Crowdsourced Trace Similarity with Smartphones

Demetrios Zeinalipour-Yazti, *Member, IEEE,* Christos Laoudias, *Student Member, IEEE,* Constandinos Costa, Michail Vlachos, Maria I. Andreou, Dimitrios Gunopulos *Member, IEEE*

Abstract—Smartphones are nowadays equipped with a number of sensors, such as WiFi, GPS, accelerometers, etc. This capability allows smartphone users to easily engage in crowdsourced computing services, which contribute to the solution of complex problems in a distributed manner. In this work, we leverage such a computing paradigm to solve efficiently the following problem: comparing a query trace Q against a crowd of traces generated and stored on distributed smartphones. Our proposed framework, coined SmartTrace$^+$, provides an effective solution without disclosing any part of the crowd traces to the query processor. SmartTrace$^+$, relies on an in-situ data storage model and intelligent top-K query processing algorithms that exploit distributed trajectory similarity measures, resilient to spatial and temporal noise, in order to derive the most relevant answers to Q. We evaluate our algorithms on both synthetic and real workloads. We describe our prototype system developed on the Android OS. The solution is deployed over our own SmartLab testbed of 25 smartphones. Our study reveals that computations over SmartTrace$^+$ result in substantial energy conservation; in addition, results can be computed faster than competitive approaches.

Index Terms—Crowdsourcing, Trajectory Similarity Search, Smartphones, Longest Common Subsequence, Android OS.

1 Introduction

Crowdsourcing refers to a distributed problem solving model where a population of undefined size, engages in the solution of a complex problem for monetary or ethical (i.e., intellectual satisfaction) benefit through an open call. Examples of commercial crowdsourcing services include *Gwap.net's ESP* image tagging game, *reCAPTCHA.net's* book correction service and specialized marketplaces for assigning crowdsourcing tasks (e.g., *Amazon's Mechanical Turk*, *Gigwalk.net* and *oDesk.com*). In addition, academic crowdsourcing frameworks and techniques are currently underway in most computer science fields including data management [19], [33], network management [29], source-code development [34], computational linguistics [46] and active learning [8].

All aforementioned crowdsourcing frameworks are inherently *participatory*, as they require the active participation of users in the solution of the assigned task. The widespread availability of smartphone and tablet devices featuring geo-location and other sensing capabilities (e.g., proximity, ambient light, accelerometer, camera, microphone, etc.) are providing new means for *opportunistic* crowdsourcing frameworks; these reside in the background requiring no user intervention. Examples of this new paradigm include Google's WiFi Access Point logging service running on Android Smartphones, which populates Google's Geo-location database [21] and *VTrack* [40], which allows smartphone users to estimate their road traffic by continuously sharing their list of nearby WiFi access points. This paradigm also includes frameworks such as *Ear-Phone* [38], which enables the construction of fine-grained noise maps from volunteer smartphone users sharing data captured by their microphone, *PotHole* [17], which allows smartphone users to share their vibration and location data in order to enable street hole identification; and a wide range of other compelling applications [30], [16], [4], [11].

In this paper, we present a crowdsourced trace similarity search framework, called SmartTrace$^+$, which enables the execution of queries in the form: "*Report the users that move more similar to Q, where Q is some query trace.* The notion of *similarity* captures the traces (i.e., trajectories) that differ only slightly, in the whole sequence, from the query Q. Our framework enables the execution of such queries in both outdoor environments (using GPS) and indoor environments (using WiFi Received-Signal-Strength), without disclosing the traces of participating users to the querying node[1]. Our framework can be utilized in large-scale urban and transit planning applications, which would otherwise be limited by data disclosure constraints [42], [49], social networking applications for smartphones [50], habitant monitoring [31] and others.

As an illustrative example, consider a transit authority that plans its bus routes and wants to know whether a specific route is taken by at least K users between 8:00-9:00am. In such a scenario, one is interested in asking a crowd of users in some target area to participate with their local trace history through an open call. In particular, the users can passively participate in the resolution of the query for monetary benefit or intellectual satisfaction, without disclosing their traces to the authority.

There are already centralized trajectory search services such as Microsoft's GeoLife Project (see Figure 1 left), GPS-Waypoints.net, ShareMyRoutes.com, and their academic

D. Zeinalipour-Yazti (Corresponding Author), Department of Computer Science, University of Cyprus, Email: dzeina@cs.ucy.ac.cy, Tel: +357-22-892755, Fax: +357-22-892701, Address: 1 University Avenue, P.O. Box 20537, CY-2109, Nicosia, Cyprus; C. Laoudias, C. Costa, University of Cyprus, P.O. Box 20537, 1678 Nicosia, Cyprus; M. Vlachos, IBM Research Zurich, 8803 Rüschlikon, Switzerland; M.I. Andreou, Open University of Cyprus, 1304 Nicosia, Cyprus; D. Gunopulos, University of Athens, 15784 Athens, Greece.

1. Software available at: http://smarttrace.cs.ucy.ac.cy/

counterparts [26], to perform trajectory search and retrieval. However, these services assume that the user trajectories are stored on a centralized or cloud-like infrastructure prior to query execution. On the other hand, the techniques proposed in this work are decentralized and maintain the data *in-situ* (i.e., on the smartphone that generated the data). When a query is posted, our algorithms collect a set of scores from participating nodes (as opposed to collecting their location continuously) and derive the answer intelligently based on these scores only without unveiling the target trajectories to the query processor. While this cannot take advantage of global knowledge structures available in a centralized setting (e.g., catalogs, indexes, etc.), our setting has the following advantages:

i. Smartphones have both expensive communication mediums but also asymmetric upload/download links, thus by continuously transferring data to the query processor can both deplete the precious smartphone battery faster, increase user-perceived delays, but can also quickly degrade the network health[2].
ii. Continuously disclosing user positional data to a central entity might compromise user privacy in serious ways. This creates services that have recently raised many concerns, especially for social networking services (e.g., Facebook, Buzz, etc.) and smartphone services[3].

In the proposed SmartTrace$^+$ framework, the tuples of each target trajectory A_i, are compared against the locations of query Q within some temporal and spatial window. SmartTrace$^+$, circumvents expensive similarity executions by executing low-cost linear-time (i.e., $O(|A_i|)$-time) computations on the smartphones in a pre-processing step. It then uses either an iterative top-K processing algorithm or a non-iterative counterpart, without ever retrieving the target trajectories to the centralized query processor. We validate our algorithms in respect to energy and time both analytically and empirically.

This paper builds on our previous work in [48], [14], in which we offered a preliminary presentation of the SmartTrace$^+$ framework for disclosure-free GPS trace-search in smartphone networks. In this paper, we introduce several new improvements and contributions summarized as follows:

- In addition to the iterative *SmartTrace (ST)* retrieval algorithm, we also offer a non-iterative algorithm, called *Non-Iterative SmartTrace (NIST)*. The latter algorithm operates in two phases and reduces substantially response time at the expense of a slight increase in network traffic.
- We provide an analytical study for the performance and convergence properties of our algorithms. In particular, we develop an Energy/Time model that is utilized to derive the analytical properties of our framework. The analytical study is validated through our trace-driven simulation and experimentation with a real programming cloud of smartphones.
- We present the building blocks of a real prototype system we have developed in Google's Android OS for smartphones. We also discuss our user interfaces, communication protocol, application-level features and design choices.
- We introduce an extensive experimental study and we present experimental evidence for the motivation and efficiency of the *ST* and *NIST* algorithms using a trace-driven experimental methodology. We utilize mobility patterns from Microsoft's Geolife project and car trajectories from Oldenburg. Finally, we deploy our actual implementation over our SmartLab testbed consisting of 25 real smartphone devices.

2. "Customers Angered as iPhones Overload AT&T", Jenna Wortham, The New York Times (online), Sept. 2, 2009.

3. "Tracking File Found in iPhones", Nick Bilton, The New York Times (online), April 20, 2011.

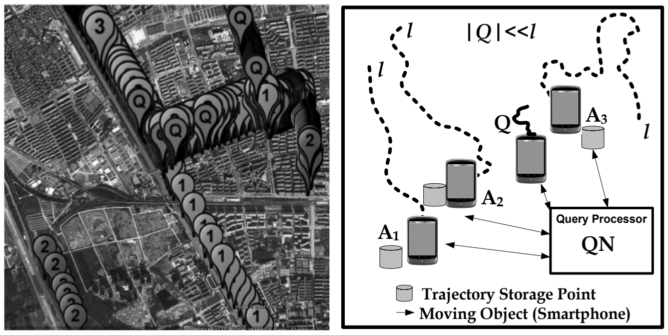

Fig. 1. This work is motivated by the fact that smartphones nowadays can log on local storage trajectories for extensive periods of time. Our objective is to enable efficient online querying of these traces without revealing the target trajectories to the query processor. *Left:* Traces captured by Microsoft's Geolife project [49] (query denoted as Q). *Right:* Our system model.

The remainder of the paper is organized as follows: Section 2 provides our system model and formulates the problem. Section 3 presents the SmartTrace$^+$ framework and its two top-K query processing algorithms, *ST* and *NIST*. Section 4 presents a detailed performance analysis of our framework, while Section 5 outlines our prototype system implemented in Android. Section 6, presents our experimental methodology and evaluation for both simulation and real executions over our testbed. Finally, Section 7 reviews the related work and Section 8 concludes the paper.

2 Problem Formulation

In this section we start out by providing the notation used in the paper. We shall then formalize our system model and formulate the problem. Finally, we will briefly provide some background material on trajectory similarity matching and some motivating micro-benchmarks. Our main symbols are summarized in Table 1.

2.1 System Model

Let $\{A_1, A_2, ..., A_m\}$ denote a set of m smartphone users moving in the xy-plane (see Figure 1 right). At each discrete time instance, object A_i ($\forall i \leq m$) generates a spatio-temporal record $A_{ij} = (t_{ij}, x_{ij}, y_{ij})$, where t_{ij} denotes A_i's temporal dimension and x_{ij}, y_{ij} A_i's spatial dimensions. Consequently, a trajectory can be thought of as a continuous sequence of l

such records, i.e., $A_i = (A_{i1}, A_{i2}, \cdots, A_{il})$ (l also denoted as $|A_i|$). Here the complete trace of a trajectory A_i is stored in its entirety locally on a smartphone (i.e., on flash memory). A smartphone (or tablet) is a feature-rich device (e.g., dual-core CPU) that utilizes batteries, thus has limited computational resources and also networking operations are expensive in terms of energy. Additionally, the link is asymmetric with the uplink being much slower than the downlink (typically one order of magnitude).

