Managing Moving Objects and Their Trajectories

Xiaohui Li

School of Computing

Computer Science Department

National University of Singapore

Supervisor: Kian-Lee TAN

A Thesis Submitted for the Degree of

Doctor of Philosophy

January 2013

I would like to dedicate this thesis to my beloved parents for their endless support and encouragement.

Acknowledgements

First and foremost I want to thank my advisor, Prof. Tan Kian-lee. I am grateful for his guidance to do research in computer science. He is always available for discussion whenever I have any questions. I really appreciate his contributions of time, ideas, and funding to make my Ph.D. experience productive and stimulating. I am also thankful for the freedom of exploring related research fields under his supervision.

I would also like to thank Prof. Christian S. Jensen and Vaida Čeikutė for their hosting in Aarhus university. My stay at AU was supported (in part) by an internationalization grant from Aarhus University. During that period, both of them have helped me a lot in both research and life. Prof. Jensen's enthusiasm for research is very encouraging and motivational. His insights into database research are invaluable for my research. I really appreciate their contributions on the papers that we have worked on together.

I am also thankful to my co-authors, Panagiotis Karras, Wu Wei, Shi Lei and Zhou Zenan. Their contributions to our papers have greatly improved it. It was great to work together with them.

I wish to extend my warmest thanks to all the wonderful friends that accompany me during my PhD studies. They have been very helpful in one way or another. They are always there when I need someone to talk to. We spend a lot of good times together. The precious memories will stay forever in my heart. I am sorry that I can only list some of them here: Luo Fei, Wang Guangsen, Su

Bolan, Chen Wei, Zhao Gang, Zhou Jian, Zhou Ye, Zhao Feng, Liao Lei, Htoo Htet Aung, Li Zhonghua, Kong Danyang, Liu Chengcheng and Lin Zhenli... It is said that PhD is a journey. I am so grateful that this journey is so memorable because of all of my friends.

This thesis would not have been possible without all these people.

Abstract

Today's Internet-enabled mobile devices are equipped with geo-positioning sensors that can readily identify location information, notably GPS data. This has resulted in the availability of rapidly increasing volumes of GPS data that record the movement histories of moving objects. In addition, real-time GPS data can stream into the server, enabling location-based services and real-time movement-pattern findings.

Many interesting applications that target moving objects have already emerged, and there is an urgent call for efficient algorithms to support these applications. At the same time, challenges to answer spatial queries efficiently in those applications also arise. In this thesis, we have identified problems that are related to moving objects and have real-life applicationsf and then proposed frameworks with efficient algorithms to solve these problems.

In particular, this thesis studies three types of spatial queries: moving continuous queries, group discovery queries, and optimal segment queries. First, we study the efficient processing of moving continuous queries. Such queries are issued by mobile clients who need to be continuously aware of other clients in its proximity. Past research on such problems has covered two extremes of the interactivity spectrum: It has offered totally centralized solutions, where a server takes care of all queries, and totally distributed solutions, in which there is no central authority at all. Unfortunately, none of these two solutions scales to intensive moving object tracking application, where each client poses a query. We propose a balanced model where servers cooperatively take care

of the global view, and handle the majority of the workload. Meanwhile, moving clients, having basic memory and computation resources, share a small portion of the workload. This model is further enhanced by dynamic region allocation and grid size adjustment mechanisms to reduce the communication and computation cost for both servers and clients.

Second, we study the processing of group discovery queries. Given a trajectory database, a group discovery query finds clusters of moving objects traveling together for a period. We propose a group discovery framework that efficiently supports their online discovery. The framework adopts a sampling-independent approach that makes no assumptions about when positions are sampled, gives no special importance to sampling points, and naturally supports the use of approximate trajectories. The framework's algorithms exploit state-of-the-art, density-based clustering to identify groups. The groups are scored based on their cardinality and duration, and the top-k groups are returned. To avoid returning similar subgroups in a result, notions of domination and similarity are introduced that enable pruning low-interest groups.

Third, we study the processing of optimal location queries. Given a road network, existing facilities, and routes of customers, an optimal location query identifies a road segment where building a new facility attracts the maximal number of customers by proximity. Optimal segment queries are a variant of the optimal region queries, which are variants of the well-studied optimal location (OL) queries. Existing works addressing the optimal region queries treat only static sites as the clients. In practice, however, routes produced by mobile clients (e.g. pedestrians, vehicles) are a more general form of clients than static points such as residences. Many types of business are also interested in both static points and mobile clients. We propose a framework to solve the optimal segment problem. The main idea of this framework is to assign each route a

score which is distributed to the road subsegments covered by the route based on an interest model. The road segments with the highest scores are identified and returned to the user.

For each framework we propose in the thesis, we conduct extensive experiments in realistic settings with both real and synthetic data sets. These experiments offer insight into the effectiveness and efficiency of the proposed frameworks.

Keywords: Moving objects, real-time location data, trajectory data, spatial query processing, range and k-nearest-neighbor query, continuous queries, group movement patterns, optimal segments, performance study

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 Motivations

Today's Internet-enabled mobile devices, e.g. Smart phones, PDAs, and laptops, are equipped with geo-positioning sensors that can readily identify location information, notably GPS data. Good connectivity (e.g. via GSM, Wi-fi, Bluetooth, or EDGE) can be easily established to communicate GPS data with the server, which provides useful services to the users, e.g. digital mapping services, traffic control, and taxi sharing etc. Recently, the rise of social networking web sites and apps has made particularly easy the sharing of small amount of location data (e.g. hiking and biking GPS traces), and thus has fueled the usage of GPS devices. For instance, Foursquare [1], a location-based social networking web site that allows a mobile user to discover friends and events that are nearby, has a community of over 15 million people worldwide. An app named MapMyRun [2] allows users to share their hiking or biking GPS traces to Facebook [3] and Twitter [4]. In addition, the development of digital mapping services has enabled the so-called third generation, more sophisticated traveling planning services, e.g., NileGuide[5] and YourTour[6].

Figure 1.1 illustrates the general infrastructure that manages moving object data and

[1]https://foursquare.com/
[2]http://www.mapmyrun.com/
[3]http://www.facebook.com/
[4]http://www.twitter.com/
[5]http://www.nileguide.com
[6]http://www.yourtour.com

queries. The mobile clients (e.g., vehicles or pedestrians) receive their current GPS locations from the satellites and update the server via WiFi (through wireless access point) or 3G network (through base stations in cellular network). The server, with the knowledge of the current location of every mobile client, is able to answer a spatial query such as "Continuously monitor my nearest 2 cars ".

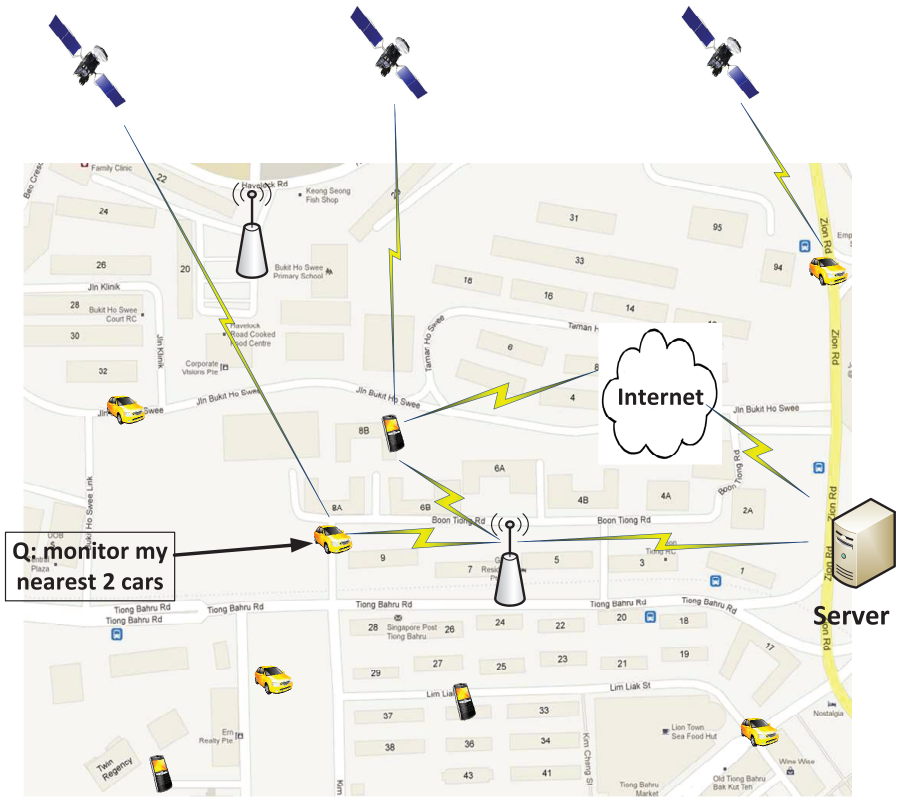


Figure 1.1: Infrastructure of Managing Moving Objects Data and Queries

From the server's perspective, moving objects data can be classified into two categories: real-time data and historical data. For some applications, moving objects data continuously stream into the server that in turn uses the data to process real-time queries. For some other applications, the increasing number of location-aware devices has resulted in the accumulation of a large amount of trajectory data that capture the movement histories of a variety of objects. In addition, the server can utilize both real-time data and trajectory for even more sophistried queries. When processing queries, the server can choose to process queries online or offline. Table 1.1 is a table that can be used to classify the works in the thesis.

	Online	**Offline**
Real-time data	Moving Continuous Query	–
Trajectories	–	Optimal Segment Query
Both	Group Query	–

Table 1.1: A Classification of Queries

The first query addressed in this thesis is real-time processing of moving continuous queries (MCQ) issued by mobile clients. In this problem, each mobile client has to be continuously aware of its neighbors in its proximity by issuing either a range query or kNN query. Several applications, such as massive multi-player online games (MMOG) (e.g., World of Warcraft), virtual community platforms (e.g., Second Life), real-life friend locator applications, and marine traffic management systems employed by port authorities, require efficient real-time processing of such queries. In all such applications, a large population of clients are moving around; their data continuously stream into the server. As Table 1.1 shows, we require that the server processes this type of query in an online setting. The large number of mobile clients and the fact that these mobile clients continuously move around have resulted in high workloads that a single server may not be able to handle well. Therefore, we design a scheme where a cluster of servers are interconnected to handle the workload cooperatively in such a highly dynamic environment.

The second query addressed in this thesis, so-called *group query*, is to find *group* patterns by examining both trajectories and real-time data. A group pattern is one where a number of moving objects travel together for a duration. With the increasing availability of trajectory data, the analysis of these data have important applications in entity behavior analysis (e.g. animal migration patterns [1]), socio-economic geography [32], transport analysis [73], and defense and surveillance areas [68]. Group patterns can be found by examining trajectories of mobile clients. Although there exist previous works in finding *flock* [34, 35], *convoy* [47], and *swarm* [61], we find none of them satisfies our four requirements, (1) Sampling independence, that is, the use of different representations (sampling points) of the same trajectories in an algorithm should not affect the outcome of the algorithm. Sampling independence prevents losing interesting patterns, as will be shown in the sequel. (2) Density connectedness, that is, members of the same group are density-connected as defined in DBScan [24]. Comparing to other clustering technique such as k-means that finds circular clusters, density-connectedness allows clusters with arbitrary shape. (3) Supporting trajectory approximation, that is, simplified trajectories can be used in place of original ones, and (4) Online processing, that is, real-time data is allowed to stream in for new patterns to be discovered. Motivated by it, we propose the GroupDiscovery framework, which is the first to satisfy all of the four requirements. From the requirements, we can see that this query falls in the category where the server online processes both trajectories and real-time data (See Table 1.1).

The third query addressed in this thesis is called *optimal segment query*. It is a new variant of the classic *facility location* problem. In this query, the server finds the optimal road segment to setup a new facility, given the road network, the customers' trajectories and existing facilities. Similar to facility location problems, it has wide applications in both private and public sectors, e.g., planning hospitals, gas stations, banks, ATMs or billboards. Earlier work aiming to solve the facility location problem has used the residences of customers as the customer locations [87, 90, 97]. However, customers do not remain stationary at their residences, but rather travel, e.g., to work. Thus, consumers are not only attracted to facilities according to the proximity of these to their residences. The increasing availability of moving-object trajectory data, e.g., as GPS traces, calls for an update to the facility location problem to also take into account the movements of the customers that are now available. When processing this query, the server processes trajectories in offline mode. It falls in the category where the server processes trajectories offline (See Table 1.1).

There is great linkage among the three pieces of works. In MCQ, the thesis only deals with real-time locations. In Group Query, the thesis takes the query processing to another level by taking into account of both real-time locations and past movement histories of moving objects in order to find co-movement patterns. In Optimal Segment Query, the thesis continues to show how the useful information buried inside a trajectory database can be valuable to identify optimal segments from a road network for various businesses.

1.2 Challenges

1.2.1 Challenges in Moving Continuous Query

Traditional techniques for continuous spatial query processing are based on a centralized client-server architecture or assume that there are significantly fewer queries than moving clients [66, 67, 80, 94]. Unfortunately, such techniques do not scale well to applications where each of a large number of mobile clients poses its own query. The applications we target call for solutions designed for the particular scalability challenges they pose. The solution to the scalability problem can be to buy a more powerful server or to buy more pieces of less powerful machines and then interconnect them to cooperatively handle the workload. We believe that the second solution is more viable and affordable than the first one. In the second solution, the challenge is to dynamically balance the workload among the servers. When mobile clients are moving around, data skew can happen, leading to deteriorated performance. In this case, servers need to re-balance the workload.

A second challenge in processing moving continuous query is that communication between the server and the clients is found to be the bottleneck to scale up. In our experiments, we found that it takes much longer time for client/server communication than the server to process queries when the workload is moderate. In addition, mobile clients have limited battery life. Too many messages sent by a client may rapidly exhaust its battery. Chapter 3 shows how these challenges are tackled.

1.2.2 Challenges in Group Query

In managing moving objects, one is not only interested in real-time data, but also in the trajectories, movement histories of moving objects accumulated over time. The volume of trajectories makes it almost impossible to extract any knowledge by plotting and observing them with human eyes on a map. In order to detect interesting moving patterns, e.g. flock, leadership, convergence, and encounter, these patterns have to be rigorously defined. And effective algorithms have to be devised.

The challenge in processing group query lies in our requirement that the framework has to satisfy four properties.

- Sampling independence. A trajectory, being a continuous function from time to location, can be sampled at different rates, called *sampling rate*. The resulted points are called *sampled points*. Many existing algorithms rely on the sampled points in order to detect moving patterns, and thus they are *sampling point dependent*. However, as will be shown in Chapter 4, a sampling point dependent algorithm suffers from missing interesting patterns. In order not to lose any interesting patterns, an algorithm has to produce the same result no matter how trajectories are sampled, a property called *sampling point independent*. Sampling point independence is formally defined in Chapter 4.

- Density connected. In our framework, the need to cluster moving objects arises at certain time points to find out candidates of groups. Density-connectedness should be used because the clusters of moving objects can be of any shape.

- Online trajectory simplification. Efficiency is a key requirement in an online processing setting. Online trajectory simplification allows to smoothen trajectories, and can improve the efficiency. It also allows the trading result accuracy with efficiency.

- Incremental processing. In an online setting, when new data stream in, results should be computed incrementally, in order to re-use the results computed before and thus improve the efficiency.

Chapter 4 shows how these challenges are tackled.

1.2.3 Challenges in Optimal Segment Query

Unlike conventional facility location problems, the optimal segment problem addressed in Chapter 5 takes route traversals as customers, which is a natural generalization of the use of static customer sites. It is the first such proposal. For route traversals, different from static customer sites, their scoring function and how they affect setting up new facilities have to be carefully designed to reflect the real-world scenario.

Second, the optimal segment problem finds optimal segments instead of optimal points on a road network. A straightforward approach that computes the optimal segment by enumerating and scoring all possible segments is not feasible, because there is an infinite number of possible segments. In order to reduce the huge search space quickly, efficient pruning techniques are devised and shown in Chapter 5.

.

1.3 Contributions

The contributions of this thesis can be divided into three parts based on the temporal dimension. In the first query, we consider *real-time* queries where the server uses the current-time location information of moving clients to process queries. The results are also sent to the clients in real time. In the second query, we combine both current-time locations and movement histories to find interesting group movement patterns. In the third query, we look even further back of the histories of mobile clients to find optimal segment(s) on a road network to set up a new facility

1.3.1 Moving Continuous Query

We formulate the *moving continuous query*. A Moving Continuous Query (MCQ) is issued by a mobile client who needs to be continuously aware of other mobile clients in its proximity. We consider two types of MCQs: range queries (MCQ-range) and kNN queries (MCQ-kNN). To answer MCQs, we present a dynamic framework where a cluster of servers cooperatively take care of the global view and handle the majority of the workload. The entire service space is also divided into smaller service regions, and the mobile clients in the same region are served by the same server. These regions are dynamic; they can be divided into smaller ones or be merged into larger ones, in order to reflect the current distribution of mobile clients. Service regions are served by servers. In the macro level, the framework balances the server workloads by region adjustment and reallocation. In the micro level, a server is allowed to fine tune its indexing structure to improve its processing efficiency and to handle data skew.

Meanwhile, moving clients, having basic memory and computation resources, handle small portions of the workload by maintaining their local results. Our experiments have proven that this approach is effective in reducing communication cost between clients and servers.

We implement the proposed framework and compare with the state-of-the-art algorithm. Experiments show that communication and computation costs for both servers and clients are reduced and our architecture is more scalable.

1.3.2 Group Query

Our proposal is the first to satisfy the four properties listed above. We propose a sampling-independent group discovery framework that efficiently supports the online, incremental discovery of moving objects that travel together. It supports the use of simplified trajectories, and exploits state-of-the-art, density-based clustering to identify groups.

In order to return most significant groups, the computed groups are scored based on their cardinality and duration, and only the top-k groups are returned. To avoid returning similar subgroups in a result, notions of domination and similarity are introduced that enable the pruning of low-interest groups.

We implement the algorithms and compare them with Convoy [47]. The experimental results show that our framework finds patterns that cannot be found by Convoy and the performance is better in most data sets and settings.

1.3.3 Optimal Segment Query

Although the optimal location problem is intensively studied before, we are the first to consider using trajectory data to solve the problem. We carefully define the optimal segment problem which takes as input a collection of routes, a collection of existing facilities and a road network, and finds the optimal segments of the road network to set up a new facility.

The following considerations are essential in solving the problem.

1. A route has a value to a business depending on factors such as its length, the number of people who take it, and the frequency that each person takes it.

2. A route is attracted by a facility if the route covers or is near the facility, because customers who take the route has a possibility of visiting the facility,

3. If a route is attracted by multiple facilities, the possibility of visiting each of them depends on the business.

4. When many high-valued routes cover the same road segment, this road segment is likely to be a candidate to set up a new facility.

For (1), each route is assigned a score based on the factors such as its length, the number of people who take it, and the total number of times that each person take it. For (2), we find out the attraction relations between routes and facilities. For (3), we propose various interest models for various businesses. We make sure our scheme is generic to different interest models. For (4), a route distributes its score to the segments covered by the route based on the specified interest model. A segment accumulates the scores distributed from its covering routes. In the end, the road segment(s) with the highest score is identified and returned to the user.

With these at place, we then propose two algorithms, AUG and ITE, to solve the optimal segment problem. AUG augments the road network graph with the facilities and the start and the end points of the routes. The augmented graph has the property that each route starts from a vertex and ends at a vertex. Then each vertex stores the identifiers of the routes covering it, and the score of each edge is the summation of the distributed scores of the routes that cover both its vertices. Next, AUG examines every edge in the augmented graph with a score and identifies the edges with the highest score (the optimal edges). Finally, AUG maps the optimal edges back to the original graph, where they are segments. Then AUG merges connected segments, if any, to form maximal segments, and returns them as the result.

The idea of the ITE algorithm is to quickly identify a subsegment of an optimal segment (optimal subsegment) and then extend the optimal subsegment into an entire optimal segment. Therefore, ITE organizes the segments using a heap such that those segments that are most likely to contain an optimal subsegment get examined first. If the segment under examination is an optimal subsegment then the entire optimal segment can be found by extending it. In addition, the optimal score can be calculated easily. Otherwise, the segment is partitioned into smaller segments, whose likelihoods of having an optimal subsegment are also calculated, upon which they are inserted back into the heap.

We conduct extensive experiments to evaluate the performance of these two algorithms. Experiment results show that they are effective and efficient.

1.4 Organization

The rest of the thesis is organized as follows.

- Chapter 2 reviews related topics. The surveyed topics include basic concepts, spatial queries and some indexing structures in Moving Object Databases (MOD).
- Chapter 3 presents our study on Moving Continuous Query. We focus on moving continuous range and kNN queries, which are two of the most fundamental queries in MOD.
- Chapter 4 presents our framework to find movement patterns (groups) from trajectory data.
- Chapter 5 presents our framework that uses routes of mobile clients to find optimal segments in a road network.
- Chapter 6 concludes this thesis and discusses several possible directions for future work.

1.5 Published Material

- The work in Chapter 3 has been published as a conference paper [60] in IEEE Mobile Data Management (MDM) 2013:

  Xiaohui Li, Panagiotis Karras, Lei Shi, Kian-Lee Tan, Christian S Jensen: Cooperative Scalable Moving Continuous Query Processing. IEEE 13th International Conference on Mobile Data Management, 2012: 69 – 78.
- The work in Chapter 4 has been published as a Journal paper [59] in IEEE Transactions on Knowledge and Data Engineering (TKDE) 2013:

  Xiaohui Li, Vaida Ceikute, Christian S Jensen, Kian-Lee Tan: Effective Online

Group Discovery in Trajectory Databases. IEEE Transactions on Knowledge and Data Engineering, 2012

- The work in Chapter 5 is ready for submission:

  Xiaohui Li, Vaida Ceikute, Christian S Jensen, Kian-Lee Tan: Trajectory Based Optimal Segment Computation in Road Network Databases, 2012

Chapter 2

Background and Related Work

In this chapter, we review existing works that are related to this thesis. As the queries addressed in this thesis are spatial queries that are usually handled in a moving object database (MOD), we first introduce MOD and selectively describe spatial queries that can be answered in MOD.

2.1 Moving Object Databases

Moving Object Databases (MOD) is an important research area and attracts a great deal of research interest during the last decade. The objective of MOD is to extend database technology to support the representation and querying of moving objects and their trajectories. MODs have become an emerging technological field due to the development of the ubiquitous location-aware devices, such as PDAs and mobile phones etc., as well as the variety of the information that can be extracted from such databases. Currently a number of decision support tasks exploit the presence of MODs, such as traffic estimation and prediction, analysis of traffic congestion conditions, fleet management systems, battlefield and animal immigration habits analysis [37].

Moving objects stored in MOD are geometries, such as points, lines and regions. If a subset of their physical phenomenons, such as shape, position, speed etc. changes over time, a trajectory mapping from time to the physical phenomenon can be derived to describe this change. In this thesis, we focus on moving points representing moving objects and trajectories recording their 2D positions over time.

2.1.1 Basic Concepts in MOD

Many real world applications involve entities that can be modelled as moving objects. As such, the histories of these entities can be modelled as trajectories. For instance, in the application that fleet management systems monitor cars in road networks, cars are viewed as moving objects, the history of which are modeled as trajectories. In MOD, the location of a moving object o at time t is $(t, \vec{p})$, where $\vec{p}$ is a point in the n-dimensional Euclidean space $\mathbb{R}^n$. For most typical applications, $n = 2$.

In some applications, the times when these locations are reported are not important. They only indicate an ordering of the locations. In this case, the sequence of locations of each moving object forms a *route*. In Chapter 5, we show the formal definition of a route and how the routes can be important for deriving the optimal locations for setting up new facilities.

Apart from a route, a trajectory takes into account the time information and can be defined as a function from the temporal domain $\mathbb{T}$ to the Euclidean space $\mathbb{R}^n$. In a 2D space, a trajectory tr is defined as

$$\mathbb{T} \rightarrow \mathbb{R}^2 : t \longmapsto a(t) = (x_t, y_t)$$

In practice, the trajectory of an object is collected by sampling the object's positions according to some policy, resulting in a set of sampling points. A trajectory is then given by the stepwise linear function obtained by connecting temporally consecutive sampling points with line segments. And in a 2D space, a trajectory tr is represented as

$$tr = \{(x_1, y_1, t_1),\ (x_2, y_2, t_2), ..., (x_n, y_n, t_n)\}$$

where x_i, $y_i \in \mathbb{R}$ and $t_i \in I$ and $t_1 < t_2 < ... < t_n$, the actual trajectory curve is approximated by applying spatio-temporal interpolation methods on the set of sample points

(Figure 2.1).

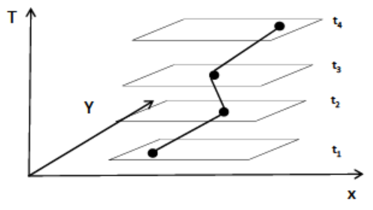


Figure 2.1: The spatio-temporal trajectory of a moving object: dots are sampled postions and lines in between represent linear interpolation.

2.1.2 Spatial Queries in MOD

Common spatial queries in MOD can be divided into location-based queries and non-location-based queries.

There are many location-based queries, and the services to answer these queries are generally referred to as location-based services. Location-based queries distinguish themselves from other spatial queries by containing location information in themselves. The conventional location-based queries that commonly appear in MOD are static range queries (also called window queries) and k Nearest Neighbors (kNN) queries. A static range query retrieves objects that are within a distance (range) from a query point. A kNN query, on the other hand, retrieves the k objects that are the nearest to the query point.

Recent years, several new kinds of location-based queries have been proposed. They include the Reverse k Nearest Neighbors (RkNN) query [52], Constrained Nearest Neighbor query [28], Group Nearest Neighbor (GNN) query [70], Nearest Surrounder query [56], and Spatial Skyline query [75].

These queries can be either snapshot or continuous, depending on whether the results have to be returned at a time instant or continuously. The former can be viewed as a special case of the latter. Depending on whether the query point is static or moving around, location based queries can be either static or mobile. Table 2.1 [88] provides a taxonomy of the location-based queries based on whether the query is a continuous query and whether the query location changes with time.

	Static query location	**Mobile query location**
Snapshot	static snapshot query	moving snapshot query
Continuous	static continuous query	moving continuous query

Table 2.1: A Taxonomy of Location-Based Queries

The first problem addressed in this thesis, the MCQ problem, falls in the lower right category, i.e., mobile query location and continuous query.

On the other hand, other spatial queries that do not contain location information themselves are non-location-based. For example, Spatial Join [36, 71] is an operation of combining two inputs based on their spatial relationships. Proximity queries [94] look for the neighbors in proximity for each moving object in the database. Convoy queries [47] search a trajectory database for patterns that several moving objects travel together for several consecutive time points. Optimal-location queries [90] find the optimal location(s) to set up new facilities based on the spatial attractions between existing facilities and customers. The second and third problem addressed in this thesis are non-location-based queries.

2.1.3 Indexing Structures in MOD

Here, we review common indexing structures in MOD. The ultimate goal of indexing is to answer queries efficiently. Indexing structures are especially important in MOD, where it is usually computationally expensive to process queries, due to the complex underlying data structures representing spatial data and the complexity of the query processing algorithms. Besides, we are faced with the challenge of dealing with a huge volume of data in MOD due to the wide-spread use of location-aware devices. We will have to resort to indexes once performance is concerned.

We summarize the main research issues in indexing moving objects data below.

- Simplicity. Minimal effort should be required to maintain the index.
- Update efficiency. Comparing with other types of databases, MOD usually receives higher volume of updates continuously, due to the dynamic nature of moving objects.

Updating operations have to be fast in order to give up resources for other operations.

- Query answer efficiency. It indicates how fast a query can be answered. Perhaps it is the most important metric in MOD.

The indexing of moving objects can be grouped into main memory based indexing or disk based indexing.

Moving object disk-based indexes can be further classified into two categories. The larger category is R-tree variants [38] (e.g. TPR-tree [84] and TPR*-tree [76]). This is actually an expected phenomenon given the popularity of the R-tree in spatial databases. These indexing structures are based on data partitioning. The distribution of data determines the index's structure. Although these indexing structures are great improvement over traditional indexing structures, they can show deficiency in update-intensive applications.

The second, smaller category of indexes relies on space partitioning. Examples are the B+-tree based indexes [43, 46, 95, 18]. They are reported to support both efficient updates and queries. In these structures, the space is divided into cells, and a space-filling curve is employed to assign identifiers to these cells. Then objects are indexed with a B+-tree with the identifiers of the cells they belongs to. The time dimension is partitioned into intervals depending on the maximum time duration between two updates of an object. Each time interval has a certain reference time. A portion of the B+-tree is reserved for each time interval.

Space partitioning based indexes surpass their counterparts that are based on data partitioning in two ways. First, both the B+-tree and the grid index are well established indexing structures present in virtually every commercial DBMS. The index can be integrated into an existing DBMS easily. No additional physical design is required to modify the underlying index structure, concurrency control and the query execution module of the DBMS. Second, in comparison with spatial indexes such as the R-tree, operations such as search, insertion and deletion on the B+-tree and the grid index can be performed very efficiently.

On the other hand, R-tree based indexes such as the TPR-tree are less susceptible to data diversities and changes as a result of MBRs splitting and merging. In contrast, because existing space partitioning indexes, such as the B^x-tree [43], partition space using a single uniform grid, the workload across different parts of the index may not be balanced. Such imbalance does impact the performance of existing indexes based on space partitioning

Recently, the main memory size becomes larger and more affordable. To meet the need of real-time monitoring and query processing, main memory indexing techniques are proposed [66, 67, 91]. These indexing structures are also space partitioning grids, due to the minimal maintenance and simplicity. They indexes partition space with a grid in advance. An object is indexed by the cell it belongs to. Same as the disk-based space partitioning, in-memory space partitioning completely ignores the feature (distribution) of objects. The indexing technique used in Chapter 3 follows the in-memory space-partitioning strategy.

2.2 Processing Moving Continuous Query

While much work has studied the problem of handling moving queries on stationary objects and stationary queries over moving objects, research on the most general form of the problem, Moving Continuous Query (MCQ), where both queries (range and kNN) and objects are moving, is still taking shape. In this general form, the problem poses the challenge of handling frequent location updates, while at the same time evaluating numerous moving continuous queries. Here, we review the principal techniques used to address these challenges.

Past approaches to this generalized problem can be divided into *distributed* and *centralized* solutions. Amir et al. [3] take a distributed approach and consider moving objects forming a *peer-to-peer* (P2P) network, where each object is a computing unit and no central server is present. Each pair of moving objects defines and maintains *safe region* information, capturing the region of their mutual proximity. Unfortunately, the performance of this method does not scale well as the number of objects, and hence peers for each of them, increases. On the other hand, in a centralized methodology [80], all proximity computations occur at the server, while each moving object sends location updates there. Each moving object also continuously checks whether its changing position falls within a moving sector region, defined by an angular threshold θ, a minimum speed V_{min}, and a maximum speed V_{max}. Farrel et al. [27] propose to relax query accuracy requirement for continuous range queries in order to cope with location uncertainty and reduce energy consumption of the clients. However, predefined values for these parameters have a detrimental effect on performance [94]. Chow et al. [20] propose a distributed query processing framework for continuous spatial queries (range and kNN queries). They argue that finding accurate results can be very costly in a mobile P2P environment, so their framework tradeoff quality of services (coverage and accuracy) with communication cost.

In a central setting, Lee et al. [56, 58] studies a new type of spatial query called *Nearest Surrounder (NS)*, which searches the nearest surrounding spatial objects around a query point. In particular, given a set of objects O and a query point p, this query returns a set of tuples in the form of $\langle o, [\alpha, \beta) \rangle$ where $o \in O$ and $[\alpha, \beta)$ is a range of angles in which o is the NN of q. Among the two works, [58] focuses on answering NS queries in moving objects environments. Compared with our scheme, NS is a more specific query, but our scheme can be applied with more general queries.

Lee et al. [57] targets general query processing in a mobile and wireless environment. They argue that when processing these queries, certain result reevaluations are not necessary when results do not change. Therefore, computing *valid scopes*, within which results do not need to be reevaluated, for query results avoids unnecessary result reevaluation and can reduce communication overhead and battery usage.

Yiu et al. [94] propose the Reactive Mobile Detection (RMD) algorithm to solve a similar problem, which is called *proximity detection (PD) problem.* Arguing that using predefined mobile safe region radius renders it difficult to control the messaging cost, they propose the RMD scheme that expands and contracts the radii of mobile regions when needed. The intuition is to contract the mobile regions to reduce the probing cost at the cost of more updates if the probing cost is too high, and vice versa, in order to minimize the overall cost. The MCQ problem is different from the PD problem. (a) In PD, a distance threshold is defined for each pair of clients, whereas in MCQ, each client can have its own distance threshold. (b) In PD, the server must have the exact solution. So it fails to exploit the computational power of clients, which can lead the way to higher scalability.

The adaptive grid-based solution in [92] treats a more general problem, where the objective is to detect whether a specified set of k objects can be enclosed by a circle of diameter at most ϵ. The server can detect definite positives and definite negative sets of objects, while the rest are probed to determine whether they satisfy the constraint or not. Still, as discussed above, this solution focuses on resource utilization at the server at the expense of already-high communication cost between server and clients.

Similar to our scheme, both MobiEyes [33] and DKNN [89] try to achieve a balance between server and client computation and aim at solving *moving continuous* range- and kNN-queries. Although a central server is employed in both works, computation occurs mostly on the moving clients, with the server and base station being mainly responsible for relaying messages among the clients, using an inflexible space-partitioning index that incurs high communication overhead. The major difference between our scheme and DKNN is whether service regions are dynamic and the extent to which mobile clients should share the workload. Different from DKNN where service regions are pre-defined and static in query processing, our scheme allows splitting and merging service regions in order to better cope with data skew and balance workloads in a highly dynamic environment. In addition, in DKNN, computation occurs mostly on the moving clients, while servers mainly relay messages. In contrast, in our scheme, mobile clients only share a small portion of the workload, saving limited battery lives of the clients.

Another related work is Wang et al. [85], where multiple servers also cooperatively handle queries. However, different from MCQ, [85] targeted *static* continuous range queries. Therefore, their scheme cannot be easily adapted for answering MCQs.

Chen et al. [19] study the efficient processing of predictive range queries, proposing spatio-temporal safe regions (STSR) to bound the movements of objects in order to reduce update costs. Adopting a centralized approach, STSRs are computed at the server side. Moving objects are only capable of sending update messages. Chen et al. [19] also consider two types of update messages, active and passive updates. An active message is sent when a moving object detects its STSR is no longer valid for the current status. A passive message is sent when the server processes predictive queries. A cost model was proposed to find the optimal STSR, by which they aimed to both reduce update cost and guarantee query accuracy.

