# Accuracy and Resource Consumption in Tracking and Location Prediction[1]

Ouri Wolfson and Huabei Yin

Department of Computer Science, 851 S. Morgan (M/C 152), Chicago, IL 60607, USA
{wolfson, hyin}@cs.uic.edu

**Abstract.** Tracking is an enabling technology for many location based services. Given that the location of a moving object changes continuously but the database cannot be updated continuously, the research issue is how to accurately maintain the current location of a large number of moving objects while minimizing the number of updates. The traditional approach used in existing commercial transportation systems is for the moving object or the cellular network to periodically update the location database; e.g. every 2 miles. We introduce a new location update policy, and show experimentally that it is superior to the simplistic policy currently used for tracking; the superiority is up to 43% depending on the uncertainty threshold. We also introduce a method of generating realistic synthetic spatio-temporal information, namely *pseudo trajectories* of moving objects. The method selects a random route, and superimposes on it speed patterns that were recorded during actual driving trips.

## 1 Introduction

Miniaturization of computing devices, and advances in wireless communication and sensor technology are some of the forces that are propagating computing from the stationary desktop to the mobile outdoors. Some important classes of new applications that will be enabled by this development include location-based services, tourist services, mobile electronic commerce, and digital battlefield. Tracking, i.e. continuously maintaining in a database the transient location of a moving object, is an enabling technology for these application classes. Other application classes that will benefit from tracking include transportation and air traffic control, weather forecasting, emergency response, mobile resource management, and mobile workforce.

Often, tracking enables tailoring the information delivered to the mobile user in order to increase relevancy; for example delivering accurate driving directions, instant coupons to customers nearing a store, or nearest resource information like local restaurants, hospitals, ATM machines, or gas stations.

In addition to enabling applications, tracking is also a fundamental component of other technologies such fly-through visualization (visualized terrain changes

[1] Research supported by NSF Grants ITR-0086144, NSF-0209190, CCR-0070738, and EIA-0000516.

continuously with the location of the user), context awareness (location of the user determines the content, format, or timing of information delivered), and augmented reality (location of both the viewer and the viewed object determines the types of information delivered to viewer).

As defined, tracking involves continuously maintaining in a database the current location of moving objects. The location of a moving object is sensed either by the cellular network using techniques of triangulation among the cellular towers, or by a Global Positioning System (GPS) receiver on board the moving object. In either case, for tracking the location is transmitted to the database. Given that the location of a moving object changes continuously, but the database cannot be updated continuously, the research issue is how to accurately maintain the current location of a large number of moving objects while minimizing the number of updates.

The traditional approach used in all existing commercial transportation systems (see for example [19, 20]) is for the moving object or the cellular network to periodically update the location database; e.g. every 2 miles, or every 5 minutes. The advantage of the distance update policy, which updates the database every *x* distance units, is that it provides a bound on the error in response to database queries. Specifically, in response to a query: "what is the current location of *m*?" the answer is: within a circle of radius *x*, centered at location *l* (provided in the last database update). Similarly, in an augmented reality or context awareness application, if a Personal Digital Assistant (PDA) can hold images or graphics that pertain to an interval of 20 meters, then the moving object will provide its location every 20 meters, and get the next set of images from the database.

In this paper we introduce and evaluate an alternative to the commonly used distance update policy. It pertains to motion on the road network given by a map, and it improves the performance of tracking by location prediction. In this sense it is a location *prediction* policy as much as a location *update* policy, but we will still use the term update for uniformity. It called the deviation policy, and it predicts that following a location update, the moving object will continue moving on the same street. This assumption provides an *expected* location at any point in time after each location update, and thus in response to future queries it will provide the expected location. The moving object or the network will update the database whenever the *actual* location deviates from the expected location by more than a given threshold *x*. Thus, in terms of accuracy, the deviation policy is identical to the distance policy in the sense that both allow for a maximum location error; in other words, given the same *threshold x*, both methods have the same location uncertainty. Thus one can compare how many updates are required by each policy to maintain a given uncertainty threshold x. This is a measure of the location prediction power of the deviation policy. We discover that the deviation policy is up to 43% more efficient than the distance policy (which is currently the state of the art) in the sense that to maintain an uncertainty threshold of 0.05 miles the distance policy uses 43% more updates than the deviation policy. The advantage of the deviation policy decreases as the uncertainty threshold increases, but it is always better than the distance policy.

Now consider a data streaming application in virtual or augmented reality, or fly through visualization. Suppose that a vehicle is driving through a city and at any point in time it needs to show on the passenger's handheld computer the objects (buildings, stores, mountains) within *r* miles of the vehicle's current location. Suppose that the computer has a limited amount of memory that allows it to store objects within *R* miles of the vehicle's location, and the data is streamed to the mobile computer by a server that tracks the vehicle. Then if the server makes a tracking error of more than *R-r* miles, then the mobile computer will have to update its database location at the server and request an up-to-date chunk of data (interrupting the display flow while waiting for that chunk). In this sense *R-r* corresponds to the error threshold mentioned above, and the distance policy will generate up to 43% more location updates/requests than the deviation policy.

Another way to view these results is as follows. There are quite a few works that examine the efficiency of indexing methods in moving objects databases (see for example [16, 21]). In other words, these works examine how to make updates (and also retrievals) more efficient. However, efficiency can be addressed not only by making each update more efficient, but also by reducing the number of required updates. And this is the approach we take in this paper, i.e. we examine which location update policy will minimize the total number of location updates for each given uncertainty threshold.

In addition to the new location update policy, another major contribution of this paper is a method of generating realistic synthetic spatio-temporal information, namely *pseudo trajectories* of moving objects. Observe that in order to compare the above location update policies for a given uncertainty threshold, one needs to know how the speed of a vehicle changes during a trip. This can be recorded by driving and using a GPS RECEIVER, however, but for an accurate comparison one would have to use traces of hundreds of trips. We solve this problem by tracing three trips of about 24 miles each, thus generating three real trajectories. Pseudo trajectories are trajectories that are not generated by driving, but by selecting a random route, and superimposing on it speed patterns that were generated along the real trajectories.

