D space and represent the result as a set of polylines, each of which is given by a sequence of *base* points $B \subset \mathbb{R}^2$. Different choices of base points lead to different road-network representations. Using many base points generally results in a higher-fidelity representation. A polyline is defined as $PL = \{(b_1, ..., b_N) \mid b_i \in B \wedge N \geq 2\}$, where b_1 and b_N is the start and end base point of the polyline, respectively.

Example 3.1 Figure 4 illustrates two intersecting polylines: $PL_1 = (b_1, b_2, b_3, b_4)$ and $PL_2 = (b_5, b_6, b_7)$. The start point of PL_1 is b_1 and the end point is b_4. □

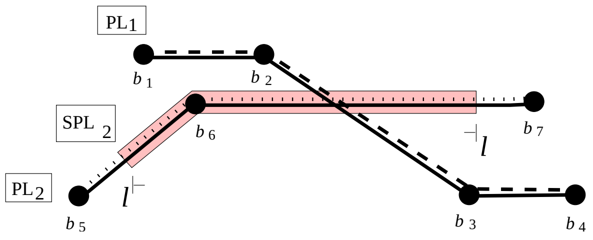


Figure 4: Example of Polylines and a Subpolyline

In our road network model, each polyline represents a bidirectional road. Without reference to the traffic directions of the roads, polylines have “directions” going from the start base points to the end base points.

We also reference the points on a road by their distance from the start of the road. Although a road’s geographical extent is approximated by a polyline, computing distances by simply summing up the Euclidean distances of segments is too inaccurate [7, 14, 22]. Rather, we assume that we have accurate distances for all or some of the base points in the polyline approximation of a road. This decouples the polyline representation of a road from the capture of distances along the road and is in keeping with current road-management practice. Using real road distances makes calculations more precise.

The measure of a base point b_i is given as l_i. The measure associated with the last base point of the polyline indicates the road length of the polyline.

If a measure is absent for a base point b_k of the polyline, we identify the base points b_i and b_j that are the nearest base points with measures before and after b_k, respectively, and we approximate the measure of b_k as follows:

$$l_k = l_i + \frac{(l_j - l_i)\sum_{n=i}^{k-1} |\overline{b_n b_{n+1}}|}{\sum_{m=i}^{j-1} |\overline{b_m b_{m+1}}|}$$

If no b_j exists, we use the Euclidean distance starting from b_i and onwards.

Example 3.2 Figure 5 exemplifies length calculation for base points of polyline $PL_1 = (b_1, b_2, b_3, b_4)$. The numbers above the line segments indicate the Euclidean distances between base point pairs. The numbers below base points hold the more accurate measures supplied by the road information provider.

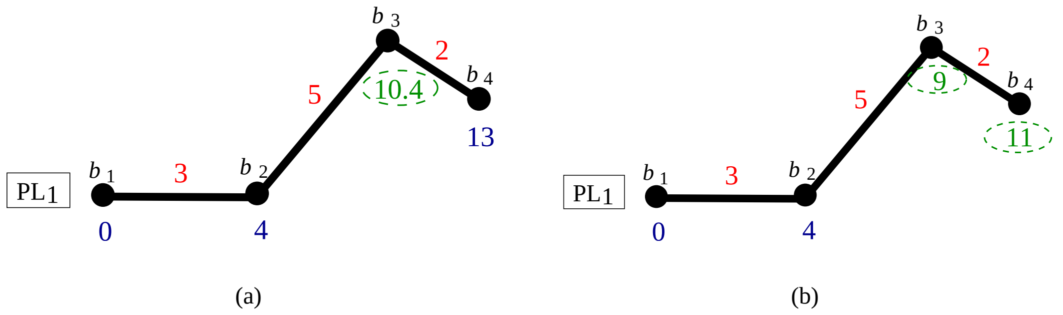


Figure 5: Length Calculations

Consider Figure 5(a). When computing the measure l_3 for b_3, $i = 2$ and $j = 4$. It may be verified that application of the formula yields $l_3 = 10.4$.

Figure 5(b) lacks measures for the last two base points, b_3 and b_4. The measure for b_3 is calculated by adding the Euclidean distance between b_2 and b_3, i.e., 5, to the measure of b_2, i.e., 4. For the base point b_4, we add the Euclidean distance between b_3 and b_4. $\square$

Definition 3.1 (Length) Function $\mathcal{L} : PL \times B \rightarrow \mathbb{R}$ takes as arguments a polyline $pl = (b_1, ..., b_N)$ and a base point b_i, $1 \leq i \leq N$, and it returns the road distance from the start of the polyline to the base point. $\square$

Here, $\mathcal{L}(pl, b_1) = 0$, and $\mathcal{L}(pl, b_N)$ is the length of the polyline. For $1 \leq i < j \leq N$, $\mathcal{L}(pl, b_j) - \mathcal{L}(pl, b_i)$ is at least the Euclidean distance between b_i and b_j. Next, a *subpolyline* models a part of a road.

Definition 3.2 (Subpolyline) Let $SPL \subset PL \times \mathbb{R}^2$ be a finite set of *subpolylines*. A subpolyline $spl = (pl, l^{\vdash}, l^{\dashv})$, where $0 \leq l^{\vdash} < l^{\dashv} \leq \mathcal{L}(pl, b_N)$, is the part of polyline pl that starts at measure $l^{\vdash}$ and ends at measure $l^{\dashv}$. $\square$

In Figure 4, the accentuated part of polyline PL_2 is a subpolyline, SPL_2. We proceed to capture the connectivity among the roads.

Definition 3.3 (Connection) Let $C \subset \{ \{(pl_1, l_1^{\vdash}), \ldots, (pl_N, l_N^{\vdash})\} \mid (pl_i, l_i^{\vdash}) \in PL \times \mathbb{R} \wedge N \geq 2\}$. Thus, C is a set of finite sets of *connections*. $\square$

Consider Figure 6(a), where polylines PL_1 and PL_2 each has a connection point at their intersection. There is a connection point at distance $l_1^{\vdash}$ from the start of PL_1, and there is one at distance $l_2^{\vdash}$ from the start of PL_2. We thus have $c = \{(PL_1, l_1^{\vdash}), (PL_2, l_2^{\vdash})\} \in C$. The connection points in Figures 6(b) and 6(c) are analogous, but illustrate situations where connection points coincide with base points. Note that when we capture the connections, we in effect obtain a graph representation of the road network.

As mentioned previously, our service users travel from and to destinations via the road network. These destinations, we term *user objects*.

Definition 3.4 (User Object) Let UO be a finite set of *user objects*. Each user object uo is a 3-tuple $(u, circle, spls)$, where

1) u belongs to U, the set of service users.
2) $circle = (x_0, y_0, rd) \in \mathbb{R}^2 \times \mathbb{R}$ denotes the circle defined by $(x - x_0)^2 + (y - y_0)^2 = rd^2$.
3) $spls = \{(pl, l^{\vdash}, l^{\dashv}) \mid \exists pl \in PL \ ((pl, l^{\vdash}, l^{\dashv}) \in getSpls(pl, circle))\}$, where function $getSpls$ returns the set consisting of all maximum subpolylines of $spls$ that are inside $circle$. $\square$

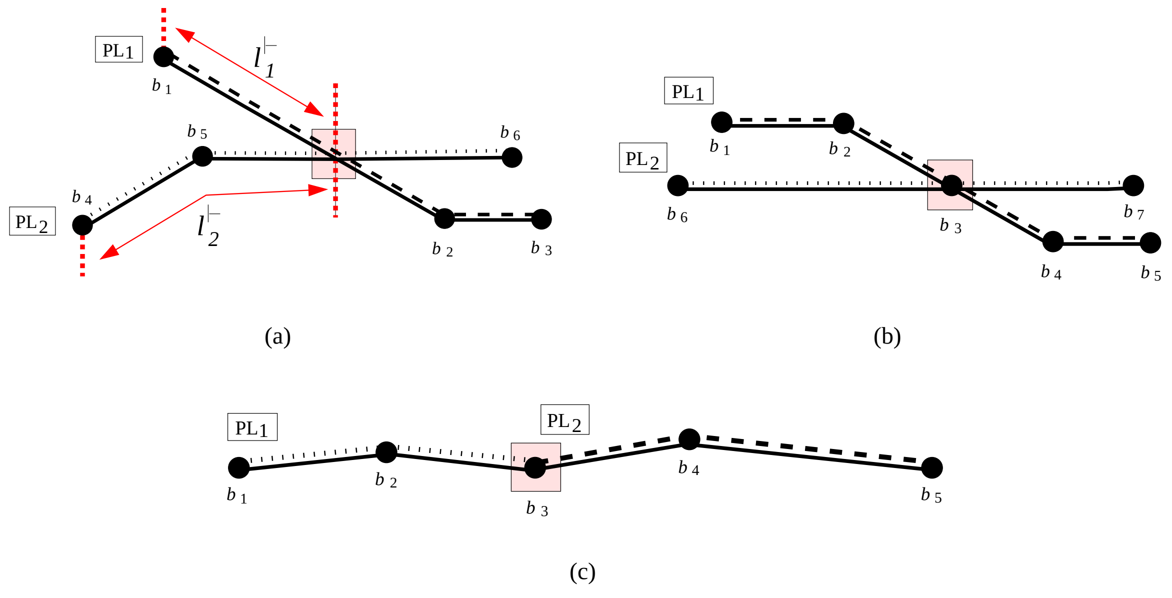


Figure 6: Connections Among Polylines

We say that user object uo belongs to user u and is located in the circular area with center (x_0, y_0) and radius rd.

Note that while it is simpler to model user objects as points than as circular areas, this is not appropriate. For example, each day a user may park in a different parking space in the same parking lot or even in a different parking lot close to the building where the user works. Thus, the same destination may have different route end and start locations on different days. Destination objects can be given different radiuses that depend on the usage patterns and the number of polylines around them.

Next, we associate usage times with routes. To be able to capture regularities in route uses, we capture the year, month, day, hour, minute, and second of each use separately. (Recall that the usage time of a route is the time when the use is initiated.)

Definition 3.5 (Usage Time) Let a *usage time* T be a finite set of 6-tuples (y, m, d, h, mn, s), where y, m, d, h, mn, and s denote *year*, *month*, *day*, *hour*, *minute*, and *second*, respectively. □

With the preceding definitions in place, we can define the notion of a route *route*.

Definition 3.6 (Route) Let R be a finite set of *routes*. Each route is a 4-tuple (RE, uo_s, uo_e, ST), where

1) $RE = ((spl_1, dir_1), \ldots, (spl_N, dir_N))$ is the sequence of subpolylines that makes up the route. For (spl_i, dir_i), where $spl_i = (pl_i, l_i^{\vdash}, l_i^{\dashv}) \in SPL$, dir_i is the motion direction along pl_i used:

$$dir_i = \begin{cases} 1 & \text{if the motion direction on subpolyline } spl_i \\ & \text{coincides with the direction of polyline } pl_i \\ -1 & \text{otherwise} \end{cases}$$

2) $uo_s = (u, circle_s, spls_s) \in UO$ is the start object of the route, and $\exists (pl, l^{\vdash}, l^{\dashv}) \in spls_s\ (pl = pl_1 \wedge (l^{\vdash} \leq l_1^{\vdash} \leq l^{\dashv} \wedge dir_1 = 1) \vee (l^{\vdash} \leq l_1^{\dashv} \leq l^{\dashv} \wedge dir_1 = -1))$.

3) $uo_e = (u, circle_e, spls_e) \in UO$ is the end object of the route, and $\exists (pl, l^{\vdash}, l^{\dashv}) \in spls_e (pl = pl_N \wedge (l^{\vdash} \leq l_N^{\dashv} \leq l^{\dashv} \wedge dir_N = 1) \vee (l^{\vdash} \leq l_N^{\vdash} \leq l^{\dashv} \wedge dir_N = -1))$.

4) $\forall spl_i = (pl_i, l_i^{\vdash}, l_i^{\dashv}), spl_{i+1} = (pl_{i+1}, l_{i+1}^{\vdash}, l_{i+1}^{\dashv}), 1 \leq i \leq N-1$ $((pl_i \neq pl_{i+1} \wedge \exists c \in C\ ((pl_i, l_1) \in c \wedge (pl_{i+1}, l_2) \in c)) \vee (pl_i = pl_{i+1} \wedge l_1 = l_2))$ where $l_1 = l_i^{\dashv}$ if $dir_i = 1$, and $l_1 = l_i^{\vdash}$ if $dir_i = -1$; $l_2 = l_{i+1}^{\vdash}$ if $dir_{i+1} = 1$, and $l_2 = l_{i+1}^{\dashv}$ if $dir_{i+1} = -1$.

5) $ST \subset T$ denotes the times when the route was used by user u. □

Thus, a route is a sequence of subpolylines with directions (item 1 in the definition), where the first/last subpolyline must intersect with the circle of the start/end destination objects (items 2 and 3) and where the sequence of subpolylines must form a (continuous) polyline (item 4).

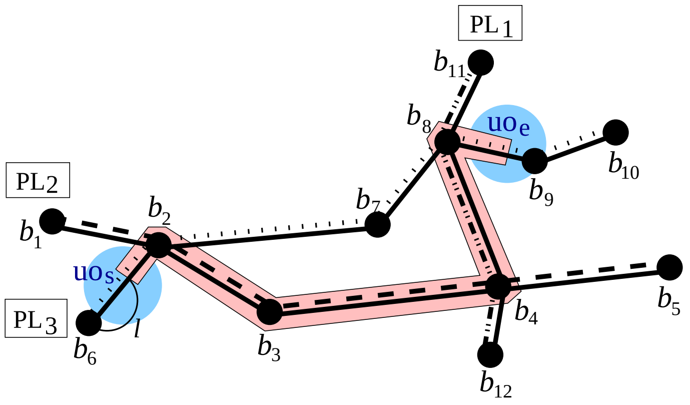


Figure 7: Example Route

Example 3.3 Figure 7 illustrates a road network with three polylines—$PL_1 = (b_{11}, b_8, b_4, b_{12})$, $PL_2 = (b_1, b_2, b_3, b_4, b_5)$, and $PL_3 = (b_6, b_2, b_7, b_8, b_9, b_{10})$. The highlighted route $r = (RE, uo_s, uo_e, ST)$ uses parts of all three polylines. Specifically, RE is a sequence of four route elements. The subpolyline of the first route element is given by $(PL_3, l, \mathcal{L}(PL_3, b_2))$, where l is a measure along subpolyline specifying a point that is in the circular area of user object uo_s. The movement direction of the subpolyline coincides with the direction of polyline PL_3. □

4 Route Construction Techniques

Techniques are first presented that identify the polylines on which a user travels. Then Sections 4.2 and 4.3 in turn cover algorithms that identify the subpolyline that are the elements of a route, and that combine such elements into entire routes.

