Optimal-Location Queries over Spatial Databases

Yang Du

A Dissertation
Presented to the Faculty
of Northeastern University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer & Information Science

April 2009
Abstract

We study the optimal-location queries in spatial databases. Given a set $S$ of sites, a set $O$ of objects, and a spatial region $Q$, the optimal-location query returns a location in $Q$ such that, if a new site is put at that location, the total benefit to the objects is maximized. The optimal-location queries are of interest in many applications such as corporation decision-support systems. There may be many variations of the optimal location because different applications may define the optimality differently. In this thesis, we consider the geometric proximity between objects and sites and study two most intuitive definitions of the optimality, namely Max-Inf and Min-Dist. Max-Inf measures the total benefit as the number of objects closer to the new site than to any existing sites and aims to maximize it. Min-Dist measures the benefit as the savings of the distance from objects to their nearest sites and aims to minimize the average distance from each object to its nearest site. We also examine the problem under three different distance metrics, namely $L_1$, $L_2$ and network shortest path. We propose efficient solutions for each optimal location query (6 in total) and evaluate them experimentally. We expect that the work will not only advance spatial database research, but also benefit the end users who deal with a large volume of spatial data.
Acknowledgements

It is hard to express in words how thankful I am to my advisor Prof. Donghui Zhang, whose guidance and support has made this journey truly magnificent. He has not only been my advisor but also a friend who has taught me so many things beyond research. I will never forget the discussions we had over countless meetings. I would also like to especially thank Prof. Betty Salzberg whose diligent comments and suggestions have helped in improving my dissertation in particular and outlook for research in general. She too has been a genuine source of inspiration and guidance relating to research and beyond.

There are so many people whose support and friendship I have enjoyed during the course of my Ph.D. In particular I would like to sincerely thank Tian Xia, who was my office mate for 6 years and co-author of a number of papers, has also been a great friend. I have learnt a lot from what he said and did. He has always been beside me during the highs and the lows for which I am very grateful. I would like to acknowledge my friends Da Lin and Jun Gong for their support, and encouragement, and making this journey immensely pleasurable. I would also like to thank members of Database lab for their support and friendship.

Finally I would like to thank my extraordinary parents, my dear wife, my lovely son, and my loyal brother for their relentless support and encouragement in all aspects of life, and making me a person who I am today.
Contents

Abstract ................................................................. iii
Acknowledgements ...................................................... iv

1 Introduction .......................................................... 1
  1.1 Problem Statement .................................................. 3
  1.2 Our Results .......................................................... 5
  1.3 Outline ............................................................... 6

2 Related Work .......................................................... 7
  2.1 Spatial Access Method ............................................... 7
    2.1.1 R-tree ............................................................ 8
    2.1.2 Disk-based Spatial Network .................................... 10
  2.2 NN and RNN Queries ............................................... 13
    2.2.1 NN query ....................................................... 13
    2.2.2 RNN Query ..................................................... 14
    2.2.3 NN and RNN over Road Networks .............................. 19
  2.3 Facility Location Problems ....................................... 21

3 Max-Inf OL under $L_1$ and $L_2$ Metric ....................... 24
  3.1 Problem Transformation .......................................... 24
  3.2 The R*-tree-based Solution ...................................... 26
    3.2.1 Overview ..................................................... 26
3.2.2 Retrieving Objects from The R*-tree . . . . . . . . . . . . . . 27
3.2.3 The Naive Plane Sweep . . . . . . . . . . . . . . . . . . . . . 28
3.2.4 Preliminaries . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
3.2.5 The aSB-tree . . . . . . . . . . . . . . . . . . . . . . . . . . . 32
3.2.6 Involving A Query Region . . . . . . . . . . . . . . . . . . . . 35
3.3 The Virtual OL-tree . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.3.1 The OL-Tree . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.3.2 The VOL-tree Structure . . . . . . . . . . . . . . . . . . . . . 40
3.3.3 The VOL-tree Query Algorithm . . . . . . . . . . . . . . . . . 42
3.3.4 The Update Algorithm . . . . . . . . . . . . . . . . . . . . . . 44
3.3.5 The Bulk-Loading Algorithm . . . . . . . . . . . . . . . . . . . 46
3.4 Max-Inf OL under $L_2$ Metric . . . . . . . . . . . . . . . . . . . . 49
3.4.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
3.4.2 Retrieving Objects from The R*-tree . . . . . . . . . . . . . . 50
3.4.3 The Naive Plain Sweep . . . . . . . . . . . . . . . . . . . . . . 51
3.4.4 Improved Intersection Detection . . . . . . . . . . . . . . . . . 53
3.4.5 Arc Maintenance . . . . . . . . . . . . . . . . . . . . . . . . . 56
3.4.6 Special Cases . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
3.4.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
3.5 Performance . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
3.5.1 Results under $L_1$ Metric . . . . . . . . . . . . . . . . . . . . 61
3.5.2 Results under $L_2$ Metric . . . . . . . . . . . . . . . . . . . . 64
3.6 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

4 Min-Dist OL under $L_1$ and $L_2$ Metric . . . . . . . . . . . . . . 67
4.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
4.2 Computing $AD(l)$ for A Location $l$ . . . . . . . . . . . . . . . . 68
4.3 Limiting the Number of Candidates . . . . . . . . . . . . . . . . . 70
5.2 Basic Solution to Max-Inf OL .................................. 110
  5.2.1 Query Processing Algorithm ................................. 110
  5.2.2 Finding Local OL on Edges ................................. 112
5.3 Improved Solution to Max-Inf OL .............................. 113
  5.3.1 Pre-computing on Vertices ................................. 114
  5.3.2 Removing Redundancy ..................................... 115
5.4 Solution to Min-Dist OL ....................................... 117
5.5 Performance ..................................................... 118
  5.5.1 The Effect of Pre-computation on Vertex ................. 119
  5.5.2 The Effect of Redundancy Removal ....................... 119
5.6 Conclusions ...................................................... 120

6 Conclusions ....................................................... 122
Chapter 1

Introduction

Spatial databases play very important roles in applications such as Geographical Information Systems and Corporation Decision-Support Systems. In these applications, the requests from end users follow certain patterns and can be modeled into several type of queries. For example, finding the nearby post office can be mapped to a Nearest Neighbor (NN) query, while finding the total number of residents within a given city can be mapped to a Range Aggregation (RA) query over spatial databases. Many queries, such as the NN and RA queries, have been extensively studied in the database research community. However, some other important queries have been paid much less attention to so far. One example is the Optimal-Location (OL) query studied in this thesis.

Roughly speaking, the Optimal-Location (OL) query can be defined as follows: given a set of (weighted) objects, a set of service sites, and a query range, the OL query returns the best location in the query range to build a new site. The OL query is motivated by many real life applications. For instance, many corporations or organizations may frequently ask the similar questions such as “where is the best location in the Boston area to open a new branch?” However, different applications have different objectives and therefore lead to different definitions of the “best” location.
Here we present two examples to manifest the motivation of two $OL$ queries studied in this thesis.

- **Max-Inf $OL$:** Consider that the McDonald’s Corporation wants to open a new branch in Boston area to compete with the other stores in the fast food business. In this case, the new location is expected to be *influential* as much as possible, i.e. it is desired to maximize the number of customers attracted to the new location. Here we assume the customers will visit the new store if it is closer to them than any existing fast food stores. It is a reasonable assumption in that geometrical proximity is a crucial factor when the customers choose the store to visit.

- **Min-Dist $OL$:** Consider another case where the Boston government wants to build a new elective office. In this case, no competition exists. Instead, the new office is expected to benefit the voters as much as possible. In this case, what matters is not just how many voters benefit from the new office, but also how much the voters benefit from it. For example, saving 10 miles for a hundred voters might be a better option than saving 0.1 miles for a thousand voters. Therefore, the criteria is to minimize the (average) distance from voters to their nearest elective offices.

The research on $OL$ queries is expected to have broad impacts on various fields that deal with a large volume of spatial data. The following examples show two of the potential applications of the $OL$ queries.

**Example 1 (Wireless communication)** In the wireless applications, the mobile devices (such as cell phones, PDAs, sensors, etc) usually communicate with the nearest stations. The quality of the communication between a mobile device and a station is primarily decided by the (Euclidean) distance between them. The $OL$-queries are meaningful in this scenario. Specifically, the Max-Inf
OL query finds the location to build a station which enhances the service quality of the most number of mobile devices, while the Min-Dist OL query finds the location to build a station which improves the overall service quality the most.

**Example 2 (Network Infrastructure)** In many network applications, the requests (such as searches, webpage fetches, etc) from the clients are routed to the nearest servers which process the received requests. The major factor that determines the response time of a request is the transmission latency between the client and the server. In the case that a new server is desired to improve the response time, the OL queries make similar sense as in the example above. The difference is that, instead of using the Euclidean distance, the network distance should be used to represent the transmission latency between network infrastructures.

In the above examples, the results of the OL queries are usually not the final location to build the new facility or server. Instead, they help the managers to evaluate different possibilities. Before making the final decisions, the managers may issue the OL queries multiple times on different datasets and query ranges. Therefore, the OL queries have to be answered quickly in a on-the-fly fashion. In this thesis, we study how to answer the OL queries efficiently over large spatial databases.

### 1.1 Problem Statement

In this section we formally define the *Max-Inf OL* query and the *Min-Dist OL* query.

**Definition 1.1.1** (The *Max-Inf OL* query). Given a set $O$ of objects, a set $S$ of sites, and a distance metric $d(\cdot, \cdot)$, we define the influence of any location $l$ as the total number of objects that are closer to $l$ than any site in $S$ with respect to the given
distance metric, i.e.

\[ \inf(l) = \left| \{ o \in O | \forall s \in S, d(o, l) < d(o, s) \} \right|. \]

Given a spatial region \( Q \), the Max-Inf OL query returns a location whose influence is maximal among all locations in \( Q \).

**Definition 1.1.2** (The Min-Dist OL query). Given a set \( O \) of objects, a set \( S \) of sites, and a distance metric \( d(\cdot, \cdot) \), we define the average distance \( AD(l) \) of any location \( l \) as the average distance between each object to its nearest site, considering all sites in \( S \) and a hypothetical new site at \( l \), i.e.

\[ AD(l) = \frac{1}{|O|} \sum_{o \in O} \min_{s \in S \cup \{l\}} d(o, s). \]

Given a spatial region \( Q \), the Min-Dist OL query returns a location whose average distance is minimal among all locations in \( Q \).

Here the objects represent the customers/clients in real applications and the sites represent the facilities/servers. We assume both objects and sites are points. In the above definitions, we did not restrict the distance metric to any specific one. The distance metric depends on the application. In this thesis, we consider three most popular distance metrics, namely the \( L_1 \) metric, the \( L_2 \) metric, and the network metric. In the \( L_1 \) metric, the distance between two points is the sum of the absolute differences between their coordinates. It is also named as the Manhattan distance because it is the shortest driving distance in a city where the streets are laid out in a lattice-like pattern as in Manhattan. In the \( L_2 \) metric, the distance between two points is the ordinary Euclidean distance. In many applications such as mobile services, the distances between service centers and users are measured by Euclidean distance. In the network metric, the distance between two points is the shortest
path distance over the network (graph). It is widely used in navigation systems and network applications.

All three metrics are very widely used. Meanwhile, they have different properties. Therefore, we study the Max-Inf OL and the Min-Dist OL queries under these three metrics.

In the past decades, researchers have proposed many index structures and algorithms to efficiently process spatial queries. Conceptually, the OL queries can be answered by applying the existing techniques as follows. First, for each candidate location \( l \), we find the objects that are closer to \( l \) than any site in \( S \) by using the existing reverse nearest neighbor search algorithms. Then we compute the influence or the average distance of candidate location \( l \). Finally, we return the candidate location who has the maximal influence or minimal average distance as the result. However, there are serious problems with this straightforward solution. In general, there are an infinite number of candidate locations, which cannot be enumerated. It is not trivial at all to reduce the candidates to a finite set. Second, even if the number \( f \) candidates is reduces to a finite number, this method is still prohibitively expensive when this number is large. In this thesis, we present techniques to process the OL queries much more efficiently.

1.2 Our Results

We propose two solutions for Max-Inf OL under \( L_1 \) metric. The first solution computes the OL on the fly. It assumes that the data objects are stored in an R*-tree and uses a plain sweep algorithm to find the OL. The second solution is based on a new specialized aggregation index called the \( VOL-tree \), which stores the precomputed results. Both solutions are extended to the Max-Inf OL under \( L_2 \) metric. For Min-Dist under \( L_1 \) metric, we propose a progressive algorithm which quickly suggests
a candidate location with the maximum possible error and keeps refining the result.
The extension of this algorithm serves as an approximation algorithm for Min-Dist under $L_2$ metric. We also propose a solution framework based on precomputation for both Max-Inf OL and Min-Dist OL under network metric. Several improvements are used to reduce the storage requirements in this solution framework.

1.3 Outline

The rest of this thesis is organized as follows. Chapter 2 reviews existing work related to the thesis. Chapter 3 solves Max-Inf OL under $L_1$ and $L_2$ metric. Chapter 4 solves Min-Inf OL under $L_1$ and $L_2$ metric. Chapter 5 solves Max-Inf OL and Min-Dist OL under road network. Finally, chapter 6 concludes the thesis.
Chapter 2

Related Work

This chapter extensively surveys and compares existing work related to the OL queries. Section 2.1 reviews the literature on spatial access methods, section 2.2 reviews nearest neighbor (NN) query and reverse nearest neighbor (RNN) query processing techniques, and section 2.3 reviews the facility location problem studied in the approximation algorithms community.

2.1 Spatial Access Method

The objects and sites in the OL queries are spatial data. They need to be indexed so that they can be accessed efficiently. In the past decades, numerous spatial access methods have been proposed to index spatial data. Among all proposed methods, the R-tree [15] is the most popular one to index spatial objects embedded in Euclidean space. Several methods are proposed to index the objects embedded in network. We review these methods in this section.
2.1.1 R-tree

In the R-tree, spatial objects are represented as either points or rectangles. To index the data objects, the R-tree clusters objects close in space into the same node, which is represented as a minimum bounding rectangle (MBR). This principle is recursively applied to the nodes created in the previous step and organize them into a dynamic, disk-based, balanced tree structure. Eventually the MBR of the root node contains all objects stored in the tree, and the MBR of each node contains all objects stored in the sub-tree of that node. As in the B-tree, every node except root has a fanout between \( m \) and \( M \), where the lower bound \( m \) guarantees a minimum page utilization as well as the maximum height of the tree and the upper bound \( M \) makes sure that each node fits in one disk page. \( M \) is also called the capacity of the nodes. Figure 2.1 is an example of the R-tree indexing a set of 2-dimensional data points. Here we assume a capacity of three entries for both index and leaf nodes.

![Diagram of R-tree example](image)

Figure 2.1: R-tree Example.

The R-tree can be used to answer many spatial queries. In general, the search is done in a top-down fashion. Starting from the root, all index entries are tested to see whether they satisfy the search predicate. Then all qualified child nodes need to be recursively visited until all qualified data objects are found. Consider the range query, which returns all objects with in the query range, as an example. Assume that the query range is \( Q \), the shadowed area in Figure 2.1(a). Since the MBR of \( N_2 \) does
not intersect with $Q$, it is guaranteed that no object in the subtree of $N_2$ intersects with $Q$. Therefore, $N_2$ is pruned and only $N_1$ is visited. In $N_1$, the MBRs of both $n_1$ and $n_2$ intersect with $Q$. Thus, both $n_1$ and $n_2$ are visited. After examining the objects within $n_1$ and $n_2$, the answer is found to be $b$ and $e$.

To answer queries efficiently, the R-tree should be arranged in the way that searches will visit a minimal number of nodes. However, due to the dynamics of the R-tree, it is hard to keep the R-tree globally optimal. In the original R-tree paper, Guttman used a heuristic, which minimizes the size (area) of MBR of each node. The insertion and deletion algorithm follow this heuristic.

To insert a data object, we traverse only a single path from the root to the leaf. At each level we choose the child node whose MBR needs the least enlargement to enclose the data object (or its MBR). If several nodes satisfy this criterion, the node who has the smallest MBR will be selected. Once the leaf level is reached, the object is inserted. If the data object enlarges the MBR of the corresponding leaf node, it is adjusted accordingly and the change is propagated upward. If the leaf node overflows, we split it into two nodes and distribute the objects among them. Again, the changes of the MBRs of both nodes will be propagated upward. Deletion is performed in a similar way. To delete a data object, we first search the object in the R-tree. We delete the object from the leaf node if it is found. If the leaf node does not underflow, we simply adjust the MBR if necessary and propagate the change upward. Otherwise, the node is deleted from the tree. After the change is propagated up the tree, its contents are reinserted into the tree.

To improve the performance of R-tree, many variants [38, 6, 22, 50] have been proposed. Among them, the R*-tree proposed by Beckmann et al. [6] is the most widely used one. Based on a careful study of the performance of the R-tree under different data distributions, they observe that the insertion phase is critical for query performance. Therefore the force reinsert policy is introduced into the insertion
algorithm: if a node overflows, instead of splitting it into two, \( p \) entries are removed from the node and reinserted into the tree. Here \( p \) is a predefined parameter which can be used to tune the system performance. They also consider different policies for node splitting: 1) minimizing the overlap between MBRs, 2) minimizing perimeters of MBRs, and 3) maximizing the storage utilization.

### 2.1.2 Disk-based Spatial Network

To study the \( OL \) queries under network metric, we need to know how to store the network properly. A network is usually modeled as a graph and represented either as a set of adjacency lists or as a matrix, each entry of which corresponds to the connectivity between a pair of nodes. Usually, adjacency list representation is more preferable for spatial networks, where the graphs are sparse. The main goal of adapting adjacency list representation to disk-based representation is to minimize the disk I/O. In order to do so, adjacency lists of adjacent nodes are expected to be clustered in the same disk page.

![A network example](image1)

(a) A network example

![The CCAM structure](image2)

(b) The CCAM structure

Figure 2.2: An example of a network represented by the CCAM.

In [39], Shekhar et al. propose the connectivity-clustered access method (CCAM) to address this issue. The CCAM generates a single-dimensional ordering of the nodes (for example, Z-ordering) and stores the lists of neighbor nodes together. The adjacency lists of the neighboring (in the Z-order) graph nodes are stored in one page.
Each entry in the list of a node contains an adjacent node and the corresponding network distance. In order to efficiently retrieve the adjacency list of a node, the list pages are indexed by a B+-tree on the node id. Figure 2.2 shows an example of a network represented by the CCAM. The nodes in Figure 2.2 (a) are labeled in the increasing order of their Z-order values, and they are indexed by a B+-tree in Figure 2.2 (b).

In [21], Jensen et al. propose a data model and the definitions of abstract functionality required for continuous k-NN queries in road networks. The data model encompasses two levels of data representation, which can be implemented using the relational tables. The first level is a detailed two dimensional representation, which captures the geographical coordinates of the roads and nodes. It uses polylines to approximate the curvy roads. Continuing the example in Figure 2.2(a), let us assume that road \( n_2n_4 \) is broken into 5 line segments by adding 4 new nodes \( n'_1 \) to \( n'_4 \). All these 6 nodes will be stored in one table in the form of \( (n_{id}, x, y) \). The line segments \( n_2n'_1, n'_1n'_2 \) to \( n'_4n_4 \) are stored in another table in the form of \( (l_{id}, start, end, length) \). Other roads are processed in the same way. Some useful information such as the speed limit can also be stored in the line table. The second level data representation is a more abstract level, which captures the topology of the roads. It uses the data from the first level to calculate the network distance of each road and store it in a table in the form of \( (r_{id}, start, end, distance) \). Note that the start and end nodes in the road table can be referred back to the node table in 2D representation.

Papadias et al. [35] propose a solution for NN queries in road networks by introducing a network storage scheme that integrates network and Euclidean information. Their model consists of three components, namely the adjacency component, the polyline component, and network R-tree. Figure 2.3 shows an instance of their storage scheme based on the configuration of Figure 2.2. Specifically, the adjacency component captures the network connectivity. Similar to the CCAM model, it stores the
adjacency lists of the nodes close in space in the same disk page. In Figure 2.3, the list $l_4$ of $n_4$ consists of 2 entries, one for each of its connected nodes ($n_2$ and $n_6$). The first entry (for edge $n_4n_2$) has the form $(p_{AL}(n_2), d_{42}, MBR(n_4n_2), p_{PL}(n_4n_2))$. The field $p_{AL}(n_2)$ points to the disk page (i.e., $P_1$) containing the adjacency list $l_2$ of $n_2$. It enables fast access to the adjacent node without any additional access, while CCAM requires look-up in B-tree. The next field ($d_{42}$) is the network distance of edge $n_4n_2$. $MBR(n_4n_2)$ records the minimum bounding rectangle of the polyline representation of $n_4n_2$, which is stored in the disk page ($P_3$) specified by field $p_{PL}(n_4n_2)$. The other adjacency entries have the same format. The poly-line component stores the detailed polyline representation of each edge in the network. A polyline entry of edge $n_in_j$ also includes a pair of pointers to the disk pages containing the adjacency lists of its nodes $n_i$ and $n_j$. The network R-tree component indexes the polylines MBRs (not the actual polylines) and supports queries exploring the spatial properties of the network. Each leaf entry contains a pointer to the disk page storing the corresponding polyline.

Figure 2.3: An example of a network storage scheme proposed by Papadias et al.
2.2 NN and RNN Queries

In this section, we review the existing NN and RNN search algorithms over the spatial access methods presented in the previous section.

2.2.1 NN query

**Definition 2.2.1** (The NN query). Given a set $O$ of objects and a query location $l$, the NN query finds the object $o \in O$ that is nearest to $l$.

The NN query is first introduced in [37] and has received vast attention in the spatial database research community. In [37], the authors propose a R-tree based NN algorithm. The algorithm searches the R-tree in a depth-first manner. It starts from the root of the tree and visits the entry with the minimum distance from the query location $l$. This process is repeated recursively until the leaf level, where the first potential nearest neighbor is found. During backtracking to the upper level, the algorithm only visits entries whose minimum distance (to $l$) is smaller than the distance of the nearest neighbor already found. Then current known nearest neighbor may be replaced by a new object if the new object has a smaller distance to $l$. Consider the example in Figure 2.4, where black dots, $a$ to $k$, are objects and the gray dot $l$ is the query location. Among the entries in the root, the algorithm first visits $N_2$, whose minimum distance to $l$ is 0. The search further goes down to $n_4$ and $i$ is selected as the candidate. Then the algorithm backtracks to $N_2$. Since the minimum distance from $l$ to $n_3$ is less than $d(l, i)$, $n_3$ is visited and the candidate will be replaced by $f$. Then the algorithm backtracks to the root and visits $N_1$ and then $n_2$. The actual nearest neighbor $e$ is discovered afterwards.

Hjaltason et al. [16] propose a best-first R-tree traversal algorithm which utilizes a priority queue. The algorithm keeps in the queue the candidate entries in increasing order of their minimum distances from the query location. In each round, the algo-
algorithm accesses the first entry in the queue and inserts its children into the queue. The process is repeated until the first data point is found. It guarantees that the first data point extracted from the queue is the nearest neighbor. This method is optimal in the sense that it only visits the nodes necessary for finding the answer. Let us use the example in Figure 2.4 again. First, $N_1$ and $N_2$ are inserted into the query. Then $N_2$ is extracted, $n_3$, $n_4$ and $n_5$ being inserted into the query. Since their minimum distances from the query location are larger than $N_1$’s, $N_1$ is extracted and $n_1$ and $n_2$ are inserted into the query. Now the queue has five entries, $n_1$ to $n_5$. In next step, $n_2$ is visited and $d$ and $e$ are inserted into the queue. Since $d(l, e)$ is the smallest, the algorithm returns $e$ as the answer. Comparing to the depth-first method, this method avoid visiting the elements in $n_3$ and $n_4$.

The methods in [37] and [16] can be easily extended to answer the kNN query, which finds the $k$ nearest neighbors to the query point. Other researchers [46, 34, 51, 49] also study the problem of continuously monitoring the NN (kNNs) when the queries and the data objects move frequently, which is less related our thesis. Therefore, the details are omitted.

2.2.2 RNN Query

The RNN query is introduced by Korn et al. [25]. Given a query location, RNN query finds the objects from the dataset that consider $l$ as their closest object. There
are two variations of the RNN query: the monochromatic RNN and the bichromatic RNN. In the monochromatic case, there is only one dataset, say \( O \). The distance between an object \( o \in O \) and the query location \( l \) is compared with the distances between \( o \) and other objects in \( O \). In the bichromatic case [43], there is another dataset: a set \( S \) of sites. The distance between \( o \) and \( l \) is only compared with the distances between \( o \) and sites in \( S \). Formally, the monochromatic and bichromatic RNN queries are defined as follows.

