Buddy tracking — efficient proximity detection among mobile friends

Arnon Amir   Alon Efrat   Jussi Myllymaki   Lingeshwaran Palaniappan   Kevin Wampler

Abstract: Global positioning systems (GPS) and mobile phone networks are making it possible to track individual users with an increasing accuracy. It is natural to ask whether one can use this information to maintain social networks. Here each user wishes to be informed whenever one of a list of other users, called the *user's friends*, appears in the user's vicinity. In contrast to more traditional positioning based algorithms, the computation here depends not only on the user's own position on a static map, but also on the dynamic position of the user's friends. Hence it requires both communication and computation resources. The computation can be carried out either between the individual users in a peer-to-peer fashion or by centralized servers where computation and data can be collected at one central location. In the peer-to-peer model, a novel algorithm for minimizing the number of *location update messages* between pairs of friends is presented. We also present an efficient algorithm for the centralized model, based on region hierarchy and quadtrees. The paper provides an analysis of the two algorithms, compares them with a naive approach, and evaluates them using the IBM City Simulator system.

Keywords: Strips algorithm, location based services, social networks, global positioning systems, dynamic nearest neighbors.

Arnon Amir and Jussi Myllymaki are with IBM Almaden Research Center, 650 Harry Rd., San Jose, CA 95120. **Email:** {*arnon,jussi*}*@almaden.ibm.com*

Alon Efrat, Lingeshwaran Palaniappan and Kevin Wampler are with the Computer Science Department, The University of Arizona **Email:** {*alon,lingesh,wamplerk*}*@cs.arizona.edu.*

I. Introduction

Global positioning systems and mobile phone networks make it possible to track individual users with an increasing accuracy. One attractive application of knowing the geographic location of users is to compute and maintain *social networks*. In these networks, each user may specify or be associated with a group of other users, called the *user's friends*. Whenever a friend moves into the user's vicinity, both users are notified by a *proximity alert message*. In a more general context, a *social group* is one that is predefined by enrollment or by matching the personal profiles of users. A group may refer to a list of individuals but also to other groups of individuals.

We use the term *vicinity* to refer to a region around the user. In this paper, a vicinity is represented by a circle of a prespecified radius, which can be uniform for all users, or defined for each pair of friends. The proposed algorithm for the peer-to-peer model can naturally accommodate a different vicinity radius for each pair of friends, as well as other convex vicinities. Other definitions of vicinity, and even dynamically changing definitions, are possible. For example, the radius might change between daytime and night time, it might depend on the user's location, and it might be a non-circular shape.

The problem of maintaining social networks is a form of a dynamic, continuous query into a database of multiple moving entities. In some applications this could as well be part of a "find" query, coupled with other properties, such as profession, employer, user profile or calendar scheduling constraint. A natural example is of a travelling businessman who attends a large conference and would like to be alerted and possibly meet other colleagues if they happen to be around. Other examples could be a road side service, where service cars need to be dynamically assigned to new customers, or a surveillance system which tracks multiple suspects and directs security personnel. Note, however, that if one would try to implement such a query in a traditional database, this would require continuous updating of the locations of all moving entities, or users, as well as a repetitive computation of all friend's distances after each such location update. This would be a very inefficient process.

Maintaining social networks based on user locations is an interesting problem from the aspect of computational geometry and from a database perspective. It is also interesting from the point of view of a distributed system; the process is computationally expensive, but there is an efficient way to split the computational task among different geographic locations.

We distinguish between two different computational frameworks. In the *centralized* computation model, users send their location information to a centralized server which keeps tracking of user locations and lists of friends and is responsible for computing and sending the alert messages to all pairs of friends. The second, *peer-to-peer* computation model, involves no central server. Instead, each pair of friends is responsible for keeping each other informed about their location, detecting vicinity events, and transmitting alert messages.

The peer-to-peer model suggests several benefits, including

Privacy. This model ensures that only the user and his friends know its exact location. At any time, a user may exchange location update messages with only those specific users it wishes to.

Energy efficiency. In general, battery drainage of small mobile devices resulting from communications is far more significant than the energy needed for computing. For example, it was shown [17] that under Rayleigh fading and fourth power distance loss, the energy cost of transmitting 1 KB a distance of 100 m is approximately the same as executing three million instructions on a 100 MIPS/W processor. The peer-to-peer model minimizes the number of location update messages sent by the user, on the expense of the much-less-costly increase of computation cost.

Localization, flexibility. The peer-to-peer algorithm may be implemented by a group of mobile devices users without any need to modify the infrastructure of the communication provider. All they need is to agree on a protocol to exchange location messages and to apply the proposed algorithm. This approach is best suitable for ad-hoc networks.

Under both frameworks there is a need for communication resources as well as for computational resources. Communication is required to deliver location updates and alerts. We characterize the amount of communication by the number of messages being exchanged, assuming that all the messages are of fixed length (i.e., a location update sent to k users, for example, would require $O(k)$ messages). A message between two users is assumed to cost the same as a message between a user to a centralized server, although some implementations might have a constant factor between them. In addition, computational resources are needed, either on the server or on the participating moving devices, to keep and maintain data structures and to generate proximity alerts. This paper considers both the communication and the computation resources. It focuses, however, on reducing the communication complexity, as air time and battery life seems to be the more expensive and restrictive implications in building a real system.

There are several considerations that impact both computational and communication complexities. A major consideration is the maximal velocity of users and the desired time/distance accuracy of alerts. Let us consider two users who wish to get an alert when the distance between them becomes smaller than R. Obviously, one cannot guarantee such an exact alert, as this would require an infinitely large number of location updates and distance computations to find the exact moment. To overcome this problem we introduce a distance tolerance into the task. An alert needs to be sent before they are within a distance R of each other, but not earlier than a distance of $R + \varepsilon$. This model allows us to compute how many messages would be required to achieve any desired alert accuracy. Last, it is assumed that messages are transferred with no delay. This is equivalent to the assumption $v \cdot t_{delay} < \varepsilon$, for v representing the velocity and thus a distance tolerance at the time of receiving the alert, $R - \varepsilon$, can naturally incorporate such practically small delays.