**In summary, the main contributions of this paper are**: (1) We introduce the deviation location update policy, and show experimentally that it is superior to the simplistic policy currently used for tracking; the superiority is up to 43% depending on the uncertainty threshold. (2) We introduce a method of generating realistic synthetic spatio-temporal information, namely *pseudo trajectories* of moving objects. The method selects a random route, and superimposes on it speed patterns that were recorded during actual driving trips.

The rest of this paper is organized as follows. The data model and two update policies are introduced in section 2. In section 3, we first present the test databases and a method of generating pseudo trajectories of moving objects; and then the experimental comparison of the update policies is discussed. Section 4 discusses the related work. Finally we conclude the paper and propose the future work in section 5.

## 2 Location Update Policies

In this section we first introduce the database model used for tracking (subsection 2.1), and then in each one of the following two subsections we introduce an update policy.

## 2.1 The Data Model

In this section we define the main concepts used in this paper. We define a map, and what tracking means in database terminology. We also define the notion of a trajectory.

An *object-class* is a set of attributes. Some object classes are designated as *spatial*. Each spatial object class is either a point class, or a polygonal line (or *polyline* for short) class.

**Definition 1.** A map is a spatial object class represented as a relation, where each tuple corresponds to a block with the following attributes:

- Polyline: Each block is a polygonal line segment in 2D. Polyline gives the sequence of the endpoints: ($x_1$, $y_1$), ($x_2$, $y_2$), …, ($x_n$, $y_n$). Where ($x_1$, $y_1$) is the start point of this polyline, and ($x_n$, $y_n$) is the end point.
- Street Name: The street name of this block.
- Street Category: six categories, from major highway, to undivided local road.
- Bid: The block id number.
- Speed: The average driving speed on this block
- One_way: (boolean) One way indicator

Plus, among others, a set of geo-coding attributes which enable translating between an (*x*, *y*) coordinate and an address, such as "*2222 W Taylor St.*" (e.g. R_f_add and R_t_add: the from and to street numbers on the right side, respectively.)

The start point ($x_1$, $y_1$) of a block is the end point ($x_n$, $y_n$) of the preceding block, and an intersection of two streets is the endpoint of the four blocks – polylines. A *street-polyline* is the concatenation of the polylines belonging to all the blocks of the street. Thus each map is a graph, with the tuples representing edges of the graph. Such maps are provided by, among the others, Geographic Data Technology[2] Co (GDT).

Point object classes are either mobile or stationary. A point object class *O* has a *location attribute L*. If the object class is *stationary*, its location attribute has two sub-attributes *L.x*, and *L.y*, representing the *x* and *y* coordinates of the object. If the object class is *mobile*, its location attribute has six sub-attributes, *L.street*, *L.location*, *L.time*, *L.direction*, *L.speed*, and *L.uncertainty*. The attribute represents the location of the object at the last location update.

The semantics of the sub-attributes are as follows.

*L.street* is (the pointer to) street polyline indicating the street on which an object in the class *O* is moving.

*L.location* is a point (*x*, *y*) on *L.street*; it is the location of the moving object at time *L.time*. In other words, *L.time* is the time when the moving object was at location *L.location*. We assume that whenever the system updates the *L* attribute of a moving object it updates at least the *L.location* sub-attribute; thus at any point in time *L.time* is also the time of the last location-update. We assume in this paper that the database updates are instantaneous, i.e. valid- and transaction-times (see [18]) are equal. Therefore, *L.time* is the time at which the location was sensed in the real world system being modeled, and also the time when the database installs the update. In practice, if the difference between these two times is not negligible, then all our

[2] www.geographic.com

results still hold by simply including the location-sensing time in the update message to the server.

*L.direction* is a binary indicator having a value 0 or 1 (these values may correspond to north-south, east-west, or the two endpoints of the street). *L.speed* is a function of time that represents the expected speed of the object after the last update, *L.time*. It enables computing the distance of the moving object from *L.location* as a function of the number $t$ of time units elapsed since that last location-update, namely since *L.time*. In its simplest form (which is the only form we consider in this extended abstract) *L.speed* represents a constant speed $v$, i.e. this distance is $v \times t$.

We define the *street-distance* between two points on a given street to be the distance along the street polyline between the two points. The street distance between two points can be easily computed in linear time (in the size of the street polyline), and so can the point at a given street-distance from another given point. The *database location* of a moving object at a given point in time is defined as follows. At time *L.time* the database location is *L.location*; the database location at time *L.time*+$t$ is the point ($x$, $y$) which is at street-distance *L.speed*·$t$ from the point *L.location*.

Loosely speaking, the database location of a moving object $m$ at a given time point *L.time*+$t$ is the location of $m$ at that time, as far as the DBMS knows; it is the location that is returned by the DBMS in response to a query entered at time *L.time*+$t$ that retrieves $m$'s location. Such a query also returns the uncertainty defined below.

Intuitively, the difference among the update policies discussed in this paper is that they treat differently the time until which the database location is computed in this fashion. Remember that the server computer does not know the destination or the future location of the object. Thus the distance policy assumes that the database location does not change until the next location-update received by the DBMS. In contrast, the deviation policies assume that following a location update, the object moves on the same street with a speed given by *L.speed*, until the object reaches the end of the street; after that time, the database location of the object is the point at the end of the street until the next location update.

Now we discuss when the next update occurs. *L.uncertainty* is a constant, representing the threshold on the location deviation (the deviation is formally defined below); when the deviation reaches the threshold, the moving object (or the network sensing its location) sends a location update message. Thus, in response to a query entered at time *L.time*+$t$ that retrieves $m$'s location, the DBMS returns an answer of the form: $m$ is within a circle with radius *L.uncertainty* centered at the point which is the database location of $m$.

Since between two consecutive location updates the moving object does not travel at exactly the speed *L.speed*, the actual location of the moving object deviates from its database location. Formally, for a moving object, the *deviation* $d$ at a point in time $t$, denoted $d(t)$, is the straight-line Euclidean distance between the moving object's actual location at time $t$ and its database location at time $t$ (notice that the deviation is not computed using the street distance). The deviation is always nonnegative. At any point in time the moving object (or the network sensing its location) knows its current location, and it also knows all the sub-attributes of its location attribute. Therefore at any point in time the computer onboard the moving object (or the network) can compute the current deviation. Observe that at time *L.time* the deviation is zero.