An assumption underlying this work is that processing and networking operations are equally expensive in terms of energy and must be avoided. Our current figures, presented at the end of Section 6.1, indicate that networking operations on smartphones and tablets are approximately twice as expensive as processing tasks (i.e., $\approx$ 300mW vs. 600mW). The same figure also holds for wireless sensor devices (e.g., Xbow's TelosB uses 23mW and 69mW for CPU and networking [2], respectively). For more powerful mobile devices with high-power and high-frequency processors (or several processors), such as laptops, the costs are usually reversed e.g., 30W for CPU vs. 19W for networking. Nevertheless, in all aforementioned devices, both processing and networking functions are equally important, thus this work aims to minimize the utilization of both parameters equally.

Let us now consider an arbitrary snapshot similarity query $Q = (Q_1, Q_2, \cdots, Q_f)$, where $f << l$ (f also denoted as $|Q|$), which aims to uncover the K most relevant trajectories to Q, for a user-defined constant K. Q might either be initiated by a smartphone and propagated towards the querying node QN, or might be initiated at QN. In order to provide a robust measure of the similarity between Q and A_i ($i \leq m$), one has to accommodate to both spatial and temporal variations in the trajectory pattern. Although we shall explain these similarity measures more precisely in the next subsection, let us for the moment assume that there is some function $LCSS(Q, A_i)$, which performs the trajectory comparison between Q and A_i accurately, but at a high computational cost. $LCSS(Q, A_i)$ returns a score in the range [0..1] (where 1 denotes highest similarity). In a smartphone network setup, $LCSS(Q, A_i)$ can either be conducted in a centralized fashion (i.e., after transferring all m trajectories to QN), or in a decentralized fashion (i.e., after having each smartphone conducting $LCSS(Q, A_i)$ locally and then returning back the outcome.)

Through our analytical and empirical analysis we show that the *Centralized* approach performs poorly in terms of energy and response time. In addition, its operation poses significant privacy risks, because users need to share their complete trajectory with QN. Similarly, the *Decentralized* approach also performs poorly in terms of energy consumption as it invokes expensive trajectory comparison metrics on all smartphone participants. The SmartTrace$^+$ approach we propose in this work performs well both in respect to query response time and energy consumption on the smartphones. More importantly, SmartTrace$^+$ never reveals the complete user trajectories, because it only returns matching scores and tentatively the matched subsequence of length $|Q|$, where $|Q| << l$.

TABLE 1
Symbol Description

Symbol	Definition
QN	The Querying Node
Q	Query trajectory
K	Number of requested results
A	Target trajectory stored on smartphone
m	Number of trajectories
l	Trajectory length (discrete points)
$LCSS(Q, A)$	Trace similarity function
δ	Time matching window for LCSS(Q,A)
ϵ	Spatial matching window for LCSS(Q,A)
$LCSS(MBE_Q, A)$	Function bounding above LCSS(Q,A)
$LCSS(LB_Q, A)$	Function bounding below LCSS(Q,A)

2.2 Background on Trajectory Similarity

Point-to-Point Matching: The fastest way to compute the similarity between two arbitrary trajectories, A and B, is to use any of the L_p*-Norm* distances (e.g., *Manhattan* (L_1), *Euclidean* (L_2) *or Chebyshev* (L_∞)). Using the L_p-Norms, one can match the data points of A and B at identical time positions and derive the distance in only $O(l)$-time, where l is the minimum length of the compared trajectories. Although the L_p*-Norm* distances can be calculated very efficiently, these are neither flexible to *out-of-phase* matches (e.g., if we have two identical trajectories but either one moves earlier in time) nor tolerant to *noisy data* (e.g., we have two identical trajectories but either one has a slight deviation in its spatial movement).

Time-shifted Matching: In order to support out-of-phase matching in trajectories one could exhaustively compare each point of one trajectory to all the points of the other trajectory. Although such a method, which has an $O(l^2)$-time cost, would yield the optimal matching points between the two trajectories, it is computationally expensive on resource-limited devices. Example techniques belonging to this class of techniques include the *Longest Common Subsequence (LCSS)*, utilized in this work and explained in more detail next, the *Dynamic Time Warping (DTW)* [7], the *Edit Distance on Real Sequences (EDR)* [28] and the *Edit Distance with Real Penalty (ERP)* [27]. All aforementioned techniques have a similar quadratic execution cost and exhibit differences which are not important in the context of this work. In particular, DTW uses a different recursion step than LCSS but is still non-metric, while EDR and ERP add time shifting support to the L_p-Norms, thus are metric distances, being more appropriate for cases where the triangle inequality is important.

In order to expose the quadratic execution cost of the time-matching methods mentioned above, we have implemented all of them in the Android operating system and tested our implementation on five different smartphone units. For ease of exposition we also present the time for executing the linear L_2 distance. Figure 2, shows that LCSS is the most efficient among the presented time-matching methods being $2.2-2.6$x faster than DTW, $4.2-6.0$x times faster than ERP and $5.9-8.0$x times faster than EDR. This is attributed to the fact that DTW, ERP and EDR have a more expensive recursion step (i.e., three recursive clauses as opposed to only two found in LCSS, see [28], [27]). Additionally, ERP and EDR are worse than DTW as the former two perform some additional

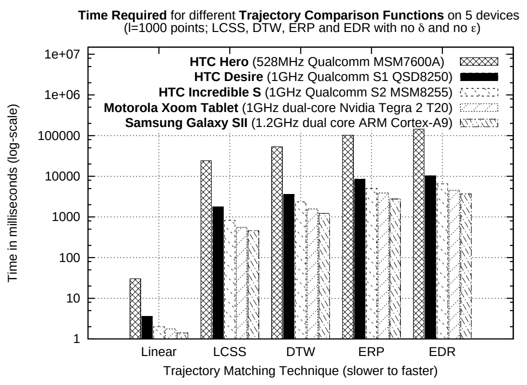

Fig. 2. Trajectory matching on popular smartphones. Our micro-benchmark reveals that LCSS is the most energy efficient approach among the $O(l^2)$-time matching functions that support time shifting.

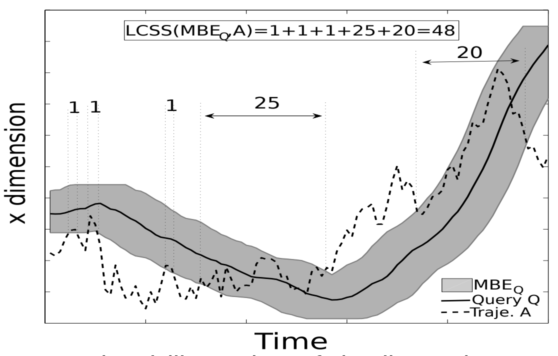

Fig. 3. A visual illustration of the linear-time matching between the query Q and the target trajectory A, using $LCSS(MBE_Q, A)$. The total matching is measured as the sum of points where A crosses the Minimum Bounding Envelope (MBE) of Q.

distance computations compared to DTW. In summary, the performance on the fastest and slowest phone, respectively, was: LCSS: 0.4s to 24s, DTW: 1.2s to 53s, ERP: 2.7s to 102s and EDR: 3.6s to 144s.

It is important to mention that the ideas presented in this work are orthogonal to the chosen similarity function. In particular, one could substitute $LCSS(Q, A_i)$, with $DTW(Q, A_i)$, $ERP(Q, A_i)$ or $EDR(Q, A_i)$ and still provide high quality matching at the same quadratic cost. Yet, one should also provide a function that would correctly bound above each of these similarity functions individually. In this work, we provide a linear-time bounding method that only applies to LCSS (i.e., $LCSS(MBE_Q, A_i)$). Providing methods that bound above the rest functions of this class, in linear or lower-order time, remains outside the scope of this work.

Longest Common Subsequence (LCSS): We shall now define the LCSS function more carefully. This matching function has been extensively used in many 1-dimensional sequence problems, such as string matching. The 2-dimensional adaptation of LCSS using the L_∞[4] is defined as following:

Definition 1: *Given integers δ and ϵ, the Longest Common Sub-Sequence similarity $LCSS_{\delta,\epsilon}(A, B)$ between two sequences A and B is defined as:*

$$LCSS_{\delta,\epsilon}(A, B) =$$

$$\begin{cases} \mathbf{0}, \quad \text{if A or B is empty} \\ \mathbf{1 + LCSS_{\delta,\epsilon}(Head(A), Head(B))} \\ \quad \text{if } |a_{x:l_1} - b_{x:l_2}| < \epsilon \text{ and} \\ \qquad |a_{y:l_1} - b_{y:l_2}| < \epsilon \text{ and } |l_1 - l_2| < \delta \\ \mathbf{max(LCSS_{\delta,\epsilon}(Head(A), B), LCSS_{\delta,\epsilon}(A, Head(B)))} \\ \quad \text{otherwise} \end{cases}$$

where δ and ϵ are application-specific parameters that allow flexible matching in the *time* domain and the *space* domain, respectively, and $Head(A) = ((a_{x:1}, a_{y:1}), \cdots, (a_{x:l-1}, a_{y:l-1}))$. LCSS deals with both aforementioned limitations of the L_p*-Norm* distances, because these cases are simply dropped from the matching.

4. We could also use L_1 or L_2 for the recursion step.

Bounding Above LCSS: Even though LCSS offers many desirable properties, its quadratic time complexity constitutes this method inefficient for long trajectories, such as those studied in this work. One idea to overcome this complexity, is to construct a *Minimum Bounding Envelope (MBE)* of the query Q[5] by replicating each trajectory point $Q[i]$ for δ time instances before and after time i and also replicate each point $Q[i]$ for ϵ space instances above and below $Q[i]$. Notice, that MBE_Q covers the area between the high part of the envelope, formally defined as $EnvHigh[i] = max(Q[j] + \epsilon), |i - j| \leq \delta$ and the low part of the envelope, formally defined as $EnvLow[i] = min(Q[j] - \epsilon), |i - j| \leq \delta$. The $LCSS(MBE_Q, A_i)$ function [44] can then be computed using the following definition (also see Figure 3):

$$LCSS(MBE_Q, A_i) = \sum_{j=1}^{|A_i|} \begin{cases} 1 & \text{if } A_i[j] \text{ within envelope} \\ 0 & \text{otherwise} \end{cases}$$

Lemma 1 ($LCSS(MBE_Q, A_i)$ Correctness): *For any two trajectories Q and A the following holds: $LCSS(Q, A_i) \leq LCSS(MBE_Q, A_i)$.*

Proof: By construction, MBE_Q covers the area between $EnvLow$ and $EnvHigh$. This area includes the points of Q plus all the points within an ϵ spatial and a δ temporal window around Q. Therefore, no possible matching between points of A and Q will be missed. It follows that $LCSS(Q, A_i) \leq LCSS(MBE_Q, A_i)$ $\square$

Bounding Below LCSS: The obvious benefit of $LCSS(MBE_Q, A_i)$, compared to $LCSS(Q, A_i)$, is that the former function can be computed in linear, as opposed, to quadratic time. Motivated by the same incentive, in this section we define a simple adaptation of $LCSS(Q, A_i)$, such that it provides a lower bound on $LCSS(Q, A_i)$, again in linear time. Such a method, denoted as $LCSS(LB_Q, A_i)$, conducts a computation of $LCSS(Q, A_i)$ ($i \leq m$) for δ equal to zero, which is the same to L_∞ provided in Definition 1 and consequently performs in linear time with respect to A_i. Although the utilization of such a lower bound is not clear at this point, we shall later show in Section 3.3, how such

5. MBE is constructed in our setup on the local A_i, rather than Q.

a lower bound can facilitate the identification of the correct answer-set to the problem we discuss in this paper.

