Enabling Routes as Context in Mobile Services

Agne Brilingaite Christian S. Jensen Nora Zokaite
Department of Computer Science, Aalborg University, Denmark
{agne,csj,nora}@cs.aau.dk

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. It is therefore 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 are important aspects of the context for a range of services. This paper presents key concepts underlying a software component that discovers the routes of a user along with their usage patterns and that makes the accumulated routes available to services. Experiences from using the component with real GPS logs are reported.

Categories and Subject Descriptors

H.2.1 [**Database Management**]: Logical Design—*Data Models*; H.2.8 [**Database Management**]: Database Applications—*Spatial Databases and GIS*

General Terms

Algorithms, Design, Experimentation, Management

Keywords

Context awareness, location-based services, road networks, destinations, routes, map matching

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 quite rapidly. With technologies such as GPRS, 2.5G (EDGE), and 3G (CDMA, UMTS), which are packet based, the user can be always on at no extra cost; bandwidth is increasing; and regulatory developments, such as the US E911 Mandate and similar developments in Asia and Europe, contribute to the spread of positioning technologies. An infrastructure is thus emerging that supports a range of location-enabled on-line mobile services [14].

However, mobile services are delivered to devices that are typically without 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 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 has it that mobile users typically travel towards a destination (rather than moving around, aimlessly) and that a user 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 "near" points of interest. More specifically, a service may suggest restaurants to the user that are near to the user's route, rather than merely to the user's current location.

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.

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. An experimental validation is reported in Section 5. Finally, Sec-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
GIS'04, November 12–13, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-979-9/04/0011 ...$5.00.

tion 6 covers related work, and Section 7 summarizes and offers directions for future work.

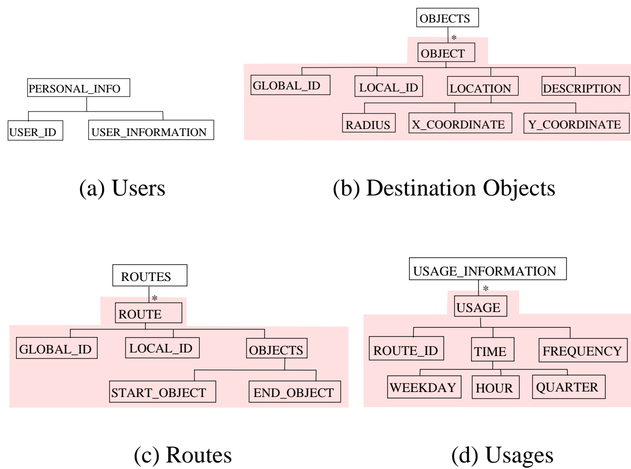
2. SYSTEM ARCHITECTURE

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 [6, 11], 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.



(a) Users (b) Destination Objects

(c) Routes (d) Usages

Figure 1: Client-Side Data

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

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.

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 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

