Online Trajectory Data Reduction using Connection-preserving Dead Reckoning

Ralph Lange Frank Dürr Kurt Rothermel
Institute of Parallel and
Distributed Systems
Universitätsstraße 38
70569 Stuttgart, Germany
<firstname.lastname>
@ipvs.uni-stuttgart.de

ABSTRACT

Moving objects databases (MODs) store objects' trajectories by spatiotemporal polylines that approximate the actual movements given by sequences of sensed positions. Determining such a polyline with as few vertices as possible under the constraint that it does not deviate by more than a certain accuracy bound ϵ from the sensed positions is an algorithmic problem known as trajectory reduction.

A specific challenge is *online trajectory reduction*, i.e. continuous reduction with position sensing in realtime. This particularly is required for moving objects with embedded position sensors whose movements are tracked and stored by a remote MOD.

In this paper, we present Connection-preserving Dead Reckoning (CDR), a new approach for online trajectory reduction. It outperforms the existing approaches by 30 to 50%. CDR requires the moving objects to temporally store some of the previously sensed positions. Although the storage consumption of CDR generally is small, it is not bounded. We therefore further present CDR$^{\mathrm{M}}$ whose storage allocation and execution time per position fix can be adjusted and limited. Even with very limited storage allocations of less than 1 kB CDR$^{\mathrm{M}}$ outperforms the existing approach by 20 to 40%.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial databases and GIS*

General Terms

Online trajectory reduction, dead reckoning, MOD

1. INTRODUCTION

A moving objects database (MOD) manages the trajectories of moving objects like vehicles, aircrafts, containers, and animals. It stores and indexes past, current, and possibly even future positions of a set of objects and processes spatiotemporal queries like retrieving all objects that were located inside a certain region during a certain time interval. MODs can be used for a multitude of applications and location-based services like toll collection, fleet management, and asset tracking.

A crucial issue in MODs is the efficient representation and storage of the objects' trajectories. A positioning sensor like a GPS receiver acquires an object's movement as a sequence of timestamped positions $s_1, s_2, s_3, \ldots$. Each timestamped position s_i has two attributes $\vec{p}$ and t, where $s_i.\vec{p}$ denotes the object's geographic position at time $s_i.t$. Thus, $\vec{p}$ is a point (position vector) in the Euclidean plane or space.

Storing every timestamped position reported by an object's positioning sensor in a MOD would consume much storage. For example, consider 100 000 objects whose positions are sensed once per second. Assuming that each position is represented by 3×8 byte this results in a data volume of about 6 TB per month.

In general, a MOD therefore only stores an approximation of an object's movement. The most common approach is representing the movement by a spatiotemporal polyline [1, 3, 4, 6]. That is, the movement is approximated by a sequence of spatiotemporal line sections $\overline{u_1\,u_2}, \overline{u_2\,u_3}, \overline{u_3\,u_4}, \ldots$ which define a piecewise linear, continuous function $\vec{u} : t \mapsto \mathbb{R}^d$ with $d = 2$ or 3 respectively. With this approximation the MOD only stores the timestamped position $u_1, u_2, u_3, \ldots$ and assumes linear movement between consecutive times $u_j.t$ and $u_{j+1}.t$.

Determining an efficient approximation such that each sensed position s_i does not deviate by more than an *accuracy bound* ϵ from the position given by $\vec{u}(t)$ is an algorithmic challenge known as *trajectory reduction* [1, 7]. More precisely the polyline $\vec{u}(t)$ shall consist of as few vertices u_j as possible under the constraint that

$$\forall i : |s_i.\vec{p} - \vec{u}(s_i.t)| \leq \epsilon \ .$$

In [1] Cao et al. present an approach for trajectory reduction using the Douglas-Peucker algorithm for line simplification. However, this approach allows for *offline* reduction only. That is, it only can reduce a *complete* (sub-)trajectory given by a *closed* sequence of sensed positions $\{s_1, \ldots, s_l\}$.

Such a scheme is not suited for a moving object with an embedded positioning sensor whose movement is being tracked and stored by a remote MOD: If the moving object

MobiQuitous 2008, July 21 – 25, 2008, Dublin, Ireland.

performs the offline reduction, then the MOD cannot track the object's current position in realtime since the trajectory data has to be transmitted in a batch-like fashion. Otherwise, if the MOD performs the reduction, then the moving object has to transmit every sensed position to the MOD, which causes a large communication overhead.

Therefore, MODs require online algorithms for trajectory reduction. That is, the moving object itself continuously reduces the sensed trajectory data in realtime and transmits it to the MOD so that the MOD always knows the object's past and current movement with guaranteed predefined accuracy bound ϵ.

Online reduction is closely related to *position update protocols* like linear dead reckoning (LDR) [5,8,9]. These protocols aim at minimizing the number of position updates from a moving object to a remote location management system that tracks the object's current position with a certain accuracy bound ϵ. However these protocols do not perform trajectory reduction since they generally approximate the object's movement by a discontinuous function in time. This particularly applies to LDR, cf. Section 2.

To the best of our knowledge the only existing approach for online trajectory reduction is given in [7]. It employs LDR with accuracy bound $\frac{1}{2}\epsilon$ ($\mathrm{LDR}_{\frac{1}{2}}$) to obtain a continuous reduced trajectory which approximates the actual movement within the accuracy bound ϵ.

In this paper, we present Connection-preserving Dead Reckoning (CDR), a new algorithm for online trajectory reduction. Though the algorithm's core also is based on LDR it is specifically tailored to trajectory reduction. Our evaluations show that CDR reduces position updates and trajectory size – i.e. the number of vertices u_j – by 30 to 50% compared to $\mathrm{LDR}_{\frac{1}{2}}$.

CDR requires the moving object to temporally store a set of past positions for deciding on the next update message. Although this set only contains positions that have been sensed after the last position update, its size generally is not bounded. Therefore, we also propose a space-bounded algorithm called $\mathrm{CDR}^{\mathrm{M}}$, which allows to trade off between the storage consumption at the moving object and the number of vertices of the reduced trajectory. Even with very limited storage allocations of less than 1 kB our algorithm still leads to trajectories that are 20 to 40% smaller compared to $\mathrm{LDR}_{\frac{1}{2}}$. In addition, we show that $\mathrm{CDR}^{\mathrm{M}}$ also provides a bound for the execution time per position fix.

The remainder of the paper is structured as follows: In Section 2 we discuss related work. In Section 3 we present our system model and important spatiotemporal functions for the following algorithms and analyses. In Section 4 we first propose a basic version of CDR and then discuss is mathematical optimization. In Section 5 we present $\mathrm{CDR}^{\mathrm{M}}$ with the underlying mathematics. We show the effectiveness of CDR and $\mathrm{CDR}^{\mathrm{M}}$ in extensive experiments with real trajectory data from different means of transportation in Section 6. The paper is concluded in Section 7 with a summary.

2. RELATED WORK

As already mentioned above, online trajectory reduction is closely related to offline trajectory reduction and position update protocols.

In [1], Cao et al. present an approach for offline trajectory reduction based on line simplification according to the Douglas-Peucker algorithm [2]. The algorithm starts with the spatiotemporal polyline given by the sequence of sensed positions $\{s_1, s_2, \ldots, s_l\}$ and then recursively removes positions from the sequence while the resulting reduced polyline $\vec{u}(t)$ and the original one deviate by less than a given accuracy bound ϵ. Thus, the vertices of the reduced trajectory are a subset of the sensed positions.

Position updates protocols aim at minimizing the wireless communication between a moving object and a location management system that tracks the object's current position. The most efficient protocol is dead reckoning [5, 8, 9]. Using the dead reckoning algorithm the object initially transmits its last sensed position and a prediction on its future movement to the location management system. As long as the object's movement satisfies the prediction with respect to a certain accuracy bound ϵ no update is required. Once the object's current position deviates from the predicted one by more than ϵ the object sends a new update message with a new prediction.