We distinguish between GPS positions and points. Thus, "GPS position" refers to the NMEA sentences generated by a typical GPS receiver, and "GPS point" refers to the coordinate pair (x, y) that is part of the GPS position. The algorithms in this section only use point information; thus, they use GPS points.

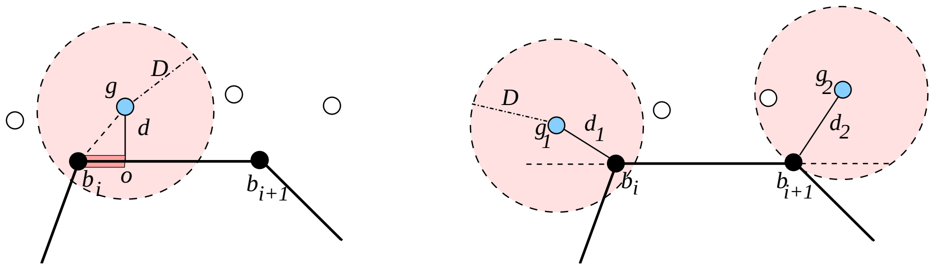
4.1 Polyline Identification

The first step in creating a route from the data received from a client is to identify the polylines on which the client is moving and the client's positions on the polylines.

We assume that GPS positions are imprecise; specifically, we assume that GPS positions are within distance D of the true position.

Point Projection onto Line Segment. Subsequent algorithms need to project a GPS position onto a line segment of a polyline. The projection must be expressed as a measure along the polyline, and the distance between the GPS position and its projection must be computed.

Figure 8 illustrates three cases for this projection. The line segment is $\overline{b_i b_{i+1}}$. Small circles indicate a sequence of GPS points, of which g, g_1, and g_2 are of interest. The large circles indicate the imprecision of the GPS points. A GPS point can be "during" the line segment, as is g in Figure 8(a). Its projection is position o, and the Euclidean distance between positions o and g is $d < D$. The coordinate can also be



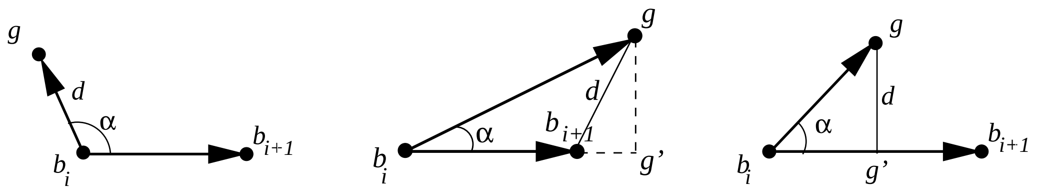
(a) During (b) Before and After

Figure 8: Projection of a GPS Position onto a Segment

before or after the polyline segment, as are g_1 and g_2 in Figure 8(b). In these cases, the projections of the GPS points are the end points b_i and b_{i+1} of the segment.

We need the distance d from a GPS point to its projection—this is used to determine the projection of a GPS point into the road network. Further, we need the distance $l^{\vdash}$ from the start of the polyline to the projection.

Distance d is calculated using vector algebra. A line segment $\overline{b_i b_{i+1}}$ is part of a polyline $(b_1, \ldots, b_N)$. The segment inherits the direction of the polyline. With b_i being the start point, we construct two vectors that emanate from b_i. One vector ends at b_{i+1}; the other ends at the GPS point g (see Figure 9). The angle



(a) Obtuse (b) Acute—I (c) Acute—II

Figure 9: Angles α and Projections

α between these vectors is used in the calculation of distance d. It is calculated using scalar multiplication:

$$\overline{b_i b_{i+1}} \cdot \overline{b_i g} = |\overline{b_i b_{i+1}}||\overline{b_i g}| \cos \alpha$$

If the angle is obtuse (Figure 9(a)), distance d is the Euclidean distance from the GPS point to the start of the segment. The measure of the projected point is that of the segment's start point:

If $90^\circ < \alpha < 270^\circ$ **then** $d = |\overline{g b_i}|, l^{\vdash} = \mathcal{L}(pl, b_i)$

If the angle is acute, there are two possibilities, as shown in Figures 9(b) and 9(c)). If the length of the projection of $\overline{b_i g}$ onto $\overline{b_i b_{i+1}}$, $|\overline{b_i g'}|$ exceeds $|\overline{b_i b_{i+1}}|$ (see Figure 9(b)), distance d is the distance between the end point b_{i+1} of the segment and the GPS point g. The measure of the projected point is that of the segment's end point:

$$\textbf{If } -90^\circ \leq \alpha \leq 90^\circ \wedge |\overline{b_i g'}| \geq (\mathcal{L}(pl, b_{i+1}) - \mathcal{L}(pl, b_i)) \textbf{ then } D = |\overline{g b_{i+1}}|, l^{\vdash} = \mathcal{L}(pl, b_{i+1})$$

If the projection length $|\overline{b_i g'}|$ is less than the length of the segment (Figure 9(c)), distance d is the perpendicular distance between the GPS coordinate to the polyline segment. The measure of the projected point is the sum of the projection length and the measure of b_i:

$$\textbf{If } -90^\circ \leq \alpha \leq 90^\circ \wedge |\overline{b_i g'}| < (\mathcal{L}(pl, b_{i+1}) - \mathcal{L}(pl, b_i)) \textbf{ then } d = |\overline{g g'}|, l^{\vdash} = \mathcal{L}(pl, b_i) + |\overline{b_i g'}|$$

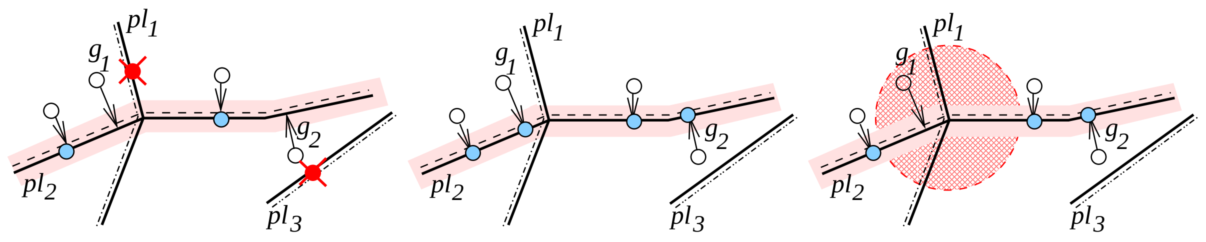
We encapsulate the computations described above in a function $calcParam$ (see Algorithm 4.1). It takes a triple (g, pl, pls) as argument and returns a pair $(d, l^{\vdash})$, where d is the distance from GPS point g to line segment pls on polyline pl, and $l^{\vdash}$ is the measure along pl of the projection of g onto pls.

Algorithm 4.1 Calculation of Projection Parameters (function $calcParam$)

```
Require: INPUT:  $g = (x, y) \in \mathbb{R}^2, pl \in PL, pls = ((x_1, y_1), (x_2, y_2)),$ where $(x_i, y_i) \in pl$
         OUTPUT: $(d, l^{\vdash}) \in \mathbb{R} \times \mathbb{R}$
 1: $\vec{v}_1 = (vx_1, vy_1) \leftarrow (x_2 - x_1, y_2 - y_1); \vec{v}_2 = (vx_2, vy_2) \leftarrow (x - x_1, y - y_1)$
 2: $|\vec{v}_1| \leftarrow \sqrt{vx_1^2 + vy_1^2}; |\vec{v}_2| \leftarrow \sqrt{vx_2^2 + vy_2^2}$
 3: $\alpha \leftarrow \arccos((vx_1\ vx_2 + vy_1\ vy_2)/(|\vec{v}_1|\ |\vec{v}_2|))$
 4: if $90^\circ < \alpha < 270^\circ$ then
 5:   $d \leftarrow |\vec{v}_2|$
 6:   $l^{\vdash} \leftarrow \mathcal{L}(pl, (x_1, y_1))$
 7: else
 8:   $projection \leftarrow |\vec{v}_2|\ \cos\alpha$
 9:   $length \leftarrow \mathcal{L}(pl, (x_2, y_2)) - \mathcal{L}(pl, (x_1, y_1))$
10:   if $length \leq projection$ then
11:     $d \leftarrow \sqrt{(x - x_2)^2 + (y - y_2)^2}$
12:     $l^{\vdash} \leftarrow \mathcal{L}(pl, (x_2, y_2))$
13:   else
14:     $d \leftarrow |\vec{v}_2|\ \sin\alpha$
15:     $l^{\vdash} \leftarrow \mathcal{L}(pl, (x_1, y_1)) + projection$
16:   end if
17: end if
18: return $(d, l^{\vdash})$
```

We do better than mapping GPS positions to the nearest polyline with the lowest value of d. For example, when a GPS position is near a crossroads, the true polyline may be the one that crosses the nearest polyline. Another example occurs when roads are close. Figure 10(a) illustrates how two GPS points g_1 and g_2 are nearest to an incorrect road. To handle such cases correctly, we consider the mapping of the previous GPS positions for subsequent GPS positions.

Initial GPS Position. For the initial GPS position, there is no collected data to consider while mapping it to a polyline. The initial GPS position is mapped to a polyline if exactly one polyline exists that has a segment with projection distance $d \leq D$. If there are several polylines that satisfy this requirement, the position cannot be mapped until some later position has been mapped correctly. Function $polyCand$ (see



(a) Incorrect Mappings to Nearest Polylines (b) Correct Mapping to Polyline (c) Elimination of Candidate Polylines

Figure 10: Polyline Identification

Algorithm 4.2) takes a GPS point g as argument and returns a set of candidate polylines with measures of the projection of g onto the polyline segments. The function uses function $calcParam$.

Algorithm 4.2 Finding Candidate Polylines (function $polyCand$)

```
Require: INPUT:  $g \in \mathbb{R}^2$
         OUTPUT: $Cand = \{(pl, l^{\vdash}) | (pl, l^{\vdash}) \in PL \times \mathbb{R}\}$
 1: $Cand \leftarrow \emptyset$
 2: for all $pl_i = (b_{i_1}, ..., b_{i_{n_i}}) \in PL$ do
 3:   $d \leftarrow \infty, l \leftarrow \infty$
 4:   for all $pls_{i_j} = (b_{i_j}, b_{i_{j+1}})$ such that $1 \leq j \leq n_i - 1$ do
 5:     $(cD, cL) \leftarrow calcParam(g, pl_i, pls_{i_j})$
 6:     if $cD \leq D \wedge cD < d$ then
 7:       $(d, l) \leftarrow (cD, cL)$
 8:     end if
 9:   end for
10:   if $d \leq D$ then
11:     $Cand \leftarrow Cand \cup (pl_i, l)$
12:   end if
13: end for
14: return $Cand$
```

All line segments of each polyline are analyzed, but only the line segment of a polyline that is nearest to the point g is considered as a candidate. The function uses two temporary variables, d and l. Variable d stores the distance to the nearest line segment of the polyline. Variable l stores the distance from the start of the polyline to the projected point on the nearest line segment.

In line 5, the parameters (cD, cL) for each line segment spl_{i_j} are calculated. If the distance cD to the current line segment is less than or equal to the imprecision value D and less than the distance d to some previous line segment of the same polyline, the current distance cD along with cL, the measure of the projection, are stored (lines 6–8). The polyline that has line segments in the imprecision distance from the point g is included into the list of candidate polylines, $Cand$, (see lines 10–11).

Subsequent GPS Positions. For subsequent positions, to obtain a result consistent with the correct mapping exemplified in Figure 10(b), we map a GPS point g_j to a polyline considering the mapping of the previous GPS point g_{j-1}. Point g_j should be mapped to the same polyline as g_{j-1} or to a polyline that shares a connection point with the previous polyline.

Considering again Figure 10(a), we see that polyline pl_3 cannot be a candidate for the mapping of g_2 because it is not connected with pl_2. To avoid wrong mappings at connections, e.g., the mapping of g_1 to pl_1, we introduce so-called connections areas and do not map GPS positions inside these areas. This is illustrated in Figure 10(c).

Function $polyId$ (see Algorithm 4.3) identifies the polyline pl for the GPS point g according to the polyline pPl that the previous GPS point is mapped to. The function returns the polyline and the distance from the start of the polyline to the projection. In lines 2–7, we check if the current GPS point is on the same

Algorithm 4.3 Polyline Identification (function $polyId$)

```
Require: INPUT:   $g \in \mathbb{R}^2, pPl = (b_1, \dots, b_n) \in PL$
         OUTPUT: $(pl, l^{\vdash}) \in PL \times \mathbb{R}$
 1: $(pl, l^{\vdash}) \leftarrow (pPl, \infty)$, $d \leftarrow \infty$
 2: for all $pl_j = (b_j, b_{j+1})$ such that $1 \leq j \leq n-1$ do
 3:     $(cD, cL) \leftarrow calcParam(g, pPl, pl_j)$
 4:     if $cD < d \land cD \leq D$ then
 5:         $l^{\vdash} \leftarrow cL$; $d \leftarrow cD$
 6:     end if
 7: end for
 8: if $d = \infty$ then
 9:     for all $pl_i = (b_{i_1}, ..., b_{i_{n_i}})$ such that $\exists\, c = (..., (pl_i, l_i), ..., (pPl, pL), ...) \in C$ do
10:         for all $pl_{i_j} = (b_{i_j}, b_{i_{j+1}})$ such that $1 \leq j \leq n_i - 1$ do
11:             $(cD, cL) \leftarrow calcParam(g, pl_i, pl_{i_j})$
12:             if $d = \infty \land cD \leq D$ then
13:                 $(pl, l^{\vdash}) \leftarrow (pl_i, cL)$; $d \leftarrow cD$
14:             else if $pl_i = pl \land cD < d$ then
15:                 $l^{\vdash} \leftarrow cL$; $d \leftarrow cD$
16:             else if $pl \neq pl_i \land d \leq D \land cD \leq D$ then
17:                 $(pl, l^{\vdash}) \leftarrow (\perp, \infty)$
18:                 return $(pl, l^{\vdash})$
19:             end if
20:         end for
21:     end for
22: end if
23: return $(pl, l^{\vdash})$
```

polyline as the previous GPS point. The distance cD to every segment of the polyline pPl is calculated, and the shortest one is chosen. But the distance also has to be less than D, the imprecision value. If this search yields an empty result (line 8), we assume that the GPS point should be mapped to a polyline that connects with the previous polyline. Thus, in lines 9–21 the function searches for polylines that are in distance D from the GPS point and intersect with pPl. If there are more than one (lines 16–18) or no such polylines, the function returns an undefined polyline and an infinite distance. The second case means that there is a gap in the GPS data, which then has to be filled in. If two segments of the polyline that intersects with pPl are within distance D of the GPS point (the second condition in line 14) then the nearest segment is chosen.