At the beginning of the trip the moving object updates[3] all the sub-attributes of its location (L) attribute. Subsequently, the moving object periodically updates its current *L.location*, *L.speed*, and *L.street* stored in the database. Specifically, a *location update* is a message sent by the moving object to the database to update some or all the sub-attributes of its location attribute. The moving object sends the location update when the deviation exceeds the *L.uncertainty* threshold. The location update message contains at least the value for *L.location*. Obviously, other sub-attributes can also be updated, and which ones will be updated when will become clear when we describe the policies. The sub-attribute *L.time* is written by the DBMS whenever it installs a location update; it denotes the time when the installation is done.

When an update message is received, before the L attribute is updated, the previous values are used in order to extend the current trajectory (see the next definition below). This trajectory is used in order to answer queries about the location of the moving object in the past.

Representing the (*location, time*) information of the moving object as a trajectory is a typical approach (see [10, 11, 12]) and we use the definition introduced in [11]:

**Definition 2.** *A* trajectory *Tr of a moving object is a piece-wise linear function from time T to 2D space Tr: T*$\rightarrow$*(X,Y), represented as a sequence of points* $(x_1, y_1, t_1), (x_2, y_2, t_2), \ldots, (x_n, y_n, t_n)$ $(t_1 < t_2 < \ldots < t_n)$*. The projection of Tr on the X-Y plane is called the* route *of Tr.*

A trajectory defines the location of a moving object as a function of time. The object is at $(x_i, y_i)$ at time $t_i$, and during each segment $[t_i, t_{i+1}]$, the object moves along a straight line, at constant speed, from $(x_i, y_i)$ to $(x_{i+1}, y_{i+1})$. Thus,

**Definition 3.** *Given a trajectory Tr, the* expected location of the object at a point in time t between $t_i$ and $t_{i+1}$ $(1 \leq i < n)$ *is obtained by a linear interpolation between* $(x_i, y_i)$ *and* $(x_{i+1}, y_{i+1})$*.*

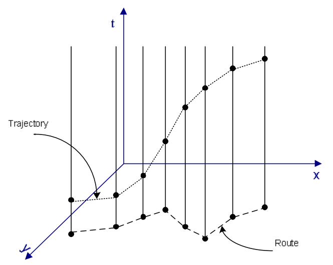


**Fig. 1.** A trajectory and its two dimensional route

An illustration of trajectory and its route is shown in Fig. 1.We can obtain the real trajectory of a moving object by using an on board GPS receiver while it moves on the road network, or simply generate it by some algorithms, such as our pseudo trajectory generation algorithm that is described in Section 3.

[3] For the rest of the paper we will assume for simplicity of presentation that the location updates are generated by the moving object itself. However, as explained above, the updates can also be generated by the network that continuously senses the moving object location.

It should be clear that the trajectory can be extended in linear time (in the number of straight line segments of the map, traversed since the last update) at each update.

Finally, let us observe that all our concepts and results are presented for motion in two dimensions, but they can be extended for 3D motion.

## 2.2 The Distance Policy

The distance location update policy is widely used in moving objects database applications due to its simplicity. It dictates that the moving object *m* sends an update message to the database server if the *Euclidean*-distance or the *route*-distance[4] between the current location and *L.location* exceeds the *L.uncertainty* threshold. *L.location* is simply the point on the road network that is closest to the GPS point that generated the last location update. Remember that *m* knows its current location from its on-board GPS receiver.

In other words, when a location update is generated, this policy "snaps" the GPS point that caused the location update onto the road network, and stores the snapped point in the sub-attribute *L.location*. Snapping is necessary since, although GPS receivers are accurate to within a couple of meters most of the time, sometimes the GPS receiver on board a vehicle traveling on the road network senses a location that is off the road.

## 2.3 The Deviation Policy

The deviation update policy proposed in this paper is as follows. The moving object *m* still updates the database when the deviation between the current location and the database location exceeds the *L.uncertainty* threshold. However, the database location is computed differently. At any time point *t* between the time point *L.time* and the time point when *m* reaches the end of *L.street* (in the travel direction given by *L.direction*), the database location is computed by $L.speed \times t$; i.e. the database location is on *L.street* at street distance $L.speed \times t$ from the last update location, i.e. *L.location*. After the time when *m* reaches the end of *L.street*, the database location is the end of the street. In other words, the prediction is that the moving object continues on the current street until its end; after the end is reached, the next location cannot be predicted; therefore the database awaits the next location update.

*L.location* is the point on the road network that is closest to the GPS point that generated the location update. *L.street* is the street on which *L.location* lies. *L.speed* is the predicted speed. It can be the current speed at the last location-update from the object (commercially available GPS receivers provide the current speed in addition to the current location), or, more generally, it can be the average of the speed-readings during the latest *x* time units. Thus, for the deviation policy, the location update message also includes the value for the *L.speed* sub-attribute.

[4] The *route-distance* between two points x and y on the road network is the length of the shortest path between x and y in the network represented by the map.

Next we discuss how the value of *L.direction* is determined. The two possible values of *L.direction* are the two endpoints of the street *L.street.* The value for *L.direction* is determined at each location update as follows. If the previous location update was on the same street, it is easy to get the value for the sub-attribute *L.direction*. It is simply the endpoint that is closer to the last update than to the previous one. For example consider street $ST_1$ in Fig. 2. Its endpoints are E and F. If A and B are two consecutive location updates on the street $ST_1$, then the *L.direction* computed at the second one is E (in other words the direction is FE). Otherwise, i.e. if the last two location updates are on two different streets, *L.direction* is computed as follows. Let C be the current location update, and let B the previous one. Let D be the closest point on the current *L.street* to the location B (i.e., the shortest path (in terms of travel time) in the map graph from B to D is not longer than the path from B to some other intersection on *L.street*). Then the endpoint of *L.street* that is closer to C than to D determines the current *L.direction*. For example, assume that B and C are two consecutive location updates on two different streets $ST_1$ and $ST_2$ respectively (see again Fig. 2). Assume further that C is on the street $ST_2$ with endpoints G and H. Then the value of *L.direction* computed at C-update time is H, where D is the closest intersection to B among all intersections on $ST_2$.