Lemma 2 ($LCSS(LB_Q, A_i)$ Correctness): *For any two trajectories Q and A the following holds:* $LCSS(LB_Q, A_i) \leq LCSS(Q, A_i)$.

Proof: it follows directly from the definition of $LCSS(LB_Q, A_i)$ that is equal to $LCSS_{0,\epsilon}(Q, A_i)$. $\square$

3 The SmartTrace$^+$ Framework

In this section we start out with an outline of the SmartTrace$^+$ framework and the two retrieval algorithms that lie at its foundation (i.e., *ST* and *NIST*). We also formally prove the correctness and convergence properties of the two presented algorithms.

3.1 Outline of Operation

First note that the similarity query Q is initiated by some querying node QN (or alternatively at some smartphone that propagates its Q towards QN). QN then disseminates Q to a crowd of active smartphone users in a pre-specified spatial boundary. Upon receiving Q, each candidate smartphone executes locally one or more inexpensive matching functions. In particular, the first algorithm we propose (i.e., *ST*) only calculates an upper bound on the matching of the query to the local trajectory. The second algorithm we propose (i.e., *NIST*) calculates both an upper bound (ub) and a lower bound (lb) on the matching of the query to the local trajectory. Let the $UB = (ub_1, \cdots, ub_m)$ and $LB = (lb_1, \cdots, lb_m)$ score vectors of all nodes be denoted as *METADATA* and the actual trajectories stored locally on each smartphone as *DATA*. Obviously, *DATA* is both disclosure-sensitive and also orders of magnitudes larger than *METADATA*, thus *DATA* needs to remain on the smartphones during query resolution. The objective of both the *ST* and the *NIST* algorithms we propose, is to intelligently exploit the *METADATA* scores in order to identify the K highest ranked answers without pulling *DATA* to QN.

3.2 The SmartTrace (ST) Algorithm

The first algorithm we developed, *SmartTrace* (ST), is a novel iterative algorithm for retrieving the K most similar trajectories to a query trajectory Q. Our proposed scheme performs well both in respect to response time and energy, but also does so without ever revealing the complete target trajectories to QN (i.e., it only returns the matched subsequence, if any.) Additionally, the identity of a user is not revealed (we use the notion of a non-unique screen name), unless the user decides to do so.

Algorithm 1 : SmartTrace (ST)

Input: Query Trajectory Q, m Target Trajectories, Result Cardinality K ($K << m$), Iteration Step Increment λ.
Output: K trajectories most similar to Q.
At the query node QN:

1) **Upper Bound (UB) Computation:** Instruct each of the m smartphones in the crowd to invoke a computation of the linear-time $LCSS(MBE_Q, A_i)$ ($i \leq m$).
2) **Collection of UB:** Receive the UBs of all m trajectories participating in the query and add those scores to the *METADATA* vector stored on QN. Let *METADATA* be sorted in descending order based on the UB scores.
3) **Identify Candidates:** Find the $\lambda + 1$ ($\lambda \geq K$) highest UBs in *METADATA*, and add the identities to an empty set S (denoted as the candidate set). If an element has already been added to S, during a previous iteration do not add it again.
4) **Full Computation:** Ask each element in the S-set to compute $LCSS(Q, A_i)$, in a decentralized manner, and then send back the next λ full similarity scores.
5) **Termination Condition:** If the (λ+1)-th UB is smaller than the K-th largest full match then stop; else goto step 3 in order to identify the next λ candidates.
6) **Ship Matching:** If the termination condition has been met, ship the respective matches to QN, based on some local trace disclosure policy.

Description: In step 1 of the *ST* algorithm (see Algorithm 1), QN instructs all m nodes to invoke the computation of the linear-time function $LCSS(MBE_Q, A_i)$ ($i \leq m$), which bounds above the expensive $LCSS(Q, A_i)$ function. Using the above method, *ST* circumvents the massive deployment of the expensive similarity function $LCSS(Q, A_i)$, presented in Section 2.2, which performs local stretching in both time and space to overcome the temporal and spatial distortions in trajectories.

In step 2, QN retrieves these upper bounds and adds them in descending order to a local *METADATA* vector. By doing this, QN obtains a quick summary of the trajectories similar to Q. Steps 3 to 5 are executed iteratively until convergence. In particular, during step 3, QN adds the identities of the objects with the $\lambda + 1$ highest upper bounds to a set named S. These objects provide the first line of candidates for the answer set, as these objects have the highest $LCSS(MBE_Q, A_i)$ value. The given objects will be analyzed more carefully in the next step of the algorithm in order to determine the correct top-K set. Notice that the objects in the S-set, do not again define the final top-K result. In particular, it is absolutely possible that some arbitrary object in the S-set with a high $LCSS(MBE_Q, A_i)$ score has a low full score $LCSS(Q, A_i)$. Consequently, the algorithm can still not converge.

The λ parameter, which applies to the *ST* algorithm only, expresses an application-specific confidence in the *METADATA* bounds. In particular, when the *METADATA* vector contains tight upper bounds, then λ might be set to a small value. So this parameter defines how aggressively some application wants to determine the top-K results. It will be proven next that *ST* will not perform more than $O(m/\lambda)$ iterations in the worst case.

In step 4, QN asks each smartphone in the S-set, to compute the full scores (if a smartphone has been contacted in a previous iteration we do not contact it again.) In particular, each smartphone is asked to locally compute $LCSS(Q, A_i)$, where A_i is stored locally, transmitting the value of $LCSS(Q, A_i)$ towards QN (i.e., the decentralized way). Alternatively, we could have also fetched the trajectories of the S-set to the sink and then compute $LCSS(Q, A_i)$ $\forall A_i \in S$ (i.e., the centralized way), however this would violate both the trace disclosure constraint and also degrade the response time of the algorithm to a level comparable to the slow centralized algorithm. Notice that the fourth step of the algorithm applies only to the elements in the S-set, as opposed to all m elements so this is really much cheaper in terms of energy consumed on the smartphone as $|S| << m$.

In step 5, we determine whether the algorithm has reached the termination condition. In particular, we check if the (λ+1)-th highest UB is smaller than the K-th highest full matching value. If this is the case, then we can safely terminate the execution of the algorithm being sure that the correct top-K has been identified. If this condition does not hold (i.e., when the UB of an object X is larger than the K-th highest full matching value Y), then we are enforced to perform another iteration as the answer is not deterministic (i.e., either X or Y can be the K-th answer). Consequently, we increase the step increment λ so that it identifies the next λ candidates in the next round.

In the final step, which occurs only once at the very end, we might ship each matched subsequence A_i^{match} ($|A_i^{match}| << |A_i|$) to QN, which can then return it to the user. Notice, that the identified nodes A_i ($i \leq K$) might choose not to share the matching or share it based on some local trace disclosure profile [12], in order to preserve k-anonymity and other higher anonymity schemes. In any case, neither QN nor the querying user will ever see the complete trajectory of participating users.

Example: Consider the example scenario of Figure 4. Assume that Q seeks to retrieve the top-2 most similar trajectories ($K = 2$). Initially, QN sends Q to all nodes. Each node then computes an upper bound of its trajectory, in respect to Q, and sends this value to QN. Subsequently, QN proceeds by determining the trajectories with the λ+1 highest *METADATA* entries, i.e., $\{A_4, A_2, A_0\}$, adding the λ trajectories to the S-set, i.e., $S = \{A_4, A_2\}$ (steps 2-3). In step 4, QN asks the smartphones in S to compute the full matching of Q to A_i ($A_i \in S$) without unveiling their A_i. The full matching scores, which are transmitted to QN, are: $LCSS(Q, A_4) = 23$ and $LCSS(Q, A_2) = 22$. Since the $(\lambda+1)$-th highest UB $(A_0, 25)$ is larger than the K-th highest full match $(A_2, 22)$, the termination condition is not satisfied in the fifth step. Therefore, the second iteration of the *ST* algorithm is initiated to compute the next λ, ($\lambda = 2$), full matching scores: $LCSS(Q, A_0) = 16$, $LCSS(Q, A_3) = 18$. Now the termination has been satisfied because the new $(\lambda + 1)$-th (i.e., $2\lambda + 1$) highest UB $(A_9, 18)$ is smaller than the K-th highest full match $(A_2, 22)$. Finally we tentatively might return the top-2 matched subsequences of trajectories with the highest full matches to the user (i.e., $\{(A_4^{match}, 23), (A_2^{match}, 22)\}$). Even if we return these subsequences to the querying node, it is important to mention that these are not the complete trajectories A_4 or A_2, but only the subsequences that correspond to the matching (e.g., one route out of their two year history log, given that the owner has agreed to release them.)

Theorem 1 (ST Correctness) *The* ST *algorithm always returns the most similar objects to the query trajectory Q.*

Proof: Let A denote some arbitrary object returned as an answer by the *ST* algorithm ($A \in Result$), and B some arbitrary object that is not among the returned results ($B \notin Result$). We want to show that $LCSS(Q, B) \leq LCSS(Q, A)$ always holds. Assume that $LCSS(Q, B) > LCSS(Q, A)$. We will

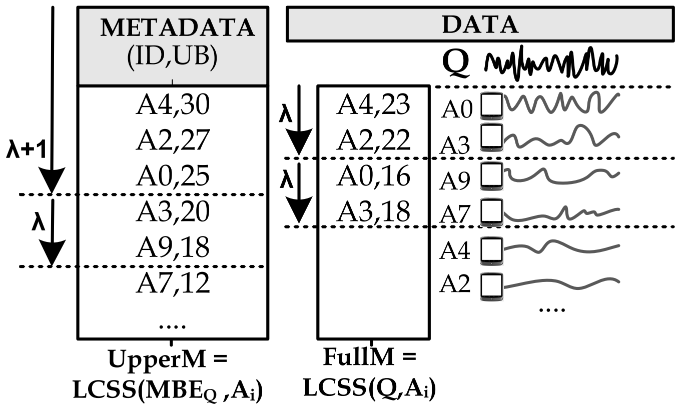

Fig. 4. Example execution of the *ST* algorithm.

show that such an assumption leads to a contradiction. In our analysis, we cover the two possible cases: i) B is not in the set of candidate objects (denoted as S-set) during the third step; and ii) B becomes part of the S-set during some arbitrary iteration of the algorithm in the third step. We will show that both cases yield a contradiction.

