Efficient Processing of Reachability Queries with Meetings

Elena V. Strzheletska
University of California, Riverside
Riverside, CA 92521, USA
elenas@cs.ucr.edu

Vassilis J. Tsotras
University of California, Riverside
Riverside 92521, USA
tsotras@cs.ucr.edu

ABSTRACT

The prevalence of location tracking systems has resulted in large volumes of spatiotemporal data generated every day. Addressing reachability queries on such datasets is important for a wide range of applications, such as security monitoring, surveillance, public health, epidemiology, social networks, etc. Given two objects O_S, O_T and a time interval I, a reachability query identifies whether information (or physical items etc.) could have transferred from O_S to O_T during I (typically indirectly through intermediaries). While traditional graph reachability queries have been studied extensively, little work exists on processing spatiotemporal reachability queries for large disk-resident trajectory datasets. Moreover, previous research assumed that information can be passed from one object to another instantaneously. However, in many applications such transfer takes time (i.e., a short conversation), thus forcing interacting objects to stay in contact for some time interval. This requirement makes the query processing even more challenging. In this paper, we introduce a novel problem, namely, *spatiotemporal reachability queries with meetings* and propose two algorithms, RICCmeetMin and RICCmeetMax. To prune the search space during query time, these algorithms precompute some reachability events: the shortest valid meetings (RICCmeetMin), and the longest possible meetings (RICCmeetMax) respectively. An extended experimental evaluation examines the efficiency and pruning characteristics of both algorithms over a variety of spatiotemporal reachability queries with meetings on large disk-resident datasets.

1 INTRODUCTION

Reachability queries are common in various spatiotemporal applications including security monitoring, surveillance, public health, epidemiology, social networks, etc. Consider a set of moving objects $O_1, O_2, ..., O_n$ (people, cars, etc.). Two objects O_i and O_j have a contact at time t, if they are within some threshold distance from each other at that time instant [29]. While being close in space, O_i and O_j may exchange some information (directly or wirelessly), a physical item, a virus, etc. As time proceeds, the location of objects O_i and O_j changes, and each of the earlier 'contacted' objects may get involved in other exchanges later. In this way, the information propagates further through the network, and more objects become carriers. Even though, two objects may had never been in direct contact with each other, information from one object may have reached the other through some intermediate contacts.

For the purposes of this paper, it is assumed that the location of each monitored object is recorded at discrete time instants $t_1, t_2, ..., t_i$, ..., and that the time interval between consecutive location recordings $\Delta t = t_{k+1} - t_k$ $(k = 1, 2, ...)$ is constant. A *trajectory* of a moving object O_i is a sequence of pairs (l_i, t_k), where l_i is the location of object O_i at time t_k. Formally, two objects, O_i and O_j that at time t_k are respectively at positions l_i and l_j, have a *contact*, if $dist(l_i, l_j) \leq d_{cont}$, where d_{cont} is the *contact distance* (a distance threshold given by the application), and $dist(l_i, l_j)$ is the Euclidean distance between the locations of objects O_i and O_j at time t_k. A *contact* between objects O_i and O_j at time t_k is denoted as $< O_i, O_j, t_k >$. Object O_T is considered to be *reachable* from object O_S during interval $I = [t_s, t_f]$ if there exists a chain of subsequent contacts $< O_S, O_{i1}, t_1 >, < O_{i1}, O_{i2}, t_2 >, ..., < O_{im}, O_T, t_k >$, with $t_s \leq t_1 < t_2 ... < t_k \leq t_f$. A reachability query Q: $\{O_S, O_T, I\}$ determines whether object O_T (the target) is reachable from object O_S (the source) during time interval I [30].

Traditional graph reachability is performed on a static graph. It is possible to reduce spatiotemporal reachability into static graph reachability by constructing contact graphs among the objects (one contact graph per time instant) and combining them into a supergraph by introducing an edge between two consecutive occurrences of each object. An example of such a construction is given in Figure 1, where solid edges connect objects that have a contact, and dotted edges connect consecutive object positions.

On a small graph, there are two naive approaches that could be used for answering a reachability query: 'precompute-all' and 'no-preprocessing'. The first approach requires to precompute and store reachability information between every pair of nodes in the graph. The second necessitates traversing the graph during the query time. Even for traditional graph reachability either approach is inefficient if a graph is large, since the first requires too much time and space for preprocessing, while the second has high query time. Spatiotemporal reachability is more complex: the graph is dynamic and object relationships may change every time instant.

Note that the spatiotemporal reachability query definition above does not consider the contact's time duration. Implicitly this assumes that objects may be able to exchange information (or physical item) instantaneously when a contact occurs. This 'instant exchange' assumption was considered in [29]. However, under such conditions, during the same time instant, information can be transferred instantly to all current contacts of an object (and all current contacts of the contacted objects, etc.)

In [30] the 'no instant exchange' reachability scenario is considered (a contacted object can broadcast its information at the next time instant). This scenario fits applications where after a contact

SIGSPATIAL'17, November 7–10, 2017, Los Angeles Area, CA, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5490-5/17/11...$15.00
https://doi.org/10.1145/3139958.3139982

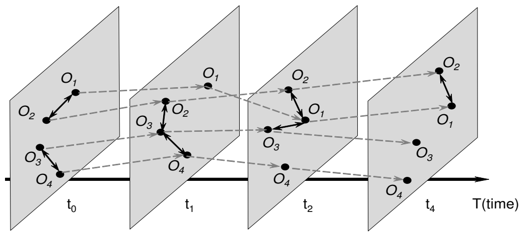


Figure 1: Constructing a supergraph by combining the contact graphs with the object trajectories.

between two objects has occurred, the contacted object may require some *processing delay*, i.e., time to process information before it can start the retransmission (it is easy to extend that approach to support any fixed processing delay).

Depending on the assumed scenario, the answer to the reachability query may be different. Consider Figure 1 and suppose that object O_2 carries some information. According to the 'instant exchange' scenario, at time t_1, object O_2 can transmit this information to O_3, and at the same time instant O_3 can retransmit it to O_4. Assuming the 'no instant exchange' scenario, at time t_1, object O_2 can still transmit information to O_3, however O_3 cannot retransmit it at this time instant. In fact, during the time interval shown in the graph, O_4 will never receive the information.

Nevertheless, for many applications simply having a contact (with or without processing delay) is not enough for exchanging information between two objects as time may be needed for the actual information to be transfered (termed as a *transfer delay* in [30]). To account for such delay, the objects are required to stay within a contact distance for some period of time; in other words, the objects need to have a *meeting*.

In this paper, we propose the first (to the best of our knowledge) solution to the problem of *spatiotemporal reachability with meetings*. As with previous works on spatiotemporal reachability [29, 30], we assume that the queries are issued against a substantial repository of trajectory data, which is too large to fit in main memory during the preprocessing or query processing; hence we seek disk I/O efficient solutions. In particular, we present two algorithms, *RICCMeetMin* and *RICCMeetMax* that consist of preprocessing and query answering stages. For simplicity, in the following description we assume no processing delay; both algorithms can be easily extended to support processing delays.

The rest of the paper is organized as follows: Section 2 discusses related work, while Section 3 defines the reachability with meetings problem. The preprocessing and query processing for the two RICCMeet algorithms appear in Sections 4 and 5, respectively, while their performance is compared in Section 6. Finally, Section 7 presents conclusions and directions for future work.

2 RELATED WORK

Graph Reachability: Static graph reachability has been studied extensively and many solutions exist for querying such graphs. All efficient approaches try to balance the preprocessing with the query processing. A categorization appears in [15] based on the underlying algorithm used, namely: (i) transitive closure compression, (ii) hop labeling, and (iii) refined online search. Approaches in the first category involve computing and compressing a transitive closure, as in interval labeling [1], dual labeling [36], chain decomposition, tree cover, etc. Hop labeling methods include 2-hop cover [7], 3-hop cover [16], and path-top [4], etc. A similar idea is used is [31], it utilizes Bloom filter labeling. The third category includes GRAIL [38], which uses indexing based on randomized multiple interval labeling; Ferrari [28], that computes indexing intervals over an optimal spanning tree; and PReaCH [21], that utilizes topological ordering techniques from GRAIL and applies the Contraction Hierarchies technique [9] to the graph reachability problem. Contraction Hierarchies is a state-of-the-art algorithm for computing shortest paths. During a preprocessing phase, each node in the graph is assigned an order (according to its importance in the graph, thus creating a hierarchy), and then contracted in that order.

