Continuous All k-Nearest-Neighbor Querying in Smartphone Networks

Georgios Chatzimilioudis[*], Demetrios Zeinalipour-Yazti[*], Wang-Chien Lee[†] and Marios D. Dikaiakos[*]
gchatzim@gmail.com dzeina@cs.ucy.ac.cy wlee@cse.psu.edu mdd@cs.ucy.ac.cy
[*]Dept. of Computer Science, University of Cyprus, P.O. Box 20537, 1678 Nicosia, Cyprus
[†]Dept. of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802, USA

***Abstract*—A highly desirable function for mobile devices is to continuously provide each user its k geographically nearest neighbors at all times, a query we coin *Continuous All k-Nearest Neighbor (*CAkNN*)*. Such queries can enhance public emergency services, allowing users to send out SOS beacons to fellow users that are located most closely to them, allowing gamers or social network users to engage in location-based interactions and many other exciting applications. In this paper, we study the problem of efficiently processing a *CAkNN* query in a cellular or WiFi network. We introduce an algorithm, coined *Proximity*, which answers *CAkNN* queries in $O(n(k+\lambda))$ time, where n denotes the number of users and λ a network-specific parameter ($\lambda << n$). *Proximity* does not require any additional infrastructure or specialized hardware and its efficiency is mainly achieved due to a smart *search space sharing* technique we introduce. Its implementation is based on a novel data structure, coined k^+-heap, which achieves constant $O(1)$ look-up time and logarithmic $O(log(k*\lambda))$ insertion/update time. *Proximity*, being parameter-free, performs efficiently in the face of high mobility and skewed distribution of users (e.g., the service works equally well in downtown, suburban, or rural areas). We have evaluated *Proximity* using mobility traces from two sources and concluded that our approach performs at least one order of magnitude faster than adapted existing work.**

I. Introduction

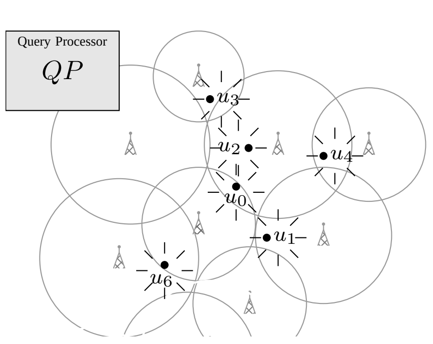
Smartphones are nowadays equipped with a number of sensors, such as WiFi, GPS, accelerometers, etc. This capability allows smartphone users to easily identify their location in indoor and outdoor spaces. The extensive sensing capabilities of these devices have brought a revolution in location-based mobile applications and services.

In this paper, we extend this sensing capability into a whole new dimension, by allowing smartphones to identify their geographically closest neighboring nodes at all times. We devise a technique that answers this query, coined *Continuous All k Nearest Neighbor (CAkNN)*, efficiently. Our technique builds upon well-studied queries like *All k Nearest Neighbors (AkNN)*, which compute the kNN for each user in the system and *Continuous k Nearest Neighbors (CkNN)*, which monitor the kNN of a user over time.

Applications of the neighborhood "sensing" capability, would allow somebody that is in a life-threatening situation to send out SOS beacons to its geographically closest neighbors. Such a futuristic application could enhance public emergency services like *E9-1-1*[1]and *NG9-1-1*[2]. Additionally, such a capability would allow somebody to engage in a location-based micro-blogging service that enables users to "follow" or "post-to" their neighborhood while being on the go. This would in effect facilitate the uptake of location-based social networks.

[1]Federal Communications Commission - Enhanced 911, March 2012, http://www.fcc.gov/pshs/services/911-services/enhanced911/

[2]Department of Transportation: Intelligent Transportation Systems New Generation 911, March 2012, http://www.its.dot.gov/NG911/



(a)

(b)

Fig. 1. (a) A snapshot of a cellular network instance, where the 2 nearest neighbors for u_0 are $\{u_1, u_2\}$. Similarly for the other users: $u_1 \rightarrow \{u_0, u_2\}$, $u_2 \rightarrow \{u_3, u_0\}$, $u_3 \rightarrow \{u_2, u_0\}$, $u_4 \rightarrow \{u_2, u_3\}$, $u_6 \rightarrow \{u_0, u_1\}$. (b) An example application of Proximity micro-blogging chat.

Consider a set of smartphone users moving in the plane of a geographic region. Let such an area be covered by a set of *Network Connectivity Points* (*NCP*) (e.g., cellular towers of cellular networks, WiFi access points of wireless 802.11 networks etc.) Each *NCP* inherently creates the notion of a *cell*. Without loss of generality, let the cell be represented by a circular area[3] with an arbitrary radius. A mobile user u is serviced at any given time point by one *NCP*, but is also aware of the other *NCP*s in the vicinity whose communication range reach u (e.g., cell-ids of different providers in an area, or MAC addresses of WiFi hot-spots in an area.)

To illustrate our abstraction, consider the example network shown in Figure 1, where we want to provide a micro-blogging chat channel between each user u and its $k = 2$ nearest neighbors. In the given scenario, each user concurrently requires a different answer-set to a globally executed query, as shown in the caption of Figure 1. Notice that the answer-set for each user u is not limited within its own *NCP* and that each *NCP* has its own communication range. Additionally, there might be areas with dense user population and others with sparse user

[3]Using other geometric shapes (e.g., hexagons, Voronoi polygons, grid-rectangles, etc.) for space partitioning would not affect the efficiency of the ideas presented in this work.

population. Consequently, finding the k nearest neighbors of some arbitrary user u could naively involve from a simple lookup in the *NCP* of u to a complex iterative deepening into neighboring *NCP*s, as we will show in Figure 2(b).

Existing work includes retrieval of the k-nearest neighbors for mobile users (*kNN* and all-*kNN*) [1], [2], [3], [4], [5] and continuous retrieval of k nearest neighbors of a single mobile user (continuous-*kNN*) [6], [7], [8], [9], [10], [11], [12]. As we will show through our extensive related work, the problem of CAkNN for spatio-temporal applications has not been covered in previous work. To address the *CAkNN* problem, one can build upon existing work that continuously provide the *kNN* to a single user and adapt them to provide the *kNN* to all users in the system. Yet, such an approach requires an instance of those algorithms to run n times, one time for each user, which is inefficient. Examples of such previous work are those by Yu *et al.* (*YPK*) [11] and Mouratidis *et al.* (*CPM*) [12], which will be described in the next section.

In this paper, we propose a parameter-free algorithm, called *Proximity*, to answer *all* k nearest neighbor queries continuously. It covers the complete search space in a batch process by iterating over all user locations just once, making only a minimal number of comparisons between them. *Proximity* exploits a novel data structure, coined k^+-heap, for dividing the search space per *NCP* and enabling search space sharing among the mobile users within each *NCP*. In this way, we avoid computing a new search space for every user in the system.

Our *Proximity* framework is robust to high mobility patterns, as it is stateless and has a fast construction time. Furthermore, *Proximity* is robust to skewed distributions of users, as its space division technique depends solely on the distribution and communication range of the *NCP*s.

In summary, this work makes the following contributions:

- We identify and define *CAkNN* queries, through a detailed categorization of existing work on a variety of *NN* problems.
- We propose a novel data structure, coined k^+-heap, for constructing the search space and facilitating search space sharing. Our structure has a $O(1)$ lookup time for each entry in the answer-set, $O(log(k * \lambda))$ insertion time and $O(k + \lambda)$ scan time, where λ is the user capacity of an *NCP*.
- We propose a novel algorithm, termed *Proximity*, for solving the *CAkNN* problem efficiently. It minimizes the computational cost for continuously deriving the *kNN* to each moving object to $O(n(k + \lambda))$ time.
- We show analytically and with an extensive experimental evaluation the superiority of our specialized *CAkNN* solution over the adaptation of state-of-the-art solutions given to similar problems.

