This is the author's version of the work. The final publication is available at www.springerlink.com. It was published in *The VLDB Journal*, Volume 20, Number 5 (2011), 641–642. http://link.springer.com/article/10.1007/s00778-011-0237-7

Efficient Real-Time Trajectory Tracking

Ralph Lange · Frank Dürr · Kurt Rothermel

Abstract Moving objects databases (MOD) manage trajectory information of vehicles, animals, and other mobile objects. A crucial problem is how to efficiently track an object's trajectory in real-time, in particular if the trajectory data is sensed at the mobile object and thus has to be communicated over a wireless network.

We propose a family of tracking protocols that allow trading the communication cost and the amount of trajectory data stored at a MOD off against the spatial accuracy. With each of these protocols, the MOD manages a simplified trajectory that does not deviate by more than a certain accuracy bound from the actual movement. Moreover, the different protocols enable several trade-offs between computational costs, communication cost and the reduction of the trajectory data: *Connection-Preserving Dead Reckoning* (CDR) minimizes the communication cost using dead reckoning, a technique originally designed for tracking an object's current position. *Generic Remote Trajectory Simplification* (GRTS) further separates between tracking of the current position and simplification of the past trajectory and can be realized with different line simplification algorithms. For both protocols, we discuss how to bound the space consumption and computing time at the moving object and thereby present an effective compression technique to optimize the reduction performance of real-time line simplification in general.

Our evaluations with hundreds of real GPS traces show that a realization of GRTS with a simple simplification heuristic reaches 85 to 90% of the best possible reduction rate, given by retrospective offline simplification. A realization with the optimal line simplification algorithm by Imai and Iri even reaches more than 97% of the best possible reduction rate.

Parts of this article appeared as "Online Trajectory Data Reduction using Connection-preserving Dead Reckoning" in the Proceedings of the 5th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous '08) [12] and as "Remote Real-Time Trajectory Simplification" in the Proceedings of the 7th IEEE International Conference on Pervasive Computing and Communications (PerCom '09) [15].

Ralph Lange · Frank Dürr · Kurt Rothermel
Universität Stuttgart
Institute of Parallel and Distributed Systems (IPVS)
Universitätsstraße 38, 70569 Stuttgart, Germany
E-mail: <*firstname.lastname*>@ipvs.uni-stuttgart.de

1 Introduction

Moving objects databases (MODs) have been proposed for managing trajectories of mobile objects like vehicles, containers, aircrafts, and animals. They store and index the objects' geographic positions over time and process spatiotemporal queries such as retrieving all objects that were located inside a certain region during a certain time interval. MODs are of crucial importance for location-based services, context-aware computing, and many other application domains.

Generally, an object's trajectory is represented by a polyline in time and space where the vertices are the timestamped positions acquired by a suitable positioning system [16,20,8]. Many of these systems (including GPS) are based on sensors that are attached to the moving objects. Tracking the trajectories of such objects therefore requires communicating the position data to the MOD over a wireless network.

Transmitting and storing every sensed position of an object's trajectory, however, causes high communication costs and generally consumes too much storage capacity. The former particularly applies if the MOD has to be informed in real-time about the object's movement, as required for many applications. For example, an ordinary GPS receiver may generate more than 30

million position data records per year, and there may be thousands of objects to be tracked.

Therefore, tracking protocols are needed that allow trading these costs off against the accuracy of the trajectory information known to the MOD. With such a protocol, the MOD manages only a simplified trajectory that is given by a subset of the sensed positions and that does not deviate by more than a certain accuracy bound ϵ from the actual movement. We refer to the problem of minimizing the amount of data that is communicated and stored as efficient real-time *trajectory tracking*. A formal problem statement is given at the end of Section 2.

Short problem analysis: Trajectory tracking is related to line simplification, on the one hand, and protocols for tracking an object's current position, on the other hand. For clarity, we refer to the latter as (real-time) *position tracking* in the following.

Line simplification refers to a multitude of algorithmic problems on approximating a given polyline by a simplified one with fewer vertices. In the terminology of line simplification, trajectory tracking is a min-# problem in $\mathbb{R}^{1+d}$ ($d = 2$ or 3) in the case of Hausdorff distance under the (time-)uniform distance metric [3,2]. A straightforward approach for trajectory tracking based on line simplification is to transmit the sensed position data from the objects to the MOD and to perform the simplification entirely on the MOD – for instance using the Douglas-Peucker algorithm [6], as explained in [4]. However, such a solution has an obvious drawback as also those positions are transferred over the wireless network that are dropped later by simplification, which may cause a substantial waste of bandwidth.

Position tracking protocols, in contrast, aim at minimizing the communication cost for informing the MOD about the current position of an object, but do not necessarily generate a simplified trajectory of the past movement. The latter particularly applies to the most efficient protocols based on dead reckoning. With this technique, a tracked object initially transmits a function predicting its future movement to the MOD. This prediction function is updated only if the object's locally sensed position impends to deviate from the predicted one by more than some accuracy bound ϵ. Consequently, only those sensing operations that require an adjustment of the prediction cause an update message to be sent. The most simple but nevertheless efficient variant is linear dead reckoning (LDR) [31,17,30]. It uses a linear prediction given by a timestamped position and a velocity vector. Dead reckoning does not generate a simplified trajectory, as it describes the object's movement by a sequence of disconnected line segments – one for each prediction.

LDR	Linear Dead Reckoning
LDRH	Linear Dead Reckoning with half ϵ
CDR	Connection-Preserving Dead Reckoning
CDR_m	... with limited sensing history (m positions)
GRTS	Generic Remote Trajectory Simplification
GRTS_k	... with limited variable part (k vertices)
GRTS_m	... with limited sensing history (m positions)
GRTS_mc	... with additional compression technique
$\text{GRTS}_\text{mc}^\text{Opt}$	... realized with optimal simplification algo.
$\text{GRTS}_\text{m}^\text{Sec}$	... realized with segment heuristic

Table 1 List of tracking protocols.

However, as shown by Trajcevski et al. in [27], a simplified trajectory deviating from the actual movement by up to 2ϵ can be computed on the basis of the linear predictions of LDR. For trajectory tracking with accuracy bound ϵ, Trajcevski et al. therefore propose to use LDR with $\epsilon' := \epsilon/2$. We refer to this approach as LDRH (linear dead reckoning with half ϵ) in the following. As explained in detail in Section 3, the use of $\epsilon/2$ is very conservative and leaves room for significant improvement.

Contribution: This problem analysis shows that trajectory tracking involves several trade-offs between accuracy, computational costs, and the two goals of reducing both the communication cost and the amount of simplified trajectory data to a minimum. Moreover, to guarantee real-time behavior, tracking algorithms with bounded computing time per sensing operation are needed.

In this paper, we propose a family of trajectory tracking protocols that allows adjusting between these goals and costs and an accuracy bound ϵ. The family is derived from two basic protocols named *Connection-Preserving Dead Reckoning* (CDR) and *Generic Remote Trajectory Simplification* (GRTS), cf. Table 1.

CDR extends LDR by a second condition for sending an update to the MOD such that the origins of the linear predictions give a simplified trajectory that approximates the actual movement by ϵ. Hence, it uses dead reckoning for tracking the current position as well as for simplifying the past trajectory. While this leads to very small message sizes and thus communication cost, the efficiency of simplification depends on dead reckoning, which has been designed for position tracking rather than trajectory simplification.

GRTS in contrast clearly separates tracking the current position from simplifying the past trajectory. It also applies dead reckoning for tracking the current position – to optimize the number of messages sent over the wireless network – but can be combined with any

line simplification algorithm suited for trajectories (e.g., [6,11,18]) to reduce the trajectory data. This separation increases the message sizes but affords significantly greater reductions of the amounts of data to be stored by the MOD.

The possibility to realize GRTS with different line simplification algorithms further enables to trade computational cost off against reduction efficiency. For example, an optimal line simplification algorithm provides the best reduction but causes the highest computational cost, whereas solutions based on heuristics lower the computational overhead at the cost of smaller reductions. We investigate two realizations of GRTS with different line simplification algorithms, namely the optimal line simplification algorithm introduced in [11] and a simple but efficient simplification heuristic [18]. For the latter, we further propose an optimization reducing the average space consumption of the algorithm by 63%.

For both CDR and GRTS, we propose space- and time-bounded variants ($\mathrm{CDR_m}$ and $\mathrm{GRTS_m}$) to limit the computing time at the moving objects, as motivated above. In case of GRTS, the input for the line simplification algorithm – referred to as *sensing history* – is limited to m sensed positions. To optimize line simplification under this constraint, we present a novel compression technique for the sensing history, leading to $\mathrm{GRTS_{mc}}$. This technique can be generally used to optimize real-time online simplification of polylines.

We further discuss how to take sensing inaccuracies and possible movements between two sensing operations into account. As physical constraint for the latter, we consider not only the maximum speed, but also the maximum acceleration.

Our evaluations with hundreds of real GPS traces show that GRTS outperforms LDRH by a factor five in terms of reduction efficiency. With the above-mentioned simplification heuristic, the reduction efficiency of GRTS is less than 15% below the best possible reduction computed offline – even when restricting the space consumption of the algorithm to less than 12 kB using the $\mathrm{GRTS_m}$ variant. The computing time of this realization $\mathrm{GRTS_m^{Sec}}$ and parametrization is bounded to less than 1.9 ms on a 600 MHz smartphone and to 0.07 ms on a 3 GHz Intel Xeon processor.

$\mathrm{GRTS_{mc}}$ realized with the optimal line simplification algorithm may even reach 97% of the best possible reduction rate, at one hundred times higher cost.

Organization of the paper: In Section 2, we describe our assumptions and introduce our notation. Furthermore, a formal definition of the efficient real-time trajectory tracking problem is given. In Section 3, we analyze the use of dead reckoning for trajectory tracking, before we propose the CDR protocol with its variants in Section 4. Then, we present the GRTS protocol with its three variants and the mentioned realizations in Section 5. Besides, we discuss how time-dependent sensing deviations can be included into GRTS. In Section 6, we explain how to determine the possible movement between two sensing operations by a given maximum acceleration and compare the resulting offsets for ϵ with the offsets obtained by a given maximum speed. In Section 7, we show and analyze results from extensive simulations with real GPS traces, before we discuss related work in Section 8. Finally, the paper is concluded in Section 9 with a summary.

2 Assumptions and Notation

We consider a collection of mobile objects with embedded positioning sensors (e.g., GPS receivers) whose trajectories are managed by a remote MOD. The objects and the MOD are connected by a wireless network. The overall number of trajectories stored by the MOD is of no relevance here.

An object's movement over time describes a continuous function $\mathbf{a} : \mathbb{R} \mapsto \mathbb{R}^d$ from time to plane ($d = 2$) or space ($d = 3$), called the object's *actual trajectory*. Let t_C denote the current time, then $\mathbf{a}(t)$ is defined up to t_C and $\mathbf{a}(t_\mathrm{C})$ is the object's current actual position.

The positioning sensor periodically senses the object's current position with period T_S, referred to as *sensing period*. It results in a sequence of *sensed positions* $(s_1, s_2, \ldots, s_\mathrm{R})$, where s_1 denotes the first and s_R the most recent sensed position. Each s_i is a data record consisting of the sensing time t and the sensed position $\mathbf{p}$, denoted by $s_i.t$ and $s_i.\mathbf{p}$, respectively.

Two consecutive positions s_i and s_{i+1} define a *spatiotemporal line segment* $\overline{s_i\, s_{i+1}}$ as

$$\overline{s_i\, s_{i+1}} : t \mapsto \frac{(s_{i+1}.t - t)\, s_i.\mathbf{p} + (t - s_i.t)\, s_{i+1}.\mathbf{p}}{s_{i+1}.t - s_i.t}$$

on the domain $[s_i.t, s_{i+1}.t]$.

Based on these line segments, the sequence of all sensed positions defines a continuous, piecewise linear function $\mathbf{s}(t)$ named *sensed trajectory* as

$$\mathbf{s} : t \mapsto \overline{s_i\, s_{i+1}}(t) \text{ where } s_i.t \leq t \leq s_{i+1}.t$$

on the domain $[s_1.t, s_\mathrm{R}.t]$. Geometrically, $\mathbf{s}(t)$ is a time-monotonous polyline in $\mathbb{R}^{1+d}$ given by the sequence of vertices $(s_1, s_2, \ldots, s_\mathrm{R})$.

Note that the domain $[s_1.t, s_\mathrm{R}.t]$ does not continuously increase over time but periodically by T_S, with

each sensing operation. For current time t_C, we thus have $t_C - T_S < s_R.t \leq t_C$.

The sensed trajectory $\mathbf{s}(t)$ generally deviates from $\mathbf{a}(t)$ due to inaccuracies of the positioning sensor and the time-discrete sensing. The former are generally described by stochastic means such as probability density functions or percentiles, which allow deriving a *maximum sensor inaccuracy* σ that holds with high probability. Inaccuracies beyond σ (typically indicated by erratic positions) are considered as errors. They have to be treated separately, e.g., by informing the MOD that there will be no valid trajectory information until further notice. Regarding the time discretization by position sensing, the movement between two sensing operations is subject to physical constraints like the maximum speed or acceleration.

Therefore, we assume that the deviation between $\mathbf{s}(t)$ and $\mathbf{a}(t)$ is bounded by a certain *maximum sensing deviation* δ, i.e., $\forall\, t' \in [s_1.t, s_R.t]$ we have $|\mathbf{s}(t') - \mathbf{a}(t')| \leq \delta$. For example, given a maximum speed v_{max}, we can conclude that

$$|\mathbf{a}(t') - \mathbf{s}(t')| \leq \sigma + v_{max}\frac{T_S}{2} =: \delta \ ,$$

as the object cannot move more than $v_{max} \cdot T_S/2$ and then return to its origin during a sensing period T_S.

The use of speed-based movement constraints to estimate the deviation between $\mathbf{s}(t)$ and $\mathbf{a}(t)$ is discussed in detail in [22]. In Section 6, we show how to incorporate acceleration-based constraints, which typically afford smaller values of δ.

The sensor inaccuracy may depend on dynamic technical conditions such as the satellite constellation of GPS, described by the dilution of precision (DOP). Therefore, σ may be time-dependent, and thus δ. For simplicity, we assume σ and δ to be fixed at first. In Section 5.3, we discuss how to account for time-dependent values of σ and δ, given with each sensed position.

Note that the physical movement constraints also allow estimating $\mathbf{a}(t')$ for $t' > s_R.t$. For example, given a maximum speed v_{max}, the actual position $\mathbf{a}(t')$ is known to be inside a circle with radius $\sigma + v_{max}(t' - s_R.t)$ around $s_R.\mathbf{p}$. This property is utilized by dead reckoning, as explained in Section 3.

In this regard, we also assume that the time for processing and transmitting an update message to the MOD is bounded by a certain maximum time span T_U called *update time*. Exceptional transmission delays and connection breaks are considered as errors and have to be detected by subsidiary mechanisms such as heartbeat messages. These practical implementation issues are addressed in Section 7.5

We further assume that clocks of the MOD and the moving object are synchronized to within few microseconds or tens of microseconds. Note that GPS provides very accurate timing signals. Given that the timestamps of the sensed positions are accurate, the clock discrepancy between the MOD and the moving object is not relevant for queries about the past movement but only for queries about the current position – and can therefore be included into T_U.

$\mathbf{a}(t)$	Actual trajectory – a function from time to $\mathbb{R}^d$
s_i	Sensed position – with position data $s_i.\mathbf{p}$ at time $s_i.t$
s_R	Most recent sensed position
$\mathbf{s}(t)$	Sensed trajectory – given by $(s_1, s_2, \ldots, s_R)$
$\mathbf{u}(t)$	Simplified trajectory – given by $(u_1, u_2, \ldots)$
u_i	Vertex of $\mathbf{u}(t)$ – with position data $u_i.\mathbf{p}$ for time $u_i.t$
ϵ	Accuracy bound – maximum tolerated deviation between $\mathbf{u}(t)$ and $\mathbf{a}(t)$
t_C	Current point in time
T_S	Sensing period – time between a s_{i-1} and s_i
σ	Max. sensor inaccuracy – between $s_i.\mathbf{p}$ and $\mathbf{a}(s_i.t)$
δ	Max. sensing deviation – between $\mathbf{s}(t)$ and $\mathbf{a}(t)$
v_{max}	Maximum speed of the moving object
T_U	Update time – upper bound for processing and transmitting an update message
$\boldsymbol{\pi}(t)$	Prediction function – cf. Section 3
π_O	Prediction origin – a sensed position
$\boldsymbol{\pi}_V$	Prediction velocity – a vector for linear prediction
a_{max}	Maximum acceleration – cf. Section 6

Table 2 List of symbols.

The MOD describes the object's trajectory by a continuous, piecewise linear function $\mathbf{u} : t \mapsto \mathbb{R}^d$ called *simplified trajectory*. Geometrically, $\mathbf{u}(t)$ is a time-monotonous polyline in $\mathbb{R}^{1+d}$ given by a sequence of vertices $(u_1, u_2, \ldots)$, like $\mathbf{s}(t)$. Each vertex u_i is a data record with attributes t and $\mathbf{p}$, just as a sensed position.