Case 1 ($A \in S$ and $B \notin S$): Since $B \notin S$, then $UB(Q, B) < LCSS(Q, X)$, where X is the K-th highest object in the array $LCSS$, by the termination condition of the algorithm (Step 5). Thus, $LCSS(Q, B) \leq UB(Q, B) < LCSS(Q, X) < ... < LCSS(Q, A)$, because object A is in the final *Result*. Hence, it holds that $LCSS(Q, B) < LCSS(Q, A)$, a contradiction.

Case 2 ($A \in S$ and $B \in S$): Since both objects A and B are now part of S, the full scores of A and B are known, by step 4 of the algorithm. By the initial assumption we know that only object A belongs to the final *Result*. Thus, $LCSS(Q, B) \leq LCSS(Q, A)$, a contradiction □

Theorem 2 (ST λ-Convergence) *Algorithm* ST *performs $O(m/\lambda)$ iterations in the worst case.*

Proof: To prove the statement of the theorem we assume that all upper bound values in *METADATA* are larger than the K-th highest full matching value (i.e., the K-th highest value in $LCSS$). In this case, the termination condition of step 5 is satisfied at the very end. Consequently, *ST* performs O(m/λ) iterations in the worse case □

3.3 The Non-Iterative SmartTrace (NIST) Algorithm

In this subsection, we extend the iterative *ST* retrieval algorithm with a non-iterative counterpart, called *Non-Iterative SmartTrace (NIST)*. The *NIST* algorithm operates in two phases only, as opposed to a $O(m/\lambda)$-bounded number of phases needed by *ST*, thus dramatically improves response time at a slight increase in network traffic. Similarly to the *ST* algorithm, the *NIST* algorithm identifies the correct answer set with neither revealing the complete target trajectories to QN (i.e., it only returns the matched subsequence) nor revealing the identity of a participating user (i.e., unless the user decides to do so.) Contrary to the *ST* algorithm, the *NIST* algorithm has the following unique characteristics: (i) It uses both an upper bound (UB) and a lower bound (LB) in order to determine whether the top K trajectories have been found; and (ii) Full

Algorithm 2 : Non-Iterative SmartTrace (NIST)

```
Input: Query Trajectory $Q$, $m$ Crowd Trajectories, Result Cardinality $K$ ($K << m$)
Output: $K$ trajectories most similar to $Q$.
At the query node QN:
  1) UB and LB Computation: Instruct each of the $m$ smartphones in the crowd to invoke a computation of the linear-time $LCSS(MBE_Q, A_i)$ and $LCSS(LB_Q, A_i)$ ($i \leq m$) functions, respectively.
  2) Collection of UB and LB: Receive the UBs and LBs of all $m$ trajectories participating in the query and add those scores to the METADATA vector stored on $QN$. Let METADATA be sorted in descending order based on the UB scores.
  3) Identify Candidates: Find the $K$-th highest LB in METADATA setting it as the cut-off threshold $\tau$. Add the identities of the $K$ trajectories $A_i$ with $LB_i \geq \tau$ to an empty set $S$ (denoted as the candidate set). Enumerate the remaining $m - K$ trajectories adding to the $S$-set the identity of any $A_i$ that has an $UB_i \geq \tau$.
  4) Full Computation: Ask each element in the $S$-set to compute $LCSS(Q, A_i)$, in a decentralized manner, and then send back their full similarity scores. Finally identify the real top-K answers based on these scores.
  5) Ship Matching: Tentatively ship the respective matches to $QN$, based on some local trace disclosure policy.
```

matching values are computed in one final step, rather than iteratively.

Description: In step 1 of the *NIST* algorithm (see Algorithm 2), QN instructs all m nodes to invoke concurrently in a single scan of A_i, the computation of the $LCSS(MBE_Q, Ai)$ function and the $LCSS(LB_Q, Ai)$ ($i \leq m$) function.

In step 2, QN retrieves the UBs and LBs and adds them to the local *METADATA* vector in descending order in respect to UB. By doing so, QN obtains a quick summary of the trajectories similar to Q. Subsequently, in step 3, QN locates the K-th highest LB in *METADATA* identifying this value as the cut-off threshold τ. The intuition is that any trajectory with an upper bound value below τ can safely be pruned away without affecting the top-K result. Specifically, all trajectories with an $UB_i \geq \tau$ are added to an empty set S denoted as the candidate set.

In step 4, QN asks each element in the S-set to locally compute $LCSS(Q, A_i)$, and transmit this single value per node towards QN. Contrary to the decentralized algorithm, this step executes $LCSS(Q, A_i)$ only on a limited number of smartphones (i.e., those with UBs exceeding the τ threshold.) In the final step, we might ship each matched subsequence A_i^{match} ($|A_i^{match}| << |A_i|$) to QN and subsequently to the user, if this is compatible with the trace disclosure profile of the user.

The intuition behind the *NIST* algorithm, compared to the *ST* algorithm, is that the termination condition can be identified based on τ, rather than the full matching. Therefore, we are not required to incrementally ask the smartphones to compute full matching values and send them to QN, incurring additional network traffic and energy consumption, but we can ask them all together in a single step.

Example: Initially, QN sends Q to all nodes, which subsequently initiate the computation of the linear-time LB and UB scores in respect to Q (step 1). These scores are subsequently transmitted over to QN (step 2) and organized in a max-heap structure based on the UB values. Figure 5 shows these bound

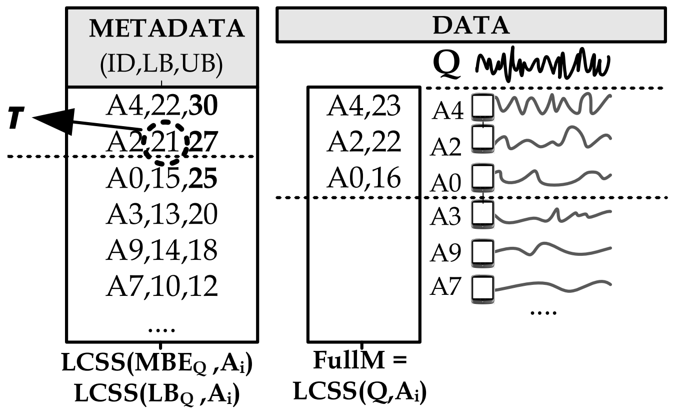

Fig. 5. Example execution of the *NIST* algorithm.

values for our working example that has K=2.

In step 3 of the *NIST* algorithm, QN locally proceeds by determining the τ cut-off threshold, which is 21 in our example (i.e., the K-th highest lower bound value.) All trajectories A_i with an UB_i larger than τ are added to the empty S-set. In particular, since $UB_4 = 30$, $UB_2 = 27$ and $UB_0 = 25$ the candidate set is formed as follows: $S = \{A_4, A_2, A_0\}$. The remaining trajectories can safely be pruned-away as $UB_3 = 20$, $UB_9 = 18$ and $UB_7 = 12$ are all smaller than $\tau = 21$.

The full matching scores of the fourth step, which are transmitted to QN, are: $LCSS(Q, A_4) = 23$, $LCSS(Q, A_2) = 22$ and $LCSS(Q, A_0) = 16$. Based on these final scores, QN can derive A_4 and A_2 as the correct answer, Finally, we tentatively might return these trajectories to the user that posted the query if A_4 and A_2 are not restricted by their owners and if the querying node decides to do so.

Theorem 3 (*NIST* Correctness) *The* NIST *algorithm always returns the most similar objects to the query trajectory Q.* □

For brevity, we have omitted the proof of this theorem, which is similar to that of Theorem 2, with the difference that we utilize the cut-off threshold τ to trigger the stopping condition of the algorithm.

4 Analytical Evaluation

In this section we analytically derive the performance of the ST and *NIST* algorithms in respect to Time and Energy. We adopt worst-case analysis as it provides a bound for all input. Our experimental evaluation in Section 6, shows that our algorithms perform under realistic and real datasets much more efficiently than the projected worst-case.

4.1 Cost Model

Let m smartphone users $\{A_1, \cdots, A_m\}$ participate in the execution of query Q, initiated at QN. Let the maximum length among all trajectories be denoted as l and $|Q| << l$, as explained earlier. All smartphones are connected to QN for the complete duration of Q's execution through some established connection (e.g., persistent TCP socket).

We are interested in deriving analytically the *Time* ($\mathcal{T}$) and *Energy* ($\mathcal{E}$) costs for resolving Q. $\mathcal{T}$ is defined as the length of time it takes for Q to be sent to the m users plus the length of time it takes for the final top-K result to be received at

QN (i.e., *the user-perceived latency for resolving* Q, formally $\mathcal{T} = MAX_{i=1}^{m}(\mathcal{T}_i)$ as the smartphones operate in parallel.) On the other hand, $\mathcal{E}$ is defined as the total energy cost incurred on each smartphone for answering Q (i.e., the *client-perceived energy consumption*). Notice that the total time a smartphone spends on a query, as opposed to $\mathcal{T}$, is naturally captured by $\mathcal{E}$, which is equal to $\sum_{i=1}^{m}(Power_i \times \mathcal{T}_i)$, where Power is measured in Watts (i.e., Volts $\times$ Amperes).

Notice that in our cost analysis we deliberately do not focus on the *Messaging* and *Bandwidth* costs, because measuring these in isolation will not expose the relevant complexities of a smartphone network environment, as explained in Section 1. For instance, a protocol with a high message complexity might transmit many small-size messages, thus consuming very little bandwidth. For ease of exposition, our analysis should use the notation $\{\mathcal{E}|\mathcal{T}\}^{CPU}$, $\{\mathcal{E}|\mathcal{T}\}^{TX}$ and $\{\mathcal{E}|\mathcal{T}\}^{RX}$, to denote the energy or time cost for *processing*, *transmitting* and *receiving* one trajectory point (^{TX} and ^{RX} also capture the incurred processing costs during communication and are approximately equivalent.) Our analysis will present the energy and time costs per participating device (i.e., user-perceived costs). We will ignore any other irrelevant energy consumption costs, such as LCD, Bluetooth, etc. Finally, the network between the server and the devices is bounded by a fixed uplink capacity, thus the time to concurrently upload m trajectories is defined as a function of m.

4.2 Performance Analysis

Lemma 3 (Centralized Performance): *The Centralized algorithm has an Energy and Time complexity of* $O(l \cdot \mathcal{E}^{TX})$ *and* $O(m \cdot l \cdot \mathcal{T}^{TX})$*, respectively.* $\square$

Lemma 4 (Decentralized Performance): *The Decentralized algorithm has an Energy and Time complexity of* $O(\delta \cdot l \cdot \mathcal{E}^{CPU})$ *and* $O(\delta \cdot l \cdot \mathcal{T}^{CPU})$*, respectively.* $\square$

We have omitted the proof for Lemmas 3 and 4, as these are trivial and follow directly from the definition of the respective techniques and our system model.