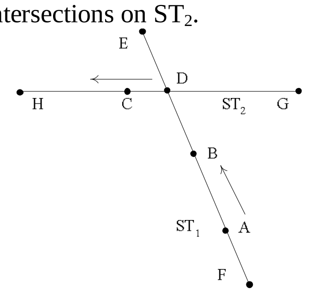


**Fig. 2.** The direction of the movement of the moving object

Finally, we make two observations. First, we discussed how the *L* sub-attributes are computed, but we did not discuss whether the computation is performed by the DBMS or the client on the moving object. If both the client and the DBMS server store the map, then the computation can be performed in either. If the map is stored only in the server, then the computations are performed in the server, which downloads to the client the street polyline from *L.location* until the endpoint, and the sub-attributes *L.direction* and *L.speed*.

The second observation is that the computation of *L.street* and *L.direction* may produce errors. Specifically, the GPS point may be snapped onto the wrong street, and even if the street is right, the computed direction may be wrong. The only implication of such errors is an increase in the number of updates. In other words, if travel is assumed on one street but actually occurs on another, then the deviation will grow faster, and the next update will occur sooner.

## 3 Experiments

In this section we present our experimental setup. In subsection 3.1 we describe the data (i.e. trajectories) used in our experiments. In subsection 3.2 we describe the experiments and the results of comparing the three location update policies.

### 3.1 Test Data

First let us point out the difficulty in generating realistic data for comparing the update policies. Observe that the number of updates generated by the distance based policy per mile can be easily generated as follows. A random route can be generated by selecting two random points on the map, and connecting them by the shortest path. For each route, the number of updates for a given threshold *th* can be easily computed by finding the first point that is at Euclidean- or route- distance *th* from the beginning of the route, then the next point, that is at Euclidean- or route- distance *th* from the previous one, etc. Thus, by generating hundreds of random routes, the average number of updates per mile, per threshold *th,* can be computed. However, the problem is computing the number of updates for the deviation policy, because this depends on the way the speed of the moving object varies over time. Thus the method outlined above for the distance policy will not work. In other words, we need to generate trajectories which reflect a realistic variation of speed over time. This is the procedure described precisely in this subsection. Intuitively, we do so as follows. First we generate a few real trajectories by actual driving on routes that include streets of the various categories, and recording the speed every two seconds. Then, the set of recorded speeds is used to generate pseudo trajectories that have a realistic speed-change pattern. In the first subsection we describe the generation of three real trajectories by actual driving on roads, and then, in the next subsection we describe the generation of 80 pseudo trajectories.

#### 3.1.1 Real Trajectories Database

We actually drove in the Chicago metropolitan area and collected 3 real trajectories over routes that cover all street categories. Each trajectory describes a trip of average length 24.36 miles from a source to a destination. The driving conditions included both, heavy and light traffic. Information related to the real (i.e. driven) trajectories is shown in Table 1.

For each trip, a real trajectory is generated. The *real* trajectory of a moving object is obtained by repeatedly reading the (*longitude*, *latitude*, *time*) from a Differential GPS device (which has an accuracy of a couple meters; see [13]) connected to a laptop. A reading is taken every two seconds. The sequential file of DGPS readings is used to generate a trajectory that has an $(x_i, y_i, t_i)$ point for every two seconds. So, we obtain a sequence of points $(x_1, y_1, t_1)$, $(x_2, y_2, t_2)$, …, $(x_n, y_n, t_n)$ $(t_1 < t_2 < \ldots < t_n)$, that is a trajectory representing the speed changes over time during an actual trip. Based on each straight line segment in a trajectory, the average speed during the respective two-second interval can be computed.

**Table 1.** Meta data about the real trajectories

| trajectory id | length | number of 3D points |
|---|---|---|
| 1 | 24.6469 | 1318 |
| 2 | 23.441 | 2276 |
| 3 | 24.9924 | 3183 |
| average | 24.3601 | 2259 |

### 3.1.2 Pseudo Trajectories Database

Pseudo trajectories are trajectories that are not generated by driving, but by selecting a random route, and superimposing on it speed patterns that were generated along the real trajectories. More precisely, the procedure of generating a pseudo trajectory is as follows:

(1) Randomly pick an integer *m* between 3 and 6 as the number of source and intermediate destinations. Then randomly generate a sequence of *m* locations in the map (see Definition 1). Then find the shortest path that travels this sequence (for example, if *m* is 3, then the pseudo trajectory starts at location $l_1$, moves to location $l_2$, and from there to the final location $l_3$. This corresponds to, for example, a delivery route that starts at $l_1$, and drops off at $l_2$ and from there it goes and drops off at $l_3$). Thus the shortest path is a polyline consisting of the concatenation of block polylines (see Definition 1). It is also the route *R* of the pseudo trajectory. The size of the route is the number of points of that route.

(2) Generate a sequence of speeds for the route *R*.

In order to explain how we execute step (2) above, we first need to define the notion of a Speed Pattern of a real trajectory:

**Definition 4.** *A* speed pattern *is a sequence of consecutive 2-second speeds in a real trajectory, for a given street category.*

Where the street category is defined by electronic maps (see Definition 1). There are six street categories in electronic maps (see for example [2]):

**A1**, LIMITED ACCESS HIGHWAYS

**A2**, PRIMARY or MAJOR HIGHWAYS

**A3**, SECONDARY or MINOR HIGHWAYS

**A4**, LOCAL ROADS

**A5**, VEHICULAR (4WD) TRAIL

**A6**, ROADS WITH SPECIAL CHARACHTERISTICS

There are several subcategories in each category. However, for our classification purposes, the subcategories do not matter. In the Chicago metropolitan area map used in our experiments, A6 streets are the entrance ramps and the exits of the expressways, and there are no streets of category A5. Since A5 is a slow road, and in general there are very few roads of this type on maps, A5 is classified as a minor road for the hybrid policy. Thus we use five different street categories for classifying the street categories into major and minor classes. The street categories analyzed are shown as in the Table 2, along with the average travel speed on each. Observe that, for highways for example, the maximum speed is 55, but the average speed given in the map is 45.