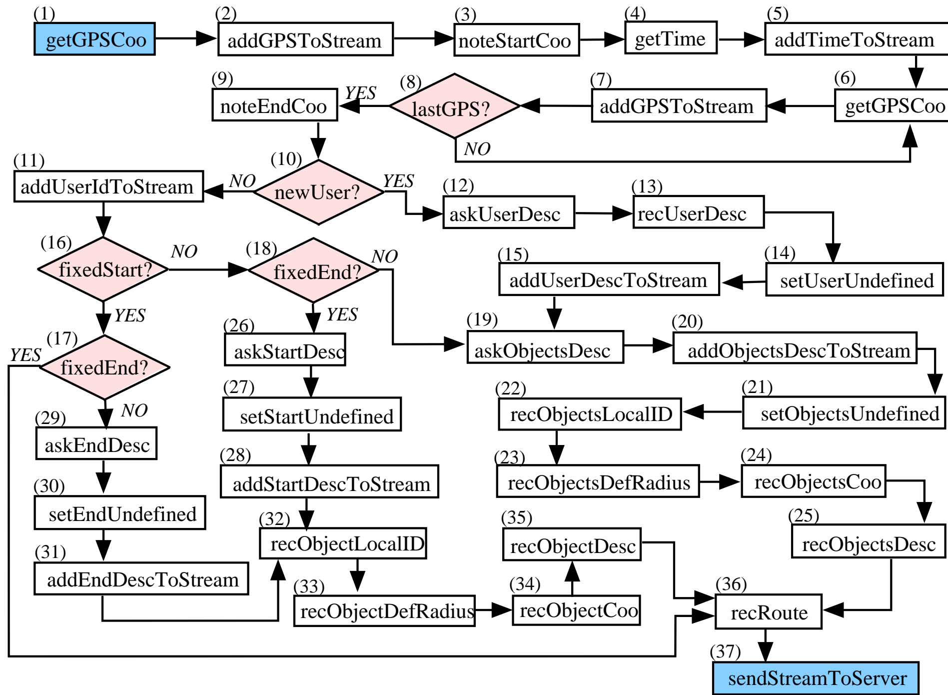


Figure 2: Client-Side Route Recording

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 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. □

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.

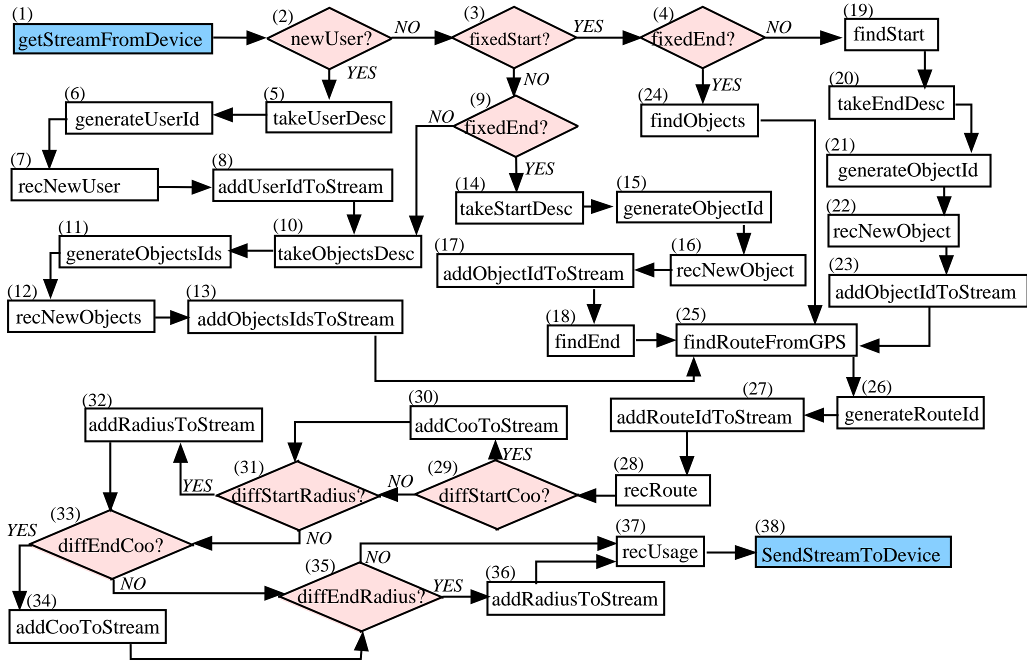


Figure 3: Server-Side Route Recording

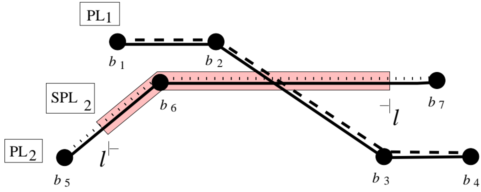
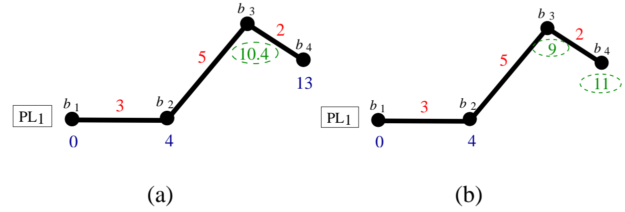


Figure 4: Example of Polylines and a Subpolyline



(a) (b)

Figure 5: Length Calculations

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 [5, 8, 13]. 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:

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

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.