Lemma 5 (ST Performance): *The* ST *algorithm has an Energy and Time complexity of* $O(\delta \cdot l \cdot \mathcal{E}^{CPU})$ *and* $O(\frac{m}{\lambda} \cdot \delta \cdot l \cdot T^{CPU})$*, respectively.*

Proof (direct): In the first (1) step, each node receives Q, which costs $|Q| \cdot \mathcal{E}^{RX}$ ($|Q| << l$) energy. Then each A_i invokes the linear-time $LCSS(MBE_Q, A_i)$ ($i \leq m$) computation, which costs $|Q| \cdot \mathcal{E}^{CPU}$, as $LCSS(MBE_Q, A_i)$ can be computed in $O(min(l, |Q|))$ and $|Q| << l$. Finally, each node transmits back a single scalar value (i.e., $\mathcal{E}^{TX}$). The second and third steps of the algorithm have no smartphone-side incurred costs. Steps 4 and 5 are executed in m/λ iterations in the worse case. In each of the iterations, we have the following costs: In step 4, the λ identified nodes execute the $LCSS(Q, A_i)$ ($i \leq m$) computation and that costs $\delta \cdot (|Q| + l) \cdot \mathcal{E}^{CPU}$, as LCSS has a time complexity of $O(\delta \cdot (l_1 + l_2))$. Yet, this cost incurs on only a few nodes (i.e., $\lambda << m$). When this operation completes, a single scalar value is shipped from each of the λ nodes to QN costing $\mathcal{E}^{TX}$. Step 5 has again no smartphone-side incurred energy cost as it takes place on QN. Also step 6 is computed only once at the very end and costs $|Q| \cdot E^{TX}$. By adding up all aforementioned values in an asymptotic manner yields an energy complexity of $O(\delta \cdot l \cdot \mathcal{E}^{CPU})$, as all other factors are small order and can thus be eliminated.

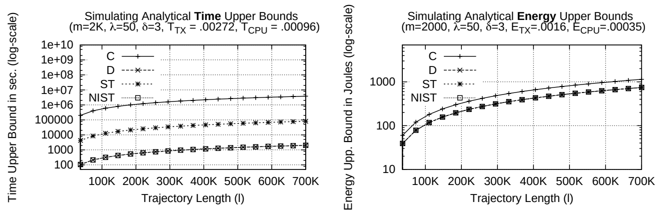

Fig. 6. Projecting the Time and Energy upper bounds for various trajectory lengths.

Similarly to the above analysis, the time complexity of ST is defined as $O(\frac{m}{\lambda} \cdot \delta \cdot l \cdot T^{CPU})$, as we are now only waiting for the slowest node during the computation of $LCSS(Q, A_i)$ and we conduct this m/λ times at most $\square$

Lemma 6 (*NIST* Performance): *The* NIST *algorithm has an Energy and Time complexity of* $O(\delta \cdot l \cdot \mathcal{E}^{CPU})$ *and* $O(\delta \cdot l \cdot \mathcal{T}^{CPU})$*, respectively.*

Proof (direct): Similarly to *ST* in the first (1) step, each node receives Q, which costs $|Q| \cdot \mathcal{E}^{RX}$ ($|Q| << l$) energy. Then each A_i invokes the linear-time $LCSS(MBE_Q, A_i)$ and $LCSS(LB_Q, A_i)$ ($i \leq m$) computations, which cost $|Q| \cdot \mathcal{E}^{CPU}$, as both functions are computed in the same scan with an O(1) cost per point (i.e., for $LCSS(MBE_Q, A_i)$ we compare against the pre-constructed envelope and $LCSS(LB_Q, A_i)$ is a closed-form equation.) Subsequently, each node transmits back a single scalar value (i.e., $\mathcal{E}^{TX}$) in the second step. In the third step QN locally identifies the τ cut-off threshold, thus incurs no smartphone-side costs. In the fourth step, each node in the candidate S-set is asked to perform a $LCSS(Q, A_i)$ computation. By adding up all aforementioned values in an asymptotic manner yields an energy complexity of $O(\delta \cdot l \cdot \mathcal{E}^{CPU})$, similarly to the *D* and *ST* algorithms. The time complexity for *NIST* is defined as $MAX_{i=1}^{m}(\delta \cdot l \cdot \mathcal{T}_i^{CPU}) \in O(\delta \cdot l \cdot \mathcal{T}^{CPU})$, as the computation is carried out in parallel on all devices $\square$

Discussion: We shall next summarize our key findings regarding the Time and Energy upper bounds of the four algorithms, *C*, *D*, *ST* and *NIST*. Those findings are complemented by a simulation of the respective bounds in Figure 6, with time and energy constants derived from our experimental testbed in Section 6. The first observation from the analytical bounds and the curves of Figure 6 (left) is that *C* has the highest worst-case time bound, of $O(m \cdot l \cdot \mathcal{T}^{TX})$, since multiple concurrent uploads limit the bandwidth available to each device. *ST* has the second worst-case time bound of $O(\frac{m}{\lambda} \cdot \delta \cdot l \cdot T^{CPU})$, since the algorithm might execute the LCSS method consecutively for m/λ times in the worst-case. Fortunately, this worst-case scenario never occurred in our experimental evaluation of Section 6. Contrary to *C* and *ST, NIST* and *D* maintain the lowest time bounds than the other two methods as these solely rely on the time it takes to execute LCSS on the longest participating trajectory, locally on each smartphone.

The second observation is that C continues to expose the worst-case energy bound of $O(l \cdot \mathcal{E}^{TX})$, compared to

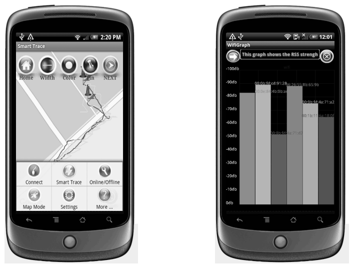

Fig. 7. Screenshots of the SmartTrace$^+$ client for outdoor environments with GPS and indoor environments with RSS signals.

$O(\delta \cdot l \cdot \mathcal{E}^{CPU})$ exposed by the other three methods. Clearly, the determining factor is that $\mathcal{E}^{TX}$ is approximately one order of magnitude larger than $\mathcal{E}^{CPU}$, as processing a single trajectory point can be conducted much more efficiently than transmitting the same point over an expensive wireless link.

5 The SmartTrace$^+$ Prototype System

In this section we describe a prototype system we have developed for SmartTrace$^+$ using the Android OS. We provide an overview of the communication protocol and the Graphical User Interface. Additional deployment details can be found in [14].

5.1 Overview

Our client-side software is developed on top of the ubiquitous Android OS and its installation package (i.e., APK) has a size of 1,106KB. Our code is written in JAVA and consists of approximately 9,000 lines-of-code (LOC). In particular, our server-code uses $\approx$ 3,470 LOC and runs over JDK 6 and Ubuntu Linux, while our smartphone code uses $\approx$ 5,530 LOC plus $\approx$ 250 lines of XML elements that go the Manifest file (settings) and the user interface XML descriptions. In the future, we plan to port the computationally and IO-intensive tasks outside the VM by implementing them in native (C) code using the Android NDK.

5.2 Graphical User Interface (GUI)

Our prototype GUI provides all the functionalities for a user participating in SmartTrace$^+$. The GUI is divided into an outdoor and an indoor interface, respectively as shown in Figure 7. The outdoor interface uses WGS84 encoded GPS trajectories along with the Android Google Maps API, in order to visualize the traces on a map. The indoor interface uses our proprietary WiFi Access Point (AP) format, which captures multi-dimensional signal strength values collected from nearby APs (i.e., each AP is identified by its network MAC address and its signal strength is measured in dBm.)

At a high level, our GUI enables the following functions: i) record traces on local storage and plot those on the screen for the outdoor case, ii) configure various logging and querying features (e.g., K, δ and ϵ); iii) connect to a SmartTrace$^+$ server and query the traces stored on other connected nodes, and iv) switch between *online* and *offline* mode to change between experimentation and real operation.

5.3 Protocol

In this section, we provide an abstraction of the TCP/IP-based data transmission protocol that lies at the foundation of SmartTrace$^+$. We implemented a text-based protocol, as opposed to a binary protocol, for portability reasons (i.e., endianness). We also did not chose an XML-based protocol implementation for performance reasons.

In the scenario that follows, a user $A1$ connects to a Query Node (QN), with a screen-name `cs7239`, requesting the execution of a top K=1 query using the `SEARCH` command (we assume the execution of the *ST* algorithm). QN responds with an MD5 `query-ID` hashcode, which serves as the query identifier. Notice that each request to QN is preceded by an `+OK` or `-ERR <code>` message, denoting the status of the request. Subsequently, QN, requests the execution of the *ST* algorithm over four other connected nodes, e.g., $A4, A5, A8, A9$. In particular, $A4$ initially executes the $LCSS(MBE_Q, A4)$ method, with the `UB` method, and then $LCSS(Q, A4)$, with the `LCSS` method.

All connected nodes conduct the operation and return to QN a `VAL` message which includes the query-id, type (1:ub, 2:lb,ub or 3:real_score) as well as a 4-byte double value. After finalizing the computation, QN returns the top-K result to $A1$. Finally, in our example $A1$ also asks for the complete matched trajectory using an out-of-band `RETR` command. The matched subsequence is routed to $A1$ through QN with the `TRAJ` command, given that $A4$ has a trace-sharing option enabled in its local profile.

```
QN (to A1): +OK READY
A1 (to QN): USER cs7239
A1 (to QN): SEARCH <k> <query-trace>
QN (to A1): +OK <query-ID>
QN (to A4): UB <query-ID>  <query-trace>
A4 (to QN): +OK VAL <query-ID> 1 30.0
QN (to A4): LCSS <query-ID>
A4 (to QN): +OK VAL <query-ID> 3  23.0
-- repeat in parallel for all nodes A5,A8,A9
QN (to A1): +OK <results>
-- tentative ship matching step
A1 (to QN): RETR <query-ID>
QN (to A4): RETR <query-ID>
A4 (to QN): TRAJ <query-ID> <result-trace>
QN (to A1): TRAJ <query-ID> <result-trace>
```

6 Experimental Evaluation

In this section we present an extensive experimental evaluation of the *ST* and *NIST* algorithms. Our experiments are conducted in two modes: i) *Trace-driven Simulation* on a PC; and ii) *Trace-driven Deployment* on our SmartLab Cluster of 25 real smartphones. We start our description with our experimental methodology and then proceed with the presentation of our results.

6.1 Methodology

Datasets and Queries: We use the following three datasets:

i) *Oldenburg* [9]: This medium-scale dataset includes 2,000 car trajectories moving in the city of Oldenburg [9]. The average length of each trajectory is $11,731 \pm 7,193$ points, while the maximum trajectory length is 42,500 points. Each query for the above dataset is randomly derived from the initial dataset. Our queries have an average size of 100 points.

ii) *GeoLife-A* [49], [50]: This large-scale dataset, by Microsoft Research Asia, includes 1,100 trajectories of a human moving in the city of Beijing over a life span of two years (2007-2009). The average length of each trajectory is $190,110 \pm 126,590$ points, while the maximum trajectory length is 699,600 points. Notice that 95% of the GeoLife dataset refers to a granularity of 1 sample every 2-5 seconds or every 5-10 meters. Our queries are randomly sampled from the dataset and have an average size of 67 points.

iii) *GeoLife-B* [49], [50]: This smaller-scale dataset, again by Microsoft Research Asia, has been derived to accommodate the hardware limitations of our programming cloud testbed. In particular, this dataset includes 25 trajectories whose average trajectory length is $25,058 \pm 3,062$ points, the maximum trajectory length is 25,850 points and the query length is 100 points.

