Title

Efficient Processing of Novel Reachability-Based Queries on Large Spatiotemporal Datasets

Author

Strzheletska, Elena Vladislavivna

Publication Date

2018

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Efficient Processing of Novel Reachability-Based Queries on Large Spatiotemporal Datasets

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Elena V. Strzheletska

September 2018

Dissertation Committee:

Dr. Vassilis J. Tsotras, Chairperson
Dr. Marek Chrobak
Dr. Vagelis Hristidis
Dr. Stefano Lonardi

The Dissertation of Elena V. Strzheletska is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

First, I would like to express my deepest gratitude to my advisor, Dr. Vassilis Tsotras, without whom this work would not have been possible. I was very fortunate to be your student. Thank you for taking your time to answer all the questions that I asked, both professional and personal. Thank you for your wisdom, encouragement, and patience.

I would like to thank my dissertation committee members, Dr. Marek Chrobak, Dr. Vagelis Hristidis, and Dr. Stefano Lonardi for their interesting questions and valuable suggestions that improved the quality of this work. I thank again Dr. Marek Chrobak, who, as Department Chair, encouraged me to teach in the CSE Department.

I would like to give special thanks to my professors from CSU San Bernardino, Dr. John Sarli and Dr. Robert Stein, as well as to my friend Elena Harris, for encouraging me to apply and pursue the PhD degree in Computer Science at UC Riverside. I thank my labmates from UCR and my friends for their support. I wish to thank Amy Ricks, Vanda Yamaguchi, and Trina Elerts for the help that they give to the graduate students.

Finally, I would like to mention how much I appreciate the love and support that came from my family.

To my beloved and loving son Alexander,

to my husband Mark, who always encourages me,

and to my parents who supported me throughout my life.

ABSTRACT OF THE DISSERTATION

Efficient Processing of Novel Reachability-Based Queries on Large Spatiotemporal Datasets

by

Elena V. Strzheletska

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2018
Dr. Vassilis J. Tsotras, Chairperson

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. While traditional graph reachability queries have been studied extensively, little work exists on processing reachability queries on large disk-resident trajectory datasets. What makes spatiotemporal reachability queries different and challenging is that the associated graph is dynamic and space-time dependent. As the spatiotemporal dataset becomes very large over time, a solution needs to be I/O-efficient.

Given two objects O_S and O_T, and a time interval I, a spatiotemporal reachability query identifies whether information (or physical item etc.) could have been transferred from O_S to O_T during I (typically indirectly through a chain of intermediate transfers). In the previous research on spatiotemporal reachability queries, it is assumed that information can be passed from one object to another instantaneously, which may not always be the case.

In this dissertation, we introduce several novel reachability-based queries. For all our problems, we assume that instant transfer is not possible, and consider reachability queries with different types of delays (processing and transfer delays), as well as queries with information decay. First, we propose the RICC (Reachability Index Construction by Contraction) framework for processing spatiotemporal reachability queries with processing delays. Next, using this framework, we address reachability queries with transfer delays (or meetings). For this purpose we design two algorithms, RICCmeetMin that precomputes some reachability events considering the shortest valid meetings duration, and RICCmeetMax which uses the longest possible meeting duration.

Our next work considers reachability queries under the scenario of information decay. Such queries arise when the value of information that travels through the chain of intermediate objects decreases with each transfer. This leads to an interesting extension: if there are many different sources of information, the aggregate value of information an object can obtain varies. As a result, we examine a top-k reachability problem, identifying the k objects with the highest accumulated information.

All proposed algorithms consist of two stages: preprocessing and query processing. To prune the search space during query time, they precompute and store some reachability information. This approach allows for efficient reachability query processing on large disk-resident datasets.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Spatiotemporal reachability queries arise naturally when determining how diseases, information, physical items can propagate through a collection of moving objects. Such queries are significant for many important domains like epidemiology, public health, social networks, surveillance, and security monitoring. The last two application areas involve performing reachability queries on spatiotemporal datasets, which are the main interest of this dissertation. Such datasets may, for instance, contain information about locations of a set of moving objects collected during some period of time.

Let $O = \{O_1, O_2, ..., O_n\}$ be a set of moving objects. Two objects O_i and O_j have a contact at time t_k (denoted as $< O_i, O_j, t_k >$), if they are within some threshold distance d_{cont} from each other at that time instant [43]. During the encounter, the proximity between O_i and O_j gives them an opportunity to exchange physical items or information (perhaps wirelessly), or a virus. As they move through the network, O_i and O_j may encounter other objects, and participate in further exchanges. This pattern permits moving objects to function as couriers, allowing two objects that remain far apart to nonetheless communicate with each other via intermediaries. A spatiotemporal reachability query determines whether two given objects O_S (the source) and O_T (the target) could have communicated (possibly through other objects), within a given time interval.

The time to exchange information (or physical items etc.) between objects affects the problem solution and it is application specific. Previous work assumes an 'instant exchange' scenario (where information can be instantly transferred and retransmitted between objects), which may not be the case in many real world applications. In this dissertation, we introduce several novel types of spatiotemporal reachability queries without the 'instant exchange' assumption.

We consider two types of delays that may occur during an exchange: *processing delay* and *transfer delay*. After two objects had a contact, the contacted object may have to spend some time to process the received information (processing delay) before being able to exchange it again; consider for example repackaging the physical item at the receiver object before resending. In other applications, for the transfer of information to occur (transfer delay), two objects are required to stay within the contact distance for some period of time; we call such elongated contact a *meeting*. An example appears if two cars exchange messages through Bluetooth and thus have to travel closely together for some time. We name these two problems *reachability with processing delay* and *reachability with transfer delay*. Later, we present efficient solutions for processing both types of reachability queries with delays.

In two reachability scenarios described above, we thought of a transferred item (e.g. information) as having a constant value, independently of the number of times the item was transferred. It is not always true in real-world applications: for example, if two people communicate over Bluetooth-enabled devices, due to some technical issues, the recipient may not get the message completely, and thus some information may be lost. During the further exchanges, the portion of the received information continues to decrease. In this situation, it is reasonable to limit the number of transfers (hops) that an item is allowed to travel from the source object. We name the problem that follows this scenario *the reachability problem with transfer decay.*

An extension of this problem is a top-k reachability problem. It may arise, for example, if there are many different sources of information that carry different items of possibly different values. Then the aggregate value of information an object can obtain may vary significantly from one object to another. A top-k reachability query would be to identify the k objects with the highest accumulated weight. Later, we describe our solutions for both, reachability with decay as well as k-top reachability queries.

There are two naive approaches that could be used to answer a reachability query on a small spatiotemporal dataset. The first approach (no-preprocessing) is to traverse the dataset at query time, from the beginning to the end of the query time interval, collecting all the objects that were reached by the source, and checking whether the target is among the collected objects (in which case the search can be stopped before the end of the interval is reached). If not, the search proceeds, etc. The second approach (precompute-all) is to precompute and store the reachability between every pair of objects for each possible time interval in advance. Both approaches are infeasible for our problem size, since they would require either too much time or space.

In this work, we consider large sets of moving objects, that are being observed over long periods of time. This means that the trajectory data cannot fit in main memory, and thus the solution must be I/O efficient.

1.2 RelatedWork

Static Graph Reachability. There are many approaches that have been proposed for the static graph reachability problem and their performance lies between the two naive approaches mentioned in the previous section. They are categorized in [25] as using: (i) transitive closure compression, (ii) hop labeling, and (iii) refined online search. The first category encompasses methods that compute and compress a transitive closure. Examples include interval labeling [1], dual labeling [53], chain decomposition, tree cover, etc. The next category includes hop labeling methods: 2-hop cover [11], 3-hop cover [26] and path-top [7]. For instance, in the 2-hop approach a node u in a graph G is assigned a label, which consists of two sets of nodes: a set L_{in} that contains nodes that can reach u, and a set L_{out} of those nodes that can be reached by u. Then a node v is reachable from u if and only if L_{in} and L_{out} have a non-empty intersection. Representatives from the third category include GRAIL [57], which uses indexing based on randomized multiple interval labeling, and PReaCH [32], that applies the Contraction Hierarchies technique [18] to the reachability problem and utilizes topological levels from GRAIL. GRAIL and PReaCH outperform other reachability methods on large static graphs.

Shortest Paths on Road Networks. In our model, the reachability question is equivalent to a shortest path query in a supergraph with edges of weight 1 for consecutive object positions and edges of weight 0 for contacts, with the restriction that a path should not contain two consecutive 0-weight edges in a row. Contraction Hierarchies [18] represent the state-of-the-art for solving shortest path problems on road networks. The preprocessing of CH consists of assigning an order to each node in the road network, and then contracting the nodes in that order, introducing shortcut edges to preserve the shortest path weight for any two nodes in the graph. A shortest path query is being answered by performing a Dijkstra search in the resulted contracted graph. Nevertheless, directly applying CH would not be efficient for our reachability problem. CH benefits from creating a hierarchy of nodes on the basis of their importance for the given road network, while in the spatiotemporal reachability problem, there is no node preference between the graph nodes. Algorithm PReaCH [32] discussed above, applies CH on the static reachability problem (and thus does not exploit the spatiotemporal properties of data).

Evolving Graphs. Evolving graphs (social, citation, biological networks, etc.) have recently experienced high popularity and received increased interest in the research community. In [29], the DeltaGraph is introduced, an external hierarchical index structure that enables efficient storing and retrieving of historical graph snapshots. For large dynamic graphs, [60] constructs a reachability index, based on a combination of labeling, ordering, and updating techniques. The work in [48] utilizes graph reachability labeling methods to develop techniques for analyzing temporal distance and reachability of temporal graphs. Information, stored in such datasets, is of a different nature, if compared with spatiotemporal data. Our problem is complicated by the need to compute the contacts between the objects, while such contacts are already available in evolving graph applications. In addition, out data has spatial properties, which is usually not the case in the analysis, for example, of social and citation networks.

Spatiotemporal Databases. *Spatiotemporal Access Methods.* There has been a large number of works on spatiotemporal access methods; these typically involve some variation on hierarchical trees [30, 39, 59, 19, 52, 12, 58, 10], or some form of a grid-based structure [38, 56] or indexing in parametric space [36, 8, 5]. A recent survey appears in [35]. Nevertheless, existing spatiotemporal indexes typically support traditional range and nearest neighbor queries and not the reachability queries we examine here.

Complex Queries on Spatiotemporal Datasets. Recent work has focused on querying/identifying the behavior of moving objects. Various methods have been developed for determining patterns and similar behavior of a group of objects during a particular time interval. Examples include discovering moving clusters [23, 28], flock patterns [49], and convoy queries [24].

Spatiotemporal Reachability Queries. Recently, [43] provided the first disk-based solutions for the spatiotemporal reachability problem, namely ReachGrid and ReachGraph. These are indexes on the contact dataset that enable faster query times. In ReachGrid, during query processing only a necessary portion of the contact network which is required for reachability evaluation is constructed and traversed. In ReachGraph, the reachability at different scales is precomputed and then reused at query time. Among the two approaches, ReachGraph is superior (and showed that it also greatly outperforms traditional graph reachability solutions like GRAIL [57]). However, what enables ReachGraph is the assumption that a contact between two objects can be instantaneous, and thus during one time instance, a chain of contacts may occur. Conceptually, this 'instant exchange' assumption, allows ReachGraph to be smaller in size (the new graph uses a single vertex for all objects that could be contacted at a given time instant) and thus reduce query time. On the other hand, ReachGrid does not require the 'instant exchange' assumption and is compared with our proposed methods through experimentation.

The work in [45] introduces two types of the 'no instant exchange' spatiotemporal reachability queries: reachability queries with *processing delay* and *transfer delay*, and proposes a solution to the first type. For the index construction, it utilizes the *path contraction* idea, introduced in Contraction Hierarchies [18]. The algorithm for processing reachability queries with *transfer delays (meetings)* is given in [46]. It proposes two algorithms, RICCmeetMin and RICCmeetMax. In order to reduce the search space during query processing time, these algorithms precompute the shortest valid meetings (RICCmeetMin), and the longest possible meetings (RICCmeetMax) respectively.

Top-k Queries. A well known Fagin's Algorithm for answering top-k queries, was described in [14]. It was modified and further developed in [16] and [34], and described in [15] . These algorithms are designed for large databases that contain objects with different attributes (color, shape, etc.). To answer a query, these algorithms access the lists with objects' information in particular order, while an aggregated function combines the scores of the attributes of the objects, and reports k objects with the highest aggregate scores.

Among very popular today are top-k spatial k-word queries and top-k spatial preference queries. The first type of queries asks to report k objects that are closest to the query location and satisfy the keyword requests [13], [54], [55], [3]. An experimental evaluation of spatial k-word query processing algorithms is given in [9]. The queries of the second type request k data objects with highest scores, where the scoring depends on the feature objects in the data objects' spatial neighborhood [40], [4]. An example of such query may be: find k hotels with the best restaurants and golf courses nearby. Finally, the work on top-k spatiotemporal queries [2], [44] identifies highest scored terms in the given location at the given time. To the best of our knowledge, the existing work on top-k queries does not address querying spatiotemporal datasets of moving objects.

1.3 Dissertation Overview

The rest of the dissertation is organized as follows: Chapter 2 presents the RICC (Reachability Index Construction by Contraction) approach for processing spatiotemporal reachability queries with processing delay. Chapter 3 proposes two RICCmeet algorithms that solve reachability with transfer delay (or reachability with meetings) problem and compares their performance. In Chapter 4, we present the RICCdecay algorithm for solving the reachability with transfer decay problem and RICCtopK algorithm for processing top-k reachability queries with decays. Finally, Chapter 5 concludes our work.

Chapter 2

Answering Reachability Queries with Processing Delay Efficiently

2.1 Problem Description

In this chapter, we discuss one of the two earlier mentioned types of spatiotemporal reachability queries without the 'instant exchange' assumption, namely, reachability queries with processing delay. Recall, that a contact occurs between two objects O_i and O_j at time t_k (it was denoted as $< O_i, O_j, t_k >$), if at this time instant they are within some threshold distance d_{cont} from each other [43]. During such contact, object O_i may transfer to O_j some information (or physical item, virus). Further, O_i and O_j may communicate with other objects, dispersing information throughout the network. As a result, two objects that have never been in contact with each other, still may have communicated through other objects.

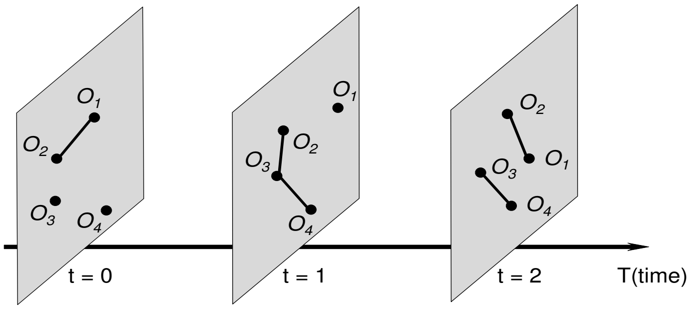


Figure 2.1: Positions and contacts between a set of moving objects during the time interval $[0, 2]$.

An example appears in Figure 2.1, where four moving objects are shown at consecutive time instants. Lines between objects denote contacts at those time instants. For example, objects O_1 and O_2 are in contact at times $t = 0$ and $t = 2$. Note, that objects O_1 and O_3 never contacted each other explicitly, however O_3 is reachable from O_1 within the time interval $[0, 1]$ through object O_2 (O_1 could pass information to O_2 at time $t = 0$, and O_2 could pass it to O_3 at time $t = 1$).

Depending on the problem application, transfers between objects may follow different scenarios, and this affects the problem solution. Earlier we talked about two possible kinds of delays: *processing delay* and *transfer delay.* The *processing delay* occurs after the contact, in case if the object that just received information needs some time to process it before it is ready to start retransmission. The *transfer delay* requires two objects to stay within the contact distance for some period of time (i.e. to have a *meeting*).

Thus one may consider the reachability problem with no delays, one type of delay (processing or transfer), and both types of delays. To distinguish among the various scenarios we use P to denote the existence of processing delay and T for transfer delay;

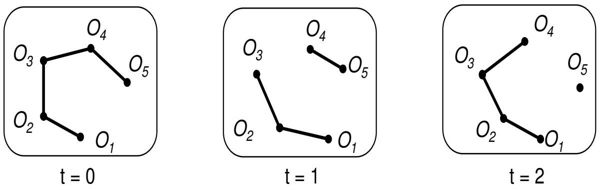


Figure 2.2: Contact graphs for a set of moving objects during time interval $[0, 2]$.

their absence will be denoted by $\bar{P}$ and $\bar{T}$ respectively. If no delays are present (i.e., $\bar{P}\bar{T}$) the exchange is considered (almost) instantaneous. This scenario (we will call it 'instant exchange') is assumed in [43]. In our work, we consider reachability scenarios with 'no instant exchange'.

Consider the example in Figure 2.1 where at time $t = 1$ a chain of contacts occurs: object O_2 contacts O_3, and O_3 contacts O_4. Assuming instantaneous exchanges, at this time instant information can travel from O_2 to its immediate contacts, and at the same time to all the current contacts of its contacts, etc., resulting in object O_4 been reached by O_2 during just one time instant $t = 1$. As another example, consider the case $P\bar{T}$, that is, with processing delay (i.e., an object receiving information at time t may not immediately retransmit it) and no transfer delay (i.e. a simple contact is enough to transfer the information). In Figure 2.1, at time $t = 1$, object O_2 contacts object O_3, and O_3 contacts O_4, but information from O_2 does not reach O_4 at that time instant.

A trajectory of a moving object O_i is a sequence of pairs (l_j, t_j), where l_j is the location of object O_i at time t_j. We assume that time is discrete, described as a sequence of time instants $(t_1, t_2, ..., t_i, ...)$ and the interval between two consecutive time instants is

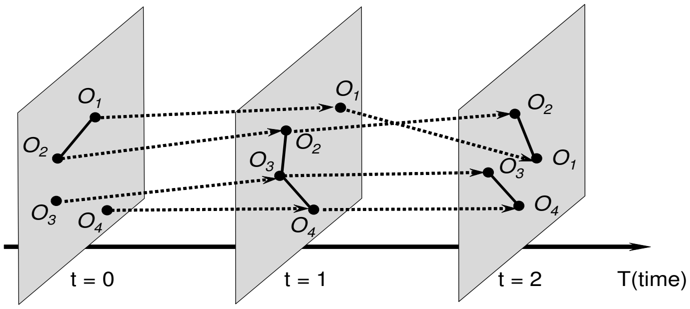


Figure 2.3: Constructing a supergraph on the time interval $[0, 2]$ by combining the contact graphs with the object trajectories.

constant (denoted as Δt). Moreover, each object reports its location at each time instant. We further assume that all contacts between objects are identified by looking at their location records (that is, Δt is small enough that we do not miss any contact between consecutive time instants).