Table 2 gives a summary of the symbols introduced in this section. With this notation, the algorithm problem of tracking a moving object's trajectory efficiently in real-time can be formally stated as follows:

Problem statement [Efficient real-time trajectory tracking]: The goals are to minimize the number of vertices of the simplified trajectory $\mathbf{u}(t)$ and the amount of data transmitted over the wireless network under the following two constraints, where t_C denotes the current time:

1. *Simplification constraint:* For a given *accuracy bound* ϵ known by the moving object and the MOD, it is

   $$\forall\, t' \in [s_1.t, t_C] \ : \ |\mathbf{u}(t') - \mathbf{a}(t')| \leq \epsilon \ .$$

2. *Real-time constraint:* At t_C, position $\mathbf{u}(t)$ is available at the MOD for every $t \in [s_1.t, t_C]$.

The goals of minimizing the number of vertices of $\mathbf{u}(t)$ and the amount of communicated data appear to imply one another. Obviously, the greater ϵ, the less vertices

are needed for $\mathbf{u}(t)$ and the less data has to be transmitted. However, the two goals might also contradict to a certain degree: To generate a $\mathbf{u}(t)$ with a very small number of vertices, it has to be revised over time – i.e., an update may need to replace multiple vertices from previous updates. This causes larger message sizes and thus communication cost. The quantitative impact of this property and respective conclusions are discussed in Section 7.

3 Analysis of Linear Dead Reckoning

Before we present our approaches CDR and GRTS, we analyze the use of linear dead reckoning (LDR) for trajectory tracking – also because CDR and GRTS make use of LDR.

As explained above, LDR is an efficient mechanism for tracking the current position of a moving object with low communication costs [31,17,30]. With LDR, the moving object and the MOD share a linear prediction function $\boldsymbol{\pi}(t)$ for determining the object's current position. $\boldsymbol{\pi}(t)$ is defined by a previously sensed position π_O called *prediction origin* and a *velocity vector* $\boldsymbol{\pi}_\mathrm{V}$ as

$$\boldsymbol{\pi} : t \mapsto \pi_\mathrm{O}.\mathbf{p} + (t - \pi_\mathrm{O}.t)\boldsymbol{\pi}_\mathrm{V}$$

for $t \geq \pi_\mathrm{O}.t$. For a given accuracy bound ϵ, LDR guarantees that $\boldsymbol{\pi}(t)$ known by the MOD approximates the objects' current actual position by ϵ. Formally, at current time t_C, it guarantees that $|\boldsymbol{\pi}(t_\mathrm{C}) - \mathbf{a}(t_\mathrm{C})| \leq \epsilon$.

After having sensed a new position at time $s_\mathrm{R}.t$, it has to be decided whether $\boldsymbol{\pi}(t)$ is going to meet this guarantee during the following sensing period $[s_\mathrm{R}.t, s_\mathrm{R}.t + T_\mathrm{S}]$ as well as during the time span until a subsequent update would have been processed. If not, the object has to send a new prediction $(\pi_\mathrm{O}, \boldsymbol{\pi}_\mathrm{V})$ right now. For this decision, the maximum sensor inaccuracy σ has to be incorporated as well.

Therefore, we say that the moving object has to send a new prediction if $|\boldsymbol{\pi}(t_\mathrm{C}) - \mathbf{a}(t_\mathrm{C})|$ *impends* to reach ϵ. For example, assuming a maximum velocity v_max, it has to send an update if

$$|s_\mathrm{R}.\mathbf{p} - \boldsymbol{\pi}(s_\mathrm{R}.t + T_\mathrm{S} + T_\mathrm{U})| + \sigma + v_\mathrm{max}(T_\mathrm{S} + T_\mathrm{U}) > \epsilon \ ,$$

as $s_\mathrm{R}.\mathbf{p}$ may deviate by up to σ from the actual position at $s_\mathrm{R}.t$, and the object may move by up to $v_\mathrm{max}(T_\mathrm{S}+T_\mathrm{U})$ until an update after the subsequent sensing operation would have been processed.[1]

[1] Note that many works do not clearly state whether they account for the sensing period and update time, or not. Some works even ignore the sensor inaccuracy σ since it can be initially offset against ϵ, as long as it does not vary over time.

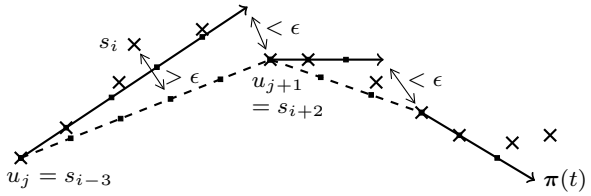

Fig. 1 Example of a violation of ϵ for trajectory tracking by LDR.

Because of the discontinuities between the different predictions, LDR is a position tracking protocol rather than a trajectory tracking protocol. More precisely, it represents the object's movement by a sequence of disconnected spatiotemporal line segments instead of a continuous polyline. Figure 1 illustrates such discontinuities. The solid arrows denote the linear predictions – and thus the line segments – whereas the small crosses indicate the sensed positions.

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

In [27], Trajcevski et al. utilize this property for trajectory tracking. They analyze the spatiotemporal polyline given by the origins of the linear predictions and prove that it approximates the actual movement by 2ϵ. Figure 1 illustrates this polyline by a dashed line. Based on this finding, they conclude that LDR allows for trajectory tracking with accuracy bound ϵ as follows:

1. The moving object reports its current position using LDR with the accuracy bound $\epsilon' := \epsilon/2$.
2. The MOD not only stores the current prediction, but also the origins of all previous predictions as vertices of the simplified trajectory $\mathbf{u}(t)$.

This approach, which we will refer to as LDRH in the following, is very conservative in terms of the reduction efficiency. Consider again the polyline $\mathbf{u}(t)$ given by the prediction origins of LDR with accuracy bound ϵ. It may deviate by more than ϵ from the actual trajectory $\mathbf{a}(t)$, as just explained. We argue that such violations are rare and will seldom reach 2ϵ:

1. The actual position $\mathbf{a}(t')$ at time t' can only deviate by more than ϵ from the corresponding line segment $\overline{u_j\, u_{j+1}}$ if $\mathbf{a}(t')$ and u_{j+1} are located at opposite sides of the predicted movement vector. This does not hold for typical movement patterns like turning off or stopping.
2. The deviation $|\mathbf{a}(t') - \overline{u_j\, u_{j+1}}|$ can be close to 2ϵ if $t' \approx u_{j+1}.t$ and $|\mathbf{a}(t') - u_{j+1}.\mathbf{p}|$ is also about 2ϵ. Thus, for such large deviations, the object has to move in a very fast and irregular fashion.

To support this argumentation, we analyzed the actual number of such violations by simulating LDR with a

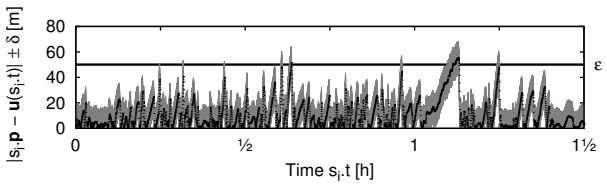

Fig. 2 Potential violations of ϵ for trajectory tracking by LDR during the first $1\frac{1}{2}$ hours of a 4-hour bicycle tour.

GPS trace of a 4-hour bicycle tour from the OpenStreetMap [21] project. For the simulation, we used $\sigma = 7.8\,\text{m}$, $v_{\max} = 10\,\text{m/s}$, $T_\text{S} = 1\,\text{s}$, $T_\text{U} = 0.2\,\text{s}$, and $\epsilon = 50\,\text{m}$.

With these assumptions and parameters, LDR generates 396 position updates, where the last one indicates the end of the tour. Hence, the resulting simplified trajectory $\mathbf{u}(t)$ consists of 395 line segments given by 396 vertices. Figure 2 illustrates the deviations between the sensed positions s_i and $\mathbf{u}(t)$ depending on time, for the first $1\frac{1}{2}$ hours of the tour. The black curve of dots denotes the sensed positions. The corresponding gray bars depict the maximum sensing deviation $\delta = 12.8\,\text{m}$.

Each time the gray bars intersect the thick horizontal line, the deviation between $\mathbf{a}(t)$ and $\mathbf{u}(t)$ may violate the accuracy bound ϵ. Formally, this applies to all s_i where $|s_i.\mathbf{p} - \mathbf{u}(s_i.t)| + \delta > \epsilon$. Thus, the difference $\epsilon - \delta$ represents the largest tolerable deviation between a sensed position s_i and $\mathbf{u}(t)$. Obviously, the violations are not distributed uniformly over time but appear at few line segments of $\mathbf{u}(t)$ only. In detail, during the four hours, they appear at 15 of the 395 line segments. Furthermore, the deviations are well below 2ϵ.

We conclude that using LDR with $\epsilon' := \epsilon/2$ (i.e., LDRH) is generally too strict. It generates needless position updates and hence simplified trajectories with unnecessary large numbers of vertices.

Next, we present an approach for trajectory tracking that extends LDR to prevent such violations, before we present the GRTS protocol, which clearly separates tracking of the current position from simplification of the past trajectory.

4 Connection-Preserving Dead Reckoning

In this section, we propose *Connection-Preserving Dead Reckoning* (CDR), which extends LDR such that the prediction origins make up the vertices of a simplified trajectory $\mathbf{u}(t)$ that approximates $\mathbf{a}(t)$ by the accuracy bound of LDR. First, we present the basic CDR algorithm executed at the moving object, before we discuss an optimization of this algorithm. Then, we present a space- and time-bounded variant of this algorithm named CDR_m.

```
1:  s_R ← sense position                    ▷ Most recent sensed position.
2:  π_O ← s_R                               ▷ Prediction origin.
3:  π_V ← 0                                 ▷ Predicted velocity.
4:  send update message (π_O, π_V) to MOD
5:  𝕊 ← {}                                  ▷ Sensing history since last update.
6:  s_R' ← s_R                              ▷ Second last sensed position.
7:  while report movement do
8:    s_R ← sense position
9:    if LDR causes update
          or ∃ s_i ∈ 𝕊 : |s_i − π_O s_R(s_i.t)| > ε − δ then
10:     π_O ← s_R'
11:     π_V ← compute new predicted velocity ...
12:     send update message (π_O, π_V) to MOD
13:     𝕊 ← {}                              ▷ Clear the sensing history.
14:   end if
15:   𝕊 ← 𝕊 ∪ {s_R}                         ▷ Add s_R to sensing history.
16:   s_R' ← s_R
17: end while
18: send final update message (s_R) to MOD
```

Fig. 3 Basic version of CDR algorithm.

4.1 Basic Version of CDR

As explained in Section 3, the prediction origins of LDR with accuracy bound ϵ make up a simplified trajectory $\mathbf{u}(t)$ that approximates $\mathbf{a}(t)$ by ϵ for most of the time. However, some line segments of $\mathbf{u}(t)$ may violate ϵ. CDR is based on the observation that the moving object has all information for detecting such violations in real-time:

1. It knows the current prediction given by the prediction origin π_O and the velocity vector $\boldsymbol{\pi}_\text{V}$.
2. It knows the most recent sensed position s_R.
3. Thus, it also knows the resulting line segment $\overline{\pi_\text{O}\, s_\text{R}}$ of $\mathbf{u}(t)$, in case s_R is used 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 $\pi_\text{O}.t$, and check whether one of these positions deviates from $\overline{\pi_\text{O}\, s_\text{R}}$ by more than $\epsilon - \delta$. If so, the line segment $\overline{\pi_\text{O}\, s_\text{R}}$ may deviate by more than ϵ from $\mathbf{a}(t)$. In the following we refer to the sensing history as $\mathbb{S} := \{s_i : s_i.t > \pi_\text{O}.t\}$.

The basic idea of CDR is that the moving object not only sends a new position update if caused by LDR, but also if one of the sensed positions since the last update message deviates from $\overline{\pi_\text{O}\, s_\text{R}}$ by more than $\epsilon - \delta$.

Figure 3 shows the pseudocode of the algorithm executed by moving object. A crucial difference to LDR is that CDR maintains a dynamic array that stores the sensing history (line 5). Another, subtle difference

to LDR is that CDR does not use the most recent sensed position s_{R} as origin of a new prediction, but the one before s_{R}, denoted by $s_{\mathrm{R}'}$ (line 10). The use of $s_{\mathrm{R}'}$ has negligible or no influence on LDR since the predicted velocity $\boldsymbol{\pi}_{\mathrm{V}}$ generally is determined by means of the last sensed positions and particularly $(s_{\mathrm{R}}.\mathbf{p} - s_{\mathrm{R}'}.\mathbf{p})/(s_{\mathrm{R}}.t - s_{\mathrm{R}'}.t)$. Yet, it is essential for $\mathbf{u}(t)$, as explained below.

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 17) as long as it wants to report its movement to the MOD.

During each iteration, it first senses its current position (line 8) and then checks whether LDR causes an update or whether the *segment condition*, given as

$$\forall\, s_i \in \mathbb{S} : |s_i - \overline{\pi_{\mathrm{O}}\, s_{\mathrm{R}}}(s_i.t)| \leq \epsilon - \delta \ ,$$

is violated (line 9). The segment condition simply states that none of the sensed positions s_i since the last update should deviate from $\overline{\pi_{\mathrm{O}}\, s_{\mathrm{R}}}$ by more than $\epsilon - \delta$, as discussed above. If there exists an s_i deviating by more than $\epsilon - \delta$, a new update message with prediction origin $\pi_{\mathrm{O}} = s_{\mathrm{R}'}$ is sent to the MOD such that the corresponding line segment of the simplified trajectory $\mathbf{u}(t)$ fulfills the segment condition.

After sending an update message, $\mathbb{S}$ is cleared to remove the sensed positions before $\pi_{\mathrm{O}}.t$ (line 13).

If the moving object wants to stop reporting its movement, it sends a final update message with the most recent sensed position – but without a new prediction – and terminates the algorithm (line 18).

The simplified trajectory $\mathbf{u}(t)$ managed by the MOD consists of two parts: the spatiotemporal polyline given by the vertices $(u_1, \ldots, u_n)$, as described in Section 2, and the prediction function $\boldsymbol{\pi}(t)$ of LDR. On receiving an update message $(\pi_{\mathrm{O}}, \boldsymbol{\pi}_{\mathrm{V}})$, the MOD simply updates $\boldsymbol{\pi}(t)$ with the new origin π_{O} and the new velocity vector $\boldsymbol{\pi}_{\mathrm{V}}$ and appends π_{O} to the sequence of vertices as $(n+1)$th vertex.

Given a query for the moving object's position at time t', the MOD answers as follows:

- $t' \leq \pi_{\mathrm{O}}.t$ *:* The MOD calculates $\mathbf{u}(t')$ as described in Section 2 and returns the result to the query issuer.
- $t' > \pi_{\mathrm{O}}.t$ *:* It calculates the predicted position at time t' using $\boldsymbol{\pi}(t') = \pi_{\mathrm{O}}.\mathbf{p} + (t' - \pi_{\mathrm{O}}.t)\boldsymbol{\pi}_{\mathrm{V}}$ and returns the result to the query issuer.

If the MOD receives the final update message (s_{R}) it removes $\boldsymbol{\pi}(t)$ and completes $\mathbf{u}(t)$ by appending s_{R} as final vertex to $(u_1, \ldots, u_n)$. In practical implementations, the MOD should also terminate the simplified trajectory during long-lasting network outages or after failures at the mobile object, which can be detected by a timeout mechanism (cf. Section 7.5).

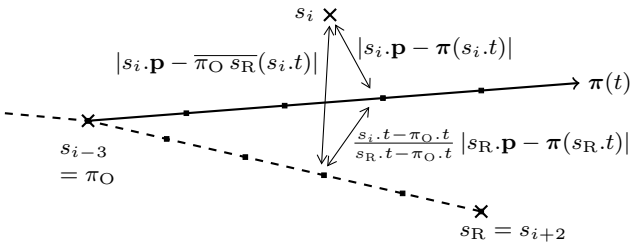

Fig. 4 Geometric illustration of triangle inequality for optimization of $\mathbb{S}$.

4.2 Optimization of Sensing History

CDR differs from LDR regarding the space requirements at the moving object. While LDR only stores the current prediction and the most recent 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.

However, this problem can be alleviated. For every sensed position $s_i \in \mathbb{S}$ there exists a certain point in time from which onwards, it cannot violate the segment condition without s_{R} causing LDR to send an update. After this time, s_i can be removed from $\mathbb{S}$, even before the next update. This significantly reduces the space consumption of CDR as well as the computing time per position fix. For example, in case of the bicycle tour mentioned in Section 3 and $\epsilon = 50\,\mathrm{m}$, the maximum size of $\mathbb{S}$ is reduced from 405 to 214 positions. The maximum computing time per position fix on a 3 GHz Intel Xeon processor (cf. Section 7) is reduced from 0.10 to 0.06 ms.

To determine this point in time for a given $s_i \in \mathbb{S}$, we analyze the state of the basic version of CDR (cf. Figure 3) right after having sensed a new position s_{R} (line 8) that does not cause an update by LDR.

We consider the line segment $\overline{\pi_{\mathrm{O}}\, s_{\mathrm{R}}}$, which is going to be the next line segment $\overline{u_n\, u_{n+1}}$ of $\mathbf{u}(t)$ if an update is sent in the subsequent iteration of the while loop.