Algorithms and Metrics: We compare the *SmartTrace* (*ST*), *Non-Iterative SmartTrace* (*NIST*), *Decentralized* (*D*) and *Centralized* (*C*) algorithms, under a variety of settings using the datasets described earlier. Our cost metrics are: *Time* ($\mathcal{T}$) and *Energy* ($\mathcal{E}$), as documented in Section 4, for varying m, K and λ parameters. For the *C* method, we do not take into account the time and energy spent on finding the answers on the query processor. The reason for neglecting this cost is that a centralized query processor can be ”infinitely” powerful with an ”infinite” power source (i.e., compared to the power-limited smartphones). Whenever we test one parameter, the complementary parameters are fixed to the following values: m to 2,000 and 1,100 for the Oldenburg and the GeoLife datasets, respectively, K to 2 and λ to 50 for the *ST* algorithm. The δ and ϵ parameters are kept constant for each dataset as those are application specific (i.e., they attempt to capture a reasonable matching scenario given the spatio-temporal coordinates in their respective dataset). Varying δ and ϵ should not affect our execution scenario in any sense, as this would simply vary the matching granularity in all algorithms. All measurements are averaged over 10 consecutive runs.

Network and Energy Model: Our communication protocol is associated with a 45 byte header (including node identifier, session identifier and other application specific parameters). In our setting, a spatio-temporal *DATA* point (18 bytes) consists of a *timestamp* that occupies 8 bytes, two 4-byte fields for the GPS coordinates and another 2 bytes for direction. In reality this overhead might be even higher (e.g., GeoLife trajectories include elevation, speed, heading direction and accuracy.) However, we omit these additional attributes as they are not necessary for computing the basic edition of our algorithm that relies only on the spatio-temporal attributes. Had we used them should have boosted the competitive advantage of *ST* and *NIST* over *C* and *D* even more. The *METADATA* comprise only a single 4-byte real field per node for the *UB*, *LB* or *LCSS* matching value. QN runs on a single host that connects to the m smartphones using an 802.11b network link that has a TCP downlink of 1022kbps and a TCP uplink of 123kbps, a 237ms TCP handshake latency and application handshake latency of 493ms (as measured with [1]). Our energy profile has been derived by running SmartTrace^{+} instances using PowerTutor [36]. In particular, CPU Idle (OS running) = 175mW, CPU Busy (Processing) = 369mW, WiFi Idle (Connected) = 38mW and WiFi Busy (Uplink 123Kbps, -58dBm) = 600mW.

6.2 Series 1: Varying the Number of Trajectories (m)

In the first experimental series we investigate the performance and scalability of our approach in respect to m. Figure 8 presents our Time ($\mathcal{T}$, left) and Energy ($\mathcal{E}$, right) results for the Oldenburg (top) and GeoLife-A (bottom) datasets, respectively. In order to generalize our conclusions to larger datasets in this series, we have synthetically increased the size of both these datasets, up to $m = 100,000$, by keeping the same distribution in the base dataset. We will start out with our main findings that concern the Oldenburg and Geolife-A datasets and then explain the results for scaling these datasets.

Analysis of Response Time ($\mathcal{T}$): The plots in the left column show that both the *NIST* and *D* algorithms consume one order of magnitude less $\mathcal{T}$ than the *C* algorithm, while *ST* consumes somehow higher but is still more efficient than *C*. In particular, for Oldenburg we observe the following average time values in seconds for the bars in our plot: *NIST*=36 ± 7, *D*=39 ± 7, *ST*=170 ± 95 and *C*=809 ± 478; while for GeoLife-A, we observe the following time values in seconds: *NIST*=614 ± 35, *D*=768 ± 2, *ST*=$1,963 \pm 550$ and *C*=$8,623 \pm 3,804$ seconds, respectively. The above results are attributed to the fact that *NIST*, *ST* and *D* mainly rely on local processing, while *C* relies on transmission, as this was shown in our analytical study.

The reason why *ST* performs worse that *D*, is that *ST* performs one *LCSS(Q,A)* iteration per round in an incremental fashion, while *D* conducts *LCSS(Q,A)* on all nodes in a single step. Such an oblivious act by *D*, consumes a lot of energy, as shown in the Energy plot analysis.

A final interesting observation regarding the time plots, is that although the worst-case time complexity of *NIST* looks the same with the respective time complexity of *D*, our experiments reveal that *NIST* is faster than *D* for the two datasets, by 3 seconds and 154 seconds, respectively. By carefully analyzing our traces, we found that this is attributed to the variable length of trajectories in our datasets. In particular, *D* is always condemned to perform the worst-case (i.e., to process the longest trajectory in its computation), while *NIST* will process these very long traces only if they belong to the top-K result.

Although in our setup a query is expected to be localized in a specific area with a limited number of users, for completeness, we have also validated our findings for very large target deployments (i.e., up to 100K nodes). Our findings in the same plots of Figure 8, show that the general trends remain the same. We additionally notice that the *ST* algorithm now obtains an execution time that is somehow too large for a realistic querying system (i.e., close to 2 hours). Contrary, the *NIST* algorithm maintains a reasonable response time (i.e., 47s and 10min, respectively), for even such large-scale scenarios.

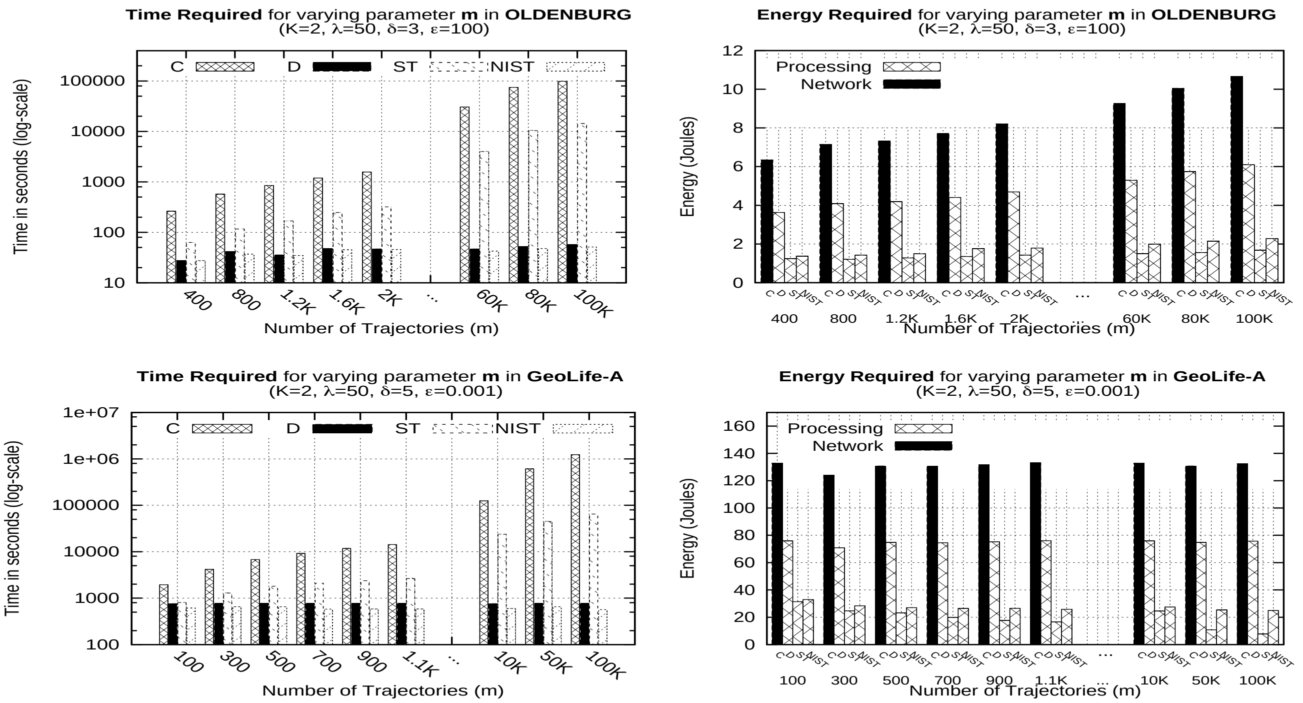

Fig. 8. Series 1: Varying the number of trajectories (m). Time and Energy results for Oldenburg and GeoLife-A.

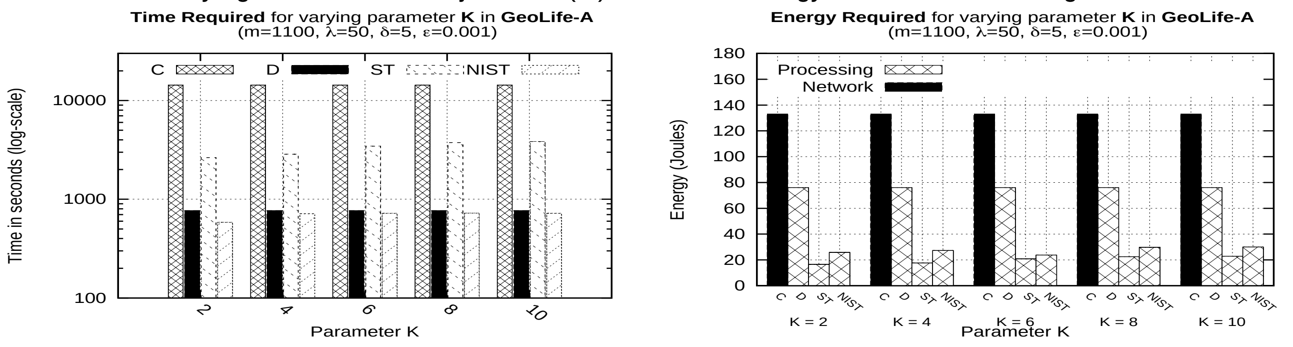

Fig. 9. Series 2: Energy (left) and Time (right) results for varying K in Geolife-A.

Analysis of Energy Consumption ($\mathcal{E}$): While somebody might claim that the competitive advantage of *NIST* over *D*, in respect to $\mathcal{T}$, is not that great (i.e., 8% and 21%, respectively), we shall next also study the incurred energy costs per device.