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. □

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. □

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 $l^{\dashv}$. □

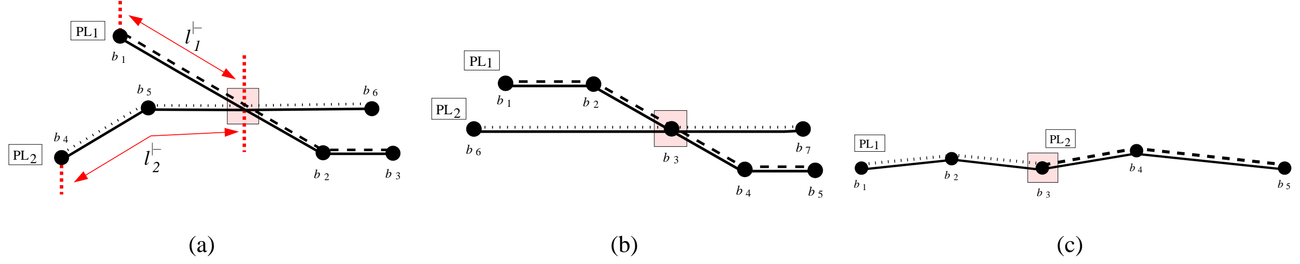


Figure 6: Connections Among Polylines

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*. □

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 $(u, \mathit{circle}, \mathit{spls})$, where

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

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) $uo_s = (u, \mathit{circle}_s, \mathit{spls}_s) \in UO$ is the start object of the route, and $\exists (pl, l^{\vdash}, l^{\dashv}) \in \mathit{spls}_s\ (pl = pl_1 \wedge (l^{\vdash} \leq l_1^{\vdash} \leq l^{\dashv} \wedge dir_1 = 1) \vee (l^{\vdash} \leq l_1^{\dashv} \leq l^{\dashv} \wedge dir_1 = -1))$.
3) $uo_e = (u, \mathit{circle}_e, \mathit{spls}_e) \in UO$ is the end object of the route, and $\exists (pl, l^{\vdash}, l^{\dashv}) \in \mathit{spls}_e (pl = pl_N \wedge (l^{\vdash} \leq l_N^{\dashv} \leq l^{\dashv} \wedge dir_N = 1) \vee (l^{\vdash} \leq l_N^{\vdash} \leq l^{\dashv} \wedge dir_N = -1))$.
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).

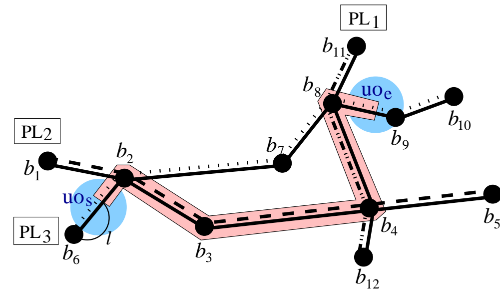


Figure 7: Example Route

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

While Section 2.2 gives an overview of the context of the essential construction of routes that occurs on the server side, we proceed to describe the route construction algorithm (Algorithm 4.1, below) in some detail. Taking a sequence G of GPS positions as input, the algorithm 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.

Algorithm 4.1 Route Finding

Require: IN: $G = (g_1, ..., g_n), g_i \in \mathbb{R}^2, n > 1$ **OUT:** $RE = ((spl_1, dir_1), ..., (spl_m, dir_m)), spl_i = (pl_i, l_i^{\vdash}, l_i^{\dashv}) \in SPL$

```
 1: let cState = ((pPl, pDst, pDir), l⊢, RE)
 2: (cState, G) ← getStartValues(G)
 3: while G is not empty do
 4:   g ← head(G), G ← tail(G)
 5:   (cPl, cDst) ← polyId(g, pPl)
 6:   if cPl = ∅ then
 7:     (cState, G) ← fillGap(cState, g, G)
 8:   else
 9:     if possibleConnection(cPl, cDst) = false then
10:       if cPl ≠ pPl then
11:         cState ← newSubOtherPoly(cState, cPl)
12:       else
13:         dir ← defDirection(pDst, cDst, pDir)
14:         if pDir = 0 then
15:           pDir ← dir
16:         else if pDir = dir then
17:           pDst ← cDst
18:         else
19:           cState ← newSubSamePoly(cState, cDst)
20:         end if
21:       end if
22:     end if
23:   end if
24: end while
25: RE ← proceedEnd(cState, cDst)
26: return (RE)
```

The state of the algorithm is captured by the data structure $cState = ((pPl, pDst, pDir), l^{\vdash}, RE)$, where pPl is the polyline the most recent, previous GPS position was mapped to, $pDst$ is the distance between that GPS position and its position on the polyline it 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 where the current subpolyline starts, and RE is a sequence of route elements.