2.3 Finding Moving Patterns from Trajectories

Several recent proposals aim to identify objects that travel together based on trajectory data.

Co-Movement Discovery

Several recent proposals aim to identify objects that travel together based on trajectory data.

The notion of flock was introduced by Laube and Imfeld [54] and further studied by others [2, 34, 35, 81, 8, 6]. A flock is a set of moving objects that travel together in a disk of radius r for a time interval whose duration is at least k consecutive time points. One similar notion identifies a set of objects as a moving cluster [50] if the objects when clustered at consecutive sampling times exhibit overlaps above a given threshold. Another similar notion, Herd [39], relies on the F-score to identify cluster overlaps at consecutive sampling times. A recent study by Jeung et al. [47] proposes the notion of convoy that uses density connectedness [24] for spatial clustering. Aung et al. [5] proposed the notion of evolving convoys to better understand the states of convoys. Specifically, an evolving convoy contains both dynamic members and persisted members. As time passes, the dynamic members are allowed to move into or out of the evolving convoy, creating many stages of the same convoy. At the end, the evolving convoys with their stages are returned. The differences between evolving convoys and our work are: (1) evolving convoys are sampling dependent, and (2) instead of collapsing convoys into stages, our work relies on a scoring function to return meaningful results. The advantages of the scoring approach are its simplicity and the control it offers.

In contrast to the above, the notions of group pattern [86] and swarm [61] permit patterns where moving objects travel together for a number of non-consecutive sampling times. Group patterns rely on disk-based clustering, while swarms use density connectedness for the spatial clustering.

Our scheme is different from all of the above proposals in that it is the first work to satisfy the four requirements that motivate our work.

In order to avoid running expensive incremental clustering (e.g., DBscan) at each time point, our proposal predicts the time when an event may occur. The motivation is that the number of events can be much fewer than the total number of time points, which can be very large, depending on the sampling technique.

The techniques used for supporting convoy and swarm discovery are capable of exploiting trajectory simplification. In convoy discovery, line segments are first clustered to identify candidate clusters. Then a refinement step considers the sampling points for these candidates. Thus, the accurate trajectories are only needed in the refinement step. The techniques used for swarm discovery apply sampling in a pre-processing step.

The techniques for computing convoys and swarms cannot be adapted easily to online settings. Convoys are computed by partitioning the time domain into intervals, upon which line segment clustering is applied to each interval. When new data stream in, the time domain needs to be redivided, and the computation needs to start from scratch. The techniques used for swarm discovery assume a static trajectory collection; without this assumption, swarms may not be maximal with respect to time. The pruning rules used in swarm discovery also require the time domain to be known beforehand. In contrast, our proposed solution employs an existing online simplification technique in the pre-processing step, monitors clusters *continuously* and records the histories of clusters. The evaluation of groups is only dependent on the most recent history, so the GroupDiscovery framework is amenable to online processing.

Moving Objects and Trajectory Clustering

The continuous clustering of the current positions of moving objects is related to the problem studied here. Jensen et al. [44] proposed a disk-based, incremental approach to continuously cluster moving objects. The scheme incrementally maintains and exploits a summary structure, called a *clustering feature*, for each cluster. During a time period, a moving object may be inserted into, or deleted from, a moving cluster. Next, the techniques monitor the radii of moving clusters, which change over time as the member objects move. When the average radius of a cluster c exceeds a threshold, c is split. In addition, a cluster is split if its cardinality exceeds a given threshold. Clusters may be merged if their cardinalities fall below a threshold.

Since this approach is disk-based, it is lossy as pointed out by Jeung et al. [47]. As such, it cannot be directly applied in our scheme, which uses density-connectedness to avoid the lossy problem.

Another line of research is trajectory clustering, in which the goal is to find the common paths of a group of moving objects. Trajectory clustering builds on advances in time-series data analysis. Thus, many clustering techniques that resemble their counterparts in time-series data analysis have been proposed, such as Dynamic Time Warping (DTW) [93], Longest Common Subsequences (LCSS) [83], Edit Distance on Real Sequence (EDR) [17], Edit distance with Real Penalty (ERP) [16], and a partition-and-group framework [55]. Li et al. [61] point out that these approaches are ill suited to find groups because the trajectories in a group may be quite different although they are close to each other (e.g., straight line trajectories vs. wave-like trajectories).

Trajectory Simplification

Trajectory simplification can improve the efficiency of many algorithms that operate on the trajectories by removing relatively unimportant data points. Trajectory simplification algorithms can be classified into batch or online algorithms. Batch algorithms, such as the Douglas-Peucker (DP) algorithm [22] and its variants, require the entire trajectory to be available, and thus are expected to produce relatively high quality approximation. In contrast, online algorithms, such as reservoir sampling algorithms [82] and sliding window algorithms [64], can work with partial trajectories and can be used for compressing data streams.

The GroupDiscovery framework employs the Normal Opening Window (NOPW) algorithm [64]. This algorithm starts by initializing an empty sliding window and setting the first point as an *anchor point* p_a. When a new location point p_i is added into the sliding window, a line segment $\overline{p_a p_i}$ is used to fit every location point in the sliding window. If no location point deviates from $\overline{p_a p_i}$ by more than a user-specified error bound, the sliding window grows by including the next new point p_{i+1}. Otherwise, the point with the highest error p_e is selected. The line segment $\overline{p_a p_e}$ is included as part of the approximate trajectory, and p_e is set as the new anchor point. The time complexity of NOPW is $O(n^2)$, where n is the number of data points in a trajectory.

In another line of research, Fagin et al. [25] propose the Threshold Algorithm (TA) that has some similarity with the history handler module. TA operates on a database where each object has m grades, one for each of m attributes, e.g., a multimedia database. Given a monotone aggregation function, e.g., min or average, that combines the individual grades to obtain an overall grade, TA finds the objects with top-k overall grades by concurrently accessing the sorted list of the attributes. It is shown that TA is instance optimal.

2.4 Finding Optimal Locations from Routes

A large body of literature on the classical facility location problem exists [15, 14, 13, 26, 30, 65, 69, 62, 41, 9, 51, 10] that assumes a static set of customer point locations. In general, the problem takes as input a finite set C of customers and a finite set P of candidate facility locations, and it returns a subset of k ($k > 0$) facilities in P that optimizes a predefined metric. However,we have not found any formalizations of and solutions to the problem we study here.

Many works [23, 87, 90, 96, 97, 13] study a variant of the facility location problem, the so-called the optimal location (OL) query, where only the optimal locations are returned from an *infinite* number of candidate locations, given that a finite set of facilities F already exists. Some works [13, 23, 87, 96, 97] study the problem in L_p space. Recently, Xiao et al. [90] extends the problem setting by taking the road network into account, using network distance in place of L_p distance. Our optimal segment query is related to OL query, but it uses route traversals instead of static customer point locations. The techniques presented in previous works cannot be applied to the optimal segment query.

Two other variants of the facility location problem are the *single facility location* problem [26, 69], and the *online facility location* problem [30, 65, 4, 31]. The single facility location problem finds one location in P that optimizes a predefined metric with respect to a given set C of customers. It requires that no facility has been built previously, whereas the optimal segment query consider the existence of a set F of facilities.

The online facility location problem assumes a dynamic setting where (i) the set C of customers is initially empty, and (ii) new customers may be inserted into C as time evolves. The solution to this problem constructs facilities one at a time, such that its quality (with respect to some predefined metric) is competitive against any solutions that are given all customer points in advance. This problem is different from the optimal segment problem because those techniques assume that the set P of candidate facility locations is finite.

Chapter 3

Processing Moving Continuous Query

3.1 Introduction

Recall that a Moving Continuous Query (MCQ) is issued by a mobile client who needs to be continuously aware of other clients in its proximity. We consider two types of queries: range query (MCQ-range) and kNN query (MCQ-kNN). In MCQ-range, each moving client issues a continuous range query with a custom radius r and centered at its own location to the database servers that are monitoring the entire service region. In contrast, in MCQ-kNN, each moving client issues a continuous kNN query. In MCQ, a moving client has a circular MCQ region that has its center at the moving client and has either a fixed radius r (in MCQ-range) or a varying radius (in MCQ-kNN). Every moving client functions both as a query issuer in its own right, and a potential participant in other clients' query results. Under these circumstances, central servers have to facilitate large-scale tracking and communication among the moving clients. Thus, the challenge is to keep both the server workload and the communication cost among servers and clients low.

In this chapter, we propose a framework that caters to these increased scalability requirements and leverages a ubiquitous computing environment. We propose a distributed architecture where servers take care of the global picture, tracking moving clients and communicating results to them; at the same time, mobile clients, having basic memory and computational resources, perform part of the required computational workload by updating their own results. In this framework, a cluster of inter-connected servers handle the majority of the workload in a cooperative manner. To that end, we partition the space into subregions (called service regions), each monitored by a separate server that is only responsible for computing the results for the moving queries in its service region. The server-server communication cost can be kept minimal by only exchanging information at the service region boundaries for cross-boundary queries. Three important metrics are considered here: client-server communication cost, client workload, and server workload. We assume that server-server communication is much less expensive than client-server communication because servers are inter-connected by high-speed cables.

The communication needs of mobile client monitoring applications can be divided into three components.

Location updates. In the framework, each server is indexing the mobile clients with an in-memory grid structure. Each server is allowed to freely have a different but coordinated grid cell side length depending on the distribution of mobile clients. This flexibility greatly reduces the total amount of update messages. To further reduce the number of update messages, each server negotiates a mobile region with every mobile client in its service region. A mobile region $\mathcal{R}$ of a mobile client o moves from the most recently updated position of o at o's most recently known velocity. Such regions are assigned by servers to clients in their service region. A client o needs not send location updates as long as it does not exit its most recently assigned mobile region [45].

Probe messages. These messages are sent from a server to its clients when the server needs to know the exact location of a client.

Results update. In the framework, results are maintained by the clients which do not need to notify the server when result objects exit the query circles. We employ an adaptation of the *trigger time* concept [40]. A trigger time is the time when a moving client no longer satisfies another client's MCQ, and a *trigger event* is the event of exiting that MCQ region. Each mobile client maintains a queue of events, ordered by their trigger times. Servers can insert events into the event queues of each client so that each client verifies whether the pending trigger events in its queue have occurred at each time point and updates its local results accordingly. The amount of communication is reduced because no messages are being exchanged.

Our major contributions can be summarized as follows:

- We propose Moving Continuous Queries (MCQ) that have wide application in both real-world and gaming applications. We consider two types of MCQs: MCQ-range queries and MCQ-kNN queries. MCQs are distinct from existing queries in the following aspects: (1) Each client is allowed to issue at least one MCQ, and thus the number of queries can be larger than the number of moving clients. (2) Each client must have exact answer to its own query, while servers do not need to store the answer to each query.

- We propose balanced computational models to efficiently answer MCQs. In these models, servers cooperatively take care of the global view and share the majority of the workload among them. Clients are only concerned with their vicinity and also carry out a small portion of the workload.

- We also propose a mechanism to dynamically allocate regions to handle the problem of skew so that it is possible to balance the workload among the servers even in the extreme case that all mobile clients cluster at one service region.

- We report on extensive experiments that offer insight into the efficiency of the proposed framework.

The rest of this chapter is structured as follows. Section 3.2 formally defines the problem that we address. Section 3.3 describes our system architecture. Section 3.4 details the operation of query processing in MCQ-range. Section 3.5 explains how the framework handles MCQ-kNN. In Section 3.6, we examine how our system adapts to a highly dynamic environment. Section 3.7 covers our experimental studies. Finally, Section 3.8 summarizes this chapter.

3.2 Problem Definition

The section formalizes the problem we set out to solve. Important notation is summarized in Table 3.1.

α	A grid cell side length
β	The ratio to determine overloading
γ	The ratio to determine skew in one service region
σ	The average number of moving objects per cell
$tr_{o_1}(o_2)$	The trigger time when o_2 is no longer o_1's result
t_d	The time threshold for dynamic region allocation
t_e	The time threshold for dynamic cell side length adjustment
R	Service Region

Table 3.1: Notation Used in the Chapter

First of all, let $D(o_1, o_2)$ denote the Euclidean distance between two moving clients o_1 and o_2. We assume an environment where each moving client issues a *continuous range query*; one moving object issues only one query at a time. The problem is to keep track of all moving clients, and to update the results of the queries they issue accordingly.

Definition 1 *(MCQ-range) Given a moving client o with a location p, a* moving continuous range query *(MCQ-range) of o is a circular range query $\odot(p, r)$ centered at p and with radius r that continuously retrieves all objects within that circle.*

As this definition suggests, it is important for a *client* to have the result of its MCQ-range query. The server may or may not have that result. A moving client is represented as $o_i = \langle oid, \vec{p}_i^{ref}, \vec{v}_i^{ref}, t_i^{ref}, r\rangle$, where oid is an object identifier, $\vec{p}_i^{ref}$ is the the most recently updated location of o at time t_i^{ref}, $\vec{v}_i^{ref}$ is the most recently updated velocity at time t_i^{ref}, and r is the radius of the query issued by o. To reduce the amount of location updates sent by mobile clients to the server, we use a linear function to model o's movement, based on the latest known location and velocity of o, at time t_i^{ref}. Thus, the location of o_i at time t is $\vec{p}_i(t) = \vec{p}_i^{ref} + \vec{v}_i^{ref} \cdot (t - t_i^{ref})$, where $t \geq t_i^{ref}$.

In our setting, space is partitioned into cells by a grid; furthermore, the server assumes the responsibility of sending to each client a set of *candidate results*, which may satisfy the client's query (circle with radius r centered at $o.p$). In effect, the server has to know which cells overlap with the query of each client o. We define that set of cells as o's *alert region*. The shaded grid cells in Figure 3.1 illustrate the alert region of o. Its query radius is r and mobile region radius is λ, which is introduced and discussed in the sequel. The server has to keep track of the alert region of each object o by indexing that object with the grid cells. Therefore, for each cell c, the server maintains a list of queries to whose alert regions contain c.

To reduce the server workload, servers take a simple approach to computing alert regions. Servers implement grids by using two dimensional arrays. Knowing that alert regions are the intersection of grid cells and bounding rectangles of MCQ-range queries, servers only need to compute the indices of the intersected cells. Suppose the location of o is $p = (x_0, y_0)$. The server of o computes

$$x_l = \max(\frac{x_0 - r}{\alpha}, 0), \qquad y_l = \max(\frac{y_0 - r}{\alpha}, 0)$$
$$y_h = \max(\frac{y_0 + r}{\alpha}, maxIdx_x), \qquad y_h = \max(\frac{y_0 + r}{\alpha}, maxIdx_y)$$

where $maxIdx_x$ and $maxIdx_y$ are the maximum x and y indices for the 2D array. The alert region of o is then $\{G[i][j] \,|\, x_l \leq i \leq x_h \wedge y_l \leq j \leq y_h \wedge (G[i][j] \cap o.q \neq \emptyset)\}$, where G is the grid and $o.q$ is o's query region.

A trigger time in our setting has the form $\langle t, oid \rangle$ indicating that the moving client with the identifier oid exits the query circle of another moving client at time t. Each client maintains a heap structure indexing on the trigger times for its results.

Definition 2 *(Trigger Time) Given a moving client o_1, suppose o_2 is another moving client that satisfies $o_1.q$. The trigger time for o_2 is the earliest time t so that o_2 no longer satisfies $o_1.q$.*

$$tr_{o_1}(o_2) = \min\{t \,|\, t > t_{ref} \wedge D(p_{o_1}(t), p_{o_2}(t)) > o_1.r\}$$

The queue is ordered on the trigger time and stored as a heap in each moving client.

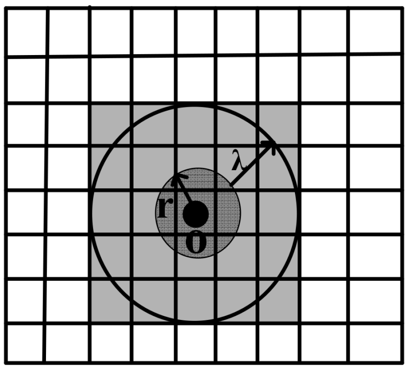


Figure 3.1: Alert Region and Query Region Augmentation

3.3 System Overview

We now proceed to present the system architecture that supports scalable query processing. We assume that moving clients and their queries fit into main memory of the database server. This assumption is valid in most applications where real-time evaluation is needed.

As we have discussed, a single-server centralized solution cannot easily scale to a large number of queries in a highly dynamic environment. Thus, our system architecture employs a cluster of servers, dividing the service space into *service regions*.

3.3.1 Space Division Model

Each server is assigned to monitor a part of the space (called service region), and it is only responsible for computing the results for moving queries in its service region. The challenge with this design is that the distribution of the moving clients can be highly skewed. In the extreme case, all mobile clients can be in a single service region. Such skew may cause unbalanced workloads among servers and significant overhead of client-server communication cost. In the framework, we introduce two mechanisms to handle skew in order to achieve a balanced server workload and minimize the client-server communication cost. The macro-level mechanism is to dynamically split and merge regions. When a server detects that its workload is significantly higher than the average workload, the server is able to further divide its service region for servers with less workload to take up. On the other hand, compatible regions with very few moving clients can be merged and then monitored as one single service region.

This model also allows the same server to monitor more than one, potentially non-adjacent, regions. In case any service region R has more than β times the average number of moving objects in each region, R is split into two new smaller service regions, one of which is then taken over by the server with the least number of moving objects in its service region. We use the term *configuration* for a space division. Obviously, there can be infinite many configurations in a system.

The micro-level mechanism is to allow each server to tune its own grid cell side length. As service regions may have different densities of clients, each server needs to pick the right granularity of the grid structure that reduces the amount of messages being exchanged between clients and the server.

3.3.2 Server Cluster Initialization

When the server cluster is initialized, a *bootstrap server*, which can be any server in the system, obtains the total number of available servers and partitions the entire service space using *binary space partitioning*. Ideally, each service region should contain almost the same number of moving objects. Each server is assigned the task of monitoring the moving objects in its service region, while the moving clients themselves are also informed about the server they should report their location updates to. We assume that each server knows the addresses of all other servers. For a small number of servers, the resulting routing tables can be stored in each server's main memory.

In the event that a new server S' requests to join an existing server cluster, the following steps occur.

1. The *bootstrap server* selects the service region R that contains the largest number of moving objects; assume R is currently assigned to server S.

2. R is divided into two regions of equal area, R_1 and R_2.

3. R_2 is assigned to S', which obtains from S the list of moving objects in R_2.

4. S informs the moving objects in R_2 that they should report to S' and removes them from its list.

5. S broadcasts messages about this event to other servers.

Each server maintains its own space-partitioning grid as its own local index. Another important task for server-server initialization is that each server tunes its cell size to the density of objects in its region aiming at the same average object per cell ratio σ. Suppose a server is monitoring N moving clients in a space of area A. Recall that α is the length of a grid. We have the following relation between σ and α.

$$\sigma = \frac{\alpha^2}{A} \cdot N \qquad (3.1) \qquad\qquad \alpha = \sqrt{\frac{A \cdot \sigma}{N}} \qquad (3.2)$$

When the system initializes, the bootstrap server picks up a value for its cell length, calculates its σ, and broadcasts σ to all other servers. Upon receiving the σ value, each server computes its own cell length with equation 3.2, substituting its own number of moving clients and service region area.

Note that after initialization, the cell side lengths may still be adjusted in circumstances such as the distribution of the moving clients is highly skewed. In system optimization, we give a discussion on how to dynamically adjust cell side lengths when the system is running.

3.4 Processing MCQ-range

We now consider query processing on a single server S. We use terms *moving query* and *moving object* interchangeably, depending on the context, as each object issues one query. We use the term *query circle* to refer to the query issued by a moving object.

As discussed, the server maintains an in-memory grid index, in which each cell has side length α. We assume there is the maximum speed limit, V_m, for all objects. It should hold that $\alpha \geq tu \cdot V_m$, where tu is the basic time unit, so that no object enters more than one cell from one time point to the next.

For each cell, the server maintains a list of its overlapping query circles, as well as a list of moving objects. The moving objects themselves maintain information for computation. Each moving object also maintains the following data: its own id (oid), the query radius (r), its velocity (v), its current server ($currS$), its current location ($\overrightarrow{p}$), a list of MOs satisfying the MCQ ($lRes$), and an event priority queue ($eventQ$). Query processing is divided into two phases, namely *initialization* and *continuous monitoring*.

3.4.1 Query Processing at Initialization

During initialization, a server S identifies all queries in its service region, and computes initial results for every query with the following steps.

1. S broadcasts a request for data to its service region R.

2. Each moving object o in R sends $\langle oid, \vec{p}_0, \vec{v}_0, t_0, r\rangle$ to S.

3. S receives the locations of all moving objects in R, populates each grid cell's list of overlapping queries, computes the initial result for every query, and sends it to the respective client o, along with the information on moving objects in o's alert region.

4. Upon receiving the message from S, a moving client o_i initializes its variables with the correct values and populates its event priority queue, $eventQ$.

The $eventQ$ is used to implement a concept similar to the *trigger time* [40], as follows: For each moving client o_i, the server calculates the expected time at which each current query result will cease being a result. The server then sends to o_i a message containing the identifier, location, velocity, and expected exit time of each result object, in the form $\langle t_j, oid_j, \overrightarrow{p}, \overrightarrow{v}\rangle$, signifying that object with identifier oid_j is expected to exit the query result at time t_j. These entries are indexed by $eventQ$ on the exit times. As will be illustrated later, the number of messages exchanged between clients and server is kept minimal with the help of the $eventQ$s.

Following the representation of a moving client in Section 3.2, the squared Euclidean distance between o_1 and o_2 at time t, $t > t_{ref}$, can be computed as follows [7].

$$\begin{aligned}
D_{o_1o_2}(t) &= (\vec{p}_1(t) - \vec{p}_2(t))^2 \\
&= (p_1^{ref} + v_1^{ref}(t - t_1^{ref}) - (p_2^{ref} + v_2^{ref}(t - t_2^{ref})))^2 \\
&= (v_1^{ref} - v_2^{ref})^2 \cdot t^2 \\
&+ 2(v_1^{ref} - v_2^{ref})(p_1^{ref} - p_2^{ref} - v_1^{ref}t_1^{ref} + v_2^{ref}t_2^{ref}) \cdot t \\
&+ (p_1^{ref} - p_2^{ref} - v_1^{ref}t_1^{ref} + v_2^{ref}t_2^{ref})^2
\end{aligned}$$

We need to compute the time t when a result object exits the query circle, which is given by finding the positive root of the formula: $D_{o_1o_2}(t) = r^2$. The calculation above is carried out by the function getExitTime().

3.4.2 Continuous Monitoring

We re-iterate that each client is responsible for monitoring its own query result, preferably *without* updating the server, unless that is deemed necessary; this approach serves to reduce both the communication cost and the computational burden at the server, at the cost of a bit more local computation.

In the next sections, we show two computational models.

3.4.3 Monitoring without Mobile Regions

We first illustrate query processing in our system without using the mobile region concept.

In this algorithm, as soon as a moving client o moves to a new location, it updates its server S_o about its new location and velocity. S_o computes the cells that appear in new alert regions of o and checks if there are moving clients in these new cells. Specifically, S_o does not need to send candidate messages if no other clients exist in new overlapping

Algorithm 1: Server Procedure without Mobile Region

```
foreach moving client $o$ that sends a location update do
    $C_1 \leftarrow$ newly contained cells by $o.q$;
    $C_2 \leftarrow$ partially intersected cells in the alert region;
    $O \leftarrow \{o' | o' \in C_1.objSet \cup C_2.objSet\}$;
    if $O \neq \emptyset$ then
        $O_{res} \leftarrow \emptyset$;
        foreach $o' \in O$ do
            if $\text{isResult}(o', o)$ then
                $t \leftarrow \text{getExitTime}(o', o)$;
                $O_{res}.add(\langle t, o' \rangle)$;
        if $O_{res} \neq \emptyset$ then sendMsg($O_{res}, o$);
```

cells. However, it is not enough to only compute the difference between old and new alert regions. Some objects may move into o's query circle even if o does not move at all. For correctness, S_o has to send candidate objects located in the *partially* intersected cells in the new alert region, even if some of them might be already in the result of o. Note that S_o does not need to send moving objects in the *fully* contained cells in the new alert region, thanks to the maximum speed assumption. The detailed steps are summarized in Algorithm 1.

Algorithm 2: Client Procedure without Mobile Region

```
input : $O_{res}$ from the server
while $eventQ.\text{peek}() \leq currTime$ do
    $(t, o) \leftarrow eventQ.\text{pop}()$;
    if $O_{res}.\text{has}(o)$ then
        $t_s \leftarrow O_{res}.\text{getNewExitTime}(o)$;
        $eventQ.\text{update}(t_s, o)$;
        $O_{res}.\text{remove}(o)$;
    else $lRes.\text{remove}(o)$;
foreach $(t, o) \in O_{res}$ do
    $lRes.\text{add}(o)$;
    $eventQ.\text{add}(t, o)$;
```

The procedure that takes place at the client side consists of two phases, namely the *deletion phase* and *addition phase*. The detailed steps are summarized in Algorithm 2.

The client procedure takes the message Q_{res} sent from server as input. In the deletion phase, from the $eventQ$, the algorithm pops out the result object that is expected to exit from the query circle (lines 1-2). If the result object is in Q_{res} (line 3), then the object is still in the query circle. The new expected exit time is taken from Q_{res} (line 4), and the $eventQ$ is updated with the new time (line 5). Q_{res} removes the result object to avoid duplicate operation (line 6). Otherwise, the result object is removed from the result set (line 7). In the addition phase, the result objects in Q_{res} are added into the result set, and into the $eventQ$ together with their expected exit times.

3.4.4 Monitoring with Mobile Regions

The procedure just outlined incurs high location update cost. To render our solution more efficient, we introduce the concept of a vector-based *mobile region*.

Definition 3 *(Mobile Region) Given a mobile client o and a radius λ, a* mobile region *is the set of points in the circular range $\odot(o.\overrightarrow{p}, \lambda, o.\overrightarrow{v})$. The mobile region has the same velocity as the mobile client o.*

In our setting, mobile regions are agreed upon by each moving client o and its server S_o at the most recent update of o. Subsequently, both parties are aware of the mobile regions. As long as o stays in its mobile region, it does not need to send location updates to S_o. We say the mobile region is *valid* for o. Since S_o does not have o's exact location, the query region of o has to be augmented by the mobile region (see Figure 3.1) to ensure correctness.

Lemma 3.4.1 *Let mr_o be the mobile region for a mobile client o. Let λ be the radius of mr_o. As the server side, the new query region of o is guaranteed to contain the original query region if the new query region radius is $o.r + \lambda$. (Proof omitted)*

Proof 3.4.1 *With the mobile region definitions, o is bounded by mr_o. The extreme case is that o is located on the boundary of mr_o, where the sum of the mobile region radius and the query region circle is $o.r + \lambda$. Therefore, a region with radius $o.r + \lambda$ is guaranteed to contain the original query region.*

Algorithm 3: Server Procedure with Mobile Region

```
foreach moving client $o$ do
    if receive message that $o$ exits $mr_o$ then
        create new $mr_o$;
        sendMsg($o, mr_o$);
    $q \leftarrow \text{augmentQuery}(mr_o)$;
    $R \leftarrow \emptyset$;
    $C_1 \leftarrow$ newly contained cells in $q$;
    $C_2 \leftarrow$ partially intersected cells in $q$;
    $O \leftarrow \{o' | \text{intersects}(mr_{o'}, C_1 \cup C_2) = \text{true}\}$;
    foreach $o' \in O$ do
        sendMsg($o'$, PROBE);
        cacheLoc($o', \tau_{oid}$);
        if contains($q, o'$) then
            $R \leftarrow R \cup \{o'\}$;
    if $R \neq \emptyset$ then sendMsg($R, o$);
```

Treating the augmented circular range as the new query, S_o can compute the alert region exactly as the case without mobile regions. In this algorithm, servers keep track of the global picture, while details are left upon the clients. However, unlike in the procedures for continuous monitoring without mobile regions, S_o no longer has the exact location of the query object qo. Instead, knowing the augmented alert region of qo, it identifies cells that *may* contain candidate results for qo, followed by probing the exact locations of these candidates and caching their exact locations. S_o then sends qo a candidate message with the exact locations. Upon receiving the candidate results, qo has to filter them based on its own location and query radius.

If a moving client exits its mobile region, a new mobile region is created to continuously bound its movement. With the mobile regions, location updates can be reduced greatly. However, this comes at the expense of increasing probe messages. This tradeoff is investigated in the experimental studies.

It is important for server S_o to cache exact locations of some moving clients because these exact locations may be re-used for queries issued by other moving clients in future. Caching exact locations can potentially save a large number of probe messages. But it is also important to allocate a cache of a small size to save precious memory space. In our implementation, the server allocates a fixed amount of buffer space to cache exact locations, e.g., 2KBytes. In addition, the server assigns identical life time τ_{oid} to each moving client o. Every time the exact location of o is re-used, its life time is renewed. If by τ_{oid} the exact location of o is not re-used, it is removed from the buffer. The detailed server procedure is summarized in Algorithm 3.

In this setting, the objects must send updated locations to their server when they exit their mobile regions. Each client also monitors its own result changes based on the $eventQ$ and its current location. As in the procedure without mobile regions, this procedure has a deletion and an addition phase. In the deletion phase, the client removes any previous results that no longer satisfy the query. In the addition phase, the moving client verifies which candidates are true results according to candidate-message sent by the server and then updates both the $lRes$ and $eventQ$ accordingly. The detailed client procedure is summarized in Algorithm 4.

Algorithm 4: Client Procedure with Mobile Region

```
   input  : $R$ from the server
1  if exit($\overrightarrow{p}$, $mr$) then
2      $sendMsg(\overrightarrow{p}, \overrightarrow{v})$;
3      $mr \leftarrow mr$ from server;
4  while $eventQ$.peek() $\leq$ $currTime$ do
5      $(t, o) \leftarrow eventQ$.pop();
6      if $o \in R$ then
7          if isResult($\overrightarrow{p}, r, o$) then
8              $t \leftarrow$ getExitTime($\overrightarrow{p}, \overrightarrow{v}, o$);
9              $eventQ$.update($t, o$);
10             $R$.remove($o$);
11     else $lRes$.remove($o$);
12 foreach $o \in R$ do
13     if isResult($\overrightarrow{p}, r, o$) then
14         $lRes$.add($roid$);
15         $t \leftarrow$ getExitTime($\overrightarrow{p}, \overrightarrow{v}, o$);
16         $eventQ$.add($t, o$);
```

3.4.5 Cross Boundary Queries

When the query circle of a client o covers several service regions, the server S_o answers the query by communicating with respective servers.

Figure 3.2 illustrates a scenario where a moving object o has its query cover both service regions of S_1 and S_2. S_1 probes the exact location of o and sends it to S_2, together with the query radius $o.r$ and the mobile region $o.\lambda$. According to Lemma 3.4.1, to ensure the correctness of the result of o, S_2 overlaps the new query region of radius $(o.r + o.\lambda)$ and retrieves its overlapping cells. Recall that each cell maintains a list of identifiers of objects inside the cell. S_2 probes the locations and velocities of these clients and sends them to S_1. S_1 computes and sends the candidate set for each query object as usual. This approach can be easily generalized to the case where a query overlaps with multiple service regions.

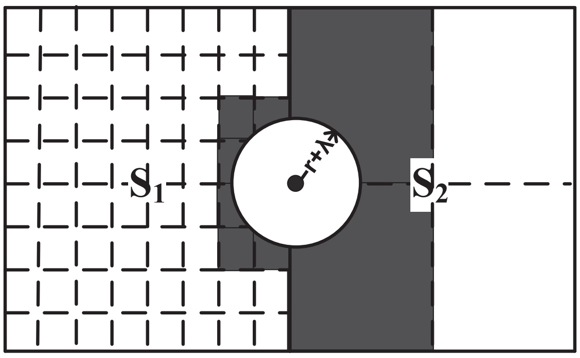


Figure 3.2: A Cross Boundary Query

3.4.6 Client Handover

When an object moves across a region boundary, the server of the old region informs the server in the new region by sending it a handover message. Upon receiving a handover message, the server in the new region probes the exact location of the moving client, and send candidate results in its service region. The handover should occur seamlessly, in that the handover should not affect the correctness of the result of the object. Handling this type of event is particularly straightforward in our query processing scheme because the moving clients compute the exact results. The only thing that the moving clients have to be aware of is to send location updates to the server responsible for the new service region.

3.5 Processing MCQ-kNN

This section describes how kNN queries are handled in the proposed framework. Note that the proposed system architecture and optimizations are orthogonal to the processing of the kNN queries. Therefore, both the architecture and the optimizations are still applicable in the kNN query. We focus on the query processing in the remainder of this section.

First, we define MCQ-kNN query.

Definition 4 *(MCQ-kNN) Given a moving client o with a location p and an integer k_o, a* moving continuous kNN query *(MCQ-kNN) of o continuously retrieves its k_o nearest neighbors.*

The main difference between MCQ-range and MCQ-kNN is that the radii of the query circles in MCQ-kNN are no longer static. For each moving client o_i, its query radius changes as the kth nearest neighbor o_j moves around. We call the Euclidean distance between o_i and o_j the *critical distance*.

In the following subsections, we explain how query processing is done in a single server. Then, we explain how the servers cooperatively handle the entire workload.

3.5.1 Query Processing at Initialization

Recall that a server S has a grid index structure for its service region in its memory. Similar to MCQ-range query processing, MCQ-kNN query processing is divided into two phases: the initialization phase and the continuous processing phase.

During initialization, S also identifies all MCQ-kNN queries in its service region and computes the initial result for every MCQ-kNN query with the following steps.

1. S broadcasts a request for data to its service region R.

2. Each moving object o in R sends $\langle oid, \vec{p}_0, \vec{v}_0, t_0, k_o \rangle$ to S.

3. S receives the locations of all moving objects in R, populates each grid cell's list of overlapping queries, computes the initial result for every query and the trigger time when the result can change, and sends it to the respective client o.

4. Upon receiving the message from S, a moving client o_i initializes its variables with the correct values and populates its event priority queue, $eventQ$.

There are several important difference between MCQ-range and MCQ-kNN initializations. First, the message sent from o to S has the integer k_o instead of a fixed radius r. Second, the initial result computation is different, as the query radius is not fixed. The framework uses Algorithm 5 to compute the initial result for each query.