**Definition 2.2.2** (The Monochromatic RNN query). Given a set \( O \) of objects and a query location \( l \), the (monochromatic) RNN query finds all the objects \( o \in O \) that have \( l \) as their nearest neighbor (i.e. no other object in \( O \) is closer to \( o \) than \( l \)).

**Definition 2.2.3** (The Bichromatic RNN query). Given a set \( O \) of objects, a set \( S \) of sites, and a query location \( l \), the (bichromatic) RNN query finds all the objects \( o \in O \) that have \( l \) as their nearest site (i.e. no site in \( S \) is closer to \( o \) than \( l \)).

While the monochromatic RNN query seems relatively easier, many real-life applications correspond to the bichromatic case. For instance, given the location of a new McDonald’s store, the residence buildings that are closer to this store than to any existing store can be computed by bichromatic RNN.

We start with algorithms for monochromatic RNN queries. The first solution to the RNN query is based on precomputation [25]. It precomputes the nearest neighbor for each object \( o \), represents \( o \) as a circle centered at \( o \) with radius equal to the distance from \( o \) to its nearest neighbor, and then builds a R-tree called \( RNN\text{-tree} \) to index the MBRs of all circles. Using the RNN-tree, the RNN queries can be efficiently answered by a point location query, which retrieves all circles that contain the query location. Because the RNN-tree is specialized for RNN, but not NN search, the authors of [25] also use an additional R-tree on the data objects for computing nearest neighbors and the radiiuses of the circles.
In order to avoid the maintenance of two separate tree structures, Yang and Lin [52] combine the two trees into the Rdnn-tree. It is an R-tree of objects, where every object stores the distance to its nearest neighbor, while every index entry stores the maximum distance from every point in the subtree to its nearest neighbor. The structure logically maintains the R-tree of circles. It remains to determine, given a query location \( l \) and an index entry \( e \), whether the sub-tree of \( e \) may contain some object whose “circle” (not physically stored) encloses \( l \). The solution is to expand the MBR of \( e \) outward by the associated maximum distance. If the expanded region does not enclose \( l \), there is no need to check the sub-tree of \( e \). Otherwise, the sub-tree of \( e \) will be visited.

Techniques [25, 52] that rely on precomputation cannot handle updates efficiently. Stanoi et al. [42] propose a method to avoid precomputing NNs for all objects. Their method utilizes the following property proved in [41]: given a set of 2-dimensional point, there are at most 6 RNNs for any query location. More specifically, consider the example in Figure 2.5 with 6 objects \( o_1 \) to \( o_6 \). The space around the query location \( l \) is divided into six equal regions \( S_1 \) to \( S_6 \). It is easy to verify that either 1) the NN of \( q \) in some region \( S_i \) is RNN of \( l \), or 2) there is no RNN of \( q \) in \( S_i \). For instance, \( o_1 \) is the NN of \( l \) in \( S_1 \) and it is also the RNN of \( l \). On the other hand, there is no RNN of \( l \) in \( S_4 \) because \( o_4 \) is closer to \( o_5 \) than to \( l \). Based on the above property, the method in [42] takes two steps. First, it retrieves the nearest neighbors of query location in six regions. These objects become the candidate. Then, it finds the actual nearest neighbor of each candidate and removes the candidate from the result if it is a false positive. In the example of Figure 2.5, the objects \( o_1 \), \( o_3 \), \( o_4 \) and \( o_6 \) are retrieved as candidates and \( o_4 \) will be removed in the refinement step. Therefore, the final result is \( o_1 \), \( o_3 \) and \( o_6 \).

Assuming that the data points are indexed by an R-tree, Tao et al. [45] also adopt a two-step framework that retrieves a set of candidate and then removes false positives.
Figure 2.5: Example of [42].

First, the algorithm browses the R-tree in a top-down fashion and retrieves a set of potential candidates in ascending order of their distance to the query location. The entries that cannot contain any candidate are pruned by a half-plane strategy and are stored (without being visited) in a refinement set. At the second step, the entries in the refinement set are used to eliminate false positives in the candidate set. The algorithm combines two steps into a single traversal of the R-tree, and therefore is more efficient.

Figure 2.6 illustrates the details of the half-plane pruning strategy. In Figure 2.6(a), the solid line is the bisector between the query location $l$ and a data object $o$. It divides the space into two half planes. Any object in the half plane containing $o$ cannot be RNN of $l$ because it is closer to $o$ than $l$. Furthermore, a node $N$ whose MBR that completely falls in this half-plane does not contain any RNN of $l$, and therefore can be pruned. Combining several objects together may provide greater pruning power. For example, in Figure 2.6(b), $N$ can not be pruned by either $o_1$ or $o_2$ separately. However, it can still be pruned by the union of the half-planes containing $o_1$ and $o_2$.

In the filter step, a set $S_c$ of candidates is found and each pruned entry is inserted in a refinement set $S_r$. In the refinement step, the entries of $S_r$ are used to eliminate false hits in $S_c$. Let $P_r \subseteq S_r$ be the set of points and $N_r \subseteq S_r$ be the set of MBRs in $S_r$. A point $p \in S_c$ can be discarded as a false hit, if (i) there is a point $p' \in P_r$ such that
Figure 2.6: Illustration of half-plane pruning strategy.

\[ d(p, p') < d(p, l) \text{ (i.e., } p \text{ is closer to } p' \text{ than to } l) \text{, or (ii) there is an node MBR } N \in N_r \text{ such that } \text{minmaxdist}(p, N) < d(p, l) \text{ where } \text{minmaxdist}(p, N) \text{ gives an upper bound of the distance between } q \text{ and its closest point in } N. \text{ On the other hand, a point } p \in S_c \text{ can be reported as an actual result without any extra node accesses, if (i) for every point } p' \in P_r \text{ } d(p, p') \geq d(p, l) \text{ and (ii) for every node } N \in N_r \text{ } \text{mindist}(p, N) \geq d(p, l). \] These rules are applied to every candidate to refine the result efficiently. It is possible that a point \( p \in S_c \) can neither be reported as a false positive nor as an actual result if there is a node \( N \in N_r \) such that \( \text{mindist}(p, N) < d(p, l) \leq \text{minmaxdist}(p, N) \). In this case, \( N \) will be visited and the rules are recursively applied to its children.

As to the bichromatic queries, the methods of [25, 52] can be applied directly by only considering the distance from objects to sites. However, they still have problem with efficient updates in that they are based on precomputation. Stanoi et al. [43] propose another solution which dynamically constructs the influence region of the query location \( l \). Here, the influence region is defined as an area which encloses and only encloses all possible RNNs of \( l \). This is equivalent to the Voronoi cell enclosing \( l \) [8]. Figure 2.7(a) shows how to construct the Voronoi cell \( VC(l) \) for a given query location \( l \). Every boundary of \( VC(l) \) is the bisector between \( l \) and an other site. Therefore, the straightforward method of computing a Voronoi cell requires to compare \( l \) with all sites. However, it is inefficient for large datasets with a large number of sites. The method in [43] follows a approximate-and-refine strategy to compute the \( VC(l) \). In the approximating step, an approximation of \( VC(l) \) is
generated by visiting at most 4 sites. In particular, it considers only the nearest neighbors of $l$ in each quadrant (centered at $l$). Figure 2.7(b) shows an example where $A$, $B$, $C$ and $D$ are the nearest neighbors and the quadrilateral is the approximation of $VC(l)$. The actual $VC(l)$, even though is unknown, should be contained by the quadrilateral. In the refining step, it uses the approximation of $VC(l)$ to infer the area that includes all the sites needed to refine $VC(l)$. Consider the circles whose diameter is the line segment between $l$ and a vertex of the quadrilateral (Figure 2.7(c) shows the circles based on Figure 2.7(b)). The authors proved that, if a site $s$ contribute to $VC(l)$, the middle of segment between $l$ and $s$ must be within one of the circles. Based on this observation, only a small subset of the sites need to be accessed in the refining step. Each site is checked against the approximation to refine $VC(l)$.

![Figure 2.7: Construction of Voronoi Cell.](image)

### 2.2.3 NN and RNN over Road Networks

There are also some researches on NN and RNN queries over road networks. Jensen et al. [21] consider the continuous (k)NN queries where query and data objects that are moving in road networks. A client-server architecture is used in their solution: the server manages the database of all data objects, while the clients (data objects) periodically report their location information to the server. To answer a (k)NN query, the server first generates a set of current nearest neighbors and sends them to the requesting client. The client keeps tracking the distances to the received data objects.
and estimates the imprecision of the received result. When the imprecision is beyond a certain threshold, it will request the server to refresh the result. To support fast query processing, a so-called database of distances is used to store pre-computed shortest distances between any pair of nodes.

Papadias et al. [35] propose another solution for nearest neighbor queries in road network databases by introducing an architecture that integrates network and Euclidean information. The authors introduce two algorithms, namely the incremental Euclidean restriction (IER) and incremental network expansion (INE), for NN queries over road network. The IER algorithm takes advantage of the Euclidean lower-bound property to prune the search space. Given a query location \( l \), it first retrieves the Euclidean nearest neighbor \( NN_1 \) of \( l \). Then, the network distance \( d_N(l, NN_1) \) between \( l \) and \( NN_1 \) is computed. Due to the Euclidean lower bound property, objects closer (to \( l \)) than \( NN_1 \) in the network, should be within \( d_N(l, NN_1) \) from \( l \) based on Euclidean distance. Then the second Euclidean NN \( NN_2 \) is then retrieved (within this distance restriction). If \( d_N(l, NN_2) < d_N(l, NN_1) \), \( NN_2 \) becomes the current NN and the distance restriction is updated to \( d_N(l, NN_2) \). This procedure is repeated until the next Euclidean NN falls out of the search region based on the distance restriction. On the other hand, the INE algorithm performs query processing directly on the network by expanding from the query point. Starting with two nodes of the edge which covers the query location \( l \), it keeps expanding the node whose network distance to \( l \) is minimal (\( i.e. \) in the way that Dijkstra algorithm works). When a node \( n \) is expanded, for each non-visited adjacent node \( n' \) of \( n \), the objects on the edge \( nn' \) will be checked. If an object has a shorter network distance from \( l \) than the current candidate, the candidate is updated. The algorithm terminates when the next node to be visited has larger network distance from \( l \) than the candidate. These two algorithms can be straightforwardly extended to kNN queries.

Kolahdouzan and Shahabi [23] propose a Voronoi-based solution for nearest neigh-
bor queries in road networks. This approach partitions a large network to small Voronoi cells based on network distance. Each cell contains exactly one object and certain number of nodes and border points, which are the intersections of edges and the border of the cell. To achieve faster query processing, it materializes the intra and inter distances for each cell. In particular, the intra-cell distances of a cell are the distances between all the nodes (or border points) of the cell to the object inside the cell. The inter-cell distance of a cell are the distances between each pair of border points. Given a query location \( l \), the NN can be simple found by locating the cell that contains \( l \) and returning the object it contains. To find the \( k \)NNs of \( l \), it employs a strategy that find the NNs according their distance from \( l \). The authors prove that the next NN of \( l \) is within the adjacent cells of the previously explored ones. Based on this observation, the algorithm keep exploring the neighboring cells to search the next NN until \( k \) objects are found. To compute the distance from \( l \) to each candidate, it utilizes the (pre-computed) intra-cell distances to find the distance from \( l \) to the borders of the cell of each candidate, and the (pre-computed) inter-cell distances to compute the actual network distance from \( l \) to each candidate.

### 2.3 Facility Location Problems

A problem closely related to our research is the facility location (FL) problem, which is a central problem in operation research. It has also been well studied in the algorithm community. The FL problem has a large number of applications. Examples are manufactories, chain stores, warehouses, post offices, wireless base stations, etc. These examples have in common that a set of facilities has to be chosen, and the objective is to meet the user demands efficiently.

There are several variants of the FL problem. The most widely studied one is the Uncapacitated Facility Location (UFL) problem. It is defined as follows. To be
consistent, we use the terms “objects” and “sites” as we did in the previous sections, instead of using “customers” and “facilities” as used in most of the facility location literatures.

**Definition 2.3.1** (the UFL Problem). Given a set $O$ of objects, a set $S$ of potential sites, a nonnegative cost $f_i$ for opening each site $s_i \in S$, and a nonnegative service cost $c_{ij}$ between each site $s_i \in S$ and object $o_j \in O$, we desire to open a subset $S'$ of the sites in $S$ so that the total cost (opening costs plus service costs)

$$\sum_{s_i \in S'} f_i + \sum_{o_j \in O} \min_{s_i \in S'} c_{ij}$$

is minimum.

Here, the term “uncapacitated” refers to the fact that there is no limit on the number of objects that each site can serve. Another well-studied FL problem is the $k$-median problem.

**Definition 2.3.2** (the $k$-Median Problem). Given a parameter $k$, a set $P$ of points, and a nonnegative service cost $c_{ij}$ between each pair of points $(p_i, p_j)$, we desire to select a set $P'$ of $k$ points (as medians) in $P$ so that the total service cost for all points

$$\sum_{p_j \in P} \min_{p_i \in P'} c_{ij}$$

is minimum.

If we start with the UFL problem, consider the case in which $O = S = P$, set each $f_i = 0$, and impose an additional constraint that there are only $k$ sites can be opened, then we obtain the $k$-median problem.

Both the UFL problem and the $k$-median problem are NP-hard. Actually, it is pointed out by Vygen [48] that, without the assumption of metric service costs,
they are as hard as the classical *Set Covering* problem [7]. In operation research and algorithm community, most research studies [17, 40, 13, 12, 26, 20, 10, 2, 29, 19, 44, 18, 28, 3, 1, 4, 9, 11, 31] are focused on the theoretical aspects, such as the hardness of the problem and the approximation ratio of the algorithms. In the context of data clustering, some algorithms in linear or sublinear time [14, 36, 33, 32] have been proposed to provide good approximations to the *k*-median problem (with high probability).

The *FL* problem is very closely related to the *OL* problem we introduced. However, there are some significant differences. First, in the *OL* problem, only one new site is to be opened; in the *FL* problem, multiple sites may be opened to reduce the service costs. It makes the *OL* problem, in general, not NP-hard. Therefore, we aim to find exact answer to the *OL* problem, instead of approximate answer. Second, in the *OL* problem, the new site can be anywhere (within the query range); in most variants of the *FL* problem, the locations of the new sites are chosen from the given candidates. Finally, in the *OL* problem, we measure the cost by simply using the distance between the site and the object instead of in arbitrary (unmetrical) ways. This specialization makes it possible to find more efficient solutions for the *OL* problem. We notice that there are some studies on the *FL* problem with metric costs [13, 44, 30, 48] or with unrestricted site locations [1, 24]. Nevertheless, these studies focus on the theoretical aspects of the *FL* problem. Their algorithms are approximation algorithms and seem to be still too slow for practical purposes. Besides the differences mentioned above, the existing algorithms for the *FL* problem assumes the data is stored in-memory. When the data volume is huge, the data is usually indexed by some index structures as we reviewed in section 2.1. These algorithms need to be externalized or redesigned to utilize the indices.
Chapter 3

Max-Inf OL under $L_1$ and $L_2$ Metric

In this chapter, we study the Max-Inf OL queries under $L_1$ and $L_2$ metric. Starting with $L_1$ case, we will first illustrate how to transform the Max-Inf OL query into a problem of finding a location with maximum overlap among some squares, then go to the details of our methods. Specifically, we propose two practical methods that accurately compute Max-Inf OLs under $L_1$ metric. The first solution is an R*-tree based solution. The second solution is based on a new specialized aggregation index called the VOL-tree. Finally, we extend the solutions to $L_2$ metric.

3.1 Problem Transformation

To efficiently compute an optimal location, we first define the concept of $nn\_buffer$, and then transform the optimal-location query into the problem of finding a location with maximum overlap among objects’ $nn\_buffers$. Formally, the $nn\_buffer$ of an object is defined as follows.

Definition 3.1.1. Given an object $o$ and its closest site $s$, the $nn\_buffer$ of $o$ is a
contour such that $\forall l$ on the contour, $d(l, o) = d(o, s)$.

Figure 3.1: The nn_buffer of an object is a diamond.

In other words, a location $l$ is inside $o.nn_buffer$ if and only if $o$ is closer to $l$ than to any site. As shown in Figure 3.1, the nn_buffer of an object $o$ under $L_1$ metric is a diamond. The object $o$ contributes to the influence of a location $l$, if and only if $l$ is inside the nn_buffer of $o$. Therefore, given a query region $Q$, an optimal location is a location $l$ inside $Q$ which maximizes the total weight of overlapped nn_buffers. Figure 3.2 gives an example of the query with four objects and two sites and its corresponding transformation. Note that there may be more than one optimal location. The query only asks for one of them.

Figure 3.2: In (a), $l$ is an optimal location, with influence 2. The transformation in (b) shows that any location in the intersection between $Q$ and region 2 is an optimal location.

Consider the coordinate which has the same origin as in the original coordinate, but whose X and Y axes are rotated 45° counter-clockwise. We call it the 45°-rotated coordinate (Figure 3.3). Our analysis shows that an object $o$ located at $(x, y)$ in the original coordinate is mapped to $(\frac{x+y}{\sqrt{2}}, \frac{-x+y}{\sqrt{2}})$ in the 45°-rotated coordinate.
Furthermore, let $t$ be the $L_1$ distance from $o$ to its closest site. The $nn\_buffer$ of $o$ in the 45°-rotated coordinate is an axis-parallel square, whose lower-left corner and upper-right corner are: $(\frac{x+y-t}{\sqrt{2}}, \frac{-x+y-t}{\sqrt{2}})$ and $(\frac{x+y+t}{\sqrt{2}}, \frac{-x+y+t}{\sqrt{2}})$.

![Figure 3.3: Illustration of the rotated coordinate.](image)

In this thesis, the R*-tree indexes the objects in the original coordinate (to satisfy the possible need for other applications), while the aSB-tree, the OL-tree and the VOL-tree indexes the $nn\_buffers$ in the 45°-rotated coordinate.

### 3.2 The R*-tree-based Solution

#### 3.2.1 Overview

Our first solution assumes that the objects are indexed by an R*-tree. Similar to how the Rdnn-tree[52] extends the R*-tree, we assume every object stores the $L_1$ distance to its closest site, and every index entry stores the maximum $L_1$ distance of objects in the sub-tree. The R*-tree indexes objects in the original coordinate, since there may be other applications that need to access the data in the original coordinate. The solution follows two steps. The first step is to retrieve from the R*-tree those objects which may affect the influence of some locations in $Q$. As the second step, a plane-sweep algorithm goes through the stream of objects once and identify an optimal location. The only objects that may affect the influence of locations in the query region $Q$ are the ones whose $nn\_buffers$ intersect with $Q$. Given an object $o$,
we denote by $o.x_{low}$ ($o.x_{high}$) the the smallest (largest) $x$-coordinate value of its $nn\_buffer$. Our approach retrieves such objects in increasing order of their $x_{lows}$ in the rotated coordinate, even though the R*-tree was built in the original coordinate. This enables the run-time plane sweep to be efficient. A naive plane-sweep solution has $O(n^2)$ cost, where $n$ is the number of $nn\_buffers$ in the stream. We propose the aggregation SB-tree (aSB-tree), extended from the SB-tree [53], to reduce the query cost to $O(n \log n)$.

In the rest of this section, subsection 3.2.2 shows how to retrieve objects, subsection 3.2.3 describes a naive plane sweep algorithm with $O(n^2)$ cost, subsection 3.2.4 reviews the segment-tree [8] and the SB-tree [53] as the preliminaries of aSB-tree, subsection 3.2.5 proposes the aSB-tree structure which can reduce the worst-case query cost to $O(n \log n)$, and subsection 3.2.6 extends the algorithms to incorporate a rotated query region.

### 3.2.2 Retrieving Objects from The R*-tree

To retrieve the objects whose $nn\_buffers$ intersect with $Q$, we can browse the R*-tree in a top-down fashion, similar to the range query. The difference is that to determine whether to expand an sub-tree, instead of checking whether its MBR intersects with $Q$, we check whether the MBR’s $nn\_buffer$ intersects with $Q$.

The remaining issue of object retrieval is how to return objects in increasing order of their $x_{lows}$ in the 45°-rotated coordinate. This is achieved by using a best-first search. That is, we keep a heap of the R*-tree’s index entries as well as objects. The entries are ordered in increasing $x_{low}$ in the 45°-rotated coordinate. Initially, the heap contains the index entry referencing the whole tree. In each iteration, the entry $e$ with minimum $x_{low}$ is extracted. If $e$ is an object, output it (to be sent, as the next element of an input stream, to the plane-sweep algorithm discussed in the next section). Otherwise, $e$ is an index entry. We examine every entry $se$ in the node
referenced by \( e \), and push \( se \) into the heap if its \( nn\text{-buffer} \) intersects with \( Q \).

### 3.2.3 The Naive Plane Sweep

![Figure 3.4: \( nn\text{-buffers} \) in the rotated coordinate.](image)

In the rotated coordinate, the \( nn\text{-buffers} \) are axis-parallel squares. To find the optimal location, the basic idea is to perform a plane sweep in increasing order of \( X \). For each particular \( X \), the \( Y \) axis is partitioned into a set of intervals, each associated with an influence value. For instance, in Figure 3.4, when \( X < 1 \) the \( Y \) axis is a single interval: \([-M, M] : 0\). Here \([-M, M]\) is the domain in \( Y\)-axis. The value 0 means before we see the first \( nn\text{-buffer} \), the influence of all locations are 0.

When the sweep line moves forward, two type of events may happen: the \textit{entering} event, which happens when the sweep line enters a \( nn\text{-buffer} \), and the \textit{leaving} enter, which happens when the sweep line leaves a \( nn\text{-buffer} \). The events trigger the update of the intervals and the corresponding influences. Whenever an entering event happens, some intervals may break into 2 intervals, and some intervals may increase their influences. for example, when \( X = 1 \), the \( Y \) axis is broken into three pieces: \([-M, 5], [5, 9] \) and \([9, M]\), and their influences are 0, 1 and 0 respectively. When \( X = 2 \) the \( Y \) axis is further broken into four intervals. Whenever a leaving event happens,
some intervals may decrease their influences and merge with other intervals.

During the process, we maintain the maximum influence seen so far along with a
rectangle, which reaches the maximum influence. If there are multiple rectangles with
the same maximum influence, one of them are selected arbitrarily. After all entering
events been processed, any location in the maintained rectangle is an optimal location.
In Figure 3.4, any location in the $X$ range of $(4,5)$ and $Y$ range of $(5,8)$ is an optimal
location, with influence 3.

The naive plane sweep has $O(n^2)$ worst-case performance. The reason is that, in
the worst case, there are $O(n)$ events to handle, and each event needs to scan through
$O(n)$ intervals. We hereby propose a data structure called the Aggregation SB-tree
($aSB$-tree). The new structure enables any event to be processed in $O(\log n)$ time,
and therefore reduces the overall cost to $O(n \log n)$. The $aSB$-tree is derived from and
shares many features with the SB-tree [53], which combines both the segment-tree [8]
and the B-tree [5]. Therefore, before going to the detail of $aSB$-tree, let us briefly
review the segment-tree and SB-tree.

3.2.4 Preliminaries

The segment-tree is originally proposed to answer the stabbing query, i.e., given a
set $I$ of intervals and a query point $q$, to find all intervals in $I$ that contain $q$. For
example, the intervals could be the span of all phone calls, and the query is to find
“all calls were made at time $X$”. The segment-tree is organized in the following way:

- The skeleton of the segment-tree is a balanced binary tree, each node $n$ of which
corresponds to an interval $I(n)$.

- For a leaf node, the corresponding interval is an elementary interval introduced
by two consecutive (but distinct) endpoints of the intervals in $I$. For an inter-
mediate node, the corresponding interval is the union of the intervals of its two
Each node $n$ stores the corresponding interval $I(n)$. Meanwhile, (the id of) each interval $i$ in $I$ is stored in some nodes that together cover $i$. To make the storage requirement as small as possible, the interval is stored at the nodes as high as possible. More precisely, (the id of) an interval $i \subseteq I$ is store at node $n$ if and only if $I(n) \subseteq i$ but $I(parent(n)) \not\subseteq i$.

Consider a set $I$ containing 4 intervals $[1, 5], [4, 7], [4, 9]$, and $[8, 11]$, which are labeled as $i_1$ to $i_4$ respectively. Figure 3.5 shows the corresponding segment-tree. The squares and the circles represent the leaf nodes and the intermediate nodes respectively. Each leaf node corresponds to the interval right below it, assuming the domain is $[-M, M]$. Starting from the leftmost leaf node, their corresponding intervals are $[-M, 1], [1, 4], [4, 5]$ and so on. The intervals aggregate up to the root. For instance, the leftmost node on level 1 (assuming leaf level is level 0) has interval $[-M, 4]$ and the leftmost node on level 2 has interval $[-M, 7]$. The root has the interval of the whole domain. The arrows point to the set of intervals stored in the nodes. As we explained, $i_3$ is stored in 2 nodes on level 1 instead of 4 nodes on level 0. To answer the query, we do a binary search on the segment-tree and report all intervals stored in the nodes along the search path.