The evaluation of an algorithm for maintaining social networks is an important issue. The number of messages would depend not only on the number of users, n, the distances between them, the vicinity radius R and the desired tolerance ε, but also on the nature of their motion trajectories and relations between them. In the computational geometry literature, the *kinetic model* (see [8], [12]) is a common paradigm for evaluating the efficiency of algorithms for maintaining dynamic structures. In this paradigm, the role of the evaluated algorithm is to maintain some geometric properties for sets of moving elements, where each element moves along a low-degree algebraic curve. From time to time, an *event* occurs, in which new elements may be inserted and existing elements may be deleted or may change their trajectories. The number of changes in the data structure is evaluated as a function of the number of events in the dynamic input data set. This part of our analysis is not shown here due to space constraints, and is found in [5].

In cellular networks (e.g. mobile phone networks), a partial approach is to try to make advantage of the natural cells structure imposed by the network. If R is approximately the radius of a cell, then one needs to keep track of friends registered to the user's own cell and neighboring cells. However, in general this approach might be unsatisfactory because the cell sizes vary greatly, ranging from large macro cells in rural areas to tiny pico cells in metropolitan areas and buildings. Different users might as well define different vicinity radii for different friends, and these might even change when they move from one place to another. For example, a marketing manager does not want to be alerted in his office of all his colleagues who are close by in the office, but may want such alerts when the same colleagues are within a city block distance on an overseas trip, as this is a chance encounter. Also note that not all wireless communication is based on cellular networks in the first place.

A. Related work: Algorithms for tracking moving objects are found in mobile computing literature, both in the database community, and in the mobile communications community. Much of the work assumes that moving objects are represented by simple point objects whose locations are continuously updated in an index. This however requires continuous updating of the locations of all users, which would result in a huge number of location messages. Trajectory-based algorithms are becoming increasingly popular [18], [19]. Storing and indexing trajectories facilitates not only efficient spatial range queries but time-and-space range queries [2]. See also [1], [20]. [21] discusses time-parameterized bounding rectangles and extends trajectory information with *expiration* information.

There is a large body of literature on maintaining a specific property of moving objects. For example, a randomized algorithm for maintaining the binary space partition of moving objects is discussed in [3], and the maintenance of the dynamic Voronoi diagram of a set of moving points in a plane is presented in [13]. For maintaining and querying a database of moving objects, see [26].

Various algorithms have been provided for indexing moving points. A quadtree based algorithm for indexing is given by [25]. Their main idea is to use a linear function of time for each of the dynamic attributes of the object, and to provide methods to regenerate the quadtree. An R^*-tree based algorithm is given by [22]. Their algorithm provides indexing of objects moving in 1, 2 or 3 dimensions. For other work in query processing for moving points, see e.g. [11], which proposes algorithms for range query and k nearest neighbours.

While the dynamic data structures or databases mentioned above may be efficient for other types of queries, the task in hand is not efficiently handled by any of them. For example, some of these data structures might be used for querying at a particular time instance who are the friends in the vicinity of a single user, around its present location. However, consider that today there are hundreds of millions of mobile phone users in the world. Repeatedly tracking all users and querying their vicinities in such a large population requires a huge number of messages to be exchanged and a lot of computation and is very inefficient. To the best of our knowledge, the problem of maintaining social networks has not been addressed before. The algorithms and data structures proposed in this paper are designed to efficiently handle this task. However they might not be as efficient for other, traditional types of spatial queries, thus being complementary to the above previous work. This work is also complementary to the problem of finding people whose personal profiles match. For this problem, commercial solutions have been offered (see e.g. [15]).

B. Our results: We present a novel distributed algorithm, denoted as the *strips algorithm*, in which a pair of moving friends make an agreement about a static buffer region between them. After the agreement is made, they do not need to track each others location until one of them enters the buffer region for the first time. By doing so, the agreement is terminated. Hence they exchange a location update message between them, check if they are within the R-vicinity of each other, and otherwise make a new agreement on a new static buffer region. We provide an exact and an approximate strips algorithms, supported by analytical and empirical results that show their efficiency.

When analyzing an algorithm for such a problem, one has to consider both communication and computation complexity. For this distributed algorithm we focus on reducing the communication complexity, or the required "air time", which has a significant impact on the battery lifetime of a mobile communication device. It is shown that the number of messages is logarithmic in the distance between the users when they start to approach each other from far away. It is also logarithmic in $1/\varepsilon$ when they are getting closer, where ε is the desired tolerance for producing the proximity alert. Hence we consider it to be a very efficient algorithm.

A second, quadtree based algorithm is presented for the centralized approach. This algorithm aims at reducing the computational cost, assuming that all users periodically update the server with their location as they move. Somewhat surprisingly, we found that the quadtree based centralized algorithm is inferior to a centralized implementation of the strips algorithm on such a central server.

II. The Strips algorithm

In this distributed, peer-to-peer model, it is assumed that each user carries a wireless device that knows its own location and has enough computational power for a local computation. In order to compute its distance from a friend it needs to get the location of that friend, and this requires a location update message to be sent. Our objective is to minimize the communication complexity, or the number of location update messages exchanged with devices of other users.

Let a, b be two users whose Euclidean distance, denoted $|b-a|$, is larger than R. Let $\ell(a,b)$ denote the bisector of the line connecting a and b; i.e., the line consisting of all points of equal distance from a and

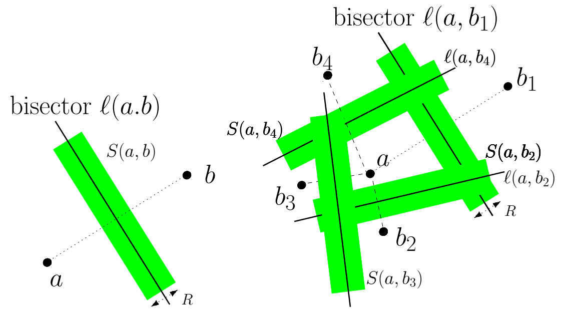

Fig. 1. Left: setting a new static strip of width R around the bisector between two mobile users. Right: user a does not need to update strips while moving inside the internal region (P).

b (see Figure 1-left). Let $S(a,b)$ denote the infinite strip of width R whose central axis is $\ell(a,b)$. Let e_i denote the line bounding $S(a,b)$ on the side closer to a. Note that while a might move continuously, the strip $S(a,b)$ is not being updated unless a specific event occurs that requires this update. Hence the strip update is a discrete event in time.