Consider the $P\bar{T}$ reachability scenario: for simplicity we assume that the processing delay is Δt, and after a contact occurs, retransmission starts at the next time instant (our solution can be easily modified to consider the case where the processing delay is a multiple of Δt). The goal of a reachability query Q: $\{O_S, O_T, I\}$ is to determine whether object O_T (target) is reachable from object O_S (source) during time interval $I = [t_s, t_f]$, or in other words 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 $t_s \leq t_1 < t_2 ... < t_k \leq t_f$. Moreover, if such a chain exists, we would like to find the earliest time instant when O_T was reached (this can have implications on the application: trying to control the spread of the disease fast, or identify the shortest time that information traveled through a network).

Note again how the answer to a reachability query depends on the transfer requirements. Consider the example in Figure 2.2: here the collection of five moving objects is observed during three time instants. Let $I = [t_0, t_2]$. The answer to the query $\{O_1, O_4, I\}$ under the $\bar{P}\bar{T}$ scenario is $t = 0$. Under the $P\bar{T}$ scenario, the answer is $t = 2$. Another query, $\{O_1, O_5, I\}$, will be answered with $t = 0$ in the first case, however, for the second case, the answer is $t = \infty$. In general, the set of objects, reached by some object O_i during time interval I under the $\bar{P}\bar{T}$ scenario is a superset of the set of objects reached under the $P\bar{T}$ scenario.

The traditional graph reachability problem examines whether a path exists between two vertices of a static graph, such as a road network. Spatiotemporal reachability is more complex, since even the underlying graph is determined by the time-varying relationships between the positions of objects traversing the road network. Moreover, the contact distance d_{cont} is a parameter, and not an edge of a static graph. One could reduce spatiotemporal reachability into static graph reachability by combining the contact graphs with the object trajectories into a supergraph (by adding an edge connecting two consecutive occurrences of each object). This appears in Figure 2.3 where dotted edges connect consecutive object positions. However this approach will be inefficient as the supergraph is very large and does not exploit the spatiotemporal properties of the dataset. The first efficient disk-based solution for a spatiotemporal reachability problem was recently given by [43]. The problem that this paper considered, was reachability with no delays ($\bar{P}\bar{T}$).

In this chapter, we present the RICC (**R**eachability **I**ndex **C**onstruction by **C**ontraction) algorithm for the $P\bar{T}$ reachability problem. In the next chapter, we show how it can be modified to work with no processing but transfer delays ($\bar{P}T$).

RICC balances preprocessing time, storage consumption, and query performance time. Its preprocessing consists of several steps: the contact network construction, the reachability network construction, and the contact and reachability index construction. For the reachability network construction, we utilize the *path contraction* idea, introduced in Contraction Hierarchies (CH) [18]. A contraction replaces a path between two nodes of a graph with a (shortcut) edge, which preserves the distance between these nodes. Methods based on CH are currently the fastest known approaches for answering shortest path queries on road networks [18, 17]. However, there are two major differences between our problem and computing shortest paths on road networks. CH gains its speed up from creating a hierarchy of nodes on the basis of their importance for the given road network, while in the spatiotemporal reachability problem, there is no preference between the graph nodes. In addition, road networks are typically static graphs, while our environment is dynamic. We thus created our version of path contraction, which decreases the size of the spatiotemporal reachability network, and thus reduces the space search, and consequently the reachability query time.

Figure 2.4(a) represents the supergraph G_1 constructed on time interval $I = [t_0, t_2)$ for the contact graphs in Figure 2.1, under the 'instant exchange' assumption ($\bar{P}\bar{T}$). At time $t = 1$ object O_2 can pass the information to the object O_3, which then can pass it further to O_4 at the same time instant. The supergraph G'_1 in Figure 2.4(b) is constructed

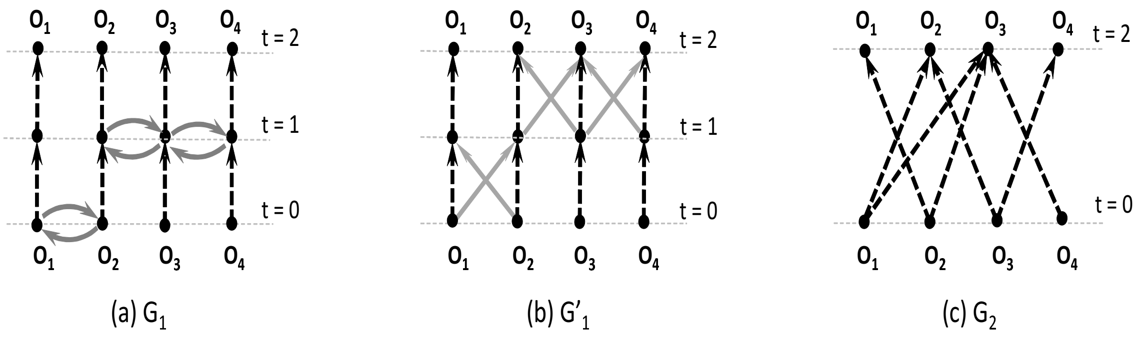


Figure 2.4: (a) G_1 is the supergraph under the $\bar{P}\bar{T}$ assumption; (b) DAG G_1' is the supergraph under the $P\bar{T}$ assumption; (c) the reachability graph G_2 constructed from G_1' for interval $I = [t_0, t_2)$.

using the same contact graphs but under the 'no instant exchange' assumption. To disallow the 'instant exchange' in G_1', for each pair of contacting objects O_i and O_j at time t_k, we remove edges that represent contacts between them. Next, we connect O_i at time t_k with O_j at time t_{k+1}, and vice versa. The resulting graph G_1' satisfies the required condition: in G_1' at time $t = 1$ object O_2 can pass the information to O_3, but O_3 cannot retransmit it to O_4 at the same time instant. Finally Figure 2.4(c) represents the reachability graph G_2, obtained from G_1' by contracting reachability paths and replacing them with new shortcut edges (and thus G_2 is a much smaller graph than G_1' while maintaining the same reachability properties).

The rest of the chapter is organized as follows: Section 2.2 introduces the RICC algorithm, its index construction and reachability query processing. In Section 2.3, we evaluate the performance of RICC using large spatiotemporal datasets representing objects moving on a real road network (created by the Brinkhoff generator [6]) as well objects moving freely on a 2-dimensional plane (based on the random waypoint model). Finally, Section 2.4 concludes the chapter.

2.2 RICC: Reachability Index Construction by Contraction

We proceed with the description of RICC. First we describe the preprocessing needed to maintain the contact and reachability networks and the indexing used to enable fast query time. Then the query processing algorithm is introduced.

2.2.1 Preprocessing

We start the preprocessing by dividing the entire time interval covered by the dataset into a number of non-overlapping subintervals, which we call *time blocks*; each of the created time blocks contains the information about the locations of all objects during the corresponding time interval. We call the number of time instants in each time block the *contraction parameter* C. Next, we partition the area covered by the dataset into spatial blocks (or grid cells), such that each cell is inscribed into a square with a side no greater than the contact distance d_{cont}.

For each time block, our algorithm performs several steps: multiple contact graph construction, reachability graph construction, and contact and reachability index construction. During the preprocessing, each time block is read into main memory only once, and all work on a block could be done as soon as the data for this particular block is collected.

Contact Graph Construction For this step, we need to materialize a contact graph for each time instant. To efficiently find all contacts between the objects during a given time instant, we start with partitioning the set of all objects that are active during this time instant into subsets on the basis of their location, and according to the area partitioning described above. Due to the size of each grid cell, all contacts of object O are located either in the same cell with O, or in adjacent cells. We can start, for example, with the left bottom cell of the grid, find all contacts between the objects in this cell, then all contacts between objects in this cell and objects in all adjacent cells. Further, we move to the next cell and proceed until all cells are visited.

After all contacts are found, a contact graph for this time instant is constructed: each object is represented by a vertex, and each contact between two objects - by an edge. Subsequently, when a contact graph is constructed for each time instant of the block, the information is recorded in the file *Contacts* as described later. First, all data about contacts between all the objects during each time instant of a block is collected. The set of the objects is being partitioned on the basis of their location at the first time instant of the block. This time, the size of the grid H (we will call it a grid resolution as in [43]), is much larger, than for the previous partition.(In the Experiments section we describe how to find a good value for H empirically.) Next, objects are sorted according to the order of cells that they belong to. Further, in this order, information about the contacts of each object during the time block, is sequentially written on disk into the file *Contacts*. A record for each object contains its contacts at each time instant of the block in time order. An example of the *Contacts* file appears in Figure 2.6.

Reachability Graph Construction To construct the reachability graph on one time block of the dataset, we start with creating a directed supergraph by collecting contact graphs for each time instant of a block (in time order) and connecting them by introducing

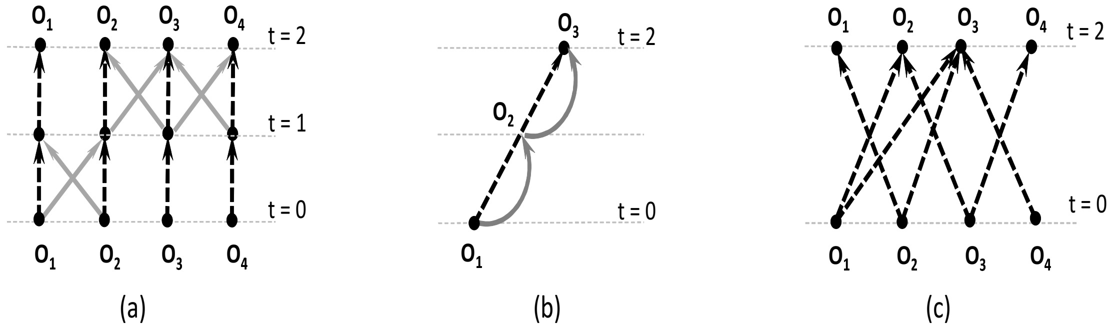


Figure 2.5: (a) Supergraph; (b) Path contraction between $O_1^{(0)}$ and $O_3^{(2)}$; (c) Non-trivial reachability graph on interval $I = [t_0, t_2)$ (contraction parameter $C = 2$).

an edge for each two consecutive occurrences of each object. Figure 2.5(a), shows a supergraph, constructed on a time block with contraction parameter $C = 2$ from two contact graphs given in Figure 2.1. The next step is to contract the reachability graph. Let $O_k^{(i)}$ denote an occurrence of object O_k during an i-th time instant of a block.

Theorem 1 *Let G^s be a supergraph constructed over a time block B. There exists a path in G^s from $O_k^{(0)}$ to $O_l^{(C-1)}$, if and only if, $O_l^{(C-1)}$ is reachable by $O_k^{(0)}$ during B.*

It follows, that to capture all reachability cases during a block, we need to answer, whether there is a path between every pair of vertices $O_k^{(0)}$ and $O_l^{(C-1)}$ in the supergraph constructed for that block. A path non-trivial if $k \neq l$. Next, we consider that any instance of object O_k is reachable from its later instance (there is a trivial path from $O_k^{(i)}$ to $O_k^{(j)}$ for $i \leq j$), and will not record it.

If there is a non-trivial path in G^s between $O_k^{(0)}$ and $O_l^{(C-1)}$, we contract this path, and replace it with an edge. In Figure 2.5(a), there is a path between $O_1^{(0)}$ and $O_3^{(2)}$, thus O_3 is reachable from O_1 during this block. This path can be contracted, and replaced by

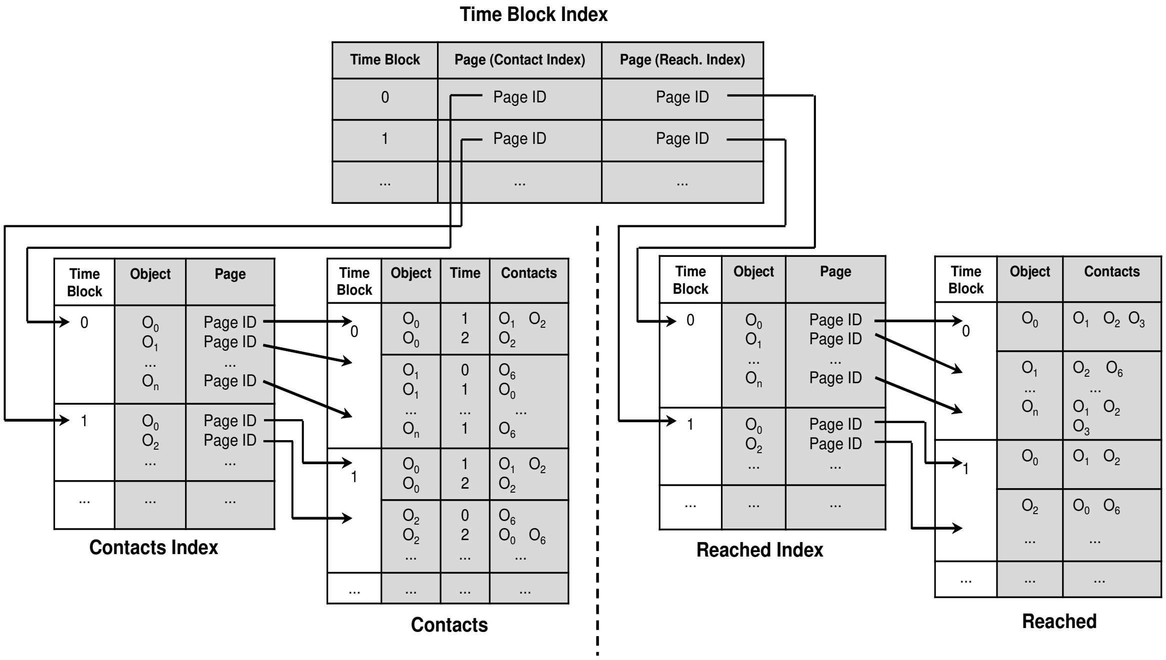


Figure 2.6: Two-level index on files Contacts and Reached.

a shortcut edge as in Figure 2.5(b). We can effectively find all the paths by using multi-source BFS from each object $O_k^{(0)}$ in G^s. Figure 2.5(c) depicts the final reachability graph. Upon construction of the reachability graph for a given block, all reachability information is written sequentially into file *Reached* in the same object order as for the contact graphs (Figure 2.6).

Contact and Reachability Index Construction. To efficiently retrieve information from disk, we use a two-level index, constructed on the files *Contacts* and *Reached.* An example of this index appears in Figure 2.6. The first level (*TimeBlockIndex*), is ordered by time block number: each record consists of the time block number, and two pointers to disk pages in the second level indexes, namely the *ContactsIndex* and the

ReachedIndex. Each record in the *ContactsIndex* is comprised of an object id and a pointer to the page in the file *Contacts*, which contains, which objects and when were contacted by this object during the given time block. Each record in the *ReachedIndex* is composed of object id and a pointer to the page in the file *Reached*, which contains, which objects were reached by this object during the given time block. The order of objects in each page of the *ContactsIndex* and *ReachedIndex* is the same as in *Contacts* and *Reached* respectively. Note that in Figure 2.6 with the exception of the Time Block Index, the time block numbers (left columns) are depicted for clarity (i.e., they are not part of the index).

2.2.2 Query Processing

Consider a query (O_S, O_T, I), where O_S is the source object, O_T is the target object, and time interval $I = [t_s, t_f]$. Before processing this query, we need to identify the time blocks that t_s and t_f belong to. Suppose, $t_s \in B_s$, and $t_f \in B_{f+1}$. Using the *TimeBlockIndex*, we can identify the starting positions of each block B_i (such that $B_s \leq B_i \leq B_f$) in the *ContactsIndex* and *ReachedIndex*. In most cases, the second level indexes, *ContactsIndex* and *ReachedIndex*, are accessed at most once per block, before accessing data related to contacts and reachability respectively. Let $S_{reached}$ denote the set of objects that have been reached so far. Initially, $S_{reached}$ contains only one element, the source object O_S. As the query proceeds, new elements are included into this set, and as soon as O_T is added to it (or the end of the last block is reached), the query processing terminates, as either the target, or the end of the query interval is reached.

Straightforward Query Processing. After $S_{reached}$ is initialized with O_S, a straightforward approach would be to start query processing from file *Contacts*. We discover objects that were in contact with O_S at time t_s, and add them to $S_{reached}$. The process has to be repeated, however now the contacts need to be found for each object that belongs to the updated $S_{reached}$ at time t_{s+1}. We proceed this way until the last time instant of the block B_s is processed. The next step is to find block B_{s+1} in file *Reached*, determine all objects that could be reached by each object from $S_{reached}$, and update $S_{reached}$. The algorithm iterates through these steps in *Reached* until either B_{f-1}-st block is processed, or the target is reached. Finally, the process returns to file $Contacts$. If O_T has not been reached, the remaining query interval that belongs to block B_f needs to be checked. On the other hand, if O_T was reached during or before B_{f-1}-st block, then the last block, processed in *Reached* has to be traversed in $Contacts$ once again, to determine the exact time of the contact, when target was reached.

Optimized Query Processing. At the beginning and at the end of the query, when processing information from $Contacts$, new objects are added to $S_{reached}$ at each time instant. This leads to an increase of disk accesses as parts of file $Contacts$ that cover the first and the last blocks may be read multiple times (in the worst case, C times, where C is the contraction parameter). This can be avoided if query processing begins from reading file $ReachedIndex$.