Figure 3.5: Example of segment-tree.
The segment-tree is a memory-based data structure and has a $O(n \log n)$ space complexity. Yang et. al. externalize it into SB-tree to efficiently answer the aggregation query, which only asks for the number of intervals containing the query point. The idea is to organize the intervals into a balanced B-tree-like structure. Meanwhile, the list of intervals associated with each node is replaced a single value $cover$, which is the cardinality of the list. In particular,

- Each node in the SB-tree consists of certain entries. The capacity (i.e. the maximum number of entries) in a (leaf or index) node is fixed. Except the root, every node must be at least half full.

- Every entry $e$ stores an interval $I(e)$. For a leaf entry, the corresponding interval is an elementary interval introduced by two consecutive (but distinct) endpoints of the intervals in $I$. For an intermediate entry, the corresponding interval is the union of the intervals of its children. The root entry has an interval as the whole domain.

- Every entry $e$ also stores a value $cover(e)$. It equals to the number of intervals in $I$ that cover $I(e)$ but not cover $I(parent(e))$, i.e. $cover(e) = |\{i | i \subseteq I \land I(e) \subseteq i \land I(parent(e)) \not\subseteq i\}|$.

![Figure 3.6: Example of segment-tree.](image)

Figure 3.6 shows a SB-tree corresponding to the segment-tree in Figure 3.5. In general, the SB-tree takes $O(n)$ space, where $n$ is the total number of intervals in $I$. 31
To answer the (aggregate) query, we do a \( n \)-nary search on the SB-tree and report sum of the \( \text{cover}(\cdot) \) values associated with the entries along the search path. Therefore, the query can be answer within \( O(\log n) \) time. To insert a new interval \( i \), we update (at most) two paths from the root to the leaf level. These two paths include the nodes whose referencing entry’s interval contains one of the endpoints of \( i \). If an entry \( e \) is found such that its interval \( I(e) \) is contained in \( i \), \( \text{cover}(e) \) is increased by 1 and sub-tree of \( e \) does not need to be visited. For example, in Figure 3.6 the insertion of \([4, 8]\) stops at the root node by updating the second entry to \([4, 8] : 2\). The insertion of \([3, 10]\) follows two paths. One path is from the root entry down to the entry \([1, 4] : 1\), which is broken into two entries \([1, 3] : 1\) and \([3, 4] : 2\). The other is from the root entry down to the entry \([9, 11] : 1\), which is broken into two entries \([9, 10] : 2\) and \([10, 11] : 1\). Meanwhile, the entries \([4, 8] : 1\) is updated to \([4, 8] : 2\) and its subtree is not visited. The deletion is treated as an insertion with value -1. An overflow/underflow\(^1\), if happens, is treated like in the B-tree. Nevertheless, the insertion and deletion can be done within \( O(\log n) \) time.

### 3.2.5 The aSB-tree

The SB-tree can be used to maintain the intervals (on \( Y \) axis) in the naive plane sweep algorithm in the following way: when the sweep line enters or leaves an \( \text{nn\_buffer} \), an interval corresponding to the \( Y \) range of the \( \text{nn\_buffer} \) is inserted into or deleted from the SB-tree. It can cut the maintenance cost from \( O(n^2) \) to \( O(n \log n) \). However, there is a gap between what the SB-tree provides and what we need. Our ultimate goal is: after all the \( \text{nn\_buffers} \) are seen, report an optimal location with its influence. In order to achieve this, we extend the SB-tree into the aSB-tree by maintaining a local maximum for each entry and a global maximum for the whole tree.

Figure 3.7 shows a two-level aSB-tree, which corresponds to Figure 3.4 right after

\(^1\)Underflow may happen because the adjacent leaf entries with the same \( \text{cover}(\cdot) \) value can be merged into one entry.
processing the event at $X = 4$ and the event at $X = 5$. The gray areas indicate the values updated after processing the event at $X = 5$. Let’s examine the aSB-tree in more details.

- Each index entry $e$ in the aSB-tree stores the local maximum influence (at current point) $maxI(e)$ within the interval $I(e)$. It is local because the actual influence is the local influence plus the sum of the cover values of $e$ and its ancestors. For instance, in the second index entry in Figure 3.7(b), the local maximum influence is 3. But the actual maximum influence is $3 + (-1) + 0 = 2$. At first glance, it seems not right to have an entry with negative cover value. However, there is nothing wrong. It just implies that, within the life span of this entry, there are more deletions whose interval covers $I(e)$ (but not $I(parent(e))$) than that type of insertions.

- Along with local maximum influence, each index entry $e$ also stores the corresponding spatial region $maxR(e)$. Any location in this region has the local maximum influence at current point. The region only has the left $X$ border...
while the right border is open. To elaborate the reason, let us look into the future and consider the local maximum after the next event. If the local maximum influence keeps unchanged, the current local maximum is still valid. Otherwise, the current local maximum need to be replaced by a new one with unknown right border. In any case, it is safe to keep the right border open. In Figure 3.7(b), the second entry in the root node shows the first case. Event though the cover value in the entry is changed, the local maximum keeps unchanged. The root entry shows the second case. The local maximum influence decreases due to the leaving event.

- Besides the local maximum for each entry, the aSB-tree also keeps a globally maximum influence and its spatial region. The global maximum is maintained after each insertion. The local maximum in the root entry is checked against the current global maximum. If the former is larger, it replaces the current global maximum; otherwise, no update is needed. Similar to the local maximum, when the global maximum is just found, the right border of its region is open. However, in order to return a valid answer to the query, it needs to be closed at some point later. It occurs at the next deletion after current event. In Figure 3.7, the global maximum region has an open right border when it is found at $X = 4$ and is closed at $X = 5$ due to the leaving event.

- Similar to SB-tree, the insertion of an interval follows (at most) two paths from the root to the leaf level to update the cover values accordingly. Afterwards, it tracks backwards along each path to update the local maximum. Consider an entry $e$ on the path. Let $n(e)$ be the node pointed to by $e$ and $e_i$ be the entry in $n(e)$ which has the largest $\maxI() + \text{cover}()$ (or the largest $\text{cover}()$ if $n(e)$ is a leaf node) among all entries in $n(e)$. The local maximum influence $\maxI(e)$ is updated to $\{\maxI(e_i) + \text{cover}(e_i)\}$ (or $\text{cover}(e_i)$ if $n(e)$ is a leaf node). The
Y-range of \( \text{maxR}(e) \) is updated to the Y-range of \( \text{maxR}(e_i) \) (or \( I(e_i) \) if \( n(e) \) is a leaf node). The left border of \( \text{maxR}(e) \) (i.e. the starting point of the X-range) is updated to the current value of \( X \). This rule is recursively applied until the local maximum of the root is updated.

To answer the OL-query, a aSB-tree is used to maintain the intervals seen in the plane sweep algorithm. After the last entering event, the global maximum is returned as the answer. As each insertion or deletion in the aSB-tree is \( O(\log n) \) as in the SB-tree, the plane-sweep algorithm integrated with the aSB-tree has \( O(n \log n) \) query cost.

There are a few things need to be clarified. First, it is possible that multiple events happen at the same \( X \). In this case, we first process all the leaving events (in arbitrary order), and then process the entering events. Otherwise, an incorrect (larger) global maximum may be found. Second, in the aSB-tree the adjacent leaf entries wont merge event if they have the same cover value, i.e. there is no underflow. It is also because of the correctness. Consider a very extreme example which has only two \( \text{nn_buffers} \) \([0, 1] \times [0, 1]\) and \([0, 1] \times [1, 2]\). If we allow underflow, the merged region \([0, 1] \times [0, 2]\) will be reported as the global maximum. Any point within it can be returned as OL. However, according to the definition, the points on \(([0, 1], [1, 1])\), the shared edge of these two \( \text{nn_buffers} \), are not OL. It is imaginable that, without underflow, the performance of aSB-tree deteriorates. However, the asymptotic cost is still the same. Last, even though we use the aSB-tree as an in-memory structure to improve the plain sweep, if needed the structure can be implemented as a disk-based index.

### 3.2.6 Involving A Query Region

In the original coordinate, the query region \( Q \) is an axis-parallel rectangle. Thus, in the 45°-rotated coordinate, the query region \( Q \) becomes a rectangle rotated 45°
clockwise (as shown in Figure 3.8). To perform the query correctly, our aSB-tree based plane-sweep algorithm needs to be modified as follows.

First, the domain of $Y$ dimension is not the whole space $(-M, M)$, but the $Y$ projection of $Q$. This is because we only care about the locations in $Q$. In Figure 3.8, the $Y$ space of the aSB-tree should be $[y_l, y_h]$. Note that, the $Y$-range of $Q$ is part of this domain and changes continuously when the sweep line moves forward.

Second, for each $nn$ buffer, the entering/leaving event happens when it enters/leaves $Q$, instead of at the left and right edges of the $nn$ buffers. In Figure 3.8, $e_1$ and $l_1$ are the events of the first $nn$ buffer, while $e_2$ and $l_2$ are the events of the second $nn$ buffer. The insertions and deletions still occur at these events. However the interval to be inserted or deletion is truncated into $[y_l, y_h]$. In Figure 3.8, the interval for event $e_1$ and $e_2$ will be $[y_l, y_{h1}]$ and $[y_l, y_{h2}]$ respectively.

Third, extra effort is needed to find the maximum influence within $Q$. Our plane sweep algorithm essentially keeps tracking the maximum influence along the sweep line at any $X$. Without the query region, it can be done by simply checking the root entry. It it is no longer true when $Q$ involves. The reason is that maximum influence maintained in the root entry may be outside $Q$. In Figure 3.8, the maximum region in the root entry at $X = e_2$ is the range of $(y_l, y_{h2})$, which is outside $Q$. To address this issue, we perform a range-max query on the aSB-tree after each insertion. That

![Figure 3.8: Illustration of a rotated query region.](image)
is to find the maximum influence within the $Y$ range of $Q$.

**Algorithm** \( \text{RangeMax}(e, Y_Q) \)

**Input:** An index entry $e$ and an interval $Y_Q$.

**Output:** The maximum influence within $Y_Q$ and the corresponding range on $Y$.

1. if $e$ is a leaf entry
2. return \((\text{Cover}(e), I(e) \cap Y_Q)\).
3. if \(Y_Q \cap \text{maxR}_{Y}(e) \neq \emptyset\), i.e. $Y_Q$ intersects with the $Y$-range of \(\text{maxR}(e)\)
4. return \((\text{Cover}(e) + \text{maxI}(e), Y_Q \cap \text{maxR}_{Y}(e))\).
5. Get the node $n(e)$ pointed to by $e$.
6. for each entry $e_i$ in $n(e)$ such that $I(e_i) \cap Y_Q \neq \emptyset$
7. \((\text{RangeMaxI}_i, \text{RangeMaxY}_i) = \text{RangeMax}(e_i, Y_Q)\).
8. Find the entry $e_j$ such that $\text{RangeMaxI}_j = \max\{\text{RangeMaxI}_i\}$.
9. Return \((\text{RangeMaxI}_j + \text{cover}(e), \text{RangeMaxY}_j)\).

Figure 3.9: The range-max query procedure.

The range-max query is processed by call the procedure in Figure 3.9 with the parameter $e$ being the root entry of the asB-tree (the setting of parameter $Y_Q$ will be explained shortly). Similar to the insertion, the range-max query follows (at most) two paths down to the leaf and traces backward to report the range-max. It can be performed in $O(\log n)$ time. It is trickier than we thought to pass the correct parameter $Y_Q$ to \(\text{RangeMax}\) procedure. Simply setting $Y_Q$ to the $Y$-range of $Q$ at current $X$ does not work. For example, if this (wrong) rule were used in Figure 3.8, the result would be incorrect. In particular, the range-max influence after each event would be 0, 1, 1, 0, respectively. Therefore, the result would have influence 1. However, it is obvious that the correct result has influence 2. The reason is that, the region with maximum influence is not a triangle and starts from $e_2$. It won't be discovered if the range-max query is issued at $e_2$. To fix this problem, the range-max query should be issued slightly after $e_2$ (but not too late so the next event won't be passed). In our implementation, the range-max query is issued at the middle of current event and the next event. If multiple events happen at the same place, only one range-max query is issued after all these events are processed.
Algorithm *RotatedOL*(R, Q)

**Input:** The root R of an R-tree and a query region Q.

**Output:** The OL.

1. Initialize a empty aSB-tree rooted at T.
2. Retrieve all nn buffers intersecting with Q.
3. Calculate the entering and leaving event of each retrieve nn_buffer.
4. **while** there are entering events need to be processed.
   (a) Get next event e (from left to right).
   (b) Insert/delete the corresponding interval into/from T.
   (c) if e is an entering event and differs from the next event e’ on X
   (d) \( \text{CurrentMax} = \text{RangeMax}(T, \frac{X(e)+X(e')}{2}) \).
   (e) update the global max in T if CurrentMax is better.
5. **return** Any point within the global max in T.

Figure 3.10: The OL query algorithm.

To recap, Figure 3.10 gives the high level description of the OL query algorithm.

### 3.3 The Virtual OL-tree

The R*-tree based solution examines all objects whose nn buffers intersect with the query region Q, and thus is not efficient when a large Q results in the examination of many objects. This section first proposes an theoretical solution to the optimal-location query based on a new index structure called the *Optimal-Location Tree (OL-tree)*. Then we extend it to a more practical and efficient solution based on the *Virtual Optimal-Location Tree (VOL-tree)*.

#### 3.3.1 The OL-Tree

The OL-tree is a k-d-B-tree-like structure which is balanced, disk-based and dynamically-updateable. Roughly speaking, it stores the nn buffers in the 45°-rotated coordinate. Like the k-d-B-tree, the OL-tree is a space-partitioning method (versus a data-partitioning method like the R*-tree). Unlike the k-d-B-tree, the OL-tree stores rect-
angular records in its leaf nodes. If a nn_buffer partially intersects with the ranges of multiple index entries, it is split and multiple fragments are inserted. However, if the square fully contains the range of some index entry, we only update a value called fullcover stored along with the index entry, without further inserting into the sub-tree.

An index entry e in the OL-tree has the following format: (range, nodeID, fullcover, maxoverlap, maxrange). Here range is the spatial range of the corresponding sub-tree, and nodeID points to the referenced node. The value fullcover is the total number of nn_buffers whose insertion stopped at e (such a nn_buffer contains e.range, but not the range of e’s parent). The value maxoverlap is the maximum local influence of locations in the sub-tree. It is local because we only consider the nn_buffers that were inserted into the sub-tree. In other words, the nn_buffers that fully contain e.range do not contribute to e.maxoverlap. The rectangle maxrange is the corresponding range. Any location in e.maxrange has local influence e.maxoverlap. Figure 3.11 is an example of (part of) a 3-level OL-tree. Each tuple gives (nodeID, fullcover, maxoverlap) of the corresponding entry, while each small black rectangle represents the maxrange. The update and query processing of the OL-tree are very similar to those of the VOL-tree, which we will present very soon. Therefore, we skip the details here.

The OL-tree may cause cascading split of child nodes if splitting an index node. We argue that the space complexity of the OL-tree in the worst case is $O(n^2/B)$, where B is the page capacity. The reasons are as follows. First, the total number of leaf entries is $O(n^2)$. With n axis-parallel squares, there are $O(n)$ different X positions and $O(n)$ different Y positions, which form $O(n^2)$ cells. In the worse case each cell is stored in the OL-tree separately. Thus there are at most $O(n^2)$ leaf entries. Second, the total number of nodes is $O(n^2/B)$. The linear storage of the k-d-B-tree can be guaranteed by re-organization of sub-trees which contain too few leaf entries.
Similarly, the OL-tree with $O(n^2)$ leaf entries needs $O(n^2/B)$ nodes.

This worst case bound reveals that the OL-tree is not a practical spatial index structure. In the next subsection we introduce a practical structure, named the VOL-tree, to solve the optimal-location query.

### 3.3.2 The VOL-tree Structure

The OL-tree has high space complexity because if an \textit{nn_buffer} is split into multiple pieces, each of them is physically stored in some leaf node. What if we do not physically store any leaf node of the OL-tree? We can use an R*-tree to store the original objects, and whenever the content of a leaf node is needed, we perform a range query on the R*-tree. This is the key idea to the Virtual OL-tree (VOL-tree).

It is challenging to implement this idea. As we already spend the space to store the R*-tree, it is ideal to have a small VOL-tree that fits in memory. On the other hand, as there are $O(n^2/B)$ leaf nodes in an OL-tree, there are $O(n^2/B^2)$ index nodes, which would be the size of the VOL-tree if we only trim off the leaf nodes from the OL-tree. There is a big gap. Thus the VOL-tree can NOT merely be an OL-tree without the leaf level. It has to be much smaller, possibly only consisting of one
or two index level besides the root. A consequence is that each leaf entry of the VOL-tree corresponds to a virtual node (content stored in the R*-tree) with much more than \( B \) \( nn\_buffers \). So a crucial issue jumps out: it is expensive to maintain \( maxrange \) and \( maxoverlap \) because an update requires us to perform plane sweeps on many \( nn\_buffers \).

To address this issue, we propose another change from the OL-tree: along with each index entry, instead of keeping the accurate \( maxoverlap \), keep two values \( lowermax \) and \( uppermax \), which are a lower bound and an upper bound of \( maxoverlap \).

In more detail, the entries in the tree are as follows:

- An index entry \( e \) has the following format: \((range, nodeID, fullcover, lowermax, maxrange, uppermax)\). Here \( range \) is the spatial range of the corresponding sub-tree, and \( nodeID \) points to the referenced node. The value \( fullcover \) is the total weight of \( nn\_buffers \) whose insertion stopped at \( e \) (such a \( nn\_buffer \) contains \( e\_range \), but not the \( range \) of \( e \)'s parent).

- The values \( lowermax \) and \( uppermax \) are some lower and upper bounds of the maximum local influence in \( e\_range \). And \( maxrange \) is a rectangle fully contained in \( e\_range \) where every location in \( maxrange \) has local influence = \( lowermax \).

- A leaf entry and an index entry have the same content, with a minor difference that a leaf entry’s \( nodeID \) is empty.

Figure 3.12 shows an example of VOL-tree. Each tuple gives \((nodeID, fullcover, lowermax, uppermax)\) of the corresponding entry, while each small black rectangle represents the \( maxrange \).
3.3.3 The VOL-tree Query Algorithm

Figure 3.13 gives the optimal-location-query algorithm in the VOL-tree. We start from the root node. In the VOL-tree, even if root.maxrange intersects with $Q$, it is possible that some location in $Q - root.maxrange$ has an influence larger than root.lowermax (when root.lowermax < root.uppermax). So as Step 1 shows, we can safely return a location in root.maxrange $\cap Q$ only if root.lowermax = root.uppermax or $Q$ is completely inside root.maxrange.

Step 2 inserts the root entry into a heap. Every entry in the heap has, besides an index entry, two values cover and upper. The cover value carries the sum of fullcover from the ancestors. The upper gives the actual (not local) upper bound of influence for locations in the sub-tree. Meanwhile, we maintain the currently seen optimal location opt.loc along with its influence opt.inf, initialized to be an arbitrary location with influence 0 (Step 3).

While the heap is not empty, we process each element at a time. In each iteration, the heap entry with maximum upper is extracted. As Step 4(b) of the algorithm
Algorithm **VOLTreeQuery**

**Input:** Query region $Q$, VOL-tree root.

**Return:** An optimal location in $Q$.

1. **if** root.maxrange $\cap Q \neq \emptyset$ and (root.lowermax = root.uppermax or $Q \subseteq$ root.maxrange), **return** any location in root.maxrange $\cap Q$.

2. heap.Insert(root, 0, root.uppermax)

3. Set opt.loc as an arbitrary location in $Q$, and opt.inf = 0.

4. while heap is not empty
   
   (a) $(e, cover, upper) =$ heap.ExtractMaxUpper().
   
   (b) **if** upper $\leq$ opt.inf, **return** opt.loc.
   
   (c) **if** $e$ refers to an intermediate node
      
      for every entry $se$ in Node(e.nodeID) s.t. $se.range \cap Q \neq \emptyset$
      
      A. Set $m =$ cover $+$ $se.fullcover$, and $u =$ cover $+$ $se.fullcover$ $+$ se.uppermax.
      
      B. **if** $u \leq$ opt.inf, **goto** next entry.
      
      C. **if** opt.inf $< m$, set opt.inf $= m$ and opt.loc be any location in $se.range \cap Q$.
      
      D. **if** $se.maxrange \cap Q \neq \emptyset$,
         
         (i) $l =$ cover $+$ $se.fullcover$ $+$ se.lowermax
         
         (ii) **if** opt.inf $< l$, set opt.inf $= l$ and opt.loc be any location in $se.maxrange \cap Q$.
         
         (iii) **if** $u \neq l$ and ($Q \cap se.range$) $\not\subseteq$ se.maxrange, heap.Insert(se, m, u)
      
      E. **else**
         
         heap.Insert(se, m, u)
   
   (d) **else** /* $e$ refers to a virtual leaf node */
      
      A. Using $e.range \cap Q$ as a new query region, retrieve $nn.buffers$ from the R*-tree of objects. Use plane sweep to find an optimal location ($inf$, loc) within the new query region.
      
      B. **if** opt.inf $< cover + inf$, set opt.loc = loc, and opt.inf $= cover + inf$.

5. **return** opt.loc.

---

Figure 3.13: Finding an optimal location using the VOL-tree.

shows, if this extracted upper is no larger than opt.inf, we can determine that opt.loc is an optimal location and thus the algorithm returns. The crucial steps are Step 4(c) which expands an index node and Step 4(d) which expands a leaf node.

To expand an index node, we examine every child entry $se$ whose range intersects with $Q$, and try to push $se.nodeID$ into the heap. Here the new lower bound is $m =$ cover $+$ $se.fullcover$, and the new upper bound is $u =$ cover $+$ $se.fullcover$ $+$ se.lowermax. There are two pruning opportunities. First, if the new upper bound $u$ is no larger than opt.inf, there is no need to expand the sub-tree (Step 4(c)B). Second, if $se.maxrange$ intersects with $Q$, we already know the influence of the locations within the intersection, and thus we may have the chance to update the maintained
optimal location before expanding the sub-tree (Step 4(c)). It likely causes other entries to be pruned earlier.

To expand a leaf node (Step 4(d)), we go to the R*-tree to retrieve the nn_buffers that intersect with e.range \( \cap Q \) and then perform a plane sweep technique of Section 3.2 to compute a location inside \( Q \) with maximum global influence \( \inf \). If this influence is bigger than \( \text{opt}_\inf \), we update the maintained \( \text{opt}_\inf \) and \( \text{opt}_\text{loc} \).

Consider the example in Figure 3.12, where the gray rectangle is the query region \( Q \). Initially, \((n_{\text{root}}, 0, 12)\) is pushed into the a heap, which uses the last field of the tuple as the sorting key. After the root node is visited, \( n_2 \) and \( n_3 \) are pushed into the heap as \((n_2, 1, 6)\) and \((n_3, 2, 12)\). The current candidate is some location from the maxrange of \( n_2 \) (with influence 5). In the next step, \( n_3 \) is extracted and \( n_{31}, n_{32} \) and \( n_{33} \) are checked. The entry \( n_{31} \) has \( \text{lowermax} = \text{uppermax} \) and its maxrange is fully contained by \( Q \). Therefore, it does not need to be pushed into the heap. The current candidate is replaced by a point in the maxrange of \( n_{31} \) with influence being 2+2+4=8. Since \( \text{lowermax} \) and \( \text{uppermax} \) of \( n_{32} \) are different, \((n_{31}, 5, 12)\) will be pushed into the heap. Since the maxrange of \( n_{32} \) does not intersect with \( Q \), the candidate remain unchanged. The entry \( n_{33} \) is skipped in that the upper bound of the influence in \( n_{33} \) is 2+1+4=7, which is less than the influence of current candidate. Then \( n_{32} \) is extracted from the heap. Since it is the level node, plain sweep algorithm is called to compute the OL within the intersection of \( Q \) and \( n_{32} \). If the result has influence larger than 8, the it is returned as the answer; otherwise, the current candidate will be returned.

### 3.3.4 The Update Algorithm

The update algorithm is shown in Figure 3.14. A new \( \text{nn_buffer} \) is always inserted the root of the VOL-tree with weight 1, while a deletion is treated as an insertion with weight -1. To insert into an index node (Step 1), we consider every child entry
Algorithm \textit{VOLTreeInsert}

Input: Range \( R \), Weight \( w \), VOL-tree index entry \( e \).

Pre-condition: \( R \) intersects with, but does not fully contain, \( e.\text{range} \).

Action: Insert range \( R \) with weight \( w \) to the sub-tree referenced by \( e \).