The algorithm uses a few additional structures. Thus, $(cPl, cDst)$ stores the polyline to which the current GPS position 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 use the 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 4.2) 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 position 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 positions and g_k is the first GPS position 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. Finally, $cList = (Cand_1, \ldots, Cand_k)$ is a list of candidate sets where $Cand_i$ contains the candidates for mapping GPS position g_i.

Algorithm 4.2 Function *getStartValues*

Require: IN: $G = (g_1, ..., g_n), g_i \in \mathbb{R}^2$ **OUT:** $(cState, G) = (((pPl, pDst, pDir), l^{\vdash}, RE), G)$

```
 1: Cand ← ∅, cList ← ∅, undG ← ∅
 2: while G not empty ∧|Cand| ≠ 1 do
 3:   g ← head(G), G ← tail(G)
 4:   Cand ← polyCand(g)
 5:   if |Cand| > 0 then
 6:     cList ← append(cList, Cand)
 7:     undG ← append(undG, g)
 8:   end if
 9: end while
10: if cList > 1 then
11:   cState ← backTrack(cList, undG)
12: else if cList = 1 then
13:   (pPl, pDst) ← head(Cand)
14:   pDir ← 0, l⊢ ← pDst, RE ← ∅
15: else
16:   EXIT
17: end if
18: return (cState, G)
```

For each position g from the GPS sequence, algorithm *getStartValues* finds candidate polylines $Cand$ using function $polyCand$ (line 4). If there are more than one candidate (line 5), the algorithm adds the GPS position to list $undG$ and also adds candidates $Cand$ to list $cList$. If the first position with only one candidate is not the first GPS position in the stream (line 10), the algorithm uses function $backTrack$ to map the previous positions correctly, if possible, and to get the current state. If the first GPS position has only one candidate (line 12), the current state becomes this candidate. If all positions in the GPS stream have more than one candidate polyline (line 16), the algorithms exits.

The next function used in Algorithm 4.1, *polyId*, identifies the polyline to which a GPS position g should be mapped, considering the polyline pPl that the previous GPS position was mapped to. Position g should be mapped to polyline pPl or to a polyline that shares a connection point with this polyline. The function returns the polyline cPl and the distance $cDst$ from the start of the polyline to the projection. If position g is not mapped onto the previous polyline and more than one candidate polyline exists that connects with the previous polyline, the function returns an undefined polyline.

To avoid mapping errors at connections, we introduce so-called connections areas and do not map GPS positions inside these areas. Function $possibleConnection$ determines whether an argument GPS position is in a connection area. If the projection of the

position is within the imprecision distance from a connection, the GPS position is in a connection area, and the function returns *true*; otherwise, it returns *false*.

Function *fillGap* fills the gap between two projections based on shortest paths search in the road network representation. This function constructs missing route elements.

Function $newSubOtherPoly$ constructs a route element when the current GPS position is mapped to polyline other than the one the previous GPS position 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.

Function $newSubSamePoly$ constructs a new route element in the case where 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.

Function $defDirection$ determines the movement direction along a polyline of two projections. If the previous measure is less than the current one, the direction coincides with the polyline's direction and 1 is returned. If the previous measure is greater than the current one, the direction is set to -1. If the two measures are equal, the direction is set to the previous direction, $pDir$.

Function $proceedEnd$ constructs the last route element. All last route elements that belong to the last polyline are approximated by one element if they are in the area of the destination object.

With the above functions at its disposal, Algorithm 4.1 first uses function $getStartValues$ to obtain a correct start state. While the GPS sequence is not empty, the next position is extracted and processed. The polyline that corresponds to the position is identified using function $polyId$. If this function returns an undefined polyline, there is a gap 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 position projection is not in the connection area, the subsequent calculations can be done.