Algorithm 5: FindMCQ-kNN

```
input  : a mobile client $o$, an integer $k_o$
output: $k_o$ nearest neighbors
1  mark every cell in the grid as unvisited;
2  $currCells \leftarrow \{cell(o)\}$;
3  $R \leftarrow$ moving clients in $cell(o)$;
4  mark $cell(o)$ visited;
5  while $|R| < k_o$ do
6      $currCells \leftarrow$ neighbors($currCells$) $-$ $currCells$;
7      foreach cell $c \in currCells$ and $c$ is unvisited do
8          $R \leftarrow R \cup$ moving clients in $c$;
9          mark $c$ visited;
10 compute the distances between the moving clients in $R$ and $o$;
11 sort $R$ based on the distances;
12 keep only the nearest $k_o$ moving clients in $R$;
13 return $R$;
```

Third, contrary to the static query radius of a mobile client o, the radius of a query circle changes continuously with the client movements. Therefore, it is necessary to update the server once the alert region is changed for correctness.

Fourth, the handling of trigger times is different from MCQ-range. After the k_o nearest neighbors of a mobile client o are identified, the query result will not change until new mobile clients enter the query circle defined by the k_oth nearest neighbor.

3.5.2 Continuous Monitoring

Continuous processing is started once the initial processing is completed. It is responsible to keep the query results updated while mobile clients are moving around and some of them may change their results. Similar to MCQ-range continuous query processing, it is conducted in an incremental manner.

It is a challenging task to continuously process MCQ-kNN. Due to the movements of both query mobile clients and result clients, continuous processing of the queries can incur substantial communication cost and server workload. The framework minimizes these costs by following the same design philosophy as in MCQ-range. The server monitors the global picture, while clients take care of their own results. In particular, MCQ-kNN uses trigger times extensively, which can save substantial messaging cost between the server and the clients, because no message needs to be exchanged between server and clients in-between two trigger times. Recall that an internal trigger time is stored at a mobile client o_i. When it triggers, o_i needs to recompute its internal trigger time based on the current locations and velocities of its result clients. No message needs to be exchanged between o_i and its server. When an external trigger time occurs, on the other hand, the server needs to send the location and the velocity of the "move-in" client to o_i, which then recomputes its k_{o_i} nearest neighbors and the trigger.

Next, we explain how to compute triggers for a mobile client o. For each of the k_o nearest neighbors, o_i, except for the k_oth neighbor o_{k_o}, the client computes the time t_i when the distance between o_i and o is no less than the critical distance. That is, $D_{oo_i}(t) - D_{ook_o}(t) = 0$, where $D(t)$ is defined above. The trigger is then the minimum of all the positive roots. Similarly, for each of the clients o_j in the alert region but outside the query circle, the server checks if o_j moves into the query radius by comparing the D_{oo_j} with the critical distance. If D_{oo_j} is no more than the critical distance, then servers sends o_j's location, velocity and D_{oo_j} to o, which in turn determines its k_o nearest neighbors.

Unlike MCQ-range, mobile safe region is not used in MCQ-kNN because a mobile client has to update the server about its critical distance, together with their current locations and velocities in a single message. Now we present the detailed server procedure in Algorithm 6.

Algorithm 6: MCQ-kNN Server Procedure

```
1  foreach moving client $o$ that sends $\langle oid, \vec{p}_0, \vec{v}_0, d, lRes \rangle$ do
2      $C_o \leftarrow$ new alert region for $o$;
3      $O_o \leftarrow$ all mobile clients in $C_o$;
4      if $O_o - lRes \neq \emptyset$ then
5          $O_{res} \leftarrow \emptyset$;
6          foreach $o' \in O_o - lRes$ do
7              if $D_{oo'} \leq d$ then
8                  $O_{res}$.add($o'$);
9          if $O_{res} \neq \emptyset$ then
10             send $O_{res}$ to $o$;
```

From the analysis above, a mobile client

- keeps the server updated about the critical distance (query radius) while mobile clients are moving around,
- updates the k_oth nearest neighbor when a mobile client from inside the query circle replaces the old k_oth nearest neighbor by using trigger
- recomputes its nearest neighbors and the critical distance when the result set changes.

The detailed steps are summarized in Algorithm 7.

3.6 System Optimization

In this section, we outline how our system adapts itself to a highly dynamic environment by dynamically re-allocating service regions to servers.

Algorithm 7: MCQ-kNN Client Procedure

```
input : $O_{res}$ from the server
1 if $O_{res} \neq \emptyset$ then
2     recompute $k_o$ nearest neighbors;
3     update critical distance;
4     update $eventQ$;
5 else if $eventQ$..peek() $\leq currTime$ then
6     $(t, o_i) \leftarrow eventQ$.pop();
7     update critical distance to be $D_{oo_i}$;
8     update $eventQ$;
9 send to server $\langle oid, \vec{p}, \vec{v}, d, lRes \rangle$;
```

In the applications that we target, the objects issuing queries and being queries move continuously. At times, a significant number of these objects may locate in the service region of a single server, possibly due to an event that has occurred at that site. The server in question is then overloaded. Meanwhile, some other servers may be idle because there are only few moving objects to serve in their regions. Under such circumstances, a necessity arises to re-balance the workload among servers so that the quality of service is kept stable.In order to detect this kind of circumstance, we follow a simple yet effective strategy to estimating each server's status. We use the number of moving clients as an indicator. Ideally, each server has $\frac{N}{M}$ moving clients if the system has N objects and M servers. A server is overloaded if it is serving more than $\frac{\beta N}{M}$ objects, where $\beta > 1$. If β is too large, a server may get overloaded and unnoticed, and the performance of the system deteriorates. But if β is too small, system re-balancing happens too often, which also wastes system resources. In our experiments, we set $\beta = 3$.

3.6.1 Adjusting the Service Region Allocation

When skew causes a server S to become overloaded, S divides its service region into two halves, and hands over the moving objects in the one half service region to a lightly utilized server S' so as to balance the workload. Next, the framework also seeks to merge *compatible* regions that are adjacent, and have the same width and height, in order to (1) prevent space fragmentation and (2) to further balance the workload.

Algorithm 8: DynamicAllocation

```
1  while $S$.numClients() $> \frac{\beta N}{M}$ do
2      $R \leftarrow$ getServiceRegions();
3      $(R_1, R_2) \leftarrow$ divideRegion($R$);
4      $S' \leftarrow$ the server with the least clients;
5      if $R_1$.numClients() $>$ $R_2$.numClients() then
6          $R' \leftarrow R_2$;
7      else $R' \leftarrow R_1$;
8      allocate($S, S', R'$);
9      foreach $o \in R'$.getObjSet() do $o.currS \leftarrow S'$;
10 foreach Region $R_1$ do
11     foreach Adjacent Region $R_2$ do
12         $n \leftarrow R_1$.numClients() $+$ $R_2$.numClients();
13         if compatible($R_1, R_2$) $\wedge$ ($n < \frac{\beta N}{M}$) then
14             $S_1 \leftarrow$ getServer($R_1$); $S_2 \leftarrow$ getServer($R_2$);
15             if $S_1 \neq S_2$ then
16                 allocate($S_1, S_2, R_1$);
17                 $R \leftarrow$ merge($R_1, R_2$);
18                 $S_2$.setServiceRegions($R$);
19                 if $S_1$.getServiceRegions() $= \emptyset$ then
20                     $R_m \leftarrow$ the region with the most objects;
21                     $(R'_1, R'_2) \leftarrow$ divideRegion($R_m$);
22                     allocate($S_m, S_1, R'_1$);
23             else merge($R_1, R_2$);
```

In the server cluster, each server continuously monitors the number of clients in its service region. When a server S is overloaded for a period longer than t_d, Algorithm 8 is invoked to dynamically adjust the service region allocation, in order to balance the workload among the servers. Suppose server S with a service region R is overloaded.

In Algorithm 8, the function getServiceRegions returns the current set of service regions. The function numClients returns the number of clients in a service region as monitored by a server. Each server by definition should have one region, but the region may not be a connected region. In line 3, the function divideRegion divides a given region into two equal halves. In line 4, the server S gets another server, S', with the least clients by sending broadcast messages to the server cluster. Then the half region with fewer clients is allocated to S' (lines 5–8). The function allocate(S, S', R') allocates the region R' belonging to S to

S'. Each object in R' then updates their server to S' (line 9).

In next loop, every pair of adjacent service regions are compared to see if they can be merged (line 13). If any two compatible regions of different servers are merged, the new region is allocated to one server (lines 14–18). Next, if the other server is idle, the server with the heaviest workload allocates half of its service region to the idle server (lines 19–22). On the other hand, if two compatible regions belong to the same server, they are simply merged (line 23).

3.6.2 Dynamic Cell Side Lengths

As discussed above, when the system is initializing, the bootstrap server picks up a value for its cell length, calculates its σ, with which the remaining servers calculates their respective cell length α with Equations 3.1 and 3.2.

However, a need for dynamically adjusting the cell side length arises when data are highly skewed at certain servers. In our setting, each server monitors the distribution of the mobile clients in its service region. If a server detects skew, it is allowed to adjust its cell length. Specifically, if a server finds that a single cell holds more than γ of all objects in the service region for a period longer than a threshold t_e, the server halves its current cell side length and rebuilds its grid. This process continues until no cell has more than γ of total moving clients. Once the distribution of moving clients is no longer skewed, the server restores its original cell length, which was saved before the adjusting occurred. Note that cell side length adjust should not happen very often because of the expensive index rebuild. In the experiments, we empirically set γ to be 50% and set t_e to be 10 time points.

3.6.3 Extension to Multiple MCQs by One Client

We also consider relaxing the assumption that one moving client is issuing one MCQ at any time. Our system could also be used to satisfy the needs of a mobile client to issue many MCQs. In this case, many queries may be sharing the same focal object or the same alert region, and a query grouping technique may be employed to reduce both the communication cost and the computation workload at server side. Specifically, if a moving client issues three MCQs at the same time, and thus three alert regions at server side. The server could group these MCQs and return the candidates in one message instead of three messages. Server could also use minimal encoding to indicate in which alert region a candidate is located in order to reduce client side processing.

3.7 Experiments

In this section, we evaluate the effectiveness and efficiency of the proposed distributed architecture and algorithms. All experiments are carried out on a Linux machine with an Intel(R) Core2 Duo 2.33 GHz CPU and 16GB memory. Each simulated server is an object that records various statistics, e.g. workload and number of messages. The servers are assumed to have a mechanism to communicate with each other. With the data sets, our simulation replays the movement of the moving clients, and records the server workload and the messages sent by both clients and servers. We use the following metrics for our performance study: (1) query result change rate, (2) message rate between servers, (3) message rate between server and clients, (4) server workload (running time for processing the messages), and (5) workload at moving object side (average number of queries each moving object monitors).

As the RMD scheme [94] cannot be directly applied to solve the MCQ problem, we need to adapt it for MCQ problem. Instead of finding pairs in proximity, RMD is modified so that the server finds the result for each client, with the contraction and expansion properties still kept. We set the parameters in the RMD algorithm according to the recommendations used in [94], namely the scale factor has default value 2. We compare MCQ-kNN with DKNN [89] for moving continuous kNN queries.

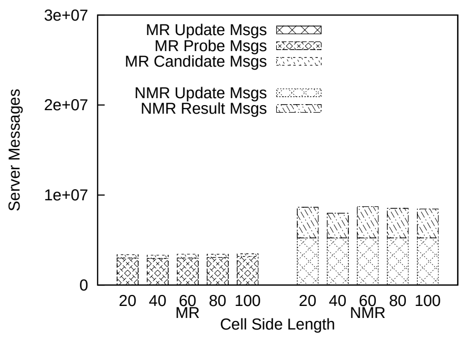
The data sets were generated using Brinkhoff's road-network-based moving-objects-generator [11]. The moving object datasets consist of the location update reports for each moving object, at each time point. Three default velocity values (slow, middle, and fast) are used to generate data of objects moving at different speeds. The Oldenburg map is used to generate trajectories. The generated data space has width 23,572 meters and height 26,915 meters. Our simulator generates cells that cover this area. As we have discussed, the cell side length α is an internal parameter for each server; its impact on the system performance is studied in our experiments. All parameters under investigation are gathered together in Table 3.2.

Parameter	Default	Range
Query radius r (meters)	20	10–50
k	4	2–10
Mobile region radius λ (meters)	20	10–50
Cell side length α (meters)	40	20–100
Number of Moving Clients N	300k	100k–500k
Number of Servers M	8	4–12

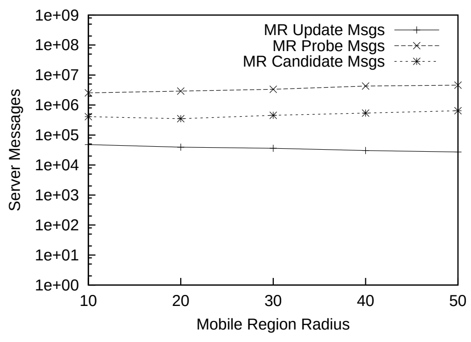
Table 3.2: Experimental Parameter Settings

The organization of the experiment part is as follows. First, we separately show the results of comparing MCQ-range with RMD and comparing MCQ-kNN and DKNN. Then we conduct experiments to show the effectiveness of the system architecture by running both MCQ-range and MCQ-kNN on it.

3.7.1 MCQ-Range: Varying Grid Side Length



(a) Effect of Cell Length



(b) Effect of Mobile Region Radius

Figure 3.3: Server Messages

We first investigate the effect of the (default) side length of grid cells. Remember that setting grid cell side length at the bootstrap server also determines the grid cell side length of other servers during initialization phase. In this set of experiments, we set various cell side lengths at the bootstrap server. This comparison is only between the two variants of scheme, the scheme without mobile safe region (NMR) and the scheme with mobile safe region(MR), since these settings do not apply to RMD. Figures 3.4 and 3.3(a) show results for computational load and exchanged messages, respectively. We observe that performance is best with a cell length of 40 meters, which we choose as our default setting.

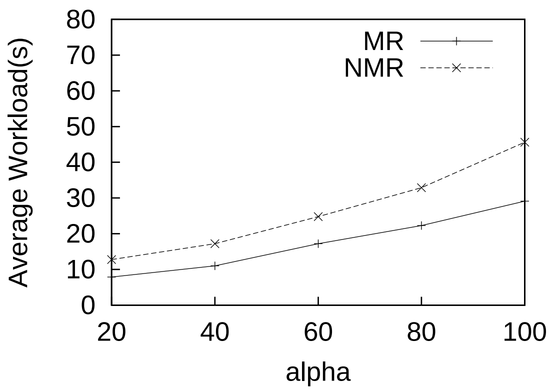


Figure 3.4: Server Workload vs. Cell Length

3.7.2 MCQ-Range: Varying Mobile Region Radius

We now evaluate the effect of the mobile (safe) region radius on both servers' workload and communication cost with our MR variant. Figure 3.3(b) shows our results. As expected, as the size of mobile regions increases, the server has to send out probe more messages. Hence the communication cost increases at the sever-side. On the other hand, as the size of safe regions increases, such events as a client exits its safe region become less frequent, and hence the communication cost of uplink message sent by clients drops.

3.7.3 MCQ-Range: Client Handover

In this set of experiments, we examine the number of moving clients that actually cross the boundaries of two servers. Both the total number and the speed of moving clients affect the number of handovers. Therefore, we generate data sets with size varying from 100K to 500K. The speed for each data set can be either slow or fast, which is the default value of the data generator. From the result shown in Figure 3.5(a), we observe that for each data set with slow speed, the handover rate is around 1% with smaller variation.

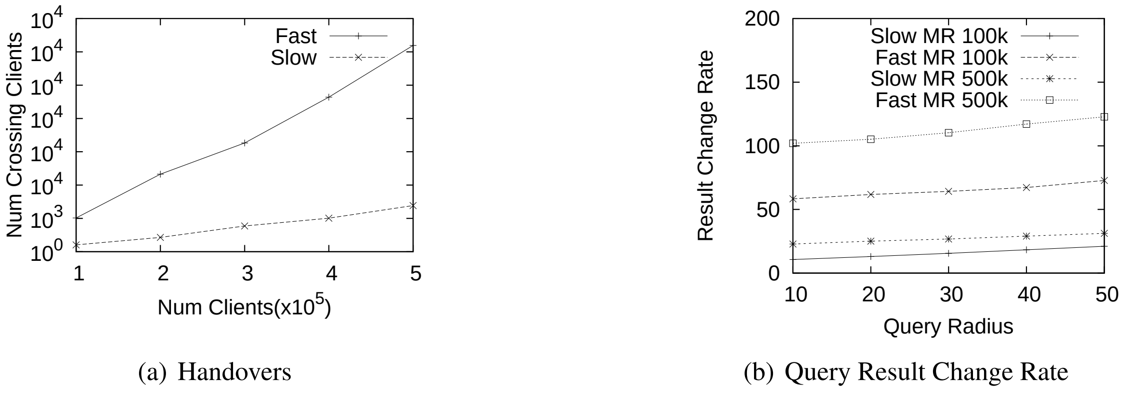


Figure 3.5: Handovers and Result Change Rate

In contrast, for each data set with fast speed, the handover rate can be as high as 6%, with bigger variation. As expected, the speed of moving clients has a great impact on the rate of handovers. It also causes bigger variations because the servers have shorter time to respond to the event that many moving clients cluster in one region. In summary, this set of experiments shows that the proposed architecture is effective in adapting to highly dynamic environment.

3.7.4 MCQ-Range: Query Result Change Rate

In this set of experiments, we conduct a sensitivity test on how the population size and the speed of moving clients affect the result change rate. We generated four different data sets: Slow100k, Fast100k, Slow500k, and Fast500k. As the names suggest, we have two data sets with 100k moving clients, and two with 500k moving clients. The speed for each data set can be either slow or fast which are the default values of the generator. We measure the average times that moving clients result changed for each time point against query radius. Figure 3.5(b) shows the results. We observe that both higher speed and larger population increase the result change rate. However, speed has greater impact than population.

We only show the result for MR algorithm. NMR exhibits the same results since speed and population affect only performance, not correctness. This set of experiments show that both MR and NMR can capture the frequent MCQ-range query result changes. Unlike centralized systems where the query processing is based on periodical location updates, MR and NMR perform true continuous monitoring. In MR and NMR, servers handle fast changes as they perform light computations, delegating part of the workload to the moving clients. Thus, they avoid the problem that the more frequently the query result changes, the more likely that it is the server gives wrong results as it occurs in a centralized solution due to the delay of updates.

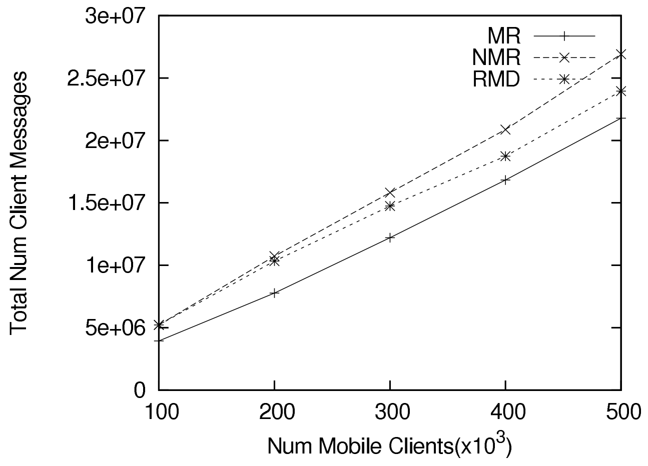


Figure 3.6: Client Messages

3.7.5 MCQ-Range: Effect of Number of Moving Clients

We also study the scalability of our method in the number of moving clients. Figure 3.6 shows that NMR has the most client messages, whereas MR has the fewest client messages, thanks to the use of mobile regions. For the same reason, RMD has fewer messages than

NMR. In Figure 3.7, both NMR and MR outperform RMD because of the push of computation to the client side. Quite a number of messages (about one order of magnitude) are actually saved in both MR and NMR. There are more messages in MR because of probe messages sent by servers. Figure 3.8 shows server workloads for the three methods. We compare total time of the four servers in MR and NMR with the time in RMD for fairness. Both MR and NMR outperform RMD by more than 50% in terms of computation time at the server side. These results indicate the scalability advantage of our methods in comparison to the RMD scheme. Again, these advantages are due to the exploitation of computational capacities at the client side.

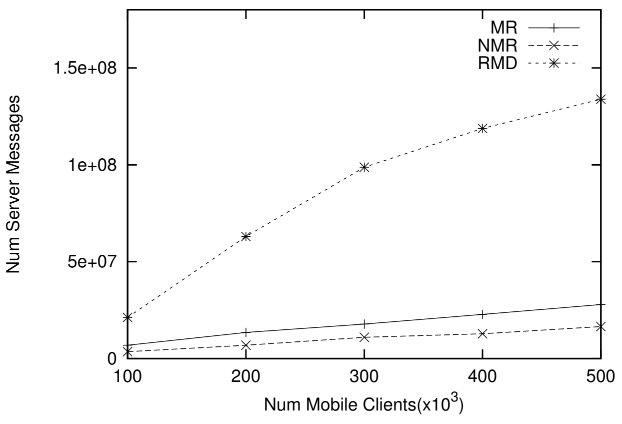


Figure 3.7: Server Messages

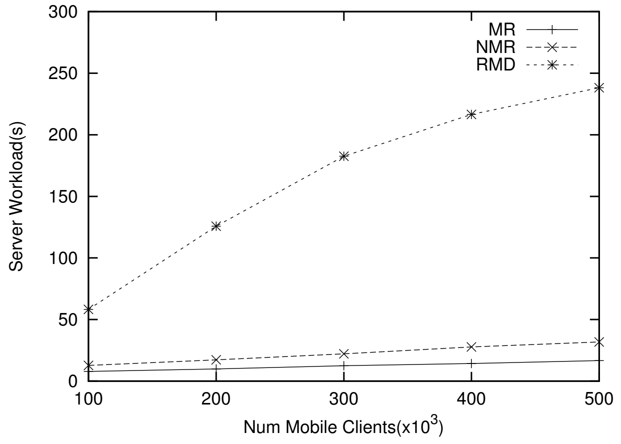


Figure 3.8: Server Workload

3.7.6 MCQ-range: Varying Query Region Radius

We also experimented with tuning the query region radius. The advantage of our system is allowing each client to select its customized query radius. But in order to compare with RMD, which assumes the same radius for each moving client, we set query radius to be the same value. Results are shown in Figures 3.9(a) and 3.9(b). It is observed that as the query radius increases, the number of cells in each client's alert region also increases, hence the number of candidate message sent by servers increase with all algorithms.

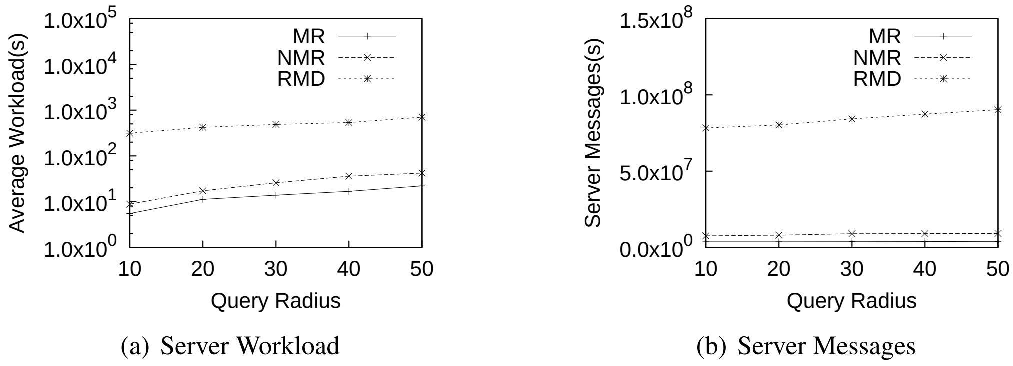


Figure 3.9: Effect of Query Region Radius

Still, both variants of our scheme outperform RMD in this experiment. This result verifies our expectations and reconfirms the advantages of our method. It also observed that MR improves the performance by a factor of 6 compared to RMD.

3.7.7 MCQ-kNN: Effect of Number of Moving Clients

Figure 3.10 and Figure 3.11 compares the number of messages sent by clients and by servers in both DKNN and MCQ-kNN.

Recall that in DKNN, the space partitioned is pre-determined by base stations and their transmission radii, whereas MCQ-kNN has a mechanism to dynamically adjust the space partitioning. The flexibility leads to a reduction in the messaging cost in MCQ-kNN by a factor around 10 in both client and server side messaging cost. There is a trade-off for saving the communication cost, as is shown in Figure 3.12 where MCQ-kNN spends around

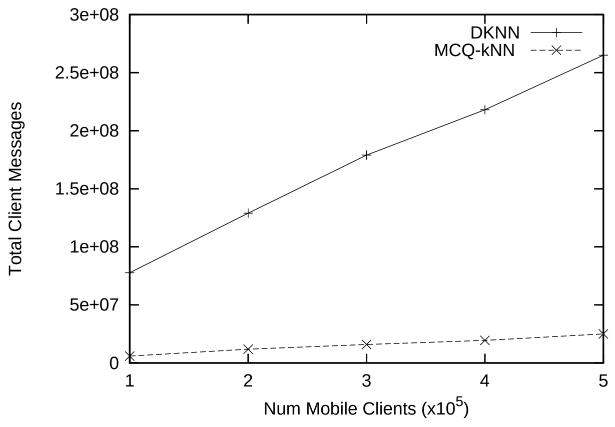


Figure 3.10: Client Messages, kNN

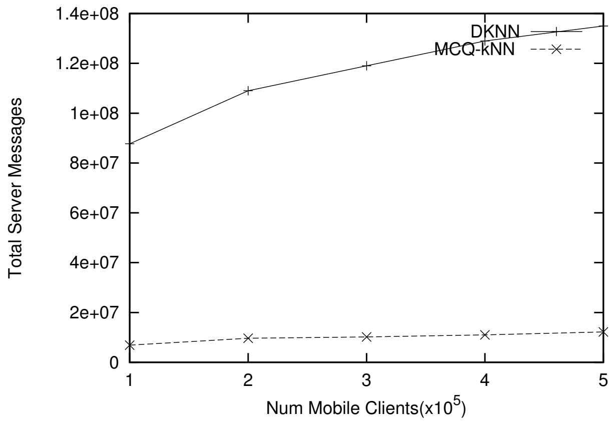


Figure 3.11: Server Messages, kNN

twice processing time at server side. This trade-off is worthwhile because in the distributed setting where both DKNN and MCQ-kNN both assume, communication cost contributes to the most of the total cost.

3.7.8 MCQ-kNN: Varying k

We compare DKNN and MCQ-kNN by varying k. The result is shown in Figure 3.13.

We observe that MCQ-kNN outperforms DKNN for every k value in terms of messaging cost. In addition, compared to DKNN, the increase in messaging cost of MCQ-kNN with the increase of k is less dramatic. The reason is that DKNN cannot handle data skew very well, especially when the number of mobile clients is big. Thanks to the macro and

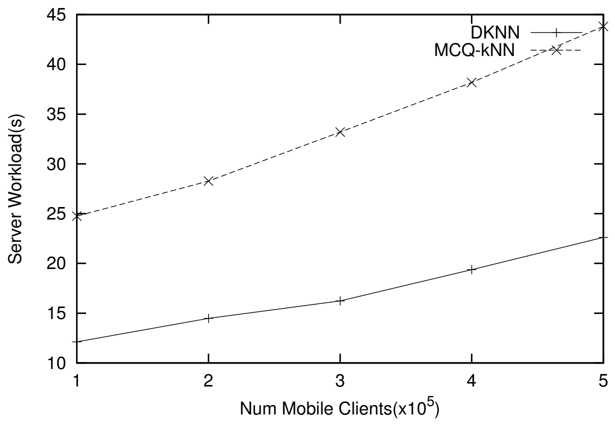


Figure 3.12: Server Workload, kNN

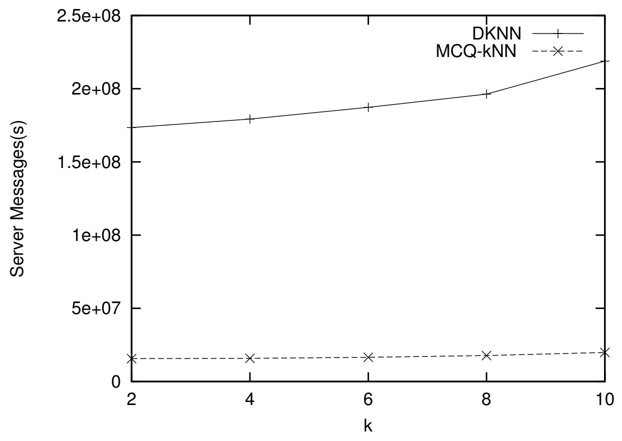
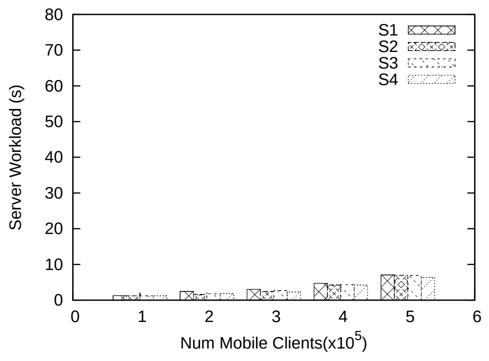


Figure 3.13: Server Workload vs k

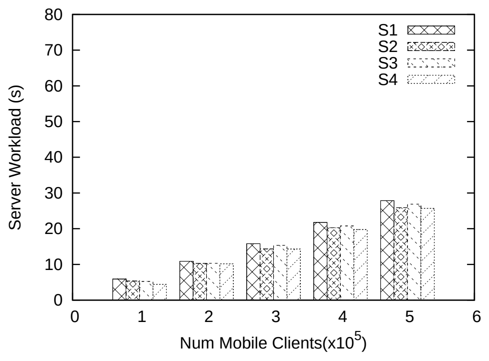
micro level tuning mechanisms, our system handles data skew better, and thus MCQ-kNN has better scalability than DKNN.

3.7.9 Effectiveness of Server Architecture

In our system, servers divide their service regions among themselves according to the distribution of moving clients therein so as to balance their workloads. In this set of experiments, we try various numbers of moving clients, and check the workload of each server. Figures 3.14(a), 3.14(b) and 3.15 show our results on each server's workload (i.e., time taken) as a function of the number of moving clients. Note that server 1 is pre-selected as the bootstrap server, therefore it has a bit more workload than the other servers. But we



(a) Server Workload, MR



(b) Server Workload, NMR

Figure 3.14: Server Workloads, Range

still observe that the servers share the workload almost equally.

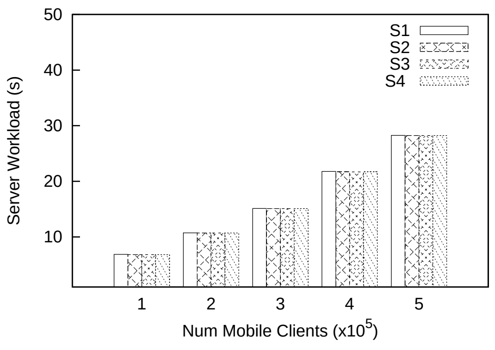


Figure 3.15: Server Workload, kNN

3.7.10 Effect of Number of Servers

We also study the scalability when the number of servers increases. Figure 3.16 presents the amount average workload on each server when the number of servers increases from 4 to 12. It is observed that with the increase of number of servers, the average workload decreases *linearly* which indicates that the architecture is able to scale well.

3.8 Summary

In this chapter, we address the problem of answering multiple moving continuous queries in an environment where each moving client poses its own range query. We propose a dis-

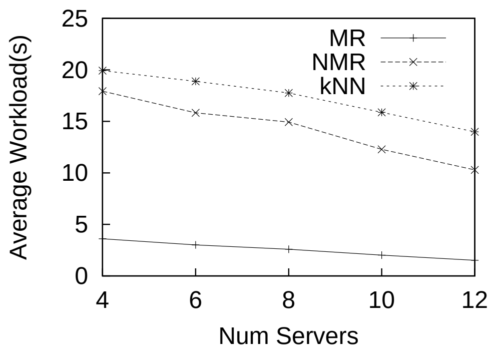


Figure 3.16: Effect of Number of Servers

tributed client-server architecture for efficiently processing such queries. In our solution, a cluster of interconnected servers takes care of the global view, while part of the computation is relegated to the clients to verify actual results. We demonstrate the effectiveness and efficiency of our approach through extensive simulation experiments. Our method reduces both the server-side workload and the client-server communication cost in comparison to the most recent state-of-the-art scheme, while it is much more scalable to growing number of moving clients.

Chapter 4

Processing Group Movement Query

4.1 Introduction

In the previous chapter, we have discussed real-time processing of range and kNN queries. In this chapter, we extend our time scope to also include the movement histories of moving objects in order to find interesting movement patterns.

The increasing availability of trajectory data renders it increasingly important to be able to extract movement patterns from trajectory data. A central pattern is that of objects that have traveled together. Existing definitions, notably *flock* [34, 35], *convoy* [47], and *swarm* [61], capture different notions of objects traveling together. For example, a flock consists of objects that travel together within a disk of a user-specified radius. However, the disk radius influences the result greatly and is a difficult-to-set parameter. Convoys, on the other hand, do away with this parameter by employing the notion of density-connectedness. Swarms also relax the requirement that objects must form groups for consecutive time points.

We may think of the true trajectory tr_{T} of a moving object as a continuous function from the time domain $\mathbb{T}$ to points in the n-dimensional Euclidean space $\mathbb{R}^n$ where the movement occurs. For most typical applications, $n = 2$. In practice, the trajectory of an object is collected by sampling the object's positions according to some policy, resulting in a set of sampling points $(t, \vec{r}) \in \mathbb{T} \times \mathbb{R}^n$. A trajectory tr is then given by the stepwise linear function obtained by connecting temporally consecutive sampling points with line segments, as shown in Figure 4.1(a). Different sets of sampling points may represent the same such function. This holds for $\{(3,3),(7,7)\}$ and $\{(3,3),(5,5),(7,7)\}$, shown in Figure 4.1(b).

The use of different representations (sampling points) of the same trajectory in an algorithm should not affect the outcome of the algorithm. We call this property *sampling independence*.