We note that the spatiotemporal reachability query under the 'no instant exchange' scenario can be answered as a shortest path query [30]. Unfortunately, one cannot use the Contraction Hierarchies approach since in the spatiotemporal reachability problem, there is no prior knowledge about the graph nodes, and no basis for creating a hierarchy.

Evolving Graphs: Time evolving graphs including social, citation, biological networks, etc., have attracted high research interest recently. The DeltaGraph, an external hierarchical index structure was introduced in [18] for efficient storing and retrieving of historical snapshots. Shortest path queries on evolving social graphs were examined in [12]. Location-aware graph reachability is studied in [27]. Graph reachability labeling methods are presented in [32], and applied for analyzing temporal distance and reachability of temporal graphs. Nevertheless, in such graphs, contacts are typically known, while for our problem they have to be efficiently computed.

Spatiotemporal Databases: Works on spatiotemporal access methods involve some variation on hierarchical trees: [6, 8, 11, 19, 26, 35, 39, 40], or some form of a grid-based structure [25, 37] or indexing in parametric space [2, 5, 23]. A recent survey appears in [22]. Nevertheless, existing spatiotemporal indexes typically support traditional range and nearest neighbor queries and not the reachability queries that we address in this paper.

There is also work that focused on querying/identifying the spatiotemporal behavior of moving objects (discovering moving clusters [13, 17], flock patterns [34], convoy queries [14], flexible patterns [33], density [24], selectivity estimation [10]) .

Spatiotemporal Reachability: The first disk-based solutions (ReachGrid and ReachGraph) for the 'instant exchange' spatiotemporal reachability problem are presented in [29]. In ReachGrid, during query processing only a portion of the contact network which is required for reachability evaluation is constructed and traversed. In ReachGraph, the reachability is precomputed at different scales, and then retrieved at query time. ReachGraph has faster query time (and outperforms traditional graph reachability solutions like GRAIL [38]). However, it takes advantage of the 'instant exchange' assumption which enables ReachGraph to have much smaller size (all objects that belong to the same connected component of a contact graph are replaced with a single vertex), and thus reduced query time. ReachGrid does not require the 'instant exchange' assumption and can be modified to work for the reachability with meetings problem (in the experimental section we compare against

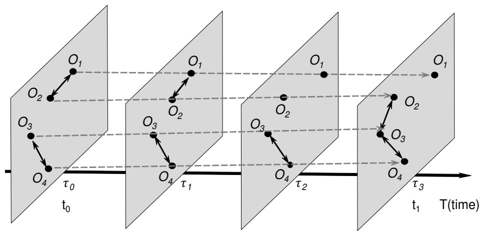


Figure 2: Discovering meetings between the objects on time interval $I = [t_0, t_1]$.

this modified ReachGrid approach). Finally, [30] presents the first solution for the 'no instant exchange' spatiotemporal reachability query by utilizing the *path contraction* idea, introduced in Contraction Hierarchies [9]. Nevertheless, this method cannot be easily modified to solve the reachability with meetings problem.

3 PROBLEM DESCRIPTION

When considering the reachability with meetings problem, it is important to determine when a pair of objects began their meeting, as well as the duration of the meeting (how long the objects stayed within the contact distance). Previous spatiotemporal reachability works [29, 30] assumed that contacts between objects could occur only at the time instant that an object's location is reported. In reality objects can have their initial contacts (and thus start a meeting) during the time between two consecutive reported locations.

To capture the beginning of a meeting as accurate as possible, we discretize the time interval between consecutive position readings $[t_k, t_{k+1})$ by dividing it into a series of r non-overlapping subintervals $[\tau_0, \tau_1)$, ..., $[\tau_i, \tau_{i+1})$... , $[\tau_{r-1}, \tau_r)$ of equal size $\Delta\tau = \tau_{i+1} - \tau_i$, such that $\tau_0 = t_k$ and $\tau_r = t_{k+1}$. Hence $\Delta t = r\Delta\tau$ (where r is some positive integer). Further, we assume that between any two consecutive reported locations each object moves linearly and with constant speed. We can thus calculate an object's approximate position at any time instant τ_i between two consecutive reported locations. We denote the instance of object O_i at time τ_j as $O_i^{(\tau_j)}$.

We proceed with the definition of a meeting. Two objects, O_i and O_j, had a *meeting* during the time interval $I_m = [\tau_s, \tau_f]$, if they had been within the threshold distance d_{cont} from each other at each time instant $\tau_k \in [\tau_s, \tau_f]$. Such a meeting is denoted $< O_i, O_j, I_m >$. The *duration* of this meeting is $m = \tau_f - \tau_s$.

The transfer delay (time to exchange information between two objects) may be different from the actual meeting duration. Hence, some meetings are long enough for an exchange while others are not. We assume that the query specifies the *required meeting duration* m_q which is the time, needed for the objects to complete the exchange (this allows a user to examine different transfer scenarios). A meeting $< O_i, O_j, [\tau_s, \tau_f] >$ between objects O_i and O_j is thus *valid* for the query if its duration satisfies $m = \tau_f - \tau_s \geq m_q$.

Furthermore, if object O_i carried some information, object O_j is considered to be *'reached'* after m_q time units from the beginning of their meeting (and thus is able to start retransmitting this information). Hence the earliest time when object O_j is reached is $\tau_R(O_j) = \tau_s + m_q$.

Consider the example in Figure 2. Suppose, $\Delta t = 3\Delta\tau$, and $m_q = 2\Delta\tau$. At time t_0, two pairs of objects have contacts: $< O_1, O_2, t_0 >$ and $< O_3, O_4, t_0 >$. In order to determine whether any meetings between these pairs occurred, we calculate for how long they had stayed within the contact distance. After the positions of objects O_1, O_2, O_3, and O_4 are determined at τ_1 and τ_2, we find the durations of each meeting as $< O_1, O_2, [\tau_0, \tau_1] >$, and $< O_3, O_4, [\tau_0, \tau_2] >$. The meeting between objects O_1 and O_2 is not valid, since it does not satisfy the required meeting duration condition $m_q = 2\Delta\tau$. Thus the only valid meeting is $< O_3, O_4, [\tau_0, \tau_2] >$. Further, if object O_3 carried some information before the meeting with object O_4, object O_4 becomes reached at time $\tau_R(O_4) = \tau_0 + 2\Delta\tau$.

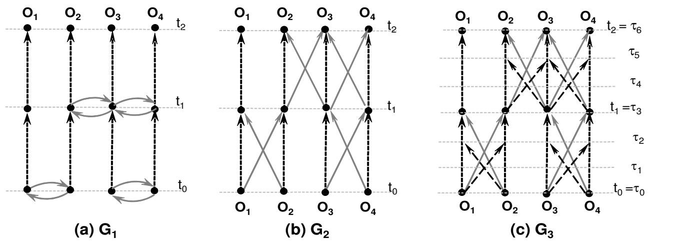


Figure 3: (a) graph G_1 represents the 'instant exchange' scenario; (b) graph G_2 depicts the 'processing delay' scenario with delay $\lambda < \Delta t$; (c) graph G_3 assumes the 'transfer delay' scenario (the time interval is $I = [t_0, t_2]$).

Object O_T is considered to be *(meeting)-reachable* from object O_S during time interval $I = [\tau_s', \tau_f']$ if there exists a chain of subsequent meetings $< O_S, O_{i_1}, I_{m_0} >$, $< O_{i_1}, O_{i_2}, I_{m_1} >$, ... ,$< O_{i_k}, O_T, I_{m_k} >$, where each $I_{m_j} = [\tau_{s_j}, \tau_{f_j}]$ is such that $\tau_{f_j} - \tau_{s_j} \geq m_q$, $\tau_s' \leq \tau_{s_0}$, $\tau_{f_k} \leq \tau_f'$, and $\tau_{s_{j+1}} \geq \tau_{f_j}$ for $j = 0, 1, ..., k - 1$. To specify that object O_T can be reached under the meeting duration m_q, we will say that O_T is (m_q)*−reachable* . Also, the earliest time when O_T can be reached (or the earliest 'reached' time) we will denote as $\tau_R(O_T)$.

A *reachability with meetings* query Q_{meet}: $\{O_S, O_T, I, m_q\}$ checks whether object O_T (target) is (m_q)-reachable from object O_S (source) during time interval $I = [\tau_s, \tau_f]$, and reports the earliest time instant when O_T was reached.