If the current polyline is not the same (line 10) as for the previous GPS position, a new subpolyline is formed. If the polyline is the same (line 12) as for the previous GPS position, the algorithm checks if the movement direction is the same as for the previous position. 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 $pDst$ becomes equal to the distance of the current GPS position. 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*. Specifically, all the last route elements constructed so far that belong to the last polyline to which GPS positions were mapped are approximated to one element if these route elements are in the area of the destination point. In Figure 8, the final point of the route is E and all subpolylines belong to the same polyline. They are inside the area of the destination point shown by the circle. Each value x_i denotes a distance from the start of the polyline.

Function *proceedEnd* 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).

A detailed description of the route construction described above is available in the associated technical report [4].

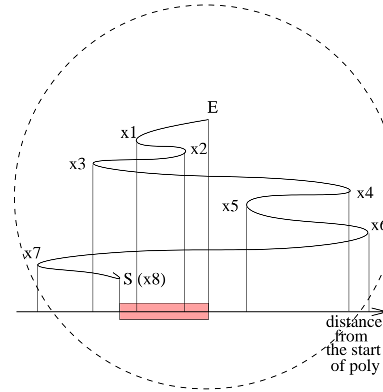


Figure 8: Approximation of the Route End

5. EXPERIMENTAL VALIDATION

To validate the data structures and algorithms described in the previous two sections, these were implemented using commercial, 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 road network and GPS log data.

5.1 Database Schema

Figure 9 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—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 point—table **POLYLINE_ELEMENTS** records these. 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 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 [10]. 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

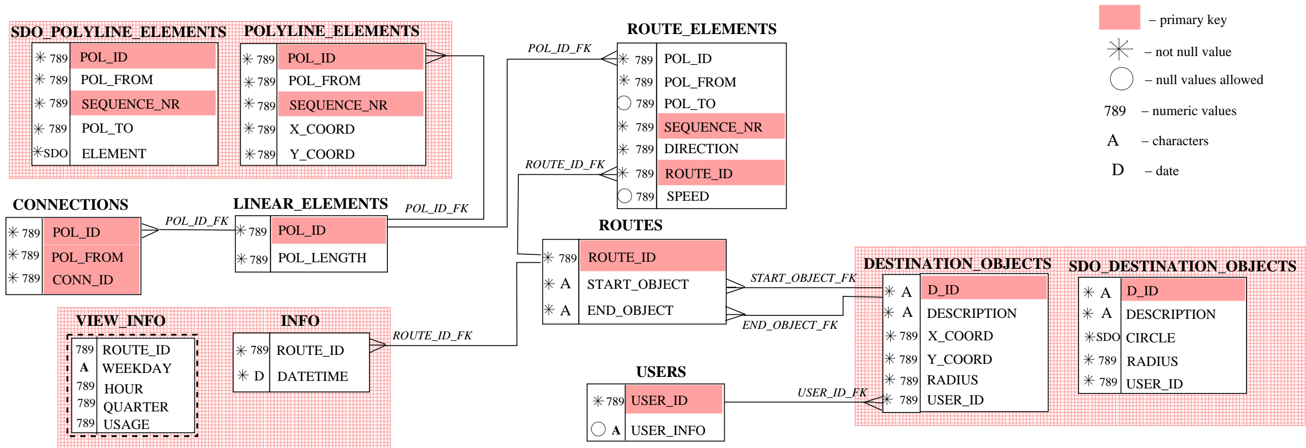


Figure 9: Relational Database Schema

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 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 algorithm described in the previous section was implemented using facilities available in Oracle Spatial [10]. Segments of polylines are spatial data objects (SDO elements in Figure 9), 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 for GPS positions and as the imprecision of connection areas.

5.3 Map and GPS Log Data

We repeatedly tested and improved the prototype component using the INFATI data [9]. This data includes a 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.