Since the prediction function $\boldsymbol{\pi}(t)$ and $\overline{\pi_{\mathrm{O}}\, s_{\mathrm{R}}}$ are linear functions in t with identical origin π_{O}, we conclude that regarding time $s_i.t$ they deviate by $\frac{s_i.t - \pi_{\mathrm{O}}.t}{s_{\mathrm{R}}.t - \pi_{\mathrm{O}}.t}\,|s_{\mathrm{R}}.\mathbf{p} - \boldsymbol{\pi}(s_{\mathrm{R}}.t)|$.

As illustrated in Figure 4, the following triangle inequality can be derived:

$$|s_i.\mathbf{p} - \overline{\pi_{\mathrm{O}}\, s_{\mathrm{R}}}(s_i.t)| \leq |s_i.\mathbf{p} - \boldsymbol{\pi}(s_i.t)| + \frac{s_i.t - \pi_{\mathrm{O}}.t}{s_{\mathrm{R}}.t - \pi_{\mathrm{O}}.t}\,|s_{\mathrm{R}}.\mathbf{p} - \boldsymbol{\pi}(s_{\mathrm{R}}.t)| \tag{1}$$

Under the above assumption that s_{R} does not cause an update, we conclude that $|s_{\mathrm{R}}.\mathbf{p} - \boldsymbol{\pi}(s_{\mathrm{R}}.t)| \leq \epsilon$ and finally estimate $|s_i.\mathbf{p} - \overline{\pi_{\mathrm{O}}\, s_{\mathrm{R}}}(s_i.t)|$ by

$$|s_i.\mathbf{p} - \overline{\pi_{\mathrm{O}}\, s_{\mathrm{R}}}(s_i.t)| \leq |s_i.\mathbf{p} - \boldsymbol{\pi}(s_i.t)| + \frac{s_i.t - \pi_{\mathrm{O}}.t}{s_{\mathrm{R}}.t - \pi_{\mathrm{O}}.t}\epsilon \ .$$

```
1: [...]
2: while report movement do
3:    s_R ← sense position
4:    while |𝕊| > 0 and s_R.t ≥ κ(peek(𝕊)) do
5:       pop(𝕊)                          ▷ Remove root of heap.
6:    end while
7:    [...]
8: end while
9: send final update message (s_R) to MOD
```

Fig. 5 Optimization of $\mathbb{S}$ in the CDR algorithm.

Clearly, this estimate decreases over time, i.e., with increasing values of $s_\mathrm{R}.t$. We now derive the point in time from which onwards it falls below $\epsilon - \delta$, as required to fulfill the segment condition definitely:

$$\begin{aligned} &\epsilon - \delta \geq |s_i.\mathbf{p} - \boldsymbol{\pi}(s_i.t)| + \frac{s_i.t - \pi_\mathrm{O}.t}{s_\mathrm{R}.t - \pi_\mathrm{O}.t}\epsilon \\ \Leftrightarrow\ & s_\mathrm{R}.t \geq \underbrace{\frac{s_i.t - \pi_\mathrm{O}.t}{\epsilon - \delta - |s_i.\mathbf{p} - \boldsymbol{\pi}(s_i.t)|}\epsilon + \pi_\mathrm{O}.t}_{=:\,\kappa(s_i)} \end{aligned} \tag{2}$$

Thus, s_i cannot violate the segment condition once time $s_\mathrm{R}.t$ fulfills the inequation (2).

So far, we assumed that s_R does not cause an update by LDR. In the general case, it follows that once $s_\mathrm{R}.t$ fulfills (2), s_i cannot violate the segment condition without s_R causing an update by LDR. Therefore, CDR can remove s_i from $\mathbb{S}$ at this point in time without affecting its future decisions on a new update.

For this purpose, CDR organizes $\mathbb{S}$ as a min-heap according to the right-hand side of (2), i.e., $\kappa(s_i)$. After position sensing, it first removes the root of $\mathbb{S}$ one by one, as long as this sensed position fulfills (2). Figure 5 shows the corresponding additional pseudocode to the basic version of CDR.

4.3 Space- and Time-bounded CDR

With the optimization presented above, CDR tries to reduce the sensing history $\mathbb{S}$ after each position fix. The space consumption is nevertheless unbounded, which can be critical for resource-constrained mobile devices.

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

$\mathrm{CDR_m}$ is based on the following idea: Besides a heap of fixed size m for storing $\mathbb{S}$, it maintains a floating-point variable $d_\mathbb{S}$ providing aggregated information on all sensed positions that could not be stored in $\mathbb{S}$ due to the space constraint. More precisely, $d_\mathbb{S}$ defines a time-dependent bound for $|s_\mathrm{R}.\mathbf{p} - \boldsymbol{\pi}(s_\mathrm{R}.t)|$. Each time $|\mathbb{S}|$ is going to exceed m, the $\mathrm{CDR_m}$ algorithm removes a sensed position from $\mathbb{S}$ and updates $d_\mathbb{S}$ accordingly.

If $|s_\mathrm{R}.\mathbf{p} - \boldsymbol{\pi}(s_\mathrm{R}.t)|$ is below the bound defined by $d_\mathbb{S}$, none of the sensed positions that could not be stored in $\mathbb{S}$ violates the segment condition for s_R.

For this purpose, the segment condition is split into two subconditions: The first subcondition is evaluated on the sensed positions currently stored in $\mathbb{S}$, just as with CDR. The second subcondition is evaluated on $d_\mathbb{S}$.

We now give the mathematical basis for $d_\mathbb{S}$ and derive the inequation for the second subcondition. First, we reconsider the triangle inequality (1), given in Section 4.2. With it, we conclude that

$$|s_i.\mathbf{p} - \boldsymbol{\pi}(s_i.t)| + \frac{s_i.t - \pi_\mathrm{O}.t}{s_\mathrm{R}.t - \pi_\mathrm{O}.t}\,|s_\mathrm{R}.\mathbf{p} - \boldsymbol{\pi}(s_\mathrm{R}.t)| \leq \epsilon - \delta$$

implies $|s_i.\mathbf{p} - \overline{\pi_\mathrm{O}\, s_\mathrm{R}}(s_i.t)| \leq \epsilon - \delta$. The former inequation can be rewritten as

$$|s_\mathrm{R}.\mathbf{p} - \boldsymbol{\pi}(s_\mathrm{R}.t)| \leq \underbrace{\frac{\epsilon - \delta - |s_i.\mathbf{p} - \boldsymbol{\pi}(s_i.t)|}{s_i.t - \pi_\mathrm{O}.t}}_{=:\,\varphi(s_i)}(s_\mathrm{R}.t - \pi_\mathrm{O}.t)\ .$$

Thus, $|s_\mathrm{R}.\mathbf{p} - \boldsymbol{\pi}(s_\mathrm{R}.t)| \leq \varphi(s_i) \cdot (s_\mathrm{R}.t - \pi_\mathrm{O}.t)$ implies that s_i does not violate the segment condition. This result is used for the $\mathrm{CDR_m}$ algorithm as follows:

1. The minimum $\varphi(s_j)$ of all sensed positions s_j that had to be removed from $\mathbb{S}$ due to the space constraint is stored in the variable $d_\mathbb{S}$.
2. Inequation $|s_\mathrm{R}.\mathbf{p} - \boldsymbol{\pi}(s_\mathrm{R}.t)| \leq d_\mathbb{S} \cdot (s_\mathrm{R}.t - \pi_\mathrm{O}.t)$ is used as second subcondition of the segment condition. Thus, an update is sent if $|s_\mathrm{R}.\mathbf{p} - \boldsymbol{\pi}(s_\mathrm{R}.t)|$ exceeds $d_\mathbb{S} \cdot (s_\mathrm{R}.t - \pi_\mathrm{O}.t)$.

Therefore, as long as the second subcondition is fulfilled, each removed position s_j fulfills the segment condition. Thus, as long as both subconditions are fulfilled, every sensed position since the last position update fulfills the segment condition. After an update, $\mathbb{S}$ is cleared – just as with CDR – and $d_\mathbb{S}$ is reset 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 most recent sensed position s_R violates the second subcondition more likely. Therefore, $\mathrm{CDR_m}$ always removes the $s_j \in \mathbb{S}$ with maximum $\varphi(s_j)$. For this purpose, it stores $\mathbb{S}$ as a max-heap according to $\varphi(s_i)$. This order is identical to the min-heap order of CDR by $\kappa(s_i)$, as explained below.

Since $d_\mathbb{S}$ aggregates all previously sensed position with $\varphi(s_j) \geq d_\mathbb{S}$, the most recent sensed position s_R need not be added to $\mathbb{S}$ if $\varphi(s_\mathrm{R}) \geq d_\mathbb{S}$. Hence, the invariant $\forall s_i \in \mathbb{S} : \varphi(s_i) \leq d_\mathbb{S}$ holds for $\mathbb{S}$. For this reason, $\mathrm{CDR_m}$ can directly assign $\varphi(\mathrm{pop}(\mathbb{S}))$ to $d_\mathbb{S}$ when

```
1: [...]                                   ▷ Same initialization like CDR.
2: d_𝕊 ← ∞                                 ▷ Indicates empty aggregation.
3: while report movement do
4:    s_R ← sense position
5:    while |𝕊| > 0 and φ(peek(𝕊)) · (s_R.t − π_O.t) ≥ ε do
6:       pop(𝕊)                            ▷ Remove root of heap.
7:    end while
8:    if LDR causes update
         or ∃ s_i ∈ 𝕊 : |s_i − π_O s_R(s_i.t)| > ε − δ
         or |s_R.p − π(s_R.t)| > d_𝕊 · (s_R.t − π_O.t) then
9:       π_O ← s_R'
10:      π_V ← compute new predicted velocity ...
11:      send update message (π_O, π_V) to MOD
12:      𝕊 ← {}                            ▷ Clear the sensing history.
13:      d_𝕊 ← ∞                           ▷ Reset the bound.
14:   end if
15:   if |𝕊| = m and φ(s_R) < d_𝕊 then
16:      d_𝕊 ← φ(pop(𝕊))                   ▷ Aggregate the root.
17:   end if
18:   if φ(s_R) < d_𝕊 then
19:      insert s_R into 𝕊
20:   end if
21:   s_R' ← s_R
22: end while
23: send final update message (s_R) to MOD
```

Fig. 6 $\mathrm{CDR_m}$ algorithm.

removing the root of $\mathbb{S}$. It does not need to determine the minimum of $\varphi(\mathrm{pop}(\mathbb{S}))$ and $d_{\mathbb{S}}$ explicitly.

By reconsidering inequation (2), it can be seen that $\kappa(s_i) = \pi_{\mathrm{O}}.t + \epsilon/\varphi(s_i)$. Therefore, the max-heap order by $\varphi(s_i)$ is identical to the min-heap order of CDR by $\kappa(s_i)$. Moreover, CDR's condition $s_{\mathrm{R}}.t \geq \kappa(\mathrm{peek}(\mathbb{S}))$ for removing a s_i from $\mathbb{S}$ can be rewritten as $\varphi(\mathrm{peek}(\mathbb{S})) \cdot (s_{\mathrm{R}}.t - \pi_{\mathrm{O}}.t) \geq \epsilon$.

From an algorithmic perspective, $\mathrm{CDR_m}$ is an extension of CDR. Figure 6 gives its pseudocode. The additional statements compared to CDR are:

- *Lines 2 and 13:* Initialize or reset $d_{\mathbb{S}}$, respectively.
- *Lines 15 to 17:* Remove the sensed position with maximum $\varphi(s_i)$ from $\mathbb{S}$ and aggregate it in $d_{\mathbb{S}}$ if $|\mathbb{S}| = m$ and s_{R} has to be added to $\mathbb{S}$.
- *Lines 18 to 20:* Insert the most recent sensed position s_{R} into the heap $\mathbb{S}$ if $\varphi(s_{\mathrm{R}}) < d_{\mathbb{S}}$.

5 Generic Remote Trajectory Simplification

LDRH and CDR use dead reckoning for two different problems, namely the tracking of the current position and the simplification of the past trajectory. While this leads to simple solutions, the efficiency of simplification depends on the quality of dead reckoning, which has been designed for position tracking only.

On the other hand, there exists a variety of efficient line simplification algorithms that could be used for this purpose. Therefore, it is a good idea to separate position tracking from simplification issues as far as possible to gain flexibility.

In this section, we propose the *Generic Remote Trajectory Simplification* (GRTS) protocol, which clearly separates tracking of the current position from simplification of the past trajectory and can be combined with any line simplification algorithm suited for trajectories.

First, we present the basic protocol and algorithm. Then, we discuss how to bound the space consumption and computing time for line simplification in GRTS and how to consider dynamic sensing deviations δ, resulting for instance from variable sensor inaccuracies. Finally, we present two realizations of GRTS, with the optimal line simplification algorithm by Imai and Iri [11] as well as with an efficient simplification heuristic [18] referred to as segment heuristic.

5.1 Basic Protocol and Algorithm

Although it is a good idea to separate position tracking from simplification issues as far as possible, the simplification process must be synchronized with position tracking to make sure that the simplified data arrives in time at the MOD. The GRTS protocol follows a synchronization pattern, which we call *per-update simplification*.

With this pattern, simplification is performed whenever the position tracking mechanism decides to send an update message. For this purpose, the moving object stores a partial history of sensed positions, which serves as input for the simplification process. Based on this input, the simplification algorithm generates a sequence of vertices for updating the simplified trajectory $\mathbf{u}(t)$, which is included in the update message. In many cases, the generated sequence replaces one or few vertices of $\mathbf{u}(t)$ only – without increasing their number. Therefore, GRTS has better reduction efficiency than LDRH and CDR, which always generate one additional vertex per update.

Depending on the line simplification algorithm used with GRTS, the simplification process may be prepared with each sensed position, to reduce the computing time for line simplification when the position tracking mechanism decides to send an update message. If GRTS is realized with an online algorithm, the simplification even can be performed with each sensing operation – resulting in *per-sense simplification* – as explained in Section 5.5.

In the following, we consider LDR for position tracking in GRTS as LDR is the most efficient, general applicable position tracking protocol [31,17,30]. However,

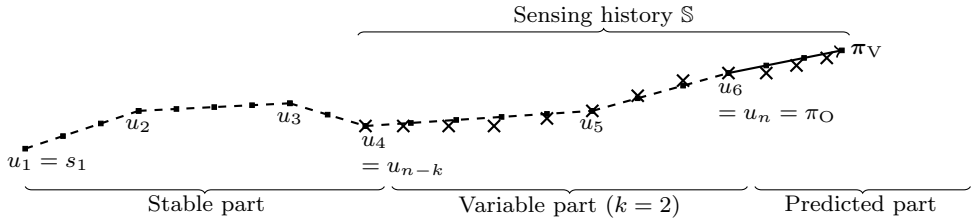

Fig. 7 Three parts of $\mathbf{u}(t)$ and corresponding $\mathbb{S}$ for GRTS.

GRTS can be realized with any position tracking protocol based on a piecewise linear prediction function $\boldsymbol{\pi}(t)$.

GRTS divides the simplified trajectory $\mathbf{u}(t)$ into three parts, as depicted in Figure 7:

1. *Stable part:* This part generally comprises a large number of vertices, stored by the MOD only – except for the last vertex denoted by u_{n-k}, which is also known to the moving object, as explained below.
2. *Variable part:* It generally comprises few vertices only, counted by k, which are known to the MOD *and* the moving object. The last vertex u_n also composes the prediction origin for the next part.
3. *Predicted part:* This part is given by the prediction function $\boldsymbol{\pi}(t)$, i.e., the origin u_n and the vector $\boldsymbol{\pi}_\mathrm{V}$, and is known to the MOD and the moving object.

Hence, n gives the number of vertices of $\mathbf{u}(t)$, excluding the current predicted position $\boldsymbol{\pi}(t_\mathrm{C})$.

The moving object not only stores the vertices of the variable part and the predicted velocity but also the *sensing history* $\mathbb{S}$ for those two parts. Note that $\mathbb{S}$ further includes the last sensed position of the stable part – i.e., the vertex u_{n-k} – as required for simplifying $\mathbf{s}(t)$ for $t > u_{n-k}.t$. Formally, $\mathbb{S}$ is the sequence of chronologically ordered sensed positions with $\mathrm{first}(\mathbb{S}) = u_{n-k}$ and $\mathrm{last}(\mathbb{S}) = s_\mathrm{R}$.