Figure 3 illustrates the difference between the graphs that represent the 'instant exchange', the 'processing delay', and the 'transfer delay' reachability scenarios. The graphs are constructed on the dataset used for Figure 1 for time interval $I = [t_0, t_2]$. In all graphs, edges connecting the same object represent the object's trajectory over time. For the 'instant exchange' case (Figure 3(a)) and the 'processing delay' case (Figure 3(b)) edges connecting *different* objects represent contacts. For the 'processing delay' case we assumed that the duration of the delay is $\lambda < \Delta t$ (as described in [30]). For the 'transfer delay' case (Figure 3(c)) edges between different objects represent possible meetings. Since m_q is query specified, it is unknown at preprocessing time. In the above example, the graph is shown for only two m_q values, namely: $m_q = 2\Delta\tau$ and $m_q = 3\Delta\tau$.

Clearly pre-constructing the meetings graph for all possible m_q values is not practical since it significantly increases the size of the corresponding graph and thus the problem complexity.

4 PREPROCESSING

As with classic graph reachability, there are two extreme approaches to answer a spatiotemporal reachability query with meetings Q_{meet}: $\{O_S, O_T, [\tau_s, \tau_f], m_q\}$. The 'no-preprocessing' approach contains

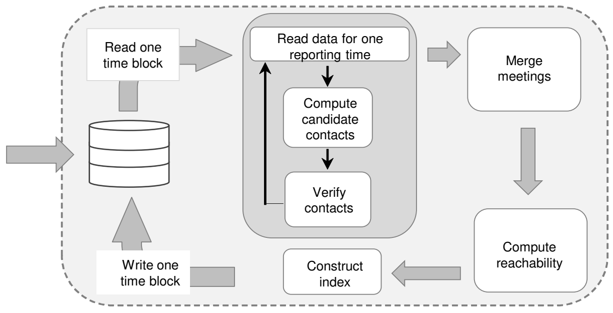


Figure 4: Preprocessing Workflow for RICCmeet algorithms

the following steps: first the distances between O_S and all the other objects O_i at time instant τ_s are computed, and all contacts of O_S are identified; this is repeated for time instants τ_{s+1}, τ_{s+2}, ... If two consecutive contacts between a pair of objects (O_S, O_i) are discovered, they create a meeting. If the meeting between objects O_S and O_i reaches the duration of m_q time units, O_i becomes reached, and is added to the set of reached objects. The process continues until the target object O_T becomes reached or τ_f is processed. Clearly this approach leads to prohibitively slow query time since for every reached object, distances with all other objects need to be computed and recomputed for every following time instant.

Instead, 'precompute-all' calculates the reachability between every pair of objects for every possible time interval and value of m_q, which results in prohibitive preprocessing time and space.

To enable fast query processing while maintaining a reasonable preprocesssing, we balance the two extreme approaches by precomputing only some information. We proceed with the description of the two proposed algorithms, namely RICCmeetMin and RICCmeetMax. In Section 6 we compare them with a baseline (ReachGridmeet), which is a modified version of ReachGrid [29] adapted to answer the reachability with meetings problem. All three algorithms include preprocessing that efficiently computes all object contacts. In addition, for the RICCmeet algorithms, we precompute all meetings, as well as the reachability between the objects for specific m_q values on short time intervals.

We assume that the dataset is organized in records of the form: $(t, object\_id, location)$, ordered by the location reporting time t. As with [29] to take advantage of temporal locality (since meetings involve trajectory locations of nearby times), the time domain is divided into a non-overlapping subintervals, or *time blocks*. Each time block (denoted as B_k) contains the records with reporting times in the corresponding time period. The number of time instants that are combined into one time block is the *contraction parameter C*; we discuss how to tune the value of C in Section 6.

For each time block, the preprocessing of each RICCmeet algorithm completes the following four steps: (i) candidate contact computation and contact verification (performed for each time instant), (ii) meetings identification, (iii) reachability precomputation, and (iv) index construction. Based on the contacts within this time block, a meetings graph is constructed that contains all meetings during this time block. Further, each algorithm pre-constructs a reachability graph; the two algorithms differ on how these reachability graphs are created. The workflow of the preprocessing stage of the RICCmeet algorithms is shown in Figure 4.

Table 1: Notation used in the paper

Notation	Definition
$\Delta\tau$	Duration between two consecutive time instants
Δt	Duration between two consecutive reporting times
O_S, O_T	A source and a target objects
$O_i^{(\tau_j)}$	Instance of object O_i at time τ_j
d_{cont}, d_{cc}	Contact distance, candidate contact distance
m_q	Required meeting duration (query specified)
μ	Minimum meeting duration
B_k, I_k	Time block k that spans time interval I_k.
C	Contraction parameter
H	Grid resolution
$\tau_R(O_i)$	Earliest time when object O_i was reached

We take advantage of spatial locality by partitioning the area into cells with side H (the *grid resolution*) - a parameter, whose tuning is discussed in Section 6. In computing contacts (as discussed below), we follow the movements of objects and their relative positions during the time period between two consecutive readings Δt. To capture this finer spatial locality, we further partition each cell with side H into many smaller cells with side d_{cc}; here d_{cc} depends on the maximum distance traveled by any object within Δt.

During preprocessing, for each object O_i we maintain important information in a data structure named *objectRecord(O_i)*. In particular, an *objectRecord* has the following fields: *Object_id*, *Cell_id* (the object's placement in the grid with side H), *ContactsRec* (a list that will maintain the contacts for the given object), *MeetingsRec* (a list that will store the meetings for the given object). At the beginning of each time block, we start with an empty *objectRecord* for each object O_i, and update it as the preprocessing proceeds. The Cell_id field is filled using the coarse cell (side H) that contains O_i's location during its first appearance in the time block. This Cell_id will not be changed even if the object moves to another coarse cell during this time block (with a large enough H this object will remain in its original coarse cell, or nearby ones, still capturing spatial locality). Finally, for each time block we maintain a hashing scheme, that allows fast access to each *objectRecord(O_i)* by O_i.

4.1 Computing Contacts

Let d_{max} denote the largest distance that can be covered by any object during Δt. Two objects O_i and O_j are *candidate contacts* at reporting time t_k if they are within distance $d_{cc} = 2d_{max} + d_{cont}$ (termed as *candidate contact distance*) from each other at that time instant. Effectively such objects can potentially have a contact between t_k and t_{k+1}. We thus assign all objects reported at time t_k into cells with side d_{cc}. Due to the size of this finer partition, candidate contacts can only appear in the same or neighboring cells. Hence we need only to compute the (Euclidean) distance between all pairs of objects that are in the same or the neighboring cells which greatly reduces computation.

When the object locations are read at the next reporting time t_{k+1}, we can verify for every pair of candidate contacts whether a contact indeed occurred at some time instant $\tau_i \in [t_k, t_{k+1})$ (using our assumption that between consecutive reporting times objects move linearly). For every object O_i, when a contact with O_j at time

τ is verified, it is appended as a contact record (τ, O_j), in the list *ContactsRec* of *objectRecord*(O_i) (such records are ordered first by contact time and then by the contact's object id). This contact will also be appended in the *ContactsRec* list of *objectRecord*(O_j).

4.2 Identifying Meetings

While each object updates its contacts in list *ContactsRec* we can start creating meetings. When considering O_i, if an object O_j was a contact at two consecutive time instants, these contacts are merged into a meeting. As meetings for object O_i are found, they are written as meeting records in the *MeetingsRec* list of *objectRecord*(O_i). Each meeting record consists of the meeting *companion* (say O_j) as well as the beginning time and the end time of the meeting. If the same companion appears consecutively, the meeting duration is extended. This process continues until we process the time block at which point the meeting durations are computed.

Our preprocessing does not assume the knowledge of the (query specified) required meeting duration m_q. Instead, we assume that there is a minimum time duration μ required by any transfer; that is, $\forall m_q, m_q \geq \mu$. As a result, any meeting with duration less than μ can be pruned. Note that meetings that start at the beginning of the time block and have duration less than μ during this block, need special attention since they may have started in the previous time block and thus qualify as valid meetings. Similarly meetings that are active at the last time instant of the time block but with duration less than μ, can still be valid because they may extend into the next time block. Such 'boundary' meetings are recorded as valid regardless of their length (and verified during query processing).