To determine whether a GPS point is in a connection area, we first need the result of function $polyId$, namely the polyline the GPS point is projected onto and the distance of the projection from the start of the polyline. Connections are given by their distance from the start of a polyline (see Definition 3.3). Thus, if the projection is within distance D from a connection on its polyline, the GPS point is in a connection area.

Later, we ignore projections of such positions.

Function $possibleConnection$ (see Algorithm 4.4) determines whether an argument GPS point is in a connection area. All connections related to the polyline are analyzed, and the distance from the projection

Algorithm 4.4 Connection Area (function $possibleConnection$)

```
Require: INPUT:  $(pl, l^{\vdash}) \in PL \times \mathbb{R}$
         OUTPUT: $conn \in \{\text{true}, \text{false}\}$
 1: $conn \leftarrow$ false
 2: for all $c_i = \{cc_1, ..., cc_n\} \in C$ such that $\exists\ cc_{i_j} = (pl, l^{\vdash}_{i_j}) \in c_i$ do
 3:   for all $cc_{i_j} \in c_i$, such that $cc_{i_j} = (pl, l^{\vdash}_{i_j})$ do
 4:     if $-D \leq l^{\vdash}_{i_j} - l^{\vdash} \leq D$ then
 5:       $conn \leftarrow$ true
 6:       return $conn$
 7:     end if
 8:   end for
 9: end for
10: return $conn$
```

to each connection is calculated. A distance less than imprecision D makes the result *true*; otherwise, the function returns *false*.

4.2 Formation of a Route Element

Recall that a route is a sequence of connected subpolylines, which thus combine to form a single polyline. We proceed to describe how subpolylines are constructed.

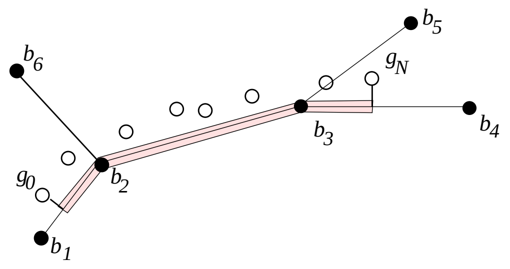
Routes are formed by four main kinds of subpolylines, as illustrated in Figure 11 and explained in the following. As in previous illustrations, the unfilled circles in the figure denote GPS points. There are three polylines in the figure: (b_1, b_2, b_3, b_4), (b_6, b_2), and (b_3, b_5); and a route is emphasized.

Figure 11(a) illustrates the simplest case of a route, one that consists of only a single subpolyline. According to our model, we never approximate the start and end of a route, but always fix the exact positions of these. This means that when we form such a subpolyline, we consider the first and the last GPS points that can be mapped correctly, i.e., g_0 and g_N in Figure11(a). The distances from the start of the polyline identify the part of the polyline that constitutes the route, and the movement direction defines the start and end.

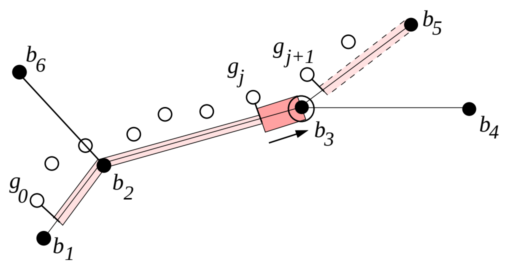
Figure 11(b) illustrates how a first subpolyline is formed. For such a subpolyline, the measure that corresponds to the start of the route is exactly the measure of the projection of the first correctly mapped GPS point (g_0).

The other measure of the subpolyline usually has to be modified slightly so that it becomes equal to the measure of the connection where the route switches to a different polyline. To illustrate this, GPS points $g_i, 0 \leq i \leq j$, in the figure are projected onto the same polyline, but point g_{j+1} is projected onto another polyline. However, the route can only switch to the new polyline at b_3; thus, the measure of the projection of g_j is approximated as the measure of the nearest intersection with the other polyline, i.e., $\mathcal{L}((b_1, b_2, b_3, b_4), b_3)$.

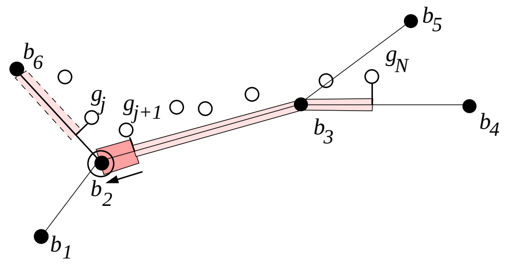
Next, Figure 11(c) illustrates how the last subpolyline of a route is formed. This case is similar, but opposite, to the case of the first subpolyline. If the movement direction is the same as the direction of the polyline, the start of the last subpolyline is approximated, as for g_{j+1} in the figure. The end of the subpolyline is determined by the last correctly mapped position of the route, i.e., g_N. If the direction is opposite, the measures are formed in the opposite way.



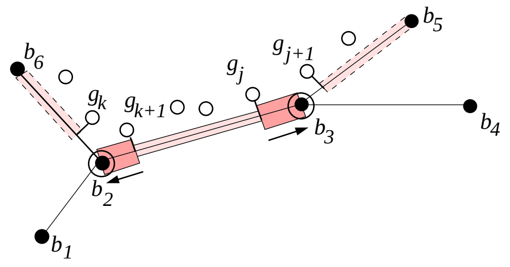
(a) Only One Subpolyline



(b) First Subpolyline



(c) Last Subpolyline

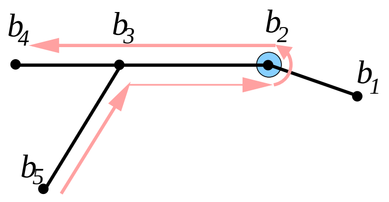


(d) Intermediate Subpolyline

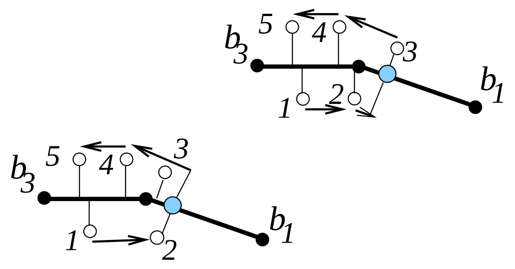
Figure 11: Kinds of Subpolylines

Finally, Figure 11(d) shows how an intermediate subpolyline is formed. This case occurs if the GPS points of the route are mapped to more than two polylines. The start and the end measures for such a subpolyline are not those of projections, but must be approximated to the measures of the connections at which the route changes polyline. In the figure, GPS points g_{k+1} and g_j are the first and last GPS points that are projected onto the intermediate polyline. Their distance values from the start of the polyline are approximated to values of the connections b_2 and b_3, i.e., $\mathcal{L}((b_1, b_2, b_3, b_4), b_2)$ and $\mathcal{L}((b_1, b_2, b_3, b_4), b_3)$.

If no neighboring subpolylines exist that belong to the same polyline, only these four cases exist. However, it is possible for a route to have neighboring subpolylines that belong to the same polyline, but have opposite directions. This happens if the user makes a u-turn. Figure 12(a) demonstrates this. In this case,



(a) Movement



(b) Formation

Figure 12: Subpolylines on the Same Polyline

the end of one subpolyline is the start of the next. Figure 12(b) offers additional detail. GPS points are numbered to indicate their order. The last point that is in the same direction as the current subpolyline is the start of the new subpolyline. This GPS point can be still on the same side of the polyline (before turning around), or it can be on the other side (after turning around).

We use a function $defineDirection$ (see Algorithm 4.5) that determines the movement direction along a polyline for two consecutive projections. The function takes the measures of the projections of the previous and current GPS points as arguments. It also considers the movement direction on the polyline until the previous GPS point. If the previous measure $pDst$ is less than the current one $cDst$, the direction coincides with the polyline's direction and is set to 1. If the previous measure is greater than the current one, the direction is the opposite and is set to -1. If the two measures are equal, the direction is set to the previous direction. This last situation happens if, e.g., the user is stuck in a traffic jam and moves so slowly that consecutive GPS points are the same.

Algorithm 4.5 Direction Identification (function $defineDirection$)

```
Require: INPUT:  $pDst, cDst \in \mathbb{R}, pDir \in \{-1, 0, 1\}$
         OUTPUT: $direction \in \{-1, 0, 1\}$
1: $direction \leftarrow 0$
2: if $pDst < cDst$ then
3:     $direction \leftarrow 1$
4: else if $pDst > cDst$ then
5:     $direction \leftarrow -1$
6: else
7:     $direction \leftarrow pDir$
8: end if
9: return $direction$
```

Movement directions are used to approximate measures when we construct "uninterruptible" routes from sequences of subpolylines. Function $findEnd$ (see Algorithm 4.6) finds the measure $endDst$ along polyline pPl of the connection where it and the current polyline cPl intersect. The function uses the measure $pDst$ of the projection of the previous GPS point and also the direction $pDir$ on polyline pPl. The function chooses the nearest connection if there are more than one connection where the polylines intersect. Temporary variable $distToConn$ stores the measure of the nearest connection found so far. All connections c_i where pPl and cPl intersect are considered (line 2). Variable $cDistToConn$ is used for

Algorithm 4.6 End Identification for a Subpolyline (function $findEnd$)

```
Require: INPUT:  $pPl, cPl \in PL, pDst \in \mathbb{R}, pDir \in \{-1, 0, 1\}$
         OUTPUT: $endDst \in \mathbb{R}$
1: $distToConn, endDst \leftarrow \infty$
2: for all $c_i = \{cc_1, ..., cc_n\} \in C$ such that $\exists cc_{i_j} \wedge cc_{i_k} \in c_i \; (cc_{i_j} = (pPl, l^{\vdash}_{i_j}) \wedge cc_{i_k} = (cPl, l^{\vdash}_{i_k}))$ do
3:     $cDistToConn \leftarrow (l^{\vdash}_{i_j} - pDst)$
4:     if $((pDir = 1 \wedge cDistToConn \geq -2D) \vee (pDir = -1 \wedge cDistToConn \leq 2D)) \wedge |cDistToConn| < distToConn$ then
5:         $endDst \leftarrow l^{\vdash}_{i_j}$; $distToConn \leftarrow |cDistToConn|$
6:     else if $pDir = 0 \wedge |cDistToConn| < distToConn$ then
7:         $endDst \leftarrow l^{\vdash}_{i_j}$; $distToConn \leftarrow |cDistToConn|$
8:     end if
9: end for
10: return $endDst$
```

calculating the distance $pDst$ from the previously projected point to each suitable connection. A connection is suitable in two cases.

- If it is ahead of the projected point or behind with distance $2D$, when the direction coincides with the polyline's direction.
- If it is behind the projected point or ahead with distance $2D$, when the direction is the opposite (line 4).

Note that $cDistToConn$ will be negative when movement direction $pDir$ is 1, but the connection is in the direction opposite of $pDst$ (behind $pDst$), as well as when the movement direction is -1 and the connection is in the same direction (ahead of $pDst$). We use notation $|cDistToConn|$ to obtain a positive value for the distance to the connection. If the distance to the connection is less than the distance to the previous connection then the new connection's measure is the candidate end measure for the subpolyline. The distance to it is noted in variable $cDistToConn$. If the direction on the subpolyline is undefined, the nearest connection is simply chosen as a candidate.

Function $findStart$ (see Algorithm 4.7) is closely related to function $findEnd$. The next subpolyline starts where the previous subpolyline ends, so function $findStart$ defines where the next subpolyline starts on the polyline cPl according to the previous subpolyline that was on polyline pPl and ended at measure $pDst$. The function returns the $startDst$ that is the distance at which the current subpolyline starts.

Algorithm 4.7 Start Identification for a Subpolyline (function $findStart$)

```
Require: INPUT:  $pPl, cPl \in PL, pDst \in \mathbb{R}$
         OUTPUT: $startDst \in \mathbb{R}$
1: for $c = \{cc_1, ..., cc_n\} \in C$ such that $\exists cc_k, cc_m(cc_k = (pPl, pDst) \wedge cc_m = (cPl, l^{\vdash}_m))$ do
2:     $startDst \leftarrow l^{\vdash}_m$
3: end for
4: return $startDst$
```

Finally, function $formSubPoly$ (see Algorithm 4.8) creates a subpolyline that satisfies the requirements for a subpolyline. Specifically, the start measure of a subpolyline must be less than its end measure. While making calculations, it may happen that the start measure exceeds the end measure. Function $formSubPoly$ solves this problem. The function takes polyline a pl and temporary start and end measures, l_s and l_e, as input parameters. The function returns a correctly created subpolyline $spl = (pl, l^{\vdash}, l^{\dashv})$ along with a movement direction dir. If the input movement direction does not coincide with the direction of the polyline, i.e., $dir = -1$, the start measure is greater than the end measure, and they are exchanged (lines 1–2). If the movement direction coincides with the direction of the polyline, i.e., $dir = 1$, the start and end measures are final (lines 3–4). If the movement direction is unknown, i.e., $dir = 0$, the direction is set according to the start and end measures, and the subpolyline is constructed (lines 5–10).

4.3 Route Construction

In Section 2.2, we provided an overview of the context of the essential construction of routes that occurs on the server side. here, we proceed to describe route construction in some detail.

The state of the algorithm is captured by the data structure $cState = ((pPl, pDst, pDir), l^{\vdash}, RE)$, where pPl is the polyline that the most recent, previous GPS point was mapped to, $pDst$ is the distance from the start of that polyline to the position on the polyline this point was mapped, $pDir$ is the direction of movement along the polyline of the GPS sequence, $l^{\vdash}$ is the distance from the start of the polyline at which the current subpolyline starts, and RE is a sequence of route elements (see Figure 13).

Algorithm 4.8 Subpolyline Formation (function $formSubPoly$)

```
Require: INPUT:  $pl \in PL, l_s, l_e \in \mathbb{R}, dir \in \{-1, 0, 1\}$
         OUTPUT: $spl = (pl, l^{\vdash}, l^{\dashv}) \in SPL, dir \in \{-1, 0, 1\}$
 1: if $dir = -1$ then
 2:     $spl = (pl, l^{\vdash}, l^{\dashv}) \leftarrow (pl, l_e, l_s)$
 3: else if $dir = 1$ then
 4:     $spl = (pl, l^{\vdash}, l^{\dashv}) \leftarrow (pl, l_s, l_e)$
 5: else
 6:     if $l_s < l_e$ then
 7:         $spl = (pl, l^{\vdash}, l^{\dashv}) \leftarrow (pl, l_s, l_e)$; $dir \leftarrow 1$
 8:     else
 9:         $spl = (pl, l^{\vdash}, l^{\dashv}) \leftarrow (pl, l_e, l_s)$; $dir \leftarrow -1$
10:     end if
11: end if
12: return $(spl, dir)$
```

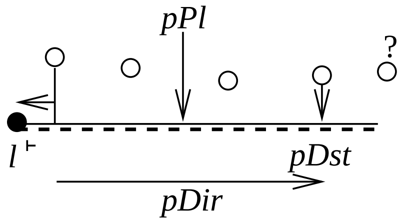


Figure 13: Current State of the Algorithm

The algorithm also uses a few additional structures. Specifically, $(cPl, cDst)$ stores the polyline to which the current GPS point is mapped and the distance from the start of the polyline to the point on the polyline to where it was mapped. Next, dir is the current direction on the polyline. We also use the well-known primitive functions **head**, **tail**, and **append** on sequences of elements of the same type.