The most simple but nevertheless efficient and general applicable variant is linear dead reckoning (LDR) [5,9]. It uses a linear prediction given by the object's last sensed position and a velocity vector. The velocity vector usually is derived out of the last sensed positions.

LDR does not perform trajectory reduction since it describes the object's movement by a discontinuous function in time. More precisely, it represents the movement by a sequence of disconnected spatiotemporal line sections – the linear predictions – which do not compose a polyline as required for storage in a MOD. Figure 1 gives an accordant example: The small crosses denote the sensed positions. The solid arrows denote the linear predictions.

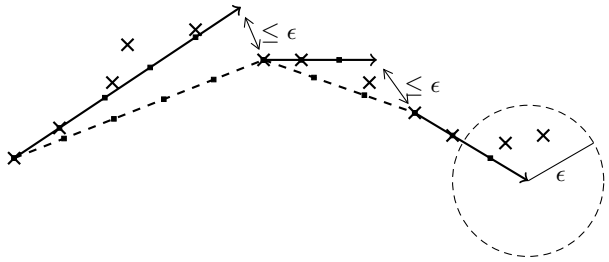

Figure 1: Linear predictions of LDR and reduced trajectory.

Nevertheless, the distance between the end point of such a line section and the start point of the subsequent one is bounded by ϵ.

In [7], Trajcevski et al. employ this property for online trajectory reduction. They analyze the spatiotemporal polyline given by the *origins* (start points) of the linear predictions of LDR and prove that it approximates the sensed movement by 2ϵ. Figure 1 denotes this polyline by a dashed line. Trajcevski et al. conclude that LDR allows for online trajectory reduction with accuracy bound ϵ as follows:

1. The moving object reports its position according to LDR with accuracy bound $\frac{1}{2}\epsilon$.
2. The MOD not only stores the current prediction but also the origins of all previous predictions as vertices of the reduced trajectory.

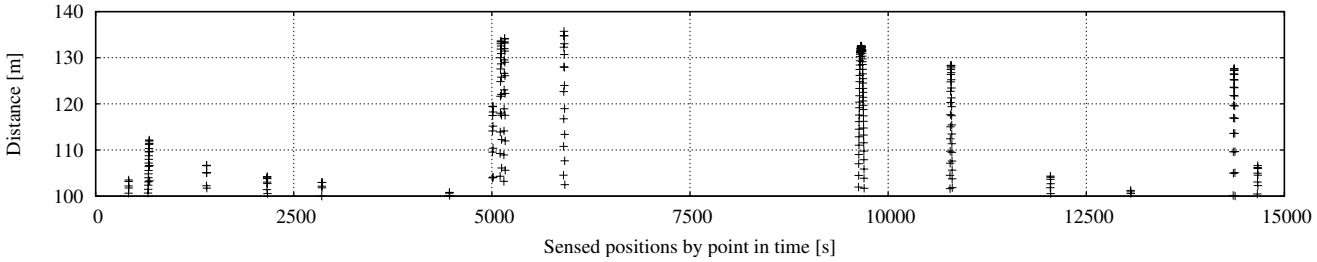

Figure 2: Violations of ϵ of a LDR-originated polyline during a 4 h car ride.

We refer to this approach as $\mathrm{LDR}_{\frac{1}{2}}$ in the following.

Our online algorithms CDR and $\mathrm{CDR}^{\mathrm{M}}$ presented in this paper are specifically tailored to trajectory reduction. They outperform the just mentioned approach by up to 50%.

3. SYSTEM MODEL AND DEFINITIONS

In this paper, we consider a moving object whose trajectory is being managed by a remote MOD, where the overall number of trajectories stored by the MOD is of no relevance here.

We assume that the moving object periodically senses its position by means of an embedded positioning sensor like a GPS receiver. The data record s_i represents the ith sensed position. It has two attributes $\vec{p}$ and t, where $s_i.\vec{p}$ denotes the object's geographic position at time $s_i.t$. The position vector $s_i.\vec{p}$ is a point in the Euclidean plane or space.

We further assume that the moving object is capable of storing and processing the sensed positions and that it has a wireless communication interface for transmitting position data and auxiliary information to the MOD.

For the sake of simplicity, we do neither consider spatial inaccuracies of the positioning sensor nor the latency for transmitting an update message to the MOD since they can be set off against ϵ. Clearly, the former increases the accuracy bound ϵ by the maximum inaccuracy of the positioning sensor, while the other adds the product of the maximum transmission latency and the object's maximum velocity.

The MOD stores a reduced trajectory of the object's movement as a spatiotemporal polyline with vertices $u_1, u_2, \ldots$, where a vertex u_j is a timestamped position with attributes $\vec{p}$ and t.

In the following, we consider three kinds of spatiotemporal functions $\vec{f} : t \mapsto \mathbb{R}^d$ with $d = 2$ or 3 respectively:

1. A *spatiotemporal line section* $\overline{u_j\, u_{j+1}}$ is a linear function on the domain $[u_j.t, u_{j+1}.t]$ given by linear interpolation:
$$\overline{u_j\, u_{j+1}} : t \mapsto \frac{(u_{j+1}.t - t)\, u_j.\vec{p} + (t - u_j.t)\, u_{j+1}.\vec{p}}{u_{j+1}.t - u_j.t}$$
2. The *spatiotemporal polyline* $\vec{u}(t)$ is a piecewise linear, continuous function $\vec{u} : t \mapsto \mathbb{R}^d$ defined by the spatiotemporal line sections $\overline{u_1\, u_2}, \overline{u_2\, u_3}, \overline{u_3\, u_4}, \ldots$.
3. A *linear prediction* $u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}$ is given by a timestamped position u_{P} – the prediction origin – and a velocity vector $\vec{v}_{\mathrm{P}} \in \mathbb{R}^d$ and defined for $t \geq u_{\mathrm{P}}.t$ as
$$u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}} : t \mapsto u_{\mathrm{P}}.\vec{p} + (t - u_{\mathrm{P}}.t)\, \vec{v}_{\mathrm{P}} \ .$$

In the following, we abbreviate the Euclidean distance between a geographic position $s_i.\vec{p}$ and the one for the corresponding point in time $s_i.t$ given by one of the above spatiotemporal functions as
$$\|s_i, \vec{f}(t)\| := |s_i.\vec{p} - \vec{f}(s_i.t)| \ .$$
For example, for a linear prediction it is
$$\|s_i, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| := |s_i.\vec{p} - u_{\mathrm{P}}.\vec{p} - (s_i.t - u_{\mathrm{P}}.t)\, \vec{v}_{\mathrm{P}}| \ .$$

4. CONNECTION-PRESERVING DEAD RECKONING

In this section we present Connection-preserving Dead Reckoning (CDR), a new algorithm for online trajectory reduction.

First, we analyze violations of the accuracy bound ϵ when using LDR for online trajectory reduction. Then, based on our observations, we introduce the basic idea of CDR and the resulting basic version of the algorithm. Subsequently, we discuss an optimization of CDR to reduce the storage consumption at the moving object.

4.1 Analysis of LDR-based Reduction

We first analyze LDR with accuracy bound ϵ for online trajectory reduction. LDR describes the object's movement by a sequence of disconnected spatiotemporal line sections, one for each position update. The polyline $\vec{u}(t)$ given by the start points of these sections is a reduced trajectory approximating the sensed positions by 2ϵ [7]. Thus, the distance $\|s_i, \vec{u}(t)\|$ between a sensed position s_i and the reduced trajectory may exceed ϵ as illustrated in Figure 3.