At the end of the current time block all meetings are persisted in file *Meetings*. During this step *objectRecords* are accessed in H cell order (so as to maintain spatial locality); within a cell they are thus ordered by object id, beginning meeting time, and companion id if meeting intervals are the same for two contacted objects.

4.3 Identifying Reached Objects

Let's assume for the time being that the value of m_q is known. To speed-up the query time, during the preprocessing for each block B_k, we can find and record for every object O_i all objects O_j, that are (m_q)-reachable from O_i during B_k. A naive solution would compute (m_q)-reachability for every directed pair (O_i, O_j) which leads to computing $O(n^2)$ (m_q)-reachability calculations (n is the number of objects). Instead we propose an algorithm that requires $O(n)$ (m_q)-reachability calculations .

We can solve our problem as a traditional reachability problem on a static graph, where computing reachability for an object is equivalent to finding a path on the graph. Let's assume that we were to construct such a static reachability graph. We could start with constructing a *meetings graph* G_k^M for each time block B_k. Given the Meetings file, the meetings graph G_k^M for time block B_k can be created as follows: for each meeting $< O_i, O_j, [\tau_s, \tau_f] >$ we introduce vertices (if they are not already created): $O_i^{(\tau_s)}$, $O_i^{(\tau_f)}$, $O_j^{(\tau_s)}$, $O_j^{(\tau_f)}$. We also introduce edges that connect two consecutive occurrences of the same object (e.g., connecting $O_i^{(\tau_s)}$ with $O_i^{(\tau_f)}$), and *meeting* edges that indicate the possible transfer of information during this meeting. Hence, for the above meeting we create two meeting edges: ($O_i^{(\tau_s)}$ to $O_j^{(\tau_f)}$) and ($O_j^{(\tau_s)}$ to $O_i^{(\tau_f)}$). All edges are directed (from smaller to larger time instants). Figure 6(a) shows the meetings graph for the dataset in Figure 5.

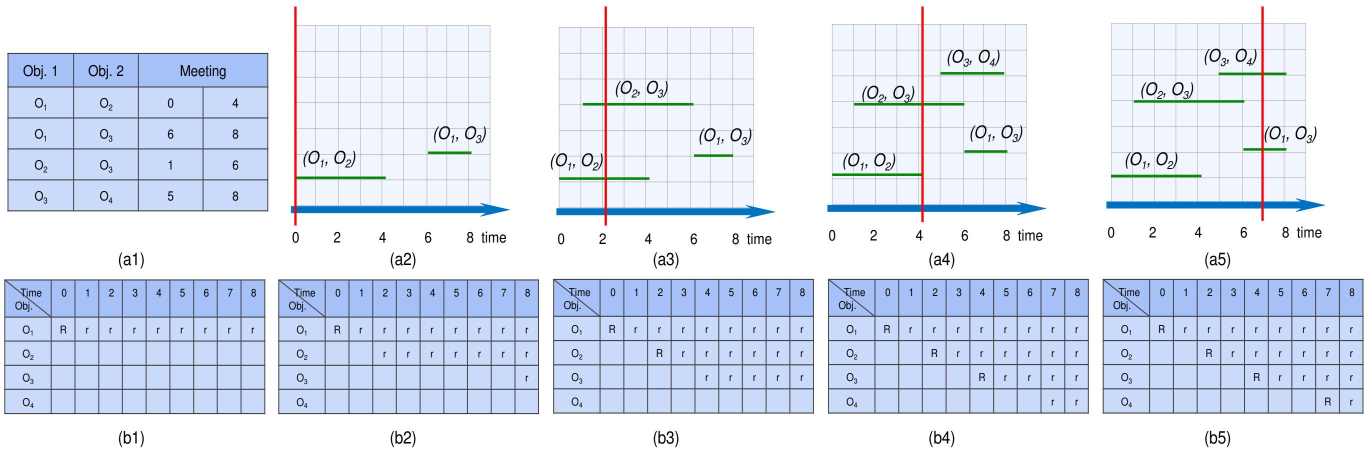
To turn graph G_k^M into a reachability graph $G_k^R(m_q)$, for each meeting $< O_i, O_j, [\tau_s, \tau_f] >$ we do the following: (1) if $\tau_f - \tau_s < m_q$ we remove a pair of 'meeting' edges; (2) if $\tau_f - \tau_s > m_q$ we introduce a vertex for each instance of objects O_i and O_j during each time instant of the interval (τ_s, τ_f), and replace a pair of meeting edges between objects O_i and O_j with a set of pairs of meeting edges that start at the instances of O_i and O_j at each time instant of the interval $[\tau_s, \tau_{f-m_q}]$ and correspond to meetings of duration m_q. The last modification is needed to account for the fact that a transfer of information does not necessarily start at the beginning of a meeting, and that the objects are required to be companions for at least m_q time units after the transfer starts.

In the G_k^R graph, an object O_j is (m_q)-reachable by O_i if and only if it belongs to some path that starts from a vertex that represents the first instance of O_i during block B_k. To efficiently discover all such paths, we can combine a Depth-First Search (DFS) and a plane-sweep algorithm. Our algorithm proposes the following strategy for the G_k^R graph traversal. We start by visiting the earliest instance of object O_i in G_k^R, move to the next available instance of O_i, and continue in DFS manner until the last instance of O_i is visited. While visiting a vertex, we explore all outgoing meeting edges from this vertex. These meeting edges point to the objects, reached by O_i. We record the earliest instance of each reached object that was discovered during the traversal of O_i into a *priority queue* S_{PQ} (giving priority to the objects that were reached earlier). After the last instance of O_i is visited, the search backtracks to the vertex that represents the instance of an object, which is at the top of the priority queue. This process continues until the last vertex from S_{PQ} is extracted.

Note, that the above discussion serves as a sketch of proof for the correctness of our algorithm; we do not actually need to construct the G_M^k and G_R^k graphs. The proposed algorithm emulates the same strategy described above (DFS and plane sweep) by visiting the *objectRecords* (and the meetings stored within such records).

The reachability status of each object is recorded into a temporary *reachability table*, which is created once per time block, and is being updated as time proceeds. This table adds a row when an object is reached and has one column per time instant of the time block. Consider example in Figure 5. For simplicity, Table (a_1) shows the actual meetings between all objects during one time block. Tables (b1) - (b5) show how the reachability table evolves over time; here $'R'$ stands for the earliest time when an object was reached, and $'r'$ - for each subsequent time instant. For this example, we set $m_q = 2\Delta\tau$, while the time block's interval is $9\Delta\tau$.

The figure shows how to find all objects reached by object O_1. At τ_0 only O_1 is reached (b1). During the given time block, O_1 had meetings with objects O_2 and O_3, which can result in them being reached by times $\tau_R(O_2) = 2$ and $\tau_R(O_3) = 8$ (in (a2, b2)). (Once a meeting $< O_i, O_j, [\tau_s, \tau_f] >$ is discovered, it is represented as a line segment with endpoints at τ_s and τ_f on the plane.) To decide which object to visit next, the plane is swept with a line in increasing time order, starting from $\tau = 0$. We move to O_2 - the object with



(a1)

Obj. 1	Obj. 2	Meeting	
O1	O2	0	4
O1	O3	6	8
O2	O3	1	6
O3	O4	5	8

(b1)

Time / Obj.	0	1	2	3	4	5	6	7	8
O1	R	r	r	r	r	r	r	r	r
O2									
O3									
O4									

(b2)

Time / Obj.	0	1	2	3	4	5	6	7	8
O1	R	r	r	r	r	r	r	r	r
O2			r	r	r	r	r	r	r
O3									r
O4									

(b3)

Time / Obj.	0	1	2	3	4	5	6	7	8
O1	R	r	r	r	r	r	r	r	r
O2			R	r	r	r	r	r	r
O3					r	r	r	r	r
O4									

(b4)

Time / Obj.	0	1	2	3	4	5	6	7	8
O1	R	r	r	r	r	r	r	r	r
O2			R	r	r	r	r	r	r
O3					R	r	r	r	r
O4								r	r

(b5)

Time / Obj.	0	1	2	3	4	5	6	7	8
O1	R	r	r	r	r	r	r	r	r
O2			R	r	r	r	r	r	r
O3					R	r	r	r	r
O4								R	r

Figure 5: Computing the (m_q)-reachable objects from O_i $(m_q = 2)$.

the earliest reached time, and check all meetings of O_2 that end after $\tau = 2$. Consider meeting $< O_2, O_3, [1, 6] >$ (a3). Even though it starts at $\tau = 1$, object O_2 itself was not reached until $\tau = 2$, and only at this time it may start retransmission. Thus O_3 can be reached at $\tau = 4$ (earlier than it was reached by object O_1), and we can update information in table (b4). Due to this update, O_3 now has enough time to reach O_4(a4), which leads to $\tau_R(O_4) = 7$ (a5, b5).