When a new user a is added to the system, it communicates with each of his friends $\{b_1 \dots b_n\}$, queries their locations, and announces its own location to them. For each b_i for which $|a-b_i| > R$, we insert the strip $S(a,b_i)$ into the data structures of a and of b_i, denoted $\mathcal{D}(a)$ and $\mathcal{D}(b_i)$, respectively. Setting a new strip is illustrated in Figure Figure 1-left. The strip $S(a,b_i)$ divides the planes into 3 regions, namely the strip $S(a,b_i)$ itself, the region $S^+(a,b_i)$ containing a, and the region $S^-(a,b_i)$ containing b_i. The idea behind this method is that as long as neither a or b_i enters $S(a,b_i)$, they do not need to exchange location update messages. The strip serves as a static buffer region between a and b_i and ensures that as long as they are on both sides neither one of them is in the vicinity of the other. Clearly this region can be of different shapes. Its desired properties are to have a Hausdorff distance $> R$ between its two borders, to provide maximum motion to each user on its side of the strip, and to postpone as much as possible the event of a or b_i intersecting with it. Since the strip is static, there is no need to exchange location or any other messages before an intersection event occurs.

Once a (resp. b_i) enters $S(a,b_i)$, it communicates with b_i (resp. a), sending a location update message to it, and receiving a location update message from b_i (resp. a). Note that all other $b_j, \quad j \neq i$ remain

```
SelfMotion()  {
   do // repeat while moving
      $a$=ReadSelfLocation()
      Test($\mathcal{D}(a)$)
         if ($a$ enters $S(a,b_i)$,
             (for some $i$))
              or MsgReceived($b_i$))
         StripUpdate($b_i$)
   enddo }
```

```
StripUpdate($b_i$)  {
   send $a$'s location to $b_i$.
   receive $b_i$'s location.
   if $|a-b_i| < R+\varepsilon$
      ProximityAlert("$b_i$ is nearby")
      Delete( $\mathcal{D}(a)$,$S(a,b_i)$)
   else
      Compute $S(a,b_i)$
      Update $\mathcal{D}(a)$, $\mathcal{D}(b_i)$ with $S(a,b_i)$
   end
}
```

Fig. 2. Pseudo code of the Strips algorithm, as is ran by user a.

intact and exchange no messages with a. Next, a and b_i both check if the distance between them is $\leq R+\varepsilon$, and if so, an alert message about their mutual proximity is triggered. If the distance between them is still larger than $R+\varepsilon$, then they compute a new strip, $S(a,b_i)$, using their current new locations, and update $S(a,b_i)$ in their data structures. The algorithm is summarized in Figure 2. The interested reader may download a Java implementation of our algorithms at *www.cs.arizona.edu/people/alon/cell.*

The Strips algorithm demonstrates a *peer-to-peer* paradigm, where a strip $S(a,b_i)$ may be interpreted as an agreement between a and b_i. As long as this agreement is not invalidated, i.e. both of them stay outside $S(a,b_i)$, there is no need to exchange any further messages.

A. The data structure $\mathcal{D}(a)$ for the Strips algorithm: Let $P=\cap_i S^+(a,b_i)$ denote the region which contains a as shown in Figure Figure 1-right. Clearly, as long as a stays inside P, and no b_i enters $S_i(b_i,a)$, there is no need for a to exchange any messages. P is a convex polygon of at most n edges, where n is the number of a's friends. The edges of P are segments from $\{e_i\}_1^n$. Below we describe how to efficiently find in time $O(\log n)$ whether a exits P through any of the edges, say e_i. Moreover, once a exits through the edge e_i, (i.e. a new $S(a,b_i)$ should be computed) we show how to update $S(a,b_i)$ in $\mathcal{D}(a)$ in time $O(\log n)$.

The data structure is based on the standard dual transformation, defined in Computational Geometry (see [9]). It transforms a point $p=(a,b)$ in the primal plane to the line $p^*=ax+b$ in the dual plane, and the line $\ell: y=mx+n$ in the primal plane to the point $\ell^*=(m,-n)$ in the dual plane. We divide the set of lines $\{e_i\}_1^n$ into two sets: U, the lines above a, and D, the lines below a. Lines which are vertical are treated separately. Let U^* and D^* denote the sets of points in the dual plane which are the dual of U and D, respectively. If a happens to cross line e_i which belongs to U, while still in the closure of P, then the corresponding line a^* would intersect the point e_i^*, which must be a vertex of the convex hull of U^*. In this case, we delete $S(a,b_i)$ from $\mathcal{D}(a)$ and insert a new strip $S'(a,b_i)$. This corresponds to deleting the vertex e_i^* from the convex hull of U^*, and inserting the point $e_i'^*$, where e_i' is the line bounding $S(a,b_i)$ on the side closer to a. Hence we need to maintain the convex hulls U^*, D^* in a dynamic way, so that their intersection with a query line, as well as deleting and inserting points from and to the convex hulls, can be accomplished efficiently. We use the data structure of [10], where an update can be handled in (amortized) $O(\log n)$ time and a query can be done in $O(\log n)$ time.

Once two friends entered the vicinity of each other, the algorithm needs to detect when they get apart. Then the system would return to its original state. When two friends are found to be at distance less then R from each other, a circle of radius $R/2+2\varepsilon$, centered at the midpoint of the line connecting the two friends, is created. This circle is treated much the same way as a strip in that so long as both users remain within the circle, it is guaranteed that they are within a distance of less than $R+4\varepsilon$ from each other. When one of the users leaves the circle, a location update message must be exchanged between the friends. If the friends are still within a distance smaller than $R+2\varepsilon$ from each other, a new circle is computed. Otherwise, a state change occurs, the friends are again apart, and a new strip is computed.