We argue that such violations of ϵ are rare and will seldomly reach 2ϵ:

1. A sensed position s_i can only deviate by more than ϵ from the corresponding line section $\overline{u_j\, u_{j+1}}$ if s_i and

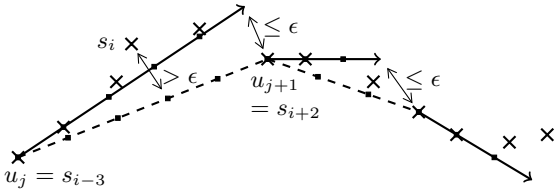

Figure 3: Violation of ϵ at s_i when using LDR for online trajectory reduction.

the sensed position u_{j+1} are located in opposite directions from the predicted movement. This does not hold for typical movement patterns like turning off or stopping.

2. $\|s_i, \overline{u_j\, u_{j+1}}\|$ only can be close to 2ϵ if s_i has been sensed immediately before u_{j+1} and $|s_i.\vec{p} - u_{j+1}.\vec{p}|$ also is about 2ϵ. Thus, the object has to move in a very fast and irregular fashion.

To support this argumentation, we analyzed such violations of ϵ for real trajectory data of a car ride of 14960 s $\approx$ 4 h. For that purpose, the car's movement was recorded once per second using a GPS receiver. See Section 6 for technical details on this experiment.

Executing LDR with $\epsilon = 100$ m on the recorded positions as if they were sensed one after another generates 488 position updates, where the last one indicates the end of the ride. Thus, the resulting reduced trajectory $\vec{u}(t)$ consists of 487 line sections given by 488 vertices. When measuring the distance between the sensed positions s_1 to s_{14960} and $\vec{u}(t)$ we observe 254 violations of ϵ. Figure 2 illustrates these violations with the distances $\|s_i, \vec{u}(t)\|$ against t. Each cross denotes a sensed position s_i with $\|s_i, \vec{u}(t)\| > \epsilon$.

The violations are not uniformly distributed over time but only appear at few line sections of $\vec{u}(t)$ – namely at 16 of the 487 line sections. Furthermore, the distances $\|s_i, \vec{u}(t)\|$ never reach 2ϵ but only about 136 m $\approx 1.4\epsilon$.

We conclude that the reduced trajectory obtained by LDR with accuracy bound ϵ approximates the sensed movement by ϵ at most points in time – i.e. violations as illustrated in Figure 3 occur rarely. Hence, using $\text{LDR}_{\frac{1}{2}}$ to prevent violations of ϵ generally is too strict. It generates unnecessary position updates and thus a reduced trajectory with an unnecessary large number of vertices.

Next, we show how to guarantee the same accuracy bound with higher reduction rates than $\text{LDR}_{\frac{1}{2}}$.

4.2 Basic Version of CDR

Our approach CDR is based on the observation that the moving object has all information for detecting the violations of the accuracy bound ϵ in realtime:

1. It knows the *current prediction* given by the *last updated position* (the prediction origin) and a *velocity vector*. In the following we denote the last updated position by u_P and the predicted velocity vector by $\vec{v}_\text{P}$.

2. It knows the *current sensed position*, denoted by s_C in the following.

3. Thus, it also knows the resulting line section $\overline{u_\text{P}\, s_\text{C}}$ of the reduced trajectory in the case of using s_C as origin of the next prediction.

4. It can store the *sensing history* since the last update message – i.e. the positions that have been sensed after $u_\text{P}.t$ – and check whether one of these positions deviates from $\overline{u_\text{P}\, s_\text{C}}$ by more than ϵ. In the following we refer to the sensing history as $\mathbb{S} := \{s_i : s_i.t > u_\text{P}.t\}$.

The basic idea of CDR is that the moving object not only sends a new position update like with LDR – i.e. if the current sensed position deviates from the predicted one by more than ϵ – but also if one of the sensed positions since the last update message deviates from $\overline{u_\text{P}\, s_\text{C}}$ by more than ϵ.

Algorithm 1 CDR – basic version

```
1:  s_C ← sense position                          ▷ Current sensed position.
2:  u_P ← s_C      ▷ Last updated position, i.e. prediction origin.
3:  v_P ← 0                                       ▷ Predicted velocity.
4:  send update message (u_P, v_P) to MOD
5:  𝕊 ← {}                          ▷ Sensing history since last update.
6:  s_L ← s_C                                     ▷ Last sensed position.
7:  while report movement do
8:      s_C ← sense position
9:      c_L ← ( ‖s_C, u_P ⊕ v_P‖ ≤ ε )            ▷ LDR condition.
10:     c_S ← true                                ▷ Section condition.
11:     for all s_i ∈ 𝕊 do
12:         c_S ← c_S ∧ ( ‖s_i, u_P s_C‖ ≤ ε )
13:     end for
14:     if ¬(c_L ∧ c_S) then
15:         u_P ← s_L
16:         v_P ← (s_C.p − s_L.p)/(s_C.t − s_L.t)
17:         send update message (u_P, v_P) to MOD
18:         remove all positions from 𝕊
19:     end if
20:     insert s_C into 𝕊
21:     s_L ← s_C
22: end while
23: send final update message (s_C) to MOD
```

Algorithm 1 shows the pseudo code of the algorithm executed by moving object. A crucial difference to LDR is that CDR maintains a dynamic array which stores the sensing history (line 5).

Initially the moving object transmits its current position and the zero vector as velocity prediction to the MOD. Then it executes the while loop (lines 7 to 22) as long as it wants to report its movement to the MOD.

Within each loop iteration it first senses its current position (line 8) and then checks for two conditions c_L and c_S whose violation cause an update message:

- *LDR condition* c_L*:* The current sensed position s_C must not deviate by more than ϵ from the predicted position for time $s_\text{C}.t$ (line 9).

- *Section condition* c_S*:* None of the sensed positions s_i since the last update is allowed to deviate by more than ϵ from the line section $\overline{u_\text{P}\, s_\text{C}}$ (lines 10 to 13).

If one of the conditions is violated, then the moving object sends a new position update and empties $\mathbb{S}$ (lines 14 to 19).

Note that with CDR the origin of a new linear prediction $(u_\text{P}, \vec{v}_\text{P})$ is not s_C but the *last sensed position* s_L. This guarantees that the resulting line section of the reduced trajectory always fulfills the section condition.

In Algorithm 1 we use one exemplary formula for predicting the moving object's future velocity $\vec{v}_\text{P}$, namely by means of the current and the last sensed position (line 16). It can be replaced by any other prediction function.

If the moving object wants to stop reporting its movement, then it sends a final update message with the current sensed position and terminates the algorithm (line 23).

The algorithm executed by the MOD is very simple: On receiving a new prediction $(u_\text{P}, \vec{v}_\text{P}) =: (u_n, \vec{v}_n)$ the MOD replaces the current prediction $(u_{n-1}, \vec{v}_{n-1})$ with it and extends $\vec{u}(t)$ by the line section $\overline{u_{n-1}\, u_n}$. Given a query for the moving object's position at time t' the MOD answers as follows:

- $t' \leq u_{\mathrm{P}}.t$ *:* The MOD calculates $\vec{u}(t')$ as described in Section 3 and returns the result to the query issuer.
- $t' > u_{\mathrm{P}}.t$ *:* It calculates the predicted position at time t' using $u_{\mathrm{P}}.\vec{p} + (t' - u_{\mathrm{P}}.t)\vec{v}_{\mathrm{P}}$ and returns the result to the query issuer.

If the MOD receives the final update message $(s_{\mathrm{C}}) =: (u_n)$ it removes the current prediction $(u_{n-1}, \vec{v}_{n-1})$ and completes $\vec{u}(t)$ by adding the line section $\overline{u_{n-1}\, u_n}$.