The procedure for computing all objects that are (m_q)-reachable by O_S is generalized in Algorithm 1. The $S_{Reached}$ set keeps all objects for which the earliest reached time has been finalized. The algorithm maintains a priority queue S_{PQ}, which contains reached objects that are not in $S_{Reached}$ yet; objects in S_{PQ} are prioritized according to their 'reached' times. After object O_i with the earliest 'reached' time is extracted from S_{PQ}, the procedure finds all companions O_j of O_i, that are not in $S_{Reached}$, and for each O_j it explores every meeting $< O_i, O_j, [\tau_s, \tau_f] >$ from the time $\tau_R(O_i)$, and until either O_j is reached (in which case $\tau_{Rnew}(O_j)$ is updated), or the last time instant of the block is processed. Next, O_j needs to be inserted into S_{PQ}. If O_j was previously found reached by some other object (at time $\tau_R(O_j)$), and is already in S_{PQ}, $\tau_{Rnew}(O_j)$ has to be compared with $\tau_R(O_j)$, and the priority of O_j in S_{PQ} may need to be updated. To precompute reachability during B_k for all objects, Algorithm Reach(m_q) has to be repeated for each object O_i.

Algorithm 1 Reach(m_q)

```
1:  Input: $O_S$
2:  for each $O_i$ do $\tau_R(O_i) = \infty$
3:  procedure ReachFixedM($O_S$, $m_q$)
4:      $time = 0$, $\tau_R(O_S) = 0$, $S_{PQ} = \{O_S\}$, $S_{Reached} = \{\emptyset\}$
5:      while (($S_{PQ}$) $\neq \{\emptyset\}$ $and\ time \leq \tau_{end}$) do        ▷ $\tau_{end}$ is the last
6:          $O_i = ExtractMin\ (S_{PQ})$                                ▷ time unit of a block
7:          $S_{Reached} = S_{Reached} \cup O_i$, $time = \tau_R\ (O_i)$
8:          for each companion $O_j$ of $O_i$ do
9:              if $O_j \notin S_{Reached}$ then
10:                 $\tau_{Rnew}(O_j) = \infty$
11:                 while $\tau_{Rnew}(O_j) \geq \tau_R(O_j)$ do
12:                     read next meeting $M_{ij} =< O_i, O_j, [\tau_s, \tau_f] >$
13:                     compute $\tau_{Rnew}(O_j)$
14:                     if $\tau_{Rnew}(O_j) < \tau_R(O_j)$ then
15:                         $Update\ (S_{PQ}, O_j)$
16:                     if ($M_{ij} = last\ meeting < O_i, O_j, I_{B_k} >$) then
17:                         $\tau_{Rnew}(O_j) = -1$
18: return $S_{Reached}$
```

We proceed with the description of our algorithms RICCmeetMin and RICCmeetMax.

RICCmeetMin. For simplicity the previous discussion assumed that m_q is known. However, m_q is query-specified, and thus unknown at the time of the preprocessing. Recall that the minimum meeting duration μ is the minimum time that is required to complete any transfer, and $\mu \leq m_q$. Let $S_{Reached}(m_q)$ denote the set of objects that are (m_q)-reachable from object O_S. Then $S_{Reached}(m_q) \subseteq S_{Reached}(\mu)$. If O_i is not (μ)-reachable from O_i, it is not (m_q)-reachable as well, which leads us to RICCmeetMin. We assume for the preprocessing that the required meeting duration is μ, and precompute $S_{Reached}(\mu)$ for each object O_i. (During query processing, all objects that are (m_q)-reachable from some object O_i will be among the objects that are found to be (μ)-reachable). Algorithm 1, described above computes $S_{reached}$ for any m_q, including $m_q = \mu$, and can be used without any modifications for RICCmeetMin.

RICCmeetMax. Consider again example in Figure 5 (a). If $m_q = 2$, object O_1 can reach objects O_2, O_3, and O_4. However if $m_q = 3$, O_2

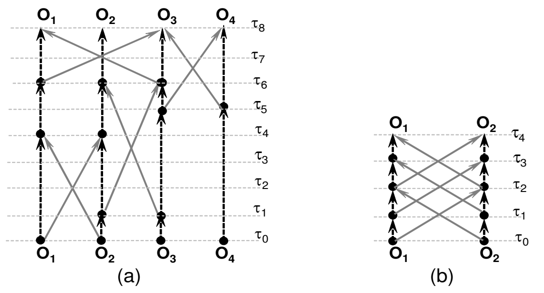


Figure 6: Meetings and reachability graphs construction: (a) Meetings graph G^M; (b) Reachability graph $G^R(\mu)$ for meeting $< O_1, O_2, [\tau_0, \tau_4] >$.

and O_3 are still reachable by O_1, while O_4 is not. Finally, if $m_q = 4$, only O_2 remains reachable by O_1. In real datasets, meeting duration can vary significantly, depending on the direction and speed of the moving objects. Thus, RICCmeetMax precomputes the (m_{max})-reachability for each pair of objects; in other words, for each pair of objects O_i and O_j, it finds the meeting duration m_{max}, such that O_j is (m_{max})-reachable from O_i, but is not $(m_{max}+1)$-reachable.

Algorithm 2 ReachMax

```
1: Input: $O_S$, $S_{Reached}(\mu)$    ▷ $S_{Reached}(\mu)$ is the result of $Reach(\mu)$
2: for each $O_i \in S_{Reached}(\mu)$ do $\tau_R(O_i) = \infty$
3: $m = \mu$
4: while $S_{Reached}(m) \neq \{\emptyset\}$ do
5:     $m = m + 1$
6:     Reach($O_S$, $m$, $S_{Reached}(m-1)$)
7:     Update $S^{max}_{Reached}$
8:     for each $O_i \in S_{Reached}(m)$ do $\tau_R(O_i) = \infty$
9: return $S^{max}_{Reached}$
```

The process of computing (m_{max})-reachability can become time and resource consuming. A straightforward way would be to find, for each object O_i and each m_q, all paths in the reachability graph $G^R_k(m_q)$, from O_i to all the other objects, and determine those that afford the longest meeting duration. We can design a more efficient algorithm by using procedure *ReachFixedM* from Algorithm 1. Reach(μ) explores and prunes a number of meetings that do not result in reachability, and $S_{Reached}(\mu)$ is a small subset of visited objects. It is clear that $S_{Reached}(m) \subseteq S_{Reached}(\mu)$ if $m \geq \mu$. We modify procedure ReachFixedM (and call a new procedure *Reach*) by replacing the condition in line 9 with the following: *if ($O_j \in S_{Reached}(m-1)$ and $O_j \notin S_{Reached}(m)$)*. Here $S_{Reached}(m-1)$ is the set of objects, that were reached by object O_S during the previous iteration. Algorithm 2 summarizes the steps. The initialization takes place in lines 2,3. In line 4, ReachMax checks whether the set of objects that can be reached under the current meeting duration is not empty. The algorithm iterates through steps in lines 5 - 8 by increasing the meeting duration, testing which objects can still be reached by O_S under the new m, and updating their 'reached' times. This process terminates when $S_{Reached}(m)$ is empty. The output of the algorithm is a set of tuples (O_i, m_{max}), where the object, reached by O_S is followed by the longest meeting duration.

Once the reachability for each object of the given time block is computed, the reachability records are written (sequentially) into the file Reached(Min) (respectively Reached(Max)). Each record in file Reached(Min) consists of object O_i itself, and a list of all objects that are (μ)-reachable from O_i. A record in file Reached(Max) consists of the object O_i followed by the list of tuples of the form (O_j, m_{max}). Reachability records are written to the Reached file in the same order as in Meetings file, thus they maintain the same cell order. Within a cell they are ordered by object id.