Next, the algorithm employs a number of additional functions. First, function *getStartValues* (see Algorithm 4.9) scans the GPS sequence for the first position for which there is only one polyline in the road network that is within the distance of imprecision (lines 2–9). So, if the first point has more than one candidate polyline, the function considers the second one; if the second position has more than one candidate, the function considers the third one; etc. The function uses a data structure $undG = (g_1, \ldots, g_k)$, where the first $k - 1$ elements are undefined GPS points and g_k is the first GPS point that is mapped correctly. Next, $Cand$ is a set of pairs $(cPl_i, cDst_i)$ of a polyline and a distance from the start of the polyline. This set records candidate polylines for a particular GPS position (line 4). Finally, $cList = (Cand_1, \ldots, Cand_k)$ is a list of candidate sets where $Cand_i$ contains the candidates for mapping GPS point g_i.

For each point g from the GPS sequence, algorithm $getStartValues$ finds candidate polylines $Cand$ using function $polyCand$ (line 4). If more than one candidate exists (line 5), the algorithm adds the GPS point to list $undG$ and also adds the candidates $Cand$ to list $cList$. If the first point with only one candidate is not the first GPS point in the stream (line 10), the algorithm uses function $backtrack$ (see Algorithm A.2 in Appendix A.2) to map the previous points correctly, if possible, and to get the current state. If the first GPS point has only one candidate (line 12), the current state becomes this candidate. If all point in the GPS stream have more than one candidate polyline or no candidates (line 16), the algorithms exits. The function returns the current state of the algorithm and a part of the GPS stream that was not analyzed yet.

Algorithm 4.9 Function $getStartValues$

```
Require: INPUT:   $G = (g_1, \dots, g_n), g_i \in \mathbb{R}^2$
         OUTPUT: $(cState, G) = (((pPl, pDst, pDir), l^{\vdash}, RE), G)$
 1: $Cand \leftarrow \emptyset$; $cList \leftarrow$ nil; $undG \leftarrow$ nil
 2: while $G$ not empty $\wedge |Cand| \neq 1$ do
 3:     $g \leftarrow$ head$(G)$; $G \leftarrow$ tail$(G)$
 4:     $Cand \leftarrow polyCand(g)$
 5:     if $|Cand| > 0$ then
 6:         $cList \leftarrow$ append$(cList, Cand)$
 7:         $undG \leftarrow$ append$(undG, g)$
 8:     end if
 9: end while
10: if $|cList| > 1$ then
11:     $cState \leftarrow backtrack(cList, undG)$
12: else if $|cList| = 1$ then
13:     $(pPl, pDst) \leftarrow$ head$(Cand)$
14:     $pDir \leftarrow 0; l^{\vdash} \leftarrow pDst; RE \leftarrow$ nil
15: else
16:     EXIT
17: end if
18: return $(cState, G)$
```

Function *fillGap* (for details, see Algorithm A.4 in Appendix A.3) fills the gap between two projections based on shortest-path search in the road network representation. This function constructs missing route elements. The function takes the current state of the algorithm, the current undefined GPS coordinate, and the remaining GPS sequence as input parameters. The function returns the new state of the algorithm and the remaining part of the GPS sequence for further analysis.

Function $newSubOtherPoly$ (see Algorithm 4.10) constructs a route element when the current GPS point is mapped to a polyline that differs from the one the previous GPS point was mapped to. The end of the subpolyline for the route element being generated is modified so that it becomes equal to the measure of the connection where the object departed from the previous polyline to reach its new polyline.

Algorithm 4.10 Subpolyline Construction for Different Polylines (function $newSubOtherPoly$)

```
Require: INPUT:   $cState = ((pPl, pDst, pDir), l^{\vdash}, RE), cPl \in PL$
         OUTPUT: $cState$
1: $l^{\dashv} \leftarrow findEnd(pPl, cPl, pDst, pDir)$
2: $(spl, dir) \leftarrow formSubPoly(pPl, l^{\vdash}, l^{\dashv}, pDir)$
3: $RE \leftarrow$ append$(RE, (spl, dir))$
4: $l^{\vdash} \leftarrow findStart(pPl, cPl, l^{\dashv})$
5: $(pPl, pDst, pDir) \leftarrow (cPl, l^{\vdash}, 0)$
6: return $cState$
```

The input parameters of the function are the current state $cState$ of the algorithm and the polyline cPl, to which the current GPS point was mapped. The function returns a new state $cState$. The function creates a subpolyline for the GPS points mapped to the previous polyline pPl. The end of the subpolyline is found first (line 1). Then a subpolyline is created (line 2) and added to the sequence of route elements (line 3). The function prepares values for the construction of the new subpolyline. It finds a start distance value (line 4) and changes the elements of the current state of the algorithm (line 5), i.e., $pPl, pDst$, and $pDir$, that are the polyline, the previously mapped point, and movement direction for the next route element.

Next, function $newSubSamePoly$ (see Algorithm 4.11) constructs a new route element in the case where the movement is along the same polyline, but the movement direction from the previous position to the current is opposite to the direction until the previous position. The end of the previous route element is the start of the new one. The input parameters of the function are the current state $cState$ and the distance

Algorithm 4.11 Subpolyline Construction—Same Polyline (function $newSubSamePoly$)

```
Require: INPUT:   $cState = ((pPl, pDst, pDir), l^{\vdash}, RE), cDst \in \mathbb{R}$
         OUTPUT: $cState$
1: $l^{\dashv} \leftarrow pDst$
2: $(spl, dir) \leftarrow formSubPoly(pPl, l^{\vdash}, l^{\dashv}, pDir)$
3: $RE \leftarrow$ append$(RE, (spl, pDir))$
4: $l^{\vdash} \leftarrow pDst$
5: $(pDst, pDir) \leftarrow (cDst, (-1)\,dir)$
6: return $cState$
```

value $cDst$ along the polyline where the current GPS point was mapped to. The function returns a new state of the algorithm. The function creates a new route subpolyline. As the movement is on the same polyline, only the movement direction is different, the distance along the polyline where the previous GPS point was mapped to becomes the end distance for the constructed subpolyline (line 1). The route element is created (line 2) and added to the sequence of route elements (line 3). The end distance of the subpolyline is the start distance for the next subpolyline (line 4) because the movement direction was changed at that position. For the algorithm state, $cDst$ becomes the previous distance, and the direction is changed to the opposite (line 5).

One last function is needed by the overall route finding algorithm. Specifically, function $proceedEnd$ constructs the last element of a route. All last route elements that belong to the last polyline are approximated by one element if they are in the area of the same destination object. Thus, all the last route elements constructed so far that belong to the last polyline to which GPS points were mapped are approximated to one element if these route elements are in the area of the destination object. In Figure 14, the final point of the route is E and all subpolylines belong to the same polyline. They are inside the destination area shown by the circle. Each value x_i denotes a distance from the start of the polyline. Function *proceedEnd* (for details, see Algorithm A.1 in Appendix A.1) starts with the end position (E in the figure) and searches backwards for the start position that is the "oldest" position on the polyline. Each element inside the destination circle is considered in turn. If an element exceeds the circle, the approximation process stops. In the figure, we start with (x_1, E) and consider (x_1, x_2). This yields (x_2, E). We then consider (x_3, x_2), obtaining (x_3, E). Next, we obtain (E, x_4). The final result of the approximation is element (S, E).

We now have the elements needed by the main route construction algorithm. Taking a sequence G of GPS points as input, this algorithm (Algorithm 4.12, below) constructs a route consisting of a sequence RE of route subpolylines. Note that this algorithm employs map matching as part of its solution to a larger problem; other map matching techniques may be used in place of the specific technique employed by the algorithm.

With the functions presented earlier in this section at its disposal, Algorithm 4.12 first uses function $getStartValues$ to obtain a correct start state. While the GPS sequence is not empty, the next point is extracted and processed. The polyline that corresponds to the point is identified using function $polyId$. If this function returns an undefined polyline, a gap exists in the GPS sequence, which has to be filled. If the function returns a polyline, it is checked if the projection is in a connection area. If the point projection is not in the connection area, the subsequent calculations can be done.

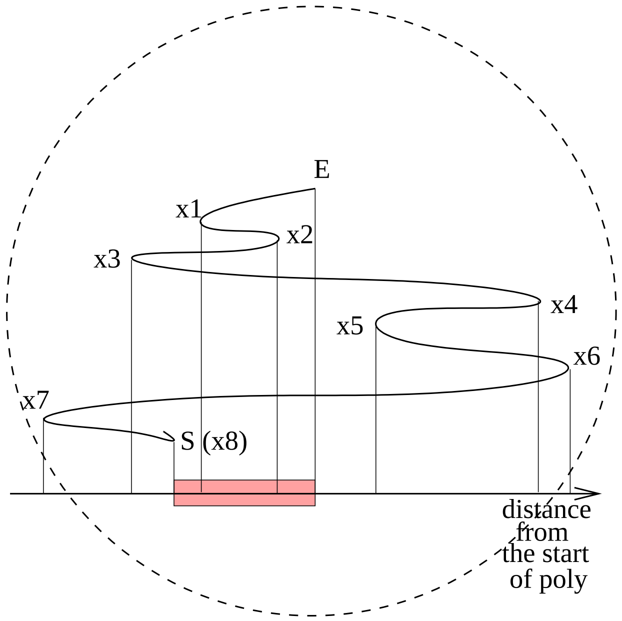


Figure 14: Approximation of the Route End

If the current polyline is not the same (line 10) as for the previous GPS point, a new subpolyline is formed. If the polyline is the same (line 12) as for the previous GPS point, the algorithm checks if the movement direction is the same as for the previous point. If the previous direction was undefined, its value is set to a value of the current direction. If the direction is the same, no calculations are done—only temporary variable $pDst$ is set to the distance of the current GPS point. If the direction is not the same, we have to form a new subpolyline, and function $newSubSamePoly$ is called.

When the GPS sequence is empty, the final route element is computed by function *proceedEnd*.

5 Experimental Validation

To validate the data structures and algorithms described in the previous two sections, these were implemented using generally available, state-of-the-art technologies, including Java, Oracle PL/SQL, and Oracle Spatial. We describe this implementation and lessons learned from testing the implementation using a real-world, digital representation of a real road network together with GPS log data obtained from vehicles.

5.1 Database Schema

Figure 15 contains a relational schema capable of capturing the data structures described in Section 3. Primary and foreign keys are indicated. Table **LINEAR_ELEMENTS** stores the main elements representing roads of the road network, namely polylines. Each tuple in this table contains the unique ID of a polyline and the length of the polyline.

Table **CONNECTIONS** captures the intersections among polylines. A tuple in this table records that a polyline (POL_ID) intersects at a distance (POL_FROM) from its start with one or several polylines at a connection (CONN_ID).

Recall from Section 3 that a polyline is given by a sequences of base points—these are recorded in table **POLYLINE_ELEMENTS**. A tuple records a base point of a polyline (POL_ID). The number of the base point in the sequence of the base points of the polyline (SEQUENCE_NR) and its distance from the

Algorithm 4.12 Route Finding

```
Require: INPUT:  $G = (g_1, ..., g_n), g_i \in \mathbb{R}^2, n > 1$
         OUTPUT: $RE = ((spl_1, dir_1), ..., (spl_m, dir_m)),\ spl_i = (pl_i, l_i^{\vdash}, l_i^{\dashv}) \in SPL$
1:  let $cState = ((pPl, pDst, pDir), l^{\vdash}, RE)$
2:  $(cState, G) \leftarrow getStartValues(G)$
3:  while $G$ is not empty do
4:    $g \leftarrow$ head($G$); $G \leftarrow$ tail($G$)
5:    $(cPl, cDst) \leftarrow polyId(g, pPl)$
6:    if $cPl = \perp$ then
7:      $(cState, G) \leftarrow fillGap(cState, g, G)$
8:    else
9:      if $possibleConnection(cPl, cDst) =$ false then
10:       if $cPl \neq pPl$ then
11:         $cState \leftarrow newSubOtherPoly(cState, cPl)$
12:       else
13:         $dir \leftarrow defineDirection(pDst, cDst, pDir)$
14:         if $pDir = 0$ then
15:           $pDir \leftarrow dir$
16:         else if $pDir = dir$ then
17:           $pDst \leftarrow cDst$
18:         else
19:           $cState \leftarrow newSubSamePoly(cState, cDst)$
20:         end if
21:       end if
22:     end if
23:   end if
24: end while
25: $RE \leftarrow proceedEnd(cState, cDst)$
26: return $RE$
```

start of the polyline (POL_FROM) are recorded, in addition to the geographical coordinates (X_COORD and Y_COORD) of the base point.

Table **SDO_POLYLINE_ELEMENTS** is created to be able to use facilities in Oracle Spatial [19]. The attributes in this table are similar to those in table **POLYLINE_ELEMENTS**. The exception is attribute ELEMENT, which does not capture the geo-information about a single base point, but captures an entire line segment with its start and end points.

A tuple in table **USERS** contains the unique ID of a mobile service user and additional information about the user.

Next, a tuple in table **DESTINATION_OBJECTS** contains the ID of a destination object, the ID of the user to whom the object belongs, a description of the object, and attributes that specify the circular area of the object. Table **SDO_DESTINATION_OBJECTS** is created to be able to use Oracle Spatial. It has an attribute CIRCLE instead of coordinates.

Three tables and a view are used for capturing routes. First, table **ROUTES** records the routes of the mobile service users. Routes start and end at destination objects. A tuple thus records the ID of a route and the start and end objects.