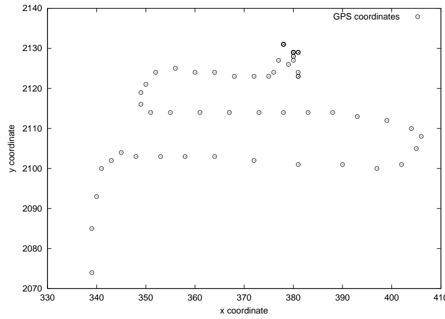
The INFATI data also includes GPS logs from twenty-some vehicles that participated in an intelligent speed adaptation project. Briefly, the position of a vehicle was logged every second when the vehicle was moving. Positions were logged for approximately six weeks.

5.4 Experimental Insights

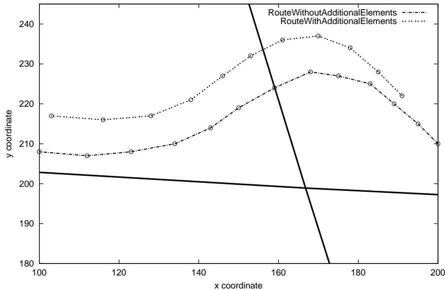
In general, the experimental validation of the component led to a more consolidated formalization of the concepts underlying the component and led to a more mature component that is able to handle the complex situations that occur in real-world application. Here, we discuss insights gained from the validation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

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 10(a), where a vehicle drives around at its destination to find an empty parking space.

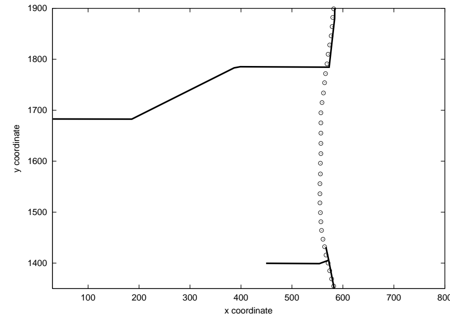


(a) End of a Route

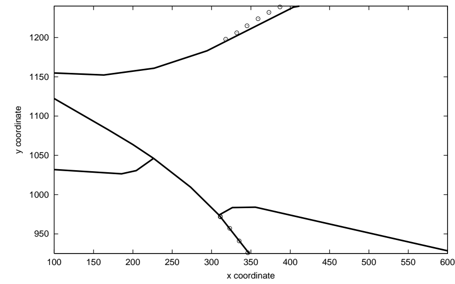


(b) Mapping at a Rotary

Figure 10: Special Cases



(a) Gap in the Map Data



(b) Gap in the GPS Data

Figure 11: Filling Gaps

The representation of rotaries in the map data 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 10(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.

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 problems. The next insights concern in large part the *absence* of map or GPS data.

Figure 11(a) illustrates a situation where a vehicle drives where the map data has no road. This occurs if the map data 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).

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 11(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 handles this case by using $fillGap$. If a gap exceeds a certain distance threshold, the component returns error.

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. More extensive empirical studies of accuracy are left for future work.

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 format depends on the number of positions to be stored. The maximum NMEA sentence length is 80 characters. For each GPS position, we use 8 sentences.

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, 5, 10, 13]. Consistent with this, our data model uses linear referencing for capturing road network as well as routes, and our data model can easily be integrated with any linear referencing model. Hage et al. [8] describe a data model that integrates representations of transportation infrastructures and geo-referenced

content. We use part of this model and extend it in order to capture routes. We note that it is also possible to model a road network as a directed graph (e.g., [7, 16]), 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. [2] 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 [3] 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 [17] 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. We map match GPS positions onto polylines. 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. [12], 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 [15]. 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.

The proposed route component makes routes available to services and may be considered as a part of a more general context-aware system. However, a more general coverage 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 routes and accumulates usage information 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 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.

9. REFERENCES

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

[2] 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.

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

[4] A. Brilingaitė, C. S. Jensen, and N. Zokaitė. Enabling Routes as Context in Mobile Services. DB Tech Report TR-9. Department of Computer Science, Aalborg University.

[5] 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.

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

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

[8] 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.

[9] C. S. Jensen, H. Lahrmann, S. Pakalnis, and J. Runge. The INFATI Data. TimeCenter TR-79, 2004. http://www.cs.auc.dk/TimeCenter.

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

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

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

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

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

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

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

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