It is natural to assume that if two friends have been notified of their proximity to each other they will meet if they choose to do so, without need for further messages. Hence the bounding circle could be made of a radius much larger than $R/2 + 2\varepsilon$ without diminishing the usability of the system. An alternative to using a circle could be to exchange messages every fixed amount of time, or after increasing periods of time. In the simulations we did not count for messages exchanged while the friends are in the vicinity of each other, as we are much more interested in efficiently detecting the proximity event than detecting the separation event.

B. An approximated Strips algorithm: The Strips algorithm requires computing a dynamic convex hull, which might be challenging if the computational power of the mobile device is limited.

The *approximated Strips algorithm* proposed here relaxes the vicinity definition to simplify the computation algorithm. Rather than maintaining a convex region of possibly $\Theta(n)$ edges, for n friends, it maintains an approximated polygon of a fixed number of edges, at fixed, predetermined slopes. The following description is for the case of using only four edges, i.e. a bounding rectangle region. This case is of a particular interest, as it can be handled by a very simple data structure. In this case, strips can only be horizontal or vertical. The strip between a and b_i is horizontal if $|x_a - x_{b_i}| < |y_a - y_{b_i}|$, and vertical otherwise.

In either case, the strip is located so that its distance from a is equal to its distance from b_i. The boundaries of all vertical strips are maintained in a balanced search tree, sorted by their x values. Similarly, the boundaries of all horizontal strips are maintained in a second balanced search tree, sorted by their y values. This data structure is of course much simpler than the dynamic convex hull described in Section II-A.

User a may move within the rectangular region around a without issuing any location updates. A (practical) update time of $O(\log n)$ is obtained in the balanced tree. At the time of an update, the strip that was hit is removed from its tree, a new strip is computed based on the updated locations, and inserted into the appropriate tree based on its (new) direction. If no strip of size R can be placed between the two users in any of the fixed slopes (i.e., in this case, neither the horizontal nor vertical distance between the users is larger than $R + \varepsilon$) , then an meeting alert is generated. The distance at the time of a meeting alert is smaller than $\sqrt{2}(R + \varepsilon)$. This approximated bound gets closer to the original $R + \varepsilon$ requirement as the fixed number of directions is set higher.

Experimental comparison results between the exact and the approximated Strips algorithms is presented in Section V.

III. Analysis of the strips algorithm

In this section we provide analytical and numerical analysis of basic cases to illustrate the efficiency of the strips algorithm.

A. The role of ε in the Strips algorithm: The selection of ε determines a tradeoff between the desired distance accuracy in generating alerts and the required number of location update messages. First we address an obvious, yet an important stability aspect of the algorithm, which depends on ε.

COROLLARY 3.1. *Two users have to move a total distance of at least 2ε between any two proximity alerts they generate.*

In order to generate two proximity alerts, one user has to enter the vicinity of the other, then exit the vicinity, and then enter it again. Hence the users are at most $R + \varepsilon$ apart when the first proximity alert is invoked, then they are at least $R + 2\varepsilon$ apart when they get apart, and then at most $R+\varepsilon$ apart when the second proximity alert is invoked. This corollary ensures that the algorithm state will not change back and forth in infinitely small time periods when the two users are moving around the boundary of the vicinity region.

Next we illustrate the role of ε in the algorithm termination criteria (i.e., announcing a proximity alert). Consider a simple case of two users a and b. Let user a be stationary, and let user b be moving on a straight line towards user a. Denote the initial distance between a and b by $R + x$, for a positive x. When user b hits the strip, its distance from user a would be $\frac{x+R}{2} + \frac{R}{2} = \frac{x}{2} + R$ (half the initial distance x plus the width of the strip, R). Similarly, the next strip will be located such that b will hit its boundary at a distance from user a of $\frac{\frac{x}{2}+R}{2} + \frac{R}{2} = \frac{x}{4} + R$. Hence it is clear that this sequence of strip-update events forms a series of distances which is the sum of one constant component, R, and a geometric series, $\frac{x}{2^k}$, $k = 1, 2, 3, \ldots$. The termination condition for this series of strip-update events is when $R + \frac{x}{2^k} < R + \varepsilon$, or just $\frac{x}{2^k} < \varepsilon$. Hence we can find k by the following corollary.

COROLLARY 3.2. *The number k of strip-update events performed by a stationary user a and a user b moving on a straight line towards a from a distance $x + R$ is $k = \lfloor \log_2 \frac{x}{\varepsilon} \rfloor + 1$*

Hence the number of messages exchanged between users a and b is logarithmic with the initial distance between them, and is also logarithmic with $1/\varepsilon$, the inverse desired tolerance. This reflects the tradeoff between the desired accuracy and the required number of location update messages. It is a very small number of updates, demonstrating the efficiency of the algorithm. As indicated earlier, as ε decreases to zero, the number of messages k increases to infinity. That is, an alert at the exact time would require an infinitely large number of location update messages. By introducing a tolerance $\varepsilon > 0$ into the model we avoid this undesired case.

B. The general case of moving on a straight line: Next we consider a more general case in which b is moving on a straight line, but the stationary user a is located at distance d from the line. Figure 3 shows a typical sequence of updates. The point b is moving from right to left along the $y = 0$ line, and its positions at the times of hitting the strips are marked by small circles, labeled with the update serial number. The strips are shown in dashed lines, and are similarly numbered. For instance, when user b hits strip number 1 it defines the location of point 2, and so on. In this example b passes nearby a but out of its vicinity and no proximity alert is produced. After the 8th update b would continue to move and no more updates will be done.