4.4 Index Construction

In addition to the Meetings and Reached(Min) (or Reached(Max)) files, we create three index structures: the *Meetings Index*, *Reached Index*, and *Time Block Index* (Figure 7). Records in the *Meetings Index*

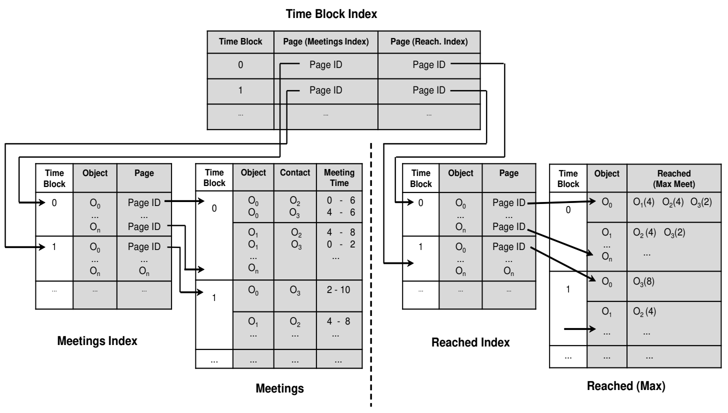


Figure 7: Two-level index on files Meetings and Reached(Max).

are clustered by time block. Each record consists of an object id and a pointer to the page with the first record for this object (for the given time block) in file *Meetings*. Similarly, in the *Reached Index*, each record has an object id and a pointer to the page with the first record for this object (for the given time block) in file *Reached*. Finally, each record in the *Time Block Index* points to the beginning of a time block in each of the other two index files.

5 QUERY PROCESSING

The query processing step is the same for both RICCmeet algorithms. To start processing query Q_{meet}: $\{O_S, O_T, [\tau_s, \tau_f], m_q\}$, we compute which time blocks B_s, ... ,B_f contain data for the time interval $[\tau_s, \tau_f]$. Next, from the *Time Block Index* (which needs to be accessed only once per query) we find what pages in the *Meetings Index* and *Reached Index* correspond to the required blocks.

In file *Reached*, we access the record for O_S during B_s, and find all objects that are reachable from O_S. Note that the set of reached objects may differ, depending on the used algorithm. RICCmeetMin collects all objects that are (μ)-reachable by the given object. Hence, every object O_i that is shown to be reached by O_S in file *Reached(Min)*, is added to the set of reached objects $S'_{Reached}$. RICCmeetMax records in *Reached(Max)* both, a companion, and the value m_{max}. If the longest meeting between O_S and O_i, $m_{max} < m_q$, then O_i is not (m_q)-reachable from O_S, and thus not added to $S'_{Reached}$. Reached objects are saved in $S'_{Reached}$ with the block number, during which each object was reached. This allows to read data efficiently from the file Meetings. After the processing for B_s is finished, we proceed to the next block in *Reached* with the updated set $S'_{Reached}$, and continue until either object O_T is added to $S'_{Reached}$ (say during the block B_i), or B_f is processed.

If O_T was not discovered by the end of B_f in *Reached*, the query terminates, as O_T cannot be reached. Otherwise, it moves to the block B_s of file *Meetings*, where the process of discovering of reached objects for each time block is similar to the one described in Algorithm 1. While crossing the boundary between two consecutive time blocks, special attention is given to the boundary meetings. A meeting between a reached object O_i and its companion O_j that ends at the end of the time block is considered to be incomplete until we start processing the next block. If there is a meeting between O_i and O_j that starts at the beginning of the following block, we merge the two boundary meetings into one new meeting.

If O_T was not confirmed to be reached by the end of B_i, and $B_i \neq B_k$, the search will move again to file *Reached*. This process continues until O_T is confirmed to be reached by the information received from *Meetings*, or the last block B_f is processed.

6 EXPERIMENTAL EVALUATION

We evaluate and analyze the performance of each of the proposed RICCmeet algorithms, and compare it with ReachGridmeet, a modification of the ReachGrid algorithm [29] that works under the 'no instant transfer' assumption. All experiments are performed on a system running Linux with a 3.4GHz Intel CPU with 16 GB RAM, 3TB disk and 4K page size. For all experiments, we set $\Delta\tau$=1 sec.

6.1 Datasets

The performance of both of our algorithms was tested on six datasets of two types: Moving Vehicles (MV) and Random Walk (RW). The MV datasets were created by the Brinkhoff data generator [3], which generates traces of objects, moving on real road networks. The underlying network is the San Francisco Bay road network, which covers an area of about 30000 km^2. These sets contain 1000, 2000, and 4000 vehicles respectively (denoted as MV_1, MV_2, and MV_4). Each vehicle's location is recorded every $\Delta t = 5$ seconds during 4 months (2, 040, 000 records for each object total). We assume $d_{cont} = 100$ meters (for a Bluetooth connection).

The RW datasets, were created with our own data generator, which utilizes the modified random waypoint model [20], often used for modeling movements of mobile users. According to this model, each user chooses the direction, speed (in our case, between $1.5m/s$ and $4m/s$), and duration of the next trip, then completes it, after which chooses the parameters for the next trip, and so on. In our settings, at each time instant, only 90% of individuals are moving, while the remaining 10% are stationary. These three sets simulate the movements of 10000, 20000, and 40000 people respectively (denoted as RW_1, RW_2, and RW_4). The location of each individual is recorded every $\Delta t = 6$sec for a period of one month (or 432,000 records for each person total), and each set covers an area of 100 km^2. For these sets, we assume $d_{cont} = 3$ meters (typical for individuals to pass a physical item or virus).

The performance was evaluated in terms of disk accesses (I/Os) during query processing. The ratio of a sequential I/O to a random I/O is system dependent; for our experiments this ratio is 20:1 (hence 20 sequential I/Os take the same time as 1 random). Using this ratio we present the equivalent number of random I/Os.

6.2 Parameter Optimization

To tune parameters C, H, we use a 5% subset of the dataset. We preprocess this subset for various values of (C, H), and test the performance of the algorithms on a set of 300 queries. (The length of each query was picked uniformly at random between 500 and 4000 sec.) The parameters were varied as follows: grid resolution - from 500 to 40000 meters for MV datasets, and from 250 to 2000 meters for RW; contraction parameter - from 1 to 140 min. For each dataset, we identified the pair of parameters that minimizes the

Table 2: Size of datasets, auxiliary files and indexes

Dataset	Size of Dataset (GB)	Auxiliary Files and Index Size (GB)	
		RICCmeet Min	RICCmeet Max
MV_1	54	4.6	5.2
MV_2	107	23.0	27.3
MV_4	213	83.3	98
RW_1	97	11.6	12.7
RW_2	194	44.9	50.0
RW_4	387	157	178.7

number of I/Os and used them for the rest of the experiments. For example, for MV_1 we use: $H = 20000$ meters, and $C = 10$ min.

6.3 Preprocessing Space and Time

The sizes of the auxiliary files (Meetings and Reached) as well as the index sizes for the two algorithms appear in Table 2. As expected RICCmeetMax uses more space because it stores the actual meeting duration m_{max} per each reached object. Further, in our experiments RICCmeetMax typically takes about 20% more time than for RICCmeetMin (since the algorithm continues until it finds m_{max}). The time needed to preprocess one hour of data for RICCmeetMin ranges from 13 sec for MV_1 to 56 min for RW_4.

6.4 Query Answering

The performance of RICCmeet algorithms was tested on sets of 100 queries of different time intervals ranging from 500 sec to 6.7 hours and different m_q varying from 2 to 16 sec, while μ was set to 2 sec.

RICCmeet vs. ReachGridmeet (Shortest Queries). We start with a brief description of **ReachGrid** algorithm [29]: ReachGrid partitions the dataset into spatial grid cells and time blocks. Each record (which consists of object id, its location and time) is assigned to a cell according to the location of the object. Data of each block is being sorted, first according to object ids, then by time. Finally an index is constructed which for each object, at each time instant, records the cell id to which the object belongs. Within each time block, for each cell the page id where the records for this cell start is recorded as well. In ReachGrid, all relationships (contacts) between the objects have to be discovered at the query time. To speed up query time, in ReachGridmeet, we precompute all the contacts between the objects during preprocessing, while leaving the index structure the same as in ReachGrid. For computing contacts, we use the same algorithm as for both RICCmeet algorithms. After all contacts are discovered, they are recorded in the same order as the data was recorded for ReachGrid. During query processing in ReachGridmeet, at each time instant, after new contacts are discovered, they have to be merged with the previous contacts or meetings into new meetings; lastly, the reachability is checked the same way as in the Algorithm 1.