4.3 Optimized Algorithm

CDR differs from LDR regarding the storage requirements at the moving object. While LDR only stores the current prediction, the last sensed position and the current sensed position, the basic version of CDR stores the whole sensing history $\mathbb{S}$ since the last update. Theoretically, the size of $\mathbb{S}$ is unbounded. For the above-mentioned car ride and $\epsilon = 100\,\mathrm{m}$ $\mathbb{S}$ would contain up to 298 positions.

However, this problem can be alleviated. For every sensed position $s_i \in \mathbb{S}$ there exists a certain point in time from that on it cannot violate the section condition without s_{C} violating the LDR condition. If this time is reached before the violation of c_{L} or c_{S}, then s_i can be removed from $\mathbb{S}$, even *before* the next update message. This significantly reduces the storage consumption of CDR as well as the execution time per position fix. For example, for the above-mentioned car ride and $\epsilon = 100\,\mathrm{m}$ the maximum size of $\mathbb{S}$ reduces from 298 to 84 positions, cf. Section 6.3. The maximum execution time per position fix (here measured in processor ticks, cf. Section 6.4) reduces from 68470 to 22650 ticks.

To determine this point in time for a given $s_i \in \mathbb{S}$ we analyze the state of Algorithm 1 right after having sensed a new position s_{C} (line 8) with $\|s_{\mathrm{C}}, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| \leq \epsilon$. Thus, we assume that the current sensed position does not violate the LDR condition.

We consider the line section $\overline{u_{\mathrm{P}}\, s_{\mathrm{C}}}$ which is going to be the next line section $\overline{u_{n-1}\, u_n}$ of $\vec{u}(t)$ if c_{L} or c_{S} are violated in the subsequent iteration of the while loop.

Since $u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}$ and $\overline{u_{\mathrm{P}}\, s_{\mathrm{C}}}$ are linear functions in t with identical origin u_{P} we conclude that for $t = s_i.t$ they deviate by

$$\frac{s_i.t - u_{\mathrm{P}}.t}{s_{\mathrm{C}}.t - u_{\mathrm{P}}.t} \|s_{\mathrm{C}}, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| \,.$$

With this result and the triangle inequality illustrated in Figure 4 it follows:

$$\|s_i, \overline{u_{\mathrm{P}}\, s_{\mathrm{C}}}\| \leq \|s_i, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| + \frac{s_i.t - u_{\mathrm{P}}.t}{s_{\mathrm{C}}.t - u_{\mathrm{P}}.t} \|s_{\mathrm{C}}, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| \quad (1)$$

With our assumption $\|s_{\mathrm{C}}, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| \leq \epsilon$ we finally estimate $\|s_i, \overline{u_{\mathrm{P}}\, s_{\mathrm{C}}}\|$ by

$$\|s_i, \overline{u_{\mathrm{P}}\, s_{\mathrm{C}}}\| \leq \|s_i, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| + \frac{s_i.t - u_{\mathrm{P}}.t}{s_{\mathrm{C}}.t - u_{\mathrm{P}}.t} \epsilon \,.$$

Clearly, this estimate decreases over time, i.e. with increasing values of $s_{\mathrm{C}}.t$. We now derive the point in time from that on it falls below ϵ:

$$\epsilon \geq \|s_i, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| + \frac{s_i.t - u_{\mathrm{P}}.t}{s_{\mathrm{C}}.t - u_{\mathrm{P}}.t} \epsilon$$

$$\Leftrightarrow \frac{\epsilon - \|s_i, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\|}{s_i.t - u_{\mathrm{P}}.t} \geq \frac{\epsilon}{s_{\mathrm{C}}.t - u_{\mathrm{P}}.t} \quad (2)$$

$$\Leftrightarrow s_{\mathrm{C}}.t \geq \underbrace{\frac{s_i.t - u_{\mathrm{P}}.t}{\epsilon - \|s_i, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\|} \epsilon + u_{\mathrm{P}}.t}_{=:\, \delta(s_i)} \quad (3)$$

Algorithm 2 CDR

```
[...]
while report movement do
    s_C ← sense position
    while |S| > 0 ∧ s_C.t ≥ δ(peek(S)) do
        pop(S)                          ▷ Remove root of heap.
    end while
    c_L ← ( ‖s_C, u_P ⊕ v_P‖ ≤ ε )      ▷ LDR condition.
    [...]
end while
send final update message (s_C) to MOD
```

Thus, s_i cannot violate the section condition once $s_{\mathrm{C}}.t$ fulfills (3).

So far we assumed that s_{C} fulfills the LDR condition. In the general case it holds that s_i cannot violate the section condition without s_{C} violating the LDR condition once $s_{\mathrm{C}}.t$ fulfills (3).

Thus, CDR can remove s_i from $\mathbb{S}$ at that point in time without affecting its future decisions on a new position update.

For this purpose, CDR organizes $\mathbb{S}$ as a min-heap according to the right-hand side of (3), i.e. $\delta(s_i)$. After position sensing it first removes the root of $\mathbb{S}$ as long as this position fulfills (3). Algorithm 2 shows the corresponding additional pseudo code with respect to the basic version of CDR.

5. SPACE- AND TIME-BOUNDED CDR

With the optimization presented above, CDR tries to reduce the sensing history $\mathbb{S}$ after each position fix. Nevertheless, the storage consumption of CDR is not bounded.

In particular, for resource-constraint mobile devices like sensor nodes this property might be critical.

In the following we present the CDR$^{\mathrm{M}}$ algorithm whose storage consumption is bounded by a predefined parameter m. CDR$^{\mathrm{M}}$ guarantees that $|\mathbb{S}| \leq m$ at every point in time. This also limits the execution time per position fix.

CDR$^{\mathrm{M}}$ is based on the following idea: Besides a heap of fixed size m for storing $\mathbb{S}$, it maintains a floating-point variable $d_{\mathbb{S}}$ that provides aggregated information on all sensed positions that could not be stored in $\mathbb{S}$ due to the storage constraint. More precisely, $d_{\mathbb{S}}$ defines a time-dependent bound regarding $\|s_{\mathrm{C}}, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\|$. Each time $|\mathbb{S}|$ is going to exceed m, the CDR$^{\mathrm{M}}$ algorithm removes a sensed position from $\mathbb{S}$ and updates $d_{\mathbb{S}}$ accordingly.

If $\|s_{\mathrm{C}}, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\|$ is below the bound defined by $d_{\mathbb{S}}$, then none of the sensed positions that could not be stored in $\mathbb{S}$

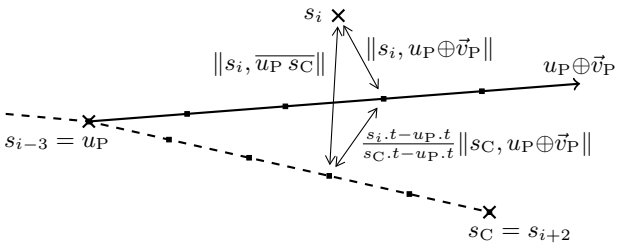

Figure 4: Triangle inequality regarding s_i.

violates the section condition for the current sensed position.

Condition c_{S} is split into two sub-conditions c_{Sa} and c_{Sb} accordingly. While c_{Sa} is evaluated on the sensed positions currently stored in $\mathbb{S}$ just as with CDR, c_{Sb} is evaluated on $d_{\mathbb{S}}$.

We now give the mathematical basis for $d_{\mathbb{S}}$ and derive the inequation for c_{Sb}.