Only the moving object distinguishes between stable and variable part; the MOD does not need to be aware of this differentiation. Once the moving object decides that a vertex u_i belongs to the stable part, this vertex will not be changed by future updates. Hence, the stable part grows in an append-only fashion. The variable part, in contrast, may be changed by future updates. An update message therefore consists of three elements:

1. The number of vertices to remove from $\mathbf{u}(t)$, starting backwards at u_n.
2. The new vertices $\mathbb{U} := (u_j, \ldots, u_n)$ to append to $\mathbf{u}(t)$. In sum, this may increase but (rarely) also decrease the overall number of vertices.
3. The new velocity vector $\boldsymbol{\pi}_\mathrm{V}$, which replaces the previous prediction.

```
1: s_R ← sense position                          ▷ Most recent sensed position.
2: 𝕌 ← (s_R)                                     ▷ New vertices for u(t).
3: π_V ← 0                                       ▷ Predicted velocity.
4: send update message (0, 𝕌, π_V) to MOD
5: 𝕊 ← (s_R)                                     ▷ Sensing history.
6: 𝕍 ← ()                                        ▷ Vertices of variable part of u(t).
7: 𝕌 ← ()
8: while report movement do
9:    s_R ← sense position
10:   𝕊 ← 𝕊 ∥ (s_R)                              ▷ Append s_R to sensing history.
11:   if LDR causes update then
12:      𝕌 ← line simplification with bound ε − δ on 𝕊
13:      𝕌 ← 𝕌 \ (first(𝕌))                      ▷ Belongs to stable part.
14:      π_V ← compute new predicted velocity ...
15:      send update message (|𝕍 \ 𝕌|, 𝕌 \ 𝕍, π_V) to MOD
16:      𝕍 ← 𝕌
17:      𝕌 ← ()
18:      if 𝕍 and 𝕊 should be reduced then
19:         𝕍′ ← some prefix of 𝕍 for stable part ...
20:         𝕊 ← (s ∈ 𝕊 | s.t ≥ last(𝕍′).t)
21:         𝕍 ← 𝕍 \ 𝕍′                           ▷ Set new variable part of u(t).
22:      end if
23:   end if
24: end while
25: 𝕌 ← line simplification with bound ε − δ on 𝕊
26: 𝕌 ← 𝕌 \ (first(𝕌))
27: send final update message (|𝕍 \ 𝕌|, 𝕌 \ 𝕍) to MOD
```

Fig. 8 Basic GRTS algorithm.

Figure 8 shows the pseudocode of the basic GRTS algorithm executed by the mobile object. Initially, the moving object transmits its most recent sensed position $s_\mathrm{R} = s_1$ as first vertex u_1 to the MOD (line 4). Thus, u_1 also serves as prediction origin until the next update. Then, the object executes the while loop (lines 8 to 24) as long as it wants to report its trajectory to the MOD.

During each iteration, the object first senses its current position (line 9) and appends it to the sensing history (line 10). Then, it checks whether it has to send an update message to the MOD (line 11).

If so, the object computes a simplified trajectory part for the movement of the variable and predicted part using $\mathbb{S}$. Since the part of $\mathbf{s}(t)$ given by $\mathbb{S}$ does not deviate by more than δ from $\mathbf{a}(t)$, it executes the line simplification algorithm with *simplification bound* $\epsilon - \delta$ and stores the resulting vertices of the simplified trajectory in $\mathbb{U}$ (line 12). Hence, the simplified trajectory

part given by $\mathbb{U}$ approximates $\mathbf{a}(t)$ on $(\text{first}(\mathbb{S}).t, s_\text{R}.t]$ according to ϵ. Note that first($\mathbb{U}$) can be safely removed from $\mathbb{U}$, as it always corresponds to first($\mathbb{S}$) and thus the last vertex u_{n-k} of the stable part. Then, the object computes a new velocity vector for LDR (line 14) and creates a corresponding update message.

To minimize the size of the update message, only those vertices of $\mathbb{U}$ that actually change the variable part are included. For this purpose, the algorithm maintains the vertices of the variable part in an array $\mathbb{V}$ (lines 6 and 16). To create the update message, the number of vertices that have to be removed from the end of the variable part (expressed by $|\mathbb{V} \setminus \mathbb{U}|$) and the new vertices that have to be added to it (expressed by $\mathbb{U} \setminus \mathbb{V}$) are computed. This information is sent together with the new predicted velocity to the MOD (line 15). Then, the object stores the vertices $\mathbb{U}$ as new vertices $\mathbb{V}$ of the variable part (line 16) and clears $\mathbb{U}$ (line 17).

Finally, the object may decide to reduce the size of the variable part by removing some prefix of $\mathbb{V}$ and $\mathbb{S}$, respectively (lines 18 to 22). A possible policy is to limit the size of $\mathbb{V}$ to some given parameter k. We refer to this variant as GRTS_k in the following. However, GRTS_k does not limit the size of the sensing history $\mathbb{S}$, which has important impact on the space consumption and computing time per position fix. Therefore, we propose the variants GRTS_m and GRTS_mc in Section 5.2, limiting $|\mathbb{S}|$ to a given parameter m.

Once the moving object wants to stop reporting its movement, it computes a last simplification of $\mathbb{S}$ (line 25) and sends a corresponding final update message to the MOD (line 27).

An update message $(|\mathbb{V} \setminus \mathbb{U}|, \mathbb{U} \setminus \mathbb{V}, \boldsymbol{\pi}_\text{V})$, received by the MOD is processed as follows: The MOD removes the $|\mathbb{V} \setminus \mathbb{U}|$ last vertices from $\mathbf{u}(t)$, appends the vertices $\mathbb{U} \setminus \mathbb{V}$ to $\mathbf{u}(t)$ and finally replaces the current predicted velocity with the new vector $\boldsymbol{\pi}_\text{V}$.

As indicated above, many updates just replace the last vertex u_n (i.e., the prediction origin) and provide a new $\boldsymbol{\pi}_\text{V}$. If the construction of the update messages is slightly modified such that *every* update message replaces or repeats the last vertex of the previous update, then u_n and $\boldsymbol{\pi}_\text{V}$ need not to be recovered after a crash of the MOD, and can thus be stored in main memory rather than on disk. This approach saves a number of write operations.[2]

[2] One may think of more elaborate approaches, e.g., to store several of the last vertices of $\mathbf{u}(t)$ in main memory only. In return, the update messages must be logged to stable storage, where simultaneous messages of different objects may be written in one operation. To cope with the update load of a large number of moving objects, however, it is inevitable to partition the MOD to multiple servers (e.g., [13]).

```
 1: [...]
 2: while report movement do
 3:    s_R ← sense position
 4:    S ← S ∥ (s_R)                    ▷ Append s_R to sensing history.
 5:    if |S| = m then
 6:       U' ← line simplification with bound ε − δ on S
 7:       U' ← U' \ (first(U'))
 8:       U ← U ∥ (first(U'))
 9:       S ← (s ∈ S | s.t ≥ last(U).t)
10:    end if
11:    if LDR causes update then
12:       U' ← line simplification with bound ε − δ on S
13:       U' ← U' \ (first(U'))
14:       U ← U ∥ U'
15:       π_V ← compute new predicted velocity ...
16:       send update message (|V \ U|, U \ V, π_V) to MOD
17:       V ← U'
18:       U ← ()
19:    end if
20: end while
21: [...]
22: send final update message (|V \ U|, U \ V) to MOD
```

Fig. 9 GRTS_m algorithm.

To determine the object's position at time t', the MOD has to distinguish two cases, similar to CDR:

1. $t' \leq u_n.t$ *:* The MOD calculates $\mathbf{u}(t')$ by linear interpolation between the vertices u_j and u_{j+1} with $u_j.t \leq t' \leq u_{j+1}.t$ as described in Section 2.
2. $t' > u_n.t$ *:* The MOD calculates $\mathbf{u}(t')$ by means of the prediction $\boldsymbol{\pi}(t)$ given by $\pi_\text{O} = u_n$ and $\boldsymbol{\pi}_\text{V}$.

5.2 Space- and Time-bounded Simplification

The basic GRTS algorithm does not define a policy when to reduce the variable part and to what extent. A possible approach is to limit $|\mathbb{V}|$ to some parameter k (e.g., $k = 1$ or 2), as explained in Section 5.1. This variant GRTS_k, however, does not limit the size of the sensing history $\mathbb{S}$, which has important impact on the space consumption and computing time per position fix, depending on the line simplification algorithm used with GRTS.

Therefore, we propose two space- and time-bounded variants named GRTS_m and GRTS_mc in the following, limiting $|\mathbb{S}|$ to a given m (e.g., $m = 100$ or 500). Since the latter variant builds on the former, we first explain GRTS_m and then GRTS_mc.

$GRTS_m$: Figure 9 shows the pseudocode of the GRTS_m algorithm executed by the moving object. The most important difference to the basic algorithm is that $\mathbb{U}$ is created incrementally, each time $|\mathbb{S}|$ reaches m. Thus, $\mathbb{S}$ is not only simplified if LDR causes an update, but also if $|\mathbb{S}| = m$.

For this purpose, $|\mathbb{S}|$ is checked after each sensing operation (line 5). If it reaches m, a simplification $\mathbb{U}'$ for $\mathbb{S}$ is computed (line 6), the first vertex is removed as in the basic algorithm (line 7), and the subsequent vertex of $\mathbb{U}'$ is added to $\mathbb{U}$ (line 8). Thereafter, the sensing history is cleared up to this vertex (line 9). The clearing determines that this vertex is considered to belong to the stable part – although $\mathbb{U}$ is not sent to the MOD until LDR causes an update.

If LDR causes an update, a simplification $\mathbb{U}'$ for the current sensing history is computed and appended to $\mathbb{U}$ (line 12 to 14). Then, $\mathbb{U}$ is sent to the MOD as in the basic algorithm (line 16). Next, $\mathbb{U}'$ is stored as new variable part in $\mathbb{V}$ (line 17), consistent with $\mathbb{S}$. Finally, $\mathbb{U}$ is cleared for the next update (line 18).

$\mathrm{GRTS_m}$ limits the input for line simplification to m sensed positions and thus bounds the space consumption and the computing time per position fix, depending on the line simplification algorithm being used. For example, with the simplification algorithm by Imai and Iri [11] and $m = 500$, it bounds the computing time to 7 ms on a 3 GHz processor – compared to computing times of up to 870 ms with $\mathrm{GRTS_k}$ (cf. Section 7.4).

On the downside, $\mathrm{GRTS_m}$ generates an additional vertex for $\mathbf{u}(t)$ at least every m sensing operations – even if the object stands still for a long period of time $\gg m \cdot T_\mathrm{S}$. This drawback, however, can be alleviated effectively by a compression technique for the sensing history, resulting in the variant $\mathrm{GRTS_{mc}}$.

$GRTS_{mc}$: The fundamental idea of this technique is the following: Once a simplification has been computed for $\mathbb{S}$ because $|\mathbb{S}| = m$, the first simplified line segment is not immediately considered for the stable part of $\mathbf{u}(t)$ – by adding the corresponding sensed position s_b as vertex to $\mathbb{U}$ – but may be *revised* during subsequent simplifications.

For this purpose, the sensed positions between $\mathrm{first}(\mathbb{S})$ and s_b are removed from $\mathbb{S}$, but s_b is kept in $\mathbb{S}$ and extended by an attribute δ that gives the maximum deviation between the removed positions and the line segment $\overline{\mathrm{first}(\mathbb{S})\, s_b}$, i.e.,

$$s_b.\delta := \max_{\mathrm{first}(\mathbb{S}).t < s_i.t < s_b.t} |\overline{\mathrm{first}(\mathbb{S})\, s_b}(s_i.t) - s_i.\mathbf{p}| \ .$$

Then, s_b may be removed from $\mathbb{S}$ during a subsequent simplification if another line segment $\overline{\mathrm{first}(\mathbb{S})\, s_{b+x}}$ that approximates s_b by $\epsilon - (\delta + s_b.\delta)$ can be found. The reason is that the property

$$|\overline{\mathrm{first}(\mathbb{S})\, s_{b+x}}(s_b.t) - s_b.\mathbf{p}| < \epsilon - (\delta + s_b.\delta)$$

guarantees by triangle inequality $\forall\, s_i$ with $\mathrm{first}(\mathbb{S}).t < s_i.t < s_b.t$ that

$$|\overline{\mathrm{first}(\mathbb{S})\, s_{b+x}}(s_i.t) - s_i.\mathbf{p}| < \epsilon - \delta \ .$$

```
1: [...]
2: c' ← 0                          ▷ Counts the compressed positions in S.
3: while report movement do
4:   s_R ← sense position
5:   S ← S ‖ (s_R)                 ▷ Append s_R to sensing history.
6:   if |S| − c' = m − c then
7:     U' ← line simplification with bound ε − δ on S
8:     search first s_a s_b in U' where s_b is not compressed
9:     s_b.δ ← 0
10:    for all s_i ∈ S with s_a.t < s_i.t < s_b.t do
11:      if s_i is compressed then
12:        s_b.δ ← max(s_b.δ, |s_a s_b(s_i.t) − s_i.p| + s_i.δ)
13:        c' = c' − 1             ▷ Because s_i is removed from S.
14:      else                      ▷ Non-compressed position.
15:        s_b.δ ← max(s_b.δ, |s_a s_b(s_i.t) − s_i.p|)
16:      end if
17:      S ← S \ (s_i)             ▷ Reduce S.
18:    end for
19:    c' = c' + 1                 ▷ Because s_b is compressed now.
20:    if c' > c then              ▷ Move compressed position to U.
21:      S ← S \ (first(S))
22:      U ← U ‖ (first(S))
23:      c' = c' − 1
24:    end if
25:  end if
26:  if LDR causes update then
27:    [...]
28:  end if
29: end while
30: [...]
```

Fig. 10 Compression technique in the $\mathrm{GRTS_{mc}}$ algorithm.

Therefore, we refer to such a sensed position s_b as *compressed* position in the following.

For example, assume $m = 500$ and consider a moving object that stands still for a long period of time, with sensing history $\mathbb{S} = (s_{201}, \ldots, s_{699})$. When adding s_{700} to $\mathbb{S}$, it is $|\mathbb{S}| = m$. Since the object is not moving, $\mathbb{S}$ can be approximated by the line segment

$$\overline{s_{201}\, s_{700}} = \overline{\mathrm{first}(\mathbb{S})\, \mathrm{last}(\mathbb{S})} \ .$$

Therefore, $\mathrm{GRTS_{mc}}$ reduces the sensing history to $\mathbb{S} := (s_{201}, s_{700})$ and extends s_{700} by δ with

$$s_{700}.\delta := \max_{201 < i < 700} |\overline{s_{201}\, s_{700}}(s_i.t) - s_i.\mathbf{p}| \ ,$$

which is about zero in case of standstill. After another 488 position fixes, $|\mathbb{S}|$ reaches m again. Assuming that the object also stood still during $[s_{700}.t, s_{1188}.t]$, the sensing history $\mathbb{S} = (s_{201}, s_{700}, s_{701}, \ldots, s_{1188})$ can be approximated by $\overline{s_{201}\, s_{1188}} = \overline{\mathrm{first}(\mathbb{S})\, \mathrm{last}(\mathbb{S})}$ since

$$|\overline{s_{201}\, s_{1188}}(s_{700}.t) - s_{700}.\mathbf{p}| \approx 0 \leq \epsilon - (\delta + s_{700}.\delta) \approx \epsilon - \delta \ .$$

Hence, the compressed position s_{700} is removed from $\mathbb{S}$ and not added to $\mathbb{U}$. The sensing history is reduced to

$\mathbb{S} := (s_{201}, s_{1188})$, where s_{1188} is a compressed position. For computing $s_{1188}.\delta$ not only

$$\max_{700<i<1188} |\overline{s_{201}\, s_{1188}}(s_i.t) - s_i.\mathbf{p}|$$

has to be taken into account, but also

$$|\overline{s_{201}\, s_{1188}}(s_{700}.t) - s_{700}.\mathbf{p}| + s_{700}.\delta \ .$$

Figure 10 shows the additional pseudocode for GRTS_{mc} compared to the GRTS_{m} algorithm. The variable c' counts the compressed positions in $\mathbb{S}$ (line 2). The compressed positions are always at the beginning of $\mathbb{S}$, right after the last vertex of the stable part, either known to the MOD or stored in $\mathbb{U}$ for the next update. Note that first($\mathbb{S}$) can be considered as non-compressed position, even if it was compressed during previous iterations, as the movement before first($\mathbb{S}$).t is no more relevant for simplification.

It is very unlikely that two or more compressed positions can be spanned by a line segment during future simplifications. To prevent that $\mathbb{S}$ gets filled with compressed positions, causing frequent but ineffective simplifications, their number should be kept small. Therefore, GRTS_{mc} moves the first compressed position to $\mathbb{U}$ once c' exceeds a certain number c (e.g., $c = 1$ or 2).

If the number of non-compressed sensed positions in $\mathbb{S}$ exceeds $m - c$ (line 6), GRTS_{mc} computes a simplification for $\mathbb{S}$, taking the δ-values of the compressed positions into account (line 7). Then, it searches for the first simplified line segment $\overline{s_a\, s_b}$ (i.e., consecutive vertices s_a and s_b in $\mathbb{U}$) where s_b is *not* compressed (line 8). Note that $c' = 0$ implies $s_a = \text{first}(\mathbb{S})$, as in the above example.

GRTS_{mc} compresses s_b by computing $s_b.\delta$ and removes the s_i spanned by $\overline{s_a\, s_b}$ from $\mathbb{S}$ (lines 9 to 18). During this computation, c' is decreased if $\overline{s_a\, s_b}$ spans another compressed position, i.e., if s_a is not the last compressed position in $\mathbb{S}$.

Since s_b is compressed now, c' is increased by one (line 19) and thus may exceed c. If so, the first compressed position (i.e., the second element of $\mathbb{S}$) is added to $\mathbb{U}$ and first($\mathbb{S}$) is removed from $\mathbb{S}$ – such that first($\mathbb{S}$) = last($\mathbb{U}$) as in the GRTS_{m} algorithm (lines 21 to 23).

For realizations with the optimal simplification algorithm by Imai and Iri [11], our evaluation results in Section 7.2 show that the proposed compression technique significantly improves the reduction performance compared to GRTS_{m}. Note that the compression technique can be generally used to optimize real-time online simplification of polylines.