In the following Section II, we present the related work; Section III defines our system model and the problem. Section IV presents the *Proximity* framework and a breakdown of our data structures and algorithms. Section V presents an analytical study of the time complexity of our framework. Section VI presents the experimental evaluation and comparison against adapted previous work. Finally, Section VII concludes the paper.

II. Background on *kNN*

We provide a summary of existing state-of-the-art work that deals with neighborhood queries and categorize them according to the characteristics of the queries they answer. Existing work can be classified in spatial data applications and spatio-temporal data applications.

A. Spatial Data

For applications where data is represented by a linear array, constant time algorithms have been proposed to solve the *All Nearest Neighbor* (*ANN*) and *All k-Nearest Neighbor* (*AkNN*) problems. There has been extensive work in the field of image processing and computational geometry (e.g., [13], [14]).

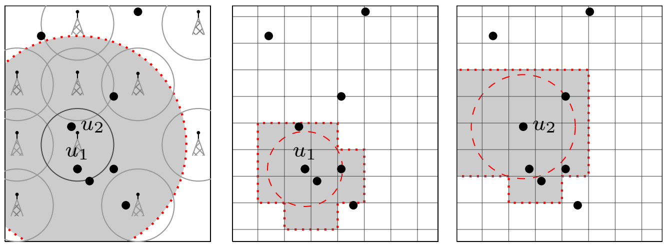
In Euclidean space (and general metric spaces), there has been also extensive work on solving the *ANN* and *AkNN* problems. For large datasets residing on disk (external memory), works like Zhang *et al.* [5], Chen *et al.* [15], and Sankaranarayanan *et al.* [16] exploit possible indices on the datasets and propose algorithms for R-tree based nearest neighbor search.

For small *ANN* and *AkNN* problems in Euclidean space, where data fits inside main memory, early work in the domain of computational geometry has proposed solutions. Clarkson *et al.* [1] was the first to solve the *ANN* problem followed by Gabow *et al.* [2], Vaidya [3] and Callahan [4]. Given a set of points, [1], [2], [4] use a special quad-tree and [3] use a hierarchy of boxes to divide the data and compute the *ANN*. The worst case running time, for both building the needed data structures and searching in these techniques, is *O(nlogn)*, where n is the number of points in the system. For the *AkNN* problem works [1], [4] propose an algorithm with *O(kn+nlogn)* and [3] an algorithm with *O(knlogn)* time complexity. Callahan's [4] main contribution is a parallel algorithm that solves the *ANN* problem in $O(logn)$ using $O(n)$ processors.

B. Spatio-Temporal Data

In spatio-temporal data applications the datasets consist of objects and queries that move over time in some Euclidean space. Existing work in this category only tackles the problem of answering a k-nearest neighbor query for a single user over time (*CkNN* query).

For large disk-resident datasets Tao *et al.* [6], Benetis *et al.* [7], Iwerks *et al.* [17], Raptopoulou *et al.* [8], and Frentzos *et al.* [18] assume that the velocity of the moving objects is fixed and the future position of an object can be estimated. Huan *et al.* [19] assume that there is only some uncertainty in the velocity and direction of the moving objects and they propose algorithms to optimize the case were the future position estimation can also be uncertain. This set of works uses time parameterized R-trees to efficiently search for the nearest neighbors. Kollios *et al.* [9] propose a method able to answer *NN* queries for moving objects in 1D space.



(a) (b)

Fig. 2. (a) In *Proximity* the search space is preconstructed for all users of the same cell (e.g., u_1 and u_2); whereas (b) for existing state-of-the-art algorithms the search space needs to be iteratively discovered by expanding a ring search for each user separately into neighboring cells.

Their method is based on the dual transformation where a line segment in the native space corresponds to a point in the transformed space, and vice-versa. Xiong *et al.* [10] focus on multiple *kNN* queries and propose an incremental search technique based on hashing objects into a regular grid, keeping CPU time in mind. The main objective of these works on disk-resident data is to minimize disk I/O operations, considering CPU time only as a secondary objective in the best case.

Main memory processing is usually mandatory for spatio-temporal applications, where objects are highly mobile. The intensity of the location updates is very restrictive for disk-based storage and indexing, and demands optimization in respect to the CPU time. Yiu et al. [20] in their work find the users in the proximity satisfying a given distance threshold. Their objective is to optimize communication cost between the server and the users.

Yu *et al.* [11] (*YPK*) followed by Mouratidis *et al.* [12] (*CPM*) optimize *kNN* queries in a similar fashion as Xiong *et al* do for disk-resident data. Data objects are indexed by a grid in main memory (see Figure 2) given a system-defined parameter value for the grid size. For each query they both use a form of iteratively enlarging a range search to find the *kNN*. For small object speeds and/or low object agility, both *YPK* and *CPM* propose a *stateful* technique to incrementally compute the result of a query of the current timestep using the result of the previous timestep. They define an influence region for the query inside the grid and depending on what happens in this region, the new result is computed using the previous result, minimizing the search space. Whenever the query object moves or the agility and speed of the objects is high, both *YPK* and *CPM* fall back to their slower *stateless* version where at each timestep the result of the query is computed from scratch. Similar work has been done by Hu *et al.* [21] who propose a similar solution to [11], [12], but also try to minimize the communication overhead by sending minimum location updates.

C. Shortcomings Of Existing Work

Techniques that optimize disk I/O are unattractive for solving *CAkNN* queries, since the CPU latency is the actual bottleneck as shown by Chen *et al.* [15]. Moreover, tree-based techniques proposed for *ANN* queries require super-linear time for their structure build-up phase (as [1], [2], [3], [4]) and need to be updated or re-built in every timestep, which is inefficient.

TABLE I
NOTATION USED THROUGHOUT THIS WORK.

Notation	**Description**
NCP	network connectivity point
c, C	single *NCP*, set of all *NCP*s
$radius_c$	range of *NCP* c
λ	the maximum number of users an *NCP* can serve
u, U	a single user, set of all users in the network
n	number of users in the network ($\|U\|$)
U_c	set of users of *NCP* c
r, R	a single user report, all user reports for a single timestep
$loc(u)$	location of user u
$ncp(u)$	the *NCP* that a user is registered to
$ncp_{vic}(u)$	list of *NCP*s whose range cover user u
S_c	the search space of *NCP* c
d_c	distance of k^{th} nearest user to the border of *NCP* c
kNN(u)	the set of k nearest neighbors of user u
kth_c	the k^{th} nearest outside user to the boundary of cell c

No previous work tackles the problem of continuous all k-nearest neighbor (*CAkNN*) queries specifically. In smartphone network applications the users are highly mobile with hard-to-predict mobility patterns and their location distribution is far from uniform [22]. This makes *stateful* techniques inefficient as shown in [11], [12], since keeping previous answers (states) of the query becomes more of a burden than a help for faster query evaluation. Furthermore, in proximity applications considered in this paper, smartphone users are moving and are both the objects of interest and the focal points of queries.