First, we reconsider the triangle inequality (1) given in Section 4.3:

$$\|s_i, \overline{u_{\mathrm{P}}\, s_{\mathrm{C}}}\| \leq \|s_i, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| + \frac{s_i.t - u_{\mathrm{P}}.t}{s_{\mathrm{C}}.t - u_{\mathrm{P}}.t} \|s_{\mathrm{C}}, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\|$$

From it we conclude that

$$\|s_i, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| + \frac{s_i.t - u_{\mathrm{P}}.t}{s_{\mathrm{C}}.t - u_{\mathrm{P}}.t} \|s_{\mathrm{C}}, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| \leq \epsilon \ ,$$

which can be rewritten as

$$\|s_{\mathrm{C}}, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| \leq \underbrace{\frac{\epsilon - \|s_i, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\|}{s_i.t - u_{\mathrm{P}}.t}}_{=:\, \varphi(s_i)} (s_{\mathrm{C}}.t - u_{\mathrm{P}}.t) \ ,$$

implies $\|s_i, \overline{u_{\mathrm{P}}\, s_{\mathrm{C}}}\| \leq \epsilon$.

In other words,

$$\|s_{\mathrm{C}}, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| \leq \varphi(s_i) \cdot (s_{\mathrm{C}}.t - u_{\mathrm{P}}.t)$$

implies that s_i does not violate the section condition.

CDR$^{\mathrm{M}}$ applies this result as follows:

1. In the variable $d_{\mathbb{S}}$ it stores the minimum $\varphi(s_r)$ of all sensed positions s_r it has removed from $\mathbb{S}$ due to the storage constraint.
2. It uses $\|s_{\mathrm{C}}, u_{\mathrm{P}} \oplus \vec{v}_{\mathrm{P}}\| \leq d_{\mathbb{S}} \cdot (s_{\mathrm{C}}.t - u_{\mathrm{P}}.t)$ as condition c_{Sb}.

Therefore, as long as c_{Sb} is fulfilled, each removed position s_r fulfills the section condition. Thus, as long as $c_{\mathrm{Sa}} \wedge c_{\mathrm{Sb}}$ is fulfilled, every sensed position since the last position update fulfills the section condition.

After an update CDR$^{\mathrm{M}}$ empties $\mathbb{S}$, just as CDR, and resets $d_{\mathbb{S}}$ to ∞.

A crucial question is, which sensed position to remove from $\mathbb{S}$ once $|\mathbb{S}|$ is going to exceed m. Clearly, for small values of $d_{\mathbb{S}}$, the current sensed position s_{C} violates c_{Sb} more likely. Therefore, CDR$^{\mathrm{M}}$ always removes the $s_i \in \mathbb{S}$ with maximum $\varphi(s_i)$. For this purpose it stores $\mathbb{S}$ as a max-heap according to $\varphi(s_i)$.

Since $d_{\mathbb{S}}$ aggregates all previously sensed position with $\varphi(s_r) \geq d_{\mathbb{S}}$ the current sensed position s_{C} need not be added to $\mathbb{S}$ if $\varphi(s_{\mathrm{C}}) \geq d_{\mathbb{S}}$. Hence, the following invariant always holds for $\mathbb{S}$:

$$\forall s_i \in \mathbb{S} : \varphi(s_i) \leq d_{\mathbb{S}} \ .$$

For this reason, CDR$^{\mathrm{M}}$ can directly assign $\varphi(s_r)$ to $d_{\mathbb{S}}$ when removing the root s_r from $\mathbb{S}$. It does not need to explicitly determine the minimum of $\varphi(s_r)$ and $d_{\mathbb{S}}$.

Note that the max-heap order by $\varphi(s_i)$ of CDR$^{\mathrm{M}}$ is identical to the min-heap order by $\delta(s_i)$ of CDR since

$$\delta(s_i) = u_{\mathrm{P}}.t + \epsilon/\varphi(s_i) \ .$$

This can be seen by comparing the inequations (2) and (3) in Section 4.3. Moreover, CDR's condition $s_{\mathrm{C}}.t \geq \delta(\mathrm{peek}(\mathbb{S}))$ for removing a s_i from $\mathbb{S}$ can be rewritten as

$$\varphi(\mathrm{peek}(\mathbb{S})) \cdot (s_{\mathrm{C}}.t - u_{\mathrm{P}}.t) \geq \epsilon \ .$$

Algorithm 3 CDRM with parameter m

```
 1: s_C ← sense position
 2: u_P ← s_C
 3: v_P ← 0
 4: send update message (u_P, v_P) to MOD
 5: 𝕊 ← {}                                   ▷ Heap with size m.
 6: d_𝕊 ← ∞                                  ▷ Indicates empty aggregation.
 7: s_L ← s_C
 8: while report movement do
 9:     s_C ← sense position
10:     while |𝕊| > 0 ∧ φ(peek(𝕊)) · (s_C.t − u_P.t) ≥ ε do
11:         pop(𝕊)                           ▷ Remove root of heap.
12:     end while
13:     c_L ← ( ‖s_C, u_P ⊕ v_P‖ ≤ ε )
14:     c_Sa ← true
15:     for all s_i ∈ 𝕊 do
16:         c_Sa ← c_Sa ∧ ( ‖s_i, u_P s_C‖ ≤ ε )
17:     end for
18:     c_Sb ← ( ‖s_C, u_P ⊕ v_P‖ ≤ d_𝕊 · (s_C.t − u_P.t) )
19:     if ¬(c_L ∧ c_Sa ∧ c_Sb) then
20:         u_P ← s_L
21:         v_P ← (s_C.p − s_L.p)/(s_C.t − s_L.t)
22:         send update message (u_P, v_P) to MOD
23:         remove all positions from 𝕊
24:         d_𝕊 ← ∞                          ▷ Reset the bound.
25:     end if
26:     if φ(s_C) < d_𝕊 ∧ |𝕊| = m then
27:         d_𝕊 ← φ(pop(𝕊))   ▷ Remove and aggregate the root.
28:     end if
29:     if φ(s_C) < d_𝕊 then
30:         insert s_C into 𝕊
31:     end if
32:     s_L ← s_C
33: end while
34: send final update message (s_C) to MOD
```

Thus, from an algorithmic perspective CDR$^{\mathrm{M}}$ is an extension of CDR. Algorithm 3 gives the pseudo code of CDR$^{\mathrm{M}}$. The additional statements compared to CDR are the following ones:

- *Lines 6 and 24:* Initialize or rather reset $d_{\mathbb{S}}$.
- *Lines 26 to 28:* If $|\mathbb{S}| = m$ and the current sensed position has to be added to $\mathbb{S}$ then remove the sensed position with maximum $\varphi(s_i)$ from $\mathbb{S}$ and aggregate it in $d_{\mathbb{S}}$.
- *Lines 29 to 31:* Insert the current sensed position s_{C} into the heap $\mathbb{S}$, if and only if $\varphi(s_{\mathrm{C}}) < d_{\mathbb{S}}$.

5.1 Execution Time per Position Fix

The execution time per position fix of CDR$^{\mathrm{M}}$ – i.e. lines 10 to 32 in Algorithm 3 – is dominated by two inner loops: (1.) Removing sensed positions from $\mathbb{S}$ that cannot any more violate the section condition without s_{C} violating the LDR condition and (2.) checking the validity of condition c_{Sa}. All other statements either have a constant execution time or their execution time logarithmically depends on m like the pop operation on the heap $\mathbb{S}$.

The execution time for checking the validity of c_{Sa} is a linear function in m, thus belongs to $\mathcal{O}(m)$.