5.3 Time-dependent Maximum Sensing Deviation

The GRTS_{mc} algorithm introduced the δ-attribute to represent the deviation along a simplified line segment $\overline{s_a\, s_b}$ at the end vertex s_b. This idea can be generalized to represent time-dependent maximum sensing deviations $\delta(t)$, in particular to incorporate varying sensor inaccuracies $\sigma(t)$.

For this purpose, *every* sensed position s_i is extended with an attribute δ that gives the maximum deviation between $\mathbf{a}(t)$ and $\overline{s_{i-1}\, s_i}(t)$, depending on physical movement constraints and inaccuracies of the positioning sensor.

When compressing a sensed position s_b to represent a line segment $\overline{s_a\, s_b}$ in GRTS_{mc}, δ is set to

$$s_b.\delta := \max_{s_a.t<s_i.t\le s_b.t} |\overline{s_a\, s_b}(s_i.t) - s_i.\mathbf{p}| + s_i.\delta \ ,$$

independent whether s_i is compressed or not. Thus, there is no difference between compressed and non-compressed positions, except that the former are counted by c'.

As there exists no global constant δ anymore, the line simplification algorithm is called with ϵ only, but has to account for the individual δ-values of the s_i.

Next, we discuss how to realize GRTS with two different line simplification algorithms and how to include the individual δ-values in these algorithms.

5.4 GRTS with Optimal Line Simplification Algorithm

Here, we describe how to combine GRTS with the optimal simplification algorithm introduced in [11]. Although this algorithm has originally been designed for offline usage, we apply it online following the per-update simplification pattern. Thus, whenever LDR decides to send a new update or the sensing history gets too large (in case of GRTS_{m} and GRTS_{mc}), the algorithm is initiated with input $\mathbb{S}$.

In detail, the algorithm first considers the sensed positions in $\mathbb{S}$ as vertices of an unweighted, directed graph and adds an edge for each pair of sensed positions (s_i, s_{i+x}), where the line segment $\overline{s_i\, s_{i+x}}$ approximates every sensed position $s_j \in \mathbb{S}$ with $i < j < i + x$ by ϵ, taking the global δ or the individual $s_i.\delta$ into account. This particularly applies to every pair (s_i, s_{i+1}).

Second, it computes a shortest path between the first vertex first($\mathbb{S}$) and the last vertex last($\mathbb{S}$) = s_{R}. The vertices $\mathbb{U}'$ of the shortest path compose a simplified trajectory that approximates $\mathbf{a}(t)$ within the time interval $[\text{first}(\mathbb{S}).t, s_{\text{R}}.t]$ by ϵ.

Due to the incremental simplification, induced by the choice of the variable part, the corresponding realizations $\text{GRTS}_{\text{k}}^{\text{Opt}}$, $\text{GRTS}_{\text{m}}^{\text{Opt}}$, and $\text{GRTS}_{\text{mc}}^{\text{Opt}}$ generally

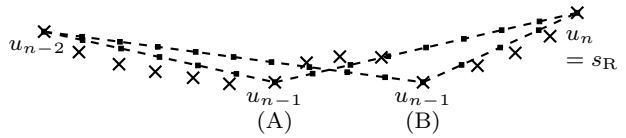

Fig. 11 Two possible simplifications (A) and (B) with minimal number of vertices $\mathbb{U}' = (u_{n-2}, u_{n-1}, u_n)$.

do not achieve the best possible reduction as it would be achieved with the optimal line simplification algorithm being applied offline to the overall sequence of sensed positions.

Certainly, the optimal reduction could be achieved by setting k or m to ∞, i.e., by removing the stable part of $\mathbf{u}(t)$. However, this causes unacceptable computing times and requires very large amounts of space, which can be already seen from the evaluation of $\mathrm{GRTS_k}$ with $k = 1$ and 3 (cf. Section 7.4). Moreover, it may cause very large update messages.

The underlying reason for the suboptimal reduction is that there may exist several simplifications with minimum number of vertices $\mathbb{U}'$ for a given $\mathbb{S}$. Figure 11 illustrates an example of two possible simplifications $\mathbb{U}' = (u_{n-2}, u_{n-1}, u_n)$, implying two possible sequences of vertices to be sent to the MOD. Generally, choosing the simplification with maximum $u_{n-1}.t$ – here (B) – is a good heuristic, as it minimizes the number of sensed positions spanned by the last line segment $\overline{u_{n-1}\, u_n}$, which is likely to be revised by future simplifications. Nevertheless, there may also be cases where choosing another simplification would yield a better overall reduction efficiency.

Note that the construction of the graph can be performed incrementally, after each sensing operation, despite the per-update simplification pattern. Such an implementation reduces the computing time after those sensing operations that cause a simplification.

5.5 GRTS with Segment Heuristic

The segment heuristic is a simple online line simplification algorithm, which has been proposed in various works including [18][3], [2], and [10].

For simplifying a sequence of sensed positions $(s_1, s_2, \ldots)$ by bound ϵ, the segment heuristic works as follows: First, it sets s_1 as vertex u_1 of the simplified trajectory. Then, it iteratively probes the line segments $\overline{s_1\, s_2}, \overline{s_1\, s_3}, \ldots$ until it finds the first segment $\overline{s_1\, s_x}$ that would violate ϵ, i.e., where

$$\exists\, s_i \in (s_1, \ldots, s_x) : |\overline{s_1\, s_x}(s_i.t) - s_i.\mathbf{p}| > \epsilon - \delta$$

or, with individual $s_i.\delta$,

$$\exists\, s_i \in (s_1, \ldots, s_x) : |\overline{s_1\, s_x}(s_i.t) - s_i.\mathbf{p}| + s_i.\delta > \epsilon\ .$$

In this case, the segment heuristic chooses the previous sensed position s_{x-1} as vertex of the simplification. Next, it repeats the above procedure starting at s_{x-1}, and so on.

Since this online algorithm processes the sensed positions iteratively, it allows for *per-sense simplification* by executing the segment heuristic for the most recent sensed position s_R after each sensing operation. The advantage of per-sense simplification is that the simplification is performed as early as possible, resulting in a smaller sensing history $\mathbb{S}$ on average. Moreover, the computing time for line simplification is distributed over all iterations of GRTS.

This property of the segment heuristic, however, is also the reason why it does not exploit variable parts consisting of more than one line segment. When realizing $\mathrm{GRTS_k}$ with the segment heuristic the parameter k should be fixed to 1 therefore.

Figure 12 shows the corresponding pseudocode of $\mathrm{GRTS_k^{Sec}}$. For each sensed position s_R, the algorithm checks whether the line segment $\overline{\mathrm{first}(\mathbb{S})\, s_\mathrm{R}}$ approximates the sensed positions in-between by simplification bound ϵ and $\delta(t)$ or not (line 4). If not, it appends the last sensed position – the one before s_R – to $\mathbb{U}$ (line 5) and reduces $\mathbb{S}$ accordingly (line 6). When LDR causes a new update, the most recent sensed position is simply appended to $\mathbb{U}$ (line 10) and a corresponding update message is sent to the MOD (line 12). Finally, $\mathrm{GRTS_k^{Sec}}$ sets $\mathbb{V}$ to $(\mathrm{last}(\mathbb{U}))$, consistent with $\mathbb{S}$, and clears $\mathbb{U}$ for the next update.

To limit the size of $\mathbb{S}$ to some parameter m, the simplification condition (line 4) can be extended by

or $|\mathbb{S}| = m - 1$

resulting in $\mathrm{GRTS_m^{Sec}}$. Adding the compression technique described in Section 5.2 finally results in $\mathrm{GRTS_{mc}^{Sec}}$. Analogous to the parameter k of $\mathrm{GRTS_k^{Sec}}$, the parameter c of $\mathrm{GRTS_{mc}^{Sec}}$ should be fixed to 1, as the segment heuristic never exploits more than one compressed position for simplification.

Optimization of $\mathbb{S}$*:* The average size of $\mathbb{S}$ can be further reduced by a novel optimization of the segment heuristic, independent of whether $|\mathbb{S}|$ is bounded to some m or not. This optimization particularly supersedes the compression technique of $\mathrm{GRTS_{mc}}$, i.e., $\mathrm{GRTS_m^{Sec}}$ achieves

[3] The authors of [18] refer to the segment heuristic as Opening-Window algorithm (OPW) and distinguish two variants with different distance metrics. The one with the better reduction efficiency, which corresponds to the segment heuristic as explained here, is called BOPW-TR.

```
1: [...]
2: while report movement do
3:    s_R ← sense position
4:    if ∃ s_i ∈ 𝕊 : |first(𝕊) s_R(s_i.t) − s_i.p| + s_i.δ > ε then
5:       𝕌 ← 𝕌 ∥ (last(𝕊))
6:       𝕊 ← (last(𝕊))
7:    end if
8:    𝕊 ← 𝕊 ∥ (s_R)                    ▷ Append s_R to sensing history.
9:    if LDR causes update then
10:      𝕌 ← 𝕌 ∥ (last(𝕊))                ▷ Append s_R as u_n.
11:      π_V ← compute new predicted velocity ...
12:      send update message (|𝕍 \ 𝕌|, 𝕌 \ 𝕍, π_V) to MOD
13:      𝕍 ← (last(𝕌))        ▷ |𝕍| > 1 would not be exploited.
14:      𝕌 ← ()
15:   end if
16: end while
17: [...]
```

Fig. 12 $\mathrm{GRTS}^{\mathrm{Sec}}_{\mathrm{k}}$ algorithm with per-sense simplification and fixed $k = 1$.

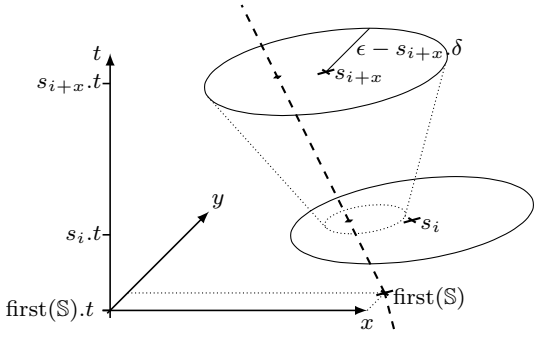

Fig. 13 Geometric illustration for optimization of the segment heuristic.

the same reduction as $\mathrm{GRTS}^{\mathrm{Sec}}_{\mathrm{mc}}$ with this optimization (cf. Section 7.2).

The basic idea of the optimization is the following: Each sensed position $s_i \in \mathbb{S}$ poses a constraint on the next line segment $\overline{\mathrm{first}(\mathbb{S})\, u_n}$ that is going to approximate $\mathbb{S}$. If the constraint given by another sensed position s_{i+x} completely encloses the one given by s_i, then s_i can be removed from $\mathbb{S}$ without affecting the simplification. In our evaluations (cf. Section 7), this reduces the space consumption of the segment heuristic by two-thirds on average.

The constraint defined by a $s_i \in \mathbb{S}$ is that the distance $|\overline{\mathrm{first}(\mathbb{S})\, u_n}(s_i.t) - s_i.\mathbf{p}| + s_i.\delta$ must not exceed ϵ. This constraint is checked for every potential line segment $\overline{\mathrm{first}(\mathbb{S})\, s_{\mathrm{R}}}$ (line 4 in Figure 12). Geometrically, for each s_i, the line segment has to pass the circle with center $s_i.\mathbf{p}$ and radius $\epsilon - s_i.\delta$ at time $s_i.t$ as illustrated in Figure 13. Since the line segment's first vertex is known, the circles of two sensed positions s_i and s_{i+x} can be normalized regarding time and compared with each other: The circle of s_{i+x} poses the same constraint like the circle with center $\overline{\mathrm{first}(\mathbb{S})\, s_{i+x}}(s_i.t)$ and radius $(\epsilon - s_{i+x}.\delta)\ \frac{s_i.t - \mathrm{first}(\mathbb{S}).t}{s_{i+x}.t - \mathrm{first}(\mathbb{S}).t}$ at time $s_i.t$. Now, if this circle is contained by the circle of s_i, as pictured in Figure 13, then s_i can be removed from $\mathbb{S}$. Thus, for each sensed position s_{R}, the realizations $\mathrm{GRTS}^{\mathrm{Sec}}_{\mathrm{k}}$, $\mathrm{GRTS}^{\mathrm{Sec}}_{\mathrm{m}}$, and $\mathrm{GRTS}^{\mathrm{Sec}}_{\mathrm{mc}}$ can remove every s_i from $\mathbb{S}$ whose circle contains the normalized circle of s_{R} at $s_i.t$, except $s_i = \mathrm{first}(\mathbb{S})$. In Figure 12 this removal should be included between the lines 7 and 8.

6 Acceleration-based Movement Constraints

The movement between two sensing operations is bounded by physical constraints such as the maximum speed or acceleration, as explained in Section 2. The corresponding values are factored into δ for trajectory simplification and into the update condition of LDR. The smaller δ, the higher is the possible reduction since δ is to be subtracted from ϵ.

In the previous sections, we exemplarily considered a given maximum speed $v_{\max}$ for simplicity and readability. Yet, for fast objects such as cars or airliners, $v_{\max}$ results in very large values of δ and thus causes low reductions. The reason is that $v_{\max}$ only provides a coarse estimate of the actual movement constraints of such objects. Consider, for example, a car with $v_{\max} = 50\,\mathrm{m/s}$ and $T_{\mathrm{S}} = 1\,\mathrm{s}$. According to the resulting speed-based movement constraint, the car is assumed to be able to travel 25 m and then return to the starting point within one second. This is obviously unrealistic, as it requires the car to accelerate (and decelerate) with at least $200\,\mathrm{m/s^2}$.

Next, we therefore explain how to take an object's maximum acceleration $a_{\max}$ into account. First, we discuss how to compute the maximum sensing deviation δ accordingly. Then, we consider the update condition of LDR.

Given an object with maximum speed $v_{\max}$ and two sensed positions s_{i-1} and s_i, we concluded in Section 2 that the object cannot deviate by more than $v_{\max} \cdot T_{\mathrm{S}}/2$ from the line segment $\overline{s_{i-1}\, s_i}$.[4] For $a_{\max}$ we analogously conclude that the object cannot deviate by more than $\frac{1}{2} a_{\max} \left(\frac{T_{\mathrm{S}}}{2}\right)^2$ from $\overline{s_{i-1}\, s_i}$ using linear kinematics. Together with the sensor inaccuracy σ, it follows $\delta := \sigma + \frac{1}{8} a_{\max} T_{\mathrm{S}}^2$.

[4] In fact, the maximum possible deviation from $\overline{s_{i-1}\, s_i}$ depends on the object's current speed – and thus the distance between $s_{i-1}.\mathbf{p}$ and $s_i.\mathbf{p}$ – since the current speed limits the speed for other velocity components to deviate from $\overline{s_{i-1}\, s_i}(t)$. Therefore, δ may depend on time, even if σ is fixed. This is discussed in detail in [22]. Yet, for significant improvements regarding δ, the object's speed has to be close to $v_{\max}$. Therefore, we neglect this optimization here and focus on $a_{\max}$.

	Human	Car	Airliner
$v_{\max}$	12.0	50.0	270.0 m/s
$a_{\max}$	5.0	10.0	20.0 m/s^2
speed-based δ	13.8	32.8	142.8 m
acceleration-based δ	8.4	9.1	10.3 m

Table 3 Values of δ for typical speed- and acceleration-based movement constraints.

	Human	Car	Airliner
$v_{\max}$	12.0	50.0	270.0 m/s
$a_{\max}$	5.0	10.0	20.0 m/s^2
speed-based offset	22.2	67.8	331.8 m
acceleration-based offset	16.3	22.9	36.1 m

Table 4 Offsets to ϵ for LDR for typical speed- and acceleration-based movement constraints.

Table 3 shows the corresponding values of δ for three typical scenarios of $v_{\max}$ and $a_{\max}$, assuming $\sigma = 7.8\,\text{m}$ and $T_\text{S} = 1\,\text{s}$. The scenarios support the argumentation that the ratio between the speed-based value of δ and acceleration-based value increases with the typical speed of the objects.

Incorporating $a_{\max}$ into the update condition of LDR is more complex. For $v_{\max}$ we showed in Section 3 that the object has to send an update if

$$|s_\text{R}.\mathbf{p} - \boldsymbol{\pi}(s_\text{R}.t + T_\text{S} + T_\text{U})| + \sigma + v_{\max}(T_\text{S} + T_\text{U}) > \epsilon \ ,$$

as $s_\text{R}.\mathbf{p}$ the object may move by up to $v_{\max}(T_\text{S} + T_\text{U})$ until an update sent after the subsequent sensing operation would have been processed.