We evaluate the query performance of the three algorithms while varying m_q on *short* queries (the query interval was set to 500 sec). Figure 8(a) shows the query performance when using the MV_1 dataset and varying m_q from 2 to 16 sec. (In all figures, the *Number of I/Os* reflects the number of random pages accessed per query.) RICCmeetMin and RICCmeetMax access the same number of pages for $m_q = 2$ sec, while RICCmeetMax performs best for the remaining m_q; in comparison, ReachGridmeet accesses about 3.5

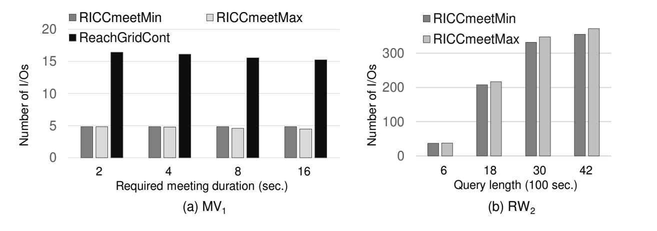


Figure 8: Query performance evaluation: (a) RICCmeet vs. ReachGridmeet, (b) Minimum meeting duration queries

times more pages than RICCmeetMin. In clock time, the RICCmeet algorithms answered these queries in under 1 sec, while it took 80 sec for ReachGridmeet. The query processing of ReachGridmeet is much slower because the algorithm needs to compute meetings and every reachability event during query processing. This was observed consistently in all of our experiments hence its performance is eliminated for the remaining figures.

Minimum Meeting Duration Queries. In this experiment, we compared the query processing of the two RICCmeet algorithms on queries with $m_q = \mu$ (μ = 2 sec.). On each dataset, we ran a set of 100 queries varying query time interval (from 500 to 3500 sec for MV datasets and from 600 to 4200 sec for RW datasets respectively), and learned that in each case either RICCmeetMin outperformed RICCmeetMax, or both algorithms accessed the same number of pages. The greatest difference between the two algorithms' performances (up to 4.8%) was observed for RW_2 dataset, which we presented in Figure 8(b). This result was expected: both RICCmeet algorithms precompute all μ-reachability events, while for RICCmeetMin the size of the auxiliary files is smaller, and thus less data needs to be traversed during the query processing.

Varying m_q. To analyze the impact of m_q on the performance of RICCmeet algorithms, we ran a set of 100 queries varying m_q from 2 to 16 sec; each query's interval was picked uniformly at random from 500 to 3500 sec for MV datasets, and from 600 to 4200 sec for RW datasets. The results are presented in Figure 9 ($a1 - b3$). It is clear that RICCmeetMax outperforms RICCmeetMin in all tests when $m_q > \mu$. As mentioned earlier, during the query processing, we first read file Reached, and may not need to access file Meetings if, according to Reached, the target object is not reached by the end of the query interval. We say that a query was **pruned** if file Meetings has not been accessed during the query processing. Recall that a Q_{meet} query checks whether object O_T is reachable from object O_S. If the answer is positive, we will call such query an *R-query* (for "reached"), and $\bar{R}$-*query* otherwise. The ratio of the number of pruned queries to the number of $\bar{R}$-queries defines the effectiveness of pruning and depends on m_q (see Figure 9(c1, c2)). As m_q increases, the ratio of queries pruned by RICCmeetMax increases from 0.82 to 0.96 while the corresponding ratio of queries pruned by RICCmeetMin decreases from 0.82 to 0.59. Since RICCmeetMin precomputes only (μ)-reachability, it does not have the pruning ability of RICCmeetMax (which has the greatest advantage when answering reachability queries with the longest m_q).

Varying Query Length. Next, we compare the performance of RICCmeet algorithms while varying query interval length. Each test was ran on a set of 100 queries varying query length from 500 to 3500 sec for MV datasets, and from 600 to 4200 sec for RW datasets, while m_q was picked uniformly at random from 2 to 16 sec. The results are shown in Figure 10. While both algorithms show almost linear increase in the number of I/Os with the increase of the query length (a benefit of spatial organization of data in the files), RICCmeetMax is superior to RICCmeetMin in all the tests with the maximum advantage achieved for the longest queries.

Scaling. We tested the effect of scaling the query interval length on the performance of RICCmeetMax. (Since RICCmeetMin performs worse on all but (μ)-reachability queries, we did not include it into the remaining tests.) For this experiment, we used RW_1 since, compared to all the other datasets, the average time needed for two objects in RW_1 to reach each other is the longest. We started with queries that are 3000 sec long, and extended the query length up to 24000 sec also varying m_q from 2 to 16 sec. Figure 11 (a) presents the results. With the increase in query interval, there are more meetings, and thus less pruning. The slowest queries were those with m_q = 16 sec, which still showed a reasonable number of I/Os.

Other Reachability-Based Queries. Until now, we discussed only one-to-one queries: queries that have one source and one target objects. Our algorithms are also efficient in answering other types of queries: one-to-many, many-to-one, and many-to-many. We give a definition of the last type. Let $S_{Source} = \{O_{S_1}, O_{S_2}, ..., O_{S_l}\}$, and $S_{Target} = \{O_{T_1}, O_{T_2}, ..., O_{T_m}\}$ be sets of the source and target objects respectively. A *many-to-many reachability with meetings* query Q'_{meet}: $\{S_{Source}, S_{Target}, I, m_q\}$ determines whether there

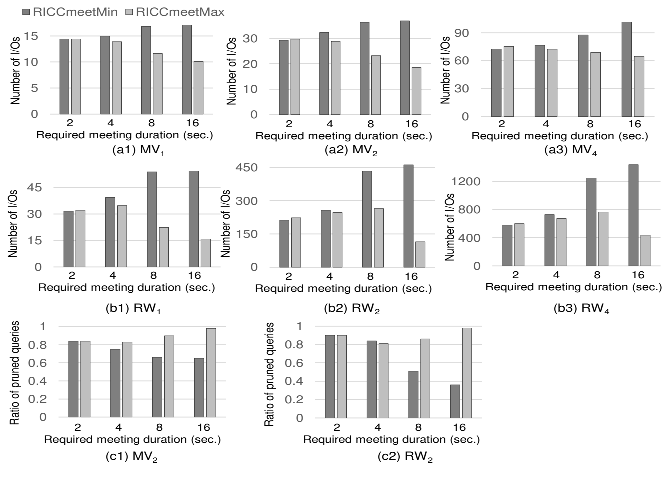


Figure 9: Varying m_q

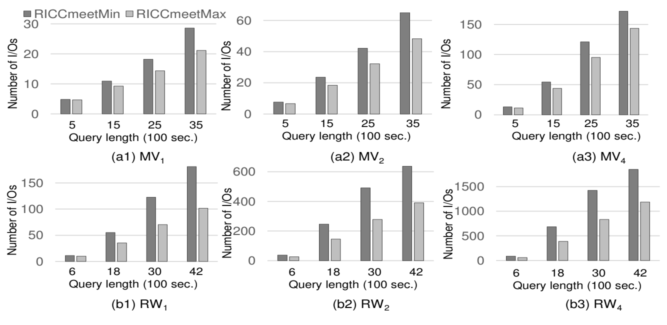


Figure 10: Varying query length

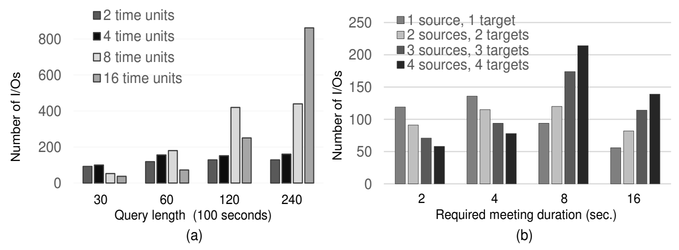


Figure 11: (a) Scaling, (b) Many-to-many queries: dataset RW_1, query length 4200 sec.

is an object $O_{T_j} \in S_{Target}$, such that: i) O_{T_j} is (m_q)-reachable by $O_{S_i} \in S_{Source}$ during time interval $I = [t_s, t_f]$, and ii) if there is more than one reached object, it reports the object with the earliest reached time. One can answer a Q'_{meet} query by running all possible queries $\{O_{S_i}, O_{T_j}, I, m_q\}$ one-by-one, which would lead to a long query processing time. RICCmeet algorithms are efficient in answering Q'_{meet} as one query. For this experiment, we chose RW_1 dataset for the same reason as above. Figure 11(b) shows how performance of RICCmeetMax varies with the increase in the number of source and target objects for Q'_{meet} queries with different m_q (when running Q'_{meet} as one query). The results varied depending on the query lengths, with the most interesting being for the longest tested queries of 4200 sec. Most of the queries with m_q = 2 sec were R-queries , while most of queries with m_q = 16 sec were $\bar{R}$-queries (as one-to-one reachability queries). Among R-queries, most efficiently are answered many-to-many queries with the largest number of source and target objects (since reachability is determined faster). Among $\bar{R}$-queries such queries are processed the least efficiently (the increase in the number of source and target objects leads to expansion of the search space).