1. if \( e \) refers to an intermediate node in the VOL-tree
   (a) for every \( se \) in \( \text{Node}(e) \) s.t. \( R \) contains \( se.\text{range} \), \( se.\text{fullcover} += w \).
   (b) for every \( se \) in \( \text{Node}(e) \) s.t. \( R \) partially intersects with \( se.\text{range} \), VOLTreeInsert(\( R \), \( w \), \( se \)).
   (c) Let \( se_0 \) be the entry in \( \text{Node}(e) \) with maximum \( se.\text{fullcover} + se.\text{lowermax} \).
   (d) Set \( e.\text{maxrange} = se_0.\text{maxrange} \) and \( e.\text{lowermax} = se_0.\text{fullcover} + se_0.\text{lowermax} \).
   (e) \( e.\text{uppermax} = \max\{se.\text{fullcover} + se.\text{uppermax}\} \) for all entry \( se \) in \( \text{Node}(e) \).

2. else /* \( e \) refers to a virtual leaf node */
   (a) if \( w > 0 \), \( e.\text{uppermax} += w \).
   (b) if \( e.\text{maxrange} \cap R = \emptyset \), \textbf{return}.
   (c) if \( w > 0 \)
      i. \( e.\text{maxrange} = e.\text{maxrange} \cap R \)
      ii. \( e.\text{lowermax} + = w \)
   (d) else
      i. if \( e.\text{maxrange} \subseteq R \), \( e.\text{lowermax} + = w \).
      ii. if \( e.\text{maxrange} \not\subseteq R \), \( e.\text{lowermax} = e.\text{maxrange} - R \).

Figure 3.14: The Insertion algorithm of the VOL-tree.

\( se \) whose \( \text{range} \) intersects with the parameter \( R \). If \( se.\text{range} \) is contained in \( R \), we simply add \( w \) to \( se.\text{fullcover} \). If \( se.\text{range} \) partially intersects with \( R \), we recursively insert into the sub-tree referenced by \( se \). After insertion, we need to re-aggregate the \( \text{lowermax} \), \( \text{uppermax} \) and \( \text{maxrange} \) if necessary.

When \( e \) refers to a virtual leaf node which is not stored, the actual object is maintained in a separate R*-tree. So we only need to modify \( e.\text{lowermax} \), \( e.\text{uppermax} \) and \( e.\text{maxrange} \). The update of \( e.\text{uppermax} \) is simple. As Step 2(a) shows, for a positive weight, \( e.\text{uppermax} \) is increased by \( w \). For a negative weight, \( e.\text{uppermax} \) remains unchanged. We may modify \( e.\text{lowermax} \) and/or \( e.\text{maxrange} \) only if \( e.\text{maxrange} \) intersects with \( R \). For a positive weight (Step 2(c)), the intersection part of \( e.\text{maxrange} \) and \( R \) is the new \( e.\text{maxrange} \), with weight increased by \( w \). For a negative weight (Step 2(d)), there are two cases. If \( e.\text{maxrange} \) is fully covered by \( R \), we decrease \( e.\text{lowermax} \). Otherwise, we shrink \( e.\text{maxrange} \) to \( e.\text{maxrange} - R \) but keep
Figure 3.15: The VOL-tree after insertion.

*e.lowermax unchanged.*

Figure 3.15 shows the VOL-tree in Figure 3.12 after inserting a new *nn_buffer* \( R \) represented by the gray rectangle. The gray fields in the tuples are the fields which is changed after insertion. Since \( R \) contains the *maxrange* of \( n_{31} \) and \( n_{33} \), both *lowermax* and *uppermax* of \( n_{31} \) and \( n_{33} \) increase by 1. However, \( R \) does not intersect with the *maxrange* of \( n_{32} \), therefore only the *uppermax* of \( n_{32} \) increase. The *lowermax* of \( n_{32} \) remains the same. The *uppermax* of \( n_{32} \) propagates upwards and changes the *uppermax* of \( n_{3} \) and \( n_{root} \).

### 3.3.5 The Bulk-Loading Algorithm

In many applications, the datasets are known in advance. For instance, the set of McDonald’s stores and the set of residential buildings can be given in advance when building the index, although changes may happen later on. This section proposes a bulk-loading algorithm (Figure 3.16) which builds the VOL-tree, by browsing the \( \text{R}^* \)-tree of objects once. After bulk-loading, every entry in the VOL-tree has accurate
Algorithm \textit{VOLTreeBulkLoad}

Input: $R^*$-tree of objects, augmented with $L_1$ distances to the closest sites.

Action: Bulk load a VOL-tree using objects from the $R^*$-tree.

1. Determine the frame (i.e. space partitioning) of the VOL-tree via sampling.

2. \textbf{for} every left edge or right edge of the \texttt{nn\_buffers} from left to right
   \begin{enumerate}[(a)]
   
   \item Insert it into the aSB-tree.

   \item Adjust the \texttt{maxrange} and \texttt{lowermax} (=\texttt{uppermax}) for every leaf entry $e$ whose \texttt{range} intersects with the sweep line. Here \texttt{lowermax} stores the actual (not local) maximum influence in $\texttt{e.range}$, for the \texttt{nn\_buffers} seen so far.

   \item \textbf{if} this is a left edge, insert \texttt{nn\_buffer} top-down into the VOL-tree to update the \texttt{fullcover} values.

   \end{enumerate}

3. \texttt{AdjustAndPropagate}($\texttt{root}$ of the VOL-tree, 0).

Algorithm \textit{AdjustAndPropagate}

Input: An entry $e$ in the VOL-tree, a value \texttt{sum} as the total number of \texttt{nn\_buffers} containing $\texttt{e.range}$.

Precondition: Every entry in \texttt{Subtree($e$)} has the correct \texttt{fullcover}, and every leaf entry stores the correct \texttt{maxrange}, but \texttt{lowermax} (=\texttt{uppermax}) is the actual (not local) maximum influence.

Action: Make sure that in \texttt{Subtree($e$)}, each entry stores the correct \texttt{lowermax} (=\texttt{uppermax}) and \texttt{maxrange}.

1. \textbf{if} $e$ references to a node in the VOL-tree
   \begin{enumerate}[(a)]
   \item \textbf{for} every entry $se$ in \texttt{Node($e$.\texttt{nodeID})}
     \begin{enumerate}[(a)]
     \item \texttt{AdjustAndPropagate($se$, \texttt{sum} + $se$.\texttt{fullcover})}.
     \end{enumerate}
   \item Let $se_0$ be the entry in \texttt{Node($e$.\texttt{nodeID})} with maximum $se$.\texttt{fullcover} + $se$.\texttt{lowermax}.
   \item Set $e$.\texttt{maxrange} = $se_0$.\texttt{maxrange} and $e$.\texttt{lowermax} = $e$.\texttt{uppermax} = $se_0$.\texttt{fullcover} + $se_0$.\texttt{lowermax}.
   \end{enumerate}

2. \textbf{else} /* $e$ refers to a virtual node */
   \begin{enumerate}[(a)]
   \item $e$.\texttt{lowermax} = $e$.\texttt{uppermax} = $e$.\texttt{lowermax} – \texttt{sum}.
   \end{enumerate}

Figure 3.16: Bulk Load the VOL-tree.

local maximum information, i.e. \texttt{lowermax} = \texttt{uppermax}. In the next section we discuss how to perform dynamic updates.

The frame (i.e. space partitioning) of the VOL-tree can be created by sampling the data (Step 1 of Algorithm \textit{VOLTreeBulkLoad}). After this step, every entry has the correct \texttt{range} and \texttt{nodeID}. The crucial task is how to compute \texttt{fullcover}, \texttt{lowermax} (which is equal to \texttt{uppermax}), and \texttt{maxrange}. If there is only one leaf entry $e$, the task can be achieved using the plane sweep algorithm in Section 3.2. When there are more than one leaf entries, it is not efficient to apply this plane sweep many times, one for each leaf entry.
Our approach is to simultaneously perform all these plane sweeps. We maintain an aSB-tree for the whole space. Whenever the aSB-tree is modified (due to the insertion of a left edge or a right edge of some \textit{nn}\_buffer), we modify the maximum influence information stored along with all leaf entries intersecting with the current sweep line. After the plane sweep is performed, every entry in \textit{Subtree}(e) has the correct \textit{fullcover}, and every leaf entry in \textit{Subtree}(e) stores the correct \textit{maxrange}, but \textit{lowermax}(=\textit{uppermax}) is the actual (not local) maximum influence. There are two remaining tasks. (a) Change the \textit{lowermax} and \textit{uppermax} for every leaf entry to be local maximum influence. (b) Fill the missing values (\textit{lowermax}, \textit{uppermax}, \textit{maxrange}) for non-leaf entries. These two tasks are achieved by the recursive algorithm \textit{AdjustAndPropagate}.

Figure 3.17(a) shows a node \( N \) in the VOL-tree before \textit{AdjustAndPropagate} is called. The rectangle with thick board is a node \( N \) in the VOL-tree, which has 3 children \( N_1, N_2 \) and \( N_3 \). Assume that the squares are all the \textit{nn}\_buffers which intersect with \( N \). The gray rectangles are the \textit{maxrange} of each entry and the numbers in the tuples are \((\textit{fullcover}, \textit{lowermax}, \textit{uppermax})\) of each entry. The \textit{lowermax} and \textit{uppermax} of \( N \) are unspecified, while those of \( N_1 \) to \( N_3 \) equal to the actual maximum influence in each entry. \textit{AdjustAndPropagate} is called on \( N \) with parameter \textit{sum} = 0, on \( N_1 \) with parameter \textit{sum} = 2, and on \( N_2 \) and \( N_3 \) with parameter \textit{sum} = 1. Figure 3.17(b) shows the results after \textit{AdjustAndPropagate} finishes.
3.4 Max-Inf OL under $L_2$ Metric

In this section, we study the Max-Inf queries under $L_2$ distance metric. Due to the difference between $L_1$ and $L_2$ metrics, the techniques we previously proposed in this chapter do not always work. We examine the proposed techniques and identify the techniques which can be adapted to $L_2$ metric. To accomplish the adaptations, new techniques need to be devised.

3.4.1 Overview

For the Max-Inf OL query under $L_2$ metric, the concept of $nn$-buffer we introduced still applies. Therefore, similar to the case of $L_1$ metric, the optimal location is still the location with maximum overlap among $nn$-buffers of objects. The difference is that, under $L_2$ distance, the $nn$-buffer of an object becomes a circle. Figure 3.18 shows an example with 4 objects and 2 sites. Given the query region $Q$, any location in the intersection between $Q$ and region 2 is an optimal location. Unlike the case of $L_1$ metric, the overlap of multiple $nn$-buffers under $L_2$ metric is irregular. This dissimilarity has different impact on the two solutions we proposed in this chapter.

![Figure 3.18: The $nn$-buffers and the optional location query under $L_2$ metric.](image)

Essentially, the VOL-tree based solution can be adapted with little change. Beside the position change that everything can be considered in the original coordinates, the only thing need to take extra care of is the maintenance of $maxrange$. Under
L_1 metric, the nn_buffers are diamonds and the overlap of multiple nn_buffers is always rectangular (in the rotated coordinate). Therefore, it can be easily maintained. On the other hand, under L_2 metric, the overlap of multiple nn_buffers (circles) is irregular. Its size varies when the number of overlapping nn_buffers changes, which makes it unfeasible to be maintained in the VOL-tree. To avoid this problem, we represent the maxrange by a rectangle inscribed with it. Other part of the VOL-tree based algorithm remains the same (except that L_2 metric is used instead of L_1). Therefore, in the rest of this section, we focus on the R*-tree based solution.

The framework of the R*-tree based solution still works under L_2 metric. However, the details of the algorithm need to be adjusted. There are two major modifications. First, the circle is invariant to any rotation transformation. Therefore, the nn_buffers can be retrieved and processed in the original coordinate. It is actually a desirable feature, which eliminates the overheads of rotating the nn_buffers and handling the rotated query region. Second, the plain-sweep algorithm need to be redesigned. The overlap area among circles is not rectangular and cannot be neatly maintained by the aSB-tree. It implies that the plain-sweep process becomes more complicated.

In the rest of this section, we demonstrate the details of modifications. Similar to the solution under L_1 metric, we assume that the objects are indexed by an R*-tree augmented with the L_2 distance from each object to its closest site and every index entry of the R*-tree stores the maximum L_2 distance of objects in the sub-tree.

### 3.4.2 Retrieving Objects from The R*-tree

The first step of the R*-tree based solution is not different from that under L_1 metric too much. It retrieves from the R*-tree those objects whose nn_buffers intersect with Q. Only these objects may affect the influence of the locations in Q. Therefore, the implementation is identical to that for L_1 metric except for two differences. First, L_2 metric is used instead of L_1 metric in all distance computations. Second, the objects
are returned in the increasing order of their \(x_{\text{low}}\)s in the original coordinate instead of the 45°-rotated coordinate. Here the \(x_{\text{low}}\) of an object has the same meaning as in \(L_1\) case, but the \(nn\_buffer\) of an object is circular. Therefore, only one point on the \(nn\_buffer\) has the x-coordinate of value \(x_{\text{low}}\). The object is still retrieved by a best-first search which browses the \(R^*\)-tree in a top-down fashion. The heap used in the search sorts the entries in the increasing order of their \(x_{\text{low}}\) in the original coordinate.

### 3.4.3 The Naive Plain Sweep

The retrieved objects are processed by a plane sweep algorithm to find the optimal location. Here we first present a naive plain sweep algorithm, which is inefficient, as the starting point. We will show how to improve it in the later sections.

Imagine a vertical sweep line \(l\) moves along the \(x\)-axis from left to right. When it enters the \(nn\_buffer\) of an object \(o\) (i.e. the sweep line is \(x = o.x_{\text{low}}\)), \(o\) is set to the active status. This status remains until the sweep line leaves the \(nn\_buffer\) of \(o\). A set of the active objects is maintained. When a new active object is added, it is checked with each of the existing active objects to detect intersections. The \(nn\_buffer\) of an object is divided into multiple arcs by the intersecting \(nn\_buffers\). Each arc is associated with a weight, which indicates the number of \(nn\_buffers\) covering it. When the first intersection of two \(nn\_buffers\) is found, the arcs of both are further divided and the weights of arcs covered by the other \(nn\_buffer\) increase by 1. After all \(nn\_buffers\) are processed, the arc with maximum weight is used to identify the area with maximum weight, which must be adjacent to the arc. Any location in the area is an optimal location.

Consider the objects in Figure 3.18 (and ignore the query region \(Q\) at this point). The sweep line first meets the \(nn\_buffer\) of \(o_1\), and currently there is no other active \(nn\_buffer\). Therefore, the circle \(o_1\) is treated as one arc with weight 1. After the sweep
line $l$ enters the $nn\_buffer$ of $o_2$ (as shown in Figure 3.19(a)), the $nn\_buffer$ of $o_1$ splits into two arcs: $i_1i_2$ and $i_2i_1$. Here we denote an arc by its two endpoints in clockwise sequence. Arc $i_1i_2$ has weight 2 and arc $i_2i_1$ still has weight 1. After the sweep line enters the $nn\_buffer$ of $o_3$ (as shown in Figure 3.19(b)), arc $i_1i_2$ of $o_1$ further splits into two: $i_1i_4$ and $i_4i_2$. The weight of $i_4i_2$ increases to 3. The $nn\_buffers$ of $o_2$ and $o_3$ are processed in similar fashion. After all $nn\_buffers$ are processed, the arc $i_4i_2$ of $o_1$, $i_2i_5$ of $o_2$ and $i_5i_4$ of $o_3$ have the maximal weight of 3. The area bounded by these three arcs contains (global) optimal locations.

Let $n$ the number of objects retrieved from the R*-tree. The naive plain sweep algorithm has the worst case complexity of $O(n^3)$. In worst case, all $nn\_buffers$ intersect with each other and there are $O(n^2)$ intersections detected. For each $nn\_buffer$, the average number of arcs need to maintained is $O(n)$. Therefore, the total cost is $O(n^3)$.

The naive plain sweep algorithm can be considered as a pipeline with two stages: intersection detection and arc maintenance. Whenever the first intersection of two $nn\_buffers$ is found, the arcs and the corresponding weights on these two $nn\_buffers$ get updated. However, both steps are inefficient and can be improved. In the coming two subsections, we address these two issues respectively. At this point, let us first skip the tricky cases where two or more $nn\_buffers$ are tangent or three or more
3.4.4 Improved Intersection Detection

The $nn\_buffers$ of two active objects may be far away from each other on $y$-axis, even though their intervals on $x$-axis intersect for sure. For example, $o_2$ and $o_4$ in Figure 3.18. However, without a well-organized structure to maintain the active $nn\_buffers$, the only way to ensure their nonintersection is to compare the new active object with all existing active objects one by one. This is what the naive plain sweep does.

To improve it, we utilize the classic line segment intersection algorithm introduced in chapter 2 of [8]. The idea of the line segment intersection algorithm is to order the segments from bottom to top as they intersect the vertical sweep line. When the sweep line moves forward, the order of the segments changes only when the sweep line just enters or leaves a segment or two segments intersect. Furthermore, only the segments adjacent in the ordered list need to be checked for intersection. After two segments intersect, they will switch their position in the ordered list.

In more details, the algorithm considers three type of events: the enter event when the sweep line just enters a segment, the leave event when the sweep line just leaves a segment, and the intersect event when two segments intersect at a point. The first two type of events are known beforehand, while the intersect events are computed on the fly. When an enter event happens, the sweep line meets a new segment. This segment must be inserted into the ordered list of the active segments and tested against its neighbor for intersections. If a new intersection is found, an intersect event should be added to the event queue. Similarly, when a leave event happens, an active segment turns into inactive. This segment must be deleted from the ordered list. Its two neighbors become adjacent and must be checked for new intersections. It is possible that the intersections have been detected earlier, i.e. these two neighbors
have been adjacent before. If this happens, the intersections are ignored. When an
*intersect* event happens, two segments that intersect change their position. Each of
them gets a new neighbor which will be tested against it for intersection. After all
segments being swept, all intersections are discovered.

The algorithm uses a priority queue to store the events and sorts the events in the
order $\prec$ defined as follows. Given two event $e$ and $e'$, $e \prec e'$ if and only if $x_e < x_{e'}$, or
$x_e = x_{e'}$ and $y_e < y_{e'}$. The queue always returns the smallest event as the next event
to be processed. Meanwhile, the algorithm keeps a sorted list of the active segments
according to the $y$-coordinates of their intersection points with the sweep line.

Although the algorithm is designed for line segments, it works for any type of
continuous curves as well. To use this algorithm for *nn_buffers* each *nn_buffer* is
split into two pieces, one corresponding to the upper half of the *nn_buffer* and one
corresponding to the lower half. Each part is treated as a separate curve.

The example in Figure 3.18 shows how the line segment intersection algorithm can
be used to find the intersections of the *nn_buffers* Initially (shown in Figure 3.20(a))
the priority queue $Q = \{e_1, e_2, e_3, v_1, e_4, v_2, v_3, v_4\}$, where $e_i$ and $v_i$ represent the *enter*
and *leave* event of $o_i$ respectively. Note that $e_4 \prec v_2$ implies that $o_4$ has to compare
with $o_2$ in the naive plain sweep algorithm. After $e_1$ is popped and processed, $l_1$ and
$u_1$, which represent the upper and lower half of $o_1$ respectively, are added into $L$.

After $e_2$ is popped and processed (shown in Figure 3.20(b)), $l_2$ and $u_2$ are added
into $L$, which becomes $\{l_1, u_1, l_2, u_2\}$. Since $l_2$ is the new neighbor of $u_1$, $o_1$ and $o_2$ are
compared for intersections. Two new events $i_1$ and $i_2$ are discovered and inserted into
$Q$. Therefor, $Q = \{i_1, e_3, i_2, v_1, e_4, v_2, v_3, v_4\}$. After $i_1$ is processed, $u_1$ and $l_2$ switch
their position in $L$ and $L$ becomes $\{l_1, l_2, u_1, u_2\}$. No new event is found since $o_1$ and
$o_2$ have already been check against each other.

After $e_3$ is processed (shown in Figure 3.20(c)), $l_3$ and $u_3$ are added into $L$, which
becomes $\{l_3, u_3, l_1, l_2, u_1, u_2\}$. Thus, $o_3$ is compared with $o_1$ (but not $o_2$) for inter-
sections. Events \( i_3 \) and \( i_4 \) are discovered and inserted into \( Q \). Note that \( o_3 \) has not been checked with \( o_2 \) at this point in that they are not neighbors in \( L \). After \( i_3 \) is processed, \( u_3 \) switches position with \( l_1 \) and becomes neighbor of \( l_2 \). Intersections \( i_5 \) and \( i_6 \) between \( o_2 \) and \( o_3 \) are then discovered.

This procedure repeats for every event. After \( e_4 \) is processed (shown in Figure 3.20(d)), the last two intersections, \( i_7 \) and \( i_8 \) between \( o_3 \) and \( o_4 \), are then discovered. However, the algorithm will keep going until all events get processed to make sure there is no new event. Note that event \( v_2 \) happens before \( i_7 \) and \( i_8 \). Therefore, \( o_2 \) and \( o_4 \) will never be neighbors and should not be compared. Imagine that there are several “layers” of \textit{nn_buffers} in the input. The improved algorithm compares much less pairs of \textit{nn_buffers} than the naive algorithm. As pointed out in [8], the improved algorithm finds all intersections in \( O(n \log n + I \log n) \) time, where \( n \) is the number of \textit{nn_buffers} to be processed and \( I \) is the number of intersections in the result.

Figure 3.20: Demonstration of the improved plane sweep.
3.4.5 Arc Maintenance

In the naive plane sweep algorithm, the arcs of a nn_buffer are simply organized as a list. Therefore, a sequential scan or binary search is performed to locate each intersection point on the nn_buffer. To increase the weight of arcs between two points, the algorithm has to go through all intervals between them. Assume the nn_buffer $o$ intersects with $m$ other nn_buffers (i.e., there are $2m$ intersection points on $o$), the average cost of updating each intersection (two points) is $\frac{m}{2} = O(m)$. We could improve it to $O(\log m)$.

Conceptually, each nn_buffer can be normalized to an interval between 0 and $2\pi$. The arcs of a nn_buffer are mapped to intervals within $[0, 2\pi]$. Recall the aSB-tree presented in this chapter. It is a perfect structure to maintain the arcs and their weights. Therefore, we build an sSB-tree for each active nn_buffer. Whenever we need to increase the weight of an arc, we map the arc to an interval of weight one and insert into the aSB-tree of the corresponding nn_buffer. The only thing need to be taken care of is that, if the arc has the point corresponding to 0 (or $2\pi$) as an internal point, it should be mapped into two intervals: one is in the form of $[0, \theta_1]$ and the other is in the form of $[\theta_2, 2\pi]$. This special treatment however does not affect the correctness of the algorithm in that even part of the arc with maximal weight is enough to identify one of the optimal locations. Meanwhile, the asymptotic cost of each update is still $O(\log m)$.

3.4.6 Special Cases

Now let us consider the special cases where two or more nn_buffers are tangent or three or more nn_buffers meet in a common point, or both. In this case, multiple events have the same coordinates. In $Q$, the events are ordered by their $x$-coordinate first and then $y$-coordinate. Therefore, each entry in the queue is a set of events (with the same intersection location) instead of a single event. It can be achieved by minor
modifications of the implementation.

![Diagram](image)

Figure 3.21: Three *nn_buffers* intersect at a common point.

We first address the case that multiple *nn_buffers* intersect at a common point. Figure 3.21 gives two examples, which have $o_1$, $o_2$ and $o_3$ intersecting at point $I$. Again we use $l_i$ and $u_i$ to denote the lower half and upper half of circle $o_i$ respectively. If we apply the improved intersection detection algorithm presented in this chapter, we can see the results on these two examples are slightly difference. In Figure 3.21(a), $l_1$ and $l_2$ are neighbors in $L$, so are $l_2$ and $l_3$, and $u_1$ and $u_3$. Therefore, the intersections (at $I$) of all three pairs have been discovered. On the other hand, in Figure 3.21(b), $o_1$ and $o_3$ are not adjacent in $L$ any time before $I$. Therefore, the intersections between $o_1$ and $o_3$ cannot be discovered. Same situation happens when more *nn_buffers* intersect at a common point $I$, which may cause unnecessary complexity if we try to process the detected intersections one by one. To avoid that, we collect all *nn_buffers* intersected at $I$ and process them together. As all neighbors in $L$ are checked at some point before $I$, all *nn_buffers* intersected at $I$ are involved in at least one comparison and therefore can be collected. Afterward, we compare each pair of them to update the arcs and weights, if this is the first intersection of this pair.

In the example of Figure 3.21, $o_1$, $o_2$ and $o_3$ intersect at $I$. After event $I$, their order in $L$ is reversed. More precisely, the order of $l_1$, $l_2$, and $l_3$ is reversed into $l_3$, $l_2$, and $l_1$. In general, if there are $k$ *nn_buffers* intersect at $I$, their order in $L$ is reversed after the intersection event. This can be inferred from the fact that each pair of the
nn_buffers switch their relative position in L after the intersection. Therefore, after comparing all pairs, we reverse the order of collected nn_buffers in L and compare the first and last nn_buffers with their new neighbors for intersections.