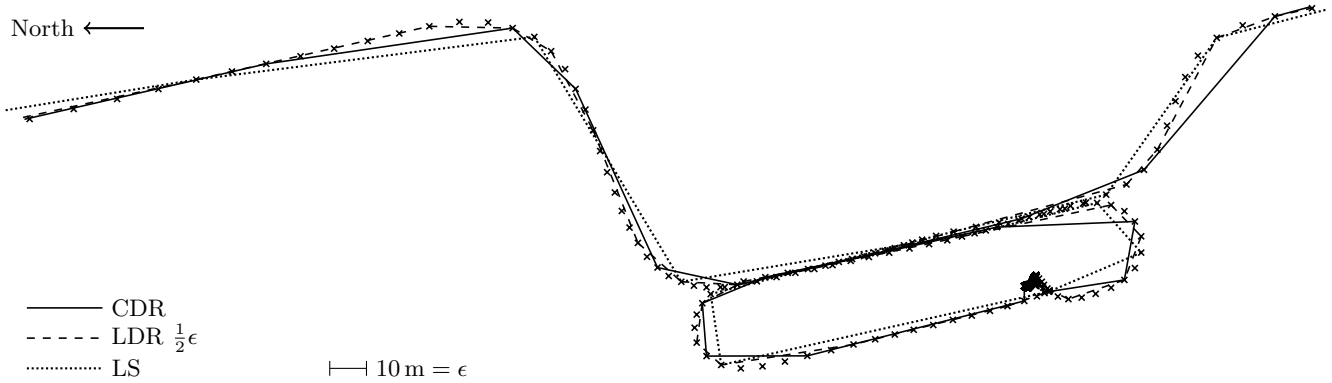

Figure 5: Small clipping of the recorded car ride and the reduced trajectories.

Yet, the removal of positions from $\mathbb{S}$ according to the optimization of CDR is more complex: Under normal circumstances only very few positions can be removed from $\mathbb{S}$. Often the root of the heap $\mathbb{S}$ cannot be removed at all and the next position fix has to be awaited.

However, in rare cases there may exist several sensed positions with equal or similar values $\varphi(s_i)$ that are removed from $\mathbb{S}$ all together at a certain point in time.

By analyzing

$$\delta(s_i) := \frac{\epsilon - \|s_i, u_\mathrm{P} \oplus \vec{v}_\mathrm{P}\|}{s_i.t - u_\mathrm{P}.t}$$

one can see that this particularly may occur if the object first deviates far off the predicted movement and then linearly returns to it. The reason is, that during the linear movement back to predicted one the numerator of $\delta(s_i)$ as well as the denominator linearly increase.

Since the time for removing the root of a heap is a function in $\mathcal{O}(\log(m))$ the worst case execution time for the removal of positions from $\mathbb{S}$ according to the optimization of CDR is in

$$\begin{aligned}\mathcal{O}(\log(m) + \log(m-1) + \ldots + 1) &= \mathcal{O}(\log(m!)) \\ &= \mathcal{O}(m \log(m)) \ .\end{aligned}$$

Thus, the execution time per position fix of CDR^M is bounded by a function in $\mathcal{O}(m \log(m))$.

6. EVALUATION

We evaluated CDR and CDR^M with real GPS trajectories from a car ride, a bicycle tour and a walk in a small town.

In the following, we first describe the evaluation setup followed by the results on data reduction, storage consumption and execution time per position fix for the trajectory of the car ride. Then, we discuss the results for the other trajectories.

6.1 Evaluation Setup

In order to evaluate CDR and CDR^M with realistic data, we first recorded three trips by different means of transportation in Southern Germany:

1. *Car ride:* The ride lasted 14960 s ≈ 4 h and covered a distance of 404 km.
2. *Bicycle tour:* The tour lasted 2934 s ≈ 50 min and covered a distance of about 17 km.
3. *Walk in town:* The walk lasted 4429 s ≈ 70 min and covered a distance of 4.5 km.

For recording the trips we used a PDA with an embedded Sirf III GPS receiver in continuous mode and stored the geographic coordinates given by the GPS receiver once per second on a SD memory card.

In a second step we projected the geographic coordinates on the Euclidean plane. Hence, the recorded trajectory of the car ride consists of 14960 timestamped positions s_1 to s_{14960}, where $s_i.\vec{p}$ is located inside a rectangle with side lengths 207.6 km × 249.6 km. The recorded trajectory of the bicycle tour consists of 2934 positions located inside a rectangle with side lengths 7.2 km × 11.7 km and the recorded trajectory of the walk consists of 4429 positions located inside a rectangle with side lengths 0.7 km × 1.1 km.

We implemented the online algorithms $\mathrm{LDR}_{\frac{1}{2}}$ – i.e. LDR with accuracy bound $\frac{1}{2}\epsilon$ according to [7] –, CDR, and CDR^M as well as the offline line simplification algorithm (LS) according to [1] in the C programming language as a Win32 application.

For each recorded trajectory and different values of ϵ, we then simulated the online execution of $\mathrm{LDR}_{\frac{1}{2}}$, CDR and CDR^M by sequentially feeding the algorithms with the recorded positions. In doing so we measured the number of vertices of the resulting reduced trajectories, the storage consumption and the execution time per sensed position for each approach. Note that we executed CDR^M with two different parametrizations, namely $m = 5$ and $m = 20$.

Furthermore, we applied LS to the three recorded trajectories and measured the number of vertices of the resulting reduced trajectories depending on ϵ. Note again, that our goal is an online algorithm rather than an offline algorithm. In our evaluation LS is used as a reference only.

All these more than 300 experiments were performed on a Lenovo Thinkpad T61 with Intel Core2 Duo CPU at 2.0 GHz and 2 GB of RAM running Microsoft Windows Vista.

In the following, we first give the results on reduction efficiency, storage consumption, and execution time per position fix for the recorded car ride. Then, we discuss the results for the bicycle tour and the walk in town.

6.2 Reduction Efficiency

The reduction efficiency of a trajectory reduction algorithm is measured by the reduction ratio r, which is the number

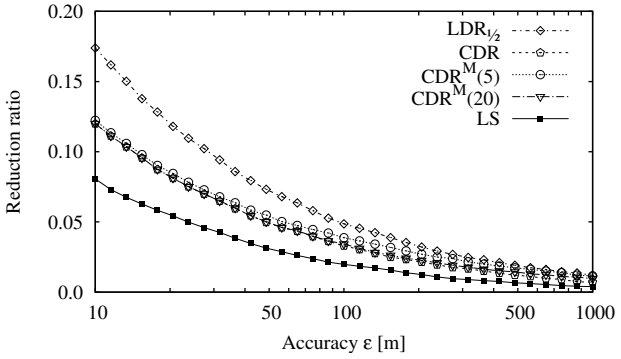

Figure 6: Reduction ratios (car ride).

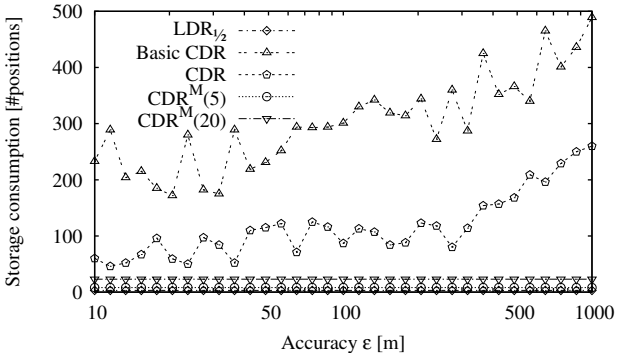

Figure 8: Storage consumption (car ride).

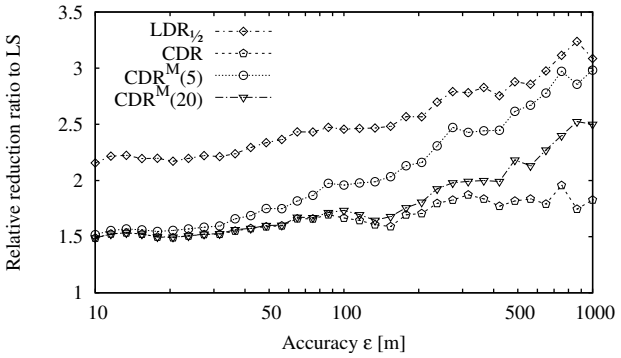

Figure 7: Ratios relative to LS (car ride).

of vertices n of the reduced trajectory $\vec{u}(t)$ divided by the number of sensed positions. Thus, for the trajectory of the car ride it is

$$r := \frac{|\{u_1, \ldots, u_n\}|}{|\{s_1, \ldots, s_{14960}\}|} = \frac{n}{14960} \ .$$

The smaller r the more efficient is the reduction by the respective approach.