7 CONCLUSIONS

We introduced a new variation on spatiotemporal reachability queries, i.e., reachability queries with meetings, and proposed two algorithms, RICCmeetMin and RICCmeetMax, for efficient processing of such queries on large disk-resident datasets. In all experiments, the RICCmeet algorithms showed significantly better performance than an adapted previous approach. RICCmeetMax outperforms RICCmeetMin in all cases except for the shortest meeting duration queries. We also showed that these algorithms can be adapted to efficiently address many-to-many reachability queries with meetings. We are currently extending our algorithms to support aggregate-based reachability queries. An interesting problem is to examine reachability queries while objects are moving using heterogeneous transportation means (e.g., driving, biking, walking).
Acknowledgment: This research was partially supported by NSF grant IIS-1527984.

REFERENCES

[1] R. Agrawal, A.Borgida, and H.V.Jagadish. 1989. Efficient Managemet on Transitive Relationships in Large Data and Knowledge Bases. In *ACM SIGMOD*. 253–262.
[2] P. Bakalov, M. Hadjieleftheriou, E. Keogh, and V.J. Tsotras. 2005. Efficient trajectory joins using symbolic representations. In *MDM*. 86–93.
[3] T. Brinkhoff et al. 2003. Generating traffic data. *IEEE Data Eng. Bull.* 26, 2 (2003), 19–25.
[4] J. Cai and C. K. Poon. 2010. Path-hop: efficiently indexing large graphs for reachability queries. In *ACM CIKM*. 119–128.
[5] Y. Cai and R. Ng. 2004. Indexing Spatio-temporal Trajectories with Chebyshev Polynomials. In *ACM SIGMOD*. ACM, 599–610.
[6] S. Chen, B. Ooi, K. Tan, and M. Nascimento. 2008. ST2B-tree: A Self-tunable Spatio-temporal B+-tree Index for Moving Objects. In *ACM SIGMOD*. 29–42.
[7] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.
[8] V. T. De Almeida and R. H. Güting. 2005. Indexing the Trajectories of Moving Objects in Networks*. *Geoinformatica* 9, 1 (2005), 33–60.
[9] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. 2008. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *7th Intl. Conf. on Experimental algorithms*. 319–333.
[10] M. Hadjieleftheriou, G. Kollios, and V.J. Tsotras. 2003. Performance Evaluation of Spatio-temporal Selectivity Estimation Techniques. In *SSDBM*. 202–211.
[11] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. 2002. Efficient indexing of spatiotemporal objects. In *EDBT*. 251–268.
[12] W. Huo and V.J. Tsotras. 2014. Efficient Temporal Shortest Path Queries on Evolving Social Graphs. In *SSDBM Conf.* 38:1–38:4.
[13] C. Jensen, D. Lin, and B. Ooi. 2007. Continuous clustering of moving objects. *IEEE TKDE* 19, 9 (2007), 1161–1174.
[14] H. Jeung, M. Yiu, X. Zhou, C. Jensen, and H. Shen. 2008. Discovery of convoys in trajectory databases. In *PVLDB*, Vol. 1. 1068–1080.
[15] E. Jin, N. Ruan, S. Dey, and J. Y. Xu. 2012. SCARAB: scaling reachability computation on large graphs. In *ACM SIGMOD*. 169–180.
[16] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 2009. 3-hop: a high-compression indexing scheme for reachability query. In *ACM SIGMOD*. 813–826.
[17] P. Kalnis, N. Mamoulis, and S. Bakiras. 2005. On discovering moving clusters in spatio-temporal data. In *SSTD*. 364–381.
[18] U. Khurana and A. Deshpande. 2013. Efficient snapshot retrieval over historical graph data. In *IEEE ICDE*. 997–1008.
[19] G. Kollios, D. Gunopulos, and V.J. Tsotras. 1999. On Indexing Mobile Objects. In *ACM PODS*. 261–272.
[20] D.A. Maltz. 1996. Dynamic source routing in ad hoc wireless networks. *Mobile Computing* 353, 1 (1996), 153–181.
[21] F. Merz and P. Sanders. 2014. PReaCH: A Fast Lightweight Reachability Index Using Pruning and Contraction Hierarchies. In *ESA Symp.* 701–712.
[22] L.V. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel. 2010. Spatio-temporal Access Methods: Part2 (2003 - 2010). *IEEE Data Engineering Bulletin* 33, 2 (2010), 46–55.
[23] J. Ni and C.V. Ravishankar. 2007. Indexing Spatiotemporal Trajectories with Efficient Polynomial Approximation. *IEEE TKDE* 19, 5 (2007).
[24] J. Ni and C.V. Ravishankar. 2007. Pointwise-Dense Region Queries in Spatio-temporal Databases. In *IEEE ICDE*. 1066–1075.
[25] J. M. Patel, Y. Chen, and V. P. Chakka. 2004. STRIPES: An Efficient Index for Predicted Trajectories. In *ACM SIGMOD*. 635–646.
[26] D. Pfoser, C. S. Jensen, and Y. Theodoridis. 2000. Novel Approaches in Query Processing for Moving Object Trajectories. In *VLDB*. 395–406.
[27] M. Sarwat and Y. Sun. 2017. Answering Location-Aware Graph Reachability Queries on GeoSocial Data. In *IEEE ICDE*. 207–210.
[28] S. Seufert, A. Anand, S. Bedathur, and G. Weikum. 2013. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *IEEE ICDE*. 1009–1020.
[29] H. Shirani-Mehr, F. Banaei-Kashani, and C. Shahabi. 2012. Efficient reachability query evaluation in large spatiotemporal contact datasets. In *PVLDB*, Vol. 5. 848–859.
[30] E. V. Strzheletska and V. J. Tsotras. 2015. RICC: fast reachability query processing on large spatiotemporal datasets. In *SSTD*. 3–21.
[31] Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. 2016. Reachability Querying: Can It Be Even Faster? *IEEE TKDE* (2016), 683–697.
[32] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. 2010. Characterising Temporal Distance and Reachability in Mobile and Online Social Networks. *ACM SIGCOMM Computer Communication Review* 40, 1 (2010), 118–124.
[33] M.R. Vieira, P. Bakalov, and V.J. Tsotras. 2010. Querying Trajectories Using Flexible Patterns. In *EDBT*. 406–417.
[34] M. R. Vieira, P. Bakalov, and V.J.Tsotras. 2009. On-line discovery of flock patterns in spatio-temporal data. In *ACM GIS*. 286–295.
[35] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and Mario A. Lopez. 2000. Indexing the Positions of Continuously Moving Objects. In *ACM SIGMOD*. 331–342.
[36] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. 2006. Dual labeling: Answering graph reachability queries in constant time. In *IEEE ICDE*. 75–75.
[37] X. Xiong, M. F. Mokbel, and W. G. Aref. 2006. LUGrid: Update-tolerant grid-based indexing for moving objects. In *MDM*, Vol. 13.
[38] H. Yildirim, V. Chaoji, and M. J. Zaki. 2010. GRAIL: scalable reachability index for large graphs. In *PVLDB*. 276–284.
[39] M. Yiu, Y. Tao, and N. Mamoulis. 2008. The Bdual-Tree: Indexing Moving Objects by Space Filling Curves in Dual Space. *VLDB J.* 17, 3 (2008), 379–400.
[40] T. Yufei, D. Papadias, and J. Sun. 2003. The TPR*-tree: An Optimized Spatio-temporal Access Method for Predictive Queries. In *VLDB*. 790–801.