The user b starts at point $(x, 0)$. The intersection with the strip occurs at $(x', 0)$, where $x' = \frac{x}{2} - \frac{d^2}{2x} + \frac{R}{2}\sqrt{1 + \frac{d^2}{x^2}}$. This process repeats iteratively until one of two termination cases occurs. One is as illustrated in Figure 3, where no alert is produced, and the other one is when $d < R + \varepsilon$, in which case b enters the vicinity of a and a proximity alert is produced. Figure 4 shows the number of updates as function of d/R and ε. As can be seen, the highest number of

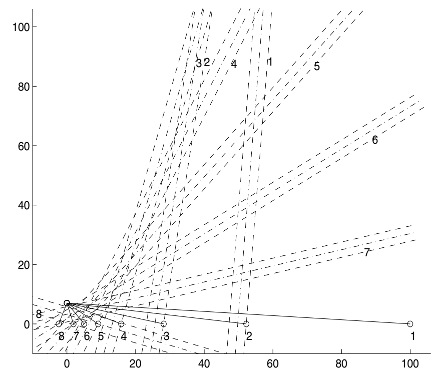

Fig. 3. A sample series of strips for a user b moving on the x axis from right to left, where user a is at $(0, d)$.

messages is required when b passes very close to a, but still keeps out of its R-vicinity. The region in the graph for which $d < R$ terminates with a proximity alert after producing a number of strings which is linear with $\log(1/\varepsilon)$ (note the logarithmic axis, which represents ε values between 0.00001 and 10.0). The region in the graph for which $d > R$ corresponds to cases where there is no proximity alert, and thus after getting away from the transition area along $d = R$, the number of messages does not depend on ε at all. Hence we see that for long distances the number of messages is logarithmic with the distance, and for short distances the number of messages is logarithmic with $1/\varepsilon$. We consider it to be a very efficient property of the algorithm.

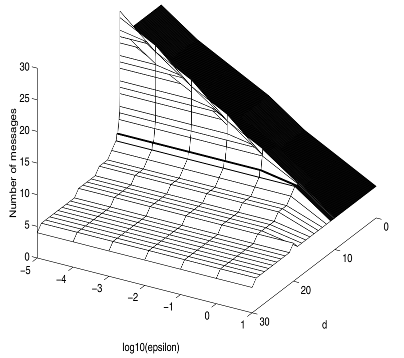

Fig. 4. The effect of d and ε on total number of strip updates ($R = 10.0$, ε is given in a logarithmic scale). Two regions are observed, one corresponds to $d < R$ and the other to $d > R$. See text for details.

We further analyze this behavior for the case of $\varepsilon = R$, which is later used in some of the simulations. Let a and b be two users, with a staying at a fixed position and b moving in a straight, vertical trajectory towards a point c. Let d denote the horizontal distance of b from a, and let v be the vertical distance of b from a. Assume also for simplicity that users exchange location update messages when they hit the center of the strip. Theorem 3.1 upper bounds the number of location update messages sent.

THEOREM 3.1. *The number of times that the strip is updated is $\leq \log_2(v/R) + 1$ if $d < \sqrt{3}R$, and is $\leq \log_2(v/d) + 1$ if $d \geq \sqrt{3}R$.*

For the same scenario in the approximated Strips algorithm the following theorem could be proved using the similar tools. The proof appears in the full version of this paper.

THEOREM 3.2. *The number of times that the strip is updated is $\leq \log(v/(d - R))$ if $d > R$, and is $\leq \log(v/R)$ if $d < R$.*

REMARK 3.1. *Users who move on arbitrary paths may cause an increasing number of strip updates over time. In the worst case, a user has to move at least $\varepsilon/2$ between two location update messages. This is a tight bound, achieved when one user follows a friend on a straight line, at the same speed, and with a distnace slightly larger than $R + \varepsilon$.*

IV. Centralized algorithms for social networks

The Strips method, which was described in a peer-to-peer distributed fashion, is very efficient even if implemented on a central computational facility. It allows the radius of vicinity to be different from user to user, and even allows a different and an asymmetric vicinity definition between pairs of users. However, it is worth mentioning an alternative approach, which might be useful if the radius of vicinity is the same between all pairs of users, and the user is willing to accept a rough level of approximation in the radius of vicinity. This approach is based on a quadtree representation, a regular data structure which is commonly used in GIS. This approach seems better suited cellular networks where each cell or a cluster of cells are capable of performing some computation.

Here we assume that the "friendship model" is symmetric; that is, user a is a friend of user b if and only if user b is a friend of user a. This assumption is not critical, and can be removed, by maintaining for every user a two lists, namely the friends of a, and the lists of users that have a as a friend. The algorithms described in this section is guaranteed to send a proximity alert message if the distance between friends is approximately R. More precisely, it guarantees to send a message if the distance is $\leq R$, and not to send a message if the distance is $\geq 2\sqrt{2}R$. This is equivalent of selecting $\varepsilon = (2\sqrt{2} - 1)R$. Like with the Strips algorithm, whenever the distance between two friends is larger than R and smaller than $R + \varepsilon$, there is no guarantee about sending or not sending a proximity alert.

The centralized scheme is designed for the case where the wireless devices carried by users do not have much computational power, or if for other reasons we prefer to perform the computation at a central site. In particular, there might be cases where the center knows the location of all the users at all times and does not need to send special messages to ask them for location updates at all.

A. The "naive" quadtree algorithm: Let Γ be a partition of the plane into *regions*, defined recursively as follow

- Initially Γ consists of a single square region, covering the entire area of operation
- Let c be a region of Γ. If c either contains ≤ 1 users, or its edge-length is $\leq R$, stop. Otherwise, replace c with 4 equal-size squares R_1, R_2, R_3, R_4, representing its four quadrants.

This division imposes a quadtree data structure $\mathcal{T}$ (see [23], [24]), with the property that every leaf-region whose size is larger than R contains at most one user. We call a region containing more than a single user a *live region*. Hence live regions are always of same edge size, $\leq R$, and from here on we assume their size is exactly R. We augment $\mathcal{T}$ so that it is a *balanced* and a *netted* quadtree [23]. That is, the difference in size between two neighboring leaf regions is at most a factor of 2, and each leaf region maintains pointers to all of its neighboring regions.