Our framework, *Proximity*, is main-memory based and *stateless*, i.e., no previous data/calculation of the previous evaluation round is used in the current round. A *stateless* *CAkNN* solution would solve an *AkNN* problem at each timestep. We also compare *Proximity* analytically to the early work of computational geometry [1], [2], [3], [4] and show that the running time complexity of our framework is better (i.e., $O(n(k+\lambda))$ as opposed to $O(knlogn)$). We compare *Proximity* experimentally against an adaptation of state-of-the-art *CkNN* solution [11], [12]. Due to the agility of the realistic mobile datasets used, these works can only make use of their *stateless* algorithm, which solves a *kNN* query in every timestep. Thus, such adaptations can only optimize a *kNN* query for each timestep separately and for each user separately, building a new search space for each user. We show that our specialized *Proximity* framework performs better, mainly due the batch processing capability of the *AkNN* queries. The most significant difference is that the *Proximity* framework groups users of the same cell together and uses the same search space for each group (*search space sharing*).

III. System Model and Problem Formulation

This section formalizes our system model and defines the problem. The main symbols are summarized in Table I.

Let U denote a set of smartphone users moving in the plane of a geographic region. Let such an area be covered by a set of *Network Connectivity Points* (*NCP*) (e.g., cellular towers found in cellular networks, WiFi access points found in wireless networks etc.) Each *NCP* inherently creates the notion of a *cell*, defined as c_i. Without loss of generality, let the cell be represented by a circular area with radius $radius_c$. The number of users λ serviced by an *NCP* is a network parameter (cell capacity). A mobile user u is serviced at any given time point by one *NCP*, but is also aware of the other *NCP*s that are in its vicinity and whose communication range cover it (e.g., cell-ids of different providers in an area, or MAC addresses of WiFi hot-spots in an area, etc.)

Assume that there is some centralized (or cloud-like) service, denoted as QP (Query Processor), which is accessible by all users in user set U. Allow each user u to report its positional information to QP regularly. These updates have the form $r_u = \{u, loc(u), ncp(u), ncp_{vic}(u)\}$, where $loc(u)$ is the location of user u[4], $ncp(u)$ is the *NCP* user u is registered to and $ncp_{vic}(u)$ is a list of *NCP*s in the vicinity of u.

The problem we consider in this work is how to efficiently compute the k nearest neighbors of all smartphones that are connected to the network, at all times. We consider a *timestep* that defines rounds where we need to recompute the *kNN*s of the users. Depending on the application, this can take place either at a preset time interval or whenever we have a number of new user location updates arriving at the server. Formally, we aim to solve a problem we coin the *CAkNN* problem.

*Definition 1 (*CAkNN *problem):* Given a set U of n points in space and their location reports $r_{i,t} \in R$ at *timestep* $t \in T$, then for each object $u_i \in U$ and *timestep* $t \in T$, the *CAkNN* problem is to find the k objects $U_{sol} \subseteq U - u_i$ such that for all other objects $u_o \in U - U_{sol} - u_i$, $dist(u_k, u_i) \leq dist(u_o, u_i)$ holds.

In order to better illustrate our definition, consider Figure 3, where we plot a *timestep* snapshot of 7 users $u_0 - u_6$ moving in an arbitrary geographic region. The result for this *timestep* to a $k = 2$ query would be *kNN*$(u_0) = \{u_1, u_2\}$, *kNN*$(u_1) = \{u_0, u_2\}$, *kNN*$(u_2) = \{u_3, u_0\}$, *kNN*$(u_3) = \{u_2, u_0\}$, *kNN*$(u_4) = \{u_2, u_0\}$, *kNN*$(u_6) = \{u_7, u_1\}$.

Obviously, the solution for a user u will not always reside inside the same NCP cell c, but might reside in neighboring cells or even further (e.g., if neighboring cells do not have any users). Computing a separate search space for every user is very expensive. On the other hand, *search space sharing* is achieved when the same search space is used by multiple users and it guarantees the correct *kNN* solution for all of them. If we apply this reasoning for all users U_c in c, then the common search space S_c for U_c would be defined as the union of the individual search spaces of every user in U_c. We efficiently build S_c with the assistance of complementary data structures we devise in this work and explain next. In Figure 3, the search space constructed by our framework for users u_0 and u_6 is the largest dotted circle.

IV. The *Proximity* Framework

In this section we start out with an outline of the *Proximity* framework and the intuition behind its operation. We then describe in detail how the search space is built-up using our k^+-heap data structure and its associated insertion and update algorithms.

[4]The location of a user can be determined either by fine-grain means (e.g., AGPS) or by coarse-grain means (e.g., fingerprint-based geolocation [23]).

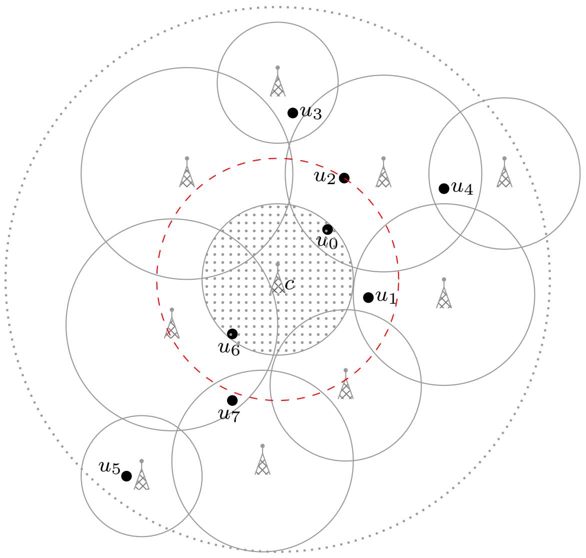


Fig. 3. The *search space* of cell c is the big circle with the dotted outline. Any user inside this circle is a *kNN* candidate for any user inside c.

A. Outline Of Operation

The *Proximity* framework is designed in such a way that it is: i) *Stateless*, in order to cope with transient user populations and high mobility patterns, which complicate the retrieval of the continuous *kNN* answer-set. In particular, we solve the *CAkNN* problem for every timestep separately without using any previous computation or data; ii) *Parameter-free*, in order to be invariant to parameters that are network-specific (such as cell size, capacity, etc.) and specific to the user-distribution; iii) *Memory-resident*, since the dynamic nature of mobile user makes disk resident processing prohibitive; iv) *Specially designed* for *highly mobile* and *skewed distribution* environments performing equally well in downtown, suburban, or rural areas; iv) *Fast and scalable*, in order to allow massive deployment; and v) *Infrastructure-ready* since it does not require any additional infrastructure or specialized hardware.

For every timestep *Proximity* works in two phases (Algorithm 1): In the first phase one k^+-heap data structure is constructed per *NCP*, using the location reports of the users (lines 1-8). In the second phase, the k nearest neighbors for each user are determined by scanning the respective k^+-heap and the results are reported back to the users (lines 9-19).

At each timestep the server QP initializes our k^+-heap for every *NCP* in the network. The k^+-heap integrates three individual sub-structures that we will explain next (see Figure 5). The user location reports are gathered and inserted into the k^+-heap of every *NCP*. After all location reports have been received and inserted, each *NCP* has its search space stored inside its associated k^+-heap. After the build phase, each user scans the k^+-heap of its *NCP* to find its k nearest neighbors.

B. Constructing The Search Space

Here we describe the intuition behind our search space sharing concept. Every user covered by an *NCP* uses the same search space to identify its *kNN* answer-set.