For acceleration-based movement constraints, however, the current velocity has to be taken into account, to be able to estimate the possible deviation between the object's movement and $\boldsymbol{\pi}(t)$ at $s_\text{R}.t+T_\text{S}+T_\text{U}$. Thus, we require an approximation for the *most recent velocity* $\mathbf{v}_\text{R}$ at $s_\text{R}.t$. An obvious solution is to use the average velocity between $s_\text{R}.t$ and the second last sensed position $s_{\text{R}'}$, i.e.,

$$\mathbf{v}_{\overline{\text{R}'\,\text{R}}} := \frac{s_\text{R}.\mathbf{p} - s_{\text{R}'}.\mathbf{p}}{s_\text{R}.t - s_{\text{R}'}.t} = \frac{s_\text{R}.\mathbf{p} - s_{\text{R}'}.\mathbf{p}}{T_\text{S}} \ .$$

The velocity $\mathbf{v}_{\overline{\text{R}'\,\text{R}}}$ is subject to two approximation errors: First, the object may accelerate (or decelerate) during $[s_{\text{R}'}.t, s_\text{R}.t]$ – also sidewards – causing an error of up to $a_{\max} \cdot T_\text{S}/2$. Second, $\mathbf{v}_{\overline{\text{R}'\,\text{R}}}$ is subject to sensor inaccuracies, causing an error of up to $2\sigma/T_\text{S}$.

We can distinguish between systematic, time-correlated inaccuracies σ_sys and random, noise-like inaccuracies σ_noise. For example with GPS, the former are caused by inaccurate ephemeris data and atmospheric effects amongst others, while the latter are caused by the receiver hardware and make only about 10% of the overall sensor inaccuracy σ [25,19,32]. Since $\mathbf{v}_{\overline{\text{R}'\,\text{R}}}$ is computed by two consecutive position fixes, it can be considered to be subject to σ_noise only. In sum, $\mathbf{v}_{\overline{\text{R}'\,\text{R}}}$ may deviate from the actual velocity $\mathbf{v}_\text{R}$ by up to $a_{\max}\frac{T_\text{S}}{2} + \frac{2\sigma_\text{noise}}{T_\text{S}}$.

Taking the inaccuracy of $s_\text{R}.\mathbf{p}$ and the possible movement and acceleration during $T_\text{S}+T_\text{U}$ into account, LDR has to cause an update if

$$\left| s_\text{R}.\mathbf{p} + \frac{s_\text{R}.\mathbf{p} - s_{\text{R}'}.\mathbf{p}}{T_\text{S}}(T_\text{S} + T_\text{U}) - \boldsymbol{\pi}(s_\text{R}.t + T_\text{S} + T_\text{U}) \right| + \sigma + \frac{1}{2}a_{\max}(T_\text{S} + T_\text{U})^2 + \left(a_{\max}\frac{T_\text{S}}{2} + \frac{2\sigma_\text{noise}}{T_\text{S}} \right)(T_\text{S} + T_\text{U}) > \epsilon.$$

The different inaccuracies and the possible movement and acceleration during $T_\text{S} + T_\text{U}$ can be considered as offset to ϵ similar to δ. Table 4 shows the corresponding values for three typical scenarios of $v_{\max}$ and $a_{\max}$, assuming $\sigma = 7.8\,\text{m}$, $\sigma_\text{noise} = 0.1\sigma$, and $T_\text{S} = 1\,\text{s}$. Note that the offsets are significantly greater than the corresponding δ values in Table 3, as the latter refer to the movement between two given positions s_{i-1} and s_i.

This property is another reason (in addition to the separation of simplification from position tracking) for the significant difference between the number of updates caused by LDR and the number of vertices generated by GRTS or offline simplification – in particular for small values of ϵ. For instance, for $\epsilon = 25\,\text{m}$, LDR causes more than 1000 updates per hour, while $\text{GRTS}_\text{m}^\text{Sec}$ with $m = 500$ generates only about 65 vertices. A possible countermeasure is to relax the real-time constraint of trajectory tracking by introducing some temporal tolerance, which has to be subtracted from $T_\text{S} + T_\text{U}$ in the above formulas. Note that such a temporal tolerance can only be introduced if position tracking is clearly separated from simplification as with GRTS.

If $v_{\max}$ and $a_{\max}$ are both given, the two kinds of movement constraints can be considered simultaneously, simply by choosing the smaller δ value and offset for the update condition of LDR.

Furthermore it would be possible to distinguish different directions of acceleration, such as real speed-up in forward direction, deceleration by braking, and sidewards or angular acceleration by steering.

7 Evaluation

For significant results on the performance of CDR and GRTS, we simulated the different variants and realizations with hundreds of real GPS traces and compared

them to LDRH as well as to offline simplification. For practical experiences, we also conducted experiments with a prototypical implementation of GRTS.

In this section, we first describe the simulation setup and then give the results on reduction efficiency, communication, and computational costs. Thereafter, we report on our experiences with the prototypical implementation. Based on these results, we finally draw conclusions for the selection of a concrete tracking approach for a given application scenario.

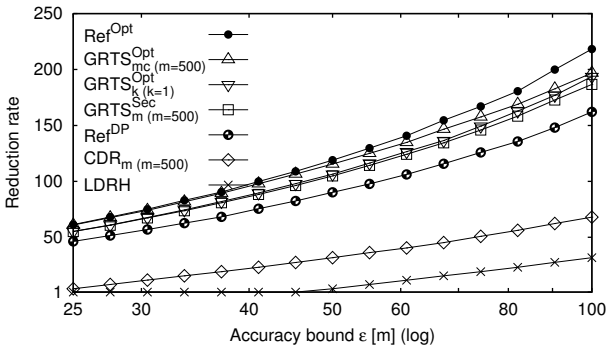

Fig. 14 Reduction rates of major real-time trajectory tracking and offline simplification approaches.

7.1 Setup

We implemented a simulation software for CDR, $\mathrm{CDR_m}$, $\mathrm{GRTS_k^{Opt}}$, $\mathrm{GRTS_m^{Opt}}$, $\mathrm{GRTS_{mc}^{Opt}}$, $\mathrm{GRTS_k^{Sec}}$, $\mathrm{GRTS_m^{Sec}}$, $\mathrm{GRTS_{mc}^{Sec}}$, and the existing trajectory tracking approach LDRH [27] as well as the optimal line simplification algorithm ($\mathrm{Ref^{Opt}}$) by Imai and Iri [11] and the Douglas-Peucker algorithm ($\mathrm{Ref^{DP}}$) [6] in the C programming language. We selected $\mathrm{Ref^{Opt}}$ as a reference for comparing our results to the best possible reduction, while $\mathrm{Ref^{DP}}$ is a commonly used offline heuristic.

For simulating these algorithms with realistic data, we downloaded several thousand GPS traces (i.e., trajectories sensed by customary GPS receivers) from the OpenStreetMap project [21]. In several processing steps, we filtered those traces that provide an individual position fix for each second – and thus have not undergone any previous data reduction – and that could be classified clearly according to their means of transportation into foot, bicycle, and motor vehicle. For classifying a trajectory, we not only relied upon its speed characteristics but also on representative tags specified on the OpenStreetMap website.

Then, we simulated the execution of the trajectory tracking approaches by sequentially feeding the algorithms with the recorded positions given in the GPS traces. For each variant and realization, we measured the number of vertices of the resulting simplified trajectories, the numbers of update messages, and the amounts of transmitted data, depending on ϵ varied from 25 to 100 m. Furthermore, we measured the space consumption and computing time per position fix.

We used a sensing period of $T_\mathrm{S} = 1\,\mathrm{s}$ and a sensor inaccuracy of $\sigma = 7.8\,\mathrm{m}$ with $\sigma_\mathrm{noise} = 0.1\sigma$ in accordance with the GPS traces and the inaccuracies reported in [28,32]. We further assumed an update time of $T_\mathrm{U} = 0.2\,\mathrm{s}$. If not stated otherwise, we considered the acceleration-based movement constraint by $a_\mathrm{max} = 10\,\mathrm{m/s^2}$, which gives $\delta = 9.1\,\mathrm{m}$ and an offset of 22.9 m in the update condition of LDR.

In addition, we applied the offline algorithms $\mathrm{Ref^{Opt}}$ and $\mathrm{Ref^{DP}}$ with bound $\epsilon - \delta$ to each GPS trace and measured the number of vertices of the resulting simplified trajectories.

All experiments were performed on 3 GHz Intel Xeon Linux servers using 2 GB RAM.

The different speeds of the means of transportation do not yield any significant differences when comparing the different approaches with each other, but only when considering the absolute values for reduction efficiency and communication. Therefore, we give the average results of the 3×100 largest GPS traces of the three means of transportation and refer to the individual means of transportation and speeds where necessary. Each of the 300 trajectories comprises 1400 to 16500 GPS positions, i.e., spans about 20 min to 5 h.

7.2 Reduction Efficiency

The reduction efficiency of trajectory simplification is measured by the *reduction rate* defined as the number of sensed positions divided by the number of vertices of the simplified trajectory $\mathbf{u}(t)$, i.e., $|(s_1,\ldots,s_l)|/|(u_1,\ldots,u_n)|$.

Figure 14 shows the reduction rates of the major tracking approaches and the two reference offline algorithms $\mathrm{Ref^{Opt}}$ and $\mathrm{Ref^{DP}}$. As discussed below, the reduction rate of $\mathrm{CDR_m}$ with $m = 500$ is equal to the reduction rate of CDR. Similarly, the reduction rate of $\mathrm{GRTS_m^{Sec}}$ with $m = 500$ is equal to the reduction rate of $\mathrm{GRTS_{mc}^{Sec}}$ and $\mathrm{GRTS_k^{Sec}}$, where the latter can be considered as $\mathrm{GRTS_m^{Sec}}$ with $m = \infty$. For $\mathrm{GRTS_{mc}}$ we used $c = 1$ if not stated otherwise. Below, we show that $c > 1$ does not give any improvement.

The reduction rate of CDR (or $\mathrm{CDR_m}$ with $m = 500$) is at least twice the reduction rate of LDRH, consistent with the analysis in Section 3. For $\epsilon < 45\,\mathrm{m}$, LDRH does not perform any reduction since its inter-

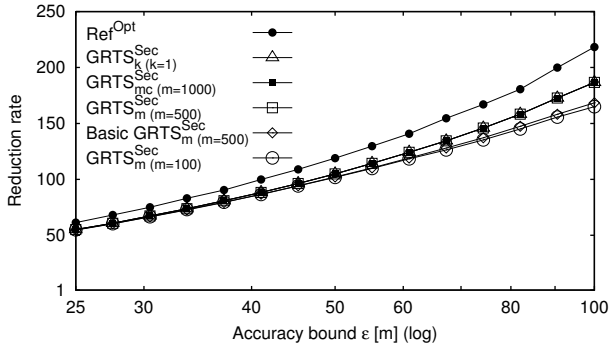

Fig. 15 Reduction rates of $\mathrm{GRTS}_{\mathrm{k}}^{\mathrm{Sec}}$, $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$, and $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Sec}}$.

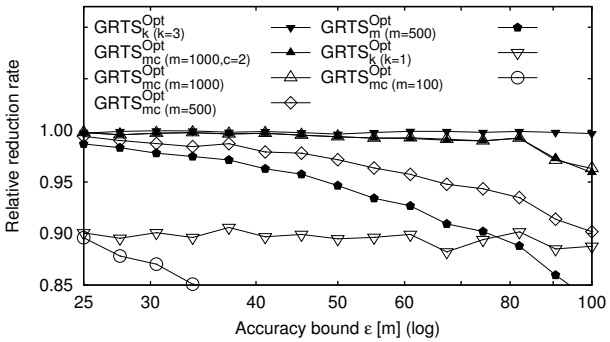

Fig. 16 Reduction by $\mathrm{GRTS}_{\mathrm{k}}^{\mathrm{Opt}}$, $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Opt}}$, $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$ relative to $\mathrm{Ref}^{\mathrm{Opt}}$.

nal accuracy bound $\epsilon' := \epsilon/2$ becomes smaller than the offset given by the sensor inaccuracy and the movement constraint in the update condition of LDR. Thus, LDR causes an update after each sensing operation.[5]

All GRTS realizations given in Figure 14 outperform the CDR variants by at least factor 2.7 and LDRH by about factor 5.5. This confirms the importance of separating tracking of the current position from simplification of the past trajectory.

The reduction rates of the GRTS realizations are always less than 15% below the best possible reduction by the optimal algorithm $\mathrm{Ref}^{\mathrm{Opt}}$. Moreover, the GRTS realizations always outperform $\mathrm{Ref}^{\mathrm{DP}}$. In detail, the reduction rate of $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$ is 15 to 19% greater than the reduction rate of $\mathrm{Ref}^{\mathrm{DP}}$, whereas $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$ even achieves up to 32%. This is a surprising result given the fact that $\mathrm{Ref}^{\mathrm{DP}}$ is performed offline on the entire GPS traces.

The reduction rate of $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$ is always 1 to 4% below the reduction rate of $\mathrm{GRTS}_{\mathrm{k}}^{\mathrm{Opt}}$ with $k = 1$. The reason is that the variable part of $\mathbf{u}(t)$ comprises only one vertex in all realizations with the segment heuristic – independent of m or k.

Figure 15 details the reduction performance of these realizations, where *Basic* $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$ refers to the realization without the optimization of the segment heuristic proposed in Section 5.5. It shows that $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$ with $m = 500$ suffices to achieve the best reduction rate that is possible with the segment heuristic. Neither the use of $\mathrm{GRTS}_{\mathrm{mc}}$, nor a parameterization of $m > 500$ gives better results. Note again that $\mathrm{GRTS}_{\mathrm{k}}^{\mathrm{Sec}}$ can be considered as $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$ with $m = \infty$.

[5] As mentioned in Section 3, many works do not clearly state whether they account for the sensing period, the update time, and the sensor inaccuracy in the update condition of LDR, or not. In the latter case, the factor between the reduction rates of CDR (or $\mathrm{CDR}_{\mathrm{m}}$ with $m = 500$) and LDRH may decrease to about 1.5.

The figure further reveals the importance of optimization of the segment heuristic in the case where the sensing history $\mathbb{S}$ is bounded by some m. For example, for $m = 500$, it increases the reduction performance by up to 10%. This optimization is also the reason why the compression technique by $\mathrm{GRTS}_{\mathrm{mc}}$ has no effect on the reduction rate of $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Sec}}$ compared to $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$.

Analogously, Figure 16 shows the reduction performance of the GRTS realizations with the optimal simplification algorithm by Imai and Iri [11]. For readability, it shows the relative reduction rate compared to the simplification algorithm being applied offline to the entire GPS traces, i.e., $\mathrm{Ref}^{\mathrm{Opt}}$.

Clearly, the relative reduction rate is always ≤ 1. The rate would be equal to one for $\mathrm{GRTS}_{\mathrm{k}}^{\mathrm{Opt}}$ with $k = \infty$ since there would be not stable part of $\mathbf{u}(t)$. Yet, the simulation results show that $k = 3$ almost suffices, as the average reduction rate is only 0.2% smaller than the best possible reduction rate. For $k = 1$, the difference is already about 10%.

However, as explained below, the computational costs of $\mathrm{GRTS}_{\mathrm{k}}^{\mathrm{Opt}}$ are too high for practical use due to the unbounded size of $\mathbb{S}$. Hence, $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Opt}}$ or $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$ have to be used, where the $\mathrm{GRTS}_{\mathrm{mc}}$ realization effectively outperforms the $\mathrm{GRTS}_{\mathrm{m}}$ realization.

Interestingly, $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$ with $m = 1000$ and $c = 1$ may also achieve more than 99% of the best possible reduction rate. Enabling more than one compressed position by choosing $c > 1$ does not give any improvement, consistent to the discussion in Section 5.2. Even with a small value of $m = 500$, the relative reduction rate of $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$ is less than 3% below the best possible rate – at least for $\epsilon < 50\,\mathrm{m}$.

For larger values of ϵ, the relative reduction rates of $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Opt}}$ and $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$ decrease noticeably. The reason is that the average number of sensed positions spanned by a line segment of the simplified trajectory $\mathbf{u}(t)$ increases with ϵ, whereas $|\mathbb{S}|$ is bounded by m. For large ϵ this problem may be alleviated by increasing

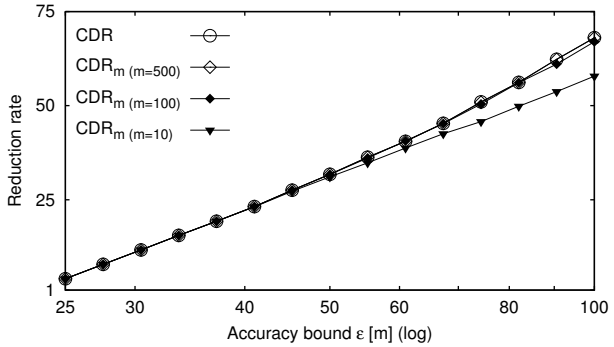

Fig. 17 Reduction rates of CDR and $\mathrm{CDR_m}$ for different m.

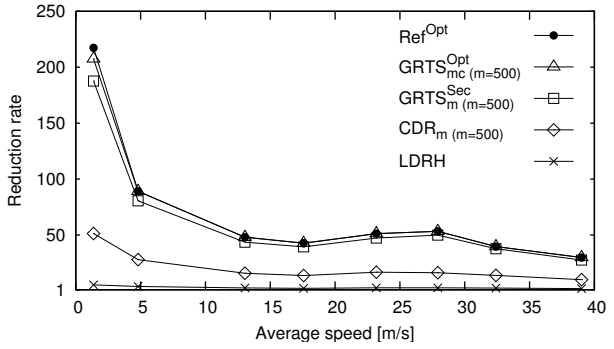

Fig. 18 Reduction rates depending on average speed for $\epsilon = 50\,\mathrm{m}$.