Theorem 2 *Let I and I' be two time intervals such that $I \subseteq I'$. If O_T is reachable from O_S during I, then O_T is reachable from O_S during I' as well. Also, if O_T is not reachable from O_S during I', then O_T is not reachable from O_S during I.*

The optimized query processing algorithm (Algorithm 1) starts from the *Reached–Index* (from the page, pointed by the $TimeBlockIndex$), and attempts to find a record for

Algorithm 1 Reachability query processing

```
procedure QUERY PROCESSING($O_S$, $O_T$, $I$)
    $S_{Reached} = \{O_S\}$, $t_{Reached} = \infty$
    find $B_s$ and $B_f$, $B_{cur} = B_s$
    $C_{Ind} = readTimeBlockIndex(B_s, B_f)$            ▷ Find position of each $B_i$ in
    $R_{Ind} = readTimeBlockIndex(B_s, B_f)$            ▷ $ContactsIndex$ and $ReachedIndex$
    while ($O_T \notin S_{Reached}$ and $B_{cur} \neq B_{f+1}$) do
        $R_{pageIDs} = \{\emptyset\}$            ▷ $R_{pageIDs}$ - list of pages to be read from $Reached$
        while ($R_{pageIDs} = \{\emptyset\}$ and $B_{cur} \neq B_{f+1}$) do
            $R_{pageIDs} = readReachedIndex(R_{ind}, S_{Reached})$
            $B_{cur}++$
        $S_{temp} = \{\emptyset\}$            ▷ $S_{temp}$ is the set of objects, reached during the block
        $S_{temp} = findReached(R_{PageIDs}, S_{Reached}, B_{cur})$
        if ($B_{cur} = B_s$ or $B_{cur} = B_f$ or $O_T \in S_{Reached}$) then
            $C_{pageIDs} = \{\emptyset\}$            ▷ $C_{pageIDs}$ - list of pages to be read from $Contacts$
            $C_{pageIDs} = readContactsIndex(C_{ind}, S_{Reached}, S_{temp})$
            $S_{new} = filterContacts(C_{PageIDs}, S_{Reached}, S_{temp})$
            $S_{Reached} = S_{Reached} \cup S_{new}$
            if ($O_T \in S_{Reached}$) then update $t_{Reached}$
        else($S_{Reached} = S_{Reached} \cup S_{temp}$)
        $B_{cur}++$
    return $t_{Reached}$            ▷ If $t_{Reached} = \infty$, then the target has not been reached
```

the source object (it will start at B_s and continue until either some record is found, or the end of the interval reached). If such record is found, it points to the page in *Reached*, from where we can determine all objects, that were reached by O_S during the current time block. However, if the current block is the first block of the query, and t_s is not the first time instant of this block, caution is needed, as (according to the theorem above) the set of objects, reached by O_S during B_s is the superset of the set of objects, reached by O_S from t_s to the end of B_s. Hence, we need to traverse *Contacts* to make sure that we filtered all the objects that do not satisfy the time condition (the only time they were reached by the source was before the beginning of the query). After the set $S_{reached}$ is finalized, the algorithm switches to file *Reached* again, and proceeds as in the previous version, with the exception of the last time block. Suppose, we arrived at the end of B_{f-1}, collected all objects that were reached so far, but O_T was not among them. Now, we continue in *Reached*, and record all objects that were reached during B_f. If the target is not one of them, the query processing is completed. However, if O_T was reached during B_f, and t_f is not the last time instant of this block, then (again, it follows from the theorem above) we have to return into *Contacts*, and confirm that the target was reached before the end of the query interval. Although this algorithm may read from *Contacts* at the beginning and/or at the end of the query, just like the straightforward query processing, the major difference is that in this case, we read a time block (or rather its portions) only once, thus minimizing the number of I/Os.

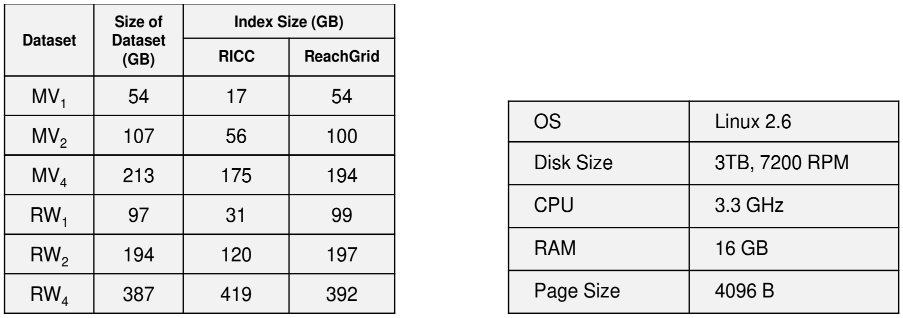
2.3 Experiments

2.3.1 Dataset Description

We tested the proposed algorithm on two types of realistic datasets. Three of the datasets were created by the Brinkhoff data generator [6], which generates traces of objects, moving on real road networks. For our experiments we chose the San Francisco Bay area road network, which covers an area of about $30000km^2$. Three datasets contain the information about $1000, 2000$, and 4000 moving (within the speed limit) vehicles respectively; the location of each vehicle was recorded every 5 seconds and collected during a four month period (a total of $2,040,000$ time instants). Further, we assume that wireless communication is held via the Dedicated Short-Range Communications protocol (DSRC), which can afford contacts for up to 300 meters. Thus, for the experiments on these datasets $d_{cont} = 300$ meters. We will refer to these sets as the Moving Vehicle datasets (or MV_1, MV_2, and MV_4 for sets of $1000, 2000$, and 4000 objects respectively).

For the second type of datasets, we created our own data generator, which utilizes the popular random waypoint model, frequently used for modeling movements of mobile users. According to this model, each user chooses the direction, speed (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. The three generated sets simulate the movements of $10000, 20000$, and 40000 individuals respectively, whose location is recorded every 6 seconds for a period of one month (432,000 time instants total), and cover the area of $100km^2$ each. These sets will be referred to as Random Waypoint datasets (or RW_1, RW_2, and RW_4 for sets of $10000, 20000$, and 40000 objects respectively). We perform two sets of experiments

Table 2.1: (a) Size of datasets and indexes, and (b) System specifications



Dataset	Size of Dataset (GB)	Index Size (GB)	
		RICC	ReachGrid
MV_1	54	17	54
MV_2	107	56	100
MV_4	213	175	194
RW_1	97	31	99
RW_2	194	120	197
RW_4	387	419	392

(a) Size of datasets and indexes

OS	Linux 2.6
Disk Size	3TB, 7200 RPM
CPU	3.3 GHz
RAM	16 GB
Page Size	4096 B

(b) System specifications

on these datasets. For the first, we presume the communication over a Bluetooth connection and a contact distance of $d_{cont} = 25$ meters. For the second set of experiments, we assume that the individuals have to transfer a physical item in order for the contact to occur, and set a contact distance to be $d_{cont} = 2$ meters. The size of each dataset is given in Table 2.1(a).

Since we consider disk-resident datasets, the performance is evaluated using the number of disk accesses (I/Os) for 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 [43]. In the rest, the total number of I/Os reports the equivalent number of random I/Os (that is, we assume that 20 sequential I/Os are equal to 1 random, and calculate the total number of I/Os using this ratio). The specifications for the system used for the experiments are given in Table 2.1(b).

Table 2.2: Parameter optimization on dataset MV_1

	Contraction Parameter (Time instants)				
		20	40	60	80
Grid Resolution (Thousand km)	20	9295	5884	5162	5779
	40	9277	5876	5192	5738
	60	9278	5874	5127	5656
	80	9260	5815	5146	5413

2.3.2 Parameter Optimization

The query performance of RICC depends on two parameters: the contraction parameter C and the grid resolution H, both of which are dataset dependent. To tune these parameters we used a subset of the dataset (of size 10%). In general, if data is time-wise homogeneous across a dataset, any portion of it could be used, while if data differs according to some pattern - day/night, rush hour, etc., a sample that reflects the pattern should be created. We tested the performance of RICC using a set of 300 queries (the length of each query was picked uniformly at random between 100 and 500 time instants), and found the pair (C, H), which minimized the number of I/Os. The results of the parameter tuning experiments for dataset MV_1 are shown in Table 2.2; based on these results for the rest of the experiments involving MV_1 we pick $(C, H) =$ (60, 60) (the values for the other datasets were picked in a similar way).

2.3.3 Preprocessing and Indexing

Preprocessing Time. The preprocessing time depends on the size of a dataset, as well as on the contraction parameter. During the parameter optimization phase, if there are cases where several pairs of parameters (C, H), give approximately the same query performance, we choose the pair with the smaller contraction parameter C as this leads to less preprocessing.

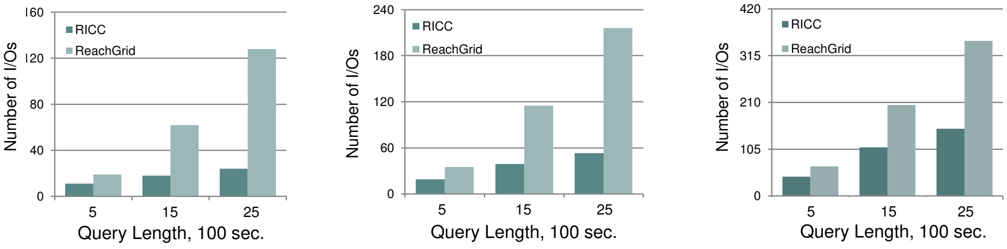
The preprocessing time for our datasets ranged from 90 minutes (for the Moving Vehicles, 1000 objects dataset) to 43 hours (for the Random Walk, 40000 objects dataset and $d_{cont} = 25$ meters). Taking into account the preprocessing speed, as well as the fact, that during the preprocessing each time block of data is read (consequently) into main memory only once, we conclude, that RICC can be applied for processing spatiotemporal data streams.)

Index Size. Fast reachability algorithms often suffer from large index size. The smallest query time is achieved when the transitive closure is precomputed (which however requires space that is quadratic on the graph size). Nevertheless, RICC can achieve very good query performance while its index size is relatively small as it can be seen from Table 2.1(a). This is because instead of transitive closure we precompute reachability for small portions of the graph.

2.3.4 Query Processing

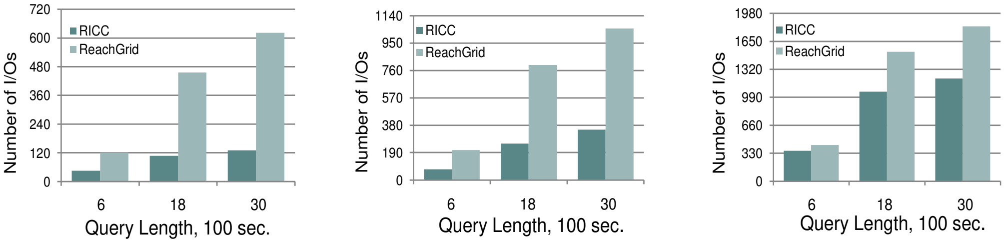
For the query processing performance evaluation, we ran different sets of 300 queries on each of the preprocessed datasets. Further we implemented the ReachGrid for the $P\bar{T}$ reachability, and optimized its parameters as described in [43].

One-to-One Queries. We first consider one-to-one queries $\{O_S, O_T, I\}$, (one source and one target). For both, the Moving Vehicles and Random Walk datasets, the



(a) MV_1 (b) MV_2 (c) MV_4

Figure 2.7: Query performance evaluation for one-to-one queries; MV datasets



(a) RW_1 (b) RW_2 (c) RW_4

Figure 2.8: Query performance evaluation for one-to-one queries; RW datasets

contact distance was set to 25 meters. We created three sets of queries: 500 sec, 1500 sec, and 2500 sec long for each of the MV datasets, and 600 sec, 1800 sec, and 3000 sec long for each of the RW datasets. The performance of RICC and ReachGrid was evaluated and compared on three sets of queries for each dataset by counting the number of I/Os. The results of these experiments are depicted in Figures 2.7 and 2.8. On all instances, our approach outperforms ReachGrid.

This improvement is because ReachGrid visits each object in a cell while RICC focuses on precomputed contacts. As the query length increases the number of objects to

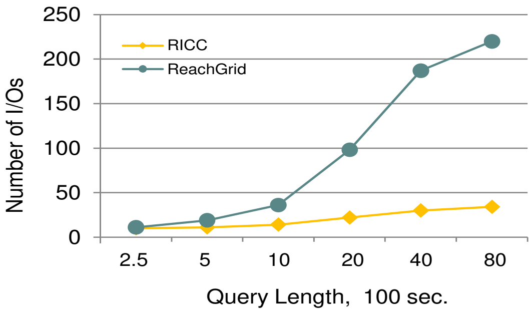


Figure 2.9: Scaling, $MV1$

be checked by ReachGrid increases rapidly. Thus the biggest advantage over ReachGrid (up to 5x improvement) is reached for the longest queries on the smallest datasets (MV_1, RW_1 which have smallest number of contacts).

Scaling. The next set of tests is used to analyze the dependence of the RICC performance on the query length. When starting processing a query we need to retrieve only a few objects from the disk. If the query specifies a large time interval, more objects become carriers, which in turn (depending on the efficiency of an algorithm) may affect the query performance. We tested our algorithm on $MV1$, the Moving Vehicles dataset with 1000 objects, with five sets of queries, with time intervals ranging from 250 to 8000 sec respectively (after 8000 time instants all objects in the MV_1 dataset were reached). As can be seen from Figure 2.9, while RICC uses a similar number of disk accesses as ReachGrid for the smallest length queries, it achieves much better query performance for the longer ones (up to 6.5 times for the 8000 sec interval). Further, RICC scales well with the size of the query length.

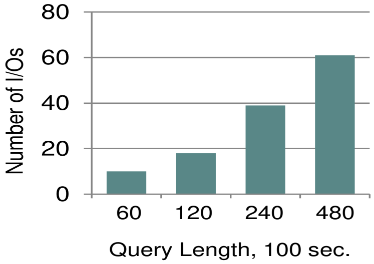


Figure 2.10: Long interval queries, $RW1$

Long Interval Queries. For this set of experiments we used RW_1, the Random Walk dataset with 1000 objects, setting $d_{cont} = 2$ meters. Since the contact distance is much smaller than previously, the average contact degree becomes smaller, which in turn leads to longer average time for two objects to reach each other. We started with queries that are 6000 time instants long, and extended the query length up to 48000 time instants (which for this dataset makes about 95% objects reachable by the end of the query interval). For these experiments, we were not able to optimize the parameters and complete the preprocessing for ReachGrid, since its query processing was very slow (ReachGrid does not scale well under the given scenario). As it can be seen from Figure 2.10, RICC can be effectively used for long interval queries as well (it scales almost linear with the query length).

Many-to-Many Queries. We proceed with the experimental results for many-to-many queries (i.e., queries with several sources and/or several targets). First we note that Single Source Multitarget Queries have the same performance as one-to-one queries. Let $(O_S, \{O_{T_1}, O_{T_2}\}, I)$ be a query with the set of targets $\{O_{T_1}, O_{T_2}\}$. Then the time to answer this query $t = max(t_{Q1}, t_{Q2})$, and $N_{IO} = max(N^1_{IO}, N^2_{IO})$ (where t_{Q_i} is the time

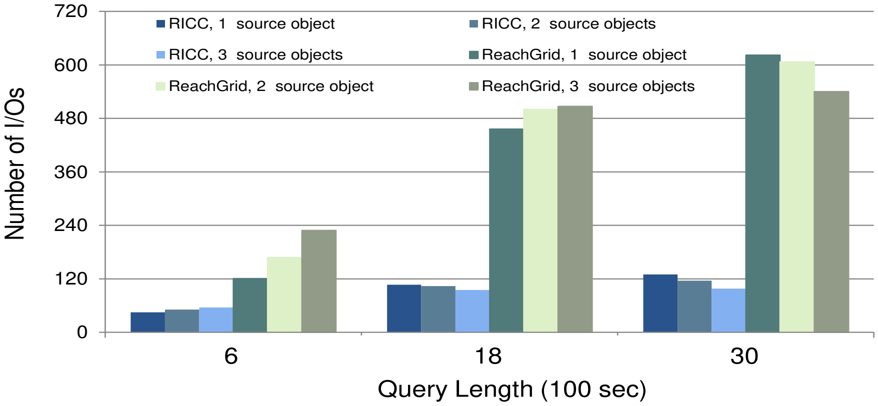


Figure 2.11: Many-to-many queries, $RW1$.

when and if the target t_i was reached (or the end of query interval otherwise), and N_{IO}^{i} is the number I/Os, needed to answer the query (O_S, O_{T_i}, I)).

More interesting are the *Multisource Queries.* In this case if an algorithm strongly utilizes a spatial locality for index construction, its performance should decrease when executing queries with more than one source. In the worst case (when sources are very far from each other), the number of I/Os of a query $(\{O_{S_1}, O_{S_2}\}, O_T, I)$ becomes $N_{IO} = N_{IO_1}^{1} + N_{IO_2}^{2}$.

For these experiments we used $RW1$ (the Random Walk dataset with 10000 objects). The contact distance d_{cont} was set to 25 meters. The testing was performed on three sets of queries that are 600 sec, 1800 sec, and 3000 sec long. As we can see from Figure 2.11, RICC outperforms ReachGrid on this set of experiments as well. Also, with the increase of the number of sources, the gap between the number of I/Os of RICC and ReachGrid, becomes larger.

2.4 Conclusions

We proposed the RICC algorithm for efficient spatiotemporal reachability query processing (without the instant exchange assumption) on large disk-resident datasets. We tested our algorithm on two types of realistic datasets and different types of queries. RICC outperformed the previous known algorithm (ReachGrid) on all experiments. In addition, our algorithm shows good performance for many-to-many queries and scales well.

Chapter 3

Efficient Processing of Reachability Queries with Transfer Delay

3.1 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 $O = \{O_1, O_2, ..., O_n\}$ (people, cars, etc.). Two objects O_i and O_j have a contact at time t_k, if they are within some threshold distance from each other at that time instant [43]. 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 chapter, 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 [45].

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 3.1, 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

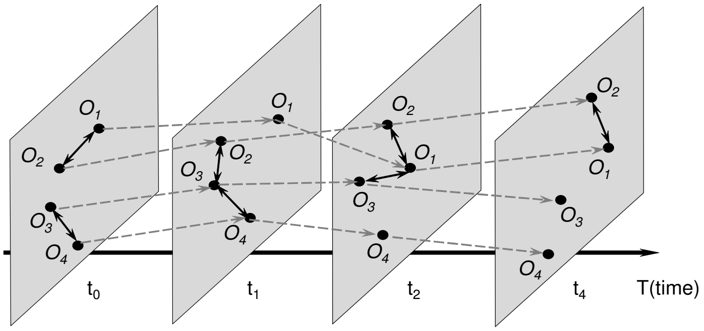


Figure 3.1: Constructing a supergraph by combining the contact graphs with the object trajectories.

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 [43]. 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 [45] 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 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 3.1: suppose 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 [45]). 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 chapter, 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 [43, 45], 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 chapter is organized as follows: Section 3.2 defines the reachability with meetings problem. The preprocessing and query processing for the two RICCmeet algorithms appear in Sections 3.3 and 3.4, while their performance is compared in Section 3.5. Finally, Section 3.6 presents our conclusions.