Algorithm 1 . Proximity Outline

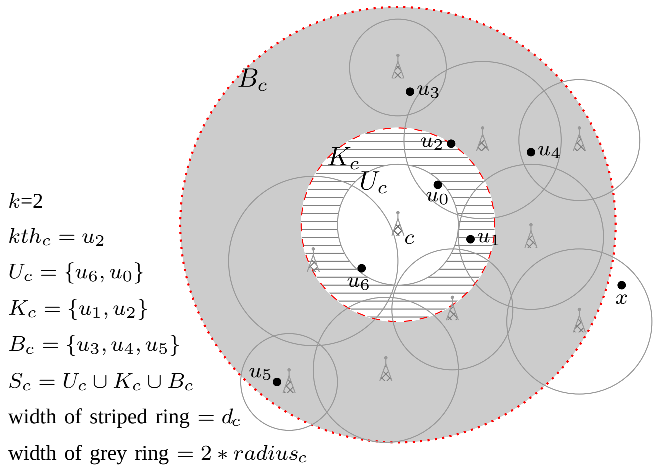
Input: User Reports R (single timestep), set C of all *NCP*s
Output: *kNN* answer-set for each user in U

```
 1: for all $c \in C$ do
 2:     initialize $k^+{}_c$                 ▷ Initialize our $k^+$-heap
 3: end for
 4: for all $r \in R$ do                     ▷ Phase 1: build $k^+$-heap
 5:     for all $c \in C$ do
 6:         $insert(r, k^+{}_c)$
 7:     end for
 8: end for
 9: $U \leftarrow users(R)$
10: for all $u \in U$ do                     ▷ Phase 2: scan $k^+$-heap
11:     $kNN_u = \emptyset$                  ▷ Conventional k-max heap
12:     $c \leftarrow r_u.ncp$
13:     for all $v \in k^+{}_c$ do
14:         if $v$ is a kNN of $u$ then
15:             $update(kNN_u, v)$
16:         end if
17:     end for
18:     report kNN to node $u$
19: end for
```

In order to construct a correct search space for each *NCP*, we need to be able to identify nodes that might be part of the *kNN* answer-set for any arbitrary user of a given *NCP*. For instance, consider two users u_0 and u_6, in Figure 3, which are positioned on the perimeter of their *NCP* c. Also, consider user u_2 being outside c and close to u_0. In such a scenario, the search space for c must obviously include u_2, as it is a better *kNN* candidate to u_0 than u_6. However, even if we were aware of the k closest users to c (besides the users in c), would not allow us to correctly determine the *kNN* for any arbitrary user in c. To understand this, consider again Figure 3 with a *2NN* query. u_1 and u_2 are the two closest outside nodes to the border of c. Yet, we can visually determine that u_7 is a more appropriate *2NN* candidate for u_6 than all aforementioned nodes, i.e., u_0, u_1, u_2.

To overcome this limitation, we define a prune-off threshold, denoted as kth_c, which determines the size of the search space of c. kth_c is the k^{th} closest outside user to the border of c, which determines the width d_c of the *search expansion* (striped ring as seen in Figure 4). Inside this ring there are k users by definition. These k users form the K-set. In our running example $kth_c = u_2$. This guarantees that the search space will have at least k users. All users at distance less that $2 * radius_c + d_c$ from c's border, are also part of the search space. This guarantees that each user inside c will find its actual kNN inside the search space.

The size of each *NCP* search space depends on the communication area of the *NCP* and the k^{th} closest outside user to the border of its communication area. The users inside c comprise set U_c and the users that are at distance greater than d_c and less than $2 * radius_c + d_c$ from the cell's border comprise set B_c (grey ring in Figure 4). Set K, set B, and the users U_c inside c form the search space S_c of c.



B_c
u_3
u_2
u_4
K_c
U_c
u_0
c
u_1
u_6
x
u_5
k=2
$kth_c = u_2$
$U_c = \{u_6, u_0\}$
$K_c = \{u_1, u_2\}$
$B_c = \{u_3, u_4, u_5\}$
$S_c = U_c \cup K_c \cup B_c$
width of striped ring $= d_c$
width of grey ring $= 2 * radius_c$

Fig. 4. An example of the common search space for the users inside cell c (white circle) for $k = 2$. The search space S_c of c is $\{u_0, u_1, u_2, u_3, u_4, u_5, u_6\}$ and is represented by the big circle with the dotted outline. Set S_c includes all users inside c (set U_c), the striped ring (set K_c) and the grey ring (set B_c). Any node outside S_c (e.g., user x) is guaranteed NOT to be a *kNN* of any user inside cell c. The 2 nearest neighbors for the nodes in c are *kNN*$(u_0) = \{u_1, u_2\}$ and *kNN*$(u_6) = \{u_0, u_1\}$.

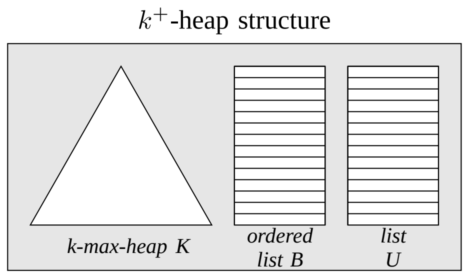


Fig. 5. A visualization of an k^+-heap. It consist of a k-max-heap K, an ordered list B and an ordinary list U.

Definition 2 (K-set): Given a set of users $u \in U - U_c$ outside *NCP* cell c that is ordered with ascending distance $dist(u, c)$ to the border of c, set K_c consists of the first k elements of this set (striped ring in Figure 4).

Definition 3 (k^{th} outside neighbor of the NCP *cell):* Given K_c (ordered as in Definition 2), the k^{th} user is called the k^{th} nearest neighbor of c and denoted kth_c.

Definition 4 (B, Boundary Set): Given an *NCP* denoted as c and its k^{th} outside neighbor kth_c, set B_c consists of all users $u \in U - (U_c \cup K)$ with distance $dist(u, c) < dist(kth_c, c) + 2 * radius_c$ from the border of c. In other words B_c consists of all users $u \in U$ with distance $dist(kth_c, c) < dist(u, c) < dist(kth_c, c) + 2 * radius_c$.

Definition 5 (S, Search Space set): Given an *NCP* c and its K_c set, the search space S_c of c consists of all users $u \in U_c \cup K_c \cup B_c$ (big circle with dotted outline in Figure 4).

In our Figure 4 example, at the end of the build phase, the k^+-heap of c includes users $\{u_6, u_0, u_1, u_2, u_3, u_4, u_5\}$. This is the common search space S_c for all users $U_c = \{u_0, u_6\}$ of c, which guarantees to include their exact k nearest neighbors.

C. Specialized Heap: The k^+-heap

Computing the search space for each cell inefficiently might be prohibitive for the application scenarios we envision as detailed in the introduction. In this section, we show in detail how the search space for an *NCP* is constructed using our k^+-heap data structure. Recall that as user reports arrive at the server QP they are inserted into each k^+-heap. A user report either stays inside a k^+-heap or eventually gets evicted using a policy that we will describe later. After all user reports have been probed through the k^+-heap of every *NCP*, each k^+-heap contains the actual search space of its *NCP*. Consequently, the build phase takes a total of $n * |C|$ insertions.

The k^+-heap consists of three separate data structures (see Figure 5): a heap for the set K_c and two lists for *Boundary* set B_c and the set U_c. The heap used for set K_c is a conventional k-max-heap. It stores only the k users outside c with the minimum distance $dist(u, c)$ from the border of c. Thus, the heap K has always kth_c at its head. The *boundary* list is a list ordered by $dist(u, c)$, which stores set B. Its elements are defined by kth_c (see Definition 4). Similarly, we use a list to store the users $U_c \subseteq U$ of c. Notice that some *NCP* cells will be overlapping, so there are areas where users are inside multiple cells. Such users are inserted into all lists U_j of $c_j \in C$ that cover them. The k^+-heap has $O(1)$ lookup time for the k^{th} nearest neighbor of c. It has worst case $O(log(k * |B|))$ insertion time and contains $|S_c| = k + |B_c| + |U_c|$ elements.