![Diagram](image)

(a) Two nn_buffers ex-tangent  
(b) Two nn_buffers in-tangent  
(c) Three nn_buffers in-tangent  
(d) More nn_buffers tangent

Figure 3.22: Cases when two or more nn_buffers are tangent at one point.

We then address the case that multiple nn_buffers are tangent at a common point. Let us start with the simple case that two nn_buffers are tangent. Figure 3.22(a) and (b) show the two possibilities. For ease of presentation, let us call them ex-tangent and in-tangent, respectively. In either case, the relative position of the nn_buffers in L does not change. For example, in Figure 3.22(a), \( u_1 \) is always in front of \( l_2 \) in L before or after event \( i \) happens. The difference of these two cases is that, in case (a) the weight of any arc of \( o_1 \) or \( o_2 \) does not change, while in case (b) the weights of all arcs of \( o_1 \) increase by 1. Therefore, if two nn_buffers ex-tangent at a point, we can simply ignore their intersection event. Otherwise, the weight of the inner nn_buffer should be increased by 1.

Things get more complex when three or more nn_buffers are tangent at a common point. Let us first consider the case that nn_buffers are in-tangent only. Figure 3.22(c) shows an example of three nn_buffers. At the first glance, it is very similar to case (b): we can ignore the intersection event and increase the weight of the inner nn_buffer by
1. However, after examining it more carefully, we realize it is not true. The reason is that $o_2$ is the only neighbor of $o_1$ and $o_1$ has no chance to compute the intersection with $o_3$. Therefore, the weight of $o_1$ is only increased once instead of twice, which is incorrect. To avoid this problem, we collect all $nn\_buffers$ tangent at $I$. For each $nn\_buffer$, the weight need to be added to is the number of $nn\_buffers$ larger than it. Different from the case of coincident intersection, the order of buffers on $L$ is unchanged.

Now let us consider the more general case that $nn\_buffers$ are in-tangent and ex-tangent at a common point. Figure 3.22(d) gives an example. In this case, the $nn\_buffers$ are divided into two groups by the line tangent to all of them. Each $nn\_buffer$ are in-tangent with the $nn\_buffers$ in the same group and ex-tangent with the $nn\_buffers$ in different group. As two ex-tangent $nn\_buffers$ do not affect each other, it is safe that they are not checked for intersections. Therefore, we can handle each group in the same way as we handle the in-tangent case. In summary, we can now handle any case that two or more $nn\_buffers$ are tangent at a common point.

The most general case is that intersection and tangent happen at the same point $I$. Consider three circles meet in the same point and two of them are tangent. The following property holds: a circle cannot get in between two tangent circles if it is not tangent with them. Due to this property, all $nn\_buffers$ tangent at $I$ must be consecutive in $L$. This simplifies the problem. We use two sets $TB$ and $IB$ to collect the $nn\_buffers$ in the tangent events and the rest of $nn\_buffers$ (i.e. the $nn\_buffers$ in intersection events but not tangent events), respectively. Each pair of $nn\_buffers$ in $IB$ are compared. Meanwhile, each $nn\_buffer$ in $TB$ is compared with every $nn\_buffer$ in $IB$. The $nn\_buffers$ in $TB$ are further divided into two groups if some $nn\_buffers$ are ex-tangent with others. The $nn\_buffers$ in each group updates their weights independently. Afterwards, we reverse the order of $nn\_buffers$ in $IB$ and keep the order of $nn\_buffers$ in $TB$ unchanged. Only the first and last $nn\_buffer$ need
to be compared with their new neighbors for new intersections.

3.4.7 Summary

In this chapter we extended the solutions under $L_1$ metric to solve the Max-Inf OL query under $L_2$ metric. In particular, the plane sweep algorithm need to be redesigned. We presented a naive plain sweep algorithm, and two improvements on it.

3.5 Performance

In this section, we report experimental results on the R*-tree based approach and the VOL-tree approach. In our experiments we used real datasets: the Digital Chart of the World from the R-tree Portal [47]. It contains two type of point data: the populated places and cultural landmarks in North America, a total of 24,493 and 9,203 points respectively. We use the populated places as the objects and cultural landmarks as the sites. From the dataset, we generated an object R*-tree for all populated place, which is augmented by the $L_1$ or $L_2$ distance from each object to its nearest site. From our preliminary experimental results, we found that it does not help to make the VOL-tree disk based. So the VOL-tree is in memory, and only the I/O of R*-tree will be measured. In most experiments, we compare the performance of three methods listed in Table 3.1. Each experiment was run on 100 random queries, starting with a clean buffer. Buffer will not be flushed during the execution of 100 queries. We set the page size to 1k and the default buffer size to 256 pages. All the programs were written in Java and run on a Pentium IV Dell PC equipped with 3.2GHz CPU and 4GB memory.
3.5.1 Results under $L_1$ Metric

To utilize the VOL-tree, the first question we need to answer is how large the VOL-tree should be. Since the VOL-tree is always in memory, it consumes part of the buffer for the R*-tree. For example, if the size of the VOL-tree is 50, the buffer available to the R*-tree retrieval should be $256 - 50 = 206$. Figure 3.23 shows the I/O of the R*-tree of various sizes (in the unit of the page size). When the size of the VOL-tree is small ($< 20$ pages), the I/Os become close to the R*-tree based method (which corresponds to VOL-tree size = 0). When the size of the VOL-tree is large ($> 80$ pages), the I/Os also increase. That is because the larger VOL-tree does not help much to prune the search space, but it uses a large proportion of the buffer, which results in the worse I/O of the R*-tree. From the results, we draw the conclusion that a small VOL-tree is sufficient. Thus, in the later experiment, we set the size of VOL-tree to 20 pages.

![Figure 3.23: The I/O performance of the VOL-tree for various size.](image)

To study the effect of the size of query range on the I/Os, we fix the ratio between
the length and width of query range to 1 (i.e query ranges are squares), and change the area of the query range. Figure 3.24 (a) and (b) shows the results when the query range is small and is large respectively. When the query area is smaller than 1% of the whole space, their performances are very close although VOL-tree methods are slightly better. When the query area is larger than 1%, the R*-tree based method has I/Os of more than 10,000 so we do not even show it in figure. An expected fact is that when the query range becomes very large, the performance of both VOL80 and VOL100 improve. That is because, the query ranges will intersect with more maxranges and VOL-tree methods are more likely to prune some subtrees in the search. Our other experiments also show that the ratio has little impact on the query performance of VOL80 and VOL100. Therefore in the remaining part of this section, we set the query area to 5% and the query ratio to 1.

![Figure 3.24: The I/O performance of the VOL-tree for various query area.](image)

(a) Queries with small area (Length/Width = 1) (b) Queries with large area (Length/Width = 1)

The updates will make the lowermax and uppermax of some entries of the VOL-tree not tight, thus decrease the pruning capability. Figure 3.25 shows how updates affect the I/O performance. We bulk load some objects and insert the others. The X-axis presents the percentage of the number of inserted objects to the number of bulk loaded objects. For example, X = 50% corresponds to the case when we bulk load 2/3 of the objects and insert the remaining 1/3. With the increase of the percentage, the I/O performance decreases. After about 50% insertions, the performance becomes
comparable to the R*-tree based method. There are two reasons for that. One is because the VOL-tree uses some buffer of the R*-tree. The other is because the VOL-tree may causes multiple scans of same page. In many real life applications, most objects are known in advance and the percentage of new updates is small. Thus the VOL-tree based method is more promising. We need to point out that even if an application is update intensive, the VOL-tree based method is still a good choice since the tree can be rebuilt in part or in full. And the rebuilding cost is amortized. Furthermore, the rebuilding can be integrated with the query processing. During the execution of queries, we may need to find the local optimal location for the intersection between the query range and some entry. If the intersection is large, we can plane sweep for the whole entry instead of just for the intersection.

Figure 3.25: The Effect of the Updates.  
Figure 3.26: The Effect of the Buffer Size.

Figure 3.26 shows how the buffer size affects the I/O performance of the VOL-tree. When buffer size is 128, the R* outperforms VOL80. That is because the VOL-tree has size of 20 and occupies about 20% the buffer. After the buffer size is doubled to 256, the I/O of VOL80 dramatically drops to below R*. With the increase of the buffer size, the performance of all the three methods get improved, while the VOL-tree based method is typically better.
3.5.2 Results under $L_2$ Metric

Under $L_2$ metric, the comparison between the R*-tree base solution and the VOL-tree based solution shows very similar results as those under $L_1$ metric. Therefore, we omit that part and focus on evaluating the redesigned part of the R*-tree based solution. For each experiment, given the number of sites, we randomly select some cultural landmarks as the sites. In real applications, the number of sites is typically very small. So, in our experiments, we keep all sites in memory. However, the sites can be organized as an R*-tree and our algorithm still applies. Since both the naive plain sweep algorithm and the improved plain sweep algorithm run in memory, we ignore the disk I/O in the experiments. Unless otherwise stated, the experiments use the default parameters as given in Table 3.2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of sites</td>
<td>100</td>
</tr>
<tr>
<td>Query size</td>
<td>1% in each dimension</td>
</tr>
</tbody>
</table>

Table 3.2: The default parameters.

To examine how much improvement we can get by using the improved intersection detection method, we compare the running time between the naive plain sweep algorithm and the algorithm with the improved intersection detection. We vary the number of sites and query size. Figure 3.27 shows the results. The algorithm with improved intersection detection is about 10 times faster than the naive algorithm. When query size increases or the number of sites decreases, the running time of both algorithm increase. However, the running time of the naive plain sweep algorithm increases at a higher rate. It is because when query size increases or the number of sites decreases, the number of $nn$-buffers need to be processed increase. In this case, the improved intersection detection can avoid more unnecessary comparison between $nn$-buffers.

Figure 3.27 compares the running time between the naive plain sweep algorithm
and the algorithm only with arc maintenance, which does not implement the improved intersection detection. When the query size changes, the difference between these two algorithm does not change a lot. When the number of sites decreases, the difference becomes much obvious. This is become the contribution of arc maintenance depends on the average intersections on each \textit{nn\_buffer}. When the query size changes, it does not change a lot. When the number of sites decreases, the \textit{nn\_buffers} become larger and have more intersects. In this case, arc maintenance greatly improves the performance.
3.6 Conclusions

In this chapter we solved the Max-Inf OL query under $L_1$ metric. We presented two solutions to accurately answer such a query. In particular, the VOL-tree approach is more efficient. The approach uses an R*-tree to index the objects, while a small, in-memory VOL-tree is used to prune the search space. The query performance is much better than the plain R*-tree approach, especially when the query size is large. (Notice that the R*-tree approach is already optimized via a new index called the aSB-tree.) For instance, if the query area is 5% of the space, the VOL-tree approach computes an optimal location 6 times faster than the R*-tree approach. If the query size increases, the improvement increases as well, which can be multiple orders of magnitude better.

Also, the size of the VOL-tree is small. In our experiments, while the R*-tree of objects is over 700 disk pages, the VOL-tree is only 20 pages. The VOL-tree has very efficient updates, as the index is small and updating it does not need to touch the R*-tree (except for ordinary object insertion/removal). One set of experiments showed that within 50% new updates, the VOL-tree approach remained to have better query performance. Of course, if there are too many updates, the VOL-tree can be re-built and the cost is amortized across all the new updates.

We adapted both solutions to $L_2$ metric case. To adapt the R*-tree based solution, we redesigned the plane sweep algorithm, in particular, the intersection detection and the arc maintenance. Experimental results show that they greatly improve the performance.
Chapter 4

Min-Dist OL under $L_1$ and $L_2$ Metric

In this chapter, we solve the Min-Dist OL query under $L_1$ and $L_2$ metric. Similar to the last chapter, we propose the solution for $L_1$ metric in sections from 4.1 to 4.5 and extend it to $L_2$ metric in section 4.6. In particular, we propose a progressive algorithm which quickly suggests a candidate location with the maximum possible error and keeps refining the result. The algorithm finds the exact answer under $L_1$ metric, while it provides arbitrary accurate estimation under $L_2$ metric.

4.1 Overview

Recall that the Min-Dist OL query finds a location $l$ with minimal average distance $AD(l)$ and the the average distance $AD(l)$ of a location $l$ is defined as the average distance between each object to its nearest site, considering all existing sites and a hypothetical new site at $l$.

To find a min-dist optimal location, there are two challenges need to be conquered. First, theoretically there are infinite number of locations in the query range $Q$. All these locations could be candidates. To address this problem, we prove some theorems
which show that only a finite set of candidate locations need to be examined and these candidates are the intersections of some vertical lines and horizontal lines. In other words, if we pick the intersection point with the smallest average distance, it is guaranteed to be an optimal location.

The second challenge is that it could be still too expensive to check the average distance for all candidates. As the remedy, our progressive algorithm partitions the query range $Q$ into a few cells (by using some of the vertical and horizontal lines), and calculates $AD(\cdot)$ for the corners of these cells. It is guaranteed that any candidate location, whose $AD(\cdot)$ is not computed yet, is in some cell and in turn can be found if the cell is partitioned. Among the cell corners, the one with the smallest $AD(\cdot)$ can be reported as a temporary optimal location.

One key technique used in our method is a lower bound estimator of $AD(\cdot)$ of all locations in a given cell. This technique offers two benefits. First, the maximum error can be reported along with the temporary optimal location. This allows the user to abort the calculation if the error is considered to be small enough. More importantly, it is possible to prune a whole cell (i.e., including the non-examined candidate locations in the cell) if its lower bound is not smaller than the $AD(\cdot)$ of the temporary optimal location. Equipped with this pruning power, our algorithm can efficiently find the exact OL without examining all candidates.

The rest of this chapter will present the details of our algorithm. For the ease of presentation, table 4.1 summarizes the notations that will be used or introduced in this chapter.

### 4.2 Computing $AD(l)$ for A Location $l$

To solve the min-dist optimal location query, we first address the following problem: Given a candidate location $l$, how do we compute the average distance $AD(l)$?
<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>the set of sites</td>
</tr>
<tr>
<td>$O$</td>
<td>the set of objects</td>
</tr>
<tr>
<td>$Q$</td>
<td>the query region</td>
</tr>
<tr>
<td>$l$</td>
<td>a candidate location for a new site</td>
</tr>
<tr>
<td>$d(p_1,p_2)$</td>
<td>the $L_1$ distance between two points $p_1$ and $p_2$</td>
</tr>
<tr>
<td>$d(o,S)$</td>
<td>the $L_1$ distance between object $o$ to its nearest site in $S$</td>
</tr>
<tr>
<td>$RNN(l)$</td>
<td>the set of objects in $O$ that are closer to $l$ than to their nearest sites in $S$</td>
</tr>
<tr>
<td>$AD(l)$</td>
<td>the average distance between an object in $O$ to its nearest site in $S$ union of $l$</td>
</tr>
<tr>
<td>$AD$</td>
<td>the average distance between an object in $O$ to its nearest site in $S$</td>
</tr>
<tr>
<td>$VC(l)$</td>
<td>the Voronoi cell of a location $l$ with regard to $S$</td>
</tr>
<tr>
<td>$VCU(R)$</td>
<td>the Voronoi cell union of region $R$ with regard to $S$</td>
</tr>
</tbody>
</table>

Table 4.1: Summary of Notations.

By definition of $AD(l)$, we could compute $d(o,S \cup \{l\})$ for every object $o$, and then take the average. But this is costly in that it needs to access all objects. Intuitively, if a new McDonald’s store is added, only the nearby residents will be affected and their distances need to be recomputed.

To formally develop this intuition, let’s first define $AD$ as the average distance from every object in $O$ to its nearest site, without considering a new site. That is:

$$AD = \frac{1}{|O|} \sum_{o \in O} d(o,S). \quad (4.1)$$

Clearly, $\forall l \in Q$, $AD(l) \leq AD$. If no object in $O$ is closer to $l$ than to its nearest site in $S$, $AD(l) = AD$. Otherwise, $AD(l) < AD$. The following theorem 4.2.1 shows that, knowing $AD$, we only need to visit objects in $RNN(l)$ to compute $AD(l)$.

**Theorem 4.2.1.**

$$AD(l) = AD - \frac{1}{|O|} \sum_{o \in RNN(l)} d(o,S) - d(o,l)$$
Proof. According to the definition,

\[ AD(l) = \frac{1}{|O|} \sum_{o \in O} d(o, S \cup \{l\}). \]  \hspace{1cm} (4.2)

From Equations 4.2 and 4.1, we have:

\[ AD - AD(l) = \frac{1}{|O|} \sum_{o \in O} (d(o, S) - d(o, S \cup \{l\})). \]

Notice that if \( o \notin RNN(l) \), \( d(o, S) = d(o, S \cup \{l\}) \). To prove the theorem, it remains to point out that for an object \( o \in RNN(l) \), \( d(o, S \cup \{l\}) = d(o, l) \).

Theorem 4.2.1 tells how to compute \( AD(l) \) for an arbitrary location \( l \). Compared with \( l \) which varies, \( S \) and \( O \) can be considered as fixed. Therefore we can pre-compute \( AD, |O|, \) and \( d(o, S) \) for every object. In order to compute \( AD(l) \), we only need to find the RNNs of \( l \) and compute \( d(o, l) \) for every \( o \in RNN(l) \).

4.3 Limiting the Number of Candidates

Given a query region \( Q \), it contains an infinite number of locations. It is impossible to check the \( AD(\cdot) \) for all of them. This section shows how to limit the number of candidate locations that need to be checked, yet still guarantees that an exact answer is found.

4.3.1 The Number of Candidate Locations is Finite

Consider Figure 4.1. The black dots are the objects. Here the thick-bordered rectangle is the query region \( Q \). The shadowed region is composed of a horizontal extension of \( Q \) and a vertical extension of \( Q \), which are defined below.
Definition 4.3.1. Given an axis-parallel rectangle $Q$, the horizontal extension of $Q$ is the area derived from infinitely extending $Q$ horizontally. The vertical extension of $Q$ is the area derived from infinitely extending $Q$ vertically.

Obviously, the intersection between the horizontal extension and the vertical extension is $Q$ itself.

Consider each horizontal line that passes through at least one object in the horizontal extension of $Q$ and each vertical line that passes through some object in the vertical extension of $Q$. Also consider the horizontal and vertical lines that pass through the corners of $Q$. For instance, in Figure 4.1 there are six such vertical lines and five such horizontal lines as shown in dashed style. (Note that one line may pass through more than one object.) They make 30 intersection points. According to Theorem 4.3.2 below, even though there is an infinite number of locations in $Q$, we only need to check these intersection points. It is guaranteed that we can find a min-dist optimal location among them.

Theorem 4.3.2. Consider the set of horizontal lines (and vertical line respectively) that go through some vertex of $Q$ or go through some object in the horizontal extension (and vertical extension respectively) of $Q$. There exists a min-dist optimal location at some intersection point of these lines.

Proof. We prove the theorem by contradiction. Assume none of the intersection points is the optimal location. There may be multiple optimal locations. Let $l$ be arbitrary one of them. According to the assumption, $l$ is either not on any horizontal
line or not on any vertical line defined in the theorem. Without losing generality, we assume that $l$ is not on any vertical line.

Consider the objects in $RNN(l)$. In general, some of them are to the left of $l$ and the rest are to the right of $l$. However, due to the contradictory assumption, no object is right above or below $l$. Assume that $n_L$ is the number of objects to the left of $l$, and $n_R$ is the number of objects to the right of $l$. Without loss of generality, we further assume that $n_L \geq n_R$.

We move a site on $l$ leftward until either it encounters the first vertical line. Let $l'$ be the location where it stops. We argue that the movement improves the $AD(\cdot)$, i.e. $AD(l') \leq AD(l)$. The reason is as follows.

$$AD(l) - AD(l') = \frac{1}{|O|} \sum_{o \in O} (d(o, S \cup \{l\}) - d(o, S \cup \{l'\}))$$  \hspace{1cm} (4.3)$$

$$= \frac{1}{|O|} \sum_{o \not\in RNN(l)} (d(o, S \cup \{l\}) - d(o, S \cup \{l'\}))$$ \hspace{1cm} (4.4)$$

$$+ \frac{1}{|O|} \sum_{o \in RNN(l)} (d(o, S \cup \{l\}) - d(o, S \cup \{l'\}))$$ \hspace{1cm} (4.5)$$

As shown in above equations, the summation in $AD(l) - AD(l')$ can be decomposed into two parts. The first part 4.4 is the summation over the objects not in $RNN(l)$, and the second part 4.5 is the summation over the objects in $RNN(l)$. We show that both parts are non-negative.

For every object $o \not\in RNN(l)$, the distance to its nearest site does not increase when a site moves from $l$ to $l'$. If $o \in RNN(l')$, the distance decreases, otherwise, it keeps unchanged. In any case, $d(o, S \cup \{l\}) - d(o, S \cup \{l'\}) \geq 0$. Therefore, the summation in 4.4 is at least 0.

For the objects in $RNN(l)$, the calculation is more complex because the distance to the nearest site may increase for some objects. Specifically, when a site moves
from \( l \) to \( l' \), each object \( o \) to the left of \( l \) will consider \( l' \) as the new nearest site. The distance decreases by \( d(o, l) - d(o, l') = d(l, l') \). Therefore, all objects to the left of \( l \) contribute \( n_L d(l, l') \) to the summation in 4.5. On the other hand, an object on the right side of \( l \) may or may not consider \( l' \) as its new nearest site. There could be some site \( s \) closer to \( o \) than \( l' \). In this case, we have \( d(o, l) \leq d(o, s) < d(o, l') \), i.e the distance to the nearest site increases by at most \( d(o, l') - d(o, l) = d(l, l') \). For all such \( n_R \) objects, their (negative) contribution to the summation is at least \( -n_R d(l, l') \). Given the assumption that \( n_L \geq n_R \), the summation in 4.5 is at least 0. Therefore, \( AD(l') \leq AD(l) \).

Similarly, if \( l' \) is not on any horizontal line, it can be moved vertically to \( l'' \) such that \( AD(l'') \leq AD(l') \). Now we have \( AD(l'') \leq AD(l) \), which implies that \( l'' \) should also be an optimal location. It is contradictory to the assumption and proves our theorem.

4.3.2 Further Limiting the Number of Candidate Locations

According to Theorem 4.3.2, to get the set of candidate locations, we considered all objects in either the horizontal extension or the vertical extension of \( Q \). In fact, if we study the theorem proof carefully, we can notice that there is no need to consider all such objects. Instead, we only need to consider the objects which belong to \( RNN(l) \) for some location \( l \) in \( Q \). Here we introduce a concept called the Voronoi cell union and discuss how it can be used to further limit the number of candidate locations.

**Definition 4.3.3.** Given a set of sites \( S \) and a spatial region \( Q \), the **Voronoi cell union** of \( Q \) with regard to \( S \) is the union of Voronoi cells of every location \( l \) in \( Q \). Formally,

\[
VCU(Q) = \bigcup_{l \in Q} VC(l).
\]

\( VCU(Q) \) is the minimum spatial region that fully contains the Voronoi cell for
every location \( l \) in \( Q \). An equivalent explanation is that \( VCU(Q) \) is a spatial region consisting of all locations which may consider a new site somewhere in \( Q \) as the nearest site.

**Theorem 4.3.4.** Consider the set of horizontal lines (and vertical line respectively) which go through some vertex of \( Q \) or go through some object in the intersection of \( VCU(Q) \) and the horizontal extension (and vertical extension respectively) of \( Q \). There exists a min-dist optimal location at some intersection point of these lines.

The proof of Theorem 4.3.4 is very similar to the proof of Theorem 4.3.2. Note that, when a site in \( Q \) is moved, only objects in \( VCU(Q) \) will be affected. Therefore, when choosing objects which will define the vertical and horizontal lines (whose intersections are the candidates), we only need to choose the objects in \( VCU(Q) \).

![Figure 4.2: The candidate locations are further reduced using \( VCU(Q) \).](image)

According to Theorem 4.3.4, the \( VCU \) can help reduce the number of candidate locations. For example, let us assume that the dashed polygon in Figure 4.2 is \( VCU(Q) \) (we will explain how to compute \( VCU(Q) \) in the next section). We only need to consider the objects in the shadowed region which is the intersection between \( VCU(Q) \) and the vertical/horizontal extensions of \( Q \). In this example, there are now four horizontal lines and four vertical lines, which result in 16 candidate locations instead of 30 as in Figure 4.1. Think the whole space as the United States, and the query region as a city. The range \( VCU(Q) \) is typically a small extension of \( Q \), and therefore the savings of this optimization may be much more significant in practice.
4.3.3 Retrieving Objects in $VCU(Q)$

In order to use Theorem 4.3.4 to reduce the number of candidates, the objects in $VCU(Q)$ need to be retrieved. We develop two methods to achieve this goal efficiently.