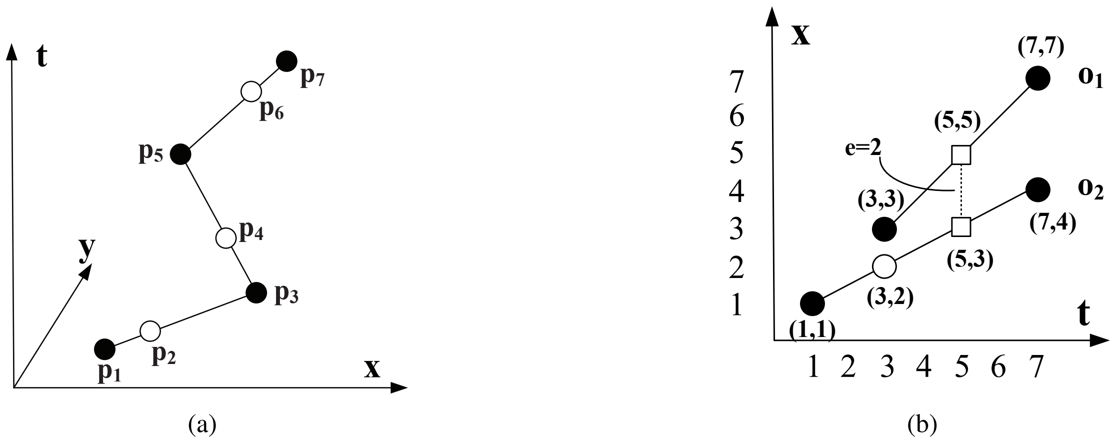


Figure 4.1: Trajectory Semantics and Pattern Loss

Definition 5 *(Sampling Independence[1]) Let two trajectory sampling point sets tr_a^{rep} and tr_b^{rep} that represent the same trajectory be denoted as $tr_a^{rep} \equiv tr_b^{rep}$. Next, two collections of trajectory sampling point sets, TR_0 and TR_1, represent the same trajectories, denoted as $TR_0 \equiv TR_1$, if and only if $\forall i \in \{0,1\}$ $(\forall tr_a^{rep} \in TR_i$ $(\exists tr_b^{rep} \in TR_{1-i}(tr_a^{rep} \equiv tr_b^{rep})))$.*

With these definitions in place, we say that an algorithm A operating on collections of trajectory point sets satisfies sampling independence *if and only if* $\forall$***TR***$_a$, $TR_b(TR_a \equiv TR_b \Rightarrow A(TR_a) = A(TR_b))$.

Sampling independence has several benefits. First, we observe that sampling-dependent approaches that simply compare trajectories as of given sampling times suffer from the lossy-pattern problem. Consider the trajectories of two objects o_1 and o_2 in Figure 4.1(b).

[1]While the term "sampling point independence" may be more precise, we use the shorter term "sampling independence" throughout this chapter.

Solid circle points are real sampling points. Hollow circle and square points are expected locations of the two objects. Assume that for a set of objects to form a group, their distance e must not exceed 2 for at least 2 time units. The two trajectories satisfy this requirement because they are within distance 2 from time 3 to time 5. However, the two trajectories are sampled at different times. If we introduce an artificial sampling point for o_2 at time 3, which existing techniques (e.g., convoys and swarms) would do to enable comparison, no group is found because we are missing a sampling point at time 5. Sampling independence rules out such approaches.

Second, the approach of adding sampling points so that all points are sampled at the same times is not scalable. To see why, assume a trajectory collection TR in which each trajectory is sampled S times, and let the sampling times be chosen at random from a continuous time domain. Each time a new trajectory is added to the collection, a total of $2|TR|S$ artificial sampling points must be added (S to each existing trajectory, and $|TR|S$ to the new trajectory). While these assumptions are extreme, the argument illustrates well that sampling dependence does not scale with $|TR|$ and S.

Third, sampling independence offers a foundation for seamless integration of approximate trajectories. With sampling independence, a computation does not have to occur at each sampling time, and approximate trajectories can be used seamlessly in place of more accurate trajectories.

Sampling independence requires decoupling defining and finding travel-together patterns from sampling points. Among the existing definitions, only flocks satisfy this property, while the other definitions, e.g., of convoy and swarm, are based on sampling points and so cannot be used in a sampling-independent framework.

Our framework achieves sampling independence by first defining a travel-together pattern that is independent of sampling points, and second, by adopting an event based approach that is also sampling independent.

The second property that the framework considers is density-connectedness that aims to avoid the lossy-flock problem identified by Jeung et al. [47]. The problem occurs because (1) a flock is very sensitive to a user-specified disc size that is independent of the data distribution. and (2) the use of circular shape may not always be appropriate. To achieve this property, our framework employs DBScan [24] for continuous clustering.

The third property of the framework is to support online processing, where sampling data arrives continuously. Efficiency is a key factor in online processing. Processing the raw trajectories may be too time consuming, so the framework needs to support online trajectory simplification that trades result accuracy with efficiency. This is the fourth property of the framework.

Table 4.1 categorizes previous work and our proposed solution (denoted Group) with respect to the four requirements as discussed above. Previous work is discussed in detail in Chapter 2

In our framework, a *group* is a cluster that has at least m moving objects being density-connected for at least a specified *duration* of time. This definition fundamentally differs from sampling dependent definitions (e.g., convoy and swarm) by requiring moving objects to be density-connected for a time duration instead of at consecutive or non-consecutive sampling times (continuous vs. discrete).

Property	**Flock**	**Convoy**	**Swarm**	**Group**
Sampling independence	✓	✗	✗	✓
Density connectedness	✗	✓	✓	✓
Traj. approximation	✓	✓	✓	✓
Online processing	✓	✗	✗	✓

Table 4.1: Algorithm Comparison

To find groups from a collection of trajectories without relying on sampling points, our main idea is to predict events that can happen in the future according to the motions of the moving objects. This is carried out by the *continuous clustering* function module in the framework (Figure 4.2). This module first uses DBScan to find clusters with an arbitrary spatial extent and to avoid a disk radius parameter. As new data arrive, these clusters are continuously monitored. Objects exiting or joining a cluster and cluster expiry or merging are all modeled as future events, and then processed when they actually occur. The event-based approach enables sampling independence—computation occurs when events occur, not at sampling times. This design also enables online processing: as new trajectory samples arrive, events can be derived and processed in due course. There is no need to recompute the entire result. Online trajectory simplification is applied when new data arrive in order to smoothen the trajectories and thus reduce the total number of events.

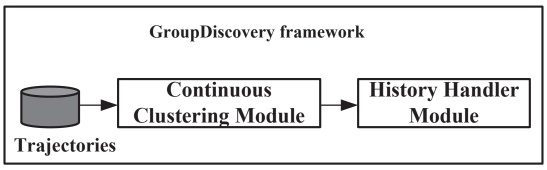


Figure 4.2: The Sampling Independent Framework

The second module serves to handle efficiently the history of each existing group to find groups to return to the user. This module inherently has high time complexity because it potentially has to consider exponential numbers of subgroups. To improve its efficiency, we avoid materializing every subgroup by exploiting the relations between subgroups. The main idea is that a group may have many subgroups that may contain a subset of its objects or may exist for a sub-interval of its time interval. Such dominated subgroups do not offer new information and are pruned early.

Next, two groups may share too many common members. We propose a similarity measure for groups and return only a set of groups such that no two groups are more similar than a given similarity threshold.

Further, we score groups according to their cardinality and duration and then return the top-k groups. To enable this functionality, the framework maintains a candidate result list and a variable $minScore$ that is the lowest score among all the candidates found so far. For each group to be scored, the framework first computes an upper bound on the score of the group for its entire lifespan. The group is pruned if the upper bound is smaller than

$minScore$. Otherwise, possible groups with higher scores in the cluster are inserted into the candidate list, and $minScore$ is updated.

In Section 4.2, we formalize the notions covered above.

The main contributions of this chapter are as follows.

- We propose a novel definition of *group* that we believe enables the computation of interesting patterns.
- We propose a corresponding, efficient group discovery framework that is the first to satisfy the four requirements listed in Table 4.1.
- We conducted an extensive empirical study on both real and synthetic datasets to evaluate the proposed framework. Our results offer insight into the effectiveness and efficiency of the proposed framework.

The remainder of the chapter is organized as follows. Preliminaries and Definitions are covered in Section 4.2. We describe the GroupDiscovery framework in Section 4.3. Section 4.4 reports on the empirical evaluation of the GroupDiscovery framework. Finally, Section 4.5 summarizes this chapter.

4.2 Preliminaries and Definitions

4.2.1 Definitions

We proceed to define the group query along with supporting definitions. The notations used in this section and throughout the chapter are summarized in Table 4.2.

As we aim to return interesting groups to the user, we introduce a scoring function for groups.

Definition 6 *(Scoring Function) Let C be a set of N_c objects that travel together during a time interval $[t_s, t_e]$. The score of C is:*

$$\mathrm{Score}(C) = \alpha N_c + (1 - \alpha)\rho \ ,$$

N	Total Number of trajectories
N_c	Cardinality of a group
δ	Tolerance for line simplification
τ	Duration threshold
e	Distance threshold in density-based clustering
m	Cardinality threshold in density-based clustering
ρ	Duration of a cluster
θ	Similarity threshold

Table 4.2: Symbols Summary

where $\alpha\,(0 \leq \alpha \leq 1)$ *is a user-provided parameter and* $\rho = (t_e - t_s)$.

The idea underlying the definition is that larger and longer groups are more interesting than smaller and shorter groups. Parameter α allows us to balance cardinality and duration.

Next, we define a domination relation that allows us to prune groups that do not contribute any new information. Some other related work, e.g., [34] and [61], has similar notions, either explicitly or implicitly.

Definition 7 *(Domination)*

Given two groups C_1 *and* C_2*, we say* C_1 dominates C_2*, denoted by* $C_1 \succ C_2$*, if:*

1. C_1 *is a superset of* C_2*, and*

2. *the time interval of* C_1 *contains that of* C_2.

Now we are ready to define the group query to be supported.

Definition 8 *(Group Query) Given a trajectory collection TR, a distance threshold* e*, a cardinality threshold* m*, a duration threshold* τ*, an integer* k*, and the scoring function given above, the group query returns* k *groups of objects, so that each group*

1. *is a density-connected cluster with respect to* e *and* m*,*

2. *has a duration no shorter than* τ*,*

3. *is not dominated by any other group, and*

4. *has a top-*k *score.*

4.3 Group Discovery Framework

In this section, we describe the GroupDiscovery framework and its two functional modules in detail. This online algorithm (Algorithm 9) first initializes the variable U to store the unclassified objects, H to store the history of the clusters, and R to store the result. It then receives new trajectory data at each iteration, and calls Normal Opening Window (NOPW) algorithm [64] to simplify the trajectories. Recall that the NOPW algorithm is an *online* algorithm that employs a sliding window to simplify trajectories. Please refer to related works in chapter 2 for a detailed description of how NOPW works. These simplified trajectories are then processed by the Continuous Cluster and History Handling modules to find groups.

Algorithm 9: DiscoverGroups($e, m, \tau, k, \delta, \theta$)

```
1 $U \leftarrow \emptyset$; $H \leftarrow \emptyset$; $R \leftarrow \emptyset$;
2 while true do
3     $TR \leftarrow$ receiveNewTR();
4     NOPW($TR, \delta$);
5     FindContinuousCluster($TR, e, m, \tau, k, \delta, U, H$) ;
6     $S \leftarrow$ RevHist($H$);
7     CheckCandidates($S, R, \theta, k$);
```

4.3.1 Continuous Clustering Module

4.3.1.1 Overview

Given a trajectory collection TR, we aim to discover groups of objects that travel together. A trajectory represents the movement history of an object for some time interval. As all trajectories are not defined for the same time interval, TR represents the movement of a dynamic set O of objects. The objects in O at time t can be retrieved by the function $getObjects(TR, t)$.

The overall approach is to maintain a heap $eventQ$ that contains events that affect the clustering of the moving objects. Each event is associated with the time when it occurs, and the heap is ordered on these times. Events are inserted into $eventQ$ that correspond to the object movements given by the trajectories in TR. The module then processes events in $eventQ$ in time order, causing additional events to be entered into $eventQ$ and eventually processed.

An event has the format $(t, obj, cid_1, cid_2, type)$, where t is the time when the event occurs, obj is an object, cid_1 and cid_2 are cluster identifiers, and $type$ is the event type, which is one of the following.

- APPEAR: an object appears in the system. Four mutually exclusive cases can happen: The object may stay by itself being unclassified; it may join an existing cluster, thus causing a JOIN event to be created; it may cause multiple clusters to merge, thus causing one or more MERGE events to be created; or it may form a new cluster with other objects.

- DISAPPEAR: an object disappears from the system. As for object appearance, there are four cases: No clusters are affected; a cluster loses a member, causing an EXIT event to be created; a cluster splits into multiple smaller clusters; or a cluster expires, which results in the creation of an EXPIRE event.

- UPDATE: an object updates its velocity, which affects the predicted exit time of object(s), which in turn possibly affects the merging or expiry of clusters.

- EXIT: an object exits from a cluster. The exit event can cause a cluster to expire, e.g., because it has fewer than m members, in which case an EXPIRE event is inserted into $eventQ$.

- JOIN: an object joins a cluster.

- EXPIRE: a cluster expires. When this occurs, the former cluster members may be in no new cluster, rendering them unclassified.

- MERGE: multiple (at least two) clusters merge.

- SPLIT: one cluster splits into multiple clusters.

Among these events, the first three events, APPEAR, DISAPPEAR, and UPDATE, can be read directly from the data stream, and thus are called *primary events*. In contrast, EXIT, JOIN, EXPIRE, MERGE, and SPLIT are *secondary events*.

Each object o in the system has a data structure $(oid, \vec{p}, \vec{v}, cid, label)$, where oid is its identifier, $\vec{p}$ and $\vec{v}$ are its current location and velocity, and cid is the identifier of the cluster o currently belongs to, if any. An object o initially has $o.cid = 0$, indicating that it is unclassified. The field $label$ represents the type of the object in the cluster. In DB-Scan [24], clusters contain two types of objects. A *core object* has at least $MinPts$ objects in its e-neighborhood, where $MinPts$ is a user-specified parameter that we set to m. Any other object is a *border object*. A core object has ($o.label = \mathrm{CORE}$); a border object has ($o.label = \mathrm{BORDER}$).

The top-level algorithm keeps the currently unclassified objects in a set U, and attempts to cluster them each time an event occurs. In addition to $eventQ$, we employ a mapping getMemByCid from cluster identifiers to the sets of objects in a cluster.

Upon termination of the algorithm, the histories of clusters are recorded in a data structure that is passed on to the history handler module for computation of domination relations and thus the true groups.

4.3.1.2 Event Processing

Algorithm 10 contains the pseudo-code of FindContinuousCluster, the top-level continuous clustering algorithm. It takes the following arguments: a trajectory collection TR, the four parameters e, m, τ, and k from the group query definition, δ for RangeQuery, and U and H for storing the unclassified objects and the cluster history, respectively.

The algorithm initially scans TR and inserts APPEAR, DISAPPEAR, and UPDATE events into $eventQ$. The algorithm then repeatedly pops and processes events. Given an event $(t, o, \perp, \perp, \mathrm{APPEAR})$, o is inserted into U. Given an event $(t, o, \perp, \perp, \mathrm{DISAPPEAR})$,

Algorithm 10: FindContinuousCluster($TR, e, m, \tau, k, \delta, U, H$)

```
insert new primary events into $eventQ$;
while $eventQ$ is not empty do
    $evt \leftarrow eventQ.pop$;
    $O \leftarrow$ getObjects($TR$, $evt.t$);
    $C \leftarrow$ getMemByCid($evt.cid$);
    if $evt.type = \text{APPEAR}$ then $U.\text{add}(evt.obj)$;
    else if $evt.type = \text{DISAPPEAR}$ then
        HandleObjDisapp($evt, O, H$);
    else if $evt.type = \text{UPDATE}$ then
        HandleObjUpdate($evt, O, H$);
    else if $evt.type = \text{EXIT}$ then
        $o \leftarrow evt.obj$; $C.\text{delete}(o)$; $o.cid \leftarrow 0$;
        $U.\text{add}(o)$; update $H$;
        if $o.label = \text{BORDER}$ then CheckCore($o, C$);
        if $o.label = \text{CORE}$ then
            $L \leftarrow o.\text{getNeighbors}()$;
            foreach $o' \in L$ do
                if $o'.label = \text{BORDER}$ then
                    $L' \leftarrow o'.\text{getNeighbors}()$;
                    if $L'$ has no core objects then
                        $C.\text{delete}(o')$; $o'.cid \leftarrow 0$; $U.\text{add}(o')$;
        if $(|C| < m) \vee (C$ has no core objects$)$ then
            insert $(evt.t, o, C.cid, \perp, \text{EXPIRE})$ into $eventQ$;
    else if $evt.type = \text{JOIN}$ then
        $o \leftarrow evt.obj$; $o.cid \leftarrow C.cid$;
        $C.\text{add}(o)$; $U.\text{delete}(o)$;
        update $H$;
        if $o.label = \text{BORDER}$ then
            $t \leftarrow$ getExitTime($o, C$);
            insert $(t, o, C, \text{EXIT})$ into $eventQ$;
    else if $evt.type = \text{EXPIRE}$ then
        mark members in $C$ unclassified;
        update $H$;
    else if $evt.type = \text{MERGE}$ then
        HandleMerge($evt, O, H$); update $H$;
    else if $evt.type = \text{SPLIT}$ then
        HandleSplit($evt, O, H$); update $H$;
    Insert($U, O, e, m, \tau, \delta, H$);
```

o is removed from the system. If o is a member of the cluster C, o is also removed from C which then expires if it has less than m members.

Given an event $(t, o, \perp, \perp, \mathrm{UPDATE})$, if o is a border object of the cluster C identified by cid, o's exit time is re-computed and updated in $eventQ$. If o is a core object, the algorithm re-computes the exit times of the border objects in o's e-neighborhood and the expected split time, and updates the $eventQ$. Lastly, if o is a unclassified object, the procedure Insert is eventually called to cluster o.

Given an event $(t, o, cid, \perp, \mathrm{EXIT})$, the object o is removed from the cluster identified by cid and marked as unclassified. If o is a border object, CheckCore procedure is called to check whether o's core object still has enough neighbors. If o is a core object, o's border neighbors that are not density-connected by any other core objects are also removed. The cluster identified by cid expires if it has fewer than m members after the event. Given an event $(t, o, cid, \perp, \mathrm{EXPIRE})$, all objects in the identified cluster C are marked as unclassified and inserted into U.

Given an event $(t, o, cid_1, cid_2, \mathrm{MERGE})$, the two clusters identified by cid_1 and cid_2 are merged. The cid field of the members of the new cluster is updated.

Given an event $(t, o, cid, \perp, \mathrm{SPLIT})$, the cluster identified by cid splits into two clusters. Depending on their specific connectedness to core members, the members in C have their cid field updated. In Algorithm 10, we provide pseudo-code for APPEAR, EXIT and JOIN events.

The Insert procedure used at the end of Algorithm 10 inserts an unclassified object into an existing cluster if the object moves into the cluster, and detects any new clusters in U that contains the set of unclassified objects. The detailed pseudo-code of the Insert procedure is shown in Algorithm 11. It first initializes a variable G that records already classified objects that are in the e-neighborhood of any object in U. Variable G is populated by calling a RangeQuery procedure for each object o in U, that returns all objects in O that are in the e-neighborhood of o (lines 2–6).

Algorithm 11: Insert($U, O, e, m, \tau, \delta, H$)

```
$G \leftarrow \emptyset$;
foreach $o \in U$ do
    $L \leftarrow$ RangeQuery($o, e + 2\delta, O$);
    foreach $o' \in L$ do
        if $o'.cid > 0$ then
            $G$.add($o'$);
foreach $o \in G$ do
    $C \leftarrow$ getMemByCid($o.cid$);
    $L \leftarrow$ RangeQuery($o, e + 2\delta, O$);
    if $|L| \geq m$ then
        $L$.delete($o$); $o.label \leftarrow$ CORE;
        ExpandCluster ($O, o, C, L, e + 2\delta, m, U, H$);
if $|U| \geq m$ then
    $CS \leftarrow$ DBScan($U, e + 2\delta, m$);
    foreach $C \in CS$ do
        foreach $o \in C$ do
            if $o.label$ = BORDER then
                $t \leftarrow$ getExitTime($o, C$);
                insert ($t, o.oid, C.cid$, EXIT) into $eventQ$;
update $H$;
```

In lines 7–11, if the RangeQuery result for an object $o \in G$ contains at least m objects, o's label is updated to CORE, and every unclassified object in o's neighborhood is removed from U and inserted into the cluster that o belongs to. The procedure ExpandCluster (line 12) recursively expands a cluster by exploring the new core object. After every object in G is considered, if U still has at least m objects, the procedure clusters the remaining objects in U (lines 13–19). It uses a variable CS that contains a set of clusters. First, it clusters all the moving objects as of the current time. Second, it calculates the exit time for each border object. These objects and their time to exit are inserted into $eventQ$ as events.

The ExpandCluster procedure expands a cluster by recursively exploring the new core object of the cluster. The detailed pseudo-code is shown in Algorithm 12. In the parameters, $crObj$ is a core object of cluster C and L is $crObj$'s current neighbors. The global variable $crTime$ records the current time. For each $o' \in L$, if o' is unclassified, a JOIN event is detected (line 3). The global variable $crTime$ records current time. Then the procedure retrieves o' neighbors in line 4. If o' has at least m neighbors, o' is also a core object, and thus ExpandCluster is called recursively (lines 5–7). If o' has fewer than m neighbors (it

Algorithm 12: ExpandCluster(O, $crObj, C, L, e, m, U, H$)

```
1  foreach $o' \in L$ do
2      if $o'.cid = 0$ then
3          insert $(crTime, o', C.cid, \mathrm{JOIN})$ into $eventQ$;
4          $L' \leftarrow$ RangeQuery($o'$, $e$, $O$);
5          if $|L'| \geq m$ then
6              $o'.label \leftarrow \mathrm{CORE}$;
7              ExpandCluster($O$, $crObj, C, L, e, m, U, H$);
8          else
9              $o'.label \leftarrow \mathrm{BORDER}$;
10             $t \leftarrow \mathrm{getExitTime}(o', C)$;
11             insert $(t, o', C.cid, \mathrm{EXIT})$ into $eventQ$;
12     else if $o'.cid \neq crObj.cid \wedge o'.label = \mathrm{CORE}$ then
13         insert $(crTime, o', C.cid, \mathrm{MERGE})$ into $eventQ$;
14 update $H$;
```

is a border object), the procedure calculates the exit time of o'. An EXIT event is created for o' and inserted into $eventQ$ (lines 8–11).

When o' is a core object from another cluster, cluster C is merged with the cluster identified by $o'.cid$. A MERGE event is created and inserted into $eventQ$ (line 13).

4.3.1.3 Detecting Cluster Expiry and Split Events

The exit of a single object can cause a cluster to *expire*. We show how to detect when cluster expiry occurs, so that repeated checking is avoided.

In DBScan, an object o is *directly density-reachable* from an object o' if o is within o''s e-neighborhood, and o' is surrounded by sufficiently many objects. Next, o is called *density-reachable* from o' if there is a sequence of objects $o_1, o_2, \cdots, o_n$ with $o_1 = o'$ and $o_n = o$ such that o_{i+1} is directly density-reachable from o_i. Then an object o is *density-connected* to an object o' with respect to e and m if there is an object o'' such that both o and o' are density-reachable from o'' with respect to e and m.

Since a group is required by definition to have at least m objects, and at least one object in the group is a core object by the definition of density-connectedness, core objects should have at least $(m-1)$ neighbors within their e-neighborhood, whereas border objects have fewer neighbor objects.

The removal of a border object o from a cluster C causes C to expire if only $(m-1)$ members remain. The removal of a core object o from C is more complicated. The neighbors of o may no longer be connected after the removal of o. Thus, we have to check (1) the possibility to remove o's neighbors and (2) the cardinality of C to determine whether or not C expires.

Two core objects may travel apart from each other, causing the cluster to split. The expected time of such each can also be calculated in a similar way.

4.3.1.4 Object Exit Time and Join

Once a cluster has formed, the getExitTime procedure computes the exit time for each border object in the cluster. It starts by identifying the boarder objects and the core objects. Next, it models the movement of an object o in $\mathbb{R}^2$ as a linear function of time: $o = (x^{t_{ref}}, y^{t_{ref}}, v_x, v_y)$, where $(x^{t_{ref}}, y^{t_{ref}})$ is the position of o at reference time t_{ref} and (v_x, v_y) is its latest reported velocity. Then it computes the exit time for a boarder object by calculating the time when the distance between boarder object and its core object is e [7].

After a boarder object exits its cluster, its core object has one less neighbor. When the core object has less than $m-1$ neighbors, this core object becomes a boarder object.

We follow a passive strategy for detecting join events, which is carried out by the Insert procedure at the end of each iteration.

4.3.1.5 Distance Bounds

As explained in Section 2.3, trajectory simplification can improve the efficiency of many algorithms that operate on the trajectories by removing relatively unimportant data points.

With trajectory simplification, in order not to miss clusters, a relationship has to be established between the simplified and original trajectories. In the Insert procedures using RangeQuery, the following lemma is applied.

Lemma 4.3.1 *Let tr_i, tr_j be the trajectories of o_i and let o_j, and let tr'_i, tr'_j be their trajectories after trajectory simplification with the sliding window algorithm explained in trajectory simplification in Section 2.3. Let $tr_i(t)$ be the position of o_i at time t. Then $D(tr'_i(t), tr'_j(t)) > 2\delta + e \Rightarrow D(tr_i(t), tr_j(t)) > e$.*

Proof 4.3.1 *To prove the lemma by contradiction, assume the inequality $D(tr_i(t), tr_j(t)) \leq e$ holds. After simplification, $tr'_i(t)$ is either the position of a sampling point or a position on a line segment. By definition of the sliding window algorithm, in the first case, $tr'_i(t) = tr_i(t)$, and in the second case, $D(tr'_i(t), tr_i(t)) \leq \delta$. In both cases, $D(tr'_i(t), tr_i(t)) \leq \delta$. Similarly, we have $D(tr'_j(t), tr_j(t)) \leq \delta$. According to the triangle inequality, we have $D(tr'_i(t), tr'_j(t)) \leq D(tr'_i(t), tr_i(t)) + D(tr'_j(t), tr_j(t)) + D(tr_i(t), tr_j(t)) \leq 2\delta + e$. This contradicts the assumption that $D(tr'_i(t), tr'_j(t)) > 2\delta + e$.*

According to Lemma 4.3.1, the range search distance $2\delta+e$ used in the GroupDiscovery framework is safe.

4.3.2 A Running Example

Figure 4.3 shows the trajectories of six moving objects, $o_1, \ldots, o_6$ being sampled at every time point from time t_0 to time t_9. It is assumed that the objects move beyond the 10 time units shown. We use $o_1, \ldots, o_6$ as the identifiers of the objects. APPEAR events are solid squares, UPDATE events are solid dots, and DISAPPEAR events are crosses. In contrast, hollow and square points mean that there are no primary events. For instance, object o_4 has an APPEAR event, an UPDATE event, and a DISAPPEAR event.

Let the group query parameters m (cardinality) and τ (duration) each be 3. Parameter e (clustering distance threshold) is as shown in the figure, and parameter k is not needed. FindContinuousCluster runs as follows.

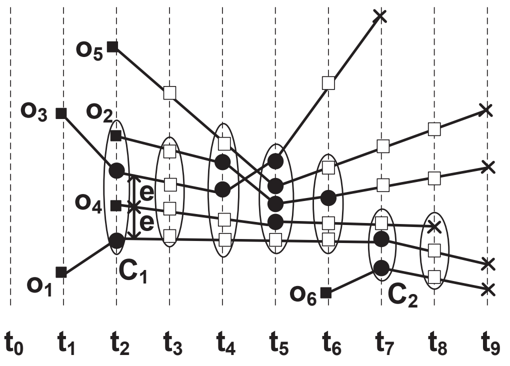


Figure 4.3: Trajectories of Six Moving Objects

As the trajectory data stream in, at t_1, events $(t_1, o_1, \perp, \perp, \text{APPEAR})$ and $(t_1, o_3, \perp, \perp, \text{APPEAR})$ are handled, resulting in $U = \{o_1, o_3\}$ (unclassified objects).

At time t_2, 5 events are handled. Objects o_2, o_4, and o_5 appear and are added to set U, and objects o_1 and o_3 update their velocities. Then Insert finds a cluster $C_1 = \{o_1, o_2, o_3, o_4\}$ where o_3 and o_4 are core objects, and o_1 and o_2 are border objects. Suppose both o_1 and o_2 are calculated to exit at t_{10}. Then exit events for border objects are inserted into $eventQ$: $(t_{10}, o_1, C_1, \perp, \text{EXIT})$ and $(t_{10}, o_2, C_1, \perp, \text{EXIT})$. After this iteration, $U = \{o_5\}$.

No events occur at time t_3. At t_4, o_2 and o_3 update their velocities. A join event for o_5 is detected and handled. The result is that $C_1 = \{o_1, o_2, o_3, o_4, o_5\}$, where o_2 is promoted to a core object. Thus, the exit event for o_2 in $eventQ$ is deleted. Finally, an exit event is created for o_5, $(t_7, o_5, C_1, \perp, \text{EXIT})$, as o_5 is a border object.

At t_5, objects o_2, o_3, o_4, and o_5 update their velocities. It is detected that o_3 becomes a border object and that o_5 becomes a core object. The exit event of o_5 in $eventQ$ is deleted. A new exit event $(t_6, o_3, C_1, \perp, \text{EXIT})$ is created and inserted into $eventQ$.

At t_6, o_6 appears and is added to U. Core object o_2 updates its velocity, causing border objects to update their expected exit time and to update the $eventQ$. Object o_3 exits C_1, and o_5 becomes a border object. An exit event is created for o_5: $(t_7, o_5, C_1, \perp, \text{EXIT})$. Now $U = \{o_3, o_6\}$ and $C_1 = \{o_1, o_2, o_4, o_5\}$.

At time t_7, o_3 disappears and o_1 updates its velocity. After the exit of o_5, o_2 downgrades to a border object, triggering the expiry of C_1 because no core objects exist. An expire event $(t_7, o_5, C_1, \perp, \text{EXPIRE})$ is created and inserted into $eventQ$. It is then immediately handled yielding $U = \{o_1, o_2, o_4, o_5, o_6\}$. Insert then finds a new cluster $C_2 = \{o_1, o_4, o_6\}$. New exit events are created for border objects o_4 and o_6. Now $U = \{o_2, o_5\}$

At time t_8, o_4 disappears, resulting in the expiry of C_2 because $|C_2| < m$. Now $U = \{o_1, o_2, o_5, o_6\}$.

4.3.3 History Handler Module

The GroupDiscovery framework needs to deal with potentially large numbers of groups. In our first approach, the history handler module manages the groups using a trie. However, we observe that not all subgroups need to be materialized. Based on this, the history handler module can employ a hash structure that maps clusterIDs to lists of cluster statuses (called *cluster histories*). We call the first method *GroupDiscovery (GD)* and the improved method *GroupDiscoveryPlus(GD+)*.

4.3.3.1 Group Discovery

In this approach, the history handler module employs a modified trie to represent subgroups during group discovery. A trie, or prefix tree, is an ordered tree data structure where each vertex represents a string (a word or a prefix). The descendants of a vertex have the string associated with that node as a common prefix; the root is associated with the empty string.

We represent a subgroup as the string that contains the identifiers of its member objects in sorted order. Each path in the trie structure represents a subgroup, and each leaf contains the subgroup members, and the subgroup's start and end time for computing the subgroup's score.

In our running example in Section 4.3.2, cluster $C_1 = \{o_1, o_2, o_3, o_4\}$ is formed at time t_2. Its subgroups are shown in Figure 4.4 with $m = 3$. The starting time of all subgroups is t_2. The ending time (t_e) is undefined initially, but is set to the time when a cluster expires.

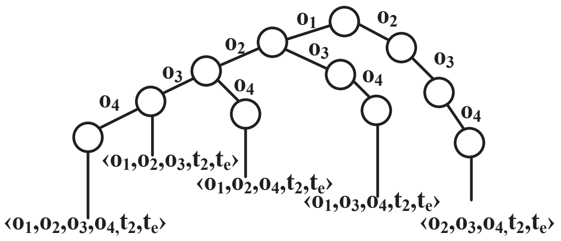


Figure 4.4: Trie for Example Cluster C_1 at Time t_2

When an object joins a cluster, new subgroups containing the object are created, and new paths are inserted into the trie. In our running example, object o_5 joins C_1 at t_4. The trie of the resulting cluster is presented in Figure 4.5. The starting time of each new path with object o_5 is set to t_4.

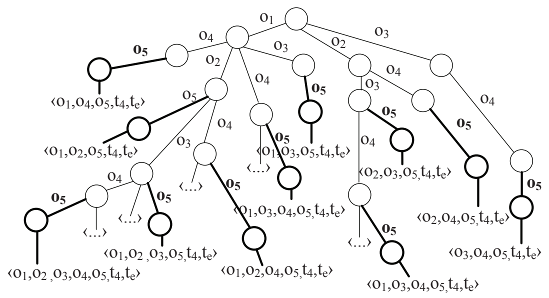
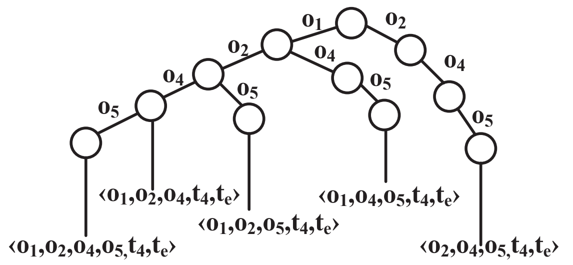


Figure 4.5: Trie After Insertion of o_5

When a border object exits a cluster, every path in the trie representing a subgroup that contains this object should be removed. Before the removal, the algorithm follows these paths, retrieve the leaf nodes, and sets the end times (t_e) in these leaf nodes to be current time. These leaf nodes represent subgroups that are then scored and returned. In our running example, when o_3 (a border object) exits C_1 at time t_6, the resulting trie structure is presented in Figure 4.6(a).

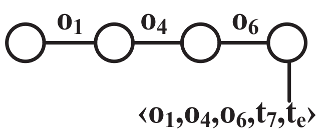
If an exiting object is a core object, its exit can cause other objects to also exit the cluster. Then every path containing any exit object is removed from the trie. Suppose object o_2 is a core object of a cluster $C_1 = \{o_1, o_2, o_4, o_5, o_6\}$ which starts from time t_7. Its exiting causes the exit of o_5. The resulting cluster is then $\{o_1, o_4, o_6\}$. The corresponding trie is shown in Figure 4.6(b).

When two clusters are merged, it is straightforward to update the trie with new paths. No paths are removed because objects in existing subgroups still travel together. However, updating the trie when a cluster expires is complicated because some objects from the old cluster may form a new cluster after re-clustering.



o_1
o_2
o_2
o_4
o_4
o_5
o_4
o_5
o_5
o_5
$\langle o_1,o_2,o_4,t_4,t_e\rangle$
$\langle o_1,o_4,o_5,t_4,t_e\rangle$
$\langle o_1,o_2,o_5,t_4,t_e\rangle$
$\langle o_1,o_2,o_4,o_5,t_4,t_e\rangle$
$\langle o_2,o_4,o_5,t_4,t_e\rangle$

(a) After o_3 Exits



(b) After o_2 and o_5 Exit

Figure 4.6: Tries after Removals

Suppose $C_1 = \{o_1, o_2, o_3, o_4\}$ expires and that $C_2 = \{o_1, o_3, o_4, o_7\}$ and $C_3 = \{o_2, o_5, o_6\}$ result from re-clustering. Although o_2 is in a new cluster, o_1, o_3, and o_4 still travel together. This fact can be found by intersecting C_1 and C_2. In general, when a cluster expires, all its subgroups are enumerated. It is then checked for each subgroup whether its members belong to the same cluster after re-clustering. If not, the path representing the subgroup is removed. After removing a path, its leaf node is then checked by the procedure CheckCandidate (Algorithm 14).