For every user b_i let $F(b_i)$ denote the list of friends of b_i. For every region $c \in \Gamma$ let $\mathcal{U}(c)$ denote the users currently inside c. The basic idea is as follows. Once user b_i registers with the system and reports its location, the system seeks friends of b_i in the region $c \in \Gamma$ containing b_i, and in the neighboring regions of c. Note that there are at most 8 neighboring cells of c, since $\mathcal{T}$ is a balanced quad-tree. When b_i moves from region c to a new region c', we only need to check if any friend(s) of b_i are found in any of the *new* neighboring cells. We next explain each stage in detail.

Finding the region c containing b_i. Let h denote the height of the quad-tree $\mathcal{T}$. A simple approach would be to use $\mathcal{T}$ itself for this point location task. It requires tracing the path of length $\leq h$ from the root of $\mathcal{T}$ and to the leaf associated with c. Alternatively, we can use the point location technique of [4] that requires only $O(\log \log h)$ time. However, in many cases, we can do better; Note that the size of all live regions containing two or more users is exactly R. The coordinates of the left lower corner of such cells is $(m_x R, m_y R)$, where m_x, m_y are integers. We refer to the pair (m_x, m_y) as the *index of* c. Note that for every point $p = (x, y)$, the index of the live leaf region containing p, if one exists, is given by $(\lfloor x/R \rfloor, \lfloor y/R \rfloor)$. Hence we store all the live regions in a hash table where the key is the index of the live region. We maintain a pointer from the hash cell to the leaf of $\mathcal{T}$ associated with this region. Hence finding users within the same region as user p is done in expected time $O(1)$. Once this region is found, then by using the properties of the netted quadtree, finding the neighbor regions is done by following the links to the neighbors, in $O(1)$ worst-case time.

Finding if there are any friend of b_i in its vicinity. For each live region c we maintain a hash table of all the users currently inside c. We also maintain for every user b_i a hash table of its friends. When b_i enters a new region c', we compare the length of the friends list with the length of the combined lists of occupants in the (up to four) neighboring regions to c' which are not neighbors of c. If the friends list is shorter, we check the distance to every friend of b_i. Otherwise, we check for all occupants of the neighboring regions which of them is a friend of b_i. Thus the running time is $O(\min\{|F(b_i)|, |\mathcal{U}(c)|\})$ (in the expected sense, due to the use of hash tables).

B. Improved centralized quadtree algorithm: The purpose of the improved algorithm is to reduce the size of the list of friends that b_i needs to check upon entering a cell c. Let q_1, q_2 and $\pi(q_1, q_2)$ denote two nodes in T and the path in $\mathcal{T}$ from q_1 to q_2, respectively. Let p, R_p denote a node in the tree and the region in the plane associated with it, respectively.

We define the lists $F_p(c_i)$ in a bottom up fashion. If p is the leaf node, we define $F_p(b_i)$ to denote the set friends of b_i which are in R_p. If p is not a leaf node, we define $F_p(b_i)$ to denote the friends of b_i, which are not in $F_{p'}$ for any decedent node p' of p. In other words $b_i \in F_p(a)$ if and only if p is the lowest common ancestor of the leaves nodes containing a and b_i. An intuitive way to understand this definition is to think of $F_q(b_i)$ as all the friends of b_i who are in the same city as b_i but are not in the neighborhood of b_i.

Entering a new user. When a new user b_i registers into the system (e.g., by turning on its cellular phone), we find the leaf region R_p containing b_i, check the list $F(b_i)$ of b_i's friends, check their location in $\mathcal{T}$, and create the lists $F_p(b_i)$. This can be done efficiently in expected time $O(h + |F(b_i)|)$, where h is the height of $\mathcal{T}$.

Handling a cell crossing event. An event happens when b_i moves from one leaf-region R_q to another leaf-region $R_{q'}$. If $R_{q'}$ contains already at least one user, we might need to split $R_{q'}$, depending on if its size is larger than R. We traverse up the tree $\mathcal{T}$ from q until we reach θ, the lowest node of $\mathcal{T}$ for which R_θ contains both $R_q, R_{q'}$ and all their neighboring leaf regions. We find all friends of b_i that occupant $R_{q'}$ or any of its neighboring leaf regions new to b_i by checking for all friends that are stored in one of the lists $F_p(b_i)$, for $p \in \pi(q', \theta)$. Let $\mathcal{L}$ denote this list of friends (that is, $\mathcal{L} = \mathcal{F}(b_i) \cup \cup_{p \in \pi(\theta, q')} F_p(b_i)$) A proximity alert is sent to each such friend that we find.

Next, for each friend a which is checked, we also need to update its lists of friends, to inform them about the change in the location of b_i. For this, we delete (resp. insert) b_i from (to) the lists L_p for all $p \in \pi(\theta, q')$. Then we update these lists.

While this data structure is more complicate than the naive quadtree, it allows us to derive some theoretical bounds on the number of updates which will occur for certain families of motion trajectories. These support our claim that this data structure is efficient.

THEOREM 4.1. *Let γ be a curve in the plane, with the property that every straight line crosses γ at at most*

K points, where K is a small constant. Let b_i move along γ, and assume that all its friends are stationary. Then the total number of checks and updates done by the improved centralized algorithm is only $O(n_i h)$, where n_i is the number of friends of b_i, and h is the height of $\mathcal{T}$. This bound is tight in the worst case.

Theorem 4.1 shows that for complexity-bounded motion trajectories the number of updates is also bounded (proof is omitted). It is harder to give any complexity bounds if all friends of b_i are allowed to move arbitrarily. For example, it seems that if all of them move together with b_i, then there is nothing we can do in this algorithm except checking all of them each time b_i enters a new cell. However, if most of them are located far away from b_i, for most of the time, then we expect this algorithm to be quit efficient.