Figure 6 gives the reduction ratios for the online algorithms $\mathrm{LDR}_{\frac{1}{2}}$, CDR, and $\mathrm{CDR}^{\mathrm{M}}$ depending on ϵ as well as for the offline approach LS.

For $\epsilon = 10\,\mathrm{m}$ $\mathrm{LDR}_{\frac{1}{2}}$ generates a reduced trajectory with $n = 2601$ vertices, i.e. $r = 0.17$. The trajectories generated by CDR and $\mathrm{CDR}^{\mathrm{M}}$ have 1794 to 1830 vertices, i.e. $r \approx 0.12$. Thus, CDR and $\mathrm{CDR}^{\mathrm{M}}$ outperform $\mathrm{LDR}_{\frac{1}{2}}$ by more than 30%.

With $n = 1206$ vertices, LS yields in $r = 0.08$ and thus less than CDR and $\mathrm{CDR}^{\mathrm{M}}$. This was to be expected since LS is an offline reduction approach and thus is not forced to process the sensed positions one after another but processes them altogether at the same time.

Of course, r decreases for each of the four approaches with increasing ϵ. Yet, the relative ratios between the approaches also vary. In particular, the relative savings of CDR compared to $\mathrm{LDR}_{\frac{1}{2}}$ increase from 30 to 40% for ϵ towards 1000 m.

Figure 7 shows the reduction ratios of the online reduction approaches relative to LS. For increasing ϵ all relative ratios increase since for large values of ϵ more sensed positions can be dropped and LS thus can better exploit its global view on the whole sequence of sensed positions.

The relative ratios of $\mathrm{CDR}^{\mathrm{M}}$ with $m = 5$ and 20 both converge from the ones of CDR to the ones of $\mathrm{LDR}_{\frac{1}{2}}$. The reason is the following: With increasing ϵ CDR generally stores more and more sensed positions in $\mathbb{S}$. $\mathrm{CDR}^{\mathrm{M}}$ accordingly aggregates more positions $d_{\mathbb{S}}$. Yet, the bound $d_{\mathbb{S}}$ generally is lower the more positions it aggregates. Hence, condition c_{Sb} of $\mathrm{CDR}^{\mathrm{M}}$ and thus $c_{\mathrm{Sa}} \wedge c_{\mathrm{Sb}}$ is violated more likely than condition c_{S} of CDR.

The actual behavior of $\mathrm{CDR}^{\mathrm{M}}$ highly depends on maximum size of $\mathbb{S}$ given by the parameter m. This can be well seen from ratios for $m = 5$ and $m = 20$ given Figure 7. For $m = 20$ and $\epsilon \leq 50\,\mathrm{m}$ the $\mathrm{CDR}^{\mathrm{M}}$ algorithm achieves almost the same reduction ratios than CDR. For larger values of ϵ $\mathrm{CDR}^{\mathrm{M}}$ performs slightly worse but sill outperforms $\mathrm{LDR}_{\frac{1}{2}}$ by at least 20%. This is notable because assuming that one position record takes 3×8 byte, $\mathrm{CDR}^{\mathrm{M}}$ with $m = 20$ allocates less than 1 kB storage at the moving object.

Figure 5 gives a small clipping of the reduced trajectories obtained by $\mathrm{LDR}_{\frac{1}{2}}$, CDR, and LS for $\epsilon = 10\,\mathrm{m}$ to point out the differences between these algorithms. The clipping shows a stop at a rest area on a motorway.

Obviously, the reduced trajectory obtained by $\mathrm{LDR}_{\frac{1}{2}}$ approximates the sensed positions much closer than with ϵ. This reinforces the analysis given in Section 4.1.

At first glance the reduced trajectories of CDR and LS seem very similar to each other. Yet, in particular with curves LS shows better results due its global view on the whole trajectory: While CDR has to start a new line section using the last sensed position as soon as c_{L} or c_{S} is violated, the offline approach LS can select an appropriate position to approximate the curve with as few vertices as possible. Therefore, CDR mostly requires three line sections to approximate a curve in the given clipping, while LS mostly requires two line sections only.

6.3 Storage Consumption

Next, we analyze the storage consumption of the three online reduction algorithms. Therefore, we describe an algorithm's storage consumption by the maximum number of positions it stores during our experiments.

LDR always stores three positions, namely the last updated position u_{P}, the last sensed position s_{L} and the cur-

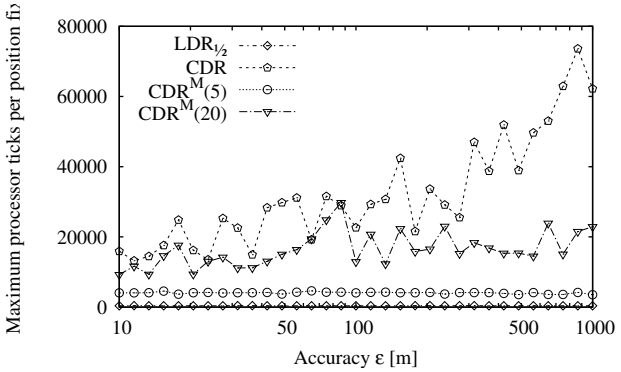

Figure 9: Maximum processor ticks (car ride).

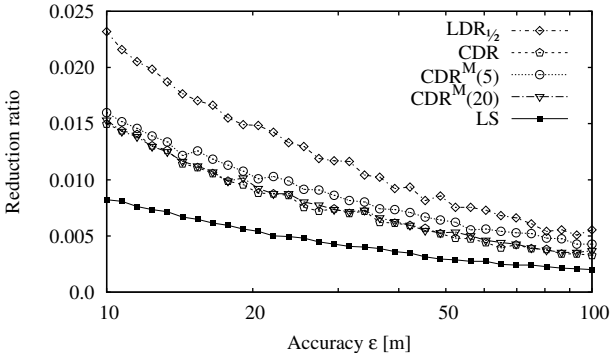

Figure 10: Reduction ratios (bicycle tour).

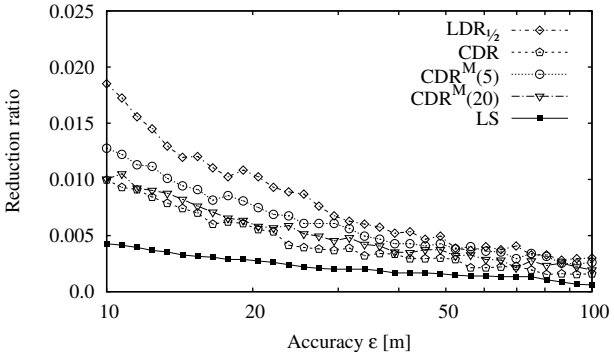

Figure 11: Reduction ratios (walk in town).

rent sensed position s_C.

The storage allocation of CDR^M is $m+3$, thus 8 or 23 in our experiments. Actually, CDR^M also fully consumed the allocated storage in all our experiments as shown in Figure 8.

The storage consumption of CDR generally is not bounded. This corresponds with our measurements. Figure 8 clearly shows that the storage consumption of CDR generally increases with larger values of ϵ in spite of the large deviations between measurements for similar values of ϵ.