The first method utilizes the pre-computation. This method indices the objects in an R*-tree. Furthermore, it pre-computes the distance from every object to its nearest site and stores the distance with the corresponding object. The distances are aggregated upward so that every index entry stores the minimum and maximum distance of objects in the sub-tree.

To retrieve the objects in $VCU(Q)$, we can browse the R*-tree in a top-down fashion. To determine whether to expand an index entry, we first compute the minimum and maximum distance between the MBR of the entry and the query region $Q$; then we compare these two distances with the distances aggregated in the entry (i.e., the minimum and maximum distance from objects in the sub-tree to their nearest sites). If the minimum distance between the MBR of the entry and $Q$ is larger than or equal to the maximum distance aggregated in the entry, no object in the subtree will be closer to any location in $Q$ than to its nearest site. Therefore, the entry can be discarded. On the other hand, if the maximum distance between the MBR of the entry and $Q$ is smaller than or equal to the minimum distance aggregated in the entry, every object in the subtree will be closer to some location in $Q$ than to its nearest site. Therefore, all the objects in the subtree will be retrieved. If none of these two cases happens, the entry need to be expanded and each child of the entry will be checked in the way as mentioned above. Figure 4.3 specifies the details of this method.

The second method computes $VCU(Q)$ on the fly and uses a range query to retrieve all objects inside $VCU(Q)$. Since the range query processing has been well studied, we focus on how to compute $VCU(Q)$ efficiently, which will be discussed in details in the next section.
Algorithm \textit{RetrieveVCUQ}
Input: Query region $Q$, R*-tree $root$.
Return: All objects in $VCU(Q)$.

1. $V = \emptyset$, $Queue = \{root\}$.
2. while $Queue \neq \emptyset$
   (a) $e = Queue.ExtractNext()$.
   (b) if $e.minDist \geq MaxDist(e.mbr, Q)$
       Add all objects in the subtree of $e$ into $V$.
   (c) if $e.maxDist > MinDist(e.mbr, Q)$
       Push all children of $e$ into $Queue$.
3. return $V$.

Figure 4.3: Retrieving objects in $VCU(Q)$ using the pre-computation.

4.4 Computing $VCU$

Conceptually, $VCU(Q)$ is the union of the Voronoi cells of every point in $Q$. However, the number of points in $Q$ is infinite. It is non-trivial to compute the union of their Voronoi cells. Meanwhile, there are many sites in the set $S$. Only a small fraction of them contribute to define the boundaries of $VCU(Q)$. An efficient algorithm should access as few sites as possible. In this section, we present the techniques we used to address these two problems. In particular, we first define the bisector between a point and a rectangle, and show how it is related to the computation of $VCU$. Then we illustrate how to compute the bisector between a point and a rectangle. Finally we present the method to reduce the number of bisectors need to be computed.

4.4.1 Analysis

We now take a closer look at the $VCU$. Let us first review the structure of a single Voronoi cell. Given two sites $p$ and $q$ in the plane, the \textit{bisector} of $p$ and $q$ is the line consisting of all points whose distance to $p$ and to $q$ are equal. This bisector splits the plane into two half-planes. We denote the half-plane containing $p$ by $H(p, q)$ and call it the half-plane of $p$ to $q$. Given a site $s$, it is well known that the Voronoi cell $VC(s)$ of
s is the intersection of the half-planes of s to each site, i.e. $VC(s) = \cap_{s' \in S, s' \neq s} H(s, s')$.

We study the structure of $VCU$ in the similar way. First, we define the bisector $B(s, Q)$ between a site s and a rectangle Q. At this point, let us consider the bisector $B(s, Q)$ as a general curve. Shortly, we will see that, under $L_1$ metric, $B(s, Q)$ is actually pairwise linear.

**Definition 4.4.1.** Given a site s and a rectangle Q, the bisector $B(s, Q)$ between s and Q is the set of points, each of which has the same distance to s as to its nearest point in Q. Formally, $B(s, Q) = \{p|\forall q \in Q (d(p, q) \geq d(p, s)) \land \exists q' \in Q (d(p, q') = d(p, s))\}$.

Similar to the bisector between two sites, the bisector $B(s, Q)$ splits the plane into two half-planes. We denote the half-plane that contains Q by $H(Q, s)$ and call it the half-plane of Q to s. Every point on $H(Q, s)$ is closer to some point in Q than to s, while every point on the other half-plane is closer to s than to any point in Q. Notice that, if a site s is inside Q, every point in the plane except s may be closer to some point in Q than to s. Therefore, the bisector between s and Q degenerates to the point s. So in the remaining discussion of this section, we always assume that the sites lie out of the range Q.

Now we present a corollary and a theorem which show how the bisector defined above is related to Voronoi cell union.

**Lemma 4.4.2.** Given a site s and a rectangle Q, the half-plane of Q to s, $H(Q, s)$, is the union of the half-plane of q to s for every q within Q, i.e. $H(Q, s) = \cup_{q \in Q} H(q, s)$.

**Proof.** On the one hand, if a point p is in $H(Q, s)$, it is closer to Q than to s. Let q be the point in Q that is closest to p. We have p is closer to q than to s, which implies that p is in $H(q, s)$. On the other hand, if a point p is in $H(q, s)$ of some point q in Q, it is closer to Q than to s and therefore is in $H(Q, s)$. In summary, a point p is in
$H(Q, s)$ if and only if there exists a point $q$ in $Q$ such that $p$ is in $H(q, s)$. It proves the corollary.

**Theorem 4.4.3.** Given a set $S$ of sites and a rectangle $Q$, the Voronoi cell union of $Q$ is the overlap of the half-planes of $Q$ to all sites, i.e. $VCU(Q) = \cap_{s \in S} H(Q, s)$.

**Proof.** According to the definition VCU and voronoi cell, $VCU(Q) = \cup_{l \in Q} VC(l)$ and $VC(l) = \cap_{s \in S} H(l, s)$. Therefore, $VCU(Q) = \cup_{l \in Q} \cap_{s \in S} H(l, s) = \cap_{s \in S} \cup_{l \in Q} H(l, s)$. □

Theorem 4.4.3 shows that $VCU(Q)$ is bounded by the bisectors between $Q$ and some sites in $S$. Thus, in order to compute $VCU(Q)$, we can first calculate the bisectors between $Q$ and every site in $S$, and then find the overlap of the half-planes defined by these bisectors. In the rest of the section, we will first present how to compute the bisector between a point and a rectangle in $L_1$ metric step by step, and then address the efficiency by pruning certain sites in the computation.

**4.4.2 The Bisector between Two Points**

The bisector between two points under $L_1$ metric is a well-known result from [27]. For the consistency of the presentation, we briefly review it here.

![Bisector between two points](image)

Figure 4.4: The Bisector between two points.

Consider the bisector $B(p, q)$ between two point $p$ and $q$. Without losing generality, we assume $q$ is not higher than $p$. We denote by $p'$ the projection of $p$ on the horizontal
line across q and by θ the angle \( \angle pqp' \). According to the degree of angle \( \theta \), we divide the configuration between \( s \) and \( q \) into two cases: a) \( \theta \leq 45^\circ \), b) \( \theta \geq 45^\circ \). The bisector for these two cases are illustrated in Figure 4.4 (a) and (b) respectively. When \((p, q)\) is a vertical (horizontal) line, the bisector is a horizontal (vertical) line. A special case is, as shown in Figure 4.4 (c), \( \theta = 45^\circ \). Here every points in the shadowed areas have the same distance to \( p \) and to \( q \). Therefore, the bisector is no longer a line. Instead it is two shadowed areas connected with the dashed diagonal. However, the half-plane \( H(p, q) \) (or \( H(q, p) \)) is still the space on \( p \)'s (or \( q \)'s) side to the bisector. As long as \( H(p, q) \) and \( H(q, p) \) can be defined without ambiguity, we can compute \( VCU(Q) \) according to theorem 4.4.3. Therefore, this special case wont invalidate our argument below.

### 4.4.3 Bisector between An Edge and A Point

Here we illustrate how to compute the bisector \( B(s, E) \) between an edge \( E \) and a point \( s \). Later we will show how it can be used to compute the bisector between a rectangle and a point.

Without losing generality, we assume \( E \) is a horizontal line segment, whose left vertex is \( u \) and right vertex is \( v \), and \( s \) is above edge \( E \). We denote the projection of \( s \) on line \((u, v)\) by \( s' \) and the projections of \( u \) and \( v \) on horizontal line across \( s \) by \( u' \) and \( v' \) respectively.

Let us first assume that the edge \( E \) is on one side, say the right side, of vertical line \((s, s')\). We will relax this assumption later. Under this assumption, the bisector \( B(s, E) \) can be as follows:

1. If the angel \( \angle sus' \leq 45^\circ \), \( B(s, E) = B(s, u) \) as shown in Figure 4.5 (a); otherwise, follow step 2 - 4 to compute \( B(s, E) \).

2. \( B(s, E) \) will intersect with line \((u, u')\), and the part of \( B(s, E) \) on the left of
(u, u') is the part of B(s, u) on the left of (u, u'), as shown in Figure 4.5 (b) and (c).

3. The part of B(s, E) inside the rectangle with diagonal (u, v') will be a line segment with slope $\frac{1}{2}$, as shown in Figure 4.5 (b) and (c).

4. If the angle $\angle svu' \leq 45^\circ$, the bisector intersects with the edge (v, v') and the remaining part is a horizontal line (as shown in Figure 4.5 (b)); otherwise (the angle $\angle svu' > 45^\circ$), the bisector intersects with the edge (u, v'), the remaining part is a vertical line (as shown in Figure 4.5 (c)).

![Figures 4.5: Computing the bisector between a point and an edge.](image)

**Theorem 4.4.4.** In Figure 4.5 (b) and (c), the bisector $B(s, E)$ has slope $\frac{1}{2}$ inside the rectangle with diagonal (u, v').

**Proof.** Without losing generality, we assume $E = (u, v)$ is on the x-axis of the coordinate and $s$ is on the y-axis. Therefore, we let $s = (0, y_s), u = (x_u, 0)$, and $v = (x_v, 0)$. Consider a point $b = (x, y)$ on the bisector $B(s, E)$ inside the rectangle with diagonal (u, v'). The distance $d(b, s) = x + (y_s - y)$ and the distance $d(b, E) = y$. Given the fact that $d(b, s) = d(b, E)$, we have $y = \frac{x + y_s}{2}$. The theorem holds.

Now let us relax the assumption that the edge $E$ is on the right side of vertical line $(s, s')$. Due to the symmetry, the case when the edge $E$ is on the left side of $(s, s')$ is straightforward. Therefore, we only consider the case when $s'$ is between $u$ and $v$ as shown in Figure 4.5 (d). In this case, we split (u, v) into two parts, (u, s')
and \((s', v)\), and then compute the bisector for both of them. The bisector \(B(s, E)\) is the combination of \(B(s, (u, s'))\) and \(B(s, (s', v))\). The part of \(B(s, E)\) on the left of line \((s, s')\) is the part of \(B(s, (u, s'))\) on the left of \((s, s')\). The part of \(B(s, E)\) on the right of line \((s, s')\) is the part of \(B(s, (s', v))\) on the right of \((s, s')\).

Denote the distance between \(s\) and \(s'\) by \(h\), the distance from \(s'\) to \(u\) by \(d_1\), and the distance from \(s'\) to \(v\) by \(d_2\). The following theorem states the property of \(B(s, E)\), which can be verified by simply calculations.

**Theorem 4.4.5.**
1. In Figure 4.5 (b) or (c) the left end of \(B(s, E)\) is a horizontal line \(\frac{h-d_1}{2}\) higher than edge \(E\).

2. In Figure 4.5 (b) the right end of \(B(s, E)\) is a horizontal line \(\frac{h+d_2}{2}\) higher than edge \(E\).

3. In Figure 4.5 (c) the right end of \(B(s, E)\) is a vertical line whose distance to \(s\) is equal to \(h\).

4. In Figure 4.5 (d) the lowest point of \(B(s, E)\) is \(\frac{h}{2}\) higher than edge \(E\).

### 4.4.4 Bisector Between A Rectangle and A Point

Given a rectangle \(Q\) and a point \(s\), the bisector \(B(s, Q)\) is the line, each point \(p\) on which has the same distance to \(s\) as to the nearest point in \(Q\). Under \(L_1\) metric, the nearest point in \(Q\) to \(p\) is always on the border of \(Q\). Therefore, the intuition is that \(B(s, Q)\) is the combination of the bisectors between \(s\) and four edges of \(Q\). After deeper study, we have the following theorem which says that \(B(s, Q)\) is actually the bisector between \(s\) and one edge of \(Q\).

**Theorem 4.4.6.** We shoot a 45° or 135° line from each vertex of \(Q\) and divide the space into 4 areas as shown in Figure 4.6. The bisector \(B(s, Q)\) is the bisector between \(s\) and the edge adjacent to the area which contains \(s\).
Proof. Without losing generality, we assume that \( s \) is in the area on top of \( Q \), as shown in Figure 4.6. The bisector between \( s \) and the top edge \( E \) of \( Q \) is one of the case (b), (c) or (d) in Figure 4.5 (but not case (a) because, if it were case (a), \( s \) would have been on the left of the 135° line). In any case, the bisector \( B(s, E) \) is above \( E \).
Consider an arbitrary point \( p \) on \( B(s, E) \). Since \( p \) is above \( E \), the nearest point in \( Q \) to \( p \) must be a point on edge \( E \). Therefore, \( d(p, Q) = d(p, E) \). Due to the definition of \( B(s, E) \), \( d(p, E) = d(p, s) \). Thus, \( d(p, Q) = d(p, s) \), which implies that \( B(s, E) \) is also the bisector between \( P \) and \( s \). The theorem holds.

4.4.5 Computing \( VCU(Q) \)

To compute \( VCU(Q) \), a straightforward solution is to compute the bisector \( B(s, Q) \) for every site \( s \in S \) and check these bisectors one by one to find the boundary of \( VCU(Q) \). However, the intuition is that some sites far away from \( Q \) can to be pruned because they wont contribute to the boundary of \( VCU(Q) \). In this subsection, we develop a method to efficiently reduce the number of sites need to be checked. For ease of presentation, we present the method in two steps. in particular, we first show how to reduce the search space of sites for each edge independently, and then discuss how to combine the edges together to further prune the search space.

According to theorem 4.4.6, each edge just needs to be checked with sites within the adjacent region shown in Figure 4.6. To reduce the search space of the sites for an edge \( E \), we first search a site \( s \) closest to \( E \) in the shadowed area as shown.
in Figure 4.7 (b). The search area goes upward unboundedly. The borders of the
shadowed area are two diagonal lines across the vertices of edge $E$ respectively. From
the result of previous subsections, we know that, for any site $s$ within the shadowed
area, the bisector $B(s, E)$ has both ends as horizontal lines. We denote the distance
between its higher end and edge $E$ by $h_s$. Then the search space for this edge can
be shrunk into the shadowed area shown in Figure 4.7 (c). The search area has two
“arms”, which go upward unboundedly with $45^\circ$ or $135^\circ$ angel. For any site $s'$ above
the search area, the lowest point of the bisector $B(s', E)$ will be at least $h_s$ higher than
$E$. It implies that the bisector $B(s', E)$ will be totally above the bisector $B(s, E)$.
Thus we can safely prune all such sites.

Figure 4.7: Pruning the search space for one edge.

The search space is pruned a lot by the method above. However, it is still un-
bounded. Our study shows that combining the adjacent edges together can fur-
ther limit the search space to a bounded area. Consider two edge $E_1 = (a, b)$ and
$E_2 = (b, c)$ of range $Q$, as shown in Figure 4.8 (a). Assume that the search spaces for
$E_1$ and for $E_2$ have width $2h_1$ and $2h_2$ respectively. We denote by $l_1$ the horizontal

Figure 4.8: Pruning the search space for the whole rectangle.
line $h_1$ higher than $E_1$ and by $l_2$ the vertical line $h_2$ far to the right of $E_2$. If we consider two edges separately, the search space will go toward upper-right corner infinitely long. However, if we consider two edges together, the space beyond some line can be cut off. As illustrated in Figure 4.8 (a), the cutting line is a 45° line across the point whose relative coordinates to the point $b$ is $(2h_2, 2h_1)$. For any site $s$ on this line, the bisector $B(s, E_1)$ is coincident with the intersection of $l_1$ and $l_2$. So is the bisector $B(s, E_2)$. For any site $s'$ beyond this line, the part of bisector $B(s', E_1)$ below $l_1$ is to the right of $l_2$, and the part of the bisector $B(s, E_2)$ to the left of $l_2$ is above $l_1$. Therefore, we can safely prune all such sites. So the search space can be reduced to the grey area shown in Figure 4.8 (a). Applying this method to each corner, we finally get the whole search space, which looks like the gray area in Figure 4.8 (b).

To get the exact $VCU$, we just retrieve all sites within the search space, compute their bisectors to the corresponding edges, and return the overlap of the spaces of $Q$ to those sites. The whole algorithm is as follows:

### 4.5 The Progressive Algorithm

We now have everything ready for a basic algorithm to accurately compute a min-dist optimal location. That is:

**Algorithm MDOL_basic**

1. Compute $VCU(Q)$.
2. Retrieve the objects in the intersection between $VCU(Q)$ and the horizontal/vertical extensions of $Q$.
3. Derive the set of candidate locations.
4. Compute the average distance $AD(l)$ for every candidate location $l$.
5. Return the candidate location with the minimum average distance.

This is a feasible algorithm if the number of candidate locations is small. However, if there are many candidate locations, this algorithm is not efficient.
Algorithm \textit{VCU}
Input: a rectangle $Q$, a set of site $S$.
Return: $\textit{VCU}(Q)$ regarding to set $S$.

1. $\textit{VCU} = \text{whole space}$
2. \text{for } each edge of $Q$
3. \quad find a site in the search area as shown in Figure 4.2 (b).
4. \quad compute the bisector between the site and the edge and get the $h_s$ value
5. compute the search space by combining the pruning condition of all edges as shown in Figure 4.8
6. \text{for } each site $s$ within the search space
7. \quad compute $B(s, R)$, the bisector between $s$ and $Q$, and get $D(Q, s)$, the $Q$-side to $s$
8. $\textit{VCU} = \textit{VCU} \cap D(Q, s)$
9. \text{return } \textit{VCU}

---

Figure 4.9: The \textit{VCU} algorithm.

4.5.1 The Algorithm Outline

We hereby introduce a progressive algorithm, which is more efficient. The idea is illustrated in Figure 4.10. Here we see five horizontal lines and five vertical lines that partition the query region $Q$. They make 25 intersection points as candidate locations. It is easily imaginable that in practice the number of candidate locations is much larger. We first partition $Q$ at a coarse granularity, which results in a subset of candidate locations as illustrated in Figure 4.10(a). The average distance $AD(l)$ is calculated for each of such candidate locations. The location with the smallest $AD(l)$ is returned to the user as a temporary optimal location. Next, we go to a finer partitioning and introduce more candidate locations (Figure 4.10(b)). A better temporary optimal location may be found. Eventually, if we go to the finest partitioning, all candidate locations will be checked and the accurate optimal location will
be returned.

Figure 4.10: The main idea of the progress algorithm.

One may wonder: “the description above still checks all candidate locations, so what do you save?” Actually, the description is a skeleton of the algorithm, and we will enhance it by providing a pruning facility based on the lower-bound computation. That is, with a coarse partitioning, the query region $Q$ is partitioned into multiple cells (e.g. $I$, $II$, $III$, and $IV$ in Figure 4.10(a)). For each cell $C$, we can compute a lower bound of $AD(l)$ for all locations in $C$, denoted as $LB(C)$. Empowered with this lower-bound computation technique, our algorithm has the following two abilities:

- We may prune complete cells. Let $C$ be a cell, and $l$ be a candidate location that has been checked. If $LB(C) \geq AD(l)$, we know no location in cell $C$ can reach a lower average distance than that of $l$. Therefore the computation of all candidate locations in $C$ can be avoided (Figure 4.10(c)).

- Along with a temporary optimal location that is reported to the user, our algorithm can report a range of average distance values. For instance, our algorithm may not only report a temporary optimal location whose average distance is 3000 (meters), but also claim that the average distance of the real optimal location should be in the range $[2500, 3000]$. Notice that with new iterations, this range can only be shrunk. The next iteration may report a new location with range $[2800, 2900]$. This ability gives the user a choice of stopping at an accurate-enough approximate result, in case the accurate answer takes a long time to finish.
4.5.2 The Data-Independent Lower-Bound

Given a cell $C$ our task is to compute a lower bound of $AD(l)$ for all locations $l$ in $C$. At this point, we know the size of $C$ and $AD(c_i)$ for the four corners $c_i$ of $C$. But let’s assume we know no further information. That is, we do not assume any knowledge of sites in $S$ or objects in $O$. Such a lower bound is called data-independent. With the help of Lemma 4.5.1, Corollary 4.5.2 shows a first data-independent lower bound, and Theorem 4.5.3 gives a better (tighter) one. In the next subsection we will provide a data-dependent lower bound.

**Lemma 4.5.1.** For any two locations $l$ and $l'$, $AD(l') - AD(l) \leq d(l, l')$.

**Proof.** We know $AD$ shows the average distance between an object to the nearest site in $S$. If a new site is built, either at $l$ or $l'$, this average distance may be improved. For an arbitrary object $o$, the benefit from $l$ is $d(o, S) - d(o, S \cup \{l\})$ and the benefit from $l'$ is $d(o, S) - d(o, S \cup \{l'\})$. This difference in benefit is $d(o, S \cup \{l'\}) - d(o, S \cup \{l\})$. Notice that $AD(l') - AD(l)$ is the average of such a difference in benefit. We will differentiate two cases to show that this difference in benefit is never more than $d(l, l')$.

And that will finish the proof of the lemma.

If $o \notin RNN(l)$, it does not help $o$ by building a new site at $l$, i.e. $d(o, S \cup \{l\}) = d(o, S)$. Since $d(o, S \cup \{l'\}) \leq d(o, S)$, the difference in benefit is less than or equal to 0, which in turn is less than or equal to $d(l, l')$.

If $o \in RNN(l)$, the difference in benefit is $d(o, S \cup \{l'\}) - d(o, l)$. Since $d(o, S \cup \{l'\}) \leq d(o, l')$, the difference in benefit is no more than $d(o, l') - d(o, l) \leq d(l, l')$. □

This lemma leads to a straightforward lower bound of a cell given by the following corollary:

**Corollary 4.5.2.** Let the corners of a cell $C$ be $c_1, c_2, c_3,$ and $c_4$. Let the perimeter of $C$ be $p$.

$$\min_{1 \leq i \leq 4} \{AD(c_i)\} - \frac{p}{4}$$
is a lower bound of $AD(l)$ for any location $l \in C$.

Proof. Consider an arbitrary location $l \in C$. Let $c_k$ be its nearest corner. According to Lemma 4.5.1, we have: $AD(l) \geq AD(c_k) - d(l, c_k)$. It is obvious that $d(l, c_k) \leq p/4$ and $AD(c_k) \geq \min_{1 \leq i \leq 4} \{AD(c_i)\}$. So we have

$$AD(l) \geq \min_{1 \leq i \leq 4} \{AD(c_i)\} - \frac{p}{4}.$$

The corollary provides a lower bound, which can be used to prune cells. However, we can do better. Theorem 4.5.3 below shows a tighter lower bound, which leads to more pruning power.

**Theorem 4.5.3.** Let the corners of a cell $C$ be $c_1$, $c_2$, $c_3$, and $c_4$, where $c_1c_4$ is a diagonal. Let the perimeter of $C$ be $p$.

$$\max\left\{\frac{AD(c_1) + AD(c_4)}{2}, \frac{AD(c_2) + AD(c_3)}{2}\right\} - \frac{p}{4} \quad (4.6)$$

is a lower bound of $AD(l)$ for any location $l \in C$.

Proof. Due to symmetry, it is sufficient to prove that the following formula holds for every location $l \in C$,

$$AD(l) \geq \frac{AD(c_1) + AD(c_4)}{2} - \frac{p}{4}.$$

According to Lemma 4.5.1, we know:

$$AD(l) \geq AD(c_1) - d(l, c_1)$$

$$AD(l) \geq AD(c_4) - d(l, c_4)$$
Therefore, we have:

\[ AD(l) \geq \frac{AD(c_1) + AD(c_4)}{2} - \frac{d(l, c_1) + d(l, c_4)}{2}. \]

It remains to point out that \( d(l, c_1) + d(l, c_4) = p/2 \) holds for any \( l \in C \) under \( L_1 \) metric.

Figure 4.11 illustrates the superiority of this new lower bound over the straightforward one. Based on Corollary 4.5.2, we get a lower bound \( 1000 - p/4 \). Theorem 4.5.3 shows a better lower bound \( 3500 - p/4 \).

\[
\begin{array}{ccc}
AD(c_1) & = & 1000 \\
AD(c_2) & = & 3000 \\
AD(c_3) & = & 4000 \\
AD(c_4) & = & 2500
\end{array}
\]

Figure 4.11: This example illustrates the superiority of the lower bound in Theorem 4.5.3.