Second, table **ROUTE_ELEMENTS** describes routes in terms of their elements. Each tuple thus describes a subpolyline. Attribute POL_FROM records the start measure of the subpolyline and attribute

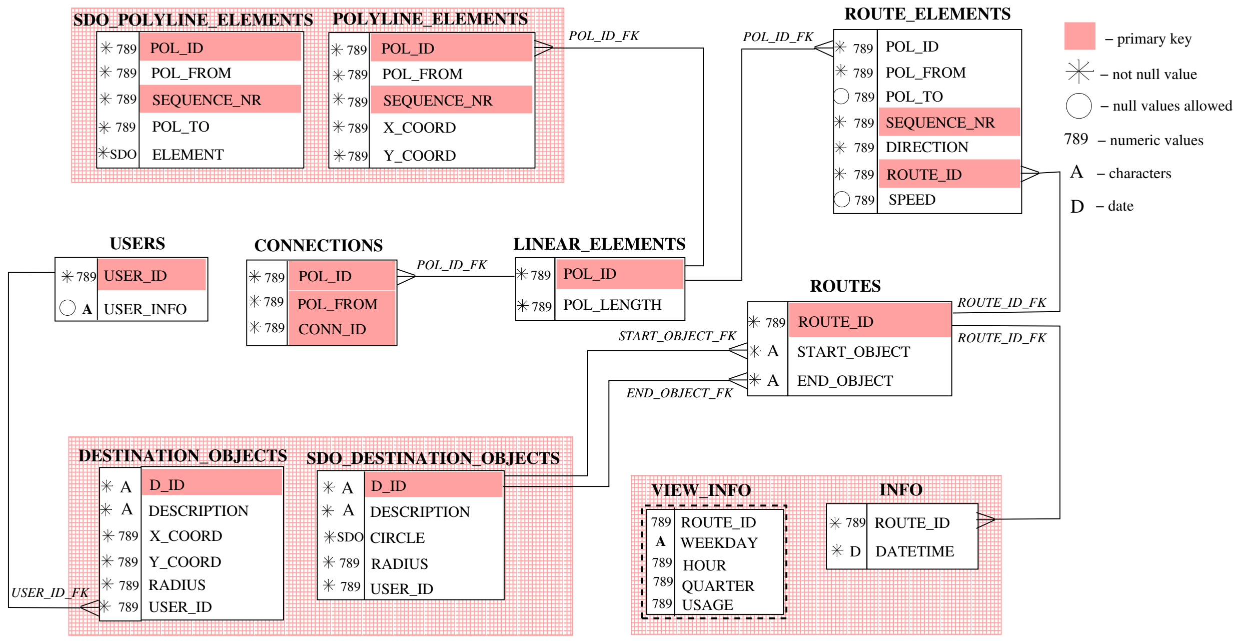


Figure 15: Relational Database Schema

POL_TO captures the end measure of the subpolyline. The number of the subpolyline in the sequence of subpolylines that make up the route it is part of is recorded by attribute SEQUENCE_NR. Attribute DIRECTION indicates whether the direction of the polyline coincides with the direction of the route on that polyline. Attribute SPEED captures the average speed of the user on the subpolyline.

Third, table **INFO** captures the usages of routes. A tuple in this table corresponds to an individual usage of a route and thus captures the ID of a route and the time of the use. A view **VIEW_INFO** is included that contains the attributes ROUTE_ID, WEEKDAY, HOUR, QUARTER, and USAGE. This view approximates the exact route usage times down to quarters of an hour. Attribute USAGE records the sum of uses of a route during a particular quarter on a particular day of the week.

5.2 Implementation Overview

Based on the database schema just described, the algorithms described in the previous section were implemented using facilities available in Oracle Spatial [19]. Segments of polylines are spatial data objects (SDO elements in Figure 15), and Oracle Spatial operators and geometry functions are used. Polyline segments are also linear referencing system (LRS) elements, which enables the use of LRS functions. To use the Oracle Spatial functions, we create an index on the spatial attribute. A spatial attribute is constructed according to the syntax of the object MDSYS.SDO_GEOMETRY.

The route finding algorithm implemented with Oracle Spatial differs a bit from the one described in Section 4. The implementation is in Java, and JDBC is used to execute SQL queries enhanced with Oracle Spatial functionality.

The built-in Java class *LinkedList* is used for storing the sequences of subpolylines that form routes. This class comes with standard list manipulation operations. The implementation uses a separate class that is responsible for the execution of SQL queries. The class that is responsible for route finding includes an instance of this class, to be able to obtain the results of SQL queries.

To identify polylines for subsequent GPS positions, we use a PL/SQL function *polyId*. This function first considers the polyline that the previous GPS position was mapped to. If the distance to that polyline exceeds the imprecision, the function searches for the nearest, connected polyline. Two Oracle Spatial operator are used. Operator SDO_NN finds the nearest spatial objects (polylines), and operator SDO_NN_DISTANCE returns the distances to these objects. We used 30 meters as the imprecision value for GPS positions and as the imprecision value of connection areas.

5.3 Map and GPS Log Data

The proposals presented in the previous sections represent the results of repeated cycles of testing and improvement of the route-recording prototype.

The testing was done using the INFATI data [17]. This data includes a digital representation of the road network of the municipality of Aalborg, Denmark. This data is quite typical of road network representations. The data is captured in a database with the schema just described. The INFATI data also includes GPS logs from twenty-some vehicles that participated in an intelligent speed adaptation project. Briefly, the position of a vehicle was logged every second when the vehicle was moving. Positions were logged for approximately six weeks.

5.4 Experimental Insights

In general, the experimental validation of the prototype component led to a more consolidated formalization of the concepts underlying the component and led to a more mature component that is able to handle the complex situations that occur in real-world applications. Here, we discuss insights gained from the validation that would be hard to gain using generated data or obtain based on purely theoretical studies.

The first insights relate to what a route really is. Typically, users use some routes frequently, e.g., routes between home and work. However, even if a user drives from home to work along the same streets each day, the resulting routes turn out to all be different. This happens because a vehicle is likely to be parked in a different location at work every day, even if it is in the same parking lot. Should it happen that the vehicle is parked in the exactly same location at the end (or start), the problem remains because the positions produced by the GPS receiver are imprecise.

In Figure 16, several routes that have the same destination are shown. The circles represent the end of each route. This destination is accessible from different roads. Figure 17(a) shows how routes end if the destination is reached from the North-East. Figure 17(b) shows how routes end if the destination is accessed from the South. Because of the GPS imprecision and the varying availability of parking, the end of the route varies.

We address this problem by first modeling destination objects as circular regions of variable size. Routes then start from the same destination object if they start within the same circular region. Second, we approximate the last elements of a route if these elements belong to the same polyline and if they are inside the destination object's circular region. Thus, we consolidate the number of route elements in cases similar to that in Figure 18(a), where a vehicle drives around at its destination to find an empty parking space.

The representation of rotaries in the map can also cause problems relating to the equivalence among routes. This occurs when a rotary happens to be represented as a regular crossroads. Consider Figure 18(b) that shows a regular crossroads that represents a rotary and sequences of GPS points corresponding to two traversals. When the lower sequence is mapped to the road network, subpolylines are created that use only the horizontal road. However, when the upper sequence is mapped to the road network, the road part that extends upwards from the crossroads is also used, corresponding to the vehicle moving from the right to the crossroads, then traveling upwards a short distance, then making a u-turn and traveling down to the crossroads, and then continuing towards the left. In general, different traversals make u-turns at different locations.

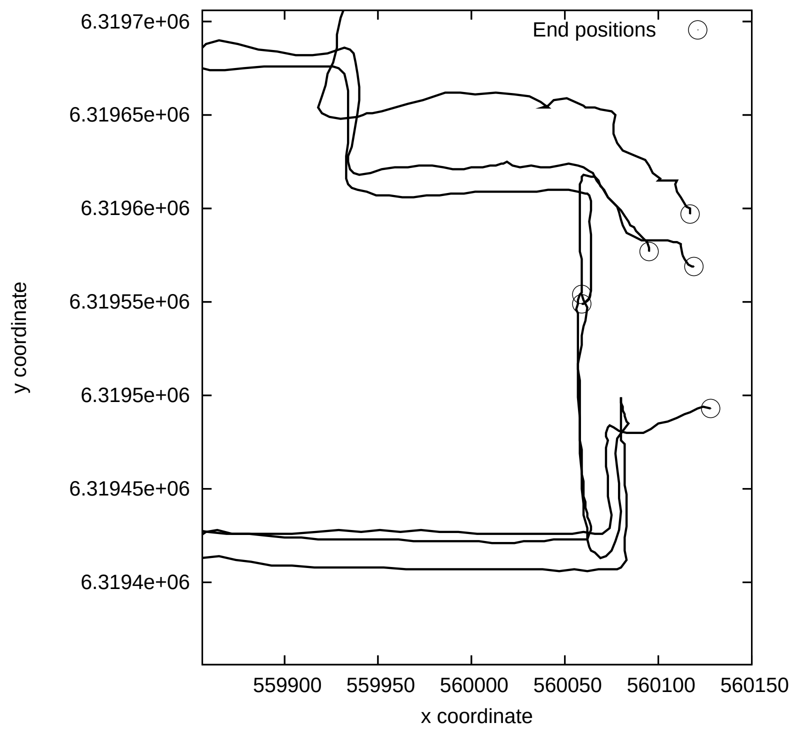
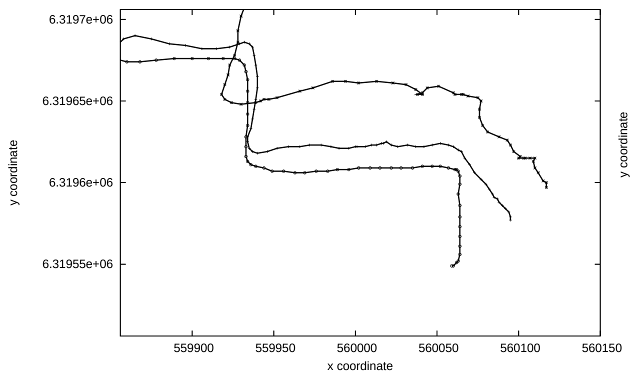
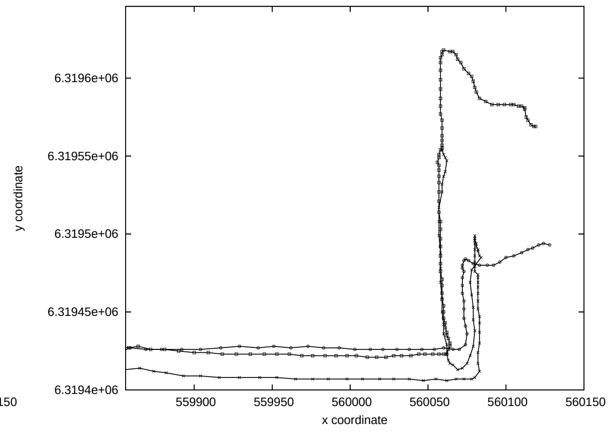


Figure 16: One End Destination Object

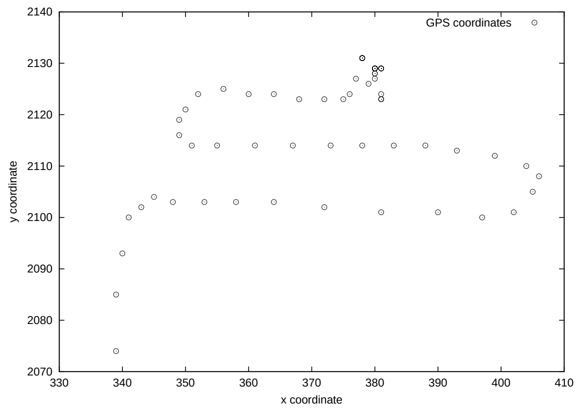


(a)

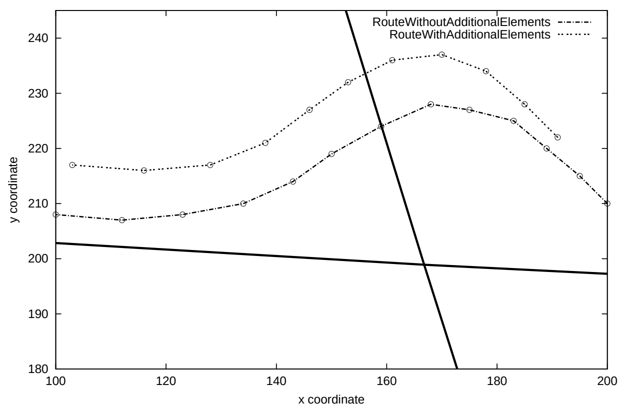


(b)

Figure 17: Different Access Types of the Destination



(a) End of a Route

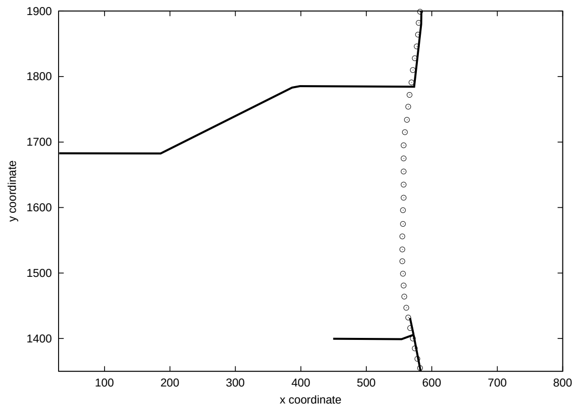


(b) Mapping at a Rotary

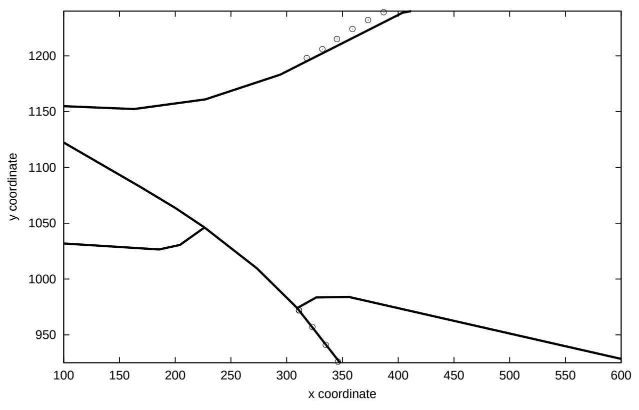
Figure 18: Special Cases

In this case, the standard imprecision value of 30 meters is too small due to the large radius of the rotary, and the algorithm will produce two different routes. One solution is to increase the imprecision value; an alternative is to obtain and use information about rotaries.

In the above discussion, the *imprecision* of map and GPS data were central sources of complications. The next insights concern in large part the *absence* of map or GPS data. Figure 19(a) illustrates a situation where a vehicle drives where the map has no road. This case occurs if the map is missing a road, e.g., the map data is outdated, or if the vehicle actually does not drive on a road (but, e.g., in a parking area or on a bike path).