For the Oldenburg dataset (i.e., Figure 8 right/top), we observe that there is a 70% and 62% percent advantage of *ST* and *NIST* over *D*, in respect to $\mathcal{E}$. This result validates that the *D* algorithm consumes large amounts of energy while both *ST* and *NIST* minimize this amount significantly. The same plot also shows that *NIST* is somehow more energy demanding than *ST*. This happens as the *ST* algorithm executes *LCSS(Q,A)* on fewer nodes while *NIST* conducts *LCSS(Q,A)* on usually more nodes, thus has a higher possibility of processing longer trajectories. The above discussion reveals the trade-off between the two algorithms, i.e., "ST *is slower but consumes less energy, while* NIST *is faster but consumes more energy"*. Consequently, one might chose either algorithm, depending on the primary optimization criterion.

Another observation is that the *C* algorithm spends all its energy on network operations, i.e., $O(l \cdot \mathcal{E}^{TX})$, while *NIST*, *ST* and *D* spend the bulk of their energy on smartphone-side processing operations. In fact, the networking costs for these algorithms is as small as $2.59mJ$ per query (this is why they don't show up in the plots). The above result, confirms that the network overhead of the *ST* algorithm is not high, as it transmits smaller size packets as opposed to the large and monolithic packets used by *C*. One final observation is that the D algorithm will execute the *LCSS(Q,A_i)* function on all nodes, thus the aggregate energy costs are orders of magnitude higher than the respective energy costs for the *NIST* algorithm, which executes *LCSS(Q,A_i)* on only those nodes in the S-set. Similar conclusions to the above, can also be drawn for the Geolife-A dataset in Figure 8 right/bottom.

In order to gain a deeper understanding on the energy savings of our algorithms, we have calculated the size of the candidate S-set, constructed during the operation of the *ST* and *NIST* algorithms, respectively. Our findings indicate that the average "|S-set| / m" ratio, where m is the number of all trajectories, in Oldenburg is 24% and 33%, for *ST* and *NIST*, respectively. For Geolife-A the average "S-set / m" ratio is 24% and 29% for *ST* and *NIST*, respectively. For all four cases the standard deviation is around 4%. This suggests that *NIST* will consume more energy than *ST*, in processing the larger S-set, as confirmed by our energy plots.

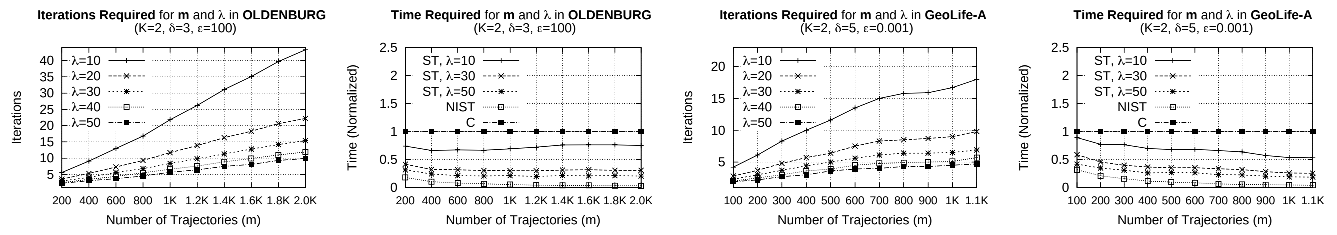

Fig. 10. Series 3: Varying the iteration step increment (λ) in SmartTrace. Number of iterations and Time Results for Oldenburg and GeoLife-A.

6.3 Series 2: Varying the Size of the Answer-set (K)

In the second experimental series, we investigate the performance and scalability of our approach in respect to K. For brevity, we will only present the results for the GeoLife-A dataset, but the respective Oldenburg results look very similar. We will additionally not cover the observations already discussed previously.

Figure 9, shows that the performance of all four algorithms is independent of K for both the $\mathcal{T}$ and $\mathcal{E}$ results. For this experiment K is increased from 2 to 10, which refers to $\approx 1\%$ of m. The results in this series are confirmed by our analytical study where we have shown that all four algorithms are independent of K (see Lemmas 3-6). In particular, the $\mathcal{E}$-complexity of both *ST* and *NIST* is $O(\delta \cdot l \cdot \mathcal{E}^{CPU})$, while the $\mathcal{T}$-complexity of *ST* and *NIST* is $O(\frac{m}{\lambda} \cdot \delta \cdot l \cdot T^{CPU})$ and $O(\delta \cdot l \cdot \mathcal{T}^{CPU})$, respectively. However, in practice we should expect that both the *ST* and *NIST* algorithms would perform worse than *C* or *D*, if K is very large as they involve some messaging overhead. Yet, top-K queries are not designated for large values of K as this is explained in Section 7. Also these workloads are not useful in our setting as a user would be overwhelmed with many less relevant answers.

6.4 Series 3: Varying the Iteration Step Increment (λ)

In the third series, we present an improved study over our prior work in [48][6], of how the iteration step increment λ affects the convergence of the *ST* algorithm. In particular, we present the number of iterations (and the respective time) the given algorithm takes for different values of λ and m.

Figure 10 shows our results for the Oldenburg and GeoLife-A datasets, respectively. The first observation is that the more aggressive λ gets, the quicker the *ST* algorithm converges. In particular, we observed that the two datasets feature an average number of iterations equal to 6.05 and 3.55, respectively, for $\lambda = 50$. Additionally, we also observe that the number of iterations grows almost linearly by increasing m. Both aforementioned observations are explained by the result of Theorem 2, where we showed that *ST* requires O(m/λ) iterations in the worse case. Interestingly, we mention that this worse case has not happened in any of our experiments, but setting this parameter optimally through some learning phase will be a subject of future research.

The time plots in the same figure Figure 10, validate that *ST* is inversely proportional to λ, i.e., Lemma 5 showed that $\mathcal{T}_{ST} \in O(\frac{m}{\lambda} \cdot \delta \cdot l \cdot T^{CPU})$. Notice that in the given experiment λ is ranging between 1% - 5% of m, which is larger than the K value we used in this study (i.e., up to 1%). Had we been more aggressive would certainly improve the response time but would have incurred unecessary LCSS computations on many nodes not in the answer-set. Although setting λ in an optimal manner would require some additional structures, configuring it to approximately 5% of m worked great for the tests we have conducted. Finally, we observe that the λ-independent algorithm *NIST* performs faster than all *ST* versions, but requires more energy as explained previously.

6.5 Series 4: Prototype Evaluation

In the final experimental series, we deploy instances of our real prototype system in Android over our SmartLab testbed [7]. We utilize the same δ and ϵ settings with smaller K and λ parameters (i.e., 1), due to the smaller size of the dataset. In order to measure power consumption in a meaningful way, we utilize PowerTutor [36], which has an average error less than 10%. All messaging goes through the 802.11b WiFi interface. Notice that Android uses a standard Linux Kernel 2.6 and each program runs within a Dalvik Java Virtual Machine, which limits the memory heap of an application to 24MB.

The Time ($\mathcal{T}$) and Energy ($\mathcal{E}$) results for our evaluation using the GeoLife-B dataset are summarized in Table 2. In respect to $\mathcal{T}$, our first observation is that *NIST* is indeed the fastest approach for retrieving the top-K answers from the distributed devices, as it performs in 24 seconds on average. The second observation is that *ST* behaves better than the *D* algorithm. By analyzing the logs we found that for this series *ST* converged with the answer very quickly (i.e., in 2 rounds), thus did both not execute LCSS(Q,A) on many nodes and also had a lower probability of running the LCSS(Q,A) method on a long trajectory. One final observation is that the *C* algorithm still requires the most time to answer the query, although it behaves somehow better than what we observed in the simulations.

In respect to $\mathcal{E}$, we observe that our results are compatible with the general trends that were analyzed in our previous series. In particular, it is confirmed that the *ST* algorithm is the most efficient among its competitors with *NIST* being more energy demanding (i.e., by 38%), while both *D* and *C* increase the gap from *ST* by 67% and 74%, respectively. Energy efficiency is becoming an important parameter in data management systems [41] and SmartTrace^{+} is addressing this parameter in its core execution algorithms.

6. The improvements relate to our implementation of MBE and a more accurate way to generate the query-set.

7. Available at: http://smartlab.cs.ucy.ac.cy/

TABLE 2
Series 4: Evaluating our prototype on SmartLab.

Algorithm	Time ($\mathcal{T}$) (Seconds)	Energy ($\mathcal{E}$) (Joules)
C	68	17.4543
D	36	13.7668
ST	28	4.5763
NIST	24	7.3186

7 Related Work

In this section we provide related research work for both spatio-temporal query processing and distributed top-K query processing, both of which lie at the foundation of SmartTrace$^+$.

Spatio-temporal queries have been an intense area of research over the years [3], with the development of efficient access methods [25], [44], [32] and similarity measures, such as *Dynamic Time Warping (DTW)* [7], the *Longest Common Subsequence (LCSS)* [15], variants of L_p-norms such as *Edit Distance with Real Penalty (ERP) [27]* and *Edit Distance on Real Sequences (EDR)* [28]. These metrics have been proposed for predictive [39], historical [44] and complex spatio-temporal queries [22]. All these techniques, as well as the frameworks for spatio-temporal queries [6], [43], [24], work in a completely centralized setting. The same applies to online trajectory searching services such as GeoLife, GPS-Waypoints, Sharemyroutes and their academic counterparts [26], which assume that user trajectories are aggregated and stored on a centralized or cloud-like infrastructure. Notice that for a centralized setting, the problem definition is considerably different, than the decentralized scenario we consider in this work, as QN maintains all trajectories locally and global-knowledge statistics can be maintained in local catalogs. Additionally, in a centralized setting the query processor can utilize spatial or spatio-temporal trajectory index structures, such as the R-trees (e.g., utilized in [26] and [20]), STR-trees or TB-Trees [37], in order to speed up the retrieval answers, assuming that the trajectories have already been transferred to the query processor. On the other hand, in a decentralized setting all of these come at a significant messaging cost and require high levels of data-disclosure.

In our previous work in [47], we have already paved the way towards trajectory processing techniques in a distributed manner (i.e., without percolating each and every user geo-location to a central authority.) However, those were both agnostic in terms of energy and time constraints that arise in a smartphone network, but also in respect to the trajectory trace disclosure issues (i.e., they assumed that the query processor can arbitrarily access the distributed trajectories.) More importantly, our previous work assumed that trajectories where vertically fragmented across n distributed sites (i.e., each distributed site holds subsequences of one or more trajectories), while this work focuses on the horizontally fragmented case (i.e., each smartphone holds the complete trajectory locally.)

Top-K queries have been studied in a variety of contexts including middleware systems [18], web accessible databases [10] and stream processors [5]. An excellent survey for relational database environments appears in [23]. It has been shown in several studies [10], that top-K query processing is meaningful only if the predicate parameter K refers to a small subset of the complete answer set (e.g., up to 1%). For larger values of K, it is more beneficial if the query optimizer retrieves the complete answer set.