4.5.3 The Data-Dependent Lower-Bound

If we know some information about the objects \( O \) and the sites \( S \), we may be able to get a tighter lower bound. In the extreme case, computing the exact optimal location within the cell provides the tightest lower bound. However, it is meaningless in that the purpose of computing the lower bound is to compute the optimal location more efficiently, not the other way around. In this section, we derive a meaningful lower bound (in Theorem 4.5.5), assuming no knowledge of \( O \) or \( S \) except two values: \( |O| \) and \( |\{o| o \in VCU(C)\}| \). The former value is the total number of objects in the whole space, which can be assumed known. The latter value is the total number of objects in the \( VCU \) of cell \( C \). It can be acquired by computing \( VCU(C) \) and then performing an aggregation query on the (index of the) set of objects.
Lemma 4.5.4. Given any two locations \(l\) and \(l'\) on space,

\[
AD(l) - AD(l') \leq \frac{|\{o|o \in RNN(l')\}|}{|O|} d(l, l').
\]

Intuitively, the lemma above can be explained as follows: \((AD(l) - AD(l')) \ast |O|\) is the total extra benefit of distances to the nearest sites for all objects by building a new site at \(l'\) instead of \(l\). The objects which may benefit more from \(l'\) than from \(l\) are those objects within \(RNN(l')\). For each of such object, the maximal extra saving is no more than \(d(l, l')\). Therefore, the total extra saving is no more than \(|\{o|o \in RNN(l')\}| \ast d(l, l')\). Formally, it can be proved as follows.

Proof. According to the definition of \(AD(l)\) and \(AD(l')\), we have:

\[
AD(l) - AD(l') = \frac{1}{|O|} \sum_{o \in O} (d(o, S \cup \{l\}) - d(o, S \cup \{l'\})).
\]

We divide the summation into two cases, one for those \(o \in RNN(l')\) and one for those \(o \notin RNN(l')\).

We know

\[
d(o, S \cup \{l'\}) = \begin{cases} 
    d(o, l'), & \text{if } o \in RNN(l') \\
    d(o, S), & \text{if } o \notin RNN(l')
\end{cases}
\]

and we know

\[
d(o, S \cup \{l\}) \leq d(o, S) \text{ and } d(o, S \cup \{l\}) \leq d(o, l).
\]

Therefore, \(AD(l) - AD(l')\)

\[
= \frac{1}{|O|} \sum_{o \in RNN(l')} (d(o, S \cup \{l\}) - d(o, S \cup \{l'\})) \\
+ \frac{1}{|O|} \sum_{o \notin RNN(l')} (d(o, S \cup \{l\}) - d(o, S \cup \{l'\}))
\]

90
Based on this lemma, we can prove the following lower bound, which is better than the data-independent lower bound we got before.

**Theorem 4.5.5.** Let the corners of a cell \( C \) be \( c_1, c_2, c_3, \) and \( c_4 \), where \( c_1c_4 \) is a diagonal. Let the perimeter of \( C \) be \( p \).

\[
\max\left\{ \frac{AD(c_1) + AD(c_4)}{2}, \frac{AD(c_2) + AD(c_3)}{2} \right\} - \frac{p \cdot |\{o|o \in VCU(C)\}|}{4 \cdot |O|}
\]

is a lower bound of \( AD(l) \) for any location \( l \in C \).

**Proof.** Due to symmetry, it is sufficient to prove that

\[
\frac{AD(c_1) + AD(c_4)}{2} - \frac{p \cdot |\{o|o \in VCU(C)\}|}{4 \cdot |O|}
\]

is a lower bound.
According to Lemma 4.5.4, we have

\[ AD(c_1) - AD(l) \leq \frac{|\{o|o \in RNN(l)\}|}{|O|} d(l, c_1) \]

or

\[ AD(l) \geq AD(c_1) - \frac{|\{o|o \in RNN(l)\}|}{|O|} d(l, c_1). \]

Similarly,

\[ AD(l) \geq AD(c_4) - \frac{|\{o|o \in RNN(l)\}|}{|O|} d(l, c_4). \]

Therefore, we have:

\[
AD(l) \geq \frac{AD(c_1) + AD(c_4)}{2} - \frac{|\{o|o \in RNN(l)\}|}{|O|} \left( d(l, c_1) + d(l, c_4) \right) \]

\[ = \frac{AD(c_1) + AD(c_4)}{2} - \frac{|\{o|o \in RNN(l)\}|}{|O|} \frac{p}{4} \]

It remains to point out that since \( l \) is in cell \( C \), \(|\{o|o \in RNN(l)\}| \leq |\{o|o \in VCU(C)\}|\).

Typically, \(|\{o|o \in VCU(C)\}| \) is much smaller than \(|O|\). Therefore the data-dependent lower bound given in Theorem 4.5.5 is generally much better (larger) than the data-independent lower bound given in Theorem 4.5.3.

### 4.5.4 The Algorithm

Given the confidence in computing a lower bound on \( AD(l) \) for locations \( l \) in any cell \( C \), we are ready to provide the progressive algorithm. Let \( LB(C) \) represent the data-dependent lower bound for cell \( C \).

**Algorithm MDOL_prog**

1. Retrieve the objects in the intersection between \( VCU(Q) \) and the horizontal/vertical extensions of \( Q \). Derive the set of horizontal and vertical lines, whose intersections are the candidate locations.
2. Maintain a heap of cells ordered by $LB(\cdot)$. Initially, the heap contains one cell: the query region $Q$.

3. Set $l_{opt}$ be the corner of $Q$ with minimum $AD(\cdot)$.

4. If the heap is empty, or if $AD(l_{opt}) \leq$ the minimum $LB(\cdot)$ of cells in the heap, return $l_{opt}$ as the optimal location.

5. Remove the cell $C$ from the heap with minimum $LB(\cdot)$.

6. If $C$ cannot be partitioned, goto Step 4.

7. Partition $C$ into a set of $k$ sub-cells.

8. Compute $AD(\cdot)$ for the corners of all sub-cells, if not computed already. If any corner $c_i$ has $AD(c_i) < AD(l_{opt})$, set $l_{opt} = c_i$.

9. Compute the lower bound $LB(\cdot)$ for every sub-cell using Theorem 4.5.5.

10. For every sub-cell $C_i$ where $LB(C_i) < AD(l_{opt})$, insert $C_i$ into the heap.

11. goto Step 4.

The algorithm contains initialization (Steps 1, 2, 3) and a loop (Steps 4 through 11). It maintains a heap of cells, and a temporary optimal location $l_{opt}$. Initially, the heap has one cell: $Q$. And the temporary optimal location is initialized to be the corner of $Q$ with minimum $AD(\cdot)$. Later on, each iteration removes a cell from the heap and partitions it into sub-cells which will be re-inserted into the heap.

To fully understand the algorithm, let’s discuss several key points of it below, and discuss the cell-partitioning issue in Section 4.5.5.

**Cell Pruning using Lower Bound**

Along with each cell $C$, we keep its lower bound $LB(C)$. It is a lower bound of $AD(l)$ for every location $l$ in $C$. Therefore, if $LB(C) \geq AD(l_{opt})$, we know no location in $C$ can be a better candidate than the current temporary optimal location $l_{opt}$. So as Step 10 of the algorithm shows, we can prune the examination of a cell $C_i$ (and all candidate locations in it) if $LB(C_i) \geq AD(l_{opt})$.

**Continuous Refinement of the Query Result**

One feature of the algorithm is that it can quickly report a temporary optimal location, along with its estimate interval, and it can keep refining the result. When the
algorithm terminates, the real optimal location can be found. But the user reserves the flexibility of aborting the execution at any time when the quality of the temporary optimal location is high enough (as indicated by the estimate interval). This feature can be helpful especially when the data volume is large and the algorithm takes a long time to terminate.

The estimate interval is \([AD_{low}, AD_{high}]\). Here \(AD_{low}\) is the minimum \(LB(C_i)\) for all cells in the heap, and \(AD_{high} = AD(l_{opt})\). Let the real optimal location be \(l_{realopt}\). It is guaranteed that \(AD(l_{realopt}) \in [AD_{low}, AD_{high}]\) is true at all times. Therefore, the shorter this interval is, the more confident we are on the temporary optimal location. When the algorithm terminates, the estimate interval shrinks to a single point, and we are certain that we have found the real optimal location.

Once the algorithm starts to run, as time passes by the estimate interval shrinks. On the left side, \(AD_{low}\) keeps increasing. This is because when we remove a cell \(C\) from the heap (Step 5) and partition it to sub-cells (Step 7) to be inserted to the heap (Step 10), we know \(LB(C_i)\) of any sub-cell \(C_i\) is at least as large as \(LB(C)\). On the right side, \(AD_{high}\) keeps decreasing. This is because we only replace \(l_{opt}\) with a better candidate location. In summary, the algorithm reports better and better candidate locations whose estimate intervals keeps improving.

**The Stopping Condition**

Step 4 shows two stopping conditions. If the heap is empty, there is no unprocessed cell and therefore there is no more candidate location to examine. So the temporary optimal location is really optimal. Another condition is when all cells in the heap have \(LB(\cdot)\) at least as large as \(AD(l_{opt})\). In this case, no candidate location in any cell in the heap can be better than \(l_{opt}\), and therefore the algorithm can terminate.

One may wonder: “Step 10 has ensured that we only insert a cell \(C_i\) into the heap when \(LB(C_i) < AD(l_{opt})\). So how can it happen that the heap contains some
cell whose $LB(\cdot) \geq AD(l_{opt})$?" The answer is that $AD(l_{opt})$ shrinks as better and better temporary optimal location is found. Therefore it is quite possible that some prunable cell was validly inserted to the heap based on an old $AD(l_{opt})$.

This observation tells us an additional thing we could do, which is not specified in the algorithm. That is, whenever $l_{opt}$ changes, we remove from the heap every cell $C_i$ where $AD(C_i) \geq AD(l_{opt})$. This additional cost of eager removal can help us keep the heap compact at all times. However, we tend not to implement this eager removal for the following reason. This operation will remove some cells with the largest $LB(\cdot)$. Because the heap is optimized to extract a cell with minimum $LB(\cdot)$, it is costly to remove from the other end.

### 4.5.5 Batch Cell Partitioning

When a cell $C$ with the smallest $LB(\cdot)$ is removed from the heap (Step 5), Step 7 of the algorithm MDOL-prog simply said “Partition $C$ into a set of sub-cells”. This section addresses the issue how to partition. In fact, we directly address the extended problem, when multiple cells may be chosen to partition together.

The motivation is that to insert a sub-cell $C_i$ into the heap, we need to access the indices storing $O$ and $S$ (in order to compute $VCU(C_i)$, $LB(C_i)$, and non-computed $AD(\cdot)$ for the corners of $C_i$), which can be expensive. Therefore we want to batch the access. In other words, we want to compute the associated information for multiple sub-cells together, for each access to the indices of $O$ and $S$.

The number of new cells to process depends on the available memory. In the extreme case when we had unlimited memory, we could partition $Q$ to the finest level and compute everything by visiting the indices of $O$ and $S$ once. In practice, the allowed number of new cells to process together is not infinity. We therefore face the following **design problems**, assuming we can process $k$ (denoted as partitioning capacity) new cells together:
1. Which cells shall we partition? How many sub-cells shall we partition each cell into?

2. Given a particular cell and the number of sub-cells we aim at partitioning it into, how to partition it?

**Solution for Design Problem One**

In order to increase the minimum $LB(\cdot)$, we should definitely partition the cells whose $LB(\cdot)$’s are the smallest among all cells in the heap. In general, we want to distribute $k$ to multiple cells, and the smaller $LB(C)$ is, the more sub-cells we partition $C$ into.

We hereby propose a scheme, where the number of sub-cells of $C$ is proportional to $1/LB(C)$. Let $C_1, \ldots, C_t$ be the set of $t$ cells in the heap with the smallest $LB(\cdot)$, where $t$ is a pre-defined constant. For every cell $C_i$, we partition it into $NSC(C_i)$ sub-cells.

$$NSC(C_i) = \frac{k}{LB(C_i) \sum_{j=1}^{t} \frac{1}{LB(C_j)}}$$

(4.7)

It can be verified that

$$\sum_{i=1}^{t} NSC(C_i) = k$$

$$\frac{NSC(C_i)}{NSC(C_i')} = \frac{1/LB(C_i)}{1/LB(C_i')}$$

**Example:** Let $t = 4$, $LB(C_1) = 10$, $LB(C_2) = 10$, $LB(C_3) = 100$, and $LB(C_4) = 100$. Let $k = 44$. We have: $\sum_{j=1}^{t} \frac{1}{LB(C_j)} = \frac{1}{10} + \frac{1}{10} + \frac{1}{100} + \frac{1}{100} = 0.22$. Thus, $NSC(C_1) = NSC(C_2) = \frac{44}{10 + 0.22} = 20$ and $NSC(C_3) = NSC(C_4) = \frac{44}{100 + 0.22} = 2$.

**Solution for Design Problem Two**

Suppose we aim at partitioning a cell $C$ into $k'$ sub-cells. We need to know how to partition. There are two sub-problems here. First, we need to decide how many partitions we should make for the $X$ axis, and how many for the $Y$ axis. Second, we
need to choose the vertical and horizontal lines. What guides our design choices in both sub-problems is that we should try to:

- Make sub-cells have similar sizes (or perimeters).
- Make sub-cells be square-shaped.

The reason why we try to make sub-cells have equal sizes is: if some sub-cell has very small sizes and some others have very large sizes, the ones with large sizes, or large perimeter $p$, may have very small $LB(\cdot)$ (according to Theorem 4.5.5), contradictory to our goal of increasing $LB(\cdot)$ of cells in the heap as much as possible.

![Diagram](image)

(a) partitioning into square-like cells

(b) partitioning into thin and long cells

Figure 4.12: The example illustrating why we partition a cell into square-shaped sub-cells.

The reason why we try to have square-shaped sub-cells is illustrated in Figure 4.12. Here cell $C$ has $hu = 9$ horizontal units and $vu = 3$ vertical units. That is, if we partition $C$ into the finest level, we will get $9 \times 3 = 27$ sub-cells. Our goal is to partition $C$ into $k' = 3$ sub-cells. Figure 4.12(a) partitions $C$ into 3 square-like cells, where each cell has perimeter $p = (3 + 3) \times 2 = 12$. On the other hand, Figure 4.12(b) partitions $C$ into 3 thin-and-long cells, where each cell has perimeter $p = (1 + 9) \times 2 = 20$. According to Theorem 4.5.5, even though both approaches partition $C$ into equal number of sub-cells, the former approach tends to produce a partitioning with larger $LB(\cdot)$ for every sub-cell.
Therefore, we should aim at having $\frac{n_x}{n_y} = \frac{w}{h}$ where $n_x$ is the number of resulted partitions on the $X$ axis (of $C$), $n_y$ is the number of resulted partitions on the $Y$ axis, $w$ is the width of $C$, and $h$ is the height of $C$. On the other hand, since we aim at partitioning $C$ into $k'$ sub-cells, we have $n_x * n_y = k'$. As a result:

$$n_x = \sqrt{\frac{w \cdot k'}{h}}, \quad n_y = \sqrt{\frac{h \cdot k'}{w}} \quad (4.8)$$

The above equation has solved the first sub-problem in determining the number of partitions in $X$ and $Y$. Let’s now tackle the second sub-problem on how to partition on each dimension. Due to symmetry we focus on the $X$-axis. The task is to choose $n_x - 1$ vertical lines (so as to create $n_x$ partitions for the $X$ range of $C$).

We propose to partition in such a way that each subrange has roughly the same width. Clearly, the $n_x - 1$ hypothetical lines that form an equi-width partitioning (of the $X$ range of $C$) may not co-locate with existing vertical lines. The first-glance approach is to compute the equi-width lines, and then for every equi-width line choose the closest existing vertical line. However, as Figure 4.13 shows, the approach may not work, since multiple equi-width lines may correspond to the same existing vertical line. In the example, both the second and the third equal-width lines consider line 5 as the closest existing line. To solve this problem, we process one equi-width line at a time, from left to right. For each such line, we assign it to the closest vertical line not assigned to any previous one yet. After processing each equi-width line, we check to make sure that the number of remaining vertical lines $\geq$ the number of remaining equi-width lines. If the condition is not met, the remaining equi-width lines are matched with the right-most existing vertical lines. As an example, in Figure 4.13, after associating vertical line 4 with the first equi-width line, the above condition is not met. Therefore the last two equi-width lines are associated with the last two existing vertical lines (4 and 5). And in turn line 3 is chosen to be associated with
Figure 4.13: Matching every equi-width line (shown as a dashed line) to the closest existing vertical line may not work.

the first equi-width line.

4.6 Min-Dist OL under $L_2$ Metric

4.6.1 Overview

In this section, we adapt the solution proposed in previous sections to $L_2$ metric. In general, the framework of the progressive algorithm still applies. However, due to the difference between $L_1$ and $L_2$ metrics, some techniques specifically work for $L_1$ metric need to be rectified. In particular, there are two major changes.

- The candidates can no longer be limited under $L_2$ metric. Thus, the progressive algorithm can only work as an approximation algorithm, i.e. it does not guarantee that the exact answer can be found. However, since the estimate interval can keep getting refined, therefore, the answer can be arbitrarily close to the exact answer. A side effect of this change is that we can partition the cells in the way we prefer, for example, making the sub cells square-shaped and of equal size.

- The $VCU$ of a cell need to be computed differently. Even though $VCU$ is not needed for computing the candidate locations, it is used to compute the data-dependent lower bound of the cells.

An interesting fact is that the lower bound of a cell presented in theorem 4.5.5 still holds under $L_2$ metric. It is because the total distance from any point within a
rectangle \( R \) to two diagonal vertices of \( R \) is still bounded by half of the perimeter of \( R \). Among two changes mentioned above, the first change can be implemented straightforwardly. Therefore, the rest of this section focuses on illustrating how to compute \( VCU \) under \( L_2 \) metric.

### 4.6.2 Computing \( VCU \) under \( L_2 \) Metric

It is well known that the bisector under \( L_2 \) metric between two points \( p \) and \( q \) is a straight line perpendicular to the line segment \((p, q)\). Therefore, we start with illustrating the bisector \( B(s, E) \) between a point \( s \) and an edge \( E \). Figure 4.14 shows examples of \( B(s, E) \). It is divided into three parts. One on the left of \( u \) which is the left vertex of \( E \), one on the right of \( v \) which is the right vertex of \( E \), and one between \( u \) and \( v \). The points on left part are equidistant from \( s \) and \( u \). Therefore, the left part is a ray (half-line), which bisects \((s, u)\). Similarly, the right part is a ray (half-line), which bisects \((s, v)\). The points on the middle part are equidistant from \( s \) and \( E \). Therefore, the middle part is part of a parabola. As shown in Figure 4.14, the position of \( s \) does not change the way to calculate \( B(s, E) \). It is not the case for the bisector \( B(s, Q) \) between a point \( s \) and a rectangle \( Q \).

![Figure 4.14: The Bisector between a point and an edge under \( L_2 \) metric.](image)

Figure 4.15(a) shows, when \( s \) is in the middle of \( Q \)'s two vertical edges, \( B(s, Q) \) is the bisector between \( s \) and the top edge of \( Q \). In Figure 4.15(b), \( s \) is on the corner of \( Q \). \( B(s, Q) \) is the combination of the bisectors between \( s \) and two edges of \( Q \), \textit{i.e.}
the left edge and the top edge. Therefore, $B(s, Q)$ is divided into five pieces, each of which is either a line segment, a ray, or part of a parabola.

To compute $VCU(Q)$, we first find a site on each corner of $Q$ and get a bounding quadrangle $BQ$ of $VCU(Q)$ as shown in Figure 4.16. Then the maximum distance $\text{maxdist}(BQ, Q)$ between bounding quadrangle and $Q$ is computed. For any site $s$, the distance from any point on $B(s, Q)$ to $Q$ is at least half of the distance $d(s, Q)$ from $s$ to $Q$. Therefore, only those sites whose distance to $Q$ is less than $2\text{maxdist}(BQ, Q)$ may have a bisector intersecting with $BQ$. All those sites are retrieved to refine the $VCU(Q)$.

Figure 4.15: The Bisector between a point and a rectangle under $L_2$ metric.

Figure 4.16: Computing $VCU(Q)$ under $L_2$ metric.
4.7 Performance

In this section we experimentally evaluate our algorithm and optimizations. We use a real dataset: the 123,593 postal addresses in northeastern part of the United States (New York, Philadelphia and Boston). The dataset is available at the R-tree Portal [47]. For each experiment, given the number of sites, we randomly select some data points as the sites and use the rest as the objects. The objects are stored in an R*-tree, augmented by the distance from each object to its nearest site. The pagesize of the R*-tree is 4KB. In real applications, the number of sites is typically very small. So, in our experiments, we keep all sites in memory. However, the sites can be organized as an R*-tree and our algorithm still applies. In each experiment, we issue 100 random queries with fixed size, and take their average running time. We use a buffer of size 128 pages and measure the total disk I/Os to the object R*-tree. All experiments are performed on a Dell Pentium IV 3.2GHz PC with 4GB memory. Unless otherwise stated, the experiments use the default parameters as given in Table 4.2. The results shown here are based on $L_1$ distance. The results on $L_2$ metric are very similar and therefore are omitted.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of sites</td>
<td>100</td>
</tr>
<tr>
<td>Query size</td>
<td>1% in each dimension</td>
</tr>
<tr>
<td>Partitioning capacity ($k$)</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 4.2: The default parameters.

4.7.1 The Effect of VCU Computation

We first verify that computing the VCU of the query range significantly reduces the number of candidate locations. As Figure 4.17 shows, computing VCU reduces the number of candidates by about two orders of magnitude. The number of candidates are roughly proportional to the area of the query range for both cases. Thus, in
Figure 4.17: The effect of VCU computation.

Figure 10, two lines increase almost at the same rate when the query size increases. In the rest of experiments, we always compute VCU of the query range.

### 4.7.2 Comparison of the Three Lower Bounds

We have proposed three versions of lower bounds of $AD(l)$ for all locations $l$ in a cell $C$, as shown in Table 4.3.

To compare their pruning power, we implement three versions of the algorithm by using these three lower-bounds, respectively. The query size is 0.25% in each dimension. Figure 4.18 shows the total disk I/Os and running time of these three algorithms. When the number of sites increases, the disk I/Os and running time of all three algorithms decrease, and the gap between DDL and the other two methods also decreases. This is because the VCU of a cell shrinks with more sites and we have fewer candidates. Thus, the number of candidates that need to be pruned also becomes smaller. So DIL has a little better pruning power than SL, while DDL is clearly superior than both of them. In other words, the data-dependent lower bound has the strongest pruning power.
<table>
<thead>
<tr>
<th>Notation</th>
<th>Lower bound</th>
<th>Where</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL</td>
<td>straightforward</td>
<td>Corollary 4.5.2</td>
</tr>
<tr>
<td>DIL</td>
<td>data-independent</td>
<td>Theorem 4.5.3</td>
</tr>
<tr>
<td>DDL</td>
<td>data-dependent</td>
<td>Theorem 4.5.5</td>
</tr>
</tbody>
</table>

Table 4.3: The three lower bounds.

(a) The total disk I/Os  
(b) The total runtime

Figure 4.18: Comparison of the three lower bounds.

### 4.7.3 Impact of Lower-Bound Pruning

To see how much improvement we can get by using lower-bound pruning, we compare the query I/O performance between the naive algorithm (denoted as *naive*) which checks all candidates, and the algorithm which prunes candidates utilizing the data-dependent lower bound. As shown in Figure 4.19, using pruning can bring multiple orders of magnitude performance improvement. This is because, with the increase of the query size and the number of candidates, the difference between Naive and DLL in the number of pruned candidates becomes larger.

### 4.7.4 The Effect of Batch Partitioning

In this section, we examine how the batch partitioning affects the performance. Figure 4.20 shows the total disk I/Os with respect to the batch-partitioning capacity, i.e. the number of new sub-cells we introduce in a single run. When the batch capacity increases, the total disk I/Os first decrease. That is because, with a larger batch...
capacity, we can compute more $AD(\cdot)$ and lower bounds at one time. However, when the batch capacity is too large, the performance becomes worse. The reason is that the cells are divided into too fine a granularity. Some disk I/Os are wasted on computing the $AD(\cdot)$ and VCUs for those sub-cells, which could be pruned by using a coarser granularity.