4.3.3.2 Group Discovery Plus

In the naive approach, many subgroups are materialized and inserted into the trie. Although search for subgroups is efficient, there are still an exponential number of subgroups. Thus, the trie becomes a bottleneck for large clusters.

The following simple, yet important, observation contributes to solving the exponential subgroup problem.

Lemma 4.3.2 *A subgroup that forms no earlier and expires no later than does a superset group is dominated by that superset group.*

Proof 4.3.2 *Let C_2 be a subgroup of C_1 with a lifetime contained in that of C_1. Then $|C_1| \geq |C_2|$, $C_1.t_s \leq C_2.t_s$ and $C_2.t_e \leq C_1.t_e$. Thus, C_1 dominates C_2.*

Thus, only subgroups that are formed earlier or expire later than their superset clusters need to be considered. The GroupDiscoveryPlus (GD+) algorithm only maintains such subgroups.

GD+ uses a hash table H that maps cluster identifiers to the history statuses of the clusters. A history status is similar to a leaf in the trie and contains the object identifiers of the members in the cluster together with a time interval. Update occurs only to the most recent information, called the *active status*. Once an event happens the active status is updated, and the score is computed. Then a new active status is appended to the list. In case a cluster expires, GD+ goes through the history of the cluster and generates each possible candidate.

In our running example, we have a cluster C_1= $\{o_1, o_2, o_3, o_4\}$ at t_2, and m=3. GD+ inserts $\langle C_1, \langle \{o_1, o_2, o_3, o_4\}, t_2, nil \rangle\rangle$ into H. At t_4, o_5 joins C_1, which becomes $C_1 = \{o_1, o_2, o_3, o_4, o_5\}$. GD+ then sets the end time in the active entry to t_3 and appends a new entry $\langle \{o_1, o_2, o_3, o_4, o_5\}, t_4, nil \rangle$ to the history for C_1 in H.

At t_6, o_3 leaves C_1. At t_7, the exits of o_2 and o_5 cause C_1 to expire. Then $C_2 = \{o_1, o_4, o_6\}$ forms. At t_9, cluster C_2 also expires. At this time, the entries in H are $\{\langle C_1, \langle \{o_1, o_2, o_3, o_4\}, t_2, t_3 \rangle, \langle \{o_1, o_2, o_3, o_4, o_5\}, t_4, t_5 \rangle, \langle \{o_1, o_2, o_4, o_5\}, t_6, t_6 \rangle\rangle,$
$\langle C_2, \langle \{o_1, o_4, o_6\}, t_7, t_8 \rangle\rangle\}$

GD+ accesses the entry for each cid in reverse order and finds (1) that o_1, o_2, and o_4 travel together from t_2 to t_6, (1) that o_1, o_2, o_3, and o_4 travel together from t_2 to t_5, (2) that o_1, o_2, o_3, o_4, and o_5 travel together from t_4 to t_5, (3) that o_1, o_2, o_4, and o_5 travel together from t_4 to t_6, and (4) that o_1, o_4, and o_6 travel together from t_7 to t_8. So the final entries in H are $\{\langle C_1, \langle \{o_1, o_2, o_3, o_4\}, t_2, t_5 \rangle, \langle \{o_1, o_2, o_3, o_4, o_5\}, t_4, t_5 \rangle,$
$\langle \{o_1, o_2, o_4, o_5\}, t_4, t_6 \rangle, \langle \{o_1, o_2, o_4\}, t_2, t_6 \rangle\rangle$ $\langle C_2, \langle \{o_1, o_4, o_6\}, t_7, t_8 \rangle\rangle\}$.

The RevHist algorithm that manages the history statuses of clusters is shown in Algorithm 13. Recall that the history statuses of a cluster is a list of items with members and their starting and ending times.

Algorithm 13: RevHist(H)

```
1  for $i \leftarrow H$.size() $- 1; i \geq 0; i--$ do
2      $curr \leftarrow H$.get($i$);
3      for $j \leftarrow i-1; j \geq 0; j--$ do
4          $prev \leftarrow H.get(j)$;
5          if $curr.t_s = prev.t_e + 1$ then
6              $mem \leftarrow curr \cap prev$;
7              if $mem = curr$ then $curr.t_s \leftarrow prev.t_s$;
8              else if $mem = prev$ then $prev.t_e \leftarrow curr.t_e$;
9              else if $mem$.size() $\geq m$ then
10                 create a new entry $e$ with members $mem$;
11                 $e.t_s \leftarrow prev.t_s$;
12                 $e.t_s \leftarrow curr.t_e$;
13                 $curr \leftarrow e$;
14                 $H$.add($e$);
```

The procedure starts from the last item ($curr$) of the list, and compares the previous items ($prev$) in reverse order. If $curr$ is a subset of $prev$, the current subgroup actually travels together from the previous item's start time. If $curr$ is a superset of $prev$, the previous subgroup travels together until the current item's end time. Lastly, if the intersection between $curr$ and $prev$ has at least m objects, a new entry is created with the previous item's start time and the current item's end time. This new entry is then inserted into the list.

4.3.4 Returning Meaningful Results

The GD and GD+ algorithms may return groups that are similar to each other. Similar groups brings little new knowledge and promises limited insight into the data. In order to improve the quality of results, we propose a similarity measure for groups.

Definition 9 *We define the similarity of two groups C_1 and C_2 as*

$$\mathrm{Sim}(C_1, C_2) = \frac{\|C_1 \cap C_2\|}{\|C_1 \cup C_2\|}$$

We then require that result groups are more diverse than a given threshold θ. The following statement should then be true. $\forall C_1, C_2 \in R(C_1 \neq C_2 \Rightarrow (\mathrm{Sim}(C_1, C_2) \leq \theta))$. The similarity checking is carried out by the algorithm 14.

The CheckCandidate procedure takes a candidate group collection S, the top-k set R, a threshold θ, and k as parameters. The groups with top-k scores are to be inserted into R, and $R.minScore = \min_{C \in R} C.score$, i.e., the smallest top-k score. The procedure iterates over every candidate group C, and validates that the duration of C exceeds τ. The variable *dominated* captures whether C is dominated by some entry in R. The variable *updated* captures whether R has been updated. The variable C_{sim} refers to a similar entry in R, if any, and is set to null initially. For each entry C' in the candidate list R, if C' dominates C, then dominated is set to true. C cannot be in the result and the program terminates. Otherwise, if C dominates C', C' is removed from R, as C' cannot be in the result. If C' and C are similar, then C_{sim} refers to C'. If C is neither dominated by nor similar to any entry, C is inserted into R, and updated is set to true. Otherwise, the algorithm includes into R the group with the higher score between C and C_{sim}, and it excludes the other group. Updated is set to true if R is updated. In the end, if R is updated and its size exceeds k, the algorithm picks the top-k groups in R. $R.minScore$ is updated accordingly.

4.3.5 Avoiding RevHist Calls

Recall that the group query returns k results that score the highest and that a cluster generates many groups or subgroups during its entire life. We can save actual-score computations (invocations of the RevHist procedure) by computing instead upper bounds on the scores of groups. If the upper bound score for a cluster is smaller than the lowest actual score of the k'th candidate group, the groups generated by the cluster cannot be in the result.

Algorithm 14: CheckCandidate(S, R, θ, k)

```
1  foreach $C \in S$ do
2      if $C.t_e - C.t_s \geq \tau$ then
3          dominated $\leftarrow$ false;
4          updated $\leftarrow$ false; $C_{sim} \leftarrow$ null;
5          foreach $C' \in R$ do
6              if $C' \prec C$ then dominated $\leftarrow$ true; break;
7              else if $C \prec C'$ then $R$.delete($C'$);
8              if Sim($C', C$) $> \theta$ then $C_{sim} \leftarrow C'$ ;
9          if $\neg$ dominated then
10             if $C_{sim}$ = null then
11                 $R$.add($C$);
12                 updated $\leftarrow$ true;
13             else if $Score(C) > Score(C_{sim})$ then
14                 $R$.delete($C_{sim}$);
15                 $R$.add($C$);
16                 updated $\leftarrow$ true;
17         if updated then
18             if $R.size() > k$ then
19                 $R \leftarrow R.topK()$;
20             update $R.minScore$;
```

Here, we develop an upper bound score of a cluster. Suppose that the candidate list initially maintains k candidates. When a cluster C expires, we compare the upper bound of C with $R.minScore$. The RevHist procedure is invoked only if the upper bound of C exceeds $R.minScore$.

The cardinality of a cluster C can change over time. We calculate the upper bound score of C by using C's maximal cardinality.

Definition 10 *Let N_{max} denote the maximal cardinality of C during its lifetime, the duration of which is ρ_C. The upper bound score of C, $\text{Score}_{UB}(C)$, is defined as:*

$$\text{Score}_{UB}(C) = \alpha N_{max} + (1 - \alpha)\rho_C$$

The upper bound score of C is never smaller than the actual score of any group of C.

Lemma 4.3.3 *Let $\text{Score}_{UB}(C)$ be as defined as above and let C^i be a group generated by*

C. We have:

$$\text{Score}(C^i) \leq \text{Score}_{UB}(C)$$

Proof 4.3.3 *Because C^i is a group of C, we have $|C^i| \leq N_{max}$. The lifetime of C^i also cannot exceed the lifetime of C, so $\rho_{C^i} \leq \rho_C$. Thus,* $\text{Score}(C^i) = \alpha N_{C^i} + (1-\alpha)\rho_{C^i} \leq \alpha N_{max} + (1-\alpha)\rho_C = \text{Score}_{UB}(C)$.

4.3.6 Complexity Analysis

In this section, we characterize the running time of the GD and GD+ algorithms.

Lemma 4.3.4 *Let TR be a trajectory collection of N trajectories, let Q be the total number of primary events in TR, and let M be the total number of events that change the status of any cluster, i.e., exit, join, expire, and merge events. The running time of FindContinuousCluster is $O(Q + MNlgN)$.*

Proof 4.3.4 *We use a Fibonacci-heap to implement $eventQ$, where an insertion takes $O(1)$ time. We also assume that RangeQuery has time complexity $O(lgN)$. FindContinuousCluster first inserts primary events into $eventQ$, taking $O(Q)$ time. For each event that changes the status of clusters, FindContinuousCluster calls the Insert procedure. In Insert, U and G are mutually exclusive sets that sum up to N. Each object in either U or G calls RangeQuery at most once, so the two loops in Insert take $O(NlgN)$ time. Lastly, the Insert procedure calls DBScan, which has complexity $O(NlgN)$. So the Insert procedure has time complexity $O(NlgN)$. The total complexity of calling Insert for M events is $O(MNlgN)$. The time complexity of FindContinuousCluster is then $O(Q + MNlgN)$.*

Next, we consider the time complexity of handling subgroups.

Lemma 4.3.5 *Given P clusters, the time complexity for the History Handler module in GD is $O(M\sum_{i=1}^{P} 2^{N_i})$ where $\sum_{i=1}^{P} N_i = N$ and $0 \leq P \leq \frac{N}{m}$.*

Proof 4.3.5 *In the History Handler module, every operation needed when the statuses of clusters are updated is performed on the trie structure. If the objects do not form clusters, there is no need to update the trie, so the worst case is that every moving object is a member of a cluster. Let the N moving objects form P clusters ($0 \leq P \leq \frac{N}{m}$) at each event time. The worst case is that these clusters have to be completely re-built at each event time. Let the cardinalities of these clusters be $N_1, N_2, \cdots, N_P$ ($\sum_{i=1}^{P} N_i = N$). For a cluster C_i, the number of combinations is:*

$$\binom{N_i}{m} + \binom{N_i}{m+1} + \cdots + \binom{N_i}{N_i} \leq 2^{N_i}$$

where m is the cardinality threshold. Then the total number of combinations for all clusters is $\sum_{i=1}^{P} 2^{N_i}$. Again, let M be the number of events that change the status of clusters, the complexity is $O(M \sum_{i=1}^{P} 2^{N_i})$.

Lemma 4.3.6 *Given P clusters, the time complexity for the History Handler module in GD+ is $O(PM^2)$.*

Proof 4.3.6 *We consider again the worst case where every object is in a cluster and let the N moving objects form P clusters at each event time. The worst case is that for each event time, every cluster has to create a new snapshot at the end of its history list. In total, each cluster has to create M snapshots. Maintaining the data structures of these snapshots takes linear time. The input size for RevHist is P clusters, each with a list of snapshots of size M. For each cluster, every snapshot may have to intersect with the snapshots in front of it, which takes time $O(M^2)$ in the worst case. Therefore, the complexity of going through every cluster is $O(PM^2)$. So the complexity of the History Handler module in GD+ is $O(PM + PM^2)$ = $O(PM^2)$.*

Theorem 4.3.1 *The time complexity for GD is $O(\sum_{i=1}^{N} n_i^2 + Q + MNlgN + M \sum_{i=1}^{P} 2^{N_i})$, whereas the time complexity for GD+ is $O(\sum_{i=1}^{N} n_i^2 + Q + MNlgN + PM^2)$, where n_i is the number of data points in the ith trajectory.*

Proof 4.3.7 *The time complexity of the framework is the total of NOPW and its two functional modules. The time complexity of NOPW is $O(\sum_{i=1}^{N} n_i^2)$. From the lemmas above, Continuous Clustering in either GD or GD+ has complexity $O(Q + MNlgN)$. In GD and GD+, the History Handler Module has complexity $O(M \sum_{i=1}^{P} 2^{N_i})$ and $O(PM^2)$, respectively. Therefore, GD has complexity $O(\sum_{i=1}^{N} n_i^2 + Q + MNlgN + M \sum_{i=1}^{P} 2^{N_i})$. GD+ has complexity $O(\sum_{i=1}^{N} n_i^2 + Q + MNlgN + PM^2)$.*

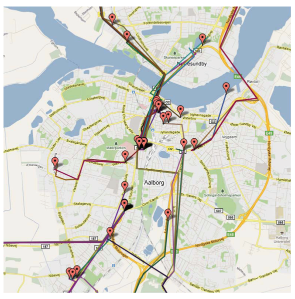
4.4 Experiments

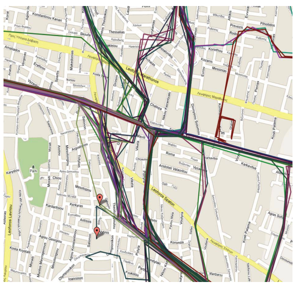
The experiments were performed on a PC with Intel Xeon (2.66Ghz) quad-core and 8 GB of main memory running Linux (kernel version 2.6.32). Every instantiation of JVM allocates 2 GB of virtual memory. All the algorithms were implemented in Java, including the Convoy algorithm [47].

4.4.1 Data Sets and Parameter Settings

To evaluate the performance of the proposed framework, we use three real trajectory data sets obtained from vehicles and animals, and one synthetic data set generated by a free space trajectory data generator, the GSTD generator [77].

Figure 4.7 shows a subset of trajectories from these data sets.


(a) Car


(b) Truck

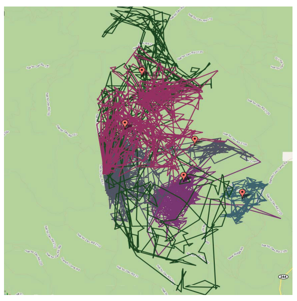

(c) Starkey

Figure 4.7: Visualization of Data Sets

Car This data set stems from the INFATI project [42], and has around 1.8 million data points obtained from 20 different vehicles collected during a 4-month period. A new trajectory is created each time the duration between two consecutive data points from the same vehicle exceeds 10 minutes. Also, to increase the probability of finding more objects that travel together, we omit the dates from timestamps and consider only the time of day. After pre-processing, 2,305 separate trajectories were obtained.

Truck This data set contains 111,930 data points, representing 267 trajectories from 50 trucks moving in the Athens, Greece region[2]. As in Car, the dates are removed. Each trip of a truck is represented as one trajectory.

Starkey This data set contains 252 trajectories obtained from the Starkey Project[3], a radio-telemetry study of elk habitats (elk, mule deer, and cattle) in Northeastern Oregon from 1993 to 1996. The collection has 268,216 data points. Each trajectory represents the movement of a particular animal. In this data set, dates were taken into account because each animal was tracked for a long period of time. Also, since the sampling varies substantially across different animals, it was difficult to consistently partition the trajectory of an animal into shorter trajectories.

GSTD We use the GSTD generator to generate data sets with varying numbers of trajectories. The data points are assumed to be taken from the same day.

We partition Car, Truck, and GSTD into 144 parts, each containing data for 10 min, and we partition Starkey into 146 parts, each containing data for 10 days. In the following experiments, after GD and GD+ process one part, the next part is streamed in as new data, resulting in new events being inserted into $eventQ$. Statistics on the data sets and default settings for key parameters are shown in Table 4.3.

We report on both efficiency and effectiveness studies of our proposed framework, and we compare with the Convoy algorithm [47]. In the studies of the effects of polyline simplification, we use the existing online simplification technique, Normal Opening Window (NOPW) [64]. For Car and Truck, the reduction rate is high even for small tolerances. Table 4.4 shows the number of remaining data points when the tolerance δ ranges from 25

[2]http://www.rtreeportal.org

[3]http://www.fs.fed.us/pnw/starkey/data/tables/

Parameter	**Car**	**Truck**	**Starkey**	**GSTD**
N	2,305	267	252	5k – 25k
m	6	10	5	5
τ	3 min	3 min	5 days	10 min
e	400 m	10 m	1,000 m	30 m
δ	25 m	5 m	100 m	10 m
θ	0.5	0.5	0.5	0.5
α	0.5	0.5	0.5	0.5

Table 4.3: Settings for Experiments

to 200 in Car and Starkey and from 5 to 25 in Truck. The numbers of data points in GSTD (both original and after simplification) are summarized in Table 4.5.

Tolerance	**Car**	**Starkey**
0	1,747,757	268,216
25	109,907	214,277
50	98,460	179,059
100	95,476	135,211
150	95,239	107,885
200	95,202	90,523

Tolerance	**Truck**
0	111,930
5	76,557
10	64,677
15	57,157
20	51,881
25	47,766

Table 4.4: Simplification

Num Traj.	**Num Data Points**	**After Simpl.**
5k	210,130	186,265
10k	420,296	382,187
15k	630,268	697,271
20k	840,813	812,276
25k	1,050,876	972,138

Table 4.5: Synthetic Data Set

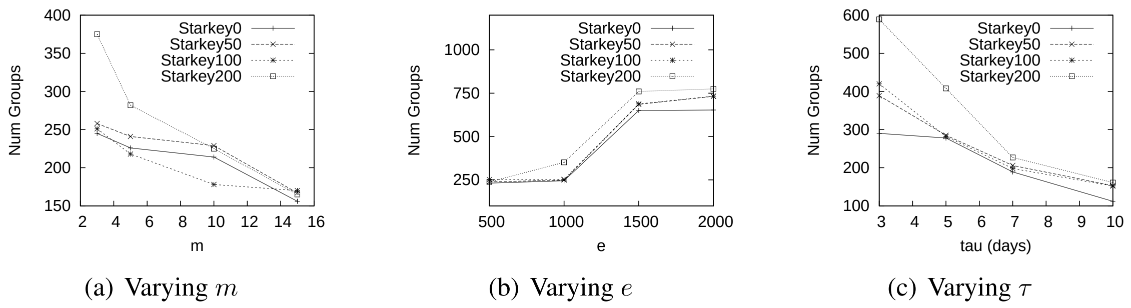
4.4.2 Effects of Varying m, e, and τ

We proceed to consider the number of groups returned when varying cardinality m, distance threshold e, and lifetime τ. We use Starkey with tolerance values being 0 (original), 50, 100, and 200 meters. Starkey0 thus serves as the ground truth. Figure 4.8 shows the result.

As the cardinality threshold m increases, the number of discovered groups decreases quickly with all simplification tolerances because more animals are needed to form clusters. For Starkey0 and Starkey50, very few groups are discovered when $m > 10$, whereas for

Starkey100 and Starkey200, a marked decrease is observed even before $m = 10$. It is also observed that Starkey50 displays the pattern that is most similar to Starkey0, whereas Starkey200 deviates the most from Starkey0. The four collections behave similarly when varying distance threshold e—as e increases, groups are allowed to be less dense, and more groups are found.

Finally, all collections exhibit a decreasing trend as parameter τ is increased.



(a) Varying m (b) Varying e (c) Varying τ

Figure 4.8: Effect of Varying m, e and τ on Groups Identified

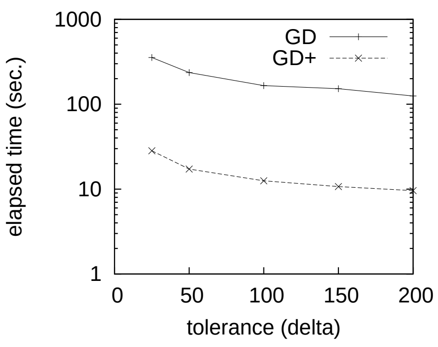
4.4.3 Comparing GD and GD+

To observe the practical implication of the observation in Lemma 4.3.2, which is exploited in GD+, we ran GD and GD+ on Car and counted the number of candidate groups. Figure 4.9(a) shows that the running time of GD is around 10 times that of GD+. Figure 4.9(b) shows that the number of checked candidates by GD+ is reduced by up to three orders of magnitude. Note that both figures have a log-scaled y-axis.

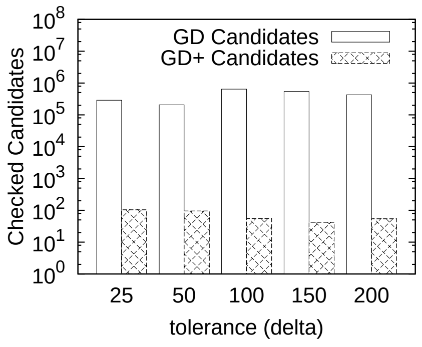
We were only able to run GD on Car as GD is rather inefficient because of frequent trie updates when the memberships of clusters changes. On the other data sets, GD exhausted the Java heap space. In conclusion, Lemma 4.3.2 greatly improves the performance of the GroupDiscovery framework.

4.4.4 Effect of Varying θ

Figure 4.10(a) (y-axis logscaled) shows the effect of θ on the number of pruned candidate groups. It is observed that with the increase of the θ value, fewer candidate groups are



(a) Running Time vs. Tolerance



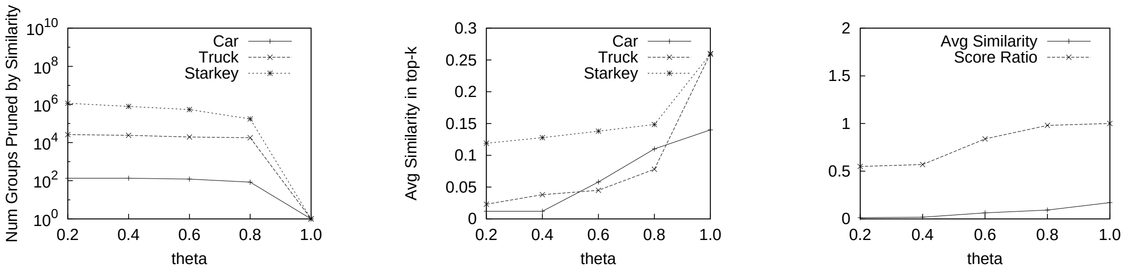
(b) Checked Candidates vs. Tolerance

Figure 4.9: Comparing GD and GD+

pruned. When θ is small, more candidate groups having common objects are treated as similar groups and are thus discarded by CheckCandidate. Among the three data sets, Car has the least pruned similar candidate groups, whereas Starkey has the most pruned similar candidate groups. Recall that for Car and Truck, we disregard the dates to increase the number of similar trajectories in order to find groups, while in Starkey, we take dates into account when tracking the same groups for three years. Thus, many more similar groups are found in Starkey due to recurring patterns. In Truck, the routes that the vehicles follow are more fixed than in Car, so more similar groups are found in Truck. Note that in both Truck and Starkey, many similar candidate groups are found along the way even when θ is large (0.8). Also note that no candidate groups are pruned when $\theta = 1.0$ for the three data sets.

Figure 4.10(b) shows that for each data set, the average similarity among the top-k groups actually increases with the increase of θ, though the difference between any two average similarity values is not significant in the same data set. It is also observed that the average similarity value depends highly on the data set.

There is a tradeoff between the total score of the groups in the top-k list and the average similarity. In order to have more diversity in the top-k list, we need to decrease the θ value, which may result in pruning some high-scoring candidates in the top-k list because they are similar to other group candidates in the top-k list. Figure 4.10(c) shows that with the increase of θ, the total score ratio (normalized by total value when $\theta = 1$) increases, and the average similarity among the top-k also increases, i.e., the diversity decreases.



(a) Pruned by Similarity (b) Average Similarity in top-k (c) Average Similarity vs. Total Score

Figure 4.10: Effect of Varying θ

4.4.5 Effect of Varying α

We conduct experiments on the effect of α on the average cardinality and duration of the results, which can be seen in Figure 4.11. The average duration of the result drops greatly when α increases from 0 to 0.2, and then decreases slowly. In contrast, the average cardinality of the result increases slowly from 5.53 to 6.28 with the increase of α.

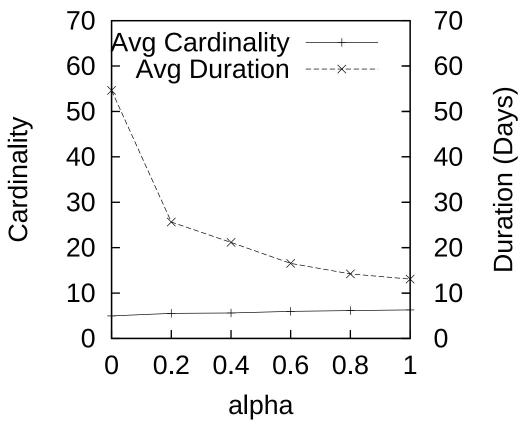
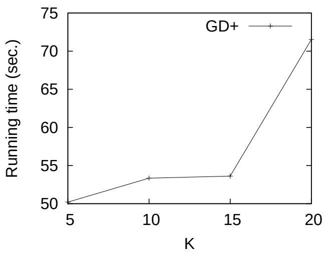


Figure 4.11: Average Cardinality and Duration vs. α

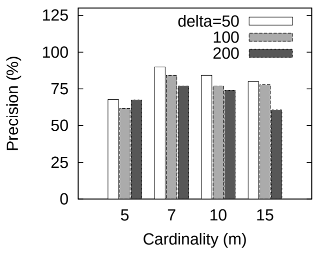
Recall the score definition in Section 4.2.1. Generally, small α values favor groups of longer durations, whereas large α values favor groups of larger sizes.

4.4.6 Effect of Varying k on Runtime

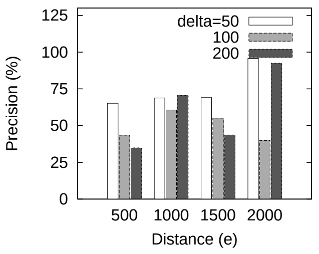
Figure 4.12(a) shows the effect on the runtime of GD+ of varying k when using Starkey. The figure shows that when k is smaller than 15, the running time is relatively low. But when k exceeds 15, the running time increases markedly. The reason is that many candidates have already been pruned based on the upper bound on group scores and $R.minScore$. Thus, the use of pruning based on the lower bound on group scores is effective, and GD+ is able to exploit small k values to obtain better performance.



(a) Effect of Varying k on Runtime



(b) Precision@10, Varying m



(c) Precision@10, Varying e

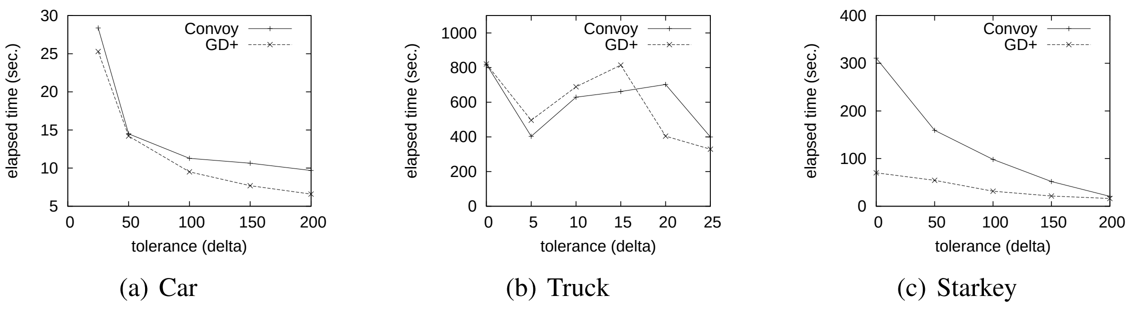
Figure 4.12: Top-k Results

4.4.7 Comparing Top-k Results

We next consider the effect of simplification on the top-k results produced on Starkey. We compare the results obtained from the original and from default simplified data sets. We apply the Precision@k ranking performance metric that measures the fraction of the number of groups that are found in both original and simplified data sets vs. total number of found groups (k). The metric Precision@k is widely used in information retrieval [63, 12].

We vary the cardinality threshold m from 5 to 15(Figure 4.12(b)) and the distance threshold e from 500 to 2,000 meters (Figure 4.12(c)), while using the simplification tolerance values 50, 100, and 200. When varying m, the precision ranges from 61% to 89%. The results are affected only little by simplification for $m = 7$ and $m = 10$. The highest drop in precision occurs for $m = 15$ and $\delta = 200$. Next, when varying e, the precision for different simplification tolerances ranges from 35% to 95%. The highest precisions are obtained when $\delta = 50$ and $e = 2,000$.

It is also observed that varying the distance threshold affects the results more than varying the cardinality threshold. The results from varying the distance threshold are also less stable.



(a) Car (b) Truck (c) Starkey

Figure 4.13: Effect of Simplification Tolerance on Efficiency

4.4.8 Comparing GD+ and Convoy

We proceed to consider the runtime performance of the GD+ algorithm when varying the polyline simplification δ, comparing with the Convoy algorithm [47]. To be fair, we run both algorithms in an offline setting, because Convoy can only be applied offline. The Douglas-Peucker (DP) polyline simplification algorithm [22] is used to simplify the trajectories for both algorithms. The results are shown in Figure 4.13.

As δ increases, fewer data points remain in the trajectories, which in turn improves the runtimes of both algorithms. The GroupDiscovery framework achieves speedups because fewer events need to be handled. The situation for the Convoy algorithm is more complex. This algorithm has a filter step that applies DBScan to line segments, and it has a refinement step that applies DBScan to data points. In both steps, with fewer sampling points, a line segment may have more time-overlapping line-segments for each clustering, but the total number of clustering performed is reduced.

On Car and Starkey, GD+ outperforms Convoy for all tolerances. However, on Truck, GD+ only runs faster than Convoy when the tolerance is high. When applied to Truck, GD+ finds much larger numbers of groups than does Convoy for small tolerance values, as shown in Figure 4.14. When the tolerance value is high, although GD+ still finds more groups, the filter step in Convoy takes more time because of larger numbers of candidates, so GD+ outperforms Convoy.

We also studied the effect of simplification on the number of discovered groups in both GD+ and Convoy. We first ran GD+ with $\delta = 0$ on each data set to obtain the total numbers of groups, which is treated as the *base-line*. Then we applied simplification with various δ values and ran both GD+ and Convoy on the simplified data sets. For each tolerance value, the groups obtained by the two algorithms are compared with each other and with the baseline. It is observed that the groups returned by Convoy were consistently subsets of those returned by GD+. Figure 4.14 shows the ratio of the number of groups found with different tolerance values with the baseline.

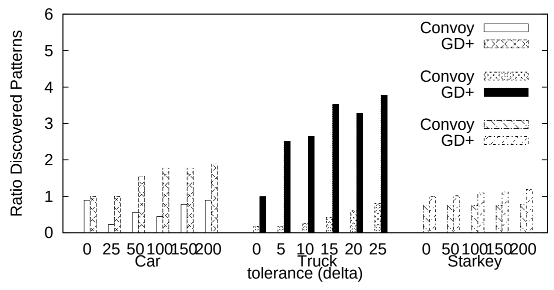


Figure 4.14: Effect of Simplification Tolerance on Error

It is seen that more groups are found than with Convoy. As discussed before, GD+ is able to find patterns that are missing by Convoy because of sampling independence. The reduction in the number of sample points due to simplification affects both Convoy and GD+. We observe that GD+ consistently finds increasing numbers of groups for all data sets as the tolerance is increased. The number of convoys found also increases with the tolerance for Truck and Starkey. The main reason is that the smoothing effect of trajectory simplification results in clusters of longer duration. However, the number of convoys in Car decreases for small tolerance values. This is mainly because with small tolerance values, the number of sampled points decreases significantly, which affects the Convoy algorithm very much. For both Car and Truck, the numbers of groups and convoys found are very sensitive to simplification because cars and trucks are constrained by a road network, and the smoothing effect of simplification is significant.

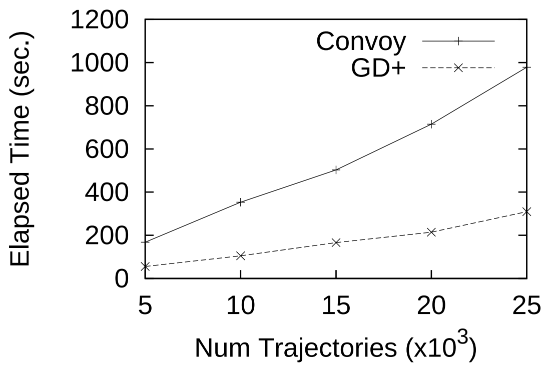


Figure 4.15: Effect of Number Trajectories

Next, we compare GD+ and Convoy on the synthetic data sets. Figure 4.15 shows the result, when the number of trajectories varies from 5k to 25k. It is observed that GD+ consistently outperforms Convoy. The performance gap between GD+ and Convoy increases with the number of trajectories.

The filtering step in Convoy relies on time-domain partitioning and segment clustering. With a larger number of trajectories, it is more likely to have many trajectories near each other during certain time periods. Thus, the filtering step in Convoy is less effective, resulting in more candidates to be checked by the time-consuming refining step. In contrary, thanks to the event-driven design, GD+ always processes events when necessary. So it is less affected by the increased number of trajectories.