**Table 2.** Categories of streets in a major U.S. metropolitan area, along with the average speed in miles/hour

| **Category** | **A1** | **A2** | **A3** | **A4** | **A6** |
|---|---|---|---|---|---|
| **SPEED** | 45 | 35 | 25 | 25 | 20 |

To obtain the speed patterns we first calculate the average speed for every two consecutive real trajectory points. A sequence of consecutive speeds that belong to (i.e. were taken on) the same street category forms a speed pattern. For example, suppose that for a real trajectory *T* we have a set of points on an A3 street, as shown in Table 3; the time *t* is relative from the start of the trajectory, in seconds. Suppose further that after the 20th second, during the trip that generated *T*, the moving object entered a different street category. Then we obtain a speed pattern for street category A3, as shown in the right table of Table 3(corresponding to the left table of Table 3); the unit of speed is miles per hour, and each speed in table 4 is called a *speed-item* of the speed pattern.

**Table 3.** Left Table: A set of trajectory points on an A3 street; Right Table: a speed pattern calculated from the left table

| **x** | **y** | **t** |
|---|---|---|
| 221.311 | 358.893 | 0 |
| 221.311 | 358.892 | 2 |
| 221.31 | 358.892 | 4 |
| 221.309 | 358.892 | 6 |
| 221.307 | 358.892 | 8 |
| 221.293 | 358.89 | 10 |
| 221.276 | 358.889 | 12 |
| 221.257 | 358.888 | 14 |
| 221.237 | 358.887 | 16 |
| 221.216 | 358.887 | 18 |
| 221.195 | 358.886 | 20 |

| **speed_id** | **speed** |
|---|---|
| 1 | 1.8 |
| 2 | 1.8 |
| 3 | 1.8 |
| 4 | 3.6 |
| 5 | 25.4558 |
| 6 | 30.6529 |
| 7 | 34.2473 |
| 8 | 36.0450 |
| 9 | 37.8 |
| 10 | 37.8428 |

The length of one speed pattern is the number of speeds of that pattern. All the speed patterns of the same street category form the *Speed Pattern Set (SPS)* of that street category. Thus a speed pattern set of a street category represents the patterns (collected during our real trajectory generation trips) of speed changes while for that street category. The size of one *SPS* is the sum of the lengths of its speed patterns. Table 4 is an example of the speed pattern set for the street category A3. There are 5 speed patterns in Table 4 whose pattern_id's are 1, 2, 3, 4, and 5 respectively. For the street categories A1, A2, A4 and A6, we have 2, 4, 5 and 3 speed patterns in our SPS repository, respectively. Overall, the average length of a speed pattern is 355.526.

**Table 4.** The speed pattern set for A3

| **pattern_id** | 1 | 1 | …… | 2 | …… | 5 | 5 | …… |
|---|---|---|---|---|---|---|---|---|
| **speed id** | 1 | 2 | …… | 1 | …… | 1 | 2 | …… |
| **speed** | 14.5121 | 16.0997 | …… | 1.8 | …… | 19.2996 | 17.9999 | …… |

Now we are ready to describe the details of step (2) above, namely the generation of a pseudo trajectory based on a route R and the SPS's for the various street categories. Intuitively, we do so as follows. Suppose that we generated the pseudo trajectory up to a location *l* on route *R*. Suppose further that *l* is on a category A1 street. Then we select from the A1 SPS a random speed from all the speeds that are equal (or, if no such exists, closest) to the speed of the moving object at *l* (see further discussion of this in the next paragraph). Then "run" the moving object with the pattern to which the speed belongs, starting at the location of the speed within the pattern; do so until the end of the current sequence of A1 streets on R, or until the end of the speed pattern, whichever occurs first. The reached location is the new *l* location, and the speed at this location is the speed of the pattern. The process repeats until the end of the route R.

Observe that the speeds that are equal to the speed at location *l* may belong to different speed patterns; and if they belong to the same pattern, they will appear at different sequential locations from the start of the pattern. For example, consider Fig. 3. Suppose that *a* in speed pattern 1, and *b*, *c* in speed pattern 2, are the speeds that are equal to the speed at location *l*. Then, if the random selection picks *a*, then speed pattern 1 is run for 5 2-second intervals (assuming that the end of the street category is not reached beforehand). If *b* is picked, then speed pattern 2 is run for 5 2-second intervals; and if *c* is picked then it is run for one interval (the last one of speed pattern 2).

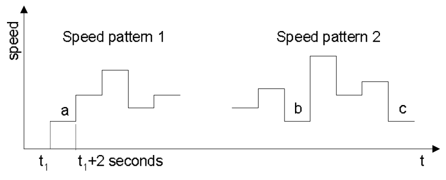


**Fig. 3.** Selection of the speed pattern

Observe additionally that when a speed pattern is "run", then trajectory points are generated at the end of each 2-second interval, and also at the end of each straight line segment of the route. Formally the trajectory generation algorithm is as follows:

```
Algorithm 1 Generate Pseudo Trajectory Algorithm
Input: Map $M$, Route given as a sequence of points (see step (1) at beginning of this subsection) $R = (r_1, r_2, \dots, r_n)$, Speed Pattern Sets $SPS$ of all street categories.
Output: Trajectory $Tr$.
1.  i = 1, $current\_category$ = the street category of $r_1$, $current\_time = 0$, $current\_speed = 0$, $current\_p = (r_1, current\_time)$.
2.  Append $current\_p$ to $Tr$.
3.  while $i \le n$
4.      Randomly select a speed pattern from the $SPS$ labeled by $current\_category$ such that the selected speed pattern has a speed item $sp_j$ that is closest to $current\_speed$ in the $SPS$.
5.      Use the speed $sp_j$ to compute the next location $p$ where street-distance between $r_i$ and $p$ is $sp_j \times 2/3600$, and the direction of the motion is from $r_i$ to $r_{i+1}$.
6.      If p exceeds the $r_{i+1}$ (i.e. the street-distance between the location of $current\_p$ and $p$ is larger than the street-distance between the location of $current\_p$ and $r_{i+1}$)
        Then compute the traveling time $t'$ from the location of $current\_p$ to $r_{i+1}$ by the speed $sp_j$. $current\_p = (r_{i+1}, current\_time + t')$, $current\_time = current\_time + t'$, and append $current\_p$ to $Tr$. $i = i + 1$, $current\_category$ = the street category of $r_i$, and goto step 8.
        Else $current\_p = (p, current\_time + 2)$, and $current\_time = current\_time + 2$, and append $current\_p$ to $Tr$.
7.      If the location of $current\_p = r_{i+1}$
        Then $i = i + 1$, $current\_category$ = the street category of $r_i$.
        Else, j = j + 1.
8.      If the current_category is changed
        Then current_speed = $sp_j$, and goto step 4
        Else j = j + 1
9.      If the end of selected speed pattern is reached
        Then current_speed = the last speed-item of the selected speed pattern, and goto step 4
        Else goto step 5
10.   end while
```