3.2 Reachability with Transfer Delay (Meetings)

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 [43, 45] 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

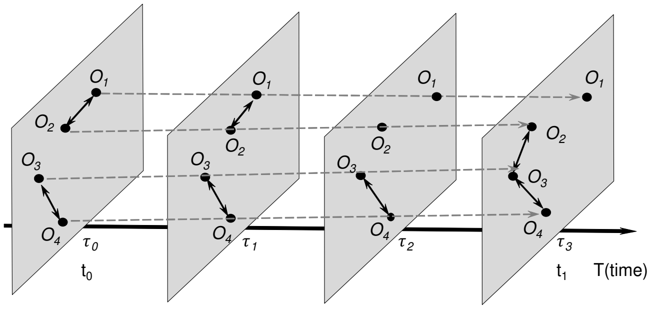


Figure 3.2: Discovering meetings between the objects on time interval $I = [t_0, t_1]$.

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 3.2. 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 $\tau_R(O_4) = \tau_0 + 2\Delta\tau$.

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 3.3 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 3.1 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 3.3(a)) and the 'processing delay' case (Figure 3.3(b))

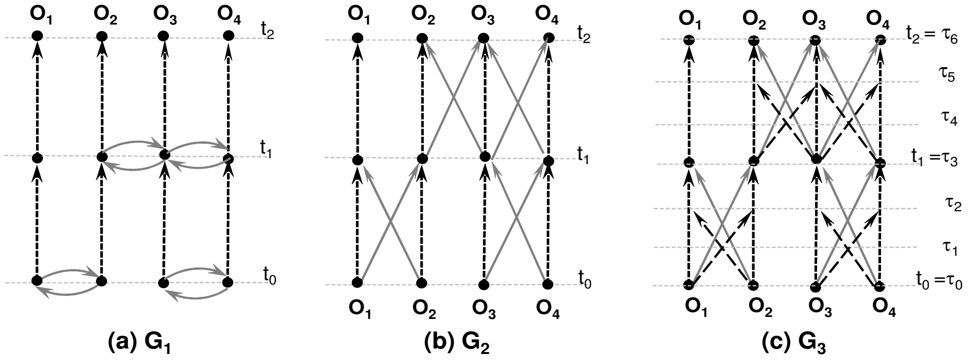


Figure 3.3: (a) graph G_1 represents the 'instant exchange' scenario; (b) graph G_2 depicts the 'processing delay' scenario with delay $\lambda < \Delta t$; (c) graph G_3 assumes the 'transfer delay' scenario (the time interval is $I = [t_0, t_2]$).

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 [45]). For the 'transfer delay' case (Figure 3.3(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.

3.3 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 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 3.5 we compare them with a baseline algorithm ReachGridmeet, which is a modified version of ReachGrid [43] 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 required meeting duration (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 [43] to take advantage of temporal locality

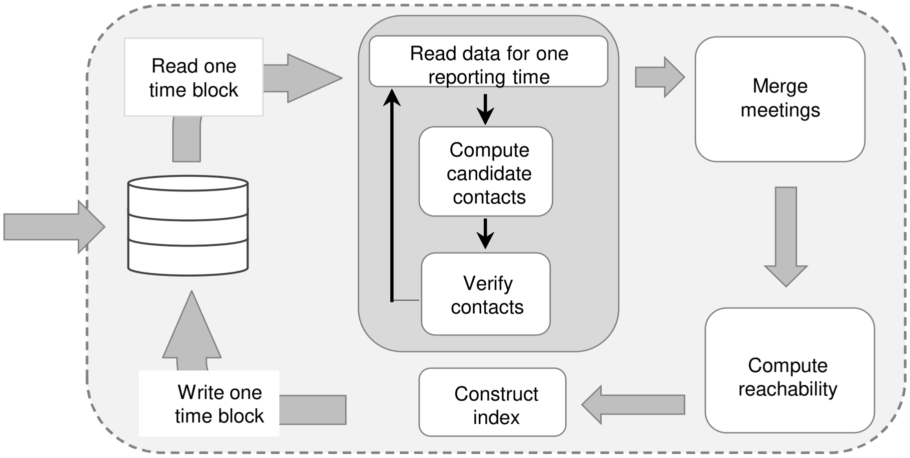


Figure 3.4: Preprocessing Workflow for RICCmeet algorithms

(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 3.5.

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 t_k), (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 3.4.

We take advantage of spatial locality by partitioning the area into cells with side H

Table 3.1: Notation used in the chapter

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
$\tau_R(O_i)$	Earliest time when object O_i was reached
B_k, I_k	Time block k that spans time interval I_k.
C	Contraction parameter
H	Grid resolution

(the *grid resolution*) - a parameter, whose tuning is discussed in Section 3.5. 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} (*candidate contact distance*); 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.

3.3.1 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 $(\tau,\ O_j)$, 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).

3.3.2 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).

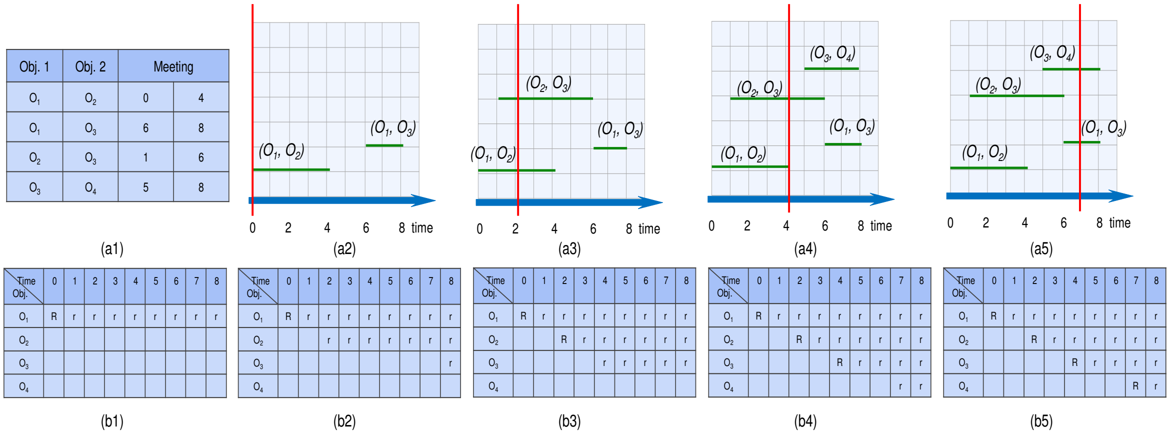
At the end of the current time block all meetings are persisted in file *Meetings*. During this step *objectRecords* are accessed in H cell order (so as to maintain spatial locality); within a cell they are thus ordered by object id, beginning meeting time, and companion id if meeting intervals are the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

3.3.3 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). The meetings graph for the dataset in Figure 3.5 is depicted in Figure 3.6(a).

To turn a meetings 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



Obj. 1	Obj. 2	Meeting	
O_1	O_2	0	4
O_1	O_3	6	8
O_2	O_3	1	6
O_3	O_4	5	8

(a1) (a2) (a3) (a4) (a5)

Time / Obj.	0	1	2	3	4	5	6	7	8
O_1	R	r	r	r	r	r	r	r	r
O_2									
O_3									
O_4									

(b1)

Time / Obj.	0	1	2	3	4	5	6	7	8
O_1	R	r	r	r	r	r	r	r	r
O_2			r	r	r	r	r	r	r
O_3									r
O_4									

(b2)

Time / Obj.	0	1	2	3	4	5	6	7	8
O_1	R	r	r	r	r	r	r	r	r
O_2			R	r	r	r	r	r	r
O_3					r	r	r	r	r
O_4									

(b3)

Time / Obj.	0	1	2	3	4	5	6	7	8
O_1	R	r	r	r	r	r	r	r	r
O_2			R	r	r	r	r	r	r
O_3					R	r	r	r	r
O_4								r	r

(b4)

Time / Obj.	0	1	2	3	4	5	6	7	8
O_1	R	r	r	r	r	r	r	r	r
O_2			R	r	r	r	r	r	r
O_3					R	r	r	r	r
O_4								R	r

(b5)

Figure 3.5: Computing the (m_q)-reachable objects from O_1 ($m_q = 2$)

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^R_k 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^R_k graph traversal. We start by visiting the earliest instance of object O_i in G^R_k, 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.

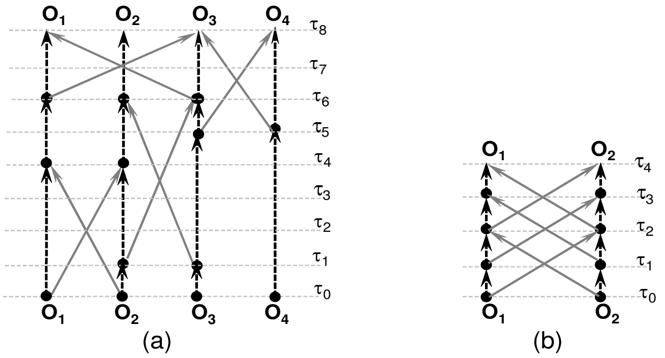


Figure 3.6: Meetings and reachability graphs construction: (a) Meetings graph G^M; (b) Reachability graph $G^R(\mu)$ for meeting $< O_1, O_2, [\tau_0, \tau_4] >$

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 3.5. 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 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 2. 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

Algorithm 2 Reach(m_q)

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

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. 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 3.5 (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 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 3 ReachMax

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

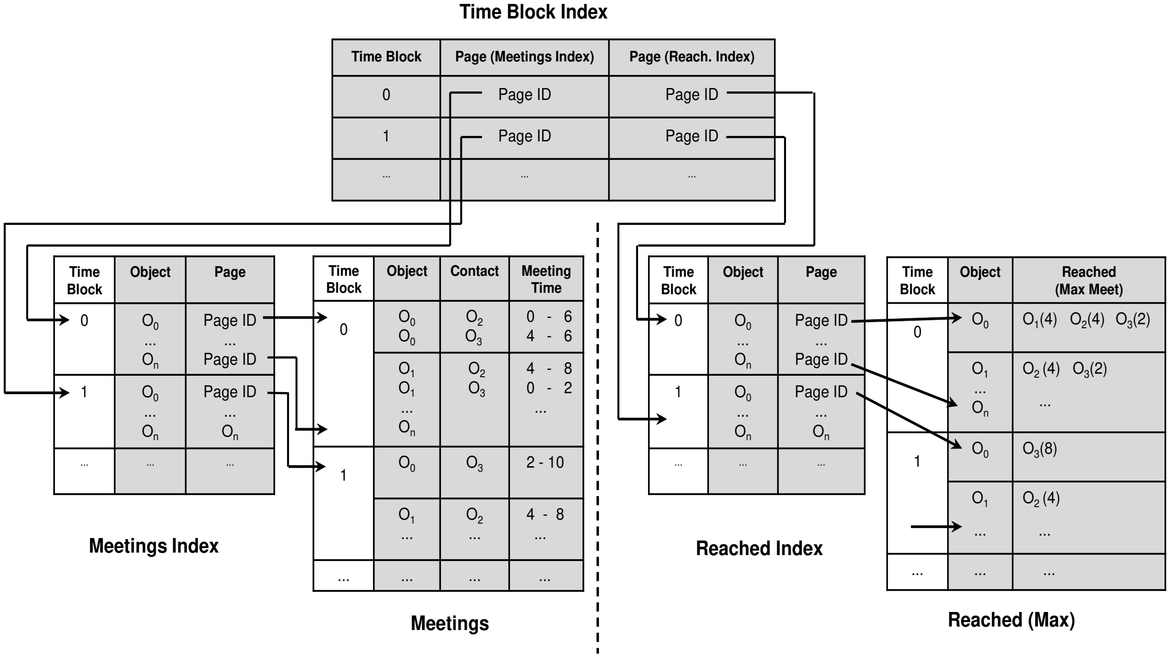


Figure 3.7: Two-level index on files Meetings and Reached(Max)

3.3.4 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 3.7). Records in the *Meetings Index* 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.

3.4 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 2. 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 $B_i \neq B_f$, 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.

3.5 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 [43] that works under the 'no instant exchange' 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.

3.5.1 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 [6], 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 [31], 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.

3.5.2 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

Table 3.2: 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

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 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.

3.5.3 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 3.2. 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.

3.5.4 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 [43]: 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 2.

We evaluate the query performance of the three algorithms while varying m_q on

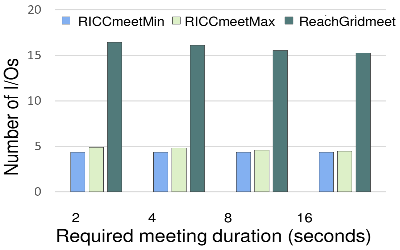


Figure 3.8: RICCmeet vs. ReachGridmeet

short queries (the query interval was set to 500 sec). Figure 3.8 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 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 $m_q = \mu$ ($\mu = 2$ 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'

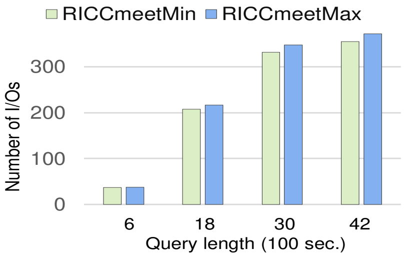
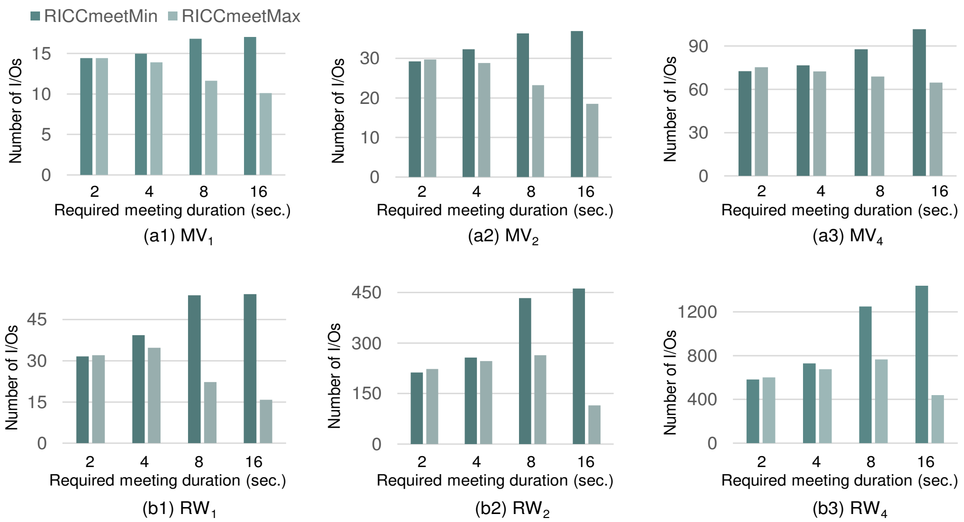


Figure 3.9: Minimum meeting duration queries

performances (up to 4.8%) was observed for RW_2 dataset, which we presented in Figure 3.9. 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 3.10 ($a1 - b3$). 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



(a1) MV_1 (a2) MV_2 (a3) MV_4

(b1) RW_1 (b2) RW_2 (b3) RW_4

Figure 3.10: Varying m_q

to the number of $\bar{R}$-queries defines the effectiveness of pruning and depends on m_q. 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 (see Figure 3.11). 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 3.12. While both algorithms show almost linear increase in the number

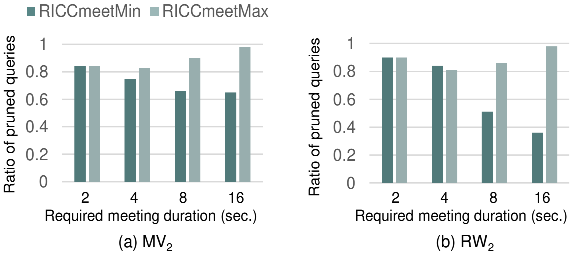


Figure 3.11: Pruning

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.

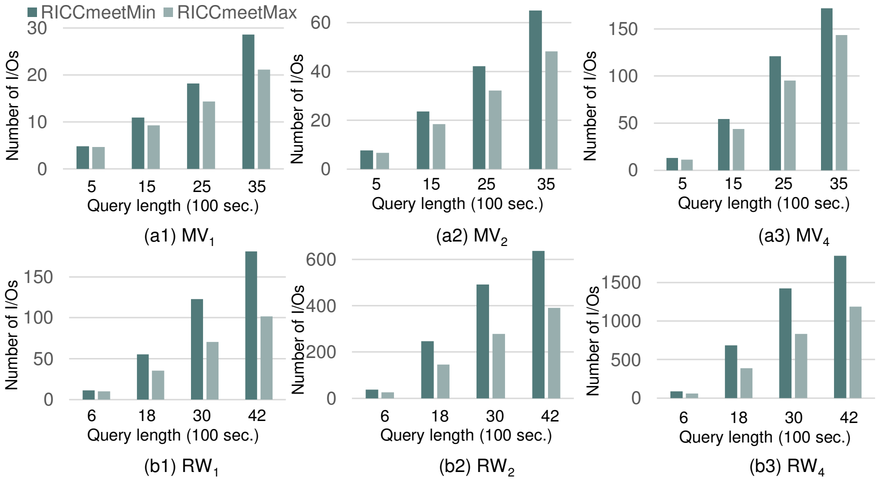


Figure 3.12: Varying query length

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 3.13 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.

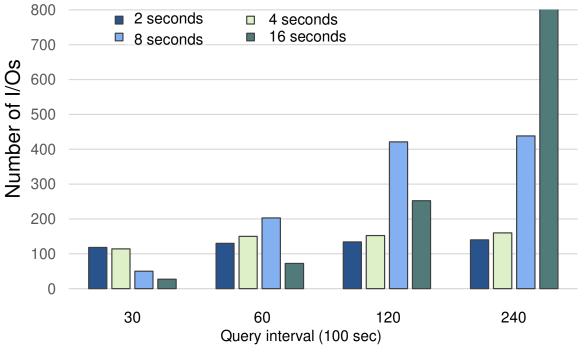


Figure 3.13: (a) Scaling, (b) Many-to-many queries: dataset RW_1, query length 4200 sec.

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

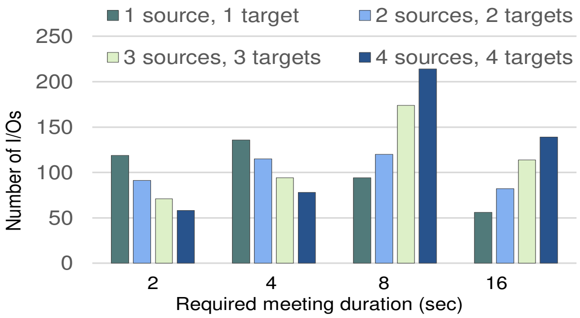


Figure 3.14: Many-to-many queries: dataset RW_1, query length 4200 sec

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 3.14 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).