4.5 Summary

In this chapter, we propose the concept of *trajectory group* that is different from related notions in previous works and that enables the prioritized discovery of interesting moving object clusters from trajectories. We also propose the first framework that satisfies the four requirements, sampling independence, online processing, density-connectedness, and supporting trajectory simplification. Techniques based on the continuous clustering of moving objects using an effective pruning strategy are proposed to efficiently discover such groups. A scoring function enables the ranking of the discovered groups according to their size and duratixon. The effectiveness and efficiency of our schemes are studied using real and synthetic data sets.

Chapter 5

Processing Optimal Segment Query

5.1 Introduction

In Chapter 3, we process the MCQ query that only focuses on real-time location data of moving objects. In Chapter 4, we solve the group discovery problem that take both real-time location data and trajectories. With the increasing availability of trajectory data, it is interesting and important to be able to use trajectories to support many real-life applications. In this chapter, we re-look at the classic *facility location* problem and explore a relatively new dimension, i.e., taking the movements of customers.

The problem of identifying new, attractive locations with respect to a given set of customer locations and existing facilities, known as the *facility location* problem, has been studied extensively [13, 23, 87, 90, 96, 97, 26, 69, 14]. This problem has applications in strategic planning of resources (e.g., hospitals, gas stations, banks, ATMs, billboards, and retail facilities) in both the public and private sectors [49, 48].

Earlier work has used the residences of consumers as the customer locations [87, 90, 97]. However, customers do not remain stationary at their residences, but rather travel, e.g., to work. Thus, consumers are not only attracted to facilities according to the proximity of these to their residences.

The increasing availability of moving-object trajectory data, e.g., as GPS traces, calls for an update to the facility location problem to also take into account the movements of the customers that are now available.

In particular, we study the *optimal segment* problem. Given a road network G, a set of facilities F, a set of route traversals R each of which can be taken by different users multiple times, the objective is to find the optimal road segments such that a new facility built on any of these segments maximizes the number of attracted route traversals. A route traversal is *attracted* by a facility if the distance between the route and the facility is within a given threshold.

Figure 5.1 shows an instance of the problem. Solid lines and dots form the road network. Hollow circles are existing facilities (f_1, f_2, f_3, and f_4). Dashed lines indicate route traversals (r_1, r_2, and r_3). We draw them next to the roads for clarity.

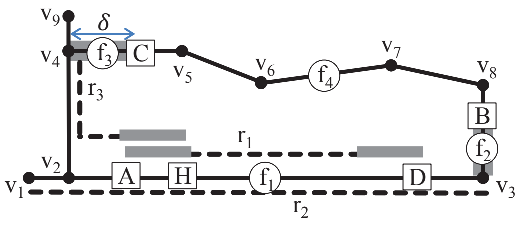


Figure 5.1: An Optimal Segment Problem Example

The gray bar that covers f_3 indicates that r_3 is attracted by f_3 because f_3 is within distance δ of one of the end points of r_3. The underlying rationale is that a facility that is sufficiently near a route will attract customers who follow the route. Therefore, the ends of each route are extended by distance δ.

With δ, r_1 starts and ends at A and D, respectively. Route r_2 starts and ends at v_1 and B, respectively. Route r_3 starts and ends at C and H, respectively. Suppose that each of the routes is traversed by one customer exactly once. Intuitively, the optimal segment for a new facility is the segment $\overline{AH}$, because building a new facility on this segment attracts the most route traversals (in this particular example, three traversals). Later, we will show that it is not only important to attract the most route traversals—not all route traversals are equally important.

The optimal segment problem is different from the traditional facility location problems [26, 69] and the optimal location problem [13, 23, 87, 90, 96, 97]. First, the optimal segment problem works with route traversals completed by mobile customers instead of static customer locations. This use of route traversals is a natural generalization of the use of static customer sites. This is so because, static customer sites can be derived from routes traversals. Often, the static customer sites are start and end points of routes. Also, in reality, a facility location close to many routes, e.g., near an intersection of several highways, but far from residential regions, may be quite attractive. Many shopping malls, fast food shops (e.g., McDonald's), and billboards target drivers. For example, billboards are often drivers' primary method of finding lodging, food, and fuel on unfamiliar highways. It is reported that there were approximately 450,000 billboards along United States highways in 2010 [74].

The second difference is that the optimal segment problem returns optimal road segments instead of optimal point locations. It is more appropriate to identify road segments than point locations because the former allows for more flexibility.

We propose a framework to solve the optimal segment problem. The main idea of the framework is to assign each route traversal a score and to distribute the score to the road network segments that are covered by the route traversal. The scoring of segments is based on three factors: the number of customers who take the route (the count), the number of traversals by each customer (the usage), and the length of the route. A scoring function satisfies the property that they are independent of the route partitioning.

Intuitively, road segments that are covered by many route traversals that are attracted by few existing facilities, are good result candidates. But customers of different types of businesses can have different spatial preferences with respect to the businesses they are likely to visit. For example, customers may visit stores that sell everyday items frequently, while they may visit stores that sell expensive items only infrequently. Likewise, customers may prefer grocery stores near their homes or work places, but may have equal probability to visit clothing stores along the routes they travel. To accommodate such preferences, we support different functions for the assignment of scores to the routes that customers follow as well as allow different models for the distribution of the score of a route to the underlying segments. Specifically, we require the framework to be generic with respect to the functions used for assigning scores to routes according to the traversals of the routes and with respect to the score distribution models that capture score distribution.

We propose two algorithms to solve the optimal segment problem. The first algorithm, AUG, uses a graph augmentation approach. The idea is to augment the original road network graph with the facilities and the start and end points of the route traversals as vertices. Each vertex in the new graph records a list of attracted routes. The score of an edge is the sum of scores of the route traversals that cover both vertices of an edge. The edges with the highest score are mapped back to the original road network graph and are possibly extended into longer road segments.

The second algorithm, ITE, uses a heap to prioritize the most promising road segments, and it iteratively partitions and scores these based on intersecting routes. ITE keeps partitioning the road segments that most likely contain an optimal subsegment until such an optimal subsegment is obtained. Then it extends the partial optimal segment to its full length and adds it to the result set.

Our contributions can be summarized as follows:

- Formalization of the new *optimal segment* problem that identifies the optimal road network segments for the placement of new facilities according to customer trajectory data.

- Development of a computational framework where route traversals are used instead of static customer sites for the optimal segment problem. This is the first such framework. Two algorithms, AUG and ITE, are proposed to solve the problem.

- The framework accommodates different scoring functions and score distribution models that may apply to a variety of businesses.

- Coverage of an empirical study that indicates that AUG and ITE are efficient in realistic settings.

The remainder of the chapter is structured as follows. Section 5.2 formalizes the problem setting. Section 5.3 presents a preprocessing procedure that is used by both of the two proposed algorithms. We describe in detail algorithm AUG and provide a theoretical analysis in Section 5.4. We then describe in detail algorithm ITE and give an accompanying theoretical analysis in Section 5.5. Section 5.7 reports the results of an empirical evaluation of the proposed algorithms. Finally, Section 5.8 summarizes this chapter.

5.2 Definitions

We proceed to model the road network and formulate the optimal segment problem along with supporting definitions.

5.2.1 Road Network Modeling

A road network is modeled as a *spatially embedded graph* $G = (V, E)$, where V is a set of vertices, and E is a set of edges that connect ordered pairs of vertices. Every vertex v_i has (x_i, y_i) coordinates in 2D space, denoted as $loc(v_i) = (x_i, y_i)$. We use either $e_{i,j}$ or (v_i, v_j) to refer to the directed edge from vertices v_i to v_j. The length of an edge is defined as the Euclidean distance between its two vertices: $\|e_{i,j}\| = \|loc(v_i), loc(v_j)\|$. Vertices and edges are assigned unique identifiers. In this model, an edge between two vertices represents a part of a road. The polyline obtained by connecting the vertices of consecutive edges approximates the center line of part of a road.

We use the term *network point* to refer to a point location anywhere on an edge.

Definition 11 *(Network Point) A road network point p is defined as $p = (eid, d)$, where eid is the identifier of an edge $e = (v_i, v_j)$ and d ($0 \leq d \leq 1$) is the ratio of the distance between vertex v_i and the point to the length of e. P denotes the set of all network points on the road network.*

It can be seen that given an edge (v_i, v_j) identified by eid, $v_i = (eid, 0)$ and $v_j = (eid, 1)$. Therefore we have $V \subset P$. For example, in Figure 5.1, the network point of f_1 is $f_1.p = (e_{2,3}, 0.5)$. Note that a vertex can have more than one eid, depending on the number of incident edges. So we use $v.EID$ to denote the set of eids of v. The distance between two network points p_i and p_j on the same edge e is defined as $dist(p_i, p_j) = \|e\| \cdot |d_i - d_j|$.

A road segment is a polyline that starts at a network point, traverses a sequence of vertices, and ends at a network point.

Definition 12 *(Road Segment) A road segment s is defined as a sequence of network points, $s = \langle p_1, p_2, \ldots, p_n \rangle$, where $n \geq 2$, $p_1, p_n \in P$, $p_i \in V$, $\exists eid$ such that $eid \in p_1.EID$ and $eid \in p_2.EID$, $\exists eid$ such that$eid \in p_{n-1}.EID$ and $eid \in p_n.EID$ and $(p_i,\ p_{i+1}) \in E\,(1 < i < n-1)$.*

The length of s is the network distance *from p_1 to p_n.*

$$\text{length}(s) = \begin{cases} dist(p_1, p_2) & n = 2 \\ dist(p_1, p_2) + \sum_{i=2}^{n-2} \|e_{i,i+1}\| + dist(p_{n-1}, p_n) & n > 2 \end{cases}$$

The set of road segments is denoted as S.

It follows from definition that an edge is also a segment, i.e., $E \subset S$. Further, we use the notion *route* for a segment that a customer has traversed.

When there is no ambiguity from the context, we use $\overline{AB}$ to mean the segment between network points A and B. For example, the short segment between A and H in Figure 5.1 is $\overline{AH}$. Otherwise, we write the segment in full, e.g., the road segment $\langle p_{f_3}, v_5, v_6, p_{f_4} \rangle$ between facilities f_3 and f_4, supposing the network points for facilities f_3 and f_4 are p_{f_3} and p_{f_4}, respectively.

5.2.2 Facilities and Route Usage

A facility f located at a network point p is denoted as (fid, p), where fid identifies the facility. F denotes the set of all facilities.

A route is a segment and thus starts at a network point, traverses a sequence of connected edges, and stops at a network point. The same route can be traversed many times by the same or many customers. For instance, many customers who live in the same building may take the same route r to the same grocery store. We use $count$ to denote the number of customers who take r. On the other hand, one customer can take the same route many times, e.g., a customer may take the same route from home to work on most weekdays. We use $usage_i$ to denote the number of times r is taken by customer i ($1 \leq i \leq count$).

Definition 13 *(Route Usage Object) A route usage object ro is defined as*

$$ro = (rid, r, count, \langle usage_1, \ldots, usage_{count} \rangle)$$

where rid identifies the object, $r \in S$ is a segment traversed by the user. R is a set of all route usage objects.

A route $ro.r$ *covers* a road segment s if $\forall p \in s' \, p \in ro.r$. A route $ro.r$ *intersects* a segment s if $\exists p \in s \, p \in ro.r$. The set of route usage objects whose routes cover s is denoted as $s.C$. The set of route usage objects whose routes intersect s is denoted as $s.I$. It is straightforward to see that $s.C \subseteq s.I$. We also say that $ro_1 \equiv ro_2$ if $ro_1.r = ro_2.r$.

In Figure 5.1, the routes r_1, r_2, and r_3 are traversed by three different customers. We assume that each route is traversed once by each customer.

A route r is *attracted* by a facility f and f is an *attractor* for r if $dist_G(f.p, ro.r) \leq \delta$, where $dist_G(p, s)$ gives the shortest network distance between a network point p and a segment s and δ is the distance threshold that was introduced earlier. Note that the same facility can attract several routes. In Figure 5.1, facility f_1 attracts routes r_1 and r_2, and f_3 attracts r_3.

5.2.3 Scoring a Route

In the optimal segment problem, route traversals play the role that customer locations play in the classical formulation of the optimal location problem. Thus, we need to decide how to assign a score to a route based on the traversals of the route. The scoring of a route is thus based on three factors that are all captured in the route usage object for the route: the number of customers taking the route, the number of traversals by each customer, and the length of the route. The route's score is subsequently distributed among the segments covered by the route. The intuition of distributing the score of a route to its segments is that when a customer traverses the route, the customer may visit facilities located on segments along the route.

To ensure that the framework yields meaningful results, the scores eventually assigned to segments must be invariant under the splitting and concatenation of route usage objects. To achieve this, we require the following property to hold.

Route Scoring Property A route scoring function should be independent of the partitioning of the route of a route usage object. Let $ro_1 \circ ro_2$ be the concatenation of $ro_1.r$ and $ro_2.r$. Let $ro = (id, r_1 \circ r_2 \circ \cdots \circ r_m, count, u)$ and $ro_i = (id_i, r_i, count, u)$ $(1 \leq i \leq m)$. Then we require $score(r) = \sum_{i=1}^{m} score(r_i)$.

This property ensures that partitioning a route usage object or combining several route usage objects with the same count and usages does not change the total score that is available for assignment to segments.

Many scoring functions are possible that satisfy the property. Next, we show two of them.

Definition 14 *(Scoring a Route) Let a route r with an associated route usage object $ro = (rid, r, count, \langle usage_1, \ldots, usage_{count}\rangle)$ be given. Then the score of r can be defined as follows*

$$score_{all}(r) = \text{length}(r) \sum_{i=1}^{count} ro.usage_i$$

$$score_{cap-x}(r) = \text{length}(r) \sum_{i=1}^{count} \min(ro.usage_i, x)$$

where x is a user-defined value.

Depending on the products or services offered by a facility, different scoring functions may be appropriate. For example, a facility that sells everyday necessities (e.g., a bakery) may attract the same customer on each route traversal by the customer. Thus, $score_{all}$ is appropriate. In contrast, if a store sells products that are bought less frequently (e.g., a furniture store), the store may not benefit from a large number of traversals by the same customer, making $score_{cap-x}$ more appropriate. Thus, we keep the framework open to the use of different scoring functions.

Unless specified otherwise, we use the function $score_{all}$ for illustration.

In Figure 5.1, assuming that the lengths of routes r_1, r_2, and r_3 are 2, 4, and 3, and the number of traversals per customer are $\langle 2, 2\rangle$, $\langle 2, 1\rangle$, and $\langle 2\rangle$, respectively. Then we have three route usage objects: $ro_1 = (id_1, r_1, 2, \langle 2, 2\rangle)$, $ro_2 = (id_2, r_2, 2, \langle 2, 1\rangle)$, and $ro_3 = (id_3, r_3, 1, \langle 2\rangle)$. The score of r_1 can be calculated as follows, $score(r_1) = \text{length}(r_1) \cdot (ro_1.usage_1 + ro_2.usage_2) = 2 \cdot (2 + 2) = 8$ Similarly, we calculate the scores of r_2 and r_3, $score(r_2) = 12$ and $score(r_3) = 6$.

5.2.4 Score Distribution Models

A score distribution model determines how to distribute the score of a route to the underlying segments. Interest models are orthogonal to the scoring of route, i.e., they do not affect how a route is scored.

Intuitively, segments covered by a route with many traversals that are *not attracted* by many other facilities are good candidates for placing a new facility. Therefore, they should be assigned high scores. But customers can have different spatial preferences for visiting different kinds of businesses. Therefore, we leave the framework open to the use of different score distribution models.

When n facilities are located on a segment, they partition the segment into k subsegments where k is one of $n - 1, n$, or $n + 1$ depending on whether two, one, or no facilities are located at the ends of the segment.

The following are example score distribution models.

- Equal weight is assigned to each subsegment. In this model, the score of a route r is distributed such that a customer has an equal probability to visit any business along the route. For example, any clothing store on the way back home. The score assigned to the ith subsegment s_i, $1 \leq i \leq k$ is $\frac{1}{k} \cdot score(r)$.

- Decreasing/increasing weights are assigned to the subsegments. The score of the ith subsegment s_i ($1 \leq i \leq k$) is given by $\frac{1}{2^i} \cdot \frac{1}{\sum_{i=1}^{k} \frac{1}{2^i}} \cdot score(r)$, $(1 \leq i \leq k)$. This definition gives exponentially decreasing scores to subsegments and normalizes the scores such that the full score of the route is distributed. This model indicates a preference for the facilities at the beginning of the route. Symmetrically, there is a model that prefers the facilities at the end of the route. For example, a customers might prefer to have a meal before the trip back home or to work, but it is also possible (with lower probability) that the customer will visit any restaurant along the route

- All of the score is evenly distributed to the first and the last subsegment. This model indicates that customers consider only businesses that are located nearest to the route destinations. For example, a customer would like to visit the store, which sells dairy products, closest to home, but for regular items closest store to work place can be used.

- The original *facility location* problem considers simply the attraction of customer locations to facility locations. In our setting, where customer route traversals are attracted to segments where facilities may be placed, the model that assigns the entire score of a route traversal to the route's first subsegment may be the one that most closely resembles the original problem.

In Figure 5.1, route r_2 is attracted by facilities f_1 and f_2, and $k = 3$. In previous examples, we showed that the score of r_2 is 12. According to the first proposed score distribution model, each subsegment ($\overline{v_1 f_1}$, $\overline{f_1 f_2}$, and $\overline{f_2 B}$) receives score $\frac{score(r_2)}{k} = \frac{12}{3} = 4$. According to the second model, $\frac{1}{\sum_{i=1}^{k} \frac{1}{2^i}} = \frac{1}{\frac{1}{2}+\frac{1}{4}+\frac{1}{8}} = \frac{1}{\frac{7}{8}} = \frac{8}{7}$, $score(\overline{v_1 f_1}) = \frac{1}{2} \cdot \frac{8}{7} \cdot 12 = \frac{4}{7} \cdot 12$, $score(\overline{f_1 f_2}) = \frac{1}{4} \cdot \frac{8}{7} \cdot 12 = \frac{2}{7} \cdot 12$, and $score(\overline{f_2 B}) = \frac{1}{8} \cdot \frac{8}{7} \cdot 12 = \frac{1}{7} \cdot 12$.

So far, we have distributed the score assigned to a single route to the segments covered by the route. However, a segment s may be covered by multiple routes that assign score to the segment. The total score of the segment, $score_M(s)$, where M indicates the score distribution model used, is simply the sum of these scores. Similarly, we can calculate the score of a network location p, $score_M(p)$.

We show how to score the subsegment $\overline{AH}$ in Figure 5.1 using the first proposed model. $\overline{AH}$ is covered by all of the three route usage objects. So $score_M(\overline{AH}) = \sum_{i=1}^{3} \frac{score(ro_i.r)}{k_i} = \frac{8}{2} + \frac{12}{3} + \frac{6}{2} = 11$.

Since our framework is generic with respect to. score distribution models, unless specified otherwise, we use the first model for illustration.

5.2.5 Problem Formulation

With the above definitions in place, we can define the optimal segment query.

Definition 15 *(The Optimal Segment Query) The optimal segment query finds every segment s_{opt} from a road network G such that*

1. $\forall p_1, p_2 \in s_{opt} : score_M(p_1) = score_M(p_2)$
2. $\forall p \in s_{opt} \, \forall p' \in P : score_M(p') \leq score_M(p)$
3. $\nexists s' \subseteq S : s_{opt} \subset s'$ *and 1 and 2 hold.*

This definition ensures that every point in the optimal segment has the same score, that the score is optimal, and that the optimal segment is maximal.

Notation used introduced this section and to be used throughout the chapter is summarized in Table 5.1.

R	The set of route usage objects
F	The set of facilities
S	The set of road segments
P	The set of sites
G, G'	The (augmented) road network graph
V, V'	The set of vertices in G, G'
E, E'	The set of edges in G, G'
n	The total number of GPS points in R
δ	The maximum distance of attraction

Table 5.1: Summary of Notation

5.3 Preprocessing

A straightforward approach to compute the optimal segment query is to enumerate and score all possible segments and then return the one with the highest score. However, this is not feasible as this approach has to test infinite number of possible segments. As will be shown by our scheme, only finite number of segments or edges need to be compared in order to find the optimal segments, because optimal segments are flanked by vertices or route start and end points which are finite.

The two algorithms we propose both rely on the same preprocessing algorithm, which we present here. This algorithm determines the relationships between the facilities and the edges, between the routes and the edges, and between the facilities and the routes. It needs to be run only once for one set of routes.

The algorithm makes each edge record its facilities and route start and end points, if any. It also makes each vertex record the covering routes' identifiers. The routes record the facilities they cover. The facilities record the edge they are located on and the covering routes, if any. Also the algorithm populates a lookup table so that given an edge, one can quickly determine the routes that intersect with the edge.

Recall that G is the spatially embedded graph, f is a facility, and r is a route. Algorithm PreProcess calls getEdges(f, G) to retrieve the edge(s) where f is located. Note that f may be located at a vertex and thus two edges. The function onEdge test if a network location is located on an edge. It also calls getEdges(r, G, δ) to retrieve the set of edges that intersect r.

The PreProcess procedure is presented in Algorithm 15 and explained next.

Algorithm 15: PreProcess(G, R, F, δ)

```
1  foreach $f \in F$ do
2      $E_1 \leftarrow \text{getEdges}(f, G)$;
3      foreach $e \in E_1$ do $e.F_c.\text{add}(f)$;
4      $f.E_c.\text{add}(E_1)$;
5  foreach $ro \in R$ do
6      $r \leftarrow ro.r$;
7      $r.E_c \leftarrow \text{getEdges}(r, G, \delta)$;
8      foreach $e = (v_s, v_e) \in r.E_c$ do
9          if $(\text{onEdge}(r.p_s, e)) \wedge (r.p_s.d \notin \{0, 1\})$ then
10             $e.O_c.\text{add}(r.p_s)$;
11         if $(\text{onEdge}(r.p_e, e)) \wedge (r.p_e.d \notin \{0, 1\})$ then
12             $e.O_c.\text{add}(r.p_e)$;
13         $e.R_c.\text{add}(r)$;
14         if $\text{contains}(r, e)$ then
15             $r.F_c \leftarrow r.F_c \cup e.F_c$;
16             foreach $f \in e.F_c$ do $f.R_c.\text{add}(r)$;
17         else if $\text{intersects}(r, e)$ then
18             $F' \leftarrow \{f | f \in e.F_c \wedge \text{attracts}(f, r, \delta)\}$;
19             $r.F_c \leftarrow r.F_c \cup F'$;
20             foreach $f \in F'$ do $f.R_c.\text{add}(r)$;
21         if $\text{attracts}(v_s, r, \delta)$ then
22             $v_s.R_c.\text{add}(r)$;
23             $i \leftarrow$ the position of $v_s$ relative to the $r.F_c$;
24             $v_s.L.\text{add}(i)$;
25         if $\text{attracts}(v_e, r, \delta)$ then
26             $v_e.R_c.\text{add}(r)$;
27             $i \leftarrow$ the position of $v_e$ relative to the $r.F_c$;
28             $v_e.L.\text{add}(i)$;
```

A facility f keeps the edge where it is located in the variable $f.E_c$, and the set of routes it attracts in $f.R_c$. An edge e keeps a set of route start and end network points that are located on e in $e.O_c$. This list is used by the AUG algorithm for augmentation purpose. Each vertex v of e maintains a list of routes that it attracts in $v.R_c$, and v's relative position in $v.L$. These two lists are used later by AUG for scoring purpose. $e.F_c$ and $e.R_c$ are the set of facilities that are located on e and the set of routes that intersect edge e, respectively. A route r keeps its set of attracting facilities in $r.F_c$.

For each facility f, PreProcess retrieves its edge E_1 so that e adds f to its set of facilities $e.F_c$, and $f.E_c$ adds E_1 (lines 1–4).

Next, for each route r, the set of intersected edges is retrieved, and the $r.E_c$ field is updated (lines 7)). Then, if the start network point of r is not a vertex in G, it is added to the $e.O_c$ set of the edge e where it is located. Similarly, r's end network point is added to a $e.O_c$ set. (lines 9–12). Route r is also added to the list $e.R_c$ (lines 13). For each edge e covered by the route r, the facilities and r record each other (lines 14–16). For each edge e intersected by a route r, on the other hand, the attraction relationship between the facilities and r is determined before updating each other's corresponding field (lines 17–20). Next, each vertex of e records r in the $v.R_c$ list if r is attracted by it. In addition, the relative position of v is also kept in $v.L$ for scoring purpose (lines 21–28).

In Figure 5.1, edge $e_{2,3}$ has $e_{2,3}.F_c = \{f_1\}$ and $f_1.e = e_{2,3}$. Route r_1 traverses one edge and is attracted by one facility, so, $r_1.E_c = \{e_{2,3}\}$ and $r_1.F_c = \{f_1\}$. Edge $e_{2,3}$ is covered by r_1, r_2 and r_3, so, $e_{2,3}.R_c = \{r_1, r_2, r_3\}$. Three start or end network points of the routes are located on edge $e_{2,3}$, so, $e_{2,3}.O_c = \{A, D, H\}$. Vertex v_2 is covered by r_2 and r_3 and vertex v_3 is covered r_2, so $v_2.R_c = \{r_2, r_3\}$ and $v_3.R_c = \{r_2\}$. Facility f_1 attracts r_1 and r_2, so $f_1.R_c = \{r_1, r_2\}$.

The following lemma states the time complexity of PreProcess.

Lemma 5.3.1 *Algorithm PreProcess has time complexity $O(|F| + |R||E_m|)$, where $|E_m|$ is the maximum number of edges that any route traverses.*

Proof 5.3.1 *The first loop in the algorithm takes time $O(|F|)$. In the second loop, the outer loop runs $O(|R|)$ times. The inner loop depends on the number of edges that a route* *traverses. Let $|E_m|$ be the maximal number of edges that any route traverses. Then the second loop has time complexity $O(|R||E_m|)$. In total, the time complexity is $O(|F| + |R||E_m|)$.*

5.4 Graph Augmentation

5.4.1 Overview

The main idea of the graph augmentation algorithm (AUG) is to augment the road network graph G with the facilities and the first and the last network points of each route. In the augmented graph $G' = (V', E')$ it is guaranteed that each route starts from a vertex and ends at a vertex. Meanwhile, each vertex in G' stores the identifiers of the covering routes.

Then each edge's score in G' can be calculated by summing up the scores distributed by the routes that cover both ends points. The score contributed by a route is calculated based on the specific score distribution model used, as discussed in Section 5.2.4.

Next, AUG examines every edge in G' with a score, and identifies the edges with the highest score (the optimal edges).

Finally, the algorithm maps the optimal edges back to the original graph G, where they are segments. Then AUG merges connected segments, if any, to form maximal segments, and returns them as the result.

Figure 5.2 illustrates the graph in Figure 5.1 after being augmented with routes r_1, r_2, and r_3 and facilities $f_1,\ f_2,$ and f_3. Note that each vertex in the augmented graph has a list of the identifiers of the routes that cover the vertex. We use $AL(v_i)$ to denote the attraction list of v_i. Intersecting the sets of two adjacent vertices gives the routes that cover the edge, whose score can then be calculated according to a score distribution model. For example, $AL(A) = \{r_1, r_2, r_3\}$ and $AL(H) = \{r_1, r_2, r_3\}$. So, the set of routes that cover edge $e_{A,H}$ is $AL(A) \cap AL(H) = \{r_1, r_2, r_3\}$. Then the score of $e_{A,H}$ is calculated based on the score distribution model used.

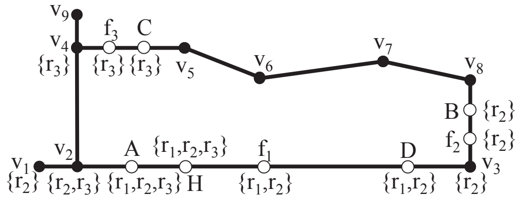


Figure 5.2: The Augmented Road Network Graph

Next, AUG finds the edges with the highest score by examining all edges in the augmented graph. These edges are then mapped back to the original graph, and become road segments, which are possibly merged into longer segments. These segments are returned as the result. In Figure 5.2, after edge $e_{A,H}$ is identified as the optimal edge with the highest score, it is mapped back to the original graph, and the segment $\overline{AH}$ is returned as the result.

5.4.2 The AUG Algorithm

Algorithm 16 presents details of the AUG algorithm. The set of edges that have network points either from facilities or routes is obtained (lines 1–2). Graph G' is obtained by augmenting graph G with the network points of F and R (line 3). Note that some network points of routes or facilities may happen to be vertices. These network points are excluded from being augmented into G. Then AUG updates the covering routes of the newly added vertices (lines 4–6). It initializes the result set S and the highest score seen so far, $optS$ (line 7). In the next loop (lines 8–14), the scores of the edges in G' are calculated according to the score distribution model used (line 10), and the optimal edges in G' are identified and stored in S. In lines 15–17, each edge in G' is mapped back to G. Mapping back to the original graph is a trivial task. Recall that each new network point has an eid field that helps identify the original edge. If a segment can be extended (i.e., the neighbouring segment is also an optimal segment), it is extended (line 17). This procedure is done by the canExtend method, which also detects cycles with an internal Depth First Search (DFS). Finally, the result set S is returned (line 18).

This process has two implications. First, an edge in G may be split into several edges in

Algorithm 16: AUG(G, R, F, δ, M)

```
1  $E_F \leftarrow$ the set of edges where $F$ are located;
2  $E_R \leftarrow$ the set of edge that $R$ intersect;
3  $G' \leftarrow \text{Augment}(G, F, R)$;
4  foreach $e \in (E_F \cup E_R)$ do
5      foreach $v_i \in e.F_c \cup e.O_c$ do
6          $v_i.R_c \leftarrow \text{getCoverRouteIds}(v_i, e.R_c)$;
7  $S \leftarrow \emptyset$; $optS \leftarrow 0$;
8  foreach $e = (v_s, v_e) \in G'.E'$ do
9      $R' \leftarrow v_s.R_c \cap v_e.R_c$;
10     $score_M(e) \leftarrow$ compute the score of $e$ based on $M$;
11     if $score(e) > optS$ then
12         $optS \leftarrow score(e)$;
13         $S \leftarrow \{e\}$;
14     else if $score(e) = optS$ then $S \leftarrow S.\text{add}(e)$;
15 foreach $s \in S$ do
16     map $s$ to $G$;
17     if $\text{canExtend}(s, G)$ then $\text{extend}(s, G)$;
18 return $S$;
```

G'. After the optimal edges are identified in G', they must be mapped back to G. Second, for an edge in G', a route either covers it or does not cover it. The *partial intersection* relationship between a route and an edge is eliminated in G'.

It can be seen that it is sufficient to just augment the original graph with the start point and the end point of each route for finding the optimal segments because the internal points in a route are vertices in G.

The algorithm splits some road segments.

- If the two end points of a route do not happen to be vertices in G, they are added as new vertices into the road network, as they are covered by at least one route.

- If a facility does not happen to be a vertex in G, it is added as a new vertex if it attracts any route, e.g., f_1, f_2, and f_3 in Figure 5.2. Facility f_4 no longer exists in the augmented graph because it does not attract any routes.

- In order to accommodate these new vertices, some edges in G are replaced with "smaller" edges in G'. For example, in Figure 5.2, the edge $e_{2,3}$ is replaced with the following edges: $e_{v_2,A}$, $e_{A,H}$, e_{H,f_1}, $e_{f_1,D}$, and e_{D,v_3}.

When the road network is augmented, every vertex in G' records the identifiers of the routes that cover this location. Figure 5.2 also shows the identifiers of the routes that are recorded at each vertex in the augmented graph.

In Figure 5.2, r_2 has score 12, and is attracted by 2 facilities. Thus, each edge in G' that is covered by r_2 should receive a score $\frac{score(r_2)}{3} = 4$. Route r_1 has score 8, and is attracted by one facility. Each edge covered by it in G' receives a score $\frac{score(r_1)}{2} = 4$. Route r_3 has score 6, and is attracted by 1 facility. Therefore, each covered edge received score $\frac{score(r_3)}{2} = 3$.

For each edge in the augmented road network, the algorithm takes an intersection of the route identifiers of its two vertices, and computes its score. For instance, $score(e_{v_2,A}) = 4 + 3 = 7$, $score(e_{A,H}) = 4 + 4 + 3 = 11$. The scores of other edges can be computed in a similar way.

After that, AUG identifies the optimal edge(s) with the highest score. Since $\overline{AH}$ has the highest score in G', the optimal edge is (A, H). It is mapped back, and becomes the segment $\overline{AH}$. As AUG cannot extend it to a longer segment, $\overline{AH}$ is returned as the result.

5.4.3 Analysis

We analyze the time complexity of the AUG algorithm, and show its completeness and correctness.

Theorem 5.4.1 *The AUG algorithm has time complexity $O((|E| + |F| + |R|)|R| + |S|)$, where S is the result set.*

Proof 5.4.1 *In AUG, the graph augmentation takes $O(|F| + 2|R|)$ (line 4), because each route contributes exactly two vertices.*

In the first nested loop (lines 5–9), the worst case is that facilities and routes are evenly distributed to the road network so that every edge in G is augmented. In this case, for an edge e, $e.F_c + e.O_c = \frac{|F|+2|R|}{|E|}$. The outer loop takes $|E_F| + |E_R| \leq |E|$. The complexity of the nested loop is the product of the two loops. That is, $(e.F_c + e.O_c) \cdot (|E_F| + |E_R|) \leq |E| \cdot \frac{|F|+2|R|}{|E|} \leq |F| + 2|R|$. So the nested loop takes time $O(|F| + 2|R|)$.

In the second loop (line 11–18), $|E'| \leq |E| + |F| + 2|R|$. In line 15, $|R'| \leq |R|$. Therefore, this loop takes time $O((|E| + |F| + 2|R|)|R|)$. The third loop takes time complexity $|S|$. Note that $|S|$ is usually very small.

To summarize, the time complexity of AUG is $O(|F|+2|R|+(|E|+|F|+2|R|)|R|+|S|)$. After simplification, the time complexity is $O((|E| + |F| + |R|)|R| + |S|)$.

We proceed to show the correctness and completeness of AUG.

Theorem 5.4.2 *A segment output by the AUG algorithm is an optimal segment.*

PROOF SKETCH. In AUG, every edge in the augmented graph is checked to find the the value for $optS$. AUG then adds a segment iff the segment has a score equal to $optS$. It implies that any segment in the result set S must be optimal.

Theorem 5.4.3 *The AUG algorithm finds every optimal segment in the graph.*

PROOF SKETCH. This theorem can be proven with the following two points. First, AUG searches the graph to make sure that every edge is scanned. Second, if an edge has a score equal to $optS$, it either is appended to an existing segment in S or is added to S as a new segment that might be extended later. Therefore, no segment with score equal to $optS$ is missed.

5.5 Iterative Partitioning

5.5.1 Overview

Although the augmentation approach is effective at finding the optimal segments, we can improve its efficiency by pruning unpromising segments.