Algorithm 2 . k^+-heap: Insert(u_{new})

```
Input: $u_{new}$, $c$ of $u_{new}$
Output: $k^+{}_c$
 1: $kth_c \leftarrow head(K_c)$
 2: if $dist(u_{new}, c) < radius_c$ then
 3:     $insert(u_{new}, U_c)$
 4: else if $dist(u_{new}, c) < dist(kth_c, c)$ then
 5:     $insert(u_{new}, K_c)$
 6:     if $K$ heap has more than $k$ elements then
 7:         $kth_c \leftarrow pophead(K_c)$
 8:         $insert(kth_c, B_c)$
 9:         $Update\_boundary(head(K_c))$
10:     end if
11: else if $dist(u_{new}, c) < dist(kth_c, c) + 2 * radius_c$ then
12:     $insert(u_{new}, B_c)$
13: else
14:     discard $u_{new}$
15: end if
```

Algorithm 3 . k^+-heap: Update_boundary(kth_c)

```
Input: $kth_c$ (the $k^{th}$ outside neighbor of NCP $c$)
Output: $B_c$ updated
 1: $d \leftarrow dist(kth_c, c) + 2 * radius_c$
 2: $i \leftarrow$ find the element with the maximum distance that is
        smaller than $d$ using binary search
 3: $remove(B_c, i + 1, end)$
```

D. Insertion Into The k^+-Heap (Algorithm 2 and 3)

When inserting a new element u_{new} into the k^+-heap of c, we distinguish among four cases (see Algorithm 2): i) u_{new} is covered by c and belongs to set U_c (line 2), ii) u_{new} belongs to set K_c (line 4), iii) u_{new} belongs to set B_c (line 11), or iv) u_{new} does not belong to the search space $S_c = U_c \cup K_c \cup B_c$ of *NCP* c (line 13). In case (i) the element is inserted into the U_c list. In case (ii) we need to insert u_{new} into heap K (line 5) and move the current head kth_c from K to the boundary list B (lines 7-8). This yields a new head kth'_c in K (line 9). Every time the kth_c changes, the boundary list B needs to be updated, since it might need to evict some elements according to Definition 4. In case (iii) we insert u_{new} into the ordered boundary list B (line 12). Note that the sets K_c and B_c are formed as elements are inserted into the k^+-heap. The first k elements inserted in the empty k^+-heap define the K_c set. In case (iv) the element is discarded.

E. Running Example

Using Figure 4 as our network example in timestep t we will next present the *Proximity* framework step-by-step.

Server QP initiates a k^+-heap for every *NCP* in C. The k^+-heap consists of heap K, ordered list B, and list U. The reports that arrive at QP are $R = r_0, r_1, r_2, r_3, r_4, r_5, r_6, r_x$. Every report is inserted into every k^+-heap on the QP (see Algorithm 1, lines 1-5). The order in which the reports are inserted into a k^+-heap does not affect the correctness of the search space. For our example, assume that the reports are inserted in the order seen in the first column of Table II. For every insertion we can see the contents of $k^+{}_c$ in the same Table. For simplicity we will only follow the operation for the k^+-heap of *NCP* c.

When report r_4 is inserted into $k^+{}_c$ it ends up inside heap K_c, since user u_4 is not inside *NCP* cell c (condition line 2) and heap K_c is empty. Next, report r_x is inserted into $k^+{}_c$ and it also ends up inside heap K_c since this is not full yet. When r_2 is inserted, it ends up inside heap K_c (line 5) and it becomes the new head of the heap kth_c. The old head of the heap was r_x and is popped out of K and is inserted into the B_c list (lines 7-8). The update on the B_c list is triggered (line 9) which, in this case, does not affect the list. Similarly, when r_3 is inserted the same operations (lines 5-10) take place as with the insertion of r_2. Next, r_1 is inserted with the same effect, only this time the B_c list is altered during its update (line 9). r_2 is the new head of heap K_c and according to Definition 5 defines a new search space radius $d = dist(u_2, c) + 2 * radius_c$ (line 1 of Algorithm 3). The report r_x inside list B_c has $dist(u_x, c) > d$, thus it belongs to the tail of the list that is discarded in line 3 of Algorithm 3. When r_5 is inserted it ends up directly inside list B_c (line 12), since it is outside c, further away than kth_c but closer than $dist(kth_c, c) + 2 * radius_c$ to the border of c. Reports r_0 and r_6 both end up directly inside list U_c (line 3), since they are covered by c, satisfying the condition in line 2.

After all reports are inserted into the k^+-heaps phase 1 of Algorithm 1 is completed and the search space is ready. For the second phase of Algorithm 1 the server scans a single k^+-heap for each user. The server can scan the k^+-heap of any

TABLE II
BUILD-UP PHASE OF THE k^+-HEAP OF *NCP* c AS USER LOCATION REPORTS ARE INSERTED.

Arriving Reports	Structure K_c	Structure B_c	Structure U_c	Algorithm 2 lines
r_4	$\{r_4\}$	$\{\}$	$\{\}$	1,4,5
r_x	$\{r_x, r_4\}$	$\{\}$	$\{\}$	1,4,5
r_2	$\{r_4, r_2\}$	$\{r_x\}$	$\{\}$	1,4-11
r_3	$\{r_3, r_2\}$	$\{r_4, r_x\}$	$\{\}$	1,4-11
r_1	$\{r_2, r_1\}$	$\{r_3, r_4\}$	$\{\}$	1,4-11
r_5	$\{r_2, r_1\}$	$\{r_3, r_4, r_5\}$	$\{\}$	1,12,13
r_6	$\{r_2, r_1\}$	$\{r_3, r_4, r_5\}$	$\{r_6\}$	1-3
r_0	$\{r_2, r_1\}$	$\{r_3, r_4, r_5\}$	$\{r_6, r_0\}$	1-3

NCP that covers a user u to get the k neighbors of u. In our Algorithm 1 the server scans the *NCP* that actually services the user $ncp(u)$ (lines 12). For users u_0 and u_6, the server Q scans ${k^+}_c = u_2, u_1, u_3, u_4, u_5, u_6$ and finds nearest neighbors $\{u_2, u_1\}$ and $\{u_0, u_1\}$ for user u_0 and u_6 respectively.

V. PERFORMANCE ANALYSIS

In this section we analytically derive the performance of the *Proximity* framework in respect to computational complexity and scalability. We adopt worst-case analysis regarding user distribution and/or user movement pattern as it provides a bound for all input. Our experimental evaluation in Section VI shows that our framework performs much more efficiently than the projected worst-case.

All computations in our framework happen on the server. *NCP*s do not participate in any processing; they just relay reports from the server to the mobile users and vice versa. We execute the query centrally on the server and assume that all the data can fit in main memory.

It is safe to say that in network setups, like the one we described in Section III, the tendency is to maximize the users serviced n and minimize the number of network connectivity points $|C|$, as is the case for cellular network companies [22]. Therefore, we can assume that $|C| << n$. Furthermore, each *NCP* has a predefined communication capacity expressed in bits/sec [22]. Depending on the user traffic there is always a limit λ of the amount of users each *NCP* can serve [24]. λ is independent of n, since the capacity of the *NCP* and the user traffic profiles are independent of n. For simplicity we regard λ as a network parameter that is constant.

Lemma 1: The build phase of *Proximity* has time complexity $O(nlog(k * \lambda))$.