3.6 Conclusions

In this chapter, 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.

Chapter 4

Answering Reachability Queries with Transfer Decay and Top-k Reachability Queries

4.1 Introduction

In the previous chapters, we discussed two types of spatiotemporal reachability problems with 'no instant exchange': reachability with processing delay and reachability with transfer delay. They were named $P\bar{T}$ and $\bar{P}T$ reachability respectively (see section 2.1). According to the $P\bar{T}$ reachability scenario, after two objects had a contact, the contacted object needs to spend some time to process the received information before it can redeliver it. In the $\bar{P}T$ reachability scenario, in order to transfer information, two objects (companions) are required to stay within the contact distance for some period of time (i.e.

to have a *meeting*).

While the problems that have been considered until this chapter covered different reachability scenarios, they had a common feature: the value of information carried by the source object (the object that initiated the information transmission process) and the value of information obtained by any reached object was assumed to remain unchanged. In the problem that we are going to introduce and address in this chapter, we remove this assumption, since it is not always valid. For example, if two people communicate over the phone (or a Bluetooth-enabled device), some information may be lost due to faulty connection.

We name a reachability problem, where the value of the transmitted item experiences a decay with each transfer, *the reachability with transfer decay.* This problem will still follow the reachability with transfer delay scenario. The formal definition of the new problem will be given in the next Section.

Another problem that we would like to present is a *top-k reachability problem with decay.* Consider a group of objects (people, cars, etc...), each of which possesses a different piece of information, and starts its transmission to other objects independently of each other. The objects that initiated the process form a set of source objects. Each of the source objects may carry information of a different value (and thus have a different weight), and during a contact, a decay of each piece of information may not be the same. As time progresses, any object may receive one or more items that originally came from different sources (possibly via other objects). It is reasonable to compute the combined weight of all the items collected by each object and rank the objects according to their aggregate weights. Those objects that aggregate most information may be of a special interest. A top-k spatiotemporal reachability query with decay asks to find the k objects with the highest aggregate weights.

The rest of the chapter is structured as follows: Section 4.2 gives a description of the problem of reachability with decay, as well as the top-k reachability with decay; Section 4.3 introduces our algorithm RICCdecay and describes its preprocessing phase, while Sections 4.4 and 4.5 present the query processing algorithms for the reachability with decay and the top-k reachability problems respectively. Section 4.6 provides the experimental evaluation of the proposed algorithms. Finally, Section 4.7 concludes the Chapter.

4.2 Problem Description

We define two novel spatiotemporal reachability problems: the problem of *reachability with decay* and its extension, the problem of *top-k reachability with decay*. Since these problems assume the reachability with transfer delay scenario as well, for completeness, we will restate some of the definitions that were given in Chapter 3.

4.2.1 Background

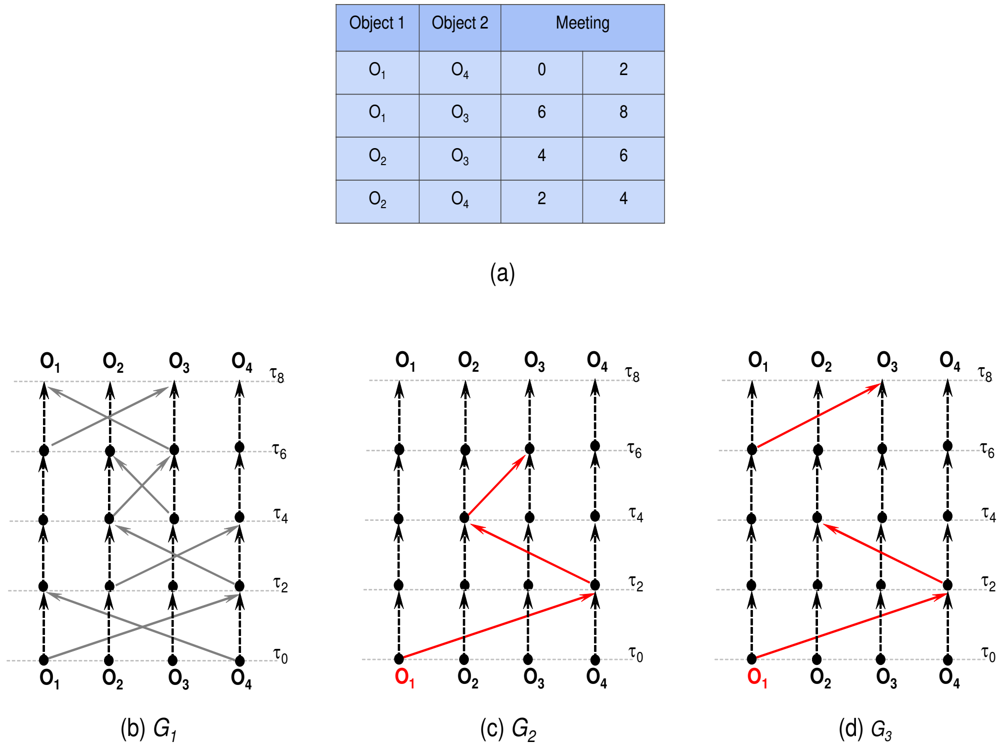
Let $O = \{O_1, O_2, ..., O_n\}$ be a set of moving objects, whose locations are recorded for a long period of time at discrete time instants $t_1, t_2, ..., t_i, ...$, with the time interval between consecutive location recordings $\Delta t = t_{k+1} - t_k$ $(k = 1, 2, ...)$ being constant. A *trajectory* 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. Two objects, O_i and O_j that at time t_k are respectively at positions

l_i and l_j, have a *contact* (denoted as $< O_i, O_j, t_k >$), 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.

The $\bar{P}T$ reachability scenario (reachability with transfer delay) requires to discretize the time interval between consecutive position readings $[t_k,\ t_{k+1})$ by dividing it into a series of non-overlapping subintervals $[\tau_0, \tau_1)$, ..., $[\tau_i,\ \tau_{i+1})$... , $[\tau_{r-1}, \tau_r)$ of equal duration $\Delta\tau = \tau_{i+1} - \tau_i$, such that $\tau_0 = t_k$ and $\tau_r = t_{k+1}$. We say that two objects, O_i and O_j, had a *meeting* $< O_i, O_j, I_m >$ 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 $\tau_k \in [\tau_s, \tau_f]$. The *duration* of this meeting is $m = \tau_f - \tau_s$. We call a meeting *valid* if its duration $m \geq m_q \Delta\tau$ (where m_q is the query specifies *required meeting duration* - time, needed for the objects to complete the exchange). Object O_T is considered to be *(m_q)-reachable* from object O_S during time interval $I = [\tau'_s, \tau'_f]$ if there exists a chain of subsequent valid 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 $j = 0, 1, ..., k-1$.

A reachability query determines whether object O_T (the target) is reachable from object O_S (the source) during time interval I.

Consider example in Figure 4.1. Table (a) shows the actual meetings between all objects during one time block. The meetings graph on this data is depicted in (b). A materialized reachability graph shows how the information is being dispersed considering that it starts with the source object and satisfies the m_q requirement. Suppose object O_1 is the source object and the required meeting duration $m_q = 2\Delta\tau$. Then graph in (c) is the materialized (m_q)-reachability graph for source object O_1 on data from (a). By looking at this graph, one can discover all objects that can be (m_q)-reached by object O_1 during the time interval $I = [\tau_0, \tau_8]$.



Object 1	Object 2	Meeting	
O_1	O_4	0	2
O_1	O_3	6	8
O_2	O_3	4	6
O_2	O_4	2	4

(a)

(b) G_1 (c) G_2 (d) G_3

Figure 4.1: (a) Record of meetings between objects O_1 - O_4; (b) graph G_1 is the meetings graph; (c) G_2 is the materialized reachability graph for 'transfer delay' scenario with the source object O_1 and $m_q = 2\Delta\tau$; (d) G_3 is the materialized reachability graph for 'transfer decay' scenario with the source object O_1, $m_q = 2\Delta\tau$, $d = 0.2$, $\nu = 0.6$. The time interval is $I = [\tau_0, \tau_8]$.

4.2.2 Reachability with Decay

In the reachability with transfer delay scenario, to complete the transfer, it is necessary for the objects to stay within the contact distance for a time interval that is at least as long as the *required meeting duration* m_q. In general, m_q may vary from one query to another depending on application. However, even if a meeting between objects O_i and O_j was long enough to satisfy the m_q requirement, under some circumstances, the transfer may still fail to occur, or the value of the transferred item may go down (e.g., a complete or partial signal loss during the communication). We propose to consider a new type of reachability scenario, namely *reachability with transfer decay* that accounts for such events.

Let d denote the *rate of transfer decay* - a part of information lost during one transfer ($d \in [0, 1)$). Then $p = 1 - d$ ($p \in (0, 1]$) will define the portion of the transfered information. Suppose, the weight of the item carried by a source object O_S is w. Then, during a valid meeting, O_S can transfer this item to some object O_i. However, considering the decay, if $d > 0$, the value of information, obtained by O_i lessens and becomes wp. With each further transfer, the value of the received item will continue to decrease. This process can be modeled with an exponential decay function.

We denote the number of transfers (hops), that is required to pass the information from object O_S to object O_i as h ($h \geq 0$). If object O_i cannot be reached by object O_S, $h = \infty$. Let $g_w : \mathbb{R} \rightarrow \mathbb{R}$ be a function that calculates the weight of an item after h transfers. Assuming that the transfer decay d and thus p are constant for the same item, $g_w(h)$ can be defined as follows:

$$g_w(h) = wp^h. \tag{4.1}$$

The number of transfers h in equation (4.1), that an item has to complete in order to be delivered from object O_S to object O_i, depends on the time τ_j when it is being evaluated, and thus denoted as $h(O_i^{\tau_j})$. Consider example in Figure 4.1. Suppose again that $m_q = 2\Delta\tau$ and object O_1 is the source object. It can reach object O_3 by $\tau = 6$ with 3 hops, while it requires only one hop for object O_1 to reach O_3 by $\tau = 8$. So, $h(O_3^{\tau_6}) = 3$ and $h(O_3^{\tau_8}) = 1$.

Note, that the scenario with $p = 1$ corresponds to the reachability with transfer delay problem described in [46]. If $p < 1$, with each transfer the value of $g_w(h)$ decreases exponentially. After a number of transfers, this value may become too small, and the user may decide to discard it. Let ν denote the *threshold weight.* If after some transfer, the weight of the item becomes smaller than the threshold weight ν, we disregard that event by assigning to the newly transferred item the weight of 0. We say, that h is the *allowed number of hops (transfers)* if it satisfies the threshold weight inequality

$$g_w(h) \geq \nu. \tag{4.2}$$

We denote the *maximum allowed number of transfers* that satisfies inequality (4.2) as h_{max}. Let function $f_w : \mathbb{R} \rightarrow \mathbb{R}$ be a function that assigns the weight to an item carried by object O_i at time τ_j, and denote it as $f_w(O_i^{(\tau_j)})$. (For brevity, we say 'the weight of object O_i at time τ_j'.) We define $f_w(O_i^{(\tau_j)})$ as follows:

$$f_w(O_i^{(\tau_j)}) = \begin{cases} g_w(h) & \text{if } h(O_i^{(\tau_j)}) \leq h_{max}, \\ 0 & \text{otherwise.} \end{cases} \tag{4.3}$$

The table in Figure 4.1 (a) shows the meetings between the objects O_1, O_2, O_3, and O_4 during the time interval $I = [\tau_0; \tau_8]$. For this example, we assume again that O_1 is the source object, $m_q = 2\Delta\tau$ and $d = 0.2$ (thus $p = 0.8$). To illustrate the difference between the actual weight of an item g_w and its assigned weight f_w, the values g_w, f_{w_1}, and f_{w_2} are computed for each object at time instants from τ_0 to τ_8 and recorded in the table (see Figure 4.2). The values for the assigned weight functions f_{w_1} and f_{w_2} are computed for $\nu = 0.6$ and $\nu = 0.7$ respectively. The graph G_3 in Figure 4.1(d) is constructed for f_{w_1}.

Time	Object Weight function	O_1	O_2	O_3	O_4
τ_0	g_w	1	0	0	0
	f_{w1}	1	0	0	0
	f_{w2}	1	0	0	0
τ_2	g_w	1	0	0	0.8
	f_{w1}	1	0	0	0.8
	f_{w2}	1	0	0	0.8
τ_4	g_w	1	0.64	0	0.8
	f_{w1}	1	0.64	0	0.8
	f_{w2}	1	0	0	0.8
τ_6	g_w	1	0.64	0.512	0.8
	f_{w1}	1	0.64	0	0.8
	f_{w2}	1	0	0	0.8
τ_8	g_w	1	0.64	0.8	0.8
	f_{w1}	1	0.64	0.8	0.8
	f_{w2}	1	0	0.8	0.8

Figure 4.2: The actual weight of an item g_w and its assigned weights f_{w_1} and f_{w_2}, calculated for objects O_1 - O_4 on data from Table 4.1(a), using object O_1 as the source object; $p = 0.8$, $\nu = 0.6$ for f_{w_1} and $\nu = 0.7$ for f_{w_2}.

We say that object O_T is *(m_q, d)-reachable* from object O_S during time interval $I = [\tau'_s, \tau'_f]$ if there exists a chain of subsequent valid and successful (under m_q, d conditions) 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, $\tau'_s \leq \tau_{s_0}$, $\tau_{f_k} \leq \tau'_f$, and $\tau_{s_{j+1}} \geq \tau_{f_j}$ for $j = 0, 1, ..., k-1$. The earliest time when O_T can be reached will be denoted as $\tau_R(O_T)$.

We assume that the values of d and ν are query specified. An *$(m_q$, $d)$-reachability* query Q_{md}: $\{O_S, O_T, w, d, I, m_q, \nu\}$ determines whether the target object O_T is reachable from the source object O_S, that caries an item whose weight is w, during time interval $I = [\tau_s, \tau_f]$, given required meeting duration m_q, rate of transfer decay d, and threshold weight ν, and reports the earliest time instant when O_T was reached.

4.2.3 Top-k Reachablility

We now consider the problem of top-k reachability with transfer decay. Let $S = \{O_{S_1}, O_{S_2}, ..., O_{S_q}\}$, $W = \{w_1, w_2, ..., w_q\}$, and $D = \{d_1, d_2, ..., d_q\}$ be the sets of source objects, weights, and decays respectively. Each object $O_{S_r} \in S$ carries a different piece of information (or physical item), whose weight is w_r, and is able to transfer this information to other objects following the (m_q, d)-reachability scenario described above. The transfer decay for the item carried by object O_{S_r} is d_r.

As the objects move through the network, source objects O_{S_r} encounter other objects, and may pass information to them. Since each source object owns a different piece of information, the transferred weight is going to differ not only depending on the number of hops, but also on the source that it came from.

Let the number of hops, that is required for object O_{S_r} to pass the information to object O_i be h_r ($h_r \geq 0$). Then we can calculate the *actual weight* of an item r after h_r transfers using equation (4.1) as

$$g_{w(r)}(h_r) = w_r p_r^{h_r},$$

where $r = (1, 2, ..., q)$. As in the previous problem, we require that each threshold weight inequality has been satisfied:

$$g_{w(r)}(h_r) \geq \nu$$

for $r = (1, 2, ..., q)$ and threshold weight ν.

Let $h_{max(r)}$ be the maximum allowed number of transfers that satisfies the inequality above for each $r = (1, 2, ..., q)$. Similarly to (4.3), function $f_{w(r)}$ assigns weight to the r^{th} item carried by object O_i at time τ_j (it will be denoted as $f_{w(r)}(O_i^{(\tau_j)})$. We define the *assigned weight* $f_w(O_i^{(\tau_j)})$ as follows:

$$f_{w(r)}(O_i^{(\tau_j)}) = \begin{cases} g_{w(r)}(h_r) & \text{if } h_r(O_i^{(\tau_j)}) \leq h_{max(r)}, \\ 0 & \text{otherwise.} \end{cases} \tag{4.4}$$

Furthermore, each object may receive more than one item. We denote the *aggregate weight* function $F_w : \mathbb{R} \to \mathbb{R}$ that assigns weight to the collection of items carried by object O_i at time τ_j as $F_w(O_i^{(\tau_j)})$, and define it as follows:

$$F_w(O_i^{(\tau_j)}) = \sum_{r=1}^{q} (f_{w(r)}(O_i^{(\tau_j)})), \tag{4.5}$$

where each $f_{w(r)}(O_i^{(\tau_j)})$ is computed as in (4.4).

Table 4.1: Notation used in the chapter

Notation	Definition
$\Delta\tau$	Duration between two consecutive time instants
Δt	Duration between two consecutive reporting times
m_q, μ	Required meeting duration and minimum meeting duration
O_S, O_T	A source and a target objects
$O_i^{(\tau_j)}$	Instance of object O_i at time τ_j
$\tau_R(O_i)$	Earliest time when object O_i was reached
d, p	Transfer decay and portion of transfered information
h, h_{max}	Actual and maximum allowed number of hops (transfers)
ν	Threshold weight
$g_w(h)$	Actual weight of an item after h transfers
$f_w(O_i^{(\tau_j)})$	Weight, assigned to an item carried by $O_i^{(\tau_j)}$ considering ν
$F_w(O_i^{(\tau_j)})$	Assigned aggregate weight of all items carried by $O_i^{(\tau_j)}$
B_k, I_k	Time block k that spans time interval I_k
C, H	Contraction parameter and grid resolution

A *top-k reachability with decay* query Q_{topK} is given in the form $\{S, W, D, I, m_q, \nu, k\}$. The goal of Q_{topK} is to find k objects with the highest aggregate weight F_w (computed according to 4.5), that was obtained during the time interval I.

4.3 Preprocessing

As with other reachability problems discussed above, there are two naive approaches to solve (m_q, d)-reachability problem: (i) 'no-preprocessing', and (ii)'precompute all'. Neither one of them is feasible for large graphs: the first does not involve any preprocessing, and thus too slow during the query processing, while the second requires too much time for preprocessing and too much space for storing the preprocessed data. To overcome the disadvantages of the second approach and still achieve fast query processing, we precompute and store only some data. We follow with the description of the preprocessing, and later, in Sections 4.4 and 4.5, describe the query processing.

In order to simplify the presentation, we assume that the minimum meeting duration μ ($\mu \leq m_q$), that may be required for a transfer by any application, is known before the preprocessing, and set $m_q = \mu$, thus fixing it. However, the proposed algorithm can be extended to work with any query specified m_q (as opposed to $m_q = \mu$) by combining it with RICCmeetMax that was described in Chapter 3 and in [46].