**Fig. 4.** Algorithm 1

All points ($x_i$, $y_i$, $t_i$) appended in order by Algorithm1 form a pseudo trajectory. The worst case time complexity of the algorithm is $O(k \lg m + n \lg m + m \lg m)$ where k is the length of the route divided by the minimum nonzero speed (i.e. the maximum number of 2-second intervals that it takes to traverse the route), n is number of vertices on the route, and m is the number of speeds in the speed patterns. The reason for this is that for every 2-seconds or for every point on the route, the equal speeds in the speed pattern set have to be found; and if the speeds in each pattern have been sorted (takes $O(m \lg m)$ ), then this can be done in $O(\lg m)$ .

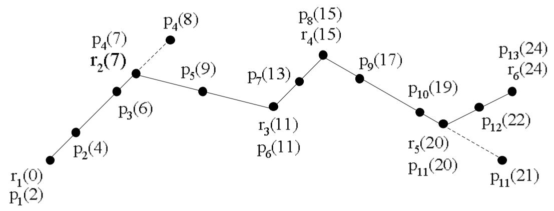


**Fig. 5.** Example 1

**Example 1** Suppose that $r_i$'s are the points of the route, and $p_i$'s are the points of the trajectory (see Fig. 5). Observe that according to the algorithm, every point of the route is also a point of the trajectory. The number in parentheses next to a $r_i$ or $p_i$ is time (in seconds) of the trajectory at the $r_i$ or $p_i$ location, for example, it takes the moving object 7 seconds to reach $p_4$. Suppose that the street category of $r_1$ and $r_2$ is A3, that's the route from $r_1$ to $r_3$ is of category A3, and the street category of $r_3$, $r_4$, $r_5$, and $r_6$ is A4, that's the route from $r_3$ to $r_6$ is of category A4. First ($r_1$, 0) is appended to *Tr* in step 2. A speed pattern from A3 *SPS* is chosen in step 4, and suppose that it is (0, 2, 2, 15, 20, 21, 23, 25.5, 26, 27, 30, 31, 26, 23, 22, 27). We use the speeds starting from 0 miles/hour to compute the locations $p_1$ to $p_4$ in four iterations of step 5; and find out in step 6 that $p_4$ exceeds $r_2$. So we re-compute the traveling time *t'* from $p_3$ to $r_2$; it is 1 second, hence the $r_2$ with its arrival time (7 = 6 + 1) becomes the next point $p_4$ of the trajectory. At $p_4$ the street category doesn't change, thus we continue the same speed pattern to compute $p_5$ and $p_6$ in the next two iterations of step 5. In step 7, we find out that $p_6$ equals to $r_3$, and in step 8 we find out that the street category changes to A4. So we select a speed pattern from A4 *SPS* in step 4 which has a speed item closest to 21 miles/hour (we end up with 21 miles/hour of the previous selected speed pattern), suppose that it's (27, 26, 21, 26, 27, 30, 31, 29, 27, 24, 27, 30, 33, 35). Then we use the speeds starting from 21 miles/hour to compute $p_7$ and $p_8$ in the next two iterations of step 5; in step 7 we find out that $p_8$ equals to $r_4$. At $p_8$ the street category doesn't change, thus we continue the same speed pattern to compute $p_9$ to $p_{11}$ in the next three iterations of step 5. Since in step 6 we find out that that $p_{11}$ exceeds $r_5$, we re-compute the traveling time $t'$ from $p_{10}$ to $r_5$; it is 1 second, hence the $r_5$ with its arrival time (20 = 19 + 1) becomes the next point $p_{11}$ of the trajectory. At $p_{11}$ the street category is still A4, thus we continue the same speed pattern to compute $p_{12}$ and $p_{13}$ in the next two iterations of step 5. In step 7, we find out that $p_{13}$ equals to $r_6$. Finally, in step 3 we find out that the end of the route is reached. The generated trajectory $Tr$ is (($r_1$, 0), ($p_1$, 2), ($p_2$, 4), ($p_3$, 6), ($p_4$, 7), ($p_5$, 9), ($p_6$, 11), ($p_7$, 13), ($p_8$, 15), ($p_9$, 17), ($p_{10}$, 19), ($p_{11}$, 20), ($p_{12}$, 22), ($p_{13}$, 24)) (see Fig. 5).

## 3.2 Experimental Comparison of Policies

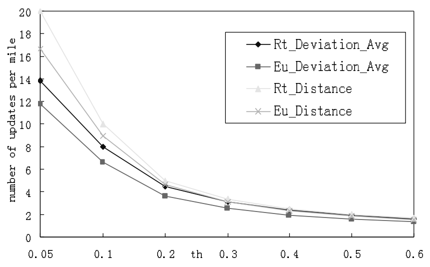
In this subsection we discuss the experimental results. We simulated the distance and deviation policies on a database of 80 pseudo trajectories. The total length of the routes traversed is 6290.325 miles, and the total number of 3D points of the trajectories is 516111. The threshold was considered in terms of both, the Euclidean distance and the route-distance. The variant of the distance policy that we used establishes the predicted speed as the average of the speeds of each time unit since the last update. In other words, the predicted speed is obtained at location-update time by taking the speed of each time unit (2 seconds) since the last location update, and averaging them. Intuitively, the motivation for this is that the predicted speed depends on the expected period of time until the next update. And assuming that the length of the time interval between consecutive location updates does not vary much, this method (called *adaptive speed prediction*) naturally adapts the number of time units to the time interval between updates.