(a) Gap in the Map Data



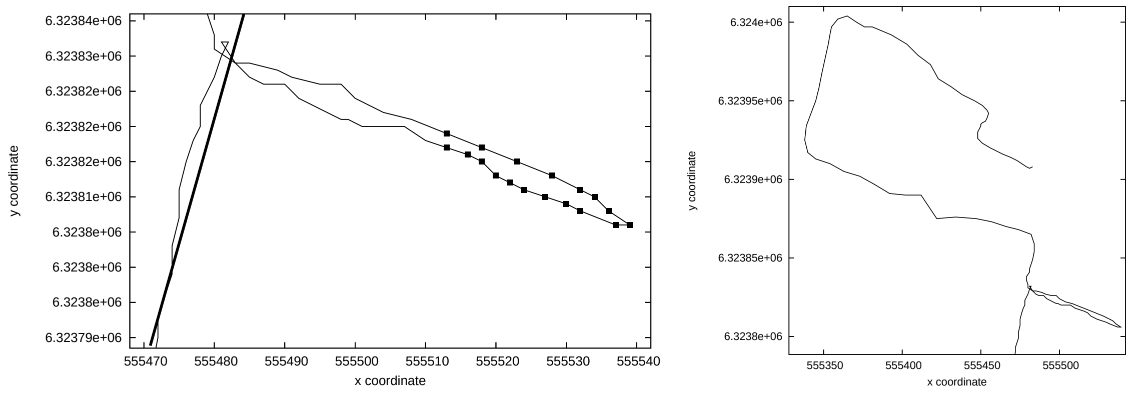
(b) Gap in the GPS Data

Figure 19: Filling of Gaps

In order to make the component resilient towards this type of situation, an algorithm $fillGap$ is used that finds the shortest path from one known point to another. If the path found is much longer than the distance traveled by the vehicle according to the GPS coordinates, the algorithm is unable to find a reasonable solution and returns an error.

Next, Figure 19(b) shows a situation with a gap in the GPS sequence. This may occur for a number of reasons. For example, the GPS coverage may be incomplete due to buildings, trees, or a tunnel. The component also handles this case by using $fillGap$. If a gap exceeds a certain distance threshold, the component returns error.

Further, function $fillGap$ is used if there are GPS positions without any polyline candidates in the middle of the route, but the gap starts and ends on the same polyline (see more details in Section A.3). Figure 20(a) represents such a situation. In the figure, the bold line represents a road in the map, and the thin line represents the movement of a car. The car moved along the road from South to North, then turned into some



(a) Unmapped Positions

(b) The Whole End of the Same Route

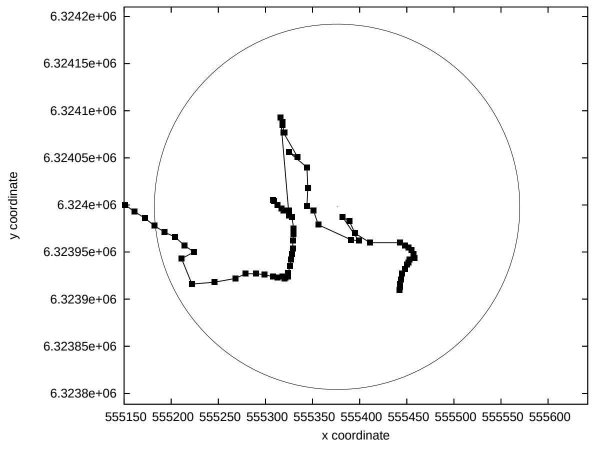
Figure 20: Unmapped GPS Positions Inside the Route

area not covered by the map used. The car drove for some time in this area, then made a u-turn, and returned to the road. Some of the positions inside the uncovered area (small black squares in the figure) were not mapped to any polyline, as they are too far from the nearest polyline. Figure 20(b) shows the whole end of the same route. This area is in the city center; thus, perhaps the driver was looking for parking, but did not find any empty parking at the first attempt, and moved on to another parking area.

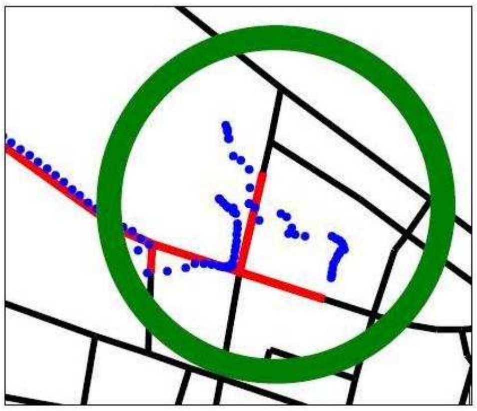
Function $backtrack$ is used for finding a good start of a route. As the initial positions along a route can be imprecise for a few minutes, the start of the route can be difficult to detect. Figure 21 shows such a situation. Figure 21(a) represents the movement of the user at the beginning of the route (black squares are GPS positions). Figure 21(b) shows how the sequence of GPS positions look on the road network.

In the experiments, visual inspection was used to determine the component's ability to accurately find routes. We found that the component works well under "normal" circumstances, but found also that the accuracy is highly dependent on the fidelity of the available representation of the road network and on the quality of the GPS positions.

The amount of the space needed to store the routes on the device was not analyzed experimentally, as it depends only on the number of routes and destinations. The routes, destinations, personal information, and usage information are stored as character strings that have a predefined schema. The space need for temporary storage of GPS positions in the NMEA [11, 20] format depends on the number of positions to be stored. The maximum NMEA sentence length is 80 characters, and each GPS position can consist of 4–6 sentences.



(a)



(b)

Figure 21: Complex Route Start

6 Related Work

We are not aware of any previous work on components that generate routes from GPS data. But our work is related to a few lines of research in mobile services, and we reuse some existing techniques.

Road network modeling is a central aspect of the paper. It is standard in industry to use linear referencing for road-network representation [1, 7, 19, 22]. Consistent with this, our data model uses linear referencing for capturing road network a as well as routes, and our data model can easily be integrated with existing linear referencing models. Using linear referencing, Hage et al. [14] describe a data model that integrates representations of transportation infrastructures and geo-referenced content. We build on this model, extending it in order to capture routes. Brakatsoulos et al. [5, 6] have recently proposed a conceptual model for trajectories in road networks. Their approach to the storage of map-matched data is slightly different, and their focus is not on routes. We also note that it is possible to model a road network as a conventional, mathematical (directed) graph (e.g., [12, 25]), in which case a route becomes a sequence of edges.

We apply several existing techniques in our setting. Shortest-path computation is used to fill gaps when we construct routes. This relates to works that consider shortest paths in graphs. Barrett et al. [3] study a generalized Dijkstra's algorithm for shortest paths in graphs on large transportation networks to do route planning.

During route construction, we map match GPS positions onto a road network. Bernstein and Kornhauser [4] explore map matching algorithms, e.g., "point-to-curve" and "curve-to-curve," that can be used to reconcile inaccurate position data with an inaccurate map. Yin and Wolfson [26] propose a weight-based off-line map matching algorithm that finds a sequence of map arcs that is similar to a trajectory given by a sequence of GPS positions. Cao and Wolfson [8] project a trajectory into the 2D (x, y) plane and then snap the resulting projection into the road network based on tolerance values. The resulting, so-called non-materialized view of the snapped trajectory is a sequence of tuples that include street names, linear referencing coordinates, and time. They use linear interpolation to infer positions in-between those recorded in tuples. When creating routes, we map match GPS positions onto the polylines that represent the geographic locations of the roads. In doing so, we use the geographic locations of the roads together with the topology of the road network, i.e., we use the connections among the polylines. Although we apply map matching in a specific data model, existing map matching techniques, such as those just mentioned, can be integrated into our work.

Our map matching involves searching for nearest neighbors. We use the allowed imprecision to control the range within which candidate polylines are to be found. This relates to the work of Roussopoulos et al. [21], in which they consider minimum and maximum distances from the query object during search. We also choose a polyline according to how the previous GPS position was map matched. The nearest neighbors for the previous positions of the moving object are considered by Song and Roussopoulos [24]. Put briefly, our use of nearest neighbor search differs from those of existing works. We search for nearest neighbors to define the movement of a user in a road network. We construct a sequence of connected polyline elements, not a set of nearest objects for every step.

Ashbrook and Starner [2] study the behaviors of people in terms of their start and end destinations, the objective being to predict future movements. They discard the GPS positions of routes that are not relevant to destinations (locations), and they build a Markov model for each location with transitions to every other location. While we, too, are interested in the start and end destinations, we also consider the specific parts of the road network that are traveled to reach one destination from another. Liao et al. [18] produce a Markov model that learns and infers daily movements of a user, again to predict future movements. A route is not specified through a deterministic sequence of edges, but through transition probabilities on the graph (road network). In contrast, we model a route as a sequence of road parts with movement direction, and a route does not itself contain spatial information. We believe that this approach is most appropriate for our purposes.

The proposed route component makes routes available to services and may be considered as a part of a more general context-aware system. For example, Harrington and Cahill [15] present a prototype implementation of a route profiling application that aims to generate information on traffic flow. They associate dynamic, contextual information—covering aspects such as weather, road-surface conditions, and road-maintenance operations—with journey (trip) information. A prototype with limited functionality is reported. A more general coverage of the notion of "context" is beyond the scope of this paper.

7 Summary and Future Work

Based on the observation that the route of a mobile user is an interesting and important context for a range of mobile services, this paper describes a system architecture along with a detailed design and a tested, relational implementation of a route component that constructs and accumulates routes and associated usage information for a mobile user based on data received from a GPS receiver that follows the user.

A route is expressed in terms of the underlying road network, as a sequence of parts of roads, or, more precisely, as a sequence of connected, linear elements, here termed subpolylines, each with a travel direction. A route connects a source and a destination object. The solution presented addresses the real-world problems that occur when attempting to derive a user's routes based on real map data and actual GPS input.

There are several possible directions in which to extend this work. We have assumed that the user controls the process of route recording. One extension is to enable the system to detect ends of routes. For example, if a user is at a particular position for some time without moving, the system may assume that the end of a route has been reached and may end the process of route recording.

Another possible extension is to enable the system to detect if a route is already recorded or to divide a long route into smaller ones when smaller parts of the route are used. Other possible extensions include the use of additional information about road networks that is available in some cases, such as allowed driving directions and turn restrictions.

8 Acknowledgments

We would like to thank the company Euman A/S for sharing their insights into road data management with us, and for constructive comments. This work was supported in part by grants 216 and 333 from the Danish National Center for IT Research. In addition to his primary affiliation, the second author is an adjunct professor at Agder University College, Norway.

References

[1] American National Standards Institute. *Geographic Information Framework—Data Content Standards For Transportation: Roads*, 2003.

[2] D. Ashbrook and T. Starner. Using GPS to learn significant locations and predict movement across multiple users. *Personal and Ubiquitous Computing*, 7(5):275–286, October 2003.

[3] C. Barrett, K. Bisset, R. Jacob, G. Konjevod, and M. Marathe. Classical and Contemporary Shortest Path Problems in Road Networks: Implementation and Experimental Analysis of the TRANSIMS Router. In *Proc. of European Symposium on Algorithms*, pp. 126–138, 2002.

[4] D. Bernstein and A. Kornhauser. An Introduction to Map Matching for Personal Navigation Assistants. New Jersey TIDE Center, 1996.

[5] S. Brakatsoulas, D. Pfoser, and N. Tryfona. Modeling, Storing, and Mining Moving Object Databases. In *Proc. of IDEAS*, pp. 68–77, 2004.

[6] S. Brakatsoulas, D. Pfoser, and N. Tryfona. Practical Data Management Techniques for Vehicle Tracking Data. In *Proc. of ICDE*, 2005. To appear.

[7] J. A. Butler and K. J. Dueker. Implementing the Enterprise GIS in Transportation Database Design. *Journal of the Urban and Regional Information Systems Association*, 13(1):17–28, 2001.

[8] H. Cao and O. Wolfson. Nonmaterialized Motion Information in Transport Networks. In *Proc. of ICDT*, pp. 173–188, 2005.

[9] A. Čivilis, C. S. Jensen, J. Nenortaitė, and S. Pakalnis. Efficient Tracking of Moving Objects with Precision Guarantees. In *Proc. of MobiQuitous*, pp. 164–173, 2004.

[10] A. Čivilis, C. S. Jensen, and S. Pakalnis. Techniques for Efficient Road-Network-Based Tracking of Moving Objects. *IEEE Transactions on Knowledge and Data Engineering*, 17(5):698–712, May 2005.

[11] CommLinx Solutions Pty Ltd. *Common NMEA Sentence Types*, 2002. http://www.commlinx.com.au/.

[12] Z. Ding and R. H. Güting. Modeling Temporally Variable Transportation Networks. In *Proc. of DASFAA*, pp. 154–168, 2004.

[13] Federal Communications Commission. *Enhanced 911 - Wireless Services*. http://www.fcc.gov/911/enhanced/.

[14] C. Hage, C. S. Jensen, T. B. Pedersen, L. Speicys, and I. Timko. Integrated Data Management for Mobile Services in the Real World. In *Proc. of VLDB*, pp. 1019–1030, 2003.

[15] A. Harrington and V. Cahill. Route Profiling - Putting Context To Work. In *Proc. of SAC*, pp. 1567–1573, 2004.

[16] X. Huang and C. S. Jensen. In-Route Skyline Querying for Location-Based Services. In *Proc. of the Fourth International Workshop on Web and Wireless Geographic Information Systems*, pp. 223–238, 2004. (Also pp. 120–135 in the Lecture Notes in Computer Science, Volume 3428, 2005.)

[17] C. S. Jensen, H. Lahrmann, S. Pakalnis, and J. Runge. The Infati Data. TIMECENTER Technical Report TR-79, July 2004, 10 pages. Also CoRR cs.DB/0410001, http://arxiv.org/abs/cs.DB/0410001.

[18] L. Liao, D. Fox, and H. A. Kautz. Learning and Inferring Transportation Routines. In *Proc. of AAAI*, pp. 348–353, 2004.

[19] C. Murray. *Oracle Spatial User Guide and Reference, Release 9.2*. Oracle Corporation, 2002.

[20] NMEA. *NMEA 0183 Standard*, 2002. http://www.nmea.org/pub/0183/.

[21] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proc. of ACM SIGMOD*, pp. 71–79, 1995.

[22] P. Scarponcini. Generalized Model for Linear Referencing. In *Proc. of ACM-GIS*, pp. 53–59, 1999.

[23] J. Schiller and A. Voisard. *Location-Based Services*. Morgan Kaufmann Publishers, 2004.

[24] Z. Song and N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *Proc. of SSTD*, pp. 79–96, 2001.

[25] M. Vazirgiannis and O. Wolfson. A Spatiotemporal Model and Language for Moving Objects on Road Networks. In *Proc. of SSTD*, pp. 20–35, 2001.

[26] H. Yin and O. Wolfson. A Weight-based Map Matching Algorithm in Moving Objects Databases. *Proc. of SSDBM*, pp. 437–438, 2004.