Proof: The build phase consists of $O(n * |C|)$ insertions into k^+-heaps. Insertion and deletion in heap K of a k^+-heap costs $O(logk)$, since it is a conventional heap with constant size k. Insertion into the ordered list of size $|B|$ has worst case cost $O(log|B|)$ using binary search. Similarly, inserting into and updating the *boundary* list costs $O(log|B|)$. Thus, the worst case insertion cost for our novel k^+-heap is $O(logk + log|B|) = O(log(k * |B|))$ and there are n insertions. Each *NCP* has a user limit λ and the boundary region B contains a finite number a of *NCP*s, thus $|B| = a * \lambda$. a is a finite number independent of n and $\alpha << |C| << n$. This makes $O(log(k * |B|)) = O(log(k * \lambda))$ ■

Lemma 2: After all k^+-heaps are built, the scanning phase has time complexity $O(n(k + \lambda))$.

Proof: The size of a k^+-heap is $|S_c|$ and each user scans a k^+-heap (Theorem 1). Consequently we have $n * |S_c|$ comparisons. $|S_c| = k + |B_c| + |U_c|$ as defined by Definition 5. The size of U_c is bounded by the maximum number of users the *NCP* can serve $|U_c|_{max} = \lambda$. $|B| = a * \lambda$ as described in proof of Theorem 1. Thus, $|S_c| = k + (a + 1) * \lambda$, which means that the time complexity of a single round of *Proximity* is $O(n(k + \lambda))$. ■

Theorem 1: Each round of *Proximity* runs in $O(n(k + \lambda))$ time.

Proof: Based on Theorem 1 and 2. ■

Using the *NCP*s for space partitioning, instead of a regular grid defined on the server, gives us the advantage of exploiting the user distribution adaptation that is inherent in the deployment of wireless or WiFi *NCP*s. It further frees us from setting a global parameter that would determine the size of the grid cell or a technique to adapt the grid size according to the user distribution, which would make our framework more complicated and possibly more time consuming.

VI. EXPERIMENTAL EVALUATION

In this section we present an evaluation of the *Proximity* framework using two mobility datasets. We describe the existing work used to compare our method, the datasets used and the setup of our experiments. We run all the experiments on an Intel Core2 Duo 2.5Ghz processor with 3GB RAM running Ubuntu Linux. Note that all figures are plotted with the time (y-axis) in log-scale, thus the differences in efficiency between the algorithms are larger than their visual difference.

A. Datasets

Both datasets used in our experiments are realistic synthetic datasets that have been used in the research area of location-based services.

1) Oldenburg dataset: The first dataset is derived from the spatiotemporal generator of [25]. The input of the generator is the road map of the city of Oldenburg, Germany in an area of 25km x 25km. The output is a set of vehicle trajectories moving on this network, where each object is represented by its location at successive timestamps. A vehicle appears on a network node, completes a path to a random destination and then disappears. It's movement is determined by the road map (speed limits, the maximum capacity of the roads) and the inter-dependencies with other vehicles (traffic) [25]. For the Oldenburg dataset we used a maximum number of 1000-5000 vehicles. According to the city's official statistics and its population of around 160K, this amount of vehicles is realistic without causing congestion and slow vehicle movement [26].

2) Manhattan dataset: The second dataset derived with the VanetMobiSim [27] vehicular mobility generator as in [28]. By employing traffic generation models, VanetMobiSim outputs detailed mobility traces over real-world city maps that can be obtained through the U.S. Census Bureau [29]. In particular, we generated vehicular traffic scenarios on a 3km x 3km area in upper-east Manhattan, New York city. All vehicles were set to follow the Intelligent Driver Model with Lane Changes (IDM-LC). Vehicle density within these areas was set to 46.56 vehicles per km^2 based on 2006 transportation statistics for U.S. cities [30] with the top speed of each vehicle bounded by the road speed limit. Given the above parameters, VanetMobiSim generated vehicular traffic for a 3 hour period, constantly maintaining 500 vehicles in the roads of the Manhattan area.

B. Setup

In our setup we place a set of network connectivity points (*NCP*s) uniformly in space using a hatched grid arrangement. According to the theoretical urban base station ranges restricted by urban landscape, we use ranges 1km, 4km and 16km for the *Oldenburg* dataset and ranges 1km and 4km for the *Manhattan* dataset. Any larger radius would lead to a single base station covering the whole space making *Proximity* run like the naive brute-force technique with time complexity $O(n^2)$. Notice that we use the small ranges correspond to restricted base station communication range due to the urban landscape.

The query to be answered, by the algorithms compared, is *"Report to each user its k nearest neighbor peers for every timestep"*. The values used for k are 1, 4, 16, 64, and 256.

C. Adapting Existing Work

Existing work has proposed solutions for the *AkNN* and *CkNN* problems. Adapting an *AkNN* solution, like [1], [2], [3], [4], would solve an *AkNN* query in every timestep. In Section II, we show that by using this adaptation with the existing solutions requires super-linear time for the build-up phase at each timestep. As shown in Section V, our solution has a better time complexity. No experimental evaluation is therefore needed to verify this statement.

Another adaptation using existing work is to use state-of-the-art *CkNN* solutions for mobile data that minimize CPU time, like Yu *et al.* [11] and Mouratidis *et al.* [12]. Extending these *CkNN* solutions to answer *CAkNN* queries involves running an instance of them for each user in the system. These methods use a form of iteratively enlarging a range search to find the *kNN* for the user (see Figure 2(b)). The search space starts from the cell of the user and iteratively visits neighboring cells until at least k neighbors are found and it is guaranteed that no further neighboring cell can have user that is closer. For small object speeds and/or low object agility, both *YPK* and *CPM* propose a *stateful* technique to incrementally compute the result of a query of the current timestep, using the search space computation of the previous timestep. They define an influence region for the query inside the grid and depending on what happens in this region, the new result is computed using the previous result, minimizing the search space.

For our experiments we use the adaptation of [11] and [12] denoted as *YPK* and *CPM* respectively, as a baseline for our comparison. We implement both the *YPK* and *CPM* algorithm using the optimal value for their cell size for each timestep separately as defined in [11].

Real world scenarios have mobile phone users that move in high speeds (e.g., mobile phones on board of vehicles). Both our datasets use highly mobile object points that force the *YPK* and *CPM* algorithms to use their "overhaul"/"from scratch" computation of *kNN* in all of our experiments. This makes those algorithm slower since previous search space computations can not be reused and the search space for each user needs to be computed from scratch at every timestep.

D. Number of Nearest Neighbors (k)

In the first experiment we vary the number of nearest neighbors and find how the algorithm's performance scales. For the first three plots in Figure 6, we use the Oldenburg dataset and the different phases of the algorithms are plotted separately. The plots show the performance for constructing the necessary data structures, finding the k nearest neighbor peers for all users and the total time, respectively.

From the top plots in Figure 6, we can conclude that the build time is the bottleneck for our *Proximity* algorithm, whereas for the adapted *YPK* and *CPM* algorithms the search time is the bottleneck. We attribute this to the operating system overhead to construct our custom data structure. The build time for our competitors, *YPK* and *CPM*, is constant as shown by the figures and proved in theory in their respective works. Their space division is independent of the number of nearest neighbors needed.

On the other hand, the real benefit of *Proximity* is with respect to the search time. Figure 6(c) shows that our *Proximity* algorithm outperforms its competitors for k values larger than 4. We were unable to obtain the search time of *YPK* and *CPM* for $k = 256$ due to the long running time and high memory requirements those algorithms impose. *YPK* and *CPM* have almost identical performance, since both are based on the same intuition. These algorithms are not scalable in respect to k and are only efficient when searching for a few nearest neighbors, $k < 4$. This is backed by analytical results, since the search space for *YPK* and *CPM* solely depends on and is proportional to k.