Suppose, our spatiotemporal datasets contain records of objects' locations in the form $(t, object\_id, location)$, ordered by the location reporting time t. Similar to the previous RICC algorithms, we start the preprocessing by dividing the time domain into a non-overlapping time intervals of equal duration (*time blocks*). Each time block (denoted as B_k) contains all records whose reporting times belong to the corresponding time period. The number of the reporting times in each block is the *contraction parameter* C. How to find an optimal value of C will be discussed in Section 4.6.

For each time block, during the preprocessing stage of the algorithm, the following steps have to be completed: (i) computing candidate contacts, (ii) verifying contacts (has to be performed for each t_k), (iii) identifying meetings, (iv) computing reachability, and (v) constructing index. Steps (i), (ii), (iii), and (v) are similar to those in Chapter 3, which contains a detailed description of them. Thus we go over these parts briefly, and concentrate on step (iv), computing reachability, which is the central and most difficult step of the preprocessing.

During preprocessing, information regarding each object O_i is saved in a data structure named *objectRecord*(O_i), which is created at the beginning of each time block B_k and deleted after all the needed information is written on the disk at the end of B_k. *ObjectRecord*(O_i) has the following fields: *Object_id*, *Cell_id* (the object's placement in the grid with side H when it was first seen during B_k), $ContactsRec$ (a list of the contacts of O_i during B_k), $MeetingsRec$ (a list of meetings of O_i during B_k). The grid side H is another parameter (in addition to the contraction parameter C), which needs to be optimized. We will discuss this question in Section 4.6. Also, area partitioning is performed into cells with side H at the beginning of each time block, and into cells with side d_{cc} at each t_k, which is described in detail in Section 3.3. In addition, for each time block we maintain a hashing scheme, that enables to access each object's information by the object's id.

4.3.1 Computing Contacts and Identifying Meetings

Two objects O_i and O_j are *candidate contacts* at reporting time t_k if the distance between them at that time is no greater than *candidate contact distance* $d_{cc} = 2d_{max} + d_{cont}$ (where d_{max} is the largest distance that can be covered by any object during Δt). Candidate contact objects can potentially have a contact between t_k and t_{k+1}. In order to force all candidate contacts of a given object O_i to be in the same or neighboring with O_i's cells, at each t_k we partition the area covered by the dataset into cells with side d_{cc}. Now, to find all candidate contacts of object O_i, we only need to compute the (Euclidean) distance between O_i and objects in the same and neighboring cells. (The distance between any pair of candidate contacts needs to be computed only once.)

Using our assumption that between consecutive reporting times objects move linearly, at t_{k+1}, we can verify if there were indeed any contacts between each pair of candidate contacts during the time interval $[t_k, t_{k+1})$. If a contact occurred, it is saved in the list $ContactsRec$ of $objectRecord$ of each contacted object.

If an object O_i had O_j for its contact at two or more consecutive time instants, these contacts are merged into a meeting, and written in the $MeetingsRec$ list of (O_i). This process continues until we process the time block, at which point the meeting durations are computed. At the end of the block all meetings with duration $m < \mu$ (with the exception of boundary meetings) are pruned, while all the remaining meetings are recorded into file *Meetings*. Boundary meetings (meetings that either start at the beginning or finish at the end of B_k) are recorded regardless of their duration since they may span more than one block, which needs to be verified during the query processing.

4.3.2 Computing Reachability

To speed up the query time, during the preprocessing stage, for each object O_i (which is active during the given time block B_k), we would like to precompute all objects that are (μ, d)-reachable from O_i during B_k. Here we are facing a challenge: to find, which objects can be (μ, d)-reached by O_i, we need to know the transfer decay d and weight threshold ν, which are assumed to be unknown at the preprocessing time.

To overcome an issue of unknown d and ν, we turn our problem of reachability with decay into *hop-reachability* problem. Recall that one of the requirements for object O_T to be reachable from object O_S is that each meeting in the chain of meetings from O_S to O_T has to be a *successful* meeting.

It follows from 4.2, that after each meeting, for each companion object O_i, the following condition must hold:

$$g_w(h) = wp^h \geq \nu.$$

Thus, the allowed number of transfers (or hops) h for a successful meeting should satisfy the following inequality:

$$h \leq \log_p \frac{\nu}{w},$$

and finally

$$h_{max} = \lfloor \log_p \frac{\nu}{w} \rfloor. \tag{4.6}$$

Now the problem can be stated as follows: for each object O_i, compute all objects, that are (μ, h_{max})-reachable from O_i. In other words, we aim to discover all objects that can be reached by O_i within h_{max} transfers, under the required meeting duration $m_q = \mu$. Moreover, for each object O_j reached by O_i, we would like to find the minimum number of such transfers $h_{min} \leq h_{max}$.

Our algorithm makes use of plane sweep algorithm, where an imaginary vertical line sweeps the xy-plane, left-to-right, stopping at some points, where information needs to be analyzed. In our case, the x-dimension is the time-dimension, and y-dimension is the order in which the meeting are discovered.

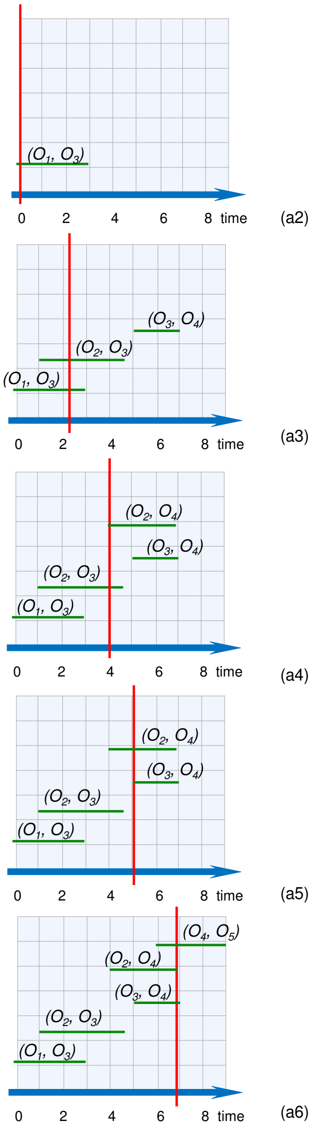
We demonstrate how the algorithm works on the Example in Figure 4.3, and later provide a pseudo-code and detailed explanation. Consider the data in the table (a1). It contains records of all actual meetings between all objects during one time block. Figures (a2)-(a6) describe how reached objects and meetings are been discovered. The information about the 'reachability' status of each object is being recorded into a temporary table, which is created at the beginning of each block. A row is added to the table for each reached object at the time when it is reached, and it is updated with any new event. The development of the reachability table is shown in Figures (b1)-(b6).

We show how to compute all objects that are reached by object O_1 during the given time block, assuming that $\mu = 2\Delta\tau$. At the beginning of the block, the sweep line is positioned at $\tau = 0$, and only object O_1 is reached (with $h_{min} = 0$), which is recorded in table (b1). During the given time block, O_1 has only one meeting, $< O_1, O_3, [0, 3] >$ which is placed on the plane (a2). As a result of this meeting, object O_3 is reached at time $\tau = 2$, with the minimum hop-value $h_{min} = 1$, which is recorded in the table (b2). The sweep line moves to the time $\tau = 2$ - time, when object O_3 was reached. Next, all meetings of O_3 that are either active at $\tau = 2$ or start after this time, are materialized. These are meetings $< O_3, O_2, [1, 5] >$ and $< O_3, O_4, [5, 7] >$. Consider the first meeting: $< O_3, O_2, [1, 5] >$. Even though it begins at $\tau = 1$, the retransmission does not start until $\tau = 2$, since only at this time O_3 becomes reached. As a result of these two meeting with object O_3, O_2 and O_4 become reached at $\tau = 4$ and $\tau = 7$ respectively, with $h_{min} = 2$ ((a3), (b3)). The line changes its position to $\tau = 4$. This process continues until the sweep line reaches the end of the time block. Note that the earliest reached time for an object may change, also an object's h_{min} value may decrease with time. For example, object O_4 was reached by O_2 with $h_{min} = 3$ at $\tau = 6$ ((a4), (b4)), however as a result of the meeting with object O_3, its h_{min} value went down to $h_{min} = 2$ at $\tau = 7$ ((a3), (b3)).

The process for computing all objects that are (h_{min})-reachable by O_S during one time block is generalized in Algorithm 4. Procedure UpdateHmin initializes and then updates the table that records the reachability status of each reached object. The $S_{ReachHop}$

Object 1	Object 2	Meeting started	Meeting finished
O_1	O_3	0	3
O_2	O_3	1	5
O_2	O_4	4	7
O_2	O_5	2	4
O_3	O_4	5	7
O_4	O_5	6	9

(a1)



(a2)

(a3)

(a4)

(a5)

(a6)

Time / Obj.	0	1	2	3	4	5	6	7	8	9
O_1	0	0	0	0	0	0	0	0	0	0
O_2										
O_3										
O_4										
O_5										

(b1)

Time / Obj.	0	1	2	3	4	5	6	7	8	9
O_1	0	0	0	0	0	0	0	0	0	0
O_2										
O_3			1	1	1	1	1	1	1	1
O_4										
O_5										

(b2)

Time / Obj.	0	1	2	3	4	5	6	7	8	9
O_1	0	0	0	0	0	0	0	0	0	0
O_2					2	2	2	2	2	2
O_3			1	1	1	1	1	1	1	1
O_4								2	2	2
O_5										

(b3)

Time / Obj.	0	1	2	3	4	5	6	7	8	9
O_1	0	0	0	0	0	0	0	0	0	0
O_2					2	2	2	2	2	2
O_3			1	1	1	1	1	1	1	1
O_4							3	2	2	2
O_5										

(b4)

Time / Obj.	0	1	2	3	4	5	6	7	8	9
O_1	0	0	0	0	0	0	0	0	0	0
O_2					2	2	2	2	2	2
O_3			1	1	1	1	1	1	1	1
O_4							3	2	2	2
O_5										

(b5)

Time / Obj.	0	1	2	3	4	5	6	7	8	9
O_1	0	0	0	0	0	0	0	0	0	0
O_2					2	2	2	2	2	2
O_3			1	1	1	1	1	1	1	1
O_4							3	2	2	2
O_5										3

(b6)

Figure 4.3: Computing all (h_{min})-reachable objects from O_1 ($\mu = 2$).

Algorithm 4 Reach(h_{min})

```
Input: $O_S$
procedure UpdateHmin ($O_i, \tau_s, \tau_f, h$)
    for each $\tau_k \in [\tau_s, \tau_f]$ do $h_{min}(O_i^{\tau_k}) = h$
for each $O_i$ do
    $\tau_R(O_i) = \infty$
    UpdateHmin($O_i, \tau_0, \tau_{end}, \infty$)        ▷ $\tau_0$ and $\tau_{end}$ are the first and last time units of a block
procedure ReachHop($O_S$)
    $time = 0$, $\tau_R(O_S) = 0$, UpdateHmin($O_S, \tau_0, \tau_{end}, 0$), $S_{PQ} = \{O_S\}$, $S_{ReachHop} = \{\emptyset\}$
    while (($S_{PQ}$) $\neq \{\emptyset\}$ and time $\leq \tau_{end}$) do
        $O_i = ExtractMin\ (S_{PQ})$
        $S_{ReachHop} = S_{ReachHop} \cup O_i$, $time = \tau_R\ (O_i)$
        for each $O_j$ that had a valid meeting with $O_i$ do
            if $O_j \notin S_{ReachHop}$ then
                $\tau_{Rnew}(O_j) = \infty$
                while $\tau_{Rnew}(O_j) \geq \tau_R(O_j)$ do
                    read next meeting $M_{ij} =< O_i, O_j, [\tau_s, \tau_f] >$
                    compute $\tau_{Rnew}(O_j)$
                    if $\tau_{Rnew}(O_j) < \tau_R(O_j)$ then
                        $Update\ (S_{PQ}, O_j),\ h = h_{min}(O_i^{time}) + 1$
                        if $\tau_R(O_j) = \infty$ then $\tau_R(O_j) = \tau_{end+1}$
                        UpdateHmin($O_j, \tau_{Rnew}, \tau_R(O_j) - 1, h$))
                    if ($M_{ij} = last\ meeting < O_i, O_j >$ in $B_k$) then
                        $\tau_{Rnew}(O_j) = -1$
return $S_{Reached}$
```

set keeps all objects for which all h_{min} values as well as the earliest reached time had been computed and finalized. Those objects that were found to be reached, but not in $S_{ReachHop}$ yet, are placed in the priority queue S_{PQ}, where priority to the objects is given according to their 'reached' times. When an object (say object O_i) that has the earliest reached time ($\tau_R(O_i)$) is extracted from S_{PQ}, it is placed into $S_{ReachHop}$ (lines 10, 11). At this time, all meetings of objects that can be reached by O_i (but not in $S_{ReachHop}$) are analyzed (lines 13 - 23). As a result, both $\tau_R(O_j)$ (and their priority in S_{PQ}) as well as their h_{min} values can be changed (lines 19 and 21). This algorithm has to be performed for each object of the dataset that is active during the given time block.

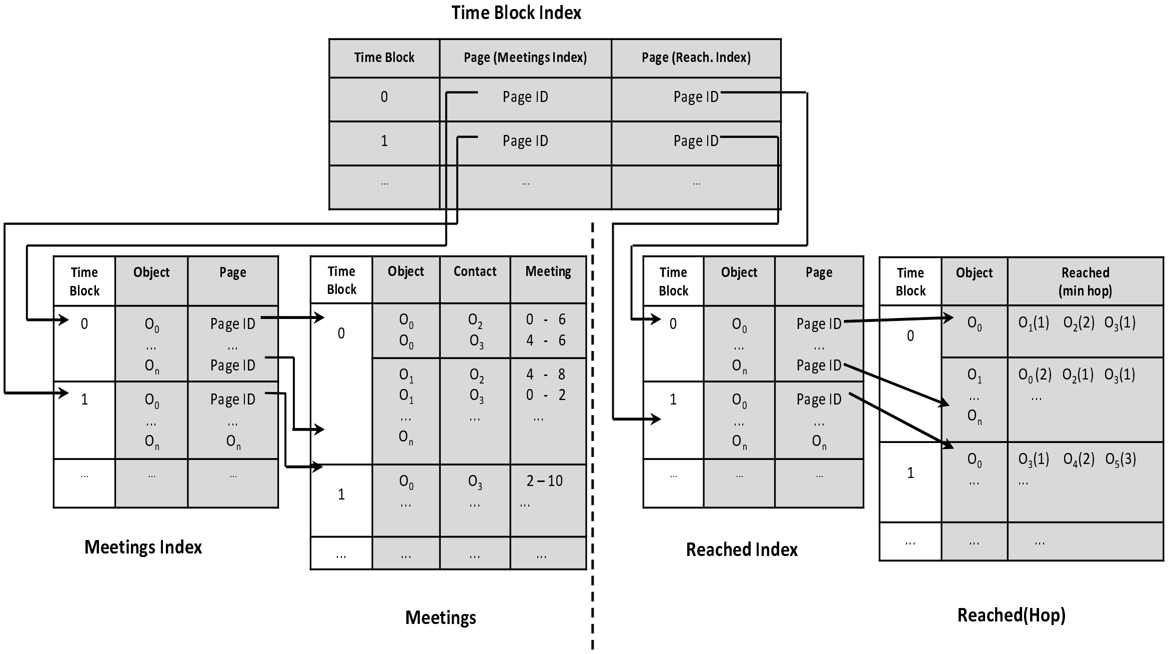


Figure 4.4: Two-level index on files Meetings and Reached(Hop).

4.3.3 Index Construction

The index structure of RICCdecay is similar to the one of RICCmeet algorithms: in order to enable an efficient search of information in the files Meetings and Reached(Hop) during the query processing, we create three index files: *Meetings Index*, *Reached Index*, and *Time Block Index* in addition to the files Meetings and ReachedHop (Figure 4.4). The records in *Meetings Index* are organized and follow the order of time blocks. Each record contains an object id and a pointer to the page with the first record for this object (for the given time block) in file *Meetings*. In the *Reached Index*, 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 *Reached(Hop)*. Each record in *Time Block Index* points to the beginning of a time block in *Meetings Index* and *Reached Index*.

4.4 Reachability Queries with Decay: Query Processing

The reachability with decay query Q_{md} is issued in the form Q_{md}:$\{O_S, O_T, w, d, [\tau_s, \tau_f], \mu, \nu\}$. (Recall that during the preprocessing, for simplicity, we set $m_q = \mu$.) First, using equation 4.6, we rewrite the problem as hop-reachability problem, replacing three parameters, w, d, and ν from Q_{md} with h_{max}. Now the new query can be written as Q_{mh}: $\{O_S, O_T, h_{max}, [\tau_s, \tau_f], \mu\}$.

The processing of Q_{mh} starts from computing the time blocks B_s, ... , B_f that contain data for the query interval $I = [\tau_s, \tau_f]$. File *Time Block Index* (accessed only once per query) points to the pages in the *Meetings Index* and *Reached Index* that correspond to the required blocks. The last two index files (accessed once per time block) in turn point to the appropriate pages in files *Meetings* and *Reached(Hop)* respectively.

The set of reached objects $S'_{reached}$ is initialized with object O_S at the beginning of the query processing. We start reading file *Reached(Hop)* from block B_s, retrieving all records for object O_S. Recall that in *Reached(Hop)* every object O_j that can be reached by object O_i is recorded together with the smallest number of transfers h_{min} that is required for O_i to reach O_j. Thus during the query processing, an object O_j cannot be considered as reached during the block B_k unless $h_{min}(O_j^{B_k}) \leq h_{max}$ (where $h_{min}(O_j^{B_k})$ is the value h_{min} of object O_j at the end of B_k). So, each objects O_j that was found to be reached by O_S (a companion of O_S), is added to $S'_{reached}$, along with the corresponding number of hops h_{min}, provided that $h_{min}(O_j^{B_s}) \leq h_{max}$. Next, we proceed to block B_{s+1}. This time, retrieving all the companions of each object from $S'_{reached}$ and updating it by either adding new objects or adjusting the h_{min} value for the objects that are already in the set. Such adjustment may be needed if, for some object $O_i \in S'_{reached}$, $h_{min}(O_j^{B_s}) > h_{min}(O_j^{B_{s+1}})$. The process continues until the target object O_T is added to $S'_{reached}$ while reading some block $B_i (i < f)$ or the last block B_f is reached.