The idea of the ITE algorithm is to quickly identify a subsegment of an optimal segment (*optimal subsegment*) and then extend the optimal subsegment into an entire optimal segment. Therefore, ITE organizes the segments using a heap such that those segments that are most likely to contain an optimal subsegment get examined first. If the segment under examination is an optimal subsegment then the entire optimal segment can be found by extending it. In addition, the optimal score can be calculated easily. Otherwise, the segment is partitioned into smaller segments, whose likelihoods of having an optimal subsegment are also calculated, upon which they are inserted back into the heap.

Given a segment s, we use the scores of the intersecting routes to measure its likelihood of having an optimal subsegment. The segment containing the optimal subsegment is likely to have many intersecting routes, from which it is likely to receive a high score.

For example, in Figure 5.1, initially the edges that intersect any route are inserted into the heap. The edge v_1v_3 has the most intersecting routes and so is likely to contain an optimal segment. So v_1v_3 is partitioned into equal-sized, smaller segments. ITE calculates the intersecting routes for each of them, and adds them to the heap. This process continues until a subsegment of an optimal segment is found. In this case, a subsegment s of $\overline{AH}$ is found. Then s is extended to find that $\overline{AH}$ is the entire optimal segment.

Both AUG and ITE partition the edges of the network graph into smaller pieces. The main difference between ITE and AUG lies in how a subsegment of the optimal segment is found. In AUG, the partitioning of edges in the network graph is unguided. Every edge that has an attracting facility or a route end point is partitioned. In ITE, the partitioning of edges is guided by the likelihoods of the edges to have an optimal subsegment.

5.5.2 The ITE Algorithm

Recall that we are interested in finding those segments that contain an optimal subsegment. Before presenting the ITE algorithm, we need definitions that relate the score of a segment to the scores of its network points, as defined in Section 5.2.4.

Definition 16 *Given a road segment s, we define its min score $s.min$ and max score $s.max$ as follows.*

$$s.min = \min_{p \in s} score_M(p)$$
$$s.max = \max_{p \in s} score_M(p)$$

By definition, an optimal segment s_{opt} has $s_{opt}.min = s_{opt}.max$.

Next, we define upper and lower bound scores of a segment s in order to only process those segments that may contain an optimal location.

Definition 17 *Given a segment s and a score distribution model M, let $s.I$ and $s.C$ be defined as in Section 5.2.2, and let $s.lb$ and $s.ub$ denote the upper and lower bound scores of s. We define:*

$$s.lb = \sum_{r_i \in s.C} w_M(j_i, k_i) score(r_i)$$
$$s.ub = \sum_{r_i \in s.I} w_M(j_i, k_i) score(r_i)$$

where s is the j_ith segment of r_i with k_i attracting facilities, and $w_M(j_i, k_i)$ computes the fraction of r_i's score to be assigned to s based on M.

If a segment has facilities located on it, the segment has subsegments that may be assigned different scores based on the score distribution model. In this case, the lower bound score of the segment still takes the smallest score value being assigned to the subsegments, while the upper bound score takes the largest score value.

Lemma 5.5.1 *Let M be a score distribution model where a route can only distribute non-negative scores. Let the min and max scores and the lower and upper bound scores of s be defined as above. Given a road segment s, we have $s.lb \leq s.min$ and $s.ub \geq s.max$.*

Proof 5.5.1 *Suppose a network location $p_1 \in s$ s.t. $score_M(p_1) = s.min$. Since each route $r \in s.C$ contains s, we have $p_1 \in r$. So p_1 at least gains the scores distributed by the routes in $s.C$. Then $score_M(p_1) \geq \sum_{r_i \in s.C} w_M(j_i, k_i) score(r_i)$.*

Let $p_2 \in s$ be a location s.t. $score_M(p_1) = s.max$. We show that the set of routes that contribute scores to p_2 is a subset of $s.I$. The set of routes that contribute scores to p_2 consists of two sets, the set of routes that contain s ($s.C$) and the set of routes that cover p_2 ($s.I'$). Each route in $s.I'$ must also intersect s, so $s.I' \subset s.I$. Since $s.C \subseteq s.I$, we have $(s.C \cup s.I') \subseteq s.I$. That is, $score_M(p_1) \leq \sum_{r_i \in s.I} w_M(j_i, k_i) score(r_i)$.

Recall that Algorithm PreProcess builds a mapping from each edge e to its intersecting routes $e.R_c$. We then compute the upper and lower bound scores for a segment $s \subseteq e$ by retrieving its $s.I$ and $s.C$ from $e.R_c$. The algorithm can use the bounds to prune the segments that cannot contain an optimal subsegment.

Lemma 5.5.2 *Given two segments s_1 and s_2, if $s_1.lb > s_2.ub$, then s_2 does not contain an optimal subsegment.*

Proof 5.5.2 *We prove lemma 5.5.2 by showing that s_2 cannot contain any optimal location. Assume two points $p_1 \in s_1$ and $p_2 \in s_2$. We have $score_M(p_1) \geq s_1.lb > s_2.ub \geq score_M(p_2)$. So p_2 cannot be an optimal location.*

With Lemma 5.5.2, segments that do not contain an optimal subsegment can be pruned.

The second strategy employed in ITE is to prune the segments that eventually lead to the same optimal segment. These segments should be detected and pruned early to avoid partitioning them further and making unnecessary calculations.

Lemma 5.5.3 *Given two segments s_1 and s_2, if $s_2.I \subset s_1.C$ and s_2 contains an optimal subsegment of an optimal segment, then s_1 also contains an optimal subsegment of the same optimal segment.*

Proof 5.5.3 *Let the optimal segment be s_{opt}, and let s_2 contain an optimal subsegment of s_{opt}. Then we have $s_{opt}.C \subseteq s_2.I$ because every route that contain the optimal subsegment must intersect with s_2.*

Since $s_2.I \subseteq s_1.C$, we have $s_{opt}.C \subseteq s_1.C$. By the definitions of segment score and optimality, we also have $s_{opt}.C = s_1.C$. Therefore, by the definition of segment score, s_1 is also an optimal subsegment of s_{opt}.

Once the result set is not empty, Lemma 5.5.3 allows us to prune segments that lead to the same optimal segment. We study the effectiveness of the pruning strategies in the experimental evaluation.

Figure 5.3 shows the edge $e_{2,3}$ from Figure 5.1. It illustrates the calculation of segment score upper bound and lower bound. The edge $e_{2,3}$ has a facility f_1 built on it, resulting in two subsegments, s_1 from the beginning to f_1 and s_2 from f_1 to the end. The routes r_1 and r_3 are attracted by one facility, whereas r_2 is attracted by two facilities. We show how to calculate the score upper and lower bounds for both s_1 and s_2. Segment s_1 is intersected by r_1 and r_3, and contained by r_2. Therefore, $s_1.lb = \frac{1}{3}score(r_2)$, $s_1.ub = \frac{1}{2}score(r_1) + \frac{1}{3}score(r_2) + \frac{1}{2}score(r_3)$

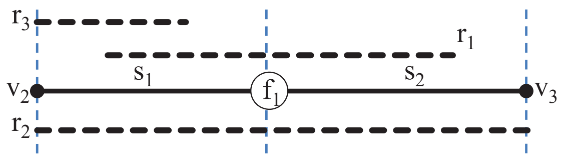


Figure 5.3: Segment Upper and Lower Bound

Similarly, s_2 is intersected by r_1, and contained by r_2. Therefore, $s_2.lb = \frac{1}{3}score(r_2)$, $_2.ub = \frac{1}{2}score(r_1) + \frac{1}{3}score(r_2)$

Algorithm 17 shows the pseudo-code of the ITE algorithm. This algorithm uses a priority queue Q that is sorted on the upper bound score of every segment. A variable called $maxLb$ is used to keep track of the maximum lower bound score seen so far.

Algorithm 17: ITE$(G, R, F, \delta, \beta, M)$

```
Init. $e \in G.E$ s.t. $e.lb \leftarrow 0$ and $e.ub \leftarrow \sum_{r_i \in e.R_c} w_M(j_i, k_i) score(r_i)$;
$S \leftarrow \emptyset$; $Q.enqueue(G.E)$; $maxLb \leftarrow 0$;
while $Q \neq \emptyset$ do
    $currSeg \leftarrow Q.dequeue$;
    $split \leftarrow false$;
    if $currSeg.ub > maxLb$ then
        $split \leftarrow true$;
    else if $currSeg.ub = maxLb$ then
        if $currSeg.ub = currSeg.lb$ then
            $S.add.(currSeg)$;
        else if $\nexists s \in S$ such that $currSeg.I \subseteq s.C$ then
            $split \leftarrow true$;
    if $split$ then
        $ss \leftarrow$ SplitSegment$(currSeg, \beta)$;
    foreach $s \in ss$ do
        foreach $r \in currSeg.I$ do
            if intersects$(r, s)$ then
                $s.I \leftarrow s.I.$add$(r)$;
                $s.ub \leftarrow s.ub + w_M(j, k) score(r)$;
            if contains$(r, s)$ then
                $s.C \leftarrow e.C.$add$(r)$;
                $s.lb \leftarrow s.lb + w_M(j, k) score(r)$;
        if $s.lb > maxLb$ then
            $maxLb \leftarrow s.lb$;
        $Q.enqueue(s)$;
foreach $s \in S$ do
    Find the entire optimal segment of $s$ by overlapping the route usage objects
    $r \in s.C$ one by one.
```

First, ITE initializes the edges such that each has a lower bound score 0 and an upper bound score computed as in Definition 7 (line 1). It also initializes the result set S, enqueues the edges $G.E$ of the road network graph, and initializes variable $maxLb$ (line 2). It then enters the loop and pops out the top element from Q (lines 3–4). The flag variable $split$, indicating whether or not the current segment needs to be partitioned, is set to false at the beginning of each iteration (line 5). Next, if the upper bound score of $currSeg$ exceeds $maxLb$ then it needs to be further partitioned, so $split$ is set to true (lines 6–7). If the upper bound score of $currSeg$ is equal to $maxLb$, we have found a result segment if the upper and lower bound scores are the same. Then $currSeg$ is added to the result set (lines 8–10). However, if the upper and lower bound scores differ, ITE tests whether $currSeg$ might lead to an optimal subsegment of a new optimal segment that is not seen before. Then ITE checks if there is a result s in S such that $s.C$ is subset of $currSeg.I$ (see Lemma 5.5.3). If no, $split$ is set to true (lines 11–12).

If $split$ is true, the function partitions $currSeg$ into subsegments with the procedure $\mathrm{SplitSegment}$, which partitions a segment G into β equal length subsegments (lines 13–14). Here β is a tunable parameter. In the experimental studies, we show the effect of β.

The intersection set, contain set, and lower and upper bound scores for each segment output by $\mathrm{SplitSegment}$ are computed and inserted into Q (lines 15–22). Next, $maxLb$ is updated if the subsegment has a higher lower bound score (lines 23–24). Then these subsegments are added back to Q (line 25).

Upon exiting the loop, each optimal segment is extended to its full length by overlapping the routes that contribute scores to the segment (lines 26–27).

We continue to use Figure 5.1 to illustrate the execution of Algorithm 17. We show the iterative partitioning of $e_{2,3}$ in Figure 5.4. Table 5.2 shows the top entries of the queue obtained from partitioning $e_{2,3}$, together with their upper and lower bound scores during the execution of ITE. Double lines separate iterations. The segment at the top of the queue is in bold.

Below, we calculate the upper and lower bound scores of $\overline{v_2p_1}$, which is intersected with r_1, r_2, and r_3, and contained by r_2 and r_3. Therefore, $ub(\overline{v_2p_1}) = \sum_{i=1}^{3} score(r_i)k_i = \frac{8}{2} + \frac{12}{3} + \frac{6}{2} = 11$ and $lb(\overline{v_2p_1}) = \sum_{i=2}^{3} score(r_i)k_i = \frac{12}{3} + \frac{6}{2} = 7$. The upper and lower bound scores of other segments can be computed in a similar way.

Since segment $\overline{v_2p_1}$ has the largest upper bound score and its upper bound is not the same as the lower bound, it is split as shown in Figure 5.4(b). The upper and lower bound scores of the subsegments are also computed. Now $maxLb = 11$

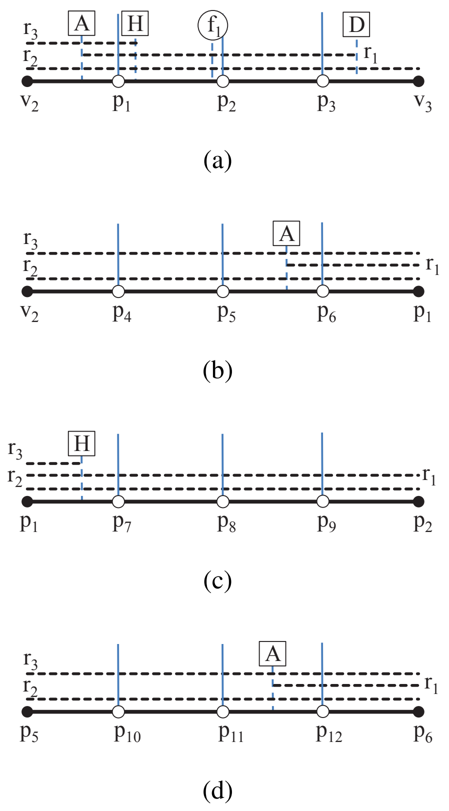


Figure 5.4: ITE Execution Example

In the next iteration, segment $\overline{p_1p_2}$ has the largest upper bound score. But still, its upper bound is not the same as its lower bound. It is then split (Figure 5.4(c)). The upper and lower bound scores of the subsegments are also computed, and $maxLb = 11$.

The next segment under examination is $\overline{p_5p_6}$, which is split again because its upper bound is different from its lower bound (Figure 5.4(d)). Still, $maxLb = 11$.

The next segment under examination is $\overline{p_6p_1}$, whose upper and lower bounds are the same. The upper bound of $\overline{p_6p_1}$ is also same as $maxLb$. Therefore, $\overline{p_6p_1}$ is added into the result set as an optimal subsegment.

The process continues until an optimal subsegment of every optimal segment in the network graph is found. Q is updated at the end of each iteration. Note that ITE does not need to examine those segments with score upper bound less than $maxLb$ (11 in this case), resulting in a substantial reduction of the search space.

Segment	ub	lb	Segment	ub	lb
$\overline{v_2p_1}$	**11**	**7**	$\overline{p_2p_3}$	8	8
$\overline{p_1p_2}$	11	8	$\overline{p_3v_3}$	8	4
$\overline{p_1p_2}$	**11**	**8**	$\overline{p_2p_3}$	8	8
$\overline{p_5p_6}$	11	8	$\overline{p_3v_3}$	8	4
$\overline{p_6p_1}$	11	11	$\cdots$		
$\overline{p_5p_6}$	**11**	**8**	$\overline{p_2p_3}$	8	8
$\overline{p_6p_1}$	11	11	$\overline{p_3v_3}$	8	4
$\overline{p_1p_7}$	11	8	$\cdots$		
$\overline{p_6p_1}$	**11**	**11**	$\overline{p_{12}p_6}$	11	11
$\overline{p_1p_7}$	11	8	$\overline{p_2p_3}$	8	8
$\overline{p_{11}p_{12}}$	11	7	$\cdots$		

Table 5.2: ITE Execution Example

In the end, the entire optimal segment $\overline{AH}$ can be found by overlapping the routes $\overline{AH}.C$, r_1, r_2, and r_3.

5.5.3 Analysis

We consider the correctness and completeness of ITE and analyze its time complexity.

The correctness of ITE depends on finding the subsegments with the maximum score correctly. Here, we prove that the algorithm terminates and returns subsegments that have the maximum score. We must show that after a finite number of iterations, ITE produces a subsegment s such that $s.ub = s.lb$ where $s.ub$ is the maximum score among all the subsegments. First, we know that when $s.ub = s.lb$, s is a consistent segment with the score $s.ub$. Since $s.ub$ is the maximum among all the subsegments ensured by the property of the priority queue, s is a subsegment with the maximum score. Since ITE always examines the subsegment with the maximal $s.ub$, we only need to show that ITE terminates. This can be shown by the following properties. (1) The maximum ub value decreases. (2) The maximum lb value increases. (3) The maximum ub and lb values converge to the same value after a number of iterations. (1) and (2) are straightforward because longer segments intersect with more routes and subsegments are more likely to be contained in a set of routes. To show the correctness of (3), line 23 in Algorithm 17 has to be true indefinitely often. It is so because there is at least one segment in the priority queue with its lb same as $maxLb$ and this removes the possibility that line 23 is not true from certain iteration. Since its ub keeps decreasing, ub converges to lb eventually and the algorithm terminates.

Next, to prove completeness, we show that for each optimal segment, ITE is able to find a subsegment with the maximum score that is contained within the optimal segment. Let s_i and s_j be the subsegment of two distinct optimal segments. Without loss of generality, suppose ITE has found s_i. We show that ITE also finds s_j instead of pruning it. Recall that ITE uses two pruning criteria to prune a subsegment. The first criterion says that s_j can be pruned if $s_i.lb > s_j.ub$. Since both s_i and s_j are subsegments of optimal segments with the same optimal score OPT, we have $s_j.ub \geq \mathrm{OPT} \geq s_i.lb$. Therefore, this pruning criterion does not apply. The second criterion states that s_j can be pruned if $s_j.I \subseteq s_i.C$. Since s_i and s_j are subsegments of different optimal segments, $s_j.C \nsubseteq s_i.C$. We also know that s_j is a subsegment with the maximum score, hence $s_j.I = s_j.C$. Putting them together, we have $s_j.I = s_j.C \nsubseteq s_i.C$. Hence the second pruning criterion also do not apply. Thus, ITE does not prune s_j, but detects it as a part of an entire segment which is also found. Therefore, ITE will find all the optimal segments.

Theorem 5.5.1 *The time complexity of ITE is $O((log|R| + |S|)|R|)$.*

Proof 5.5.4 *In the priority queue operations, ITE iteratively splits the segment with the maximal score upper bound. The number of splits corresponds to the height of the tree with fan-out β. If $\beta = 4$, we get a quadtree. According to [29], the asymptotic height of the quadtree is $log|R|$. For each subsegment s, ITE uses a loop to find its intersection set $s.I$ and contain set $s.C$. We have $ss.I \subseteq R$, so the time complexity of the loop is $O(|R|)$. Thus, time complexity of the while loop is $O(|R|log|R|)$.*

The second loop depends on the size of the result set S. Since $s.C \subseteq R$, the time complexity of this loop is $O(|R||S|)$.

In total, the time complexity of ITE is $O((log|R| + |S|)|R|)$.

5.6 Finding topK segments

In real life, often there are factors that may not be taken into account the scoring function, e.g. policy-related factors and qualitative factors, our scheme is able to find topK segments in order to provide the user with more choices. For example, when building a new gasoline service station, decision makers might need to consider issues such as land price and pollution control. In this case, the right candidate may not be the optimal segment. Hence, our scheme computes topK segments to provide more choices.

5.6.1 AUG-topK

The original AUG algorithm finds top1 segment. It is straightforward to extend it to find topK segments. In AUG-topK, as in AUG, the graph is first augmented and the score for each edge in the augmented graph is calculated (lines 1–6 in AUG). AUG-topK then initializes an empty candidate set to store the best candidates found so far. While iterating through the edges in the augmented graph (corresponding to the second loop in AUG), the edges with the highest topK scores seen so far are kept in the set. Then every edge in the set is mapped back to the original graph to find its corresponding segment (the third loop in AUG). In the end, the set, containing topK optimal segments, is returned as the result. Note that in order to output k different segments, AUG-topK takes adjacent edges with the same score as one in the second loop.

Following the example in Figure 5.1. Suppose $k = 2$. That is, AUG finds top2 segments. After augmentation, the network graph is the same as in Figure 5.2. The score for each edge in the augmented graph is computed. From the above, we already know that $\mathrm{score}e_{A,H} = 11$. Slightly smaller than $\mathrm{score}\overline{AH}$, $\mathrm{score}e_{H,f_1} = \frac{score(r_1)}{1+1} + \frac{score(r_2)}{2+1} = \frac{8}{2} + \frac{12}{3} = 8$, $\mathrm{score}e_{f_1,D} = \frac{score(r_1)}{1+1} + \frac{score(r_2)}{2+1} = \frac{8}{2} + \frac{12}{3} = 8$, and $\mathrm{score}e_{v_2,A} = \frac{score(r_2)}{2+1} + \frac{score(r_3)}{1+1} = \frac{12}{3} + \frac{6}{2} = 7$. So, $e_{A,H}$, e_{H,f_1}, and ef_1, D are the edges having top2 scores. Mapped back to the original graph, they become segments $\overline{AH}$, $\overline{Hf_1}$, and $\overline{f_1D}$. AUG finds that $\overline{Hf_1}$ and $\overline{f_1D}$ are adjacent to each other and have the same score. Therefore, these two segments are merged, resulting in $\overline{HD}$. In the end, $\overline{AH}$ and $\overline{HD}$ are returned as result.

5.6.2 ITE-topK

Recall that the ITE algorithm prunes sub-optimal segments early in order to achieve a good performance. However, in ITE-topK, we have to avoid pruning those segments (early-pruning) that do not have the optimal score but are among the topK segments. It is a challenge to identify them during the runtime of ITE. Recall that in ITE, the score lower bounds are used to prune sub-optimal segments (See lemma 5.5.2). To solve the early-pruning problem, we delay updating the maximum score lower bound seen so far ($maxLb$). In particular, we use the set S_k to keep candidate segments with the topK score seen so far. The $maxLb$ is only updated when there are already k segments in S_k and there is an update in the membership.

Another problem arises when S_k is maintained. Many segments in S_k may be subsegments of the same segment, resulting in fewer than k segments being found after finding the entire segment for each segment in S_k.

The following lemma helps solve the problem.

Lemma 5.6.1 *Given two segments $s_i \in C$ and $s_j \in C$, if $s_i.C \neq s_j.C$, s_i and s_j are subsegments of two different segments.*

Proof 5.6.1 *We prove this lemma by contradiction. Suppose, on the contrary, s_i and s_j are subsegments of the same segment s. Since s is a consistent segment, we have $score(s_i) = scores_j = scores$. Then we have $s_i.C = s_j.C = s.C$, which contradicts to $s_i.C \neq s_j.C$. So we conclude that s_i and s_j are subsegments of two different segments.*

The ITE-topK algorithm is presented in Algorithm 18 and explained next. Same as ITE, ITE-topK also uses a priority queue Q that is sorted on the upper bound score of every segment. A variable called $maxLb$ is used to keep track of the maximum lower bound score seen so far. S_k denotes the result set, and its function S_k.size() computes the number of distinct scores in S_k.

First, ITE-topK initializes the edges such that each has a lower bound score 0 and an upper bound score computed as in Definition 7 (line 1). It also initializes the result set S_k, enqueues the edges $G.E$ of the road network graph, and initializes variable $maxLb$ (line 2). It then enters the loop and pops out the top element from Q (lines 3–4). The flag variable $split$, indicating whether or not the current segment needs to be partitioned, is set to false at the beginning of each iteration (line 5). Next, if the upper bound score of $currSeg$ is no less than $maxLb$ then there are two cases (line 6). If $currSeg.ub \neq currSeg.lb$, $currSeg$ is not a consistent segment yet and needs to be further split. So $split$ is set to true (lines 7–8). Otherwise, if $currSeg.ub = currSeg.lb$ and no other candidates in S_k have the same set of covering routes (line 9), ITE-topK considers adding $currSeg$ into S_k. First, if S_k.size() is less than k then S_k adds $currSeg$ (lines 10–11). $maxLb$ is updated if S_k.size() now becomes k (lines 12–13). Second, if S_k.size() is k then S_k adds $currSeg$ if $currSeg.ub = maxLb$ (lines 14–16). If $currSeg.ub > maxLb$, however, the segments with smaller scores are removed from S_k, S_k adds $currSeg$ and updates $maxLb$ (lines 17–20).

If $split$ is true, the function partitions $currSeg$ into subsegments with the procedure $\mathrm{SplitSegment}$, which partitions a segment G into β equal length subsegments (lines 21–22). Similar to ITE, β is a tunable parameter.

The intersection set, contain set, and lower and upper bound scores for each segment output by $\mathrm{SplitSegment}$ are computed and inserted into Q (lines 23–31). .

Upon exiting the loop, each optimal segment is extended to its full length by overlapping the routes that contribute scores to the segment (lines 32–33).

With the example in Figure 5.1, the execution of ITE-topK is quite similar to what is demonstrated in Section 5.5. The difference exists in maintaining the result set S_k. In ITE-topK, candidates with k different scores are kept in the result set instead of just one as in ITE.

Algorithm 18: ITE-topK($G, R, F, \delta, \beta, M, k$)

```
1  Init. $e \in G.E$ s.t. $e.lb \leftarrow 0$ and $e.ub \leftarrow \sum_{r_i \in e.R_c} w_M(j_i, k_i) score(r_i)$;
2  $S_k \leftarrow \emptyset$; $Q.enqueue(G.E)$; $maxLb \leftarrow 0$;
3  while $Q \neq \emptyset$ do
4      $currSeg \leftarrow Q.dequeue$;
5      $split \leftarrow false$;
6      if $currSeg.ub \geq maxLb$ then
7          if $currSeg.ub \neq currSeg.lb$ then
8              $split \leftarrow true$;
9          else if $\nexists s_i \in S_k, s_i.C = currSeg.C$ then
10             if $S_k$.size() $< k$ then
11                 $S_k$.add($currSeg$);
12                 if $S_k$.size() $= k$ then
13                     $maxLb \leftarrow \min_{s_i \in S_k}$ score($s_i$);
14             else if $S_k$.size() $= k$ then
15                 if $currSeg.ub = maxLb$ then
16                     $S_k$.add($currSeg$);
17                 else if $currSeg.ub > maxLb$ then
18                     remove $\forall s_i \in S_k$, $s.lb = maxLb$;
19                     $S_k$.add($currSeg$);
20                     $maxLb \leftarrow \min_{s_i \in S_k}$ score($s_i$);
21     if $split$ then
22         $ss \leftarrow$ SplitSegment($currSeg, \beta$);
23     foreach $s \in ss$ do
24         foreach $r \in currSeg.I$ do
25             if intersects($r, s$) then
26                 $s.I \leftarrow s.I$.add($r$);
27                 $s.ub \leftarrow s.ub + w_M(j, k) score(r)$;
28             if contains($r, s$) then
29                 $s.C \leftarrow e.C$.add($r$);
30                 $s.lb \leftarrow s.lb + w_M(j, k) score(r)$;
31             $Q.enqueue(s)$;
32 foreach $s \in S$ do
33     Find the entire optimal segment of $s$ by overlapping the route usage objects
       $r \in s.C$ one by one.
```

5.6.3 Theoretical Analysis

We analyze the correctness, completeness, and complexity of ITE-topK.

The correctness of ITE-topK depends on finding the subsegments with the maximum score correctly. Here, we prove that the algorithm terminates and returns subsegments that have the topK scores. We must show that after a finite number of iterations, ITE-topK produces a subsegment s such that $s.ub = s.lb$ where $s.ub$ is the topK scores. First, we know that when $s.ub = s.lb$, s is a consistent segment with the score $s.ub$ (or $s.lb$). Since $s.ub$ is larger than $\mathrm{score}_\mathrm{l}(s)$, s is a subsegment with the topK scores. Since ITE-topK always splits subsegment s with the $s.ub$ greater than $\mathrm{score}_\mathrm{l}(C)$, we only need to show that ITE-topK terminates. This can be shown by the following properties. (1) The maximum ub value decreases, (2) The maximum lb increases, and (3) The maximum ub and lb values converge to the same value after a number of iterations.

Next, to prove completeness, we show that for each optimal segment, ITE-topK is able to find a subsegment of it. Let s_i and s_j be two subsegments of distinct optimal segments. Without loss of generality, suppose ITE-topK has found s_i and $s_i.lb = mathrmscore_l(C)$. We show that ITE-topK also finds s_j instead of pruning it. Recall that ITE-topK uses two pruning criteria to prune a subsegment. The first criterion says that s_j can be pruned if $s_i.lb > s_j.ub$. Since both s_i and s_j are subsegments of topK segments, we have $s_j.ub \geq mathrmscore_h(C) \geq s_i.lb$. Therefore, this pruning criterion does not apply. The second criterion states that s_j can be pruned if $s_j.I \subseteq s_i.C$. Since s_i and s_j are subsegments of different topK segments, $s_j.C \nsubseteq s_i.C$. We also know that s_j is a consistent subsegment, hence $s_j.I = s_j.C$. Putting them together, we have $s_j.I = s_j.C \nsubseteq s_i.C$. Hence the second pruning criterion also does not apply. Thus, ITE-topK does not prune s_j, but detects it. Therefore, ITE-topK finds all the optimal segments.

Theorem 5.6.1 *The time complexity of ITE-topK is $O((klog|R| + |S|)|R|)$.*

Proof 5.6.2 *In the priority queue operations, ITE-topK iteratively splits the segment with the maximal score upper bound. The number of splits corresponds to the height of the tree with fan-out β. If $\beta = 4$, we get a quadtree. According to [29], the asymptotic height of the quadtree is $log|R|$, For each subsegment s, ITE-topK uses a loop to find its intersection set $s.I$ and contain set $s.C$. We have $ss.I \subseteq R$, so the time complexity of the loop is $O(|R|)$. ITE-topK maintains k best segments seen so far. So the total cost of the while loop is $O(k|R|log|R|)$.*

The second loop depends on the size of the result set S. Since $s.C \subseteq R$, the time complexity of this loop is $O(|S||R|)$.

In total, the time complexity of ITE-topK is $O((klog|R| + |S|)|R|)$.

5.7 Experimental Study

This section reports on empirical studies that aim to elicit design properties of the proposed framework and, in particular, of the AUG and ITE algorithms. The studies use a real spatial network and real facility and trajectory data, as well as synthetic data.

The experiments covered in this section were performed on an Intel Xeon (2.66Ghz) quad-core machine with 8 GB of main memory running Linux (kernel version 2.6.18). Both of the algorithms were implemented in Java. Every instantiation of JVM was allocated 2 GB of virtual memory. We first describe the data used in the experiment as well as the parameter settings. Then we cover experiments that target different aspects of the algorithms.

5.7.1 Data Sets and Parameter Settings

5.7.1.1 Road Network

The digital road network TOP10DK [1] was used for our experiments. It contains all of Denmark at a fine granularity. To construct the road network graph, we first identify the

[1]http://tinyurl.com/bqtgh2g

vertices. An edge exists between two vertices v_1 and v_2 as long as there exists a road segment connecting v_1 and v_2. In total, the graph contains 465,057 vertices and 920,218 edges.

In order to study the performance of the algorithms thoroughly, we used a real-world data set and a synthetic data set. Each data set contains a collection of routes (GPS recordings received from drivers) and a collection of facilities. Both data sets share the same underlying road network.

5.7.1.2 Route Data Preparation

We obtained the real route traversal data set from the "Pay as You Speed" project [53] [2]. The data set is obtained from vehicles driving in North Jutland, Denmark. The data set contains 39,688,695 GPS points produced by 151 different drivers in the period from October 1, 2007 to January 31, 2008. In this data set, each route is represented by a sequence of GPS points that may deviate from the underlying road network. To solve this problem, we use an existing technique by Tradišauskas et al. [79] to map-match the route data onto the underlying road network. Then the sequence of traversed edges has also to be determined because two consecutive GPS points may be matched to different edges. To achieve it, we use a bidirectional Dijkstra's algorithm provided by Pohl [72].

In addition, stationary points, when reported GPS locations are the same for consecutive time points for the same user, are removed. Further, different trips of users were identified from the set of GPS recordings. We distinguish a new route when the time period between two consecutive GPS points is more than 3 minutes. In total, we obtain 51,146 routes. The median number of GPS points of the routes is 488. The median length of the routes in the real data sets is 6524.81.

We generate synthetic routes by simulating the movement of a vehicle that emits GPS points with a fixed frequency (e.g., 0.1 Hz). The length of the each route is thus the speed of the car times the number of GPS points it emits. In the simulation, routes are allowed to

[2] http://www.trafikdage.dk/td/papers/papers07/tdpaper27.pdf

have variable lengths. So when starting a new synthetic route, we first generate a random number between 480 and 520 for the number of GPS points. We use 480 and 520 because the median number of GPS points of the routes in the real data set is 488. Then we randomly select a network point to start a new route. When taking the next point, we follow the graph and traverse to the next edge (randomly pick one if more than one outgoing edge exists). The sampling frequency is fixed for one data set to simulate a real life application. We then vary the sampling frequency, resulting in three different data sets, i.e., short, medium, and long, with the median lengths of routes being 3405.76, 8030.42, and 12890.12, respectively.

In both the real and synthetic data sets, for each route, we use a random number generator to generate the user count and route usage randomly from 1 to 20.

5.7.1.3 Facilities

The facility data set contains 16,577 places of interest located throughout Denmark. The exact address of each facility can be looked up from yellow pages. Since it is meaningless to take businesses of different types, we group the facilities according to their types (e.g., fast food, salon, supermarket). In all the experiments below, the facilities are of the same type. When generating the synthetic facility data set, we randomly pick network points from the network. Every facility in either the real data set or the synthetic attracts at least one route.

Statistics on the data sets and the settings for key parameters are summarized in Table 5.3. The default values are in bold.

5.7.1.4 Scoring Function and Score Distribution Model

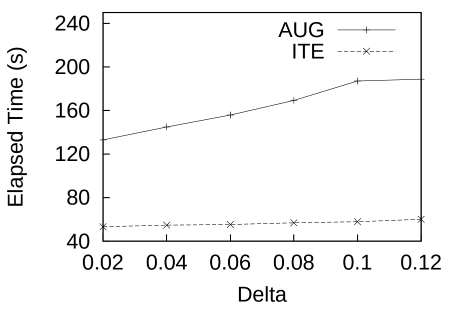
We observe from the experiments that the scoring function and the score distribution model do not affect the performance of the two algorithms. Therefore, we show the experimental results produced when using the first scoring function and the first proposed model here. Experimental results for using the other scoring function and interest models are also shown.

Parameter	**Range**
δ	0.02, 0.04, **0.06**, ..., 0.12
β	2, 3, **4**, 5, 6
Num Routes in Real Data	5k, 10k, ..., **25k**
Num Routes in Synthetic Data	10k, 15k, ..., **30k**
Num Facilities	600, 800, **1k**, ..., 1.4k

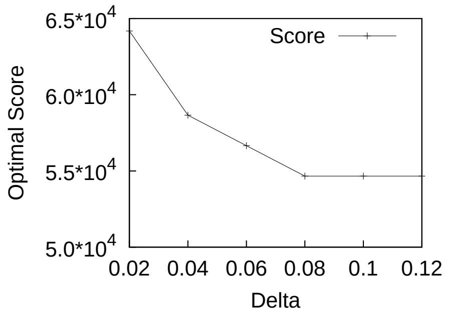
Table 5.3: Experimental Settings

5.7.2 Effect of δ

Recall that a facility attracts a route if their distance is no further than δ. Figure 5.5 shows the performance and optimal scores when varying δ on real data.