T_S, despite the resultant increase of δ. This is also the reason why we do not give any results for $\epsilon > 100\,\mathrm{m}$.

For the sake of completeness, Figure 17 gives the reduction rates of CDR and $\mathrm{CDR_m}$ for different values of m. As mentioned above, the reduction performance of CDR is equal to the performance of $\mathrm{CDR_m}$ with $m = 500$. However, even with only $m = 10$, $\mathrm{CDR_m}$ achieves a remarkable performance compared to CDR.

All the relative reduction rates similarly apply to the individual means of transportation (foot, bicycle, motor vehicle). For example, for $\epsilon = 50\,\mathrm{m}$, the reduction rate of $\mathrm{GRTS}^{\mathrm{Opt}}_{\mathrm{mc}}$ with $m = 500$ always is at least 95% of the best possible rate.

The absolute reduction rates, however, depend on the means of transportation due to the different ratio between the typical speed and ϵ. For instance, for $\epsilon = 50\,\mathrm{m}$, the reduction rate of $\mathrm{GRTS}^{\mathrm{Opt}}_{\mathrm{mc}}$ ($m = 500$) is 208.1 for pedestrians, 89.0 for bicycles, and 49.5 for motor vehicles.

Figure 18 renders these differences more precisely by showing the reduction rate depending on the speed. For this purpose, we grouped the GPS traces by their average speed and then computed the average reduction rate for each group and approach and parameterization for $\epsilon = 50\,\mathrm{m}$.

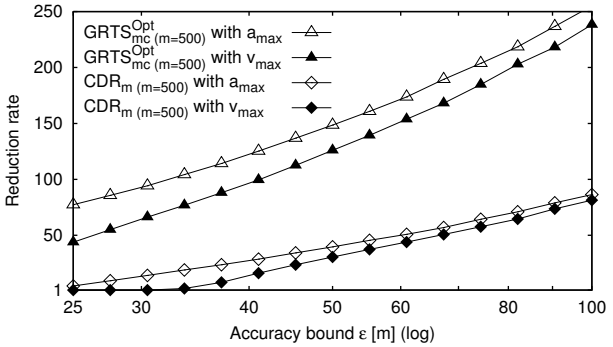

Fig. 19 Reduction rates of $\mathrm{CDR_m}$ and $\mathrm{GRTS}^{\mathrm{Opt}}_{\mathrm{mc}}$ for $v_{\max} = 20\,\mathrm{m/s}$ and $a_{\max} = 10\,\mathrm{m/s^2}$. (Only for GPS traces recorded by foot or bicycle.)

The reduction rates are comparatively high for slow objects, as to be expected. In addition, the rates largely decrease with increasing speed. Exceptions for an average speed of more than 15 m/s result from the fact that the average speed correlates with the kind of road (streets, rural roads, highways), implying different movement characteristics.

Figure 19 shows the reduction rates of $\mathrm{CDR_m}$ and $\mathrm{GRTS}^{\mathrm{Opt}}_{\mathrm{mc}}$ for the acceleration-based movement constraint given by $a_{\max} = 10\,\mathrm{m/s^2}$ and the speed-based movement constraint by $v_{\max} = 20\,\mathrm{m/s}$. The figure only gives the results from GPS traces recorded by foot or bicycle, as $v_{\max} = 20\,\mathrm{m/s}$ does not apply to cars. With GRTS, the use of $a_{\max}$ increases the reduction rate by 7 to 76% compared to $v_{\max}$, depending on ϵ. The larger ϵ, the smaller is the increase, as the difference between the corresponding δ values vanishes in comparison to ϵ.

Similar applies to the CDR variant. With $v_{\max} = 20\,\mathrm{m/s}$, CDR does not perform any simplification for $\epsilon \leq 30\,\mathrm{m}$ since the simplification is based on LDR and the offset in the update condition exceeds ϵ, causing an update after each sensing operation.

7.3 Communication Costs

Figure 20 shows the number of update messages generated by LDRH, $\mathrm{CDR_m}$ ($m = 500$), and GRTS per hour depending on ϵ. It allows verifying that the number of update messages by GRTS is independent on the variant and parameterization of k or m, as it only depends on LDR.

The figure also shows that the number of updates caused by the additional segment condition of CDR and the limitation of $|\mathbb{S}|$ is negligible (less than 1%) compared to the number of updates caused by LDR, consistent with the analysis of LDR in Section 3.

For $\epsilon \leq 45\,\mathrm{m}$, LDRH sends an update after each sensing operation since the internal accuracy

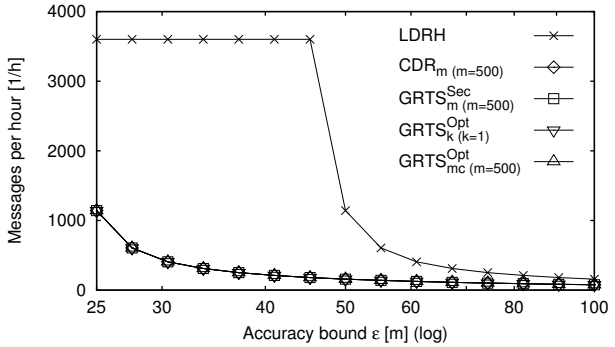

Fig. 20 Update messages sent by major tracking approaches.

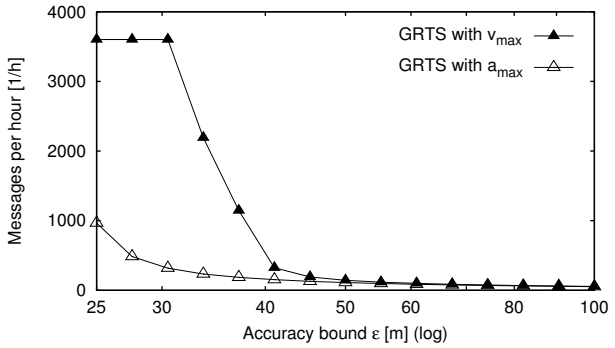

Fig. 21 Update messages sent by GRTS for $v_{\max} = 20\,\mathrm{m/s}$ and $a_{\max} = 10\,\mathrm{m/s^2}$. (Only for GPS traces recorded by foot or bicycle.)

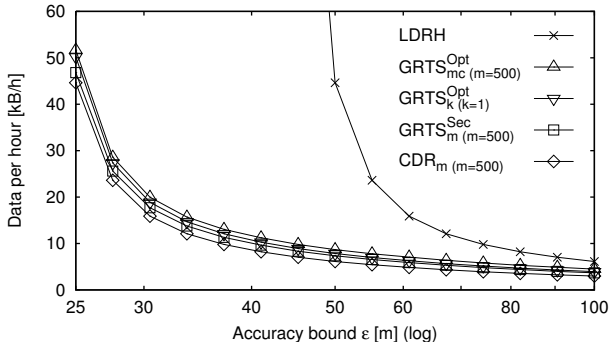

Fig. 22 Amounts of data transmitted by major tracking approaches.

bound $\epsilon' := \epsilon/2$ is smaller than the offset in the update condition given by $a_{\max}$.

This offset is also the reason why GRTS sends more than 1000 updates per hour for $\epsilon = 25\,\mathrm{m}$. This number could by reduced by relaxing the real-time constraint of trajectory tracking, as discussed at the end of Section 6.

This problem is intensified with the speed-based movement constraint given by $v_{\max} = 20\,\mathrm{m/s}$, as depicted in Figure 21, in line with the analytical comparison of speed-based and accuracy-based constraints in Section 6.

The update messages of LDRH and CDR contain only a prediction, where the origin gives a vertex of $\mathbf{u}(t)$. GRTS, in contrast, additionally inserts the number of vertices to remove from the variable part of $\mathbf{u}(t)$ and the vertices to add. Obviously, this causes GRTS to transmit a higher amount of data than CDR, as illustrated in Figure 22. However, the additional amount of transmitted data is small compared to the higher reduction rates of the GRTS variants of more than a factor two. For example, $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$ transmits only 5 to 23% more data than CDR. Assuming a header size of 28 byte (UDP/IP) per message, the difference is only 2 to 13%.

The differences between the GRTS realizations are caused by the different sizes of the variable part of $\mathbf{u}(t)$. In case of $\mathrm{GRTS}_{\mathrm{k}}^{\mathrm{Opt}}$ $(k = 1)$ and $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$, the variable part comprises only one vertex, whereas it may comprise multiple vertices with $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$. Therefore, the update messages of $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$ are slightly larger and replace more vertices on average than the update messages of the other two realizations.

The difference between $\mathrm{GRTS}_{\mathrm{k}}^{\mathrm{Opt}}$ $(k = 1)$ and $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$ of about 7% is caused by the fact that $\mathrm{GRTS}_{\mathrm{k}}^{\mathrm{Opt}}$ replaces the one vertex of the variable part more frequently than $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$, to achieve a better reduction rate. This shows that the two goals of the trajectory tracking problem – to minimize the communication cost and to minimize the number of vertices of the simplified trajectory – contradict for high reduction rates, as indicated at the end of Section 2.

7.4 Computational Costs

We now analyze the maximum space consumptions and computing times of LDRH, CDR, $\mathrm{CDR_m}$, and the GRTS realizations. The space consumption is measured in kilobytes by summing up the space consumption of the different variables and arrays, particularly including the sensing history $\mathbb{S}$. The maximum computing time for processing a new sensed position is measured in milliseconds using the processor's time stamp counter. To filter out interrupts of the process under test, we simulated the trajectory tracking algorithms without other user processes and repeated each measurement ten times.

Figure 23 shows the maximum space consumption of the tracking algorithms for all GPS traces and ϵ values. The space consumption of LDRH is negligible since it does not store a sensing history. In our simulations, the space consumption of CDR is well below 100 kB, although the size of sensing history of CDR is theoretically unbounded. Note that space consumption of CDR without the optimization of the sensing history

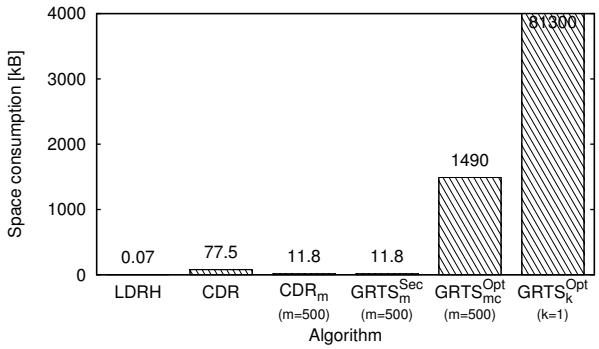

Fig. 23 Space consumption of major tracking algorithms.

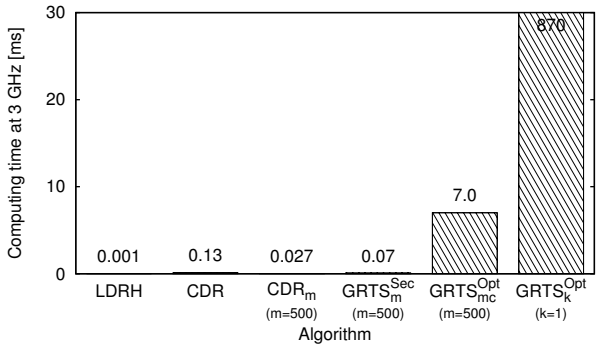

Fig. 24 Maximum computing times of major tracking algorithms.

proposed in Section 4.2 is 151.1 kB. Thus, the optimization saves 49%. On average, it even saves 56%.

More important, the space consumption of $\mathrm{CDR_m}$ with $m = 500$ is only 11.8 kB, despite the fact that there is no noticeable difference between the reduction performance of CDR and $\mathrm{CDR_m}$ with this parameterization.

$\mathrm{GRTS_m^{Sec}}$ with $m = 500$ also consumes only 11.8 kB, as the segment heuristic resembles the segment condition of CDR and does not require any extensive data structures in addition to $\mathbb{S}$.

$\mathrm{GRTS_{mc}^{Opt}}$ with $m = 500$, in contrast, consumes 1.49 MB as it constructs a graph with up to $m{\cdot}(m{-}1)/2$ edges over $\mathbb{S}$. Yet, the space consumption is bounded to this value, which can be seen from the fact that in our simulations $\mathrm{GRTS_k^{Opt}}$ with $k = 1$ consumes up to 55 times more space – although the reduction performance of $\mathrm{GRTS_{mc}^{Opt}}$ with $m = 500$ is higher.

The huge space consumption by $\mathrm{GRTS_k^{Opt}}$ is reflected in the maximum computing time per position fix, given in Figure 24. It shows that the computational costs of $\mathrm{GRTS_k^{Opt}}$ are too high for practical use. On the 3 GHz processor, the computing time reaches almost the sensing period T_S.

The maximum computing time of $\mathrm{GRTS_{mc}^{Opt}}$, in contrast, is only 7 ms and thus a fraction of the typical sensing period of $T_\mathrm{S} = 1\,\mathrm{s}$.

Nevertheless, the computational costs of $\mathrm{GRTS_{mc}^{Opt}}$ are huge compared to $\mathrm{GRTS_m^{Sec}}$, namely about a factor 100. Therefore, $\mathrm{GRTS_{mc}^{Opt}}$ should be preferred to $\mathrm{GRTS_m^{Sec}}$ only if the moving object has sufficient computational resources and reduction efficiency is of highest priority.

7.5 Prototypical Implementation of GRTS

In order to gather practical experiences with GRTS, we implemented a fully functional MOD system for tracking GPS-enabled smartphones and mobile computers.

The system consists of two components named *mobile component* and *MOD server*. The mobile component is executed by the smartphone or mobile computer. It reads the sensed position data from the corresponding GPS receiver and executes the $\mathrm{GRTS_m^{Sec}}$ algorithm with $m = 500$ on it. A separate thread transmits the update messages via UDP to the MOD server. We implemented two variants of the mobile component: First, a Java application for mobile computers. Second, an app for Android smartphones. Figure 25 gives a screenshot of the Android app. It shows the sensed and the simplified trajectory plotted on the map of the OpenStreetMap project [21]. The large circle depicts the accuracy bound ϵ. The small circle illustrates the maximum sensing deviation δ.

The MOD server receives the update messages and stores the vertices of the trajectories persistently in a PostgreSQL database. The predictions (i.e., the predicted velocities $\boldsymbol{\pi}_\mathrm{V}$ and the prediction origins u_n) are stored in a main memory table to reduce the number of write operations, as proposed at the end of Section 5.1. Therefore, only a subset of the update messages requires accessing the hard disk. Due to our focus on the tracking protocol, we did not implement any specialized spatiotemporal index structures within the database.

Google Earth is used as sample application to visualize all trajectories stored by the MOD. For this purpose, it is launched with a small file in the Keyhole Markup Language (KML). This file contains the host name of the MOD server and instructs Google Earth to query the MOD server once per second for a KML representation of all trajectories. A lightweight HTTP server attached to the MOD server receives those requests, queries the database and predictions table, translates the result into KML and sends the response to the Google Earth client.

Our implementation slightly extends the GRTS protocol by a timeout mechanism to terminate the sim-

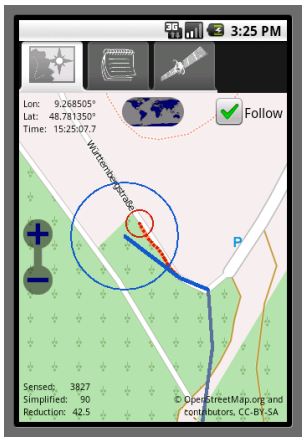

Fig. 25 Screenshot of prototypical app for Android.

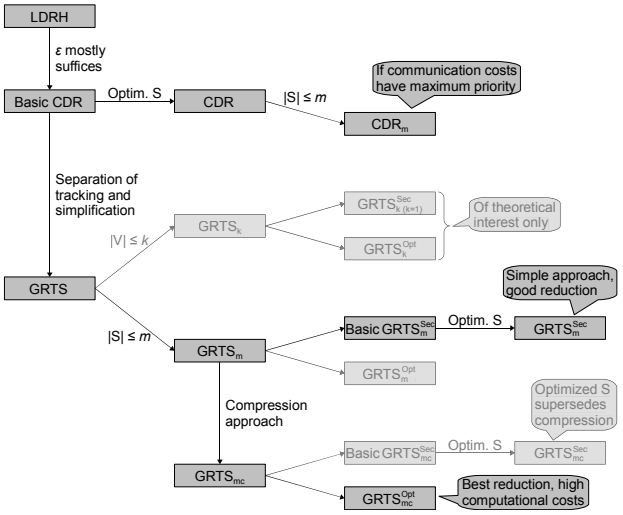

Fig. 26 Overview of proposed real-time trajectory tracking approaches.

plified trajectory during long-lasting network outages or after failures of the mobile component. Furthermore, the MOD server acknowledges every update message, so that the mobile component can detect message losses. Moreover, for clock synchronization, the update messages and acknowledgements contain measuring data about the network round-trip time.

Details on the architecture and implementation of the MOD system can be found in [14].

We conducted several experiments driving a car equipped with an OQO subnotebook and a Wintec WBT-300 GPS receiver. During our experiments, we used $\epsilon = 25\,\mathrm{m}$. Besides four network outages lasting several minutes, the system successfully allowed for tracking the car and its trajectory for more than nine hours from several PCs. During this experiment, we measured a reduction rate of 70. Per hour, only 60 kB of data were transmitted to the MOD server. These experimental results coincide with the results of our simulations.