If at the end of processing B_f, $S'_{reached}$ does not contain the target object O_T, the query processing can be aborted, otherwise it moves to the file *Meetings*. Now the process of identifying reached objects inside each block is the same as the one described in Algorithm 4. If there is a meeting between objects O_i and O_j, that ends at the end of the time block, but is shorter than m_q, we check if it continues in the next block, and merge two meetings into one if needed. Also, if object O_i was reached by the source object O_S during the block B_k with $h_{min}(O_i^{B_k}) = h_1$, and in a later block B_m, object O_j was reached by O_i within h_2 hops, $h_{min}(O_j^{B_m}) = h_1 + h_2$. Object O_j is considered to be reached by O_S if $h_{min}(O_j^{B_m}) \leq h_{max}$.

If by the end of B_i, O_T was not found to be reached, and $B_i < B_f$, the search will switch to file *Reached(Hop)*. 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.

4.5 Top-k Reachability: Query Processing

To process top-k reachability queries efficiently, we will use the preprocessed data and index structure from RICCdecay, described in the previous section. For that reason, we named our top-k reachability query processing algorithm *RICCtopK*. The top-k query Q_{topK} is issued in the form $\{S, W, D, [\tau_s, \tau_f], \mu, \nu, k\}$, where $S = \{O_{S_1}, O_{S_2}, ..., O_{S_q}\}$, $W = \{w_1, w_2, ..., w_q\}$, and $D = \{d_1, d_2, ..., d_q\}$ are the sets of source objects, weights, and decays respectively. To make use of the precomputed data from RICCdecay, the top-k reachability with decay problem has to be translated into top-k hop-reachability problem. To achieve this, for each source object $O_{S_r} \in S$, we compute the maximum number of allowed transfers (hops) $h_{max(r)}$ applying inequality (4.6) to each triple $\{O_{S_r}, w_r, d_r\}$ as follows:

$$h_{max(r)} = \lfloor \log_{p_r} \frac{\nu}{w_r} \rfloor,$$

where $p_r = 1 - d_r$, and $r = \{1, 2, ...q\}$.

Now each top-k query can be thought of as written in the form $\{S, H_{hop}, [\tau_s, \tau_f], \mu, \nu, k\}$, where $H_{hop} = \{h_{max(1)}, h_{max(2)}, ..., h_{max(q)}\}$. Note that the top-k query processing is the extension of the reachability with decay query processing algorithm, and thus we will avoid repeating some details concerning the use of the index structure during the query processing that were described earlier in section 4.4.

First, the set of *Top-k Candidates* is initialized by adding to it all source objects. We start reading file *Reached(Hop)* from time block B_s, checking all records for each source object from set S (in order of their appearance in the file). Once an object, that was reached by at least one source object, is discovered, it is added to *Top-k Candidates.* For each top-k candidate O_i, we keep the information about the source object(s), that it was reached by, as well as the smallest number of hops $h_{min(r)}$ required to transfer information from each source to O_i. The search continues in this manner until time block B_f is processed, after which the weight of each object from *Top-k Candidates* is computed. Note, that this is not the actual weight F_w of an object, but the maximum weight F_{max} that this object may receive.

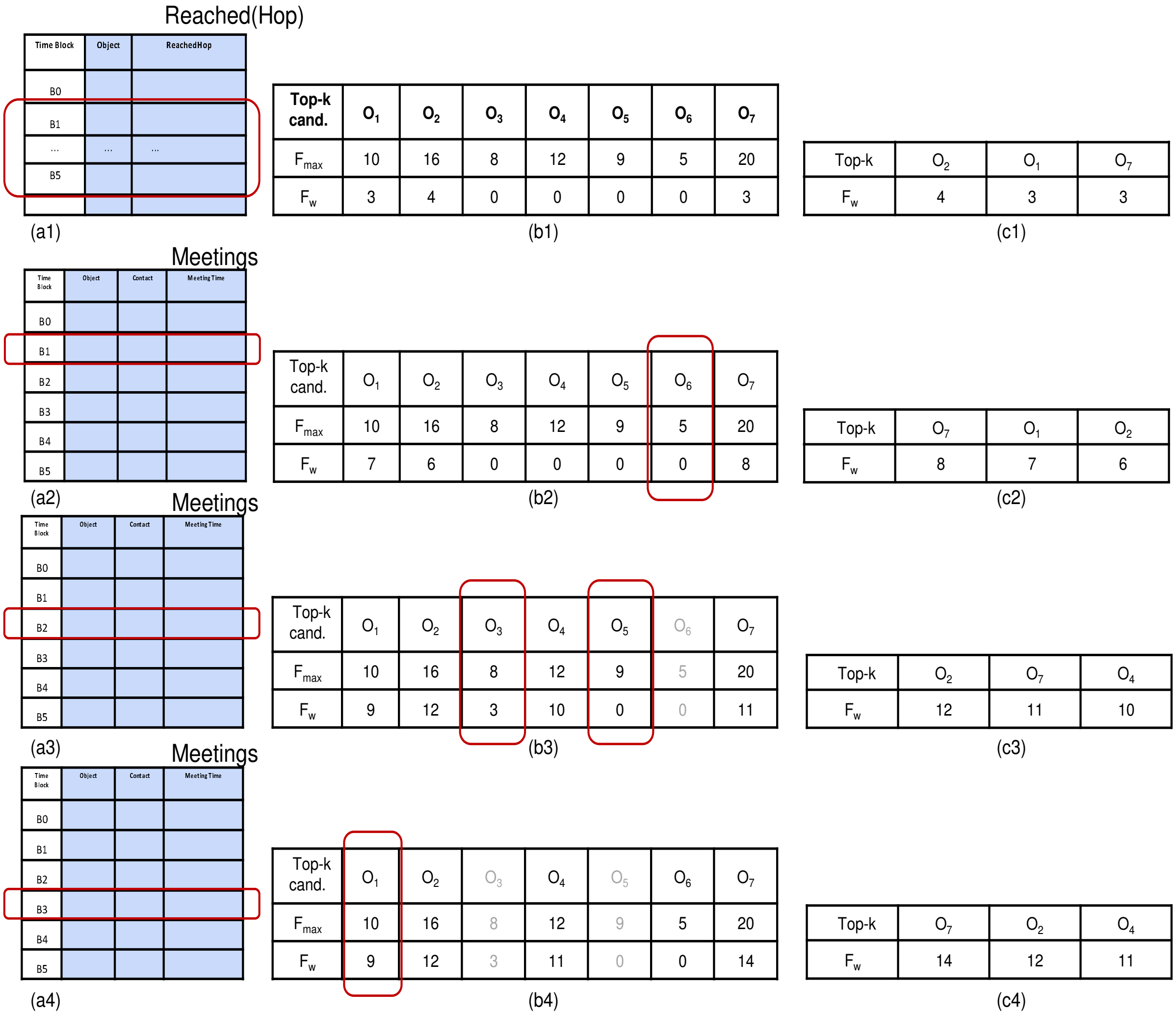
Next, the query processing moves to the file *Meetings.* Here, the algorithm maintains two structures: *Top-k Candidates* and *Top-k*, that have to be updated at the end of each block. *Top-k Candidates* contains: (i) the ids of all reached objects, (ii) their corresponding maximum weights F_{max} (both, (i) and (ii), were computed in the previous step), as well as (iii) the current weight F_w of each candidate top-k object. At the beginning, the weight F_w of each source object O_{S_r} is set to its initial weight w_r, while the rest of the objects' weights F_w are set to 0 (since these objects have not been seen in file *Meetings* yet). *Top-k* is initialized by adding to it k source objects from set S with the top k weights; the weight F_w of each top-k object is recorded as well.

Let us denote the lowest weight F_w among the objects in *Top-k* as $F_w min$. If *Top-k* contains k objects, and the object with the smallest value carries weight $F_w min$, then any object O_i, such that $F_{max}(O_i) < F_w min$, cannot be among the top-k.

In file *Meetings*, the query processing starts from time block B_s. After one time block is processed, the aggregate weight F_w of objects from *Top-k Candidates* that were involved in some transfers, may increase, and has to be updated. This may lead to changes in *Top-k*. After *Top-k* and $F_w min$ are updated, all objects O_i from *Top-k Candidates*, such that $F_{max}(O_i) < F_w min$, can be removed from the set of candidates. When the work on block B_s is completed, we proceed to the next block. This process continues until either the last time block B_f of the query is reached or the size of *Top-k Candidates* is reduced to the size of *Top-k*. In either case, the final state of *Top-k* answers the query.

For example, consider the top-k query with three source object O_1, O_2, and O_7, whose corresponding weights are 3, 4, and 3. Suppose, the query interval $[\tau_s, \tau_f]$ is contained in time blocks B_1 - B_5. Figure 4.5 illustrates the example. Figures (a1)-(a4) show the time blocks in files *Reached(Hop)* and *Meetings* that are being processed at the given stage, tables (b1)-(b4) display the *Top-k Candidates* with their maximum possible aggregate weights F_{max} and current aggregate weights F_w. The last column of tables, (c1)-(c4), keeps track of the current state of the *Top-k* set. Both, *Top-k Candidates* and *Top-k* are created after *Reached(Hop)* is processed and updated after the corresponding time block of file *Meetings* is processed.

The query answering begins in *Reached(Hop)*. The relevant data is read from blocks $B1$ - $B5$, and by the end of $B5$, the superset of all objects that can be reached by the source object is identified. These objects are *Top-k Candidates*. They are recorded in the *Top-k Candidates* table, together with their maximum possible aggregate weight F_{max}



Reached(Hop)

Time Block	Object	ReachedHop
B0		
B1		
...	...	...
B5		

(a1)

Top-k cand.	**O_1**	**O_2**	**O_3**	**O_4**	**O_5**	**O_6**	**O_7**
F_{max}	10	16	8	12	9	5	20
F_w	3	4	0	0	0	0	3

(b1)

Top-k	O_2	O_1	O_7
F_w	4	3	3

(c1)

Meetings

Time Block	Object	Contact	Meeting Time
B0			
B1			
B2			
B3			
B4			
B5			

(a2)

Top-k cand.	O_1	O_2	O_3	O_4	O_5	O_6	O_7
F_{max}	10	16	8	12	9	5	20
F_w	7	6	0	0	0	0	8

(b2)

Top-k	O_7	O_1	O_2
F_w	8	7	6

(c2)

Meetings

Time Block	Object	Contact	Meeting Time
B0			
B1			
B2			
B3			
B4			
B5			

(a3)

Top-k cand.	O_1	O_2	O_3	O_4	O_5	O_6	O_7
F_{max}	10	16	8	12	9	5	20
F_w	9	12	3	10	0	0	11

(b3)

Top-k	O_2	O_7	O_4
F_w	12	11	10

(c3)

Meetings

Time Block	Object	Contact	Meeting Time
B0			
B1			
B2			
B3			
B4			
B5			

(a4)

Top-k cand.	O_1	O_2	O_3	O_4	O_5	O_6	O_7
F_{max}	10	16	8	12	9	5	20
F_w	9	12	3	11	0	0	14

(b4)

Top-k	O_7	O_2	O_4
F_w	14	12	11

(c4)

Figure 4.5: Top-K Query Processing (source objects: O_1, O_2, O_7)

(b1). Since at this stage the aggregate weight F_w is known only for the source objects, the objects O_1, O_2, and O_7 are placed in the *Top-k* (c1). The query processing moves to time block $B1$ in file *Meetings* (a2). At the end of $B1$, the aggregate weight of some objects F_w is updated, and thus both, *Top-k Candidates* and *Top-k* are updated as well ((b2), (c2)). We notice that the lowest weight of the top-k object O_2 $F_w min(O_2) = 6$. Thus all objects O_i with maximum weight $F_{max}(O_i) < F_w min(O_2)$ can be removed from the set of candidates. (Such objects are shown in gray in (b3) and (b4).) The next block to process is $B2$ (a3), and after updating both tables ((b3) and (c3)), we see that objects O_3 and O_5 can be excluded from further consideration. After processing the next block, $B3$, we remove object O_1 from *Top-k Candidates*. Even though, the query interval ends only in $B5$, we can suspend the query as the size of *Top-k Candidates* is reduced to the size of *Top-k*. (In case, if the *Top-k* is required to be answered in the order of object's weights, the remaining blocks will have to be processed as well.)

4.6 Experimental Evaluation

In this section, we describe the results of the experimental evaluation of our algorithms RICCdecay and RICCtopK. Since there are no other algorithms for processing spatiotemporal reachability queries with decay, we modified the RICCmeetMin algorithm, which is presented in Chapter 3, to enable it to answer such queries. We compare the performance of our new RICCdecay and RICCtopK algorithms with that of RICCmeetMin.

All the experiments were performed on a system running Linux with a 3.4GHz Intel CPU, 16 GB RAM, 3TB disk and 4K page size. All programs were written on C++

and compiled using gcc version 4.8.5 with optimization level 3.

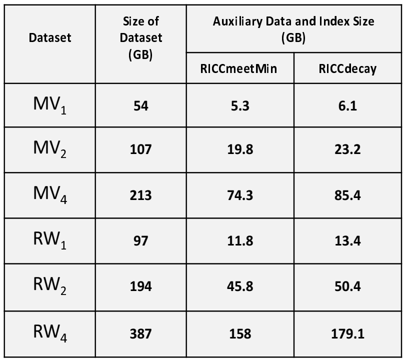
4.6.1 Datasets

All experiments were performed on six realistic datasets of two types: Moving Vehicles and Random Walk. The first three datasets, Moving Vehicles (MV) were created by the Brinkhoff data generator [6], which generates traces of objects, moving on real road networks. For the underlying network in these experiments we chose the San Francisco Bay road network, which covers an area of about 30000 km^2. These sets contain information about $1000, 2000$, and 4000 vehicles respectively (denoted as MV_1, MV_2, and MV_4). The location of each vehicle is recorded every $\Delta t = 5$ seconds during 4 months, which results in $2,040,000$ records for each object. The size of each dataset (in GB) appears in Table 4.2. For the experiments on these sets, the contact distance d_{cont} is assumed to be equal to 100 meters (for a (class 1) Bluetooth connection).

For the three Random Walk datasets (RW), we created our own generator, which utilizes the modified random waypoint model [31], and is frequently used for modeling movements of mobile users. In our model, 90% of individuals are moving, while the remaining 10% are stationary. At the beginning of the first trip, each user chooses whether to move or not (in the ratio of $9:1$). Further, each out of 90% moving users chooses the direction, speed (between $1.5m/s$ and $4m/s$), and duration of the next trip, and then completes it. At the end of each trip, each person determines the parameters for the next trip, and so on. Random Walk datasets consist of trajectories of $10000, 20000$, and 40000 individuals respectively (denoted as RW_1, RW_2, and RW_4). Each set covers an area of 100 km^2. The location of each user is recorded every $\Delta t = 6$ sec for a period of one month (or 432,000 records for each person). For the Random Walk datasets, we set the contact distance equal to 10 meters (the range of a small personal (class 3) Bluetooth-enabled device).

The performance of the algorithms 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 (20 sequential I/Os take the same time as 1 random). For all our experiments, we present the equivalent number of random I/Os using this ratio.

Table 4.2: Size of datasets, auxiliary files and indexes



Dataset	Size of Dataset (GB)	Auxiliary Data and Index Size (GB)	
		RICCmeetMin	RICCdecay
MV_1	54	5.3	6.1
MV_2	107	19.8	23.2
MV_4	213	74.3	85.4
RW_1	97	11.8	13.4
RW_2	194	45.8	50.4
RW_4	387	158	179.1

4.6.2 Parameter Optimization

The values of the contraction parameter C and the grid resolution H, that are used for the preprocessing, depend on the datasets. For each dataset, the parameters C and H were tuned on the 5% subset as follows. We performed the preprocessing of this subset for different values of (C, H), and tested the performance of RICCdecay algorithm on a set of 200 queries. The length of each query was picked uniformly at random between 500 and 3500 sec for the Moving Vehicles datasets, and between 600 and 4200 sec for the Random

Walk datasets. The maximum allowed number of transfers h_{max} was picked uniformly at random from 1 to 4. The parameters C and H were varied as follows: grid resolution H - from 500 to 40000 meters for Moving Vehicles datasets, and from 250 to 2000 meters for Random Walk datasets; contraction parameter C - from 0.5 to 30 min. For each dataset, the pair of parameters (C, H) that minimized the number of I/Os was used for the rest of the experiments. For example, for MV_1 we used $H = 20000$ meters and $C = 14$ min, while for RW_4 we used $H = 500$ meters and $C = 2$ min.

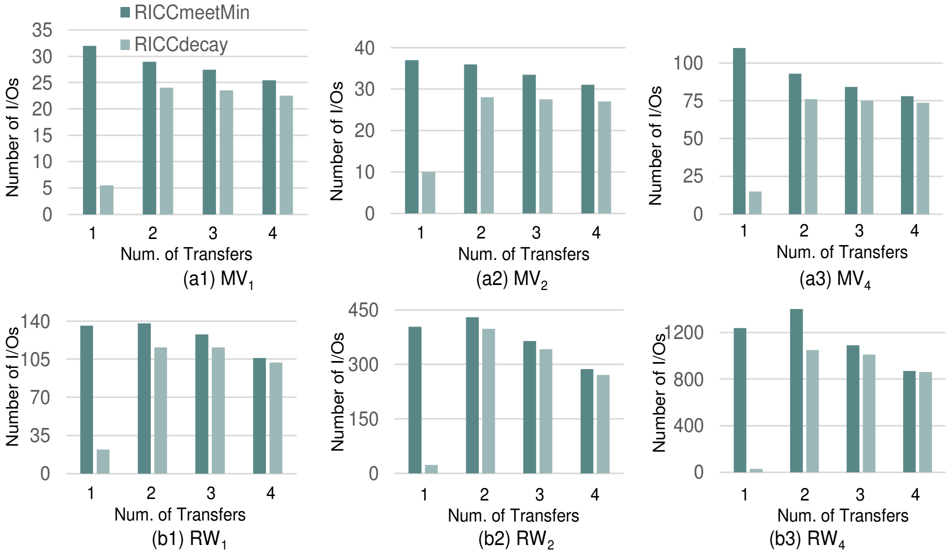
4.6.3 Preprocessing Space and Time

The sizes of the auxiliary files as well as the index sizes for the two algorithms, RICCmeetMin and RICCdecay, appear in Table 4.2. RICCdecay uses about 13.5% more space compare to RICCmeetMin, since it records more information into the file *Reached(Hop)* (For each reached object, in addition to its id, it saves its hop value as well.). The time needed to preprocess one hour of data for RICCdecay ranges from 14 sec for MV_1 to 91 min for RW_4. For comparison, the preprocessing time for RICCmeetMin ranges from 13 sec for MV_1 to 56 min for RW_4.