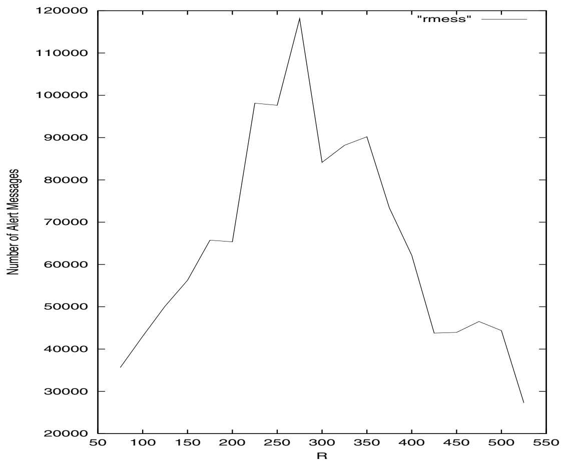

Fig. 5. Number of Proximity Alert Messages Generated as a Function of R

V. Experimental Results

We conducted simulation experiments with the Strips algorithm and the quadtree methods on synthetic dynamic location data. Location trace data was created with the City Simulator [16], a toolkit that simulates an arbitrary number of mobile users moving in a city, driving on streets or walking on sidewalks and entering buildings. Several aspects of the toolkit and trace can be controlled, including traffic flow patterns, traffic congestion, and blocked streets and routes. The position (location report) of each user is computed at fixed intervals and output to a trace file. For these experiments we created a trace with 2000 users and 200 location reports per user.

In Figure 5 we show the number of proximity alert messages, plotted against R, the radius of user's vicinity. Note that this simulation is independent of the algorithm used. It can be seen that the number of alerts reaches a maximum at a midpoint, for $R = 275$. This is expected, as if R is very small then getting two friends close enough has very small chance. On the other hand, if R is very large, then nearly everyone are in the vicinity of all their friends, and thus very few new meeting events are likely to occur. In between, for mid range R values, there is a higher chance for a pair of users to alternately get close enough to each other and then get apart from each other. This might occur several times while they move along their traces.

Figure 7 shows the total number of cell-crosses by all users, in the quadtree algorithm. This number decreases as R increases, because the minimal cell size is also R. Note that this number represents the number of times the algorithm needs to check the neighboring cells for friends of the user who crosses cells.

Figure 8 shows the number of strip updates, each correspond to one location update message in the Strips algorithm. In general, increasing R causes less number of strip updates to occur. It is analogue to shortening the traces, as the actual length of the trace should be measured with respect to ε, here equal to R. Although it is somewhat similar to the result for the quadtree algorithm, Figure 7, note that here we count all the communications between pairs, while in the previous graph we count only the individual's events of passing from cell to cell. One depends on the number of friends, while the other does not.

The exact and approximate Strips algorithms.

Figure 6 compares the number of messages sent by the exact Strips algorithm and by the approximated Strips algorithm with bounding rectangles. The experiment was repeated for various values of R, plotted on the horizontal axis. As expected, the number of messages transmitted by the approximated Strips algorithm is higher than the (exact) Strips algorithm. The approximated Strips algorithm turns out to send approximately 1.8 times more location update messages than the exact Strips algorithm. While requiring more messages and being less accurate, the approximate Strips algorithm uses a simpler data structure, less

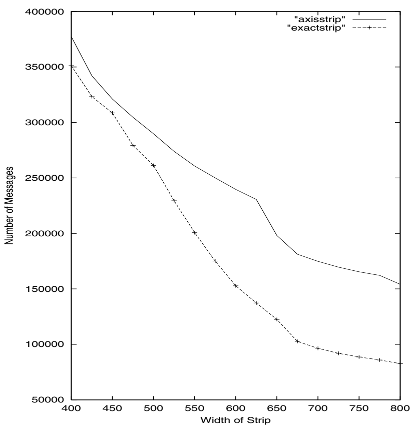

Fig. 6. Comparison of Exact Strips and Approximated Strips approach

memory and less computational resources than the exact one.

The Strips algorithm vs. the quadtree algorithm. Comparing between these two algorithms is not an obvious task. The Strips algorithm, designed for peer-to-peer operation, aims at minimizing the communication complexity, namely the number of location update messages being sent between pairs of users. On the other hand, the centralized, quadtree-based algorithm, aims at minimizing the computational complexity, assuming that it knows where are all the users at all times (or, at least at all cell crossing events). Also note that the Strips algorithm accommodates any values of R and ε, while the quadtree algorithm is constraint to a uniform R value and a single, rough tolerance criteria, $\varepsilon = (2\sqrt{2} - 1)R$. These differences limit the ability to compare between the two algorithms. We compare between the two by counting basic operations in both, thus mixing between computational and communication complexities for the sake of plotting one combined graph.

Figure 9 shows the number of basic operations relative to the number of friends per user, for the Improved quad tree approach, naive quadtree approach and Strips approach. A basic operation in the Strips algorithm is a location update message, which correspond also to one strip update. A basic operation in the centralized algorithm is checking whether a particular friend of the user is in any of the nearby cells. Thus each time a user crosses a cell boundary, the number of basic operations performed by his cell phone is equal to the minimum between the number of friends and the number of neighbors the user has. The graph shows that the number of basic operations is about linear with the number of friends in all the three cases. Note however that, depending on the number of users in the neighboring cells, the number of operations could grow slower than linear. Also notice that the number of basic operations in the strips algorithm is much lower than the quadtree based algorithms. It is apparent from the figure that Strips algorithm performs better than the improved quadtree approach which in turn performs better than the naive quadtree approach. If we put the communication aspect aside, it means that the Strips algorithm also computes less times the distances between user pairs. Thus when implemented on a centralized server, the Strips algorithm will be more efficient than the quadtree algorithm. Thus we see that the Strips algorithm outperforms the quadtree algorithm in both centralized and distributed settings.

Acknowledgments: We would like to thank Rick Snodgrass for helpful discussions.

References

[1] P. Agarwal and C. M. Procopiuc, Advances in indexing mobile objects, *IEEE Bulletin of Data Engineering*, to appear.

[2] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. of the ACM Symposium on Principles of Database Systems (PODS)*, 2000, 175–186.

[3] P. K. Agarwal, J. Erickson, L.J. Guibas, Kinetic Binary Space Partitions for Intersecting Segments and Disjoint Triangles, *Proc. Ninth Symposium on Discrete Algorithms,* 1998, 107–116.

[4] A. Amir, A. Efrat, P. Indyk and H. Samet, Efficient algorithms and regular data structures for dilation, location and proximity problems. *Algorithmica, Special Issue on GIS* 30 (2001), 164–187.