Note that for $\epsilon \leq 50\,\text{m}$ the storage consumption of CDR is up to five times larger than the storage consumption of CDR^M with $m = 20$ although there is no noticeable difference in the reduction ratios of both approaches, as explained above.

To prove the effectiveness of removing positions from $\mathbb{S}$ according to the optimization of CDR, we also measured the maximum number of positions being stored by the basic version of CDR. In our experiments the storage consumption of the basic version always exceeds the storage consumption of CDR by 60 to 530%.

6.4 Execution Time per Position Fix

We now analyze the maximum execution time per position fix for each of the online reduction approaches depending on the accuracy bound ϵ.

To be independent of the processor speed we measure the execution time in processor ticks. For that purpose, we read out the processor time stamp counter before and after processing a sensed position for each of the online reduction algorithms.

A critical issue in measuring processor ticks are interrupts of the process under test by the scheduler since an interrupt generates an invalid measurement. Therefore, to minimize the number of interrupts, we executed our implementation of LDR$_{\frac{1}{2}}$, CDR, and CDR^M with realtime priority on one core of the computer's dual core processor. Furthermore, we repeated each experiment ten times and filtered out the remaining invalid measurements by using the median of the ten measurements for each recorded position.

In case of LDR$_{\frac{1}{2}}$ the maximum tick count per position fix always is between 370 and 420 ticks. This corresponds with the fact that LDR constantly maintains three positions independent of ϵ.

With CDR the maximum tick count varies between 13230 and 73610 as shown in Figure 9. The measurements well go with the storage consumption of CDR shown in Figure 8. This provides evidence that CDR's maximum execution time per position fix directly depends on the maximum size of $\mathbb{S}$.

In case of CDR^M and $m = 5$ the maximum tick counts slightly vary between 3530 and 4650 as shown in Figure 9. Yet, for $m = 20$ they vary much more – between 9250 and 29770 ticks. This is surprising at first glance since $|\mathbb{S}|$ always reached 20 in our experiments.

The reason is the following: As explained in Section 5.1, the execution time for a given s_C highly depends on the number of sensed positions $s_i \in \mathbb{S}$ that cannot violate the section condition any more without s_C violating the LDR condition and thus are removed from $\mathbb{S}$.

For $m = 5$ this number can come close to m easily. Yet, for $m = 20$ this is very unlikely since it would require that a large number of $s_i \in \mathbb{S}$ have equal or similar values $\varphi(s_i)$. Therefore, the larger m the higher the variance of the execution times per position fix and thus also the maximum execution time for a given sequence of sensed positions. This effect even is increased by the fact that insert and pop operations on $\mathbb{S}$ take longer the larger m.

Nevertheless, Figure 9 shows for CDR^M that the maximum tick counts do not increase for increasing ϵ.

6.5 Results for Other Trajectories

We now discuss the results for the two other recorded trajectories, i.e. the bicycle tour and the walk in town.

First of all, in conclusion, it can be said that the results

for those two trajectories are very similar to the previous ones.

- *Storage consumption:* With maximum values of 155 (bicycle tour) and 453 positions (walk in town) the storage consumption of CDR is in the same order of magnitude than for the recorded car ride.

- *Execution time per position fix:* With maximum values of 46180 processor ticks per position fix (bicycle tour) and 108230 ticks (walk in town) the execution times of CDR also can be compared to the ones for the car ride.
  For CDR^{M} with $m = 20$ the maximum values of 27680 (bicycle tour) and 19620 ticks (walk in town) even are below the maximum value of 29770 ticks for the car ride. The same applies to CDR^{M} with $m = 5$.

- *Reduction efficiency:* The reduction ratios differ from the ones given above due to the different velocity magnitudes of the three means of transportation. Yet, the relative differences between the ratios are very similar to the ones given above, as explained in the following.

Figure 10 gives the reduction ratios for $\text{LDR}_{\frac{1}{2}}$, CDR, CDR^{M}, and LS depending on ϵ for the recorded bicycle tour. Note that ϵ ranges from 10 to 100 m only, since for larger values of ϵ the number of vertices of the reduced trajectory is too small for being meaningful.

For $\epsilon = 10\,\text{m}$ $\text{LDR}_{\frac{1}{2}}$ generates a reduced trajectory with $n = 347$ vertices, i.e. $r = 0.12$. The trajectories generated by CDR and CDR^{M} have 224 to 239 vertices, i.e. $r \approx 0.08$. Thus, CDR and CDR^{M} outperform $\text{LDR}_{\frac{1}{2}}$ by more than 30%, like for the car ride trajectory.

For other values of ϵ the relative savings of CDR compared to $\text{LDR}_{\frac{1}{2}}$ range from 30 to 42% similar to the savings for the car ride.

The relative savings of CDR^{M} with $m = 20$ compared to $\text{LDR}_{\frac{1}{2}}$ range from 30 to 37%. Thus, they are even greater than for the car ride.

With the recorded walk in town the reduction ratios given in Figure 11 are more scattered than the previous given ones. The reason is, that the recorded walk comprises numerous short rest periods and thus is not a continuous, smooth movement like the recorded car ride or the bicycle tour.

The savings of CDR and CDR^{M} compared to $\text{LDR}_{\frac{1}{2}}$ are even greater than the previous mentioned ones: In case of CDR they range from 35 to 50% with one outlier of 26% and three outliers of about 55%. For CDR^{M} with $m = 20$ they range from 20 to 40% except for three outliers of about 15% and another three outliers of about 45%.

We conclude, that CDR generally outperforms online trajectory reduction using $\text{LDR}_{\frac{1}{2}}$ by 30 to 50% and that CDR^{M} with a storage allocation of less then 1 kB generally outperforms $\text{LDR}_{\frac{1}{2}}$ by 20 to 40%.

7. CONCLUSIONS

In this paper we studied online trajectory reduction for efficient storage of moving objects' trajectories in MODs. This kind of trajectory reduction particularly is required for objects with embedded position sensors whose movements are tracked and stored by a remote MOD.

Therefore, we contributed CDR and CDR^{M}, two new approaches for online trajectory reduction. We presented both algorithms in detail and discussed the underlying mathematics. We also explained that CDR achieves a higher reduction while CDR^{M} features a limited, adjustable storage allocation and execution time per position fix.

That followed we gave evaluation results from more than 300 experiments with prototype implementations of CDR and CDR^{M} using real trajectory data obtained by GPS. Our evaluation shows that CDR outperforms the existing approaches by 30 to 50%. CDR^{M} shows similar results: It outperforms the existing approaches by 20 to 40%, even with very limited storage allocations of less than 1 kB.

8. ACKNOWLEDGMENTS

The work described in this paper was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center (SFB) 627.

9. REFERENCES

[1] H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal*, 15(3):211–228, Sept. 2006.

[2] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10(2):112–122, Dec. 1973.

[3] R. H. Güting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann Publishers, San Francisco, California, USA, 2005.

[4] J. A. C. Lema, L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. Algorithms for moving objects databases. *The Computer Journal*, 46(6):680–712, 2003.

[5] A. Leonhardi and K. Rothermel. A comparison of protocols for updating location information. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 4(4):355–367, Oct. 2001.

[6] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches to the indexing of moving object trajectories. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB 2000)*, pages 395–406, Cairo, Egypt, Sept. 2000.

[7] G. Trajcevski, H. Cao, P. Scheuermann, O. Wolfson, and D. Vaccaro. Online data reduction and the quality of history in moving objects databases. In *Proceedings of the 5th ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE 2006)*, Chicago, Illinois, USA, June 2006.

[8] A. Čivilis, C. S. Jensen, and S. Pakalnis. Techniques for efficient road-network-based tracking of moving objects. *IEEE Transactions on Knowledge and Data Engineering*, 17(5):698–712, May 2005.

[9] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–287, July 1999.