(a) Performance vs δ, Real



(b) Score vs δ, Real

Figure 5.5: Effect of δ

Although the running times for both algorithms increase when δ increases (Figure 5.5(a)), the two algorithms exhibit different patterns. When δ increases from 0.02 to 0.1, AUG increases much faster than ITE. AUG has to explore further on the edges to find the attracting facilities for each route traversal, in order to decide whether to include them in the augmented graph. This may be the reason why AUG increases more rapidly than ITE. When δ increases from 0.1 to 0.12, the running time of AUG increases slower. The reason may be that less facilities are taken into account. In reality, some facilities prefer locations near the junctions, so the density of facilities in the middle of roads might be less. The increase of the running time of ITE is less, and there is no sudden change, indicating that δ has little effect on ITE.

Since the optimal scores output by both algorithms are the same, we plot one figure to show the effect of δ (Figure 5.5(b)). The optimal scores decrease like a staircase. The reason is that increasing δ may increase the number of attracting facilities for a route, resulting in decreased scores of segments received from the routes according to the score distribution model.

5.7.3 Effect of β

Recall that β is the number of subsegments produced when a segment is partitioned. It is a user-specified parameter. Figure 5.6 shows the effect of β on the running time of ITE.

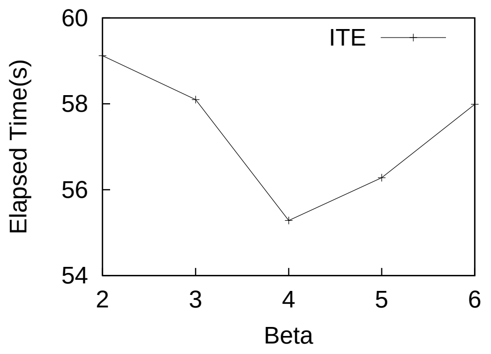
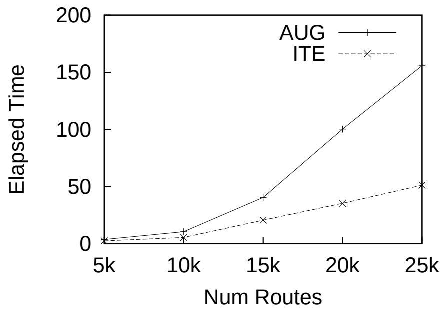


Figure 5.6: Effect of β, Real

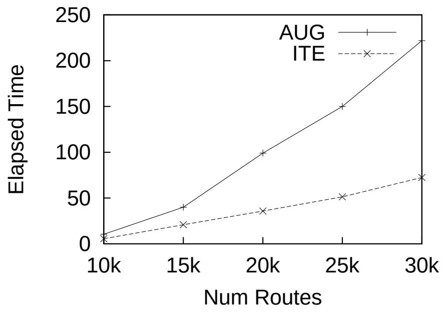
Initially, as β value increases, the running time decreases. However, beyond a certain β value (4 in the figure), with further increase in the next value, the running time starts to increase. The best performance of ITE occurs when $\beta = 4$. When the β value is smaller than 4, the “zooming-into” an optimal subsegment may not be as fast as when $\beta = 4$. On the other hand, when the β value is greater than 4, computing the lower and upper bounds of the subsegments can take substantial time, and thus the increase in running time.

5.7.4 Effect of the Number of Routes

Figure 5.7 shows the performance when varying the number of routes using real and synthetic data. Algorithms AUG and ITE perform equally well for a small number (5k) of routes. But the running time of AUG grows much more rapidly than that of ITE with the increase of the quantity of routes. This is expected from the time complexity analysis of AUG and ITE.



(a) Performance vs. Num Routes, Real



(b) Performance vs. Num Routes, Synthetic

Figure 5.7: Effect of the Number of Routes on Performance

5.7.5 Effect of Route Length

In this set of experiments, we study the effect of the length of routes on the performances of both algorithms. The three data sets used in the experiments are explained above. Figure 5.8 shows the results. For both algorithms, more time is needed for longer routes when the number of route traversals ranges from 5k to 25k. Again, the running time of AUG increases faster than that of ITE.

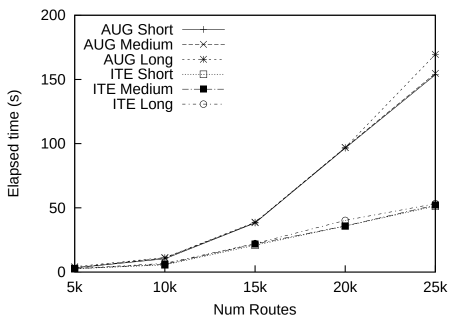
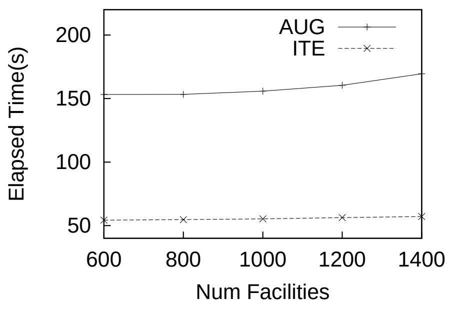


Figure 5.8: Effect of Route Length

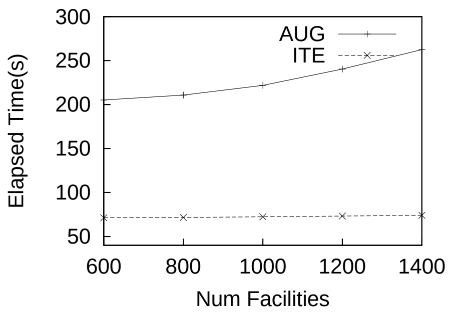
For AUG, computing the Attraction List for the vertices takes longer time as each route covers more vertices on average. For ITE, each edge intersects more routes on average. So after splitting a segment, more routes have to be examined to calculate the lower and upper bound scores of the subsegments, resulting in longer running time.

5.7.6 Effect of the Number of Facilities

Figure 5.9 shows the running time of AUG and ITE when varying the number of facilities.



(a) Performance vs. Num Facilities, Real



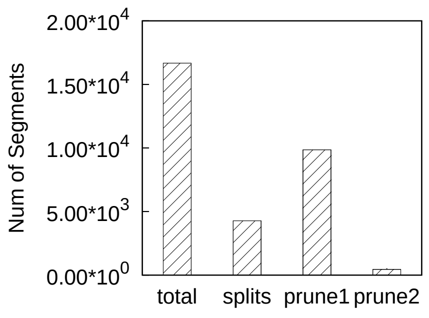
(b) Performance vs. Num Facilities, Synthetic

Figure 5.9: Effect of the Number of Facilities

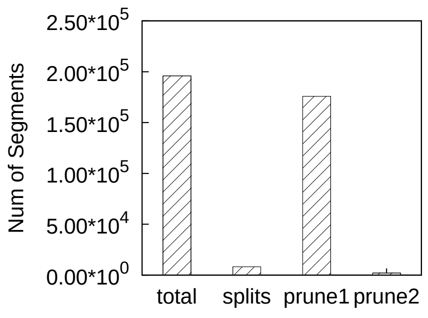
For both real and synthetic data sets, it can be seen that AUG is affected a little more by the increased number of facilities. The reason is that in AUG, facilities have to be augmented, and then the attraction lists have to be calculated for them, resulting in much computational overhead. In contrast, there is no need to augment facilities in ITE. When ITE partitions the segments, no house-keeping work is necessary for facilities. It just needs to take care of the relative positions of the newly produced segments to the facilities.

5.7.7 Effectiveness of Pruning Strategies

In this set of experiments, we study the effectiveness of the pruning strategies in ITE by keeping track of the number of segments generated, partitioned, and pruned in the course of finding the optimal segments. Figure 5.10 shows the respective segments generated, split, and pruned by Lemma 5.5.2 and Lemma 5.5.3 when running ITE with default settings. Label "total" means the total number of generated subsegments, "splits" is the number of subsegments that needs further splitting, "prune1" is the number of subsegments that are pruned using lemma 5.5.2, and "prune2" is the number of subsegments that are pruned using lemma 5.5.3.



(a) Real



(b) Synthetic

Figure 5.10: Effect of Pruning Strategies

It is observed that in the real data set the number of segments that require further splitting is 4,273, which is approximately 25% of the number of total segments, whereas the number is between 5% and 10% in the synthetic data set.

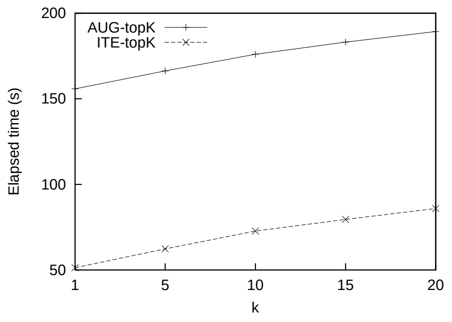
In the real data set, almost 60% of the generated segments that cannot contain an optimal segment are been pruned by Lemma 5.5.2. In contrast, Lemma 5.5.3 prunes 2.7% of the total segments.

In the synthetic data set where route traversals are generated more evenly throughout the entire map, Lemma 5.5.2 prunes almost 10% of the total generated segments. Lemma 5.5.3 prunes around 1% of the total segments.

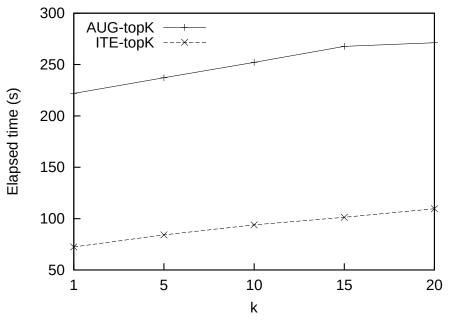
5.7.8 AUG-topK and ITE-topK

Here, we compare the performances of AUG-topK and ITE-topK by varying k. The experiment results are shown in Figure 5.11.

For both data sets, we observe the running times of both AUG-topK and ITE-topK increase with k. In addition, for both data sets, ITE-topK outperforms AUG-topK by a wide margin, thanks to the pruning techniques that greatly reduce the search space in ITE-topK. The difference is even bigger in the synthetic data set where the number of routes is greater and the chance for routes to overlap with each other is bigger.



(a) Real

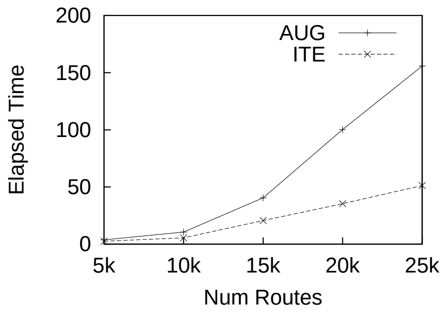


(b) Synthetic

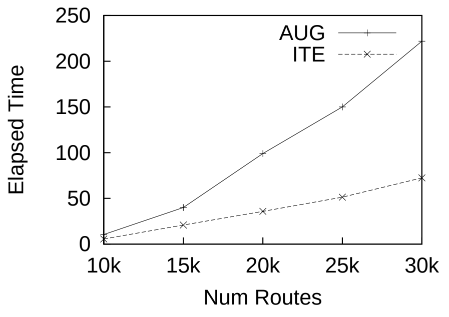
Figure 5.11: Effect of k

5.7.9 Effect of Scoring Functions

We show experimental results of AUG and ITE with the scoring function $score_{cap-x}(r)$. The results are shown in Figure 5.12



(a) Performance vs. Num Routes, Real, Scoring



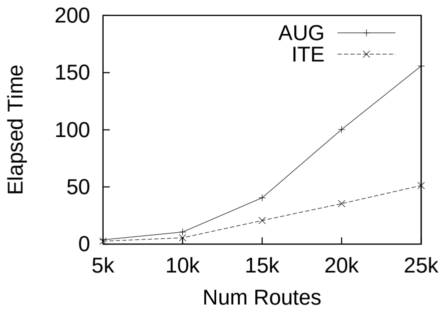
(b) Performance vs. Num Routes, Synthetic, Model

Figure 5.12: Effect of the Number of Routes on Performance

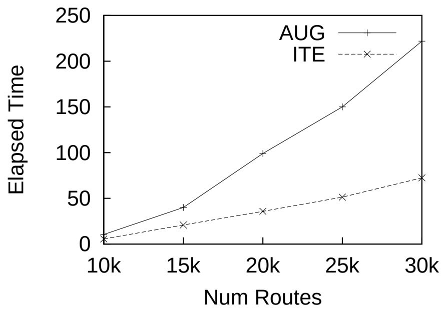
It is verified with ITE consistently outperforms AUG with different scoring functions. Comparing these two figures with Figure 5.13(a) and Figure 5.13(b), it is also verified that scoring functions have very little effect on the performances.

5.7.10 Effect of Interest Models

We also show experimental results for the second interest model. The results are shown in Figure 5.13



(a) Performance vs. Num Routes, Real, Model



(b) Performance vs. Num Routes, Synthetic

Figure 5.13: Effect of the Number of Routes on Performance

We also observe that the running time is almost the same with different interest models. The result is reasonable because to both AUG and ITE, absolute values of the scores of the segments or edges do not have any meanings. Instead, only their relative values to each other are meaningful. Different scoring functions and interest models only affect the absolute value of the score of each segment or edge. Therefore, they have little effect on the running times of the experiments.

5.8 Summary

The chapter formalizes a modern version of the classical facility location problem that takes into account the availability of customer trajectory data that is constrained to a road network, rather than simply assuming the availability of static customer locations. In the resulting framework, route traversals by customers rather than customer locations are attracted by facilities. The framework enables a wide variety of choices for assigning scores to the routes traversed by customers and for distributing these scores to segments in the underlying road network, thus offering flexibility that aims to enable applications with different types of facilities. We believe that this is the first work to explore this natural generalization of the classical facility location problem.

Two algorithms, AUG and ITE, are provided that solve this generalized problem. AUG takes a graph augmentation approach, ITE iteratively partitions road segments into smaller pieces (subsegments), and uses a scoring mechanism to guide the selection of promising segments for further partitioning. We also extend AUG and ITE to find topK optimal segments, in case that users might want to take other factors into account. We report experimental results with both real and synthetic data that demonstrate the efficiency of our proposed algorithms. For the data sets considered, ITE outperforms AUG and ITE-topK outperforms AUG-topK, thanks to the pruning techniques employed by ITE to reduce the search space quickly.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

With the proliferation of GPS-enabled devices and the accumulation of GPS data, it is challenging to devise efficient algorithms to process large-scale, real-time location-based queries, and devise sophisticated algorithms to extract information and knowledge from trajectory data. This thesis addresses these challenging tasks by working on three types of queries, the Moving Continuous Query (MCQ), the group query, and the optimal segment query.

A location-based query is called a Moving Continuous Query (MCQ) if both the query issuer and query result are mobile clients, and the query issuer has to continuously know the result. We propose a distributed client-server architecture for efficiently processing such queries. In our solution, a cluster of interconnected servers takes care of the global view, while part of the computation is relegated to the clients to verify actual results. In particular, we consider processing two most important queries in spatial-temporal database, range query and kNN query. We demonstrate the effectiveness and efficiency of our approach through extensive simulation experiments. Our method reduces both the server-side workload and the client-server communication cost in comparison to the most recent state-of-the-art scheme, while it is much more scalable to growing number of moving clients.

The notion of *trajectory group* enables the prioritized discovery of interesting moving object clusters from trajectories that are sampled differently. Techniques based on the continuous clustering of moving objects using an effective pruning strategy are proposed to efficiently discover such groups. A scoring function enables the ranking of the discovered groups according to their size and duration. Unlike existing techniques, our approaches offer sampling independence, integrate trajectory simplification seamlessly, and function in online settings. The effectiveness and efficiency of our schemes are studied using real data sets.

We study the problem of finding optimal locations to setup new facilities for a more general form of customer data, the routes, than static customer points. To the best of our knowledge, this is the first work which exploits the movement history of customers instead of static customer points in the line of research. This knowledge allows us to find better locations. Two interest models are presented to reflect various types of businesses. Two algorithms, AUG and ITE, have been designed to solve this generalized problem with these interest models. AUG takes a graph augmentation approach, whereas ITE iteratively partitions the road segments into smaller pieces (subsegments), and uses a scoring mechanism to guide the selection of promising subsegments for further partitioning. We have also extended AUG and ITE to find topK segments for users who may wish to take other factors into consideration. Extensive experiments with a real and a synthetic data set were conducted. The results have demonstrated the effectiveness and efficiency of our proposed algorithms.

6.2 Future Work

There are several directions that we would like to work on in the future.

For Chapter 3, the proposed architecture for processing MCQ queries is actually very generic with respect to. queries. We are interested in adopting it for more spatial queries (e.g., Continuous Reverse kNN query) and in comparing its performance with other schemes.

In Chapter 4, in the group discovery framework, in order to improve its efficiency, we need to smoothen the trajectories and thus reduce the number of possible events. In this thesis, we have explored the possibility of applying online trajectory simplification technique. Future work can also employ shared prediction-based tracking [21, 78] when collecting locations points. In addition, tries are used widely due to their simplicity and efficiency. Although the computational overhead is substantial, using a trie for storing and representing clusters is promising, and future work that aims to reduce the computational overhead is in order. Another direction is to relax the distance constraint in order to allow a member to exit the cluster for a short while and then join back. Relaxing distance constrain is expected to have two effects. Some previously un-discovered groups may be found by relaxing distance constraint. Some groups that are already found may also have higher scores because of the longer duration, and thus become more important.

In Chapter 5, interesting directions also exist for the optimal segment query. We show two of them. First, the optimal segments can be incrementally evaluated when new routes are available. Incremental evaluation allows more flexibility when several sets of routes are continuously added and may help improve the performance. Second, future work may consider finding top$k$ segments. It may be both valuable and interesting to decision makers.

With the rise of online social networking, it is interesting to support location-aware applications in online social networks with real-time location information and/or trajectories. One direction is to target more sophiscated path queries. Given a number of places of interest and time/financial constraints, it is useful to propose possible paths to visit them, especially when these places are different cities. Symmetrically, after the user selects her favorite path, it is useful to propose places of interest along the path to make the journey more fun. Taking it one step further. It can be attractive to a traveler to find friends who share a common (partial) traveling path, so that they can travel together. In order to support such applications, efficient algorithms operating on real or hypothetical itinerary are required.

Another possible direction is to support applications in the so-called “Smart City” project. It is possible look at past traffic conditions and other supplementary information (e.g., working day/holiday, weather condition, and calendar etc.) to predict today's traffic condition and to provide transportation suggestions to the the user, in order to have a smoother journey in a city.

Many works in this area, including this thesis, do not take into account of user location privacy. However, in many real-life applications, location privacy is an important issue, especially when these applications target people. We believe that a good location privacy protection mechanism should be employed in many applications. In the future, we would work on proposing location privacy protection techniques and investigate how the location privacy-protected data may affect existing schemes.

References

[1] Porcupine caribou herd satellite collar project.

[2] Ghazi Al-Naymat, Sanjay Chawla, and Joachim Gudmundsson. Dimensionality reduction for long duration and complex spatio-temporal queries. In *SAC*, pages 393–397, 2007.

[3] Arnon Amir, Alon Efrat, Jussi Myllymaki, Lingeshwaran Palaniappan, and Kevin Wampler. Buddy tracking - efficient proximity detection among mobile friends. *Pervasive and Mobile Computing*, 3(5):489–511, 2007.

[4] Aris Anagnostopoulos, Russell Bent, Eli Upfal, and Pascal Van Hentenryck. A simple and deterministic competitive algorithm for online facility location. *Information and Computation*, 194(2):175 – 202, 2004.

[5] Htoo Htet Aung and Kian-Lee Tan. Discovery of evolving convoys. In *SSDBM*, SSDBM'10, pages 196–213, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] Htoo Htet Aung and Kian-Lee Tan. Finding closed memos. In *SSDBM*, pages 369–386, 2011.

[7] Rimantas Benetis, Christian S. Jensen, Gytis Karčiauskas, and Simonas Šaltenis. Nearest and reverse nearest neighbor queries for moving objects. *The VLDB Journal*, 15(3):229–249, 2006.

[8] Marc Benkert, Joachim Gudmundsson, Florian Hübner, and Thomas Wolle. Reporting flock patterns. *Comput. Geom. Theory Appl.*, 41:111–125, November 2008.

[9] Guy E. Blelloch and Kanat Tangwongsan. Parallel approximation algorithms for facility-location problems. In *SPAA*, pages 315–324, 2010.

[10] Lee Breslau, Ilias Diakonikolas, Nick G. Duffield, Yu Gu, Mohammad Taghi Hajiaghayi, David S. Johnson, Howard J. Karloff, Mauricio G. C. Resende, and Subhabrata Sen. Disjoint-path facility location: Theory and practice. In *ALENEX*, pages 60–74, 2011.

[11] Thomas Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[12] Stefan Bttcher, Charles L. A. Clarke, and Gordon V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines.* MIT Press, Cambridge, Mass., 2010.

[13] Sergio Cabello, José Miguel Díaz-Báñez, Stefan Langerman, Carlos Seara, and Inmaculada Ventura. Reverse facility location problems. In *CCCG*, pages 68–71, 2005.

[14] Sergio Cabello, José Miguel Díaz-Báñez, Stefan Langerman, Carlos Seara, and Inmaculada Ventura. Facility location problems in the plane based on reverse nearest neighbor queries. *European Journal of Operational Research*, 202(1):99 – 106, 2010.

[15] J. Cardinal and S. Langerman. Min-max-min geometric facility location problems. *22nd European Workshop on Computational Geometry (EWCG)*, 2006.

[16] Lei Chen and Raymond Ng. On the marriage of lp-norms and edit distance. In *VLDB*, pages 792–803, 2004.

[17] Lei Chen, M. Tamer Özsu, and Vincent Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA, 2005. ACM.

[18] Su Chen, Beng Chin Ooi, Kian-Lee Tan, and Mario A. Nascimento. St2b-tree: a self-tunable spatio-temporal b+-tree index for moving objects. In *SIGMOD*, pages 29–42, 2008.

[19] Su Chen, Beng Chin Ooi, and Zhenjie Zhang. An adaptive updating protocol for reducing moving object database workload. *PVLDB*, 3(1-2):735–746, 2010.

[20] Chi-Yin Chow, Mohamed F. Mokbel, and Hong Va Leong. On efficient and scalable support of continuous queries in mobile peer-to-peer environments. *IEEE Transactions on Mobile Computing*, 10(10):1473–1487, October 2011.

[21] Alminas Civilis, Christian S. Jensen, and Stardas Pakalnis. Techniques for efficient road-network-based tracking of moving objects. *TKDE*, 17:698–712, May 2005.

[22] D. Douglas and T. Peucker. Algorithms for the reduction of the number of points required to represent a line or its character. *The American Cartographer*, 10(42):112–122, 1973.

[23] Yang Du, Donghui Zhang, and Tian Xia. The optimal-location query. In *SSTD*, pages 163–180, 2005.

[24] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, pages 226–231, 1996.

[25] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[26] Reza Zanjirani Farahani and Masoud Hekmatfar. *Facility Location: Concepts, Models, Algorithms and Case Studies*. Contributions to Management Science. Physica-Verlag, 2009.

[27] Tobias Farrell, Kurt Rothermel, and Reynold Cheng. Processing continuous range queries with spatiotemporal tolerance. *IEEE Trans. on Mobile Computing*, 10(3):320–334, 2011.

[28] Hakan Ferhatosmanoglu, Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. Constrained nearest neighbor queries. In *SSTD*, SSTD '01, pages 257–278, London, UK, UK, 2001. Springer-Verlag.

[29] Philippe Flajolet, Gaston H. Gonnet, Claude Puech, and J. M. Robson. Analytic Variations on Quadtrees. *Algorithmica*, 10(6):473–500, 1993.

[30] Dimitris Fotakis. Incremental algorithms for facility location and k-median. *Theor. Comput. Sci.*, 361(2-3):275–313, 2006.

[31] Dimitris Fotakis. On the competitive ratio for online facility location. *Algorithmica*, 50:1–57, 2008.

[32] Andrew U. Frank, Jonathan Raper, Frank Andrew, Jonathan Raper, and J. P. Cheylan, editors. *Life and Motion of Socio-Economic Units*.

[33] Bugra Gedik and Ling Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.

[34] Joachim Gudmundsson and Marc van Kreveld. Computing longest duration flocks in trajectory data. In *GIS*, pages 35–42, 2006.

[35] Joachim Gudmundsson, Marc van Kreveld, and Bettina Speckmann. Efficient detection of motion patterns in spatio-temporal data sets. In *GIS*, pages 250–257, 2004.

[36] Oliver Günther. Efficient computation of spatial joins. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 50–59, 1993.

[37] Ralf Hartmut Guting and Markus Schneider. *Moving Objects Databases*. Morgan Kaufmann Publishers, 2005.

[38] Antomn Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, volume 14(2), pages 47–57, 1984.

[39] Yan Huang, Cai Chen, and Pinliang Dong. Modeling herds and their evolvements from trajectory data. In *GIScience*, pages 90–105, Berlin, Heidelberg, 2008. Springer-Verlag.

[40] Glenn S. Iwerks, Hanan Samet, and Kenneth P. Smith. Maintenance of $k$nn and spatial join queries on continuously moving points. *ACM Trans. Database Syst.*, 31(2):485–536, 2006.

[41] Kamal Jain and Vijay V. Vazirani. Approximation algorithms for metric facility location and k-median problems using the primal-dual schema and lagrangian relaxation. *J. ACM*, 48(2):274–296, March 2001.

[42] C. S. Jensen, Lahrmannm H., S. Pakalnis, and J. Runge. The INFATI data. *Aalborg University, TimeCenter TR-79*, 2004.

[43] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and update efficient b+-tree based indexing of moving objects. In *VLDB*, VLDB '04, pages 768–779. VLDB Endowment, 2004.

[44] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Continuous clustering of moving objects. *IEEE TKDE*, 19:1161–1174, Sep. 2007.

[45] Christian S. Jensen and Stardas Pakalnis. Trax: real-world tracking of moving objects. In *VLDB*, pages 1362–1365, 2007.

[46] Christian S. Jensen, Dalia Tie!sytye, and Nerius Tradi!sauskas. Robust b+-tree-based indexing of moving objects. In *MDM*, page 12, 2006.

[47] Hoyoung Jeung, Man Lung Yiu, Xiaofang Zhou, Christian S. Jensen, and Heng Tao Shen. Discovery of convoys in trajectory databases. *PVLDB*, 1(1):1068–1080, 2008.

[48] H. Jia, F. Ordez, and M. M. Dessouky. A modeling framework for facility location of medical services for large-scale emergencies. *Special Issue of IIE Transactions on Homeland Security*, 39:41–55, 2007.

[49] H. Jia, F. Ordez, and M. M. Dessouky. Solution approaches for facility location of medical supplies for large-scale emergencies. *Computers and Industrial Engineering*, 52:257–276, 2007.

[50] Panos Kalnis, Nikos Mamoulis, and Spiridon Bakiras. On discovering moving clusters in spatio-temporal data. In *SSTD*, pages 364–381, 2005.

[51] Samir Khuller, Y. J Sussmann, Randeep Bhatia, and Sudipto Guha. Facility location with dynamic distance functions. *Journal of Combinatorial Optimization*, 2:199 – 217, 1998.

[52] Flip Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD*, pages 201–212, 2000.

[53] H. Lahrmann, N. Agerholm, N. Tradisauskas, and J. Juhl. Spar paa farten - an intelligent speed adaptation project in denmark based on pay as you drive principles. *Proceedings of the 6th European Congress on Intelligent Transport Systems and Services*, 2007.

[54] Patrick Laube and Stephan Imfeld. Analyzing relative motion within groups of trackable moving point objects. In *GIS*, pages 132–144, 2002.

[55] Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. Trajectory clustering: a partition-and-group framework. In *SIGMOD*, pages 593–604, 2007.

[56] Ken C. K. Lee, Wang-Chien Lee, and Hong Va Leong. Nearest surrounder queries. In *ICDE*, page 85, 2006.

[57] Ken C. K. Lee, Wang-Chien Lee, Va Hong Leong, Brandon Unger, and Baihua Zheng. Efficient valid scope computation for location-dependent spatial queries in mobile and wireless environments. In *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*, ICUIMC '09, pages 131–140, New York, NY, USA, 2009. ACM.

[58] Ken C. K. Lee, Josh Schiffman, Josh Zheng, Wang chien Lee, and Wang chien Va Leong. Tracking nearest surrounders in moving object environments, 2006.

[59] Xiaohui Li, Vaida Ceikute, Christian S. Jensen, and Kian Lee Tan. Effective online group discovery in trajectory databases. *IEEE Transactions on Knowledge and Data Engineering*, 2012.

[60] Xiaohui Li, Panagiotis Karras, Lei Shi, Kian Lee Tan, and Christian S. Jensen. Cooperative scalable moving continuous query processing. In *IEEE 13th International Conference on Mobile Data Management*, pages 69 – 78, 2012.

[61] Zhenhui Li, Bolin Ding, Jiawei Han, and Roland Kays. Swarm: Mining relaxed temporal moving object clusters. *PVLDB*, 3(1):723–734, 2010.

[62] Mohammad Mahdian, Yinyu Ye, and Jiawei Zhang. Improved approximation algorithms for metric facility location problems. In *In Proceedings of the 5th International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 229–242, 2002.

[63] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[64] Nirvana Meratnia and Rolf A. de By. Spatiotemporal compression techniques for moving point objects. In *EDBT*, pages 765–782, 2004.

[65] Adam Meyerson. Online facility location. In *FOCS*, pages 426–431, 2001.

[66] Mohamed F. Mokbel, Xiaopeng Xiong, and Walid G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, 2004.

[67] Kyriakos Mouratidis, Dimitris Papadias, and Marios Hadjieleftheriou. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, pages 634–645, 2005.

[68] R. T. Ng. *Detecting outliers from large datasets*. Taylor & Francis, 2001.

[69] Stefan Nickel and Justo Puerto. *Location theory: a unified approach*. Springer, 2005.

[70] Dimitris Papadias, Qiongmao Shen, Yufei Tao, and Kyriakos Mouratidis. Group nearest neighbor queries. In *ICDE*, page 301, 2004.

[71] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *SIGMOD*, pages 259–270, 1996.

[72] Ira Pohl. Bi-directional search. *Machine Intelligence*, 6:127–140, 1971.

[73] Yunyao Qu, Changzhou Wang, and X. Sean Wang. Supporting fast search in time series for movement patterns in multiple scales. In *Proceedings of the seventh international conference on Information and knowledge management*, CIKM '98, pages 251–258, New York, NY, USA, 1998. ACM.

[74] Matt Richtel. Driven to distraction: Digital billboards, diversions drivers can‘t escape. *The New York Times*, March 01 2010.

[75] Mehdi Sharifzadeh and Cyrus Shahabi. The spatial skyline queries. In *VLDB*, pages 751–762, 2006.

[76] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The TPR*-tree: an optimized spatio-temporal access method for predictive queries. In *VLDB*, pages 790–801, 2003.

[77] Yannis Theodoridis, Jefferson R. O. Silva, and Mario A. Nascimento. On the generation of spatiotemporal datasets. In *SSD*. Springer-Verlag, 1999.

[78] Dalia Tiesyte and Christian S. Jensen. Recovery of vehicle trajectories from tracking data for analysis purposes. *Proceedings of the Sixth European Congress and Exhibition on Intelligent Transport Systems and Services*, June 18-20 2007.

[79] Nerius Tradišauskas, Jens Juhl, Harry Lahrmann, and Christian S. Jensen. Map matching for intelligent speed adaptation. In *6th European Congress on Intelligent Transport Systems and Services*, 2007.

[80] Georg Treu, Thomas Wilder, and Axel Küpper. Efficient proximity detection among mobile targets with dead reckoning. In *MobiWac*, pages 75–83, 2006.

[81] Marcos R. Vieira, Petko Bakalov, and Vassilis J. Tsotras. On-line discovery of flock patterns in spatio-temporal data. In *GIS*, pages 286–295, 2009.

[82] Jeffrey S. Vitter. Random sampling with a reservoir. *ACM TOMS*, 11:37–57, March 1985.

[83] Michail Vlachos, George Kollios, and Dimitrios Gunopulos. Discovering similar multidimensional trajectories. In *ICDE*, pages 673–684, 2002.

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

[85] Haojun Wang, Roger Zimmermann, and Wei shinn Ku. Distributed continuous range query processing on moving objects. In *DEXA*, pages 655–665, 2006.

[86] Yida Wang, Ee-Peng Lim, and San-Yih Hwang. Efficient mining of group patterns from user movement data. *DKE*, 57:240–282, Jun. 2006.

[87] Raymond Chi-Wing Wong, M. Tamer Özsu, Philip S. Yu, Ada Wai-Chee Fu, and Lian Liu. Efficient method for maximizing bichromatic reverse nearest neighbor. *PVLDB*, 2(1):1126–1137, 2009.

[88] Wei Wu. *Ecient Processing of Location-based Queries*. PhD thesis, National University of Singapore, 2009.

[89] Wei Wu, Wenyuan Guo, and Kian-Lee Tan. Distributed processing of moving k-nearest-neighbor query on moving objects. In *ICDE*, pages 1116–1125, 2007.

[90] Xiaokui Xiao, Bin Yao, and Feifei Li. Optimal location queries in road network databases. In *ICDE*, pages 804–815, 2011.

[91] Xiaopeng Xiong, Mohamed F. Mokbel, and Walid G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, Washington, DC, USA, 2005. IEEE Computer Society.

[92] Zhengdao Xu and Arno Jacobsen. Adaptive location constraint processing. In *SIGMOD*, pages 581–592, New York, NY, USA, 2007. ACM.

[93] Byoung-Kee Yi, H. V. Jagadish, and Christos Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE*, pages 201–208, 1998.

[94] Man Lung Yiu, Hou U Leong, Simonas Šaltenis, and Kostas Tzoumas. Efficient proximity detection among mobile users via self-tuning policies. *PVLDB*, 3(1):985–996, 2010.

[95] Man Lung Yiu, Yufei Tao, and Nikos Mamoulis. The bdual-tree: indexing moving objects by space filling curves in the dual space. *The VLDB Journal*, 17(3):379–400, 2008.

[96] Donghui Zhang, Yang Du, Tian Xia, and Yufei Tao. Progressive computation of the min-dist optimal-location query. In *VLDB*, pages 643–654, 2006.

[97] Zenan Zhou, Wei Wu, Xiaohui Li, Mong-Li Lee, and Wynne Hsu. Maxfirst for maxbrknn. In *ICDE*, pages 828–839, 2011.