[5] A. Amir, A. Efrat, J. Myllymaki, L. Palaniappan and K. Wampler, "Buddy tracking — efficient proximity detection among mobile friends," IBM Research Report RJ10250, 2002.

[6] Lars Arge, "The buffer tree: A new technique for optimal I/O-algorithms," *WADS* 1995, 334–345.

[7] J. Basch, L. J. Guibas, and L. Zhang, "Proximity problems on moving points," In *13th Symposium of Computational Geometry*, 1997, 344–351.

[8] J. Basch, L. J. Guibas and J. Hershberger, "Data Struc-

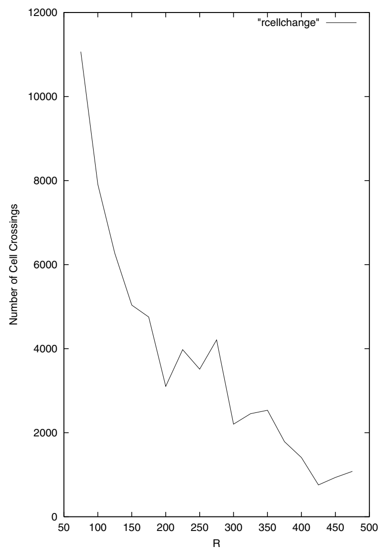

Fig. 7. Total number of cell crossing in the centralized quadtree-based algorithm.

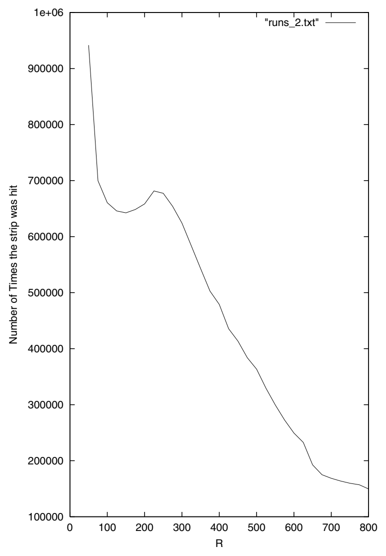

Fig. 8. Total number of strip updates in the distributed Strips algorithm.

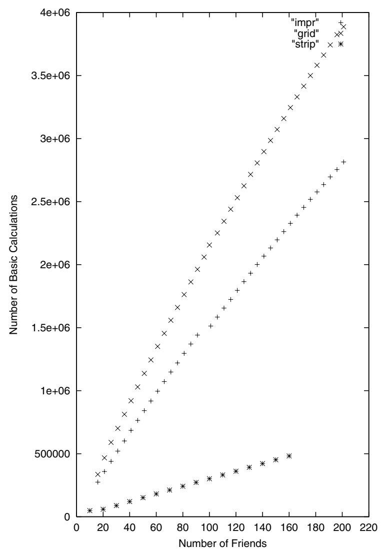

Fig. 9. Total number of basic operations (distance computation) in the three methods

tures for Mobile Data," *Proc. of the eighth Annual ACM-SIAM Symposium on Discrete Algorithms* 1997, 747–756.

[9] M. de Berg, M. van Kreveld, M. H. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 2nd ed., 2000.

[10] G. Brodal and R. Jacob, "Dynamic Planar Convex Hull," *43rd Annual Symposium on Foundations of Computer Science,* 2002, 617-626.

[11] H. D. Chon, D. Agrawal and A. E. Abbadi, "Range and kNN Query Processing for Moving Objects in Grid Model," *Mobile network and Applications,* To Appear.

[12] L. J. Guibas, "Kinetic data structures: a state of the art report," in *Proc. Workshop Algorithmic Found. Robotics*, 1998, 191-29.

[13] D. P. Huttenlocher, K. Kedem, J. M. Kleinberg, "On Dynamic Voronoi Diagrams and the Minimum Hausdorff Distance for Point Sets Under Euclidean Motion in the Plane," in *Symposium on Computational Geometry,* 1992, 110–119.

[14] C. Jermaine, A. Datta, E. Omiecinski, "A Novel Index Supporting High Volume Data Warehouse Insertion," *VLDB 1999* , 235–246.

[15] http://www.profilium.com/about/whatwedo.html.

[16] J. Kaufman, J. Myllymaki, and J. Jackson "City Simulator,", IBM AlphaWorks developer web site. *http://www.alphaworks.ibm.com/tech/citysimulator*, 2001.

[17] G. Pottie and W. Kaiser, "Wireless integrated network sensors," *Comm. of the ACM,* 43, (2000), 51–58.

[18] D. Pfoser and C. S. Jensen, "Querying the trajectories of on-line mobile objects," *Proc. of the ACM Int. Workshop on Data Engineering for Wireless and Mobile Access*, 2001, 66–73.

[19] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao, "Modeling and querying moving objects," in *Proc. of the IEEE Int. Conference on Data Engineering (ICDE)*, 1997, 422–432.

[20] C. M. Procopiuc, P. Agarwal and S. Har-Peled, "STAR-tree: An efficient self-adjusting index for moving points," in *Algorithm Engineering and Experimentation (ALENEX)*, 2002, 178–193.

[21] S. Saltenis and C.S. Jensen, "Indexing of moving objects for location-based services," in *Proc. of the IEEE Int. Conference on Data Engineering (ICDE)*, 2002, 463–472

[22] S. Saltenis, C. S. Jensen, S. T. Leutenegger and M.A.Lopez, "Indexing positions of continuously moving objects," *SIGMOD Conference 2000*, 331-342.

[23] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison-Wesley, Reading, MA, 1990.

[24] H. Samet *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1990.

[25] J. Tayeb, O. Ulusoy, O. Wolfson, "A Quadtree Based Dynamic Attribute Indexing Method," *The Computer Journal* (1997), 185–200.

[26] O. Wolfson, A. P. Sistla, S. Chamberlain and Y. Yesha, "Updating and Querying Databases that Track Mobile units," *Distributed and Parallel Databases,* 1999, 257–387.