4.6.4 Query Processing

The performance of RICCdecay algorithm was tested on sets of 100 queries of different time intervals, ranging from 500 to 3500 sec for the Moving Vehicles datasets, and from 600 to 4200 sec for the Random Walk datasets. In addition, testing was done on various maximum allowed number of hop values: $h_{max} = 1, 2, 3, 4$. The minimum meeting duration μ was set to 2 sec, and the initial weight w of the item carried by the source object

O_S was set to 1 for all the experiments.



(a1) MV_1 (a2) MV_2 (a3) MV_4

(b1) RW_1 (b2) RW_2 (b3) RW_4

Figure 4.6: Increasing maximum allowed number of transfers

Increasing the Maximum Allowed Number of Transfers. In this set of experiments, we analyze the impact of the maximum allowed number of transfers h_{max} on the performance of the RICCdecay algorithm, and compare RICCdecay with RICCmeetMin. (RICCmeetMin's query processing part was modified to enable it to answer reachability queries with decay.) We ran a set of 100 queries varying h_{max} from 1 to 4; each query's interval was picked uniformly at random from 500 to 3500 sec for the Moving Vehicles datasets, and from 600 to 4200 sec for Random Walk datasets. The results are presented in Figure 4.6 $(a1 - b3)$. RICCdecay accesses from 1.8 (for $MV2$ dataset) to 11.5 (for $RW4$ dataset) times less pages than RICCmeetMin. The biggest advantage of RICCdecay over RICCmeetMin is achieved for $h_{max} = 1$ for all datasets, and in general, the smaller the

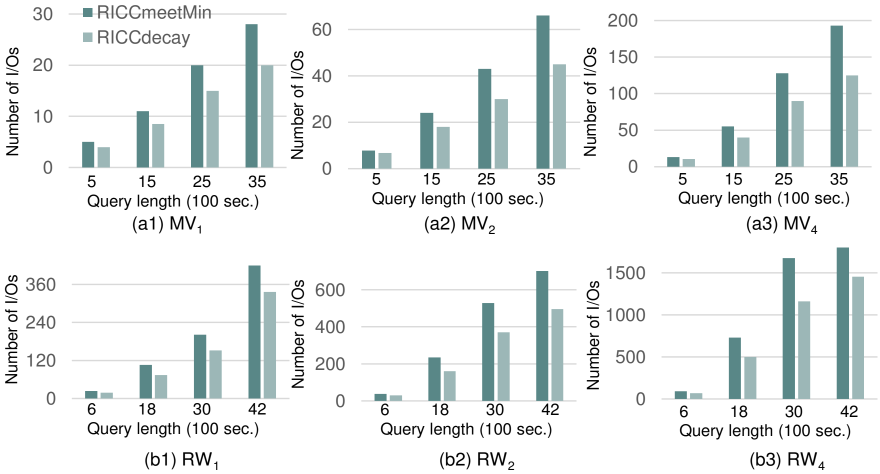


Figure 4.7: Increasing query length

maximum allowed number of transfers, the better is the performance of the RICCdecay algorithm.

This pattern can be explained as follows. When answering a query Q_{mh}, we read file *Reached(Hop)* first. File *Meetings* needs to be read next, but only if during traversing file *Reached(Hop)*, the target object appears among the objects, reached by the source (i.e. if $O_T \in S'_{Reached}$). However, $S'_{Reached}$ is a superset of the set of objects that can be reached by O_S during the query interval I. We say that a query is *pruned*, if it aborts after reading file *Reached(Hop)* because of not finding the target among the reached objects. By precomputing the hop value of each reached object, *Reached(Hop)* gives more accurate information, than RICCmeet, which reduces the size of $S'_{Reached}$. The smaller the h_{max} value, the less objects are in $S'_{Reached}$, and thus the higher percent of queries can be pruned.

Increasing Query Length. Now we test the performance of RICCdecay algorithm for various query lengths. Each test was run 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. The maximum allowed number of transfers h_{max} for each query was picked uniformly at random from 1 to 4. The results of comparison of the performance of RICCdecay with that of RICCmeetMin are shown in Figure 4.7 ($a1 - b3$). For these sets of queries, RICCdecay outperforms RICCmeetMin in all the tests, accessing about 44% less pages in average, and this result does not change significantly from one dataset to another.

Top-K Reachability Queries. The major difference between all the queries that were considered in this section until now is that those were one-to-one queries: they had one source and one target object. Top-k queries that we described in section 4.2 are an example of many-to-many queries: they may have more that one source and/or one target objects. Multiple sources lead to the increase in the search space, while multiple undefined targets prohibit from the early query suspension (in case of one defined target, if the target is discovered in file *Meetings* in the middle of the query interval, there is no need to continue the search). In addition, the need to calculate and compare the aggregate weights of the reached objects makes it impossible to prune a query (suspend it after just searching the file *Reached(Hop)*).

For each of our top-k experiments, we used sets of 100 queries, where query length was 3500 sec for MV datasets and 4200 sec for RW datasets. For each query, the number of source objects was 4: $S = \{O_{S_1}, O_{S_2}, O_{S_3}, O_{S_4}\}$, and each weight was assigned a value of 1. Further, $D = \{0.10, 0.15, 0.20, 0.25\}$, $\nu = 0.6$, while k was randomly picked from 4 to 20. The area covered by each dataset is very large, so to force objects to be reached by several sources, for each query, we picked source objects from the same cell (with the side equal to the candidate contact distance) at the beginning of the query interval. The results are depicted in Figure 4.8. They indicate that for top-k queries RICCtopK accesses in average about 37% less pages than RICCmeetMin for the MV datasets, and about 30% less pages for RW. The advantage of RICCtopK owes to both, the RICCdecay index, constructed during the preprocessing, and RICCtopK itself (the query processing algorithm). Information from RICCdecay's preprocessing allows for computing the maximum possible aggregate score F_{max} using information from file *Reached(Hop)*, while RICCtopK reduces the number of objects that have to be accessed when the query reads the file *Meetings*.

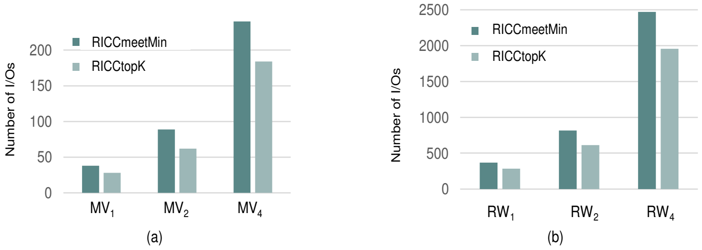


Figure 4.8: Top-k reachablility queries

4.7 Conclusion

In this chapter, we presented two novel reachability problems: reachability with transfer decay, and top-k reachability with transfer decay. Both problems assume the reachability with meetings scenario. The algorithms for the reachability with meetings, RICCmeetMin and RICCmeetMax were presented in the previous chapter. One of this algorithms, RICCmeetMin was modified to answer reachability with transfer decay and top-k queries, and served as a benchmark for our new algorithms. We designed two algorithms: RICCdecay and RICCtopK. The first algorithm allows to process reachability with decay queries efficiently and consists of the preprocessing and query processing stages. The second algorithm is designed for answering top-k queries and uses the preprocessing of RICCdecay. We tested our algorithms on six realistic datasets, varying query duration and the maximum allowed number of hops. The comparison of the performance of our new algorithms with that of RICCmeetMin proved that RICCdecay and RICCtopK can answer the types of queries that they were designed for more efficiently than the algorithm for the reachability with meetings problem.

Chapter 5

Conclusions and Future Work

Conclusions. In our work on efficient processing of novel reachability-based queries on large spatiotemporal datasets, we introduced several types of reachability-based queries: reachability queries with delayed exchange (considering processing and transfer delays), as well as reachability queries with transfer decay, and proposed several algorithms for answering each type of queries efficiently. All algorithms consist of the preprocessing and query processing stages. For the first stage, we use RICC-index or its modification. All algorithms were tested extensively on queries of different types, and proved to outperform their predecessors in the majority of the experiments.

Future Work. To efficiently answer k-top reachability queries with decay (described in Chapter 4), our algorithm RICCtopK currently uses the preprocessed data and index structure from RICCdecay. That allows to compute the upper bound of each reached object's aggregate weight and use it later to reduce the number of top-k candidates. The next step may be to modify the preprocessing part of RICCdecay in such a way that by providing more information, it will also allow to find the lower bounds on the reached objects' aggregate weights, which should greatly reduce the size of the *Candidate top-k set*, and as a result - improve the performance of the top-k algorithm.

One of the interesting directions on spatiotemporal reachability queries is reachability with uncertainty. In such problems, one can, for example, assume that during a meeting, a transfer occurs with some probability, which may be different from one transfer to another, depending on the area where the meeting occurred, time, etc. Then the reachability query, instead of answering whether the source object reached the target object during some interval I, will have to find the probability of the source object reaching the target object.

Another useful problem is on reachability with missing data, which frequently happens in real spatiotemporal datasets when location readings for some objects are not reported for some time intervals or are not accurate time- or location-wise. Then in order to complete the preprocessing and answer a query, some predictions will have to be made about the missing data, and/or multiple trajectory segments in place of each missing one may have to be considered. This would require very efficient algorithms for both, preprocessing and query processing, that can estimate or predict the missing records.

Acknowledgment: This research was partially supported by NSF grant IIS-1527984.

Bibliography

[1] R. Agrawal, A.Borgida, and H.V.Jagadish. Efficient Managemet on Transitive Relationships in Large Data and Knowledge Bases. In *ACM SIGMOD*, pages 253–262, 1989.

[2] P. Ahmed, M. Hasan, A. Kashyap, V. Hristidis, and V. J. Tsotras. Efficient computation of top-k frequent terms over spatio-temporal ranges. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1227–1241. ACM, 2017.

[3] Ritesh Ahuja, Nikos Armenatzoglou, Dimitris Papadias, and George J Fakas. Geo-social keyword search. In *International Symposium on Spatial and Temporal Databases*, pages 431–450. Springer, 2015.

[4] M. Attique, H. Cho, R. Jin, and T. Chung. Top-k spatial preference queries in directed road networks. *ISPRS International Journal of Geo-Information*, 5(10):170, 2016.

[5] P. Bakalov, M. Hadjieleftheriou, E. Keogh, and V.J. Tsotras. Efficient trajectory joins using symbolic representations. In *MDM*, pages 86–93, 2005.

[6] T. Brinkhoff et al. Generating traffic data. *IEEE Data Eng. Bull.*, 26(2):19–25, 2003.

[7] J. Cai and C. K. Poon. Path-hop: efficiently indexing large graphs for reachability queries. In *ACM CIKM*, pages 119–128, 2010.

[8] Y. Cai and R. Ng. Indexing spatio-temporal trajectories with chebyshev polynomials. In *ACM SIGMOD*, pages 599–610. ACM, 2004.

[9] L. Chen, G. Cong, C. S. Jensen, and D. Wu. Spatial keyword query processing: an experimental evaluation. In *Proceedings of the VLDB Endowment*, volume 6, pages 217–228. VLDB Endowment, 2013.

[10] S. Chen, B. Ooi, K. Tan, and M. Nascimento. St2b-tree: A self-tunable spatio-temporal b+-tree index for moving objects. In *ACM SIGMOD*, pages 29–42, 2008.

[11] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[12] V. T. De Almeida and R. H. Güting. Indexing the trajectories of moving objects in networks*. *Geoinformatica*, 9(1):33–60, 2005.

[13] I. De Felipe, V. Hristidis, and N. Rishe. Keyword search on spatial databases. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 656–665. IEEE, 2008.

[14] R. Fagin. Combining fuzzy information from multiple systems. *Journal of computer and system sciences*, 58(1):83–99, 1999.

[15] R. Fagin. Combining fuzzy information: an overview. *ACM SIGMOD Record*, 31(2):109–118, 2002.

[16] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

[17] R. Geisberger, M. N. Rice, P. Sanders, and V. J. Tsotras. Route Planning with Flexible Edge Restrictions. *ACM Journal of Experimental Algorithms*, 17(1):1–2, 2012.

[18] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *7th Intl. Conf. on Experimental algorithms*, pages 319–333, 2008.

[19] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Efficient indexing of spatiotemporal objects. In *EDBT*, pages 251–268, 2002.

[20] M. Hadjieleftheriou, G. Kollios, and V.J. Tsotras. Performance evaluation of spatio-temporal selectivity estimation techniques. In *SSDBM*, pages 202–211, 2003.

[21] W. Huo and V.J. Tsotras. Efficient temporal shortest path queries on evolving social graphs. In *SSDBM Conf.*, pages 38:1–38:4, 2014.

[22] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[23] C. Jensen, D. Lin, and B. Ooi. Continuous clustering of moving objects. *IEEE TKDE*, 19(9):1161–1174, 2007.

[24] H. Jeung, M. Yiu, X. Zhou, C. Jensen, and H. Shen. Discovery of convoys in trajectory databases. In *PVLDB*, volume 1, pages 1068–1080, 2008.

[25] E. Jin, N. Ruan, S. Dey, and J. Y. Xu. Scarab: scaling reachability computation on large graphs. In *ACM SIGMOD*, pages 169–180, 2012.

[26] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *ACM SIGMOD*, pages 813–826, 2009.

[27] Ruoming Jin, Lin Liu, Bolin Ding, and Haixun Wang. Distance-constraint reachability computation in uncertain graphs. *PVLDB*, 4(9):551–562, 2011.

[28] P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. In *SSTD*, pages 364–381, 2005.

[29] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *IEEE ICDE*, pages 997–1008, 2013.

[30] G. Kollios, D. Gunopulos, and V.J. Tsotras. On indexing mobile objects. In *ACM PODS*, pages 261–272, 1999.

[31] D.A. Maltz. Dynamic source routing in ad hoc wireless networks. *Mobile Computing*, 353(1):153–181, 1996.

[32] F. Merz and P. Sanders. PReaCH: A Fast Lightweight Reachability Index Using Pruning and Contraction Hierarchies. In *ESA Symp.*, pages 701–712, 2014.

[33] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-temporal Access Methods. *IEEE Data Engineering Bulletin*, 26(2):40–49, 2003.

[34] S. Nepal and MV Ramakrishna. Query processing issues in image (multimedia) databases. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 22–29. IEEE, 1999.

[35] L.V. Nguyen-Dinh, W. G. Aref, and M. F. Mokbel. Spatio-temporal Access Methods: Part2 (2003 - 2010). *IEEE Data Engineering Bulletin*, 33(2):46–55, 2010.

[36] J. Ni and C.V. Ravishankar. Indexing spatiotemporal trajectories with efficient polynomial approximation. *IEEE TKDE*, 19(5), 2007.

[37] J. Ni and C.V. Ravishankar. Pointwise-dense region queries in spatio-temporal databases. In *IEEE ICDE*, pages 1066–1075, 2007.

[38] J. M. Patel, Y. Chen, and V. P. Chakka. Stripes: An efficient index for predicted trajectories. In *ACM SIGMOD*, pages 635–646, 2004.

[39] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB*, pages 395–406, 2000.

[40] JB Rocha-Junior, A. Vlachou, C. Doulkeridis, and K. Nørvåg. Efficient processing of top-k spatial preference queries. *Proceedings of the VLDB Endowment*, 4(2):93–104, 2010.

[41] M. Sarwat and Y. Sun. Answering location-aware graph reachability queries on geosocial data. In *IEEE ICDE*, pages 207–210, 2017.

[42] S. Seufert, A. Anand, S. Bedathur, and G. Weikum. Ferrari: Flexible and efficient reachability range assignment for graph indexing. In *IEEE ICDE*, pages 1009–1020, 2013.

[43] H. Shirani-Mehr, F. Banaei-Kashani, and C. Shahabi. Efficient reachability query evaluation in large spatiotemporal contact datasets. In *PVLDB*, volume 5, pages 848–859, 2012.

[44] A. Skovsgaard, D. Sidlauskas, and C. S. Jensen. Scalable top-k spatio-temporal term querying. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 148–159. IEEE, 2014.

[45] E. V. Strzheletska and V. J. Tsotras. RICC: fast reachability query processing on large spatiotemporal datasets. In *SSTD*, pages 3–21, 2015.

[46] E. V. Strzheletska and V. J. Tsotras. Efficient processing of reachability queries with meetings. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 22. ACM, 2017.

[47] Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. Reachability querying: Can it be even faster? *IEEE TKDE*, pages 683–697, 2016.

[48] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. Characterising Temporal Distance and Reachability in Mobile and Online Social Networks. *ACM SIGCOMM Computer Communication Review*, 40(1):118–124, 2010.

[49] M. R. Vieira, P. Bakalov, and V.J.Tsotras. On-line discovery of flock patterns in spatio-temporal data. In *ACM GIS*, pages 286–295, 2009.

[50] M.R. Vieira, P. Bakalov, and V.J. Tsotras. Querying Trajectories Using Flexible Patterns. In *EDBT*, pages 406–417, 2010.

[51] M.R. Vieira, P. Bakalov, and V.J. Tsotras. FlexTrack: a System for Querying Flexible Patterns in Trajectory Databases. In *Int. Symp. on Advances in Spatial and Temporal Databases (SSTD)*, pages 478–480, 2011.

[52] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and Mario A. Lopez. Indexing the positions of continuously moving objects. In *ACM SIGMOD*, pages 331–342, 2000.

[53] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu. Dual labeling: Answering graph reachability queries in constant time. In *IEEE ICDE*, pages 75–75, 2006.

[54] D. Wu, M. L. Yiu, G. Cong, and C. S. Jensen. Joint top-k spatial keyword query processing. *IEEE Transactions on Knowledge and Data Engineering*, 24(10):1889–1903, 2012.

[55] D. Wu, M. L. Yiu, C. S. Jensen, and G. Cong. Efficient continuously moving top-k spatial keyword query processing. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 541–552. IEEE, 2011.

[56] X. Xiong, M. F. Mokbel, and W. G. Aref. Lugrid: Update-tolerant grid-based indexing for moving objects. In *MDM*, volume 13, 2006.

[57] H. Yildirim, V. Chaoji, and M. J. Zaki. GRAIL: scalable reachability index for large graphs. In *PVLDB*, pages 276–284, 2010.

[58] M. Yiu, Y. Tao, and N. Mamoulis. The bdual-tree: Indexing moving objects by space filling curves in dual space. *VLDB J.*, 17(3):379–400, 2008.

[59] T. Yufei, D. Papadias, and J. Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, pages 790–801, 2003.

[60] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: a total order approach. In *ACM SIGMOD*, pages 1323–1334, 2014.