A Algorithms

The following appendices give the details about the algorithms presented, but not fully described, in Section 4. Section A.1 describes the approximation of the ends of routes. Section A.2 covers the detection of the starts of route. Section A.3 covers the filling of information gaps. Section A.4 describes how to find a path between two projections on the road.

A.1 Approximation of the End of the Route

As already mentioned in Section 4, function $proceedEnd$ (see Algorithm A.1) constructs the end of a route and approximates the last subpolylines to one subpolyline if they are on the same polyline.

The function takes as arguments the state of the main algorithm at the end of the analysis of the GPS stream and the distance from the start of the polyline to the projection of the last correctly mapped GPS point, i.e., $cState$ and $cDst$, respectively. The function returns a sequence RE that contains all constructed route elements.

Function $proceedEnd$ analyses the sequence of constructed route elements (line 1). In the state of the algorithm, RE contains all constructed route elements, except the last one. The state stores the information about the last route element, i.e., pPl is the polyline and $l^{\vdash}$ is the start distance of the subpolyline for the route element. The second input parameter $cDst$ provides the end distance of the subpolyline for the last route element. The function looks for the earliest distance measure on the same last polyline pPl, so that it is possible to approximate the end of the route to one subpolyline. The variables S and E are used to represent the earliest (start) and latest (end) positions (line 2). A visual explanation is given in Figure 14 (see Section 4). The latest position E on the polyline is the distance of the last correctly mapped GPS point, $cDst$. This value does not change throughout the algorithm. The earliest distance value should be found. At the beginning of the analysis, S is equal to $l^{\vdash}$ that is the start of the subpolyline for the last route element (it is not added to the sequence RE). Then the function checks route elements to approximate the end of the route. The variable $checked$ (line 3) is used to indicate whether the elements are already checked (true) or not (false).

The route elements are analyzed starting from the last to the first (line 5). If the last subpolyline in the sequence RE belongs to the same polyline pPl (line 6), the subpolyline is checked to determine whether the earliest position S can be approximated to the earliest position on the subpolyline: if the movement direction is -1, the earliest position is the end distance $l^{\dashv}$; otherwise, it is the start distance $l^{\vdash}$. The part of the subpolyline that disappears when approximated should be inside the circular area of the end destination. For example, in Figure 14 when subpolyline $(x3, x4)$ is analyzed, S is approximated to $x4$, but $(x3, E)$ should be inside the circular area. Then the subpolyline is removed from the sequence of route elements (lines 7–10).

If the disappearing part of the subpolyline does not belong to the circular destination area, the checking process should stop, i.e., $checked$ becomes equal to true (lines 11–12). If the currently last subpolyline does not belong to the last polyline pPl, the check process should also be stopped (lines 14–15).

When the last subpolylines are checked, the earliest and latest positions on the last polyline are known. The last route element is constructed (line 18) and appended to the sequence of route elements (line 19).

A.2 Detection of the Start of the Route

In most cases during route recording, the first GPS positions cannot be mapped to polylines correctly at once. Function $getStartValues$ (described in Section 4) analyses the stream of GPS points. The function collects all the candidate polylines for each GPS point until a point with one candidate polyline is found.

Algorithm A.1 Approximation of the End of the Route (function *proceedEnd*)

```
Require: INPUT:  $cState = ((pPl, pDst, pDir), l^{\vdash}, RE)$, $cDst \in \mathbb{R}$
         OUTPUT: $RE$
 1: let $RE = ((spl_1, dir_1), \ldots, (spl_n, dir_n))$, where $spl_i = (pl_i, l_i^{\vdash}, l_i^{\dashv})$
 2: $E \leftarrow cDst; S \leftarrow l^{\vdash}$
 3: $checked \leftarrow$ false
 4: while $RE$ is not empty $\wedge$ not $checked$ do
 5:   $((pl, l^{\vdash}, l^{\dashv}), dir) \leftarrow$ getLast($RE$)
 6:   if $pl = pPl$ then
 7:     if $dir = -1 \wedge ((E \leq l^{\vdash} < l^{\dashv})$
                $\vee(l^{\vdash} < E < l^{\dashv} \wedge insideCircle(l^{\vdash}, E))$
                $\vee(l^{\vdash} < l^{\dashv} \leq E \wedge insideCircle(l^{\vdash}, l^{\dashv})))$ then
 8:       $S \leftarrow l^{\dashv}; RE \leftarrow$ removeLast($RE$)
 9:     else if $dir = 1 \wedge ((l^{\vdash} < l^{\dashv} \leq E)$
                $\vee(l^{\vdash} < E < l^{\dashv} \wedge insideCircle(E, l^{\dashv}))$
                $\vee(E \leq l^{\vdash} < l^{\dashv} \wedge insideCircle(l^{\vdash}, l^{\dashv})))$ then
10:       $S \leftarrow l^{\vdash}; RE \leftarrow$ removeLast($RE$)
11:     else
12:       $checked \leftarrow$ true
13:     end if
14:   else
15:     $checked \leftarrow$ true
16:   end if
17: end while
18: $(spl, dir) \leftarrow formSubPoly(pPl, S, E, 0)$
19: $RE \leftarrow$append($RE, (spl, dir)$)
20: return $RE$
```

Function $backtrack$ (see Algorithm A.2) determines the start of the route, looking back through the candidate polylines for each GPS point, and it constructs route elements for the unmapped GPS points if possible. Function $backtrack$ takes a sequence of candidate sets $cList$ together with a list of unmapped GPS points $undG$ as its arguments. The function returns the current state of the algorithm $cState$.

When the function receives the list of candidates $cList$, the last set in this sequence (line 1) contains only one element. The last set has only one candidate polyline because it is the set for the first correctly mapped GPS point. The candidates are analyzed from the last set to the first. Thus, the sequence of route elements is constructed in reverse order; this is changed at the end of the algorithm.

The function uses temporary variables to store values. The set $Cand$ contains candidate polylines with distances from the start of the polyline to the projection for a particular GPS point. Variable pPl stores the polyline of the route element that is being constructed. Next, $l^{\vdash}$ and $l^{\dashv}$ are the start and end of the current subpolyline (route element) on the polyline, and $pDir$ is the direction on the polyline in the current route element. Variable $checked$ indicates whether the analysis of the candidates should be finished. If $checked$ is false, the analysis should continue; otherwise, it should be stopped. Throughout the algorithm, the elements of the sequence $undG$ are removed. This sequence is used for determining the user object. The currently last element represents the point for which candidates are currently analyzed.

For each GPS point, the set of candidate polylines is taken from the sequence $cList$ (line 4). If there exists a candidate (pl, l) such that pl equals pPl (line 5) then the movement direction is checked (line 6).

Algorithm A.2 Determination of the Start of the Route (function $backtrack$)

```
Require: INPUT:  $cList = (Cand_1, \dots, Cand_n)$, where $Cand_i = \{(pl,l)|(pl,l) \in PL \times \mathbb{R}\}$,
                 $undG = (g_1, \dots, g_n)$, where $g_i \in \mathbb{R}^2$
         OUTPUT: $cState = ((pPl, pDst, pDir), l^{\vdash}, RE)$
 1: $Cand = \{(pPl, l^{\vdash})\} \leftarrow$ getLast($cList$); $cList \leftarrow$ removeLast($cList$)
 2: $l^{\dashv} \leftarrow l^{\vdash}; pDir \leftarrow 0; RE \leftarrow$ nil; $checked \leftarrow$ false; ($undG \leftarrow$ removeLast($undG$))
 3: while $cList$ is not empty $\wedge$ not $checked$ do
 4:   $Cand \leftarrow$getLast($cList$); $cList \leftarrow$ removeLast($cList$)
 5:   if $\exists(pl,l) \in Cand$ such that $pl = pPl$ then
 6:     $dir \leftarrow defineDirection(l^{\dashv}, l)$; ($undG \leftarrow$ removeLast($undG$))
 7:     if $pDir = 0$ then
 8:       $pDir \leftarrow dir; l^{\dashv} \leftarrow l$
 9:     else if $pDir = dir$ then
10:       $l^{\dashv} \leftarrow l$
11:     else
12:       $((pPl, l^{\dashv}, pDir), l^{\vdash}, RE) \leftarrow newSubSamePoly(((pPl, l^{\dashv}, pDir), l^{\vdash}, RE), l)$
13:     end if
14:   else
15:     $polys \leftarrow \{(pl,l)|pl \in PL, l \in \mathbb{R}, \exists c \in C((pPl, l_1) \in c \wedge (pl,l) \in c \wedge pPl \neq pl \wedge$
              $\nexists c_1 \in C(c \neq c_1 \wedge (pPl, l_2) \in c_1 \wedge |l_1 - l^{\dashv}| > |l_2 - l^{\dashv}| \wedge$
              $((pDir = 1 \wedge l_1 - l^{\dashv} \leq D \wedge l_2 - l^{\dashv} \leq D) \vee$
              $(pDir = -1 \wedge l_1 - l^{\dashv} \geq -D \wedge l_2 - l^{\dashv} \geq -D))))\}$
16:     if $\exists(pl_k, l'_k) \in polys \wedge \exists(pl_k, l_k) \in Cand \wedge \nexists(pl_h, l'_h) \in polys \wedge \exists(pl_h, l_h) \in Cand$ then
17:       $((pPl, l^{\dashv}, pDir), l^{\vdash}, RE) \leftarrow newSubOtherPoly(((pPl, l^{\dashv}, pDir), l^{\vdash}, RE), pl_k)$
18:       ($undG \leftarrow$ removeLast($undG$))
19:     else if $\nexists(pl_k, l_k) \in Cand((pl_k, l'_k) \in polys)$ then
20:       $checked \leftarrow$ true
21:     end if
22:   end if
23: end while
24: $(spl, dir) \leftarrow formSubPoly(pPl, l^{\vdash}, l^{\dashv}, pDir)$; $RE \leftarrow$ append $(RE, (spl, dir))$
25: $cState \leftarrow updateState(RE)$
26: return $cState$
```

1. If the movement direction was unknown up till now, the current direction is stored (lines 7–8).
2. If the direction is the same as until the previous point, the current point is stored (lines 9–10) for the end distance of the subpolyline.
3. If the direction is the opposite of the previous direction, a new route element is constructed (lines 11–12).

If the same polyline is not among the candidates for the particular GPS point (line 14), all the polylines that intersect with the current polyline at the nearest connection are found (line 15). Temporary variable $polys$ stores these intersecting polylines with their distances from the start of the polyline to the connection. The connection is ahead no further than distance D or behind on the polyline pPl.

If there is only one candidate polyline in $Cand$ that is also in $polys$ (line 16), this polyline is chosen and a new subpolyline is constructed (line 17). If there is no candidate polyline in $Cand$ that is also in $polys$, the value of $checked$ becomes $\mathtt{true}$ (lines 19–20), and the analysis of the candidates is stopped. If there is more than one candidate that is also in $polys$, the algorithm does not select any of them, and continues the analysis of the candidates.

When the analysis of the candidates is finished, the last route element is constructed and added to the sequence of route elements (line 24). As mentioned earlier, the function analyses the unmapped GPS points from the last one to the first one and constructs route elements in reverse order and with opposite directions.

Thus, function $updateState$ (see Algorithm A.3), which reverses a sequence of route elements, is applied. The function takes a sequence of route elements RE^* as its input and returns the current state $cState$ of the algorithm. Initially, the function $updateState$ takes the first element from the sequence RE^* (line 1).

Algorithm A.3 Update the Current State of the Algorithm (function $updateState$)

```
Require:  INPUT:   $RE^* = ((spl_1, dir_1), \ldots, (spl_n, dir_n))$
          OUTPUT:  $cState = ((pPl, pDst, pDir), l^{\vdash}, RE)$
 1: $((pPl, l_s, l_e), dir) \leftarrow$ head $(RE^*), RE^* \leftarrow$ tail $(RE^*)$
 2: if $dir = 1$ then
 3:     $pDst \leftarrow l_s; l^{\vdash} \leftarrow l_e; pDir \leftarrow -1$
 4: else
 5:     $pDst \leftarrow l_e; l^{\vdash} \leftarrow l_s; pDir \leftarrow 1$
 6: end if
 7: $RE \leftarrow$ nil
 8: while $RE^*$ is not empty do
 9:     $(spl, dir) \leftarrow$ getLast $(RE^*); RE^* \leftarrow$ removeLast $(RE^*)$
10:     $RE \leftarrow$ append $(RE, (spl, (-1)\ dir))$
11: end while
12: return $cState$
```

This element should be the last element in the sequence of route elements. But it cannot be constructed because some further GPS points, yet to be analyzed, may belong to the same subpolyline. Thus, this element represents the current state of the algorithm (lines 2–6). All other elements are constructed with the opposite direction and in the opposite order (lines 8–11).

A.3 Filling of Gaps

For various reasons, gaps may occur in a GPS stream or in the digital representation of a road network. The route finding algorithm (see Algorithm 4.12 in Section 4) also needs gap filling when the current GPS point cannot be mapped correctly based on information about previous, correctly mapped GPS points. This happens when the current point cannot be mapped to the polyline of the previous GPS point and either more than one or no candidate polylines exist that intersect with the polyline of the previous point.

Function $FillGap$ (see Algorithm A.4) takes the current state $cState$ of the main algorithm $findRoute$ (see Algorithm 4.12), the GPS point g that could not be mapped correctly using algorithm $polyId$ (see Algorithm 4.3), and the GPS stream G as arguments. The function returns a new state for the algorithm and the GPS stream without the points that are used in the gap filling.

At the beginning of the gap filling (lines 1–6), the algorithm searches for a GPS point that can be mapped correctly to the polyline, i.e., has only one candidate polyline. A temporary variable $Cand$ is used for two-tuples of candidate polylines together with distances from the start of the polyline to the projection on the polyline. $Cand$ is calculated by function $polyCand$ (see Algorithm 4.2). A temporary variable $undG$ is used to collect points that have either more than one or no candidate polyline.