Fig. 6 and Fig. 7 present the number of updates per mile, as a function of *th* which ranges from 0.05 to 5 miles. The four curves correspond to the two policies, with each one evaluated for the route- and Euclidean- distances. The number of updates per mile UPM is expressed as the following formula:

$$\text{UPM} = \text{the total number of updates/the total length of the routes} \qquad \mathbf{(1)}$$

We observe that the deviation policy is at least as good as the distance policy, for both Euclidean- and route- distance thresholds. The advantage of the deviation policy starts at 43% for *th* = 0.05, and it decreases as the threshold increases. The reason for the decrease is that the likelihood that the vehicle stays on the same street decreases as the threshold (and therefore the time interval between consecutive updates) increases.

Next we wanted to determine to the speed-prediction power of the adaptive speed prediction method mentioned above at the beginning of the section. Thus we compared it with the naïve method that simply predicts that the speed will be the same as during the last time unit (last 2 seconds). The results of this experiment are that surprisingly to us, the difference between the two methods is not very significant.



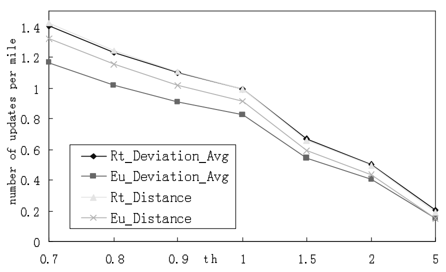


**Fig. 6** and **Fig. 7.** The number of updates per mile with *th* = *0.05 −5.0*

## 4 Related Work

As mentioned in the introduction, this paper makes contributions in update policies and synthetic spatio-temporal database generation, thus the relevant work applies to two these categories.

In the area of update policies, several works have been published [1, 17]. In [1] we propose dead-reckoning update policies to reduce the update cost. However these policies are only applicable when the destination and motion plan of the moving objects is known a priori. In other words, inherent in the policies is that the route is fixed and known to both the moving object and the server, and the update policy is used to revise the time at which the moving object is expected to be at various locations on the fixed route. However, often the destination or future route of a tracked object is not known. For example, the wireless service provider tracks every cellular phone, but it does not know the future route or destination. The user may simply not be willing to enter this information into the computer and provide it to the tracking server, or this information may change too often to make it practical.

In [17], Lam et al. propose two location update mechanisms for increasing the level of "correctness" of location-dependent continuous query results returned to clients, while reducing the location update cost of moving objects. Their methods are based on the assumption that the moving objects "covered" by the answers of the queries need more updates. In fact, their idea can be used in the present work in the sense that moving objects that are covered by continuous queries have a lower threshold, resulting in a higher location accuracy. In other words, their work discusses the selection of uncertainty thresholds for each moving object for continuous queries, whereas our work addresses the selection of update policies after the thresholds are decided.

The performance of update policies on moving objects databases with secondary access methods, for instance, R-tree or Quad-tree, is quite an interesting research issue. [21] examines the techniques that make updates more efficient: 1) by processing the individual updates more efficiently, and 2) by reducing the number of updates. Consistent with the second point, our approach seeks heuristics for future movement prediction.

In the area of synthetic spatio-temporal database generation, four methods have been published. First method is by Theodoridis et al. (see [3, 4]). They propose the GSTD (Generate_Spatio_ Temporal_Data) algorithm. It generates a set of trajectories stemming from a specified number of moving objects. Random or probability functions are used in simulating the movements of the objects as a combination of several parameters. The motion of points are always ruled by a random distribution of the form random(x, y), so that an unbiased spread of the points in the workspace can be achieved. The trajectory is a sequence of points which can be connected by linear interpolation. Thus the speed of each moving object is generated randomly. In contrast, our speed generation is based on actual driving traces (real trajectories).

Brinkhoff proposes a strictly network-followed algorithm to generate spatio-temporal databases which can simulate the moving behaviors of the real world [5, 6]. He stresses many user-defined factors which affect the behaviors of moving objects. One of these factors is "the number of moving objects influences the speed of the objects if a threshold is exceeded". The route is generated on the network by the A* algorithm, a modification of Dijkstra's shortest path algorithm. The speed is influenced by many factors, i.e. the max speed of the class of vehicles which the moving objects belongs to, the max speed of the route on which the object moves, traffic jams, and special weather conditions. However, the speed of a moving object is fixed for relatively large periods of time (for example, the model will not capture slowing down to avoid a pedestrian or a patch of ice on the road). This makes Brinkhoff’s algorithm inapplicable to comparing our update policies, since the number of updates of the deviation policy critically depends on fine speed variations.

In [7], Saglio and Moreira propose an interesting approach to model the moving objects behaviors, i.e. according to their present environment instead of randomly. They argue that "a generator producing completely random data will not fulfill the principle of representativeness", since the surrounding environment affects their behaviors, e.g. seeking food and avoiding danger. They choose as the objects fishing boats that move in the direction of the most attractive shoals of fish while trying to avoid storm areas. During the period of generation, the objects of one class may be attracted or repulsed by other classes of objects in the evolution. The drawbacks of this generator are: (1) Attracting and repulsing relationship is hard to define; (2) It is a limited application scenario; and (3) It is not applicable to road networks, which is the focus of our work.

The spatio-temporal database used in [14] is generated by CitySimulator, a scalable, three-dimensional model city that enables the creation of dynamic spatial data simulating the motion of up to 1 million objects. It can simulate any mobile objects moving about in a city, driving on the streets, walking on the sidewalks and even entering buildings, where they can go up and down the floors and stay on a floor and then leave for the streets. For the motion on the streets, an optional traffic flow model which causes traffic jams (or shock waves) is included in the simulator. However the users of CitySimulator have to provide some simulation control parameters, such as enter or exit probability (in a building), up/down probability (in a building), drift probability (on the streets), scatter probability (on intersections), etc. Furthermore, CitySimulator is designed for evaluation of database algorithms for indexing (see [15]). From the documentation it is not clear if it is possible to generate patterns of speed change for moving objects. Again, such patterns are critical in our case in order to compare update policies.