In particular, we found that around 100 seconds is the average time for our *Proximity* algorithms to solve the *CAkNN* problem. Thus, using a conventional workstation we can report the k nearest neighbor peers to each user approximately every 2 minutes. With a high performance server machine, this can be done in seconds. The efficiency of *Proximity* is not affected by k, since the size of the search space for *Proximity* is barely affected by k (see Section V).

An even more interesting observation that favors our *Proximity* algorithm is that its build time even drops as k increases. This happens because the machine's memory scheduler makes more efficient use of buffers when the search spaces are larger. Thus, *NCP* communication range makes the build process shorter. Larger cells means less *NCP*s with larger capacity to service users. In exchange we pay a small price in the search time.

Figure 6(d) shows the combined effect, specifically, the total time for answering a *CAkNN* query. It verifies the above findings. Note that all figures are plotted with the time (y-axis) in log-scale, thus the differences in efficiency between

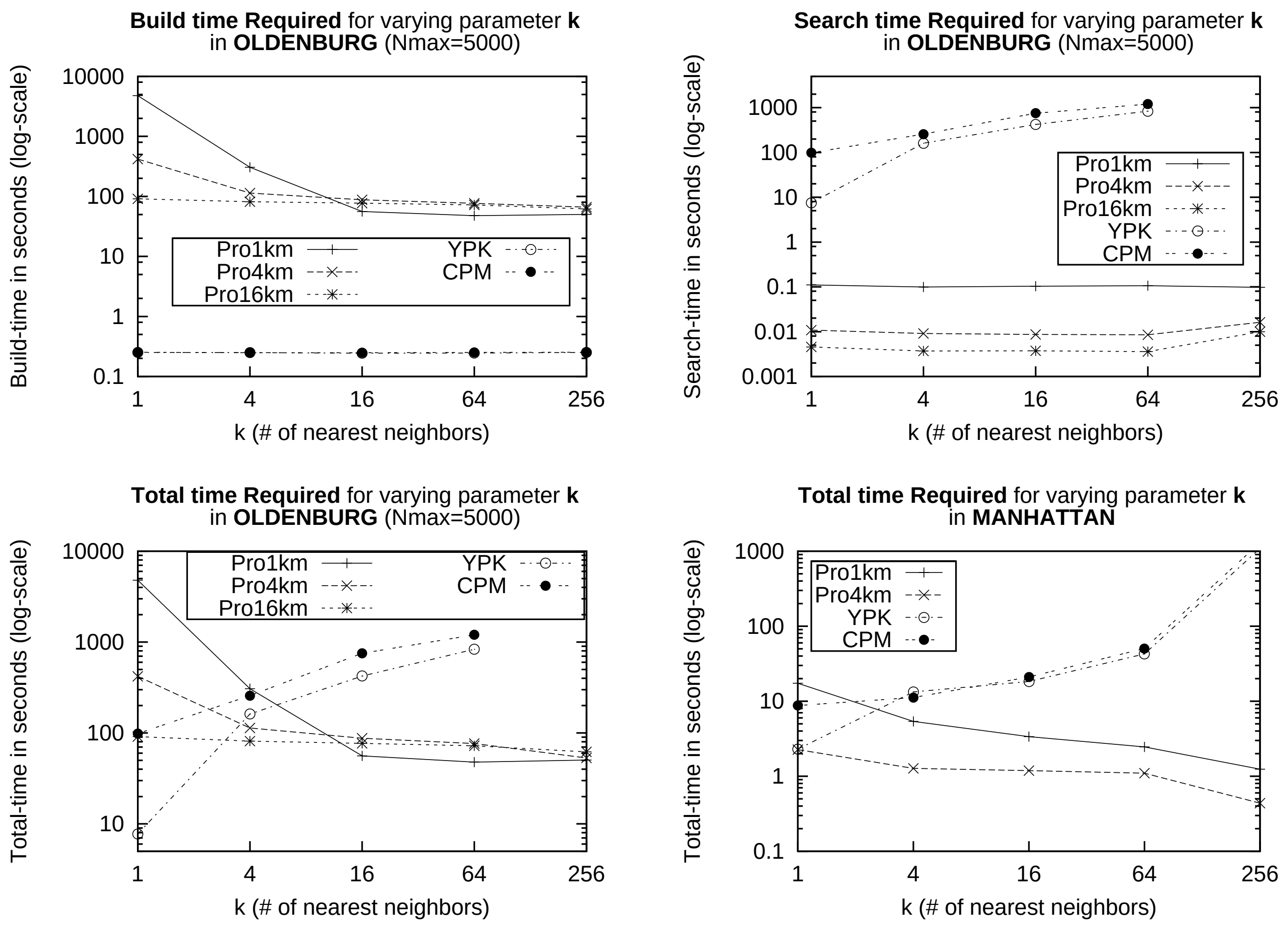


Fig. 6. CPU time (averaged over all timesteps) for (a) building the data structures (top-left), (b) searching for the *kNN* of each user (top-right), and (c) the total time for answering a *CAkNN* query for the Oldenburg dataset with $N_{max} = 5000$, varying the number k of nearest neighbors (bottom-left). In the bottom-right plot (d) we show the total CPU time for the Manhattan dataset of n=500 users, varying the number of nearest neighbors k.

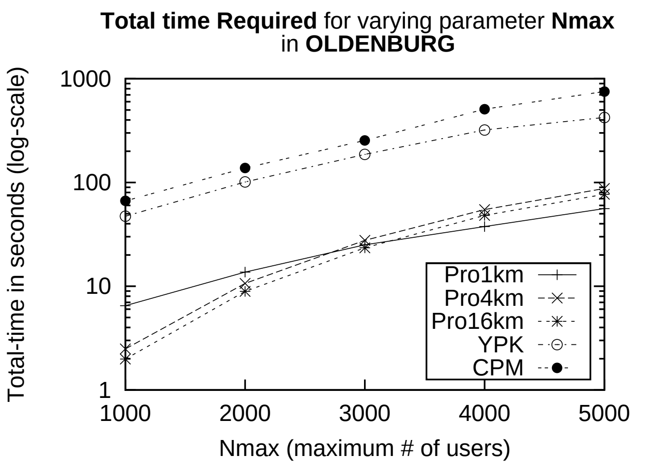


Fig. 7. Total CPU time (averaged over all timesteps) for answering a *CAkNN* query for the Oldenburg dataset with k=16, varying the number of users in the network N_{max}.

the algorithms are larger than their visual difference. *Proximity* performs almost 100 times faster than the adapted previous work.

E. Maximum Number of Users ($Nmax$)

Figure 7 shows the performance for varying the maximum number of users in the Oldenburg dataset. $Nmax$ is a parameter of the dataset generator that makes traffic build up to this maximum number and holds it close to this bound. For this experiment the number of nearest neighbors to find is set to $k = 16$. The CPU time needed for answering the *CAkNN* query shows that our *Proximity* algorithm scales with the number of users in space in realistic traffic scenarios and outperforms its competitors by an order of magnitude. It can also be seen that the algorithm do not converge for higher values of $Nmax$.

VII. Summary and Future Work

We have presented *Proximity*, an algorithm for continuously answering all k nearest neighbor queries in a cellular network. It is based on a division of the search space based on the network connectivity points and exploiting search space sharing among users of the same connectivity point. The *Proximity* framework has a better time complexity compared to solutions based on existing work. Our experiments verify the theoretical efficiency and shows that *Proximity* is very well suited for large scale scenarios.

Proximity is also easily adaptable to provide some location privacy in terms of *spatial cloaking*. It is naturally extendible to the scenario where users report to the server using *spatial cloaking* [31]. The location of a user is represented by an area, instead of a simple point. We will further investigate privacy extensions for our algorithms aiming towards strong privacy in future work. Existing work towards this direction has been published by Papadopoulos et al [32], where they answer a single snapshot of a *kNN* query guaranteeing strong privacy.