8 Conclusion

This work presented a crowdsourced framework for executing distributed similarity search queries on trajectories that are stored in-situ on smartphones. SmartTrace$^+$ enables the execution of such queries in both outdoor environments (using GPS coordinates) and indoor environments (using WiFi Received-Signal-Strength measurements), without disclosing the traces of participating users to the querying node. We have evaluated our algorithms on both synthetic and real workloads. Our algorithms reach the same results as the fully-centralized and the fully-decentralized approaches, while consuming considerably less energy and returning the results faster. Our experimental results also confirm our analytical study. In the future we plan to conduct a large-scale field study of a crowdsourcing service for transit planning using our prototype system that will be released as an open-source project. We finally also intend to embed additional trajectory similarity metrics to our framework and to assess their quality.

Acknowledgments: We would like to thank the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by the first author's Startup Grant, funded by the University of Cyprus, EU's FP7 CONET project, EU's FP6 Marie Curie TOK "SEARCHiN" project and EU's FP7 "SemSorGrid4Env" and "MODAP" projects.

References

[1] MobiPerf, `http://mobiperf.com/`, 2011.
[2] Andreou P., Zeinalipour-Yazti D., Pamboris A., Chrysanthis P.K., Samaras G., "Optimized Query Routing Trees for Wireless Sensor Networks", In *Information Systems*, Vol. 36, Iss. 2, pp. 267-291, 2011.
[3] Al-Aha K. , Snodgrass R., Soo M., "Bibliography on Spatiotemporal Databases," In SIGMOD Record, 22(1), 1993.
[4] Azizyan M., et. al. "SurroundSense: mobile phone localization via ambience fingerprinting," In *MobiCom*, 2009.
[5] Babcock B. and Olston C., "Distributed Top-K Monitoring", In *SIGMOD*, 2003.
[6] Bakalov P., Hadjieleftheriou M., Tsotras V., "Time Relaxed Spatiotemporal Trajectory Joins," In *GIS*, 2005.
[7] Berndt D., Clifford J., "Using Dynamic Time Warping to Find Patterns in Time Series," In *KDD*, 1994.
[8] Brew A., Greene D., Cunningham P., "Using Crowdsourcing and Active Learning to Track Sentiment in Online Media," In em ECAI, 2010.
[9] Brinkhoff T., "A Framework for Generating Network-Based Moving Objects," In *GeoInformatica*, 6(2), 2002.
[10] Bruno N., Gravano L., Marian A., "Evaluating Top-K Queries Over Web Accessible Databases," In *ICDE*, 2002.
[11] Campbell A., Eisenman S., Lane N., Miluzzo E., and Peterson R., "People-centric urban sensing," In *WICON*, 2006.
[12] Chow C-Y., Mokbel M.F., Aref W.G., "Casper*: Query Processing for Location Services without Compromising Privacy" In *ACM TODS* 34(4), pp. 1-48, 2009.
[13] Chun B. et. al., "PlanetLab: An Overlay Testbed for Broad-Coverage Services", In *ACM SIGCOMM CCR* 33(3), 2003.
[14] Costa C., Laoudias C., Zeinalipour-Yazti D. and Gunopulos D., "SmartTrace: Finding Similar Trajectories in Smartphone Networks without Disclosing the Traces", In *ICDE*, 2011 (demo).
[15] Das G., Gunopulos D., Mannila H., "Finding Similar Time Series," In *PKDD*, 1997.
[16] Das T., Mohan P., Padmanabhan V.N., Ramjee R., Sharma A., "PRISM: platform for remote sensing using smartphones," In *MobiSys*, 2010.

[17] Eriksson J., Girod L., Hull B., Newton R., Madden S., Balakrishnan H., “The pothole patrol: using a mobile sensor network for road surface monitoring,” *MobiSys*, 2008.
[18] Fagin R., Lotem A., Naor M., “Optimal Aggregation Algorithms For Middleware,” In *PODS*, 2001.
[19] Franklin M.J., et. al. “CrowdDB: answering queries with crowdsourcing.” In *SIGMOD*, 2011.
[20] Frentzos E., Gratsias K., Theodoridis Y., “Index-based Most Similar Trajectory Search,” In *ICDE*, 2007.
[21] Google, Geolocation API, `http://tinyurl.com/65cv2m`
[22] Hadjieleftheriou M., Kollios G., Bakalov P., Tsotras V.J., “Complex Spatio-Temporal Pattern Queries,” In *VLDB*, 2005.
[23] Ilyas I.F., Beskales G., and Soliman M.A., “A Survey of Top-k Query Processing Techniques in Relational Database Systems”, In *ACM Computing Surveys* 40(4), 2008.
[24] Jeung H., Lung Yiu M., Zhou X., Jensen C.S., Tao Shen H., “Discovery of convoys in trajectory databases,” In *PVLDB* 1(1), 2008.
[25] Kollios G., et. al. “Indexing Animated Objects Using Spatiotemporal Access Methods,” In *TKDE* 13(5), 2001.
[26] Chen Z., Shen H-T., Zhou X., Zheng Y., Xie X. “Searching trajectories by locations: an efficiency study,” In *SIGMOD*, 2010.
[27] Chen L., Ng R.-T., “On The Marriage of Lp-norms and Edit Distance,” In *VLDB*, 2004.
[28] Chen L., Ozsu T., Oria V., “Robust and Fast Similarity Search for Moving Object Trajectories,” In *SIGMOD*, 2005.
[29] Choffnes D.R., Bustamante F.E., Ge Z., “Crowdsourcing service-level network event monitoring,” In *SIGCOMM*, 2011.
[30] Krishna R., Khoa B., Nguyen X., Jiang L., “Real-time trip information service for a large taxi fleet”, In *MobiSys*, 2011.
[31] Liu T., Sadler C.M., Zhang P., Martonosi M., “Implementing Software on Resource-Constrained Mobile Sensors: Experiences with Impala and ZebraNet,” In *MobiSys*, 2004.
[32] Ni J., Ravishankar C.V. “Indexing Spatio-Temporal Trajectories with Efficient Polynomial Approximations,” In em TKDE 19(5), 2007.
[33] Marcus A., Wu E., Madden S., Miller R.C., “Crowdsourced Databases: Query Processing with People,” In *CIDR*, 2011.
[34] Money N.A., “Glory and Cheap Talk: Analyzing Strategic Behavior of Contestants in Simultaneous Crowdsourcing Contests on TopCoder.com” In *WWW*, 2011.
[35] Musolesi M., Piraccini M., Fodor K., Corradi A., Campbell A.-T., “Supporting Energy-Efficient Uploading Strategies for Continuous Sensing Applications on Mobile Phones,” In *PerCom*, 2010.
[36] PowerTutor Tool, `http://powertutor.org/`.
[37] Pfoser D., Jensen C. S., and Theodoridis, Y., Novel “Approaches to the Indexing of Moving Object Trajectories”, In *VLDB*, 2000.
[38] Rana R.K., et. al. “Ear-phone: an end-to-end participatory urban noise mapping system,” In *IPSN*, 2011.
[39] Tao Y., Sun J., Papadias D., “Analysis of Predictive Spatio-Temporal Queries,” In *ACM TODS* 28(4), 2003.
[40] Thiagarajan A., et. al., “VTrack: Accurate, Energy-aware Road Traffic Delay Estimation using Mobile Phones,” In *SenSys*, 2009.
[41] “NSF Workshop on Sustainable Energy Efficient Data Management”, Arlington, VA, USA, May 1-3, 2011.
[42] U.S. Department of Transportation Federal Transit Administration, “Crowdsourcing Public Participation in Transit Planning”, 2008.
[43] Vieira M., Bakalov P., Tsotras V., “On-Line Discovery of Flock Patterns in Spatio-Temporal Data,” In *GIS*, 2009.
[44] Vlachos M., Hadjieleftheriou M., Gunopulos D., Keogh E., “Indexing multi-dimensional time-series with support for multiple distance measures,” In *SIGKDD*, 2003.
[45] Werner-Allen G., Swieskowski P., Welsh M., “Motelab: a wireless sensor network testbed.” In *IPSN*, 2005.
[46] Zaidan O., Callison-Burch C., “Crowdsourcing Translation: Professional Quality from Non-Professionals.” In *ACL*, 2011.
[47] Zeinalipour-Yazti D., Lin S., Gunopulos D., “Distributed Spatio-Temporal Similarity Search,” In *CIKM*, 2006.
[48] Zeinalipour-Yazti D., Laoudias C., Andreou M., Gunopulos D., “Disclosure-free GPS Trace Search in Smartphone Networks”, In *MDM*, 2011.
[49] Zheng Y., et. al. “Learning transportation mode from raw gps data for geographic applications on the web,” In *WWW*, 2008.
[50] Zheng Y., Zhang L., Xie X., Ma W.-Y., “Mining interesting locations and travel sequences from gps trajectories,” In *WWW*, 2009.

Demetrios Zeinalipour-Yazti (Ph.D., University of California - Riverside, 2005) is a Lecturer of Computer Science at the University of Cyprus. He has held positions as a Lecturer at the Open University of Cyprus and as a visiting researcher at the network intelligence lab of Akamai Technologies (MA, USA). His primary research interests include Data Management in Systems and Networks, in particular Distributed Query Processing, Storage and Retrieval Methods for Sensor, Smartphone and Peer-to-Peer Systems.

Christos Laoudias (M.Sc., University of Patras, 2005) is a PhD Candidate at the Department of Electrical and Computer Engineering and a graduate research assistant at KIOS Research Center, University of Cyprus. His research interests revolve around Wireless Networks, Positioning and Tracking Technologies, Mobile Communications and Location Based Services.

Costandinos Costa (B.Sc., University of Cyprus, 2011) is a Graduate Student at the Department of Computer Science at the University of Cyprus. His research interests include Crowdsourcing, Spatio-temporal query processing and Mobility Infrastructures.

Michail Vlachos (Ph.D., University of California - Riverside, 2004) is a Research Staff Member with the IBM Zurich Research Laboratory, Switzerland. Previously he has worked at IBM Research, NY and he has also visited Microsoft Research, Seattle. His research interests include data mining, time series analytics and data visualization. His current research is funded by a Marie-Curie International Reintegration Grant and an ERC Starting Grant.

Maria I. Andreou (Ph.D., University of Patras, 2004) is currently a Tutor at the Open University of Cyprus in the School of Pure and Applied Sciences. Before that she was a visiting Lecturer at the University of Cyprus and an Assistant Lecturer at the University of Nicosia. Her primary research interests include Graph Theoretic Problems, Sensor Networks/VANETs and Algorithmic Engineering.

Dimitrios Gunopulos (Ph.D., Princeton University, 1995) is an Associate Professor in the Department of Informatics and Telecommunications, University of Athens, Greece. He has held positions as a Postoctoral Fellow at MPII, Germany, Research Associate at IBM Almaden, USA, Assistant, Associate, and Professor of Computer Science and Engr. at UC-Riverside, USA. His research is in the areas of Data Mining, Knowledge Discovery in Databases, Databases, Sensor Networks and Peer-to-Peer systems.