## 5 Conclusion and Future Work

In this paper we addressed tracking of moving objects, i.e. the continuous representation of their location in the database. We introduced the deviation database update policy for this purpose, and compared it with the straightforward policy currently used by commercial tracking systems, namely the distance policy. Using the distance policy the moving object updates the database every *x* distance units, and thus *x* is the maximum location uncertainty of a moving object. We showed that for any given location accuracy (i.e. location uncertainty threshold) the deviation policy outperforms the distance policy by up to 43% in the sense that it uses less updates for enabling the same accuracy. The advantage of the new policy depends on the location accuracy, and it decreases as the accuracy decreases, i.e. the uncertainty threshold increases.

We also introduced a method of generating realistic synthetic (maybe a better term is "semi-synthetic") spatio-temporal information, namely *pseudo trajectories* of moving objects. The method selects a random route, and superimposes on it speed patterns that were recorded during actual driving trips.

The deviation policy is aimed at vehicles moving on road networks. It predicts that following a location update the moving object will continue at the current speed until the end of the street. There are variants of this policy with which we have experimented, without great improvement. One variant is the hybrid policy. It uses the deviation policy on major streets and the distance policy on minor streets. The idea behind the hybrid was that a vehicle is likely to continue on a major street but not necessarily on a minor one. Actually, the policy used depended on both the category of street, and the location accuracy. The optimal policy for each (street category, uncertainty) pair was found experimentally; i.e. we constructed a matrix that gives the best policy for each pair, and defined the hybrid policy to behave according to the matrix. It turns out that the hybrid policy slightly improves over the deviation policy, but the change is relatively small, thus we did not report on it in the present paper. Anyway, one of the advantages of the hybrid policy is that it points in a direction for further improvement in the future.

In terms of future work, observe that some variations on the hybrid policy may improve its resource- consumption/accuracy tradeoff. For example, suppose that we introduce another class of streets, medium, in addition to major and minor. And suppose further that on this class of streets, the moving object is assumed to continue at the current speed until the next major intersection, i.e. intersection with a street in a higher category. Observe that this is a middle-ground between the assumption for a minor street (the moving object stops at the last location update) and a major street (the object continues until the end of the street). It is interesting to determine experimentally whether this variation improves on the policy analyzed in this paper. For this purpose, the experimental methodology can be the same as the one used in this paper.

Another direction for future work is a better speed-prediction for the deviation policy. More generally, how does the future speed of a moving object depend on its past speed behaviour? The pseudo trajectory generation algorithm will come in handy for answering this question.

Another direction of future work is to determine whether our results hold for trajectories generated in rural areas and in foreign countries.

## References

[1] Ouri Wolfson, A.Prasad Sistla, Sam Chamberlain, and Yelena Yesha, Updating and Querying Databases that Track Mobile Units. Special issue of the Distributed and Parallel Databases Journal on Mobile Data Management and Applications, 7(3), 1999.

[2] GDT: www.geographic.com/support/docs/D20_101.pdf

[3] Theodoridis Y., Silva J.R.O., Nascimento M.A., On the Generation of Spatiotemporal Datasets. SSD1999, July 20-23, 1999. LNCS 1651, Springer, pp. 147-164.

[4] Pfoser D., Theodoridis Y., Generating Semantics-Based Trajectories of Moving Objects. Intern. Workshop on Emerging Technologies for Geo-Based Applications, Ascona, 2000.

[5] Brinkhoff Thomas, Generating Network-Based Moving Objects. In Proc. of the 12th Inter. Conf. on Scientific and Statistical Database Management, July 26-28, 2000.

[6] Brinkhoff Thomas, A Framework for Generating Network-Based Moving Objects. Tech. Report of the IAPG, http://www2.fh-wilhelmshaven.de/oow/institute/iapg/personen/brinkhoff/paper/TBGenerator.pdf

[7] Jean-Marc Saglio and José Moreira, Oporto: A Realistic Scenario Generator for Moving Objects. In Proc. of 10th Inter. Workshop on Database and Expert Systems Applications, IEEE Computer Society, Florence, Italy, 1999, pp. 426-432, ISBN 0-7695-0281-4.

[8] Kollios, G., Gunopulos, D., Tsotras, V., Dellis, A., and Hadjieleftheriou, M. Indexing Animated Objects Using Spatiotemporal Access Methods. IEEE Transactions on Knowledge and Data Engineering, Sept/Oct 2001, Vol.13, No.5, p 758-777.

[9] Tao, Y., and Papadias, D. The MV3R-tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. VLDB 2001.

[10] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel Approaches to the Indexing of Moving Object Trajectories. VLDB'00, Cairo, Egypt, September 2000.

[11] G. Trajcevski, O. Wolfson, F. Zhang, and S. Chamberlian, The Geometry of Uncertainty in Moving Objects Databases. EDBT 2002.

[12] M. Vlachos, G. Kollios, and D. Gunopulos, Discovering Similar Multi-dimensional Trajectories. ICDE 2002.

[13] Trimble Navigation Limited. Differential GPS Sources: www.trimble.com/gps/dgps.html

[14] Jussi Myllymaki and James Kaufman, LOCUS: A Testbed for Dynamic Spatial Indexing. In IEEE Data Engineering Bulletin 25(2), p48-55, 2002.

[15] CitySimulator:http://alphaworks.ibm.com/tech/citysimulator

[16] Zhexuan Song, and Nick Roussopoulos, Hashing Moving Objects. MDM 2001.

[17] Kam-Yiu Lam, Ozgur Ulusoy, Tony S. H. Lee, Edward Chan and Guohui Li, Generating Location Updates for Processing of Location-Dependent Continuous Queries. DASFAA 2001, Hong Kong.

[18] Richard T. Snodgrass and Ilsoo Ahn, The Temporal Databases. IEEE Computer 19(9), p35-42, September, 1986.

[19] QUALCOMM Inc.: http://www.qualcomm.com

[20] At Road Inc.: http://www.atroad.com/

[21] Christian S. Jensen and Simonas Saltenis, Towards Increasingly Update Efficient Moving-Object Indexing. Bulletin of the IEEE on Data Engineering, 2002.