Extensions and future plans for this work are also placed in parallelizing the *Proximity* algorithm, specializing it for cloud environments and adapt our search space sharing for user-defined k values, instead of a system-defined global k value.

The novel problem of *continuous all k-nearest neighbor queries (CAkNN)* opens the path for many new and exciting spatio-temporal applications. Further the smartphone crowd is constantly moving and sensing, providing large amounts of opportunistic data that enables new crowdsourcing services and applications. Beside neighborhood-based applications, the efficiency of the Proximity framework can be used for novel social network analysis metrics. For example, we can identify users that find themselves in the proximity of the most users or the most diverse crowds or social cliques throughout a day. Existing metrics in social networks, such as centrality, can be extended and new ones defined based on the geographical neighborhood characteristics of a user revealing a new level of insight into more intricate roles of users in a network.

Acknowledgements: We are grateful to Dr. Georgios Pallis and Dr. Manolis Spanakis for the help with the Manhattan dataset. This work was supported by the second author's Startup Grant funded by University of Cyprus, EU's FP6 Marie Curie TOK "SEARCHiN", EU's FP7 CONET and EU's FP7 "MODAP" projects.

References

[1] K. L. Clarkson, "Fast algorithms for the all nearest neighbors problem," *Foundations of Computer Science, Annual IEEE Symposium on*, vol. 0, pp. 226–232, 1983.

[2] H. N. Gabow, J. L. Bentley, and R. E. Tarjan, "Scaling and related techniques for geometry problems," in *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, ser. STOC '84. New York, NY, USA: ACM, 1984, pp. 135–143.

[3] P. M. Vaidya, "An o(n log n) algorithm for the all-nearest-neighbors problem," *Discrete Comput. Geom.*, vol. 4, pp. 101–115, January 1989.

[4] P. B. Callahan, "Optimal parallel all-nearest-neighbors using the well-separated pair decomposition," in *Proceedings of the 1993 IEEE 34th Annual Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1993, pp. 332–340.

[5] J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao, "All-nearest-neighbors queries in spatial databases," *Scientific and Statistical Database Management, International Conference on*, vol. 0, p. 297, 2004.

[6] Y. Tao, D. Papadias, and Q. Shen, "Continuous nearest neighbor search," in *Proceedings of the 28th international conference on Very Large Data Bases*, ser. VLDB '02. VLDB Endowment, 2002, pp. 287–298.

[7] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis, "Nearest neighbor and reverse nearest neighbor queries for moving objects," in *Proceedings of the 2002 International Symposium on Database Engineering & Applications*, ser. IDEAS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 44–53.

[8] K. Raptopoulou, A. N. Papadopoulos, and Y. Manolopoulos, "Fast nearest-neighbor query processing in moving-object databases," *Geoinformatica*, vol. 7, pp. 113–137, June 2003.

[9] G. Kollios, D. Gunopulos, and V. J. Tsotras, "Nearest neighbor queries in a mobile environment," in *Proceedings of the International Workshop on Spatio-Temporal Database Management*, ser. STDBM '99. London, UK: Springer-Verlag, 1999, pp. 119–134.

[10] X. Xiong, M. F. Mokbel, and W. G. Aref, "Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases," in *Proceedings of the 21st International Conference on Data Engineering*, ser. ICDE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 643–654.

[11] X. Yu, K. Q. Pu, and N. Koudas, "Monitoring k-nearest neighbor queries over moving objects," in *Proceedings of the 21st International Conference on Data Engineering*, ser. ICDE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 631–642.

[12] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou, "Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 634–645.

[13] Y.-R. Wang, S.-J. Horng, and C.-H. Wu, "Efficient algorithms for the all nearest neighbor and closest pair problems on the linear array with a reconfigurable pipelined bus system," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, pp. 193–206, March 2005.

[14] T. H. Lai and M.-J. Sheng, "Constructing euclidean minimum spanning trees and all nearest neighbors on reconfigurable meshes," *IEEE Trans. Parallel Distrib. Syst.*, vol. 7, pp. 806–817, August 1996.

[15] Y. Chen and J. Patel, "Efficient evaluation of all-nearest-neighbor queries," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, 2007, pp. 1056 –1065.

[16] J. Sankaranarayanan, H. Samet, and A. Varshney, "A fast all nearest neighbor algorithm for applications involving large point-clouds," *Computers & Graphics*, vol. 31, no. 2, pp. 157 – 174, 2007.

[17] G. S. Iwerks, H. Samet, and K. Smith, "Continuous k-nearest neighbor queries for continuously moving points with updates," in *Proceedings of the 29th international conference on Very large data bases - Volume 29*, ser. VLDB '2003. VLDB Endowment, 2003, pp. 512–523.

[18] E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis, "Algorithms for nearest neighbor search on moving object trajectories," *Geoinformatica*, vol. 11, pp. 159–193, June 2007.

[19] Y.-K. Huang, S.-J. Liao, and C. Lee, "Efficient continuous k-nearest neighbor query processing over moving objects with uncertain speed and direction," in *Proceedings of the 20th international conference on Scientific and Statistical Database Management*, ser. SSDBM '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 549–557.

[20] M. L. Yiu, L. H. U, S. Šaltenis, and K. Tzoumas, "Efficient proximity detection among mobile users via self-tuning policies," *Proc. VLDB Endow.*, vol. 3, pp. 985–996, September 2010.

[21] H. Hu, J. Xu, and D. L. Lee, "A generic framework for monitoring continuous spatial queries over moving objects," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '05. New York, NY, USA: ACM, 2005, pp. 479–490. [Online]. Available: http://doi.acm.org/10.1145/1066157.1066212

[22] T. Rappaport, *Wireless Communications: Principles and Practice*, 2nd ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[23] (2011, Nov.) Geolocation API website. [Online]. Available: http://code.google.com/apis/gears/api_geolocation.html

[24] (2011, Nov.) Universal Mobile Telephone System World website. [Online]. Available: http://www.umtsworld.com/technology/capacity.htm

[25] T. Brinkhoff, "A framework for generating network-based moving objects," *GeoInformatica*, vol. 6, no. 2, pp. 153–180, 2002.

[26] (2011, Nov.) The City of Oldenburg website. [Online]. Available: http://www.oldenburg.de/stadtol/index.php?id=4316&L=0

[27] J. Harri, M. Fiore, F. Filali, and C. Bonnet, "Vehicular mobility simulation with vanet-mobisim," *Transactions of The Society for Modeling and Simulation*, September 2009.

[28] N. Loulloudes, G. Pallis, and M. D. Dikaiakos, "The dynamics of vehicular networks in urban environments," Department of Computer Science, University of Cyprus, Technical Report TR-10-2, June 2010.

[29] (2011, Nov.) U.S. Census Bureau website. [Online]. Available: http://www.census.gov/geo/www/tiger/

[30] (2011, Nov.) Transportation Statistics website. [Online]. Available: http://www.nationmaster.com/

[31] R. Cheng, Y. Zhang, E. Bertino, and S. Prabhakar, "Preserving user location privacy in mobile data management infrastructures," in *Privacy Enhancing Technologies*, ser. Lecture Notes in Computer Science, G. Danezis and P. Golle, Eds., vol. 4258. Springer, 2006, pp. 393–412.

[32] S. Papadopoulos, S. Bakiras, and D. Papadias, "Nearest neighbor search with strong location privacy," *PVLDB*, vol. 3, no. 1, pp. 619–629, 2010.