Furthermore, we measured the computing time of $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$ ($m = 500$) on a Sony Ericsson Xperia X8 smartphone with a 600 MHz Qualcomm MSM7227 processor, running the Android 2.1 operating system. For this purpose, we recorded the GPS trace of a three hour bicycle tour in the raw NMEA format and replayed the Android app with it. More precisely, we executed the app four times with the trace, to be able to filter out interrupts of the process under test.[6] Our measurements yield a maximum computing time of 1.86 ms per position fix. Thus, the computing time on the 600 MHz smartphone is about 27 times greater than the computing time on the 3 GHz Intel Xeon Linux servers. Nevertheless, considering typical delays in wireless networks, it constitutes only a fraction of the update time T_{U}.

[6] This procedure allows to measure the CPU time in a higher resolution than provided by the getElapsedCpuTime-method of the android.os.Process class.

Therefore, the computing time of $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$ on smartphones is more than acceptable.

7.6 Conclusions for Selection of a Tracking Approach

With CDR, $\mathrm{CDR}_{\mathrm{m}}$, and the various GRTS realizations, we proposed and evaluated several approaches for real-time trajectory tracking. We also explained and showed that the communication cost and the number of vertices of the simplified trajectory cannot be completely minimized both together, but contradict to some (small) degree. Figure 26 depicts all these approaches and their relations.

As stated above, $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Opt}}$ is completely outperformed by $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$, whereas with $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Sec}}$ the compression technique of $\mathrm{GRTS}_{\mathrm{mc}}$ is superseded by the optimization of $\mathbb{S}$ proposed in Section 5.5. Therefore, those approaches are dimmed in Figure 26.

Moreover, in consideration of the extensive computational costs of $\mathrm{GRTS}_{\mathrm{k}}^{\mathrm{Opt}}$, we advise to use tracking approaches with bounded space consumptions and computing times. For this reason, we deem $\mathrm{CDR}_{\mathrm{m}}$, $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$, and $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$ to be particularly suited for practical use, depending on the actual requirements and resources in a given application scenario:

1. $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$: This approach affords high reduction performance at comparatively low computational costs. Besides, it is simple to implement compared to realizations of GRTS with the optimal line simplification algorithm by Imai and Iri [11]. Therefore, we suppose that $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$ meets the requirements of most use cases.
2. $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$: If reduction has maximum priority and the moving objects have sufficient computational

power, this approach should be used, as it affords to reach almost best possible reduction rates, depending on m. The average reduction rates of $\text{GRTS}_{\text{m}}^{\text{Sec}}$, in contrast, are 10 to 15% below the best possible ones.

3. CDR_{m}: In case that communication costs have maximum priority, CDR_{m} should be used since it minimizes the amounts of transmitted data. Note, however, again that CDR_{m} reaches only about one third of the reduction of $\text{GRTS}_{\text{m}}^{\text{Sec}}$ and $\text{GRTS}_{\text{mc}}^{\text{Opt}}$, even for large ϵ.

The processing cost at the MOD may be another decision criterion: An update message by CDR_{m} always adds a new vertex to $\mathbf{u}(t)$, whereas many update messages by GRTS only replace the last vertex u_n and the predicted velocity $\boldsymbol{\pi}_{\text{V}}$ and may be processed without any write operation (cf. Section 5.1). In return, some update messages require to remove or replace several vertices of $\mathbf{u}(t)$. Due to these different characteristics, it is impossible to give a general assessment concerning the MOD-sided processing costs. Therefore, for a concrete decision, the actual implementation of the MOD has to be taken into account.

8 Related Work

In this section, we first give a brief overview to line simplification algorithms in general, before we discuss existing approaches for offline and online trajectory simplification and real-time trajectory tracking in particular. We omit dead reckoning protocols for tracking the current position of moving objects since these have been analyzed comprehensively in Section 3.

Line simplification in general: Line simplification refers to a multitude of algorithmic problems on approximating a given polyline by a simplified one with fewer vertices. The two basic problem classes are:

1. min-#: Minimizing the number of vertices of the simplified polyline under a given accuracy bound.
2. min-ϵ: Minimizing the deviation between the two polylines under a given number of vertices for the simplified one.

The min-# problems can be further classified by the *dimensionality* of the underlying space (e.g., $\mathbb{R}^2$ or $\mathbb{R}^3$), the *distance metric* to measure the distance between two points (e.g., Manhattan distance (L_1), Euclidean distance (L_2), or uniform metric), and the *error measure* to determine the distance between two polylines from the pairwise distances of their points [2]. For the latter, most works implicitly consider the Hausdorff distance, defined as the largest distance from an arbitrary point of the one polyline to the closest point of the other polyline. However, there also exists works considering the Fréchet distance (e.g., [2,1]).

According to these criteria, efficient real-time trajectory tracking can be considered as min-# problem in the case of Hausdorff distance under the (time-)uniform distance metric in $\mathbb{R}^{1+d}$ with $d = 2$ or 3.

As further explained above, the Douglas-Peucker algorithm [6], the optimal algorithm by Imai and Iri [11], and the segment heuristic [18,2,10] are three prominent approaches for min-# simplification. Several works including [9] and [7] propose variants of the Douglas-Peucker algorithm with improved worst-case running times of $O(n \log n)$ and $O(n \log^k n)$ ($k = 2$ or 3) instead of $O(n^2)$.

Similar applies to the optimal algorithm: A naive implementation has running time $O(n^3)$ [11]. Imai and Iri already describe realizations with running times of $O(n^2 \log n)$ or even $O(n \log n)$ [11]. Agarwal and Varadarajan propose an algorithm with running time $O(n^{4/3+\tau})$, for any $\tau > 0$ [3]. Others are given in [5] and [29]. However, these improvements are limited to $\mathbb{R}^2$ and specific distance metrics and error measures.

Offline trajectory simplification: Cao et al. [4] discuss the use of the Douglas-Peucker heuristic for *offline* trajectory simplification. They consider four different distance metrics, including the time-uniform distance metric, and compare the Douglas-Peucker heuristic against the optimal algorithm regarding reduction performance and computing time. Their results on the reduction performance are in line with our results. For the distance metric E_2, which disregards the temporal component of the trajectories but only considers the Euclidean distance, they use the optimal simplification algorithm by Chan and Chin [5] with running time $O(n^2)$, tailored to $\mathbb{R}^2$. Nevertheless, they measure more than a factor 1000 between the computing times of the optimal algorithm and the Douglas-Peucker heuristic. Note that the disregard of the temporal component by the distance metric E_2 is problematic for many applications since the point of the simplified line segment $\overline{u_j\, u_{j+1}}$ that is closest to a given sensed position s_i with $u_j.t \leq s_i.t \leq u_{j+1}.t$ generally differs from the interpolated position $\overline{u_j\, u_{j+1}}(s_i.t)$.

Gudmundsson et al. [7] likewise propose the use of the Douglas-Peucker heuristic for offline trajectory simplification and argue against the optimal algorithm because of its running time.

With GRTS_{m} and GRTS_{mc} the choice of the line simplification algorithm may not be critical concerning computing time, as the division of the trajectory into stable, variable, and predicted part strictly limits the amount of simplification work per position – com-

pared to considering an entire trajectory. Therefore, the *online* algorithm $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$ achieves reduction rates close to the best possible offline rates (and significantly greater than the reduction rates of the Douglas-Peucker heuristic) at acceptable computing times.

Online trajectory simplification: Meratnia and de By [18] propose the segment heuristic for online and offline trajectory simplification but with a different error measure based on the *average* deviation between corresponding points of the original and the simplified trajectory. In detail, they refer to the segment heuristic as Opening-Window algorithm (OPW) and distinguish two variants. The one with the better reduction efficiency, which corresponds to the segment heuristic as explained here, is called BOPW-TR. Due the different error measure, the maximum deviation between the original trajectory and the simplified one is not bounded and depends on the simplification algorithm.

This also applies to threshold-guided sampling, a heuristic for online trajectory simplification proposed in [24]. It adds the most recent sensed position s_{R} as vertex to the simplified trajectory only if the speed or direction of the latest velocity compared to the velocity between the previous sensed positions and the average velocity between the last two vertices of the simplified trajectory exceeds a certain threshold. Therefore, the deviation between the original trajectory and the simplified one is not bounded.

In [23], the same authors propose the AmTree, a data structure for managing an incoming stream of sensed positions with constant storage consumption. The AmTree "forgets" more and more positions over time so that fewer positions are known for the far past than for the recent past. Again, the deviation between the original trajectory and the resulting simplified one is not bounded by some predefined accuracy.

In [10], a software component for online preprocessing position data of mobile objects is presented. The component aims at reducing the position data to be stored by a database according to a given accuracy bound. The authors propose five different reduction algorithms, where in fact only one – the segment heuristic – performs line simplification, i.e., yields a *connected* simplified trajectory.

None of the above works considers real-time trajectory tracking for remote moving objects.

Real-time trajectory tracking approaches: In [26], Tiešytė and Jensen present an approach for real-time trajectory tracking based on LDR. They propose an algorithm for computing a connected trajectory on the basis of linear predictions, which approximates the actual trajectory according to the same accuracy bound used with LDR. However, their findings only apply to pre-known routes like bus lines, i.e., movement in $\mathbb{R}^1$.

In [27], Trajcevski et al. prove that the simplified trajectory given by the origins of the linear predictions of LDR with accuracy bound ϵ approximates the actual trajectory by 2ϵ [27]. Based on this finding they conclude that LDRH, i.e., LDR with $\epsilon' := \epsilon/2$, allows for trajectory tracking with accuracy bound ϵ. As discussed in detail in Section 3, this approach is very conservative and therefore is outperformed by GRTS by a factor five in terms of reduction efficiency.

9 Summary

In this paper, we presented the *Connection-Preserving Dead Reckoning* (CDR) and *Generic Remote Trajectory Simplification* (GRTS) protocols for tracking the trajectories of moving objects with embedded positioning sensors at a remote MOD efficiently.

For this purpose, the objects sense their positions periodically but report only a subset of the positions to the MOD so that the resulting simplified trajectory approximates the actual movement according to a predefined accuracy bound. To inform the MOD about the current position, CDR and GRTS use dead reckoning.

CDR is solely based on dead reckoning whereas GRTS separates the tracking of the current position from the simplification of the past trajectory. Therefore, GRTS outperforms CDR by more than a factor two in terms of reduction performance, while CDR minimizes the amount of communicated data.

For both CDR and GRTS, we proposed optimized algorithms with bounded space consumption and computing time. In addition, we investigated different realizations of GRTS with two important line simplification algorithms and evaluated the resulting trade-off between computational costs and reduction efficiency.

The realization $\mathrm{GRTS}_{\mathrm{m}}^{\mathrm{Sec}}$ with a simple line simplification heuristic affords substantial reduction performance at low computational costs. In detail, it reaches 85 to 90% of the best possible (offline) reduction rate at computing times of less than 1.9 ms on a 600 MHz smartphone and of 0.07 ms on a 3 GHz Intel Xeon processor. The realization $\mathrm{GRTS}_{\mathrm{mc}}^{\mathrm{Opt}}$ with the optimal line simplification algorithm by Imai and Iri [11], in contrast, may reach more than 97% of the best possible reduction at one hundred times higher cost.

Acknowledgements

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

References

1. M. A. Abam, M. de Berg, P. Hachenberger, and A. Zarei. Streaming Algorithms for Line Simplification. In *Proc. of 23rd Symp. on Computational Geometry (SCG)*, pages 175–183, Gyeongju, South Korea, 2007.
2. P. K. Agarwal, S. Har-Peled, N. H. Mustafa, and Y. Wang. Near-Linear Time Approximation Algorithms for Curve Simplification. *Algorithmica*, 42(3–4):203–219, 2005.
3. P. K. Agarwal and K. R. Varadarajan. Efficient Algorithms for Approximating Polygonal Chains. *Discrete and Computational Geometry*, 23(2):273–291, 2000.
4. H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *VLDB Journal*, 15(3):211–228, 2006.
5. W. S. Chan and F. Chin. Approximation of Polygonal Curves with Minimum Number of Line Segments. In *Proc. of 3rd Int'l Symp. on Algorithms and Computation (ISAAC)*, pages 378–387, Nagoya, Japan, 1992.
6. 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, 1973.
7. J. Gudmundsson, J. Katajainen, D. Merrick, C. Ong, and T. Wolle. Compressing spatio-temporal trajectories. In *Proc. of 18th Int'l Symp. on Algorithms and Computation (ISAAC)*, pages 763–775, Sendai, Japan, 2007.
8. R. H. Güting and M. Schneider. *Moving Objects Databases*. Morgan Kaufmann, San Francisco, CA, 2005.
9. J. Hershberger and J. Snoeyink. An O(n log n) Implementation of the Douglas-Peucker Algorithm for Line Simplification. In *Proc. of 10th Symp. on Computational Geometry*, pages 383–384, Stony Brook, NY, 1994.
10. N. Hönle, M. Großmann, D. Nicklas, and B. Mitschang. Preprocessing Position Data of Mobile Objects. In *Proc. of 9th Int'l Conf. on Mobile Data Management (MDM)*, pages 41–48, Beijing, China, 2008.
11. H. Imai and M. Iri. *Computational Morphology*, chapter Polygonal Approximations of a Curve – Formulations and Algorithms, pages 71–86. North-Holland Publishing Company, 1988.
12. R. Lange, F. Dürr, and K. Rothermel. Online Trajectory Data Reduction using Connection-preserving Dead Reckoning. In *Proc. of 5th Int'l Conf. on Mobile and Ubiquitous Systems (MobiQuitous)*, Dublin, Ireland, 2008.
13. R. Lange, F. Dürr, and K. Rothermel. Scalable Processing of Trajectory-Based Queries in Space-Partitioned Moving Objects Databases. In *Proc. of 16th ACM SIGSPATIAL Int'l Conf. on Advances in Geographic Information Systems (ACM GIS)*, pages 270–279, Irvine, CA, 2008.
14. R. Lange, F. Dürr, and K. Rothermel. Efficient Tracking of Moving Objects using Generic Remote Trajectory Simplification (Demo Paper). In *Proc. of 8th IEEE Int'l Conf. on Pervasive Computing and Communications Workshops (PerCom Workshops)*, Mannheim, Germany, 2010.
15. R. Lange, T. Farrell, F. Dürr, and K. Rothermel. Remote Real-Time Trajectory Simplification. In *Proc. of 7th IEEE Int'l Conf. on Pervasive Computing and Communications (PerCom)*, pages 184–193, Galveston, TX, 2009.
16. 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.
17. 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, 2001.
18. N. Meratnia and R. A. de By. Spatiotemporal Compression Techniques for Moving Point Objects. In *Proc. of 9th Int'l Conf. on Extending Database Technology (EDBT)*, pages 765–782, Heraklion, Crete, 2004.
19. P. Misra and P. Enge. *Global Positioning System: Signals, Measurements and Performance*. Ganga-Jumuna Press, 2001.
20. M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-Temporal Access Methods. *IEEE Data Engineering Bulletin*, 26(2):40–49, 2003.
21. OpenStreetMap. http://www.openstreetmap.org/.
22. D. Pfoser and C. S. Jensen. Capturing the Uncertainty of Moving-Object Representations. In *Proc. of 6th Int'l Symp. on Advances in Spatial Databases (SSD)*, pages 111–131, Hong Kong, China, 1999.
23. M. Potamias, K. Patroumpas, and T. Sellis. Amnesic Online Synopses for Moving Objects. In *Proc. of 15th ACM Int'l Conf. on Information and Knowledge Management (CIKM)*, pages 784–785, Arlington, VA, 2006.
24. M. Potamias, K. Patroumpas, and T. Sellis. Sampling Trajectory Streams with Spatiotemporal Criteria. In *Proc. of 18th Int'l Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 275–284, Vienna, Austria, 2006.
25. J. Rankin. GPS and Differential GPS: An Error Model for Sensor Simulation. In *Position Location and Navigation Symp.*, pages 260–266, 1994.
26. D. Tiešytė and C. S. Jensen. Recovery of Vehicle Trajectories from Tracking Data for Analysis Purposes. In *Proc. of 6th European Congress and Exhibition on Intelligent Transport Systems and Services*, Aalborg, Denmark, 2007.
27. G. Trajcevski, H. Cao, P. Scheuermann, O. Wolfson, and D. Vaccaro. Online Data Reduction and the Quality of History in Moving Objects Databases. In *Proc. of 5th ACM Int'l Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, Chicago, IL, 2006.
28. U.S. Dept. of Defense. Global Positioning System Standard Positioning Service Performance Standard, 2001.
29. K. R. Varadarajan. Approximating Monotone Polygonal Curves Using the Uniform Metric. In *Proc. of 12th Symp. on Computational Geometry*, pages 311–318, Philadelphia, PA, 1996.
30. A. Čivilis, C. S. Jensen, and S. Pakalnis. Techniques for Efficient Road-Network-Based Tracking of Moving Objects. *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 17(5):698–712, 2005.
31. O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and Querying Databases that Track Mobile Units. *Distr. and Parallel Databases*, 7(3):257–287, 1999.
32. Jean-Marie Zogg (u-blox AG). Essentials of Satellite Navigation (Compendium). http://www.u-blox.com/, 2009.