Algorithm A.4 Filling the GPS Gap (function $FillGap$)

```
Require:  INPUT:   $cState = ((pPl, pDst, pDir), l^{\vdash}, RE), g \in \mathbb{R}^2, G = (g_1, \ldots, g_n)$ where $g_i \in \mathbb{R}^2$
          OUTPUT: $cState, G$
 1: $Cand \leftarrow polyCand(g); undG \leftarrow$ nil
 2: while $((\forall(pl, l) \in Cand\ \exists(pl', l') \in Cand(pl \neq pl')) \vee Cand$ is empty $) \wedge G$ is not empty do
 3:    $undG \leftarrow$ append$(undG, g)$
 4:    $g \leftarrow$ head$(G); G \leftarrow$ tail$(G)$
 5:    $Cand \leftarrow polyCand(g)$
 6: end while
 7: if $\forall(pl, l) \in Cand\ \nexists(pl', l') \in Cand(pl \neq pl')$ then
 8:    if $pl = pPl$ then
 9:       $dir \leftarrow defineDirection(pDst, l, 0)$
10:       if $dir = pDir$ then
11:          $pDst \leftarrow l$
12:       else
13:          $cState \leftarrow newSubSamePoly(cState, l)$
14:       end if
15:    else
16:       $gpsLength \leftarrow findGpsLength(undG)$
17:       $Path \leftarrow findPath(pPl, pDst, pl, l, gpsLength)$
18:       $(pPl, pC) \leftarrow$ head$(Path)$; $Path \leftarrow$ tail$(Path)$
19:       $dir \leftarrow defineDirection(pDst, l^{\star}, 0)$, where $(pl^{\star}, l^{\star}) \in pC, pl^{\star} = pPl$
20:       if $dir = pDir \vee pDir = 0$ then
21:          $(spl, dir) \leftarrow formSubPoly(pPl, l^{\vdash}, l^{\star}, dir); RE \leftarrow$ append$(RE, (spl, dir))$
22:       else
23:          $(spl, dir) \leftarrow formSubPoly(pPl, l^{\vdash}, pDst, pDir); RE \leftarrow$ append$(RE, (spl, dir))$
24:          $(spl, dir) \leftarrow formSubPoly(pPl, pDst, l^{\star}, dir); RE \leftarrow$ append$(RE, (spl, dir))$
25:       end if
26:       while $Path$ is not empty do
27:          $(cPl, cC) \leftarrow$ head$(Path)$; $Path \leftarrow$ tail$(Path)$
28:          $(spl, dir) \leftarrow formSubPoly(cPl, l_s, l_e, 0), (cPl, l_s) \in pC, (cPl, l_e) \in cC$
29:          $RE \leftarrow$ append$(RE, (spl, dir))$
30:          $pC \leftarrow cC$
31:       end while
32:       $dir \leftarrow defineDirection(l^{\star}, l, 0)$, where $(pl^{\star}, l^{\star}) \in cC, pl^{\star} = pl$
33:       $cState = ((pPl, pDst, pDir); l^{\vdash}, RE) \leftarrow ((pl, l, dir), l^{\star}, RE)$
34:    end if
35: else
36:    EXIT
37: end if
38: return $cState$
```

First of all, the candidates for the input GPS point g are found, and a set $undG$ is empty (line 1). Then the GPS points are analyzed until a point with one candidate polyline is found, or until the GPS stream becomes empty (lines 2–6). Points with no unique candidate polyline are added to $undG$ (line 3). If the GPS stream becomes empty, but no point with a unique candidate polyline is found, the algorithm exits the whole route finding algorithm (lines 35–36).

When a GPS point with one candidate polyline pl is found (line 7), candidate (pl, l) is checked. If this polyline is the same as the polyline for the previous, correctly mapped point, i.e., it is the same polyline as pPl in $cState$ (line 8) then the movement direction from the previous position $pDst$ to l is also calculated (line 9). If the direction is the same, only the position $pDst$ in the current state is modified (lines 10–11). If the direction is the opposite, a new route element is constructed based on state $cState$, and the construction of a new subpolyline is started (lines 12–13).

If the candidate polyline pl is different from polyline pPl of the previous, correctly mapped position, the path from the previous projection to the current projection is found. Before that, length $gpsLength$ of the stream of the undefined GPS points in $undG$ is calculated. Function $findGpsLength$ (see Algorithm A.5) takes a sequence of GPS points G and returns the length of the polyline constructed from stream G. While calculating the distance between two neighboring points, the algorithm checks that this distance is no greater than $D^\star$, which is the maximum allowed distance between points. If the distance is greater than this, the algorithm exits the route finding algorithm. The length of the GPS stream is used to decide whether the

Algorithm A.5 Finding GPS Length (function $findGpsLength$)

```
Require: INPUT:  $G = (g_1, \dots, g_n), n > 2$ where $g_i \in \mathbb{R}^2$
         OUTPUT: $length \in \mathbb{R}$
 1: $length \leftarrow 0$
 2: $g_1 \leftarrow$ head$(G); G \leftarrow$ tail$(G)$
 3: while $G$ is not empty do
 4:     $g_2 \leftarrow$ head$(G); G \leftarrow$ tail$(G)$
 5:     if $|g_1 g_2| > D^\star$ then
 6:         EXIT
 7:     end if
 8:     $length \leftarrow length + |g_1 g_2|$
 9:     $g_1 \leftarrow g_2$
10: end while
11: return $length$
```

path found can be consistent with the GPS stream in terms of length (more about paths in Section A.4). The information gap starts at distance $pDst$ on polyline pPl and ends at distance l on polyline pl. Function $findPath$ finds the path that fills this gap (line 17).

The returned path $Path$ represents the sequence of nodes to pass through to move from the start of the gap to the end of the gap: $Path = (node_1, \dots, node_m)$, where $node_i = (pl_i, c_i) \in PL \times C, 1 \leq i < m$ represents the part of the path from connection c_{i-1} to connection c_i on polyline pl_i.

The information about the start and end of the gap is already stored in variables $pPl, pDst, pl$, and l. Then the elements in $Path$ represent the path from polyline pPl to polyline pl. The current state $cState$ captures the previously constructed subpolyline on polyline pPl: from distance $l^\vdash$ to distance $pDst$. The first element in $Path$ indicates whether this subpolyline should be extended to the distance at connection c_1, or whether there should be two subpolylines on pPl. The first element of the path is considered as (pPl, pC), and is removed from the sequence of path nodes (line 18). The movement direction from $pDst$ to connection pC on pPl is defined in line 19, where $l^\star$ is the distance from the start of polyline pPl to connection pC along the polyline.

If the movement direction is the same as it was until position $pDst$ (or the previous direction was unknown), there is only one subpolyline from $l^{\vdash}$ to $l^{\star}$ on pPl (lines 20–21), and one route element is constructed. If the movement direction is the opposite, there should be two subpolylines on pPl, one from $l^{\vdash}$ to $pDst$, and another from $pDst$ to $l^{\star}$ (at connection pC) (lines 22–24).

Now the remaining nodes of the path are analyzed (lines 26–31). The currently first node in the path is taken and removed from the path (line 27). The node consists of polyline cPl and connection cC (line 27). The subpolyline for a new route element is on polyline cPl from connection pC to connection cC (line 28). This route element is appended to RE. The current connection then becomes the previous connection for the next node (line 30).

When all the nodes of the path have been analyzed, the gap is filled all the way to the polyline pl on which the end of the gap is. The movement direction dir from the final connection (distance $l^{\star}$) to the end of the gap l is defined (line 32). The current state to return is the polyline pl, the start for the new subpolyline $l^{\star}$, the current end for the new subpolyline l, the direction dir, and all the route elements RE (line 33).

A.4 Calculating the Shortest Path Between Two Projections

As part of filling information gaps, it is necessary to find a path between two projections located on different polylines.

Function $\mathit{findPath}$ takes five arguments, namely two polylines, two distances, and the maximum allowed length for the path. Polyline pPl with distance $pDst$ is the start of the gap. Polyline cPl with distance $cDst$ is the end of the gap. Argument $gpsLength$ is the maximum allowed length for the path. The function returns a path $Path_0$ that is the shortest path from position $pDst$ on polyline pPl to position $cDst$ on polyline cPl.

Throughout the algorithm, temporary paths $Path$ and $Path_0$ represent a sequence of nodes to pass through from the required position: $Path = (node_0, \ldots, node_m)$, where $node_0$ is the position where all analyzed paths start, i.e., $node_0 = (pPl, pDst)$, and $node_i = (pl_i, c_i) \in PL \times C, 1 \leq i \leq m$ is a polyline and one of its connections. The last element of the found path $Path_0$ contains an artificial node $node_m = (cPl, \{(cPl, cDst)\})$ that represents the end of the information gap. The first and the last elements are removed from the path before returning the path (line 34). The polyline of each node is also contained in the previous connection, i.e., $\forall node_i = (pl_i, c_i), 1 < i \leq m, \exists (pl_i, l_i) \in c_i, (pl_i, l'_i) \in c_{i-1}$.

For example, assuming the path $((pl_1, c_1), (pl_2, c_2), (pl_3, c_3))$ is returned by function $\mathit{findPath}$, the necessary subpolylines are constructed using nodes to fill the gap: from $pDst$ to c_1 along polyline pPl ($pPl = pl_1$), from c_1 to c_2 along pl_2, from c_2 to c_3 along pl_3 (c_3 is a connection with polyline cPl), and from c_3 to $cDst$ along cPl.

The algorithm also uses other temporary variables. A set $usedCon$ contains all the connections c_i that are already analyzed while constructing paths. A set $conns$ contains connections of a particular polyline. A sequence $Paths$ contains paths in ascending order of their lengths. Path length is defined as follows: $|Path| = \sum_{i=2}^{m} |l_i - l'_i| + |pDst - l_1|$, where $(pPl, l_1) \in c_1, (cPl, Path = (node_0, \ldots, node_m), node_i = (pl_i, c_i), \forall i, 1 < i \leq m$ $(\exists (pl_i, l_i) \in c_i$ $((pl_i, l'_i) \in c_{i-1}))$. Thus, $Paths = (Path_1, \ldots, Path_n)$, where $|Path_i| \leq |Path_{i+1}|, 1 \leq i < n$.

At the beginning of function $\mathit{findPath}$, the shortest available path $Path_0$ is constructed from the first node that should be in the path, i.e., $node_0$ that represents the start of the gap (line 2). Then all the connections on polyline pPl are found (line 3). The connections represent all possible ways to reach other polylines from the start of the gap. For each of the connections, a path is constructed (lines 4–7). A node with polyline pPl and a particular connection c_i is created (line 5) and added to $Path_0$ by function $modifyPaths$ (line 6). Function $modifyPaths$ (see Algorithm A.7) takes the sequence of paths $Paths$, the currently analyzed path $Path_0$, a new node $node$, and the maximum allowed path length $gpsLength$ as arguments. A new path is constructed by adding a new node to the currently analyzed path (line 2). If the length of the path does not

Algorithm A.6 Finding the Shortest Path (function *findPath*)

```
Require: INPUT:  $pPl, pDst, cPl, cDst, gpsLength$, where $pPl, cPl \in PL, pPl \neq cPl,$
                                                  $pDst, cDst, gpsLength \in \mathbb{R}$
         OUTPUT: $Path_0 = (node_1, \dots, node_m)$, where $node_i = (pl_i, c_i) \in PL \times C, 1 \leq i \leq m$
 1: $usedCon \leftarrow \emptyset; Path \leftarrow \text{nil}; Paths \leftarrow \text{nil}$
 2: $node_0 \leftarrow (pPl, pDst); Path_0 \leftarrow$ append$(node_0)$
 3: $conns \leftarrow \{c_1, \dots, c_n\}$, where $c_i \in C \wedge (\forall c_i \in conns\ \exists (pl, l) \in c_i(pl = pPl)) \wedge$
                                  $(\nexists c' \in C((pPl, l') \in c' \wedge c' \notin conns))$
 4: for all $c_i \in conns$ do
 5:   $node \leftarrow (pPl, c_i)$,
 6:   $Paths \leftarrow modifyPaths(Paths, Path_0, node, gpsLength)$
 7: end for
 8: $found \leftarrow \text{false}$
 9: while not $found \wedge Paths$ not empty do
10:   $Path_0 \leftarrow$ head$(Paths)$; $Paths \leftarrow$ tail$(Paths)$
11:   $(pl, c) \leftarrow$ getLast$(Path_0)$
12:   if $(cPl, cDst) \in c$ then
13:     $found \leftarrow \text{true}$
14:   else
15:     if $c \notin usedCon$ then
16:       if $\exists (pl', l') \in c(pl' = cPl)$ then
17:         $node \leftarrow (cPl, \{(cPl, cDst)\})$
18:         $Paths \leftarrow modifyPaths(Paths, Path_0, node, gpsLength)$
19:       end if
20:       $usedCon \leftarrow$ add$(usedCon, c)$
21:       for all $(pl', l') \in c(pl' \neq pl)$ do
22:         $conns \leftarrow \{c_1, \dots, c_n\}$, where $c_i \in C \wedge (\forall c_i \in conns\ \exists (pl^\star, l^\star) \in c_i(pl^\star = pl'))$
                          $\wedge (\nexists c_k \in C((pl', l'') \in c_k \wedge c_k \notin conns \wedge c_k \notin usedCon))$
23:         for all $c_i \in conns$ do
24:           $node \leftarrow (pl', c_i)$
25:           $Paths \leftarrow modifyPaths(Paths, Path_0, node, gpsLength)$
26:         end for
27:       end for
28:     end if
29:   end if
30: end while
31: if not $found$ then
32:   EXIT
33: end if
34: $Path_0 \leftarrow$ tail$(Path_0)$; $Path_0 \leftarrow$ removeLast$(Path_0)$
35: return $Path_0$
```

exceed the maximum allowed length, the path is inserted into the sequence of all paths (lines 3–5) based on the rules discussed in the previous paragraph. Thus, non-competitive paths are pruned. A constant F is introduced to allow a variation of path lengths. The function returns the sequence of paths.

Then function *findPath* searches for the shortest path to the end of the gap represented by $(cPl, \{(cPl, cDst)\})$. This is a greedy algorithm that chooses the currently shortest path to search for the end of the gap. Tempo-

Algorithm A.7 Modification of Paths (function $modifyPaths$)

```
Require: INPUT:  $Paths, Path_0, node, gpsLength$
         OUTPUT: $Paths$
1: $Path \leftarrow Path_0$
2: $Path \leftarrow$ append$(Path, node)$
3: if $|Path| \leq F\ gpsLength$ then
4:     $Paths \leftarrow$ insert$(Paths, Path)$
5: end if
6: return $Paths$
```

rary variable $found$ represents the state for the search. At the beginning it is false (line 8) as the path has yet to be found. Variable $found$ becomes true when the node of the gap end is found at the end of the currently analyzed path. During the search, the first (current shortest) path is taken and removed from the sequence of paths (line 10). The last node of the path is analyzed (line 11). If the connection in the last node is the required end of the gap, $found$ becomes true and the search stops (lines 12–13).

If the current connection is not the end of the gap, it should be checked that it is not in the set of analyzed connections (line 15). If it is not in this set, the possible ways from this connection to other polylines are detected.

If the connection is not the end of the gap, but relates to the required polyline cPl, the current path is duplicated. The end node is created and added to the path (lines 16–18). The path is inserted into the sequence of paths based on the length (line 18) by function $modifyPaths$.

The current connection is added to the set of used connections (line 20). Connections are detected on all polylines pl' that intersect at the current connection, except for the current polyline pl. A connection is not considered if it is already in $usedCon$. New paths are created by adding a new connection to the duplicated, currently analyzed path (lines 23–26).