4.7.5 The Progressiveness

Our progressive algorithm quickly reports a temporary optimal location, and keeps refining the result. In this section we examine how fast the quality of the query result can improve. Since queries are randomly generated, the optimal locations of different queries may have different $AD(\cdot)$ values. To measure the progressiveness in a unified
way, we normalize the result of each query as follows: the final result (i.e. $AD(\cdot)$ of the actual optimal location) is scaled to be 0, the minimal $AD(\cdot)$ of the four corners of the query range is scaled to be 1, and the initial lower bound of the query range is scaled to be -1. After each batch process, we get a new upper bound and lower bound as the intermediate result. They are scaled to real numbers between [-1,1]. We run 100 random queries with the default parameters, take the first 100 steps for each run, scale them to the [-1,1] range and report the average. The result is plotted in Figure 4.21. The (scaled) upper bound approaches the actually result very fast. It will become less than 1% within 20 steps. The (scaled) lower bound, however, approaches the actual result slower. It will become less than 1% in about 80 steps. (Here on average it takes an algorithm 200 steps to find the exact answer.) It tells us that, in many cases, we find a good approximation very fast, but it takes some time to verify whether it is the actual real optimal location.

### 4.8 Conclusions

This chapter solved the Min-dist OL query under $L_1$ and $L_2$ metric. Even though there is an infinite number of locations in a query range, we proved that under the $L_1$ metric we only need to check a finite number of candidates to get an exact answer. We then proposed a progressive algorithm, MDOL$_{prog}$, that first partitions the
query range into some cells, and then recursively partitions each cell into smaller cells. Since any candidate location is the corner of some cell, by partitioning to the finest granularity it is guaranteed that we can find the optimal location. We introduced three lower-bound estimators which enable the pruning of complete cells (and all candidate locations in these cells). The lower-bound estimators provide the progressive nature of the algorithm, as results from earlier runs can be used to prune the search space of later runs. Finally, we proposed the batch-partitioning method. To apply the progressive algorithm under $L_2$ metric as an approximation algorithm, we proposed new method to calculate the $VCU$. Although the exact answer may not be found, the answer can be as accurate as the user wants. Experimental results revealed that one of the three lower-bound estimators, namely the data-dependent lower bound, is clearly better than the other two, and that our progressive algorithm $MDOL_{prog}$ is efficient.
Chapter 5

OLs On Road Network

5.1 Overview

We discuss the OL queries on road network in this chapter. The road network is modeled as a graph \( G = (V, E) \). \( V \) is the set of \( n \) vertices, and \( E \subseteq V \times V \) is the set of edges, each of which is an ordered pair of vertices. For ease of presentation, we assume \( G \) is undirected. Nevertheless, our methods can be effortlessly extended to the case of directed graph. In addition, we assume the space embedding the network is a Euclidean space, which implies that the network distance between any two vertices cannot be shorter than their Euclidean distance. Furthermore, we allow objects located not long in vertices but also on edges. We say a location \( l \) is on the network if \( l \) is either a vertex of \( G \) or on one edge of \( G \). The OL queries on road network considers a set \( S \) of sites and a set \( O \) of objects, both of which are on the network, and finds the \( OLs \) on the network as well.

In the discussion of this chapter, we assume that the road network is stored using the CCAM method surveyed in related work. In particular, for each vertex \( v_i \), the information to be stored on disk includes its location and an adjacent list. Each entry in the adjacent list stores the information of the edge from \( v_i \) to the neighbor \( v_j \), which
includes the location of $v_j$, the polyline approximation of the edge from $v_i$ to $v_j$, and the sites located on this edge. To cluster the information of vertices in disk pages, the Hilbert values of the vertices are used to generate the one-dimensional ordering of all vertices. An B+-tree is kept on top of the vertices to efficiently locate the information of any vertex. Meanwhile, the objects are indexed separately by R*-tree. Each object is stored with its location and the edge on which it is located. Comparing the the underlying road network and sites, the objects are updates more frequently. The separation of their storage provides more flexibility and better support to updates.

![Figure 5.1: The $nn$ buffer under network metric.](image)

Under network metric, the concept of $nn$ buffer introduced in section 3 still holds. However, the $nn$ buffer of an object is no longer in a regular shape such as a diamond or a circle. Instead, it becomes a fraction of the network centered at the object. For example, in Figure 5.1, the $nn$ buffer of $o_1$ is the part of the network inside the dashed circle. Therefore, none of the previous methods applies to the network metric. Nevertheless, the optimal location is still the location with maximum overlap among $nn$ buffers. In the rest of this chapter, we first present a basic solution to the Max-Inf OL based on pre-computing the $nn$ buffers of all objects, then show methods to improve it, and finally we extend the solution to the Min-Dist OL query.
5.2 Basic Solution to Max-Inf OL

Our basic solution logically pre-computes and stores the \textit{nn\_buffer} of each object. Since the \textit{nn\_buffer} of an object is part of the network, we integrate it into the presentation of the network. Specifically, for each object \( o \), we pre-compute \( d(o, S) \), the shortest path distance to the nearest site. For each vertex \( v \), we maintain a \textit{coverage set} \( C(v) \) of objects whose \textit{nn\_buffers} cover \( v \). For every object \( o \) in the set \( C(v) \), we also store \( d(o, v) \), the shortest path distance from \( o \) to \( v \).

The pre-computation step is straightforward. Conceptually, we apply any single-source shortest path algorithm, such as Dijkstra’s algorithm, on each object \( o \). For each vertex \( v \) visited in the search, we add \( o : d(o, v) \) into the coverage set \( C(v) \). The search stops when the first cite is explored in the search, and the distance \( d(o, S) \) is discovered. The pre-computation of multiple objects, for example the objects on the same edge, can be combined to speed up the pre-computation.

5.2.1 Query Processing Algorithm

With the pre-computed information, we are able to answer the Max-Inf optimal location queries efficiently.

Since the set \( C(v) \) is maintained for each vertex, the influences of all vertices are known. Thus, it remains to check whether there is a better location somewhere in the middle of an edge. We propose a method working as follows. Given a query region \( Q \), it examines each edge \((v, w)\) intersecting with \( Q \) in the decreasing order of its maximal coverage \( MS(v, w) = |C(v)| + |C(w)| \), \textit{i.e.} the sum of the size of \( C(v) \) and the size of \( C(w) \). It computes the local optimal location along the edge. If the new local optimal location is inside \( Q \) and has a larger influence than currently known optimal location, the currently known optimal location is updated to the new location. The algorithm repeats the above procedure until the maximal coverage size of the edge is no more
Algorithm *BasicNet*

Input: The road network $G$, query region $Q$.

Return: The Max-Inf OL in $Q$.

1. Retrieve all edges intersecting with $Q$ to $E$.
2. Sort edges in $E$ in the decreasing order of their maximal coverage.
3. $CurOL = NULL$.
4. For $i = 1$ to $\text{size}(E)$
5. Let $e_i$ be the $i$th edge in $E$.
6. If $MS(e_i) \leq CurOL.inf$
   return $CurOL$.
7. $LocOL = \text{FindLocOL}(e_i)$.
8. If $LocOL.inf > CurOL.inf$
   $CurOL = LocOL$.
9. return $CurOL$.

Figure 5.2: Finding Min-Inf OL in $Q$.

than the influence of the currently known optimal location. Then the currently known optimal location is returned as the answer. Since the influence of any location on an edge cannot be larger than the maximal coverage of the edge, the correctness of the algorithm is straightforward. Figure 5.2 gives the formal description of the algorithm.

![Figure 5.3: An example of the Max-Inf optimal location query.](image)

An example in Figure 5.3 shows the network intersecting with the query region, which is part of a larger network. There are five vertices, nine edges and three objects. The edges fully within query region are labeled with their lengths and each object is labeled with the shortest distance to the nearest site. Assume that there are no other object which affects the influence of locations in the query region. After the pre-computation, we have $C(v_1) = \{o_1 : 1\}$, $C(v_2) = \{o_1 : 3\}$, $C(v_3) = \{o_2 : 1, o_3 : 4\}$,
\[ C(v_4) = \{ o_1 : 3 \}, \text{and} \; C(v_5) = \{ o_3 : 1 \}. \]

The algorithm starts with the edge \((v_2, v_3)\) or \((v_3, v_5)\), whose maximal coverage is 3. Since \(d(o_1, S)\) is 5 and \(d(o_1, v_2)\) is 3, it is easy to get that \(nn\_buffer\) of \(o_1\) covers the left \(\frac{2}{3}\) of \((v_2, v_3)\). Similarly, \(nn\_buffers\) of \(o_2\) and \(o_3\) cover the right \(\frac{2}{3}\) of \((v_2, v_3)\). Therefore, the local optimal location on edge \((v_2, v_3)\) is any point in range \((1,2)\) of the edge with influence 3. Similarly, the local optimal location on edge \((v_3, v_5)\) is any point in range \((0,2)\) from vertex \(v_3\) with influence 2 (covered by \(o_2\) and \(o_3\)). The other edges can be pruned because their maximal coverage is at most 2, which is less than the influence of current known optimal location. Therefore, the final answer is any point in range \((1,2)\) of the edge \((v_2, v_3)\).

### 5.2.2 Finding Local OL on Edges

In the algorithm described in Figure 5.2, we did not specify the details of the procedure \(FindLocOL(e)\). For simple graphs as the example above, we can find the local OL on edges with ease. However, it becomes complicated when there are many objects covering the vertices of an edge. Figure 5.4 shows the outline of this procedure.

**Procedure \(FindLocOL(e)\)**

1. \(R = \emptyset\).
2. For each object \(o \in C(v) \cup c(w)\)
   (a) If \(e\) is not fully covered by \(o\)
   (b) If \(o \in C(v)\), add the range starting from \(v\) and covered by \(o\) into \(R\).
   (c) If \(o \in C(w)\), add the range starting from \(w\) and covered by \(o\) into \(R\).
3. Sort the ranges in \(R\) according to the locations of the ends other than \(v\) and \(w\).
4. Go through sorted ranges to return the location covered by the most number of ranges.

Figure 5.4: Finding local OL on an edge.

Let us explain the procedure in more details. Consider an edge \(e = (v, w)\). Any location on \(e\) can be only covered by objects in \(C(v)\) or \(C(w)\). Furthermore, only the object partially covering \(e\) may have different influence on different locations on \(e\).
Therefore, the procedure $\text{FindLocOL}$ first eliminates the objects which fully cover $e$ (step 2(a)). Specifically, an object $o$ fully covers $e$ if and only if $o \in C(v)$, and $o \in C(w)$ and $(d(o, S) - d(o, v)) + (d(o, S) - d(o, w)) \geq d(v, w)$. For the rest of the objects, the range starting from $v$ and/or $w$ and covered by $o$ is added into a set $R$, which is initially empty. Assume that the range of $(v, w)$ is $[0, d(v, w)]$ . If $o \in C(v)$, the range $[0, d(o, S) - d(o, v))$ is added to $R$. We call this kind of ranges $v$-type ranges. If $o \in C(w)$, the range $(d(v, w) - (d(o, S) - d(o, v)), d(v, w)]$ is added to $R$. We call this kind of ranges $w$-type ranges. Next step is to sort the ranges. More precisely, the ranges are sorted according to the ends which are neither $v$ or $w$. These ends divides $(v, w)$ into small segments. Starting from $v$ with influence being $|C(v)|$, we go through all ends to check possible influences of each segment. Whenever the end of a $v$-type range is passed, the influence of next segment is decreased by 1. On the contrary, the influence is increased by 1 if the end of a $w$-type range is passed. After checking all ends, we report (one point of) the segment with largest influence as the local optimal location. Note that if the edge is not fully contained by the query region $Q$, only the segments inside $Q$ need to be checked.

### 5.3 Improved Solution to Max-Inf OL

The solution above is efficient with regards to the running time, however it has two major shortcomings. The first one is that the storage requirement is huge. In the worst case, the set $C(v)$ of each vertex $v$ has size $|O|$. Therefore the total extra storage for the whole network is $O(|OV|)$, which is prohibitively expensive if the number of objects is huge. The second one is that the update is inefficient. Whenever an object is inserted, or deleted, or updated its location, all vertices covered by it need to be updated. We propose two methods to address these two problems.

1. **Pre-computing on vertex:** we use the pre-computation on each vertex $v$
to replace the pre-computation on each object $o$. Specifically, we pre-compute $d(v, S)$ instead of $d(o, S)$, and the vertex coverage set $VC(v)$ instead of $C(v)$. The vertex coverage set $VC(v)$ of a vertex $v$ contains vertices whose $nn\_buffers$ cover $v$ and their distance to $v$. This technique makes the storage cost reduced dramatically when the number of objects dominates the number of vertices, which is the most common case in real applications. Another great advantage is that the pre-computation becomes irrelevant to the objects, which makes the updates of objects no longer be a problem.

2. **Removing redundancy:** we compress the $C(v)$ or $VC(v)$ sets by getting rid of the redundancy in nearby vertices. Specifically, given an object $o$ (or vertex $w$), we do not store it in $C(v)$ (or $VC(v)$) of every vertex $v$ covered by $o$ (or $w$). Instead, we store it in a carefully chosen subset. This technique further reduces the storage with tunable tradeoff between the storage cost and the running time.

These two methods can be used separately or combined, depending on the size of graph and datasets. The cost of them is that the query processing algorithm gets more complex and takes more time than the basic solution. In the rest of this section, we show the details of them and the changes of the basic solution to accommodate the improvements.

### 5.3.1 Pre-computing on Vertices

The pre-computation step of this method is straightforward. We just need to apply the single-source shortest path algorithm on each vertex instead of each object. In order to reuse the basic solution, we need to restore $d(o, S)$ and $C(v)$ from $d(v, S)$ and $VC(v)$ on the fly. Consider an object $o$ on edge $(v, w)$. The path from $o$ to any site goes through either $v$ or $w$. Therefore, $d(o, S)$ can be computed as $\min\{d(o, v) + d(v, S), d(o, w) + d(w, S)\}$. The remaining problem is how to restore $C(v)$ from $VC(v)$.
If an object \( o \) is in \( C(v) \), at least one end of the edge on which \( o \) is located must be in \( VC(v) \). Therefore, we go through the edges which have at least one end in \( VC(v) \) and check the objects on those edges. If both ends of an edge is in \( VC(v) \), \( o \) must be in \( C(v) \). Otherwise, we compare the distance from \( o \) to \( v \) with \( d(o, S) \) to decide if \( o \) is in \( C(v) \) or not. Figure 5.5 gives the details of this procedure. We could plug this procedure into the basic solution to compute the \( C(v) \) of the vertices. The rest of the algorithm keeps unchanged.

**Procedure comp\( C(v) \)**

1. \( C(v) = \emptyset \).
2. **For** each edge \( e = (u, w) \) such that \( u \in VC(v) \) or \( w \in VC(v) \)
3.   **If** both \( u \) and \( w \in VC(v) \)
4.     Add every object \( o \) on \( e \) to \( C(v) \) with \( d(o, v) = \min\{d(o, u) + d(u, v), d(o, w) + d(w, v)\} \).
5.   **else**
6.     **If** \( u \in VC(v) \)
7.     Add objects \( o \) on \( e \) to \( C(v) \) with \( d(o, v) = d(o, u) + d(u, v) \) if \( d(o, v) < d(o, S) \).
8.     **If** \( w \in VC(v) \)
9.     Add objects \( o \) on \( e \) to \( C(v) \) with \( d(o, v) = d(o, w) + d(w, v) \) if \( d(o, w) < d(o, S) \).

Figure 5.5: Compute \( C(v) \) from \( VC(v) \).

### 5.3.2 Removing Redundancy

The redundancy removal method can be applied to both coverage sets and vertex coverage sets. For simplicity, we assume that \( VC(v) \) is pre-computed for each vertex \( V \) in the remaining discussion. The other case can be handled in a very similar way.

Given a vertex \( v \) and a set \( U \) of vertices covered by \( v \), this method select a subset \( U' \subset U \) to store \( v \) in their \( VC() \). There are many ways to select the subset \( U' \). For example, we could randomly pick a certain number of vertices from \( U \). However, a poor subset selection policy may make the restoral of the missing vertices in the \( VC() \) inefficient. In the worst case, it could be as inefficient as recomputing from scratch. To avoid this problem, we propose two policies: 1) the subset is picked to guarantee
that, for every vertex \( w \in U \), \( v \) and \( d(v, w) \) can be restored by checking vertices whose distance to \( w \) is at most \( \delta \). 2) the subset is picked to guarantee that, for every vertex \( w \in U \), \( v \) and \( d(v, w) \) can be restored by checking vertices at most \( \gamma \) hops away from \( w \).

To impose these policies, the pre-computation need to be slightly modified. Let us use the policy 1 as the example. We still apply the single-source shortest path algorithm on each vertex \( v \). However, for each vertex \( w \) visited in the search, we do not necessarily add \( v : d(v, w) \) into the vertex coverage set \( VC(w) \). Instead, we keep tracking the distance from \( w \) to the vertex \( w' \), which is the last vertex on the shortest path from \( v \) to \( w \) that \( v \in VC(w') \). We skip adding \( v \) to \( VC(w) \) until the distance \( d(w', w) \) is greater or equal than \( \delta \). Figuratively, \( v \) is the center and the selected subset forms “ripples” around \( v \) until it reaches the boundary of the \( nn\_buffer \). The second policy can be imposed in the similar way. The only difference is the number of hops from \( w' \) will be tracked.

**Procedure** \( compVC(v) \)

1. Push each neighbor \( n_i \) of \( v \) into priority queue \( P \) with key \( d(v, n_i) \).
2. **While** \( P \) has some vertex with key smaller than \( \delta \)
3. Pop from \( P \) the vertex \( w \) with the smallest \( d(w, v) \), which is the key.
4. If \( w \notin VC(v) \)
5. Add \( w : d(w, v) \) into \( VC(v) \).
6. **For all vertices** \( x \in VC(w) \)
7. If \( x \notin VC(v) \) and \( d(x, w) + d(w, v) < d(x, S) \)
8. Add \( x \) to \( VC(v) \) with value \( d(x, w) + d(w, v) \).
9. If \( x \in VC(v) \) and \( x.value > d(x, w) + d(w, v) \)
10. Update \( x.value \) to \( d(x, w) + d(w, v) \).
11. **For all neighbors** \( y \) of \( w \)
12. If \( y \notin P \)
13. Push \( y \) to \( P \) with key \( d(y, w) + d(w, v) \)
14. If \( y \in P \) and \( y.key > d(y, w) + d(w, v) \)
15. Update \( y.key \) to \( d(y, w) + d(w, v) \).

**Figure 5.6:** Restore \( VC(v) \).
Figure 5.6 describes the details of the procedure to restore the $VC(v)$ of a vertex $v$. Similar to Dijkstra’s algorithm, it uses a priority queue to maintain the vertices to be visited. However, it only visits vertices whose distance to $v$ is at most $\delta$. For each vertex being visited, it checks if any vertex in its $VC()$ can be added to $VC(v)$ or has a better value to update $VC(v)$.

### 5.4 Solution to Min-Dist OL

To handle Min-Dist OL, the basic solution of Max-Inf OL need to be redeveloped, while the improvements do not need any change. Since $C(v)$ is stored or can be efficiently computed for each vertex $v$, the distance from $o$ to $v$ is known for all objects whose $nn\_buffers$ cover $v$. Therefore, the $AD()$ of each vertex can be computed. Similarly to the solution of Max-Inf OL query, we need to consider the local optimal location in the middle of the edges.

**Theorem 5.4.1.** Consider an edge $(v, w)$. The local Min-Dist optimal location $l^*$ can only be $v$, $w$, or the locations the objects on $(v, w)$. The lower bound of $AD(l^*)$ is $\min\{AD(v), AD(w)\} - \frac{n_e \cdot d(v, w)}{|O|}$, where $n_e$ is the number of objects on edge $e$.

**Proof.** Consider an arbitrary location $l$ on edge $(v, w)$. We use the same technique as we used in proof of Theorem 4.3.2 (which limits the OL to the intersection points of some vertical and horizontal lines): trying to move $l$ to a better location. Similar to the argument in Theorem 4.3.2, we can move $l$ toward either $v$ or $w$ depending on which direction benefits more objects. If there is no object on edge $(v, w)$, $l$ can be moved to $v$ or $w$ to get better average distance. If there is some object on edge $(v, w)$, $l$ may stop at the location of certain object. Therefore, the local optimal location $l^*$ is either one of $v$ and $w$, or among the locations of these objects.

With the same reason, If there were no object on the edge, $AD(l^*) \geq \min\{AD(v), AD(w)\}$. For each object $o$ on the edge, moving $l^*$ to $v$ or $w$ will at most increase
\[ d(o, S) \] by \( d(v, w) \). Therefore, according to the definition of \( AD() \), the largest difference between \( AD(l*) \) and \( \min\{AD(v), AD(w)\} \) is \( \frac{n_e \cdot d(v, w)}{|O|} \).

Guided by theorem 5.4.1, we can develop the basic solution of Min-Dist OL queries. The algorithm is shown in Figure 5.7. Essentially, it is the same as the basic solution of Max-Inf OL queries. Only the key to sort the edges are different.

**Algorithm NetMin**

Input: The road network \( G \), query region \( Q \).

Return: The Min-Dist OL in \( Q \).

1. Retrieve all edges intersecting with \( Q \) to \( E \).
2. Sort edges in \( E \) in the increasing order of the lower bound of \( AD(l*) \).
3. \( CurOL.AD = \infty \).
4. For \( i = 1 \) to \( \text{size}(E) \)
5. Let \( e_i \) be the \( i \)th edge in \( E \).
6. If \( CurOL.AD \leq \) the lower bound of \( AD(l*) \) on \( e_i \)
   return \( CurOL \).
7. \( LocOL = \text{FindLocOL}(e_i) \).
8. If \( LocOL.AD < CurOL.AD \)
   \( CurOL = LocOL \).
9. return \( CurOL \).

Figure 5.7: Finding Min-Dist OL in \( Q \).

### 5.5 Performance

In this section we experimentally evaluate the basic algorithm and the proposed optimizations. Our evaluation is performed using real data for the road network and synthetic data for the objects and sites. In particular, our road network is built on a real dataset of 20,461 edges and 14,456 nodes, representing all roads in the Suffolk county of Massachusetts. The dataset is extracted from U.S. Census Bureau, 2003 Tiger/Line database and normalized to the unit space \([0, 1] \times [0, 1] \). The objects and sites are generated by the procedure, which first randomly picks an edge and then randomly select a location on the edge. As we mentioned at the beginning of this
chapter, the road network and the sites are stored together using the CCAM method. The objects are indexed separately by R*-tree. In experiments, we only show the results of Max-Inf OL query. The results of Min-Dist OL query are very similar and therefore ignored. All programs are written in C++ and performed on a Dell Pentium IV 3.2GHz PC with 4GB memory. Unless otherwise stated, the experiments use the default parameters as given in Table 5.1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of sites</td>
<td>100</td>
</tr>
<tr>
<td>Number of objects</td>
<td>500000</td>
</tr>
<tr>
<td>Query size</td>
<td>1% in each dimension</td>
</tr>
<tr>
<td>$\delta$</td>
<td>1% of one unit</td>
</tr>
</tbody>
</table>

Table 5.1: The default parameters.

5.5.1 The Effect of Pre-computation on Vertex

To examine the effect of pre-computing on vertex, we compare the storage costs and running time between Basic, the basic solution, and VTX, the solution with pre-computing on vertex. We vary the number of objects and the query size to compare the storage costs and running time respectively. Figure 5.8 shows the results. As we expected, the storage cost of VTX is irrelevant to the number of objects, while the storage cost of Basic is almost linear to the number of objects. The running time of Basic beats VTX. However, the gap becomes smaller when the query size increase. It implies that the cost of restoring $C(v)$ from $VC(v)$ does not dominate the total cost.

5.5.2 The Effect of Redundancy Removal

To examine the effect of redundancy removal, we compare the storage costs and running time between VTX, the solution with pre-computing on vertex and VTX+RRM, the solution with both improvements. We vary the parameter $\delta$ to compare the stor-
age costs and running time. Figure 5.9 shows the results. When $\delta$ increase, the storage cost of VTX+RRM decreases but slows down when $\delta$ is large. The trend of the running time is right the opposite. It indicates that there is a tradeoff between the storage cost and running time. In general, we could find a good balance point, which has a small storage cost with affordable running time. In our dataset, the balance point is around $\delta = 1\%$.

5.6 Conclusions

In this chapter we solved both the Max-Inf OL query and the Min-Dist OL under $L_2$ metric. We propose a unified method, which pre-computes the $nn\_buffers$ of each
object and stores them with the network. We proposed two methods to avoid the high storage cost and inefficiency in updates. From the experiments, these two methods with carefully chosen parameter can greatly improve the performance of the basic solution.
Chapter 6

Conclusions

In this thesis, we have studied the Max-Inf and Min-Dist OL queries under $L_1$, $L_2$ and network distance metrics. We first proposed two solutions for Max-Inf OL under $L_1$ metric. One is R*-tree based, which does not require any pre-computation. The other utilizes a novel data structure called VOL-tree to store some pre-computed results. We extended both solutions to $L_2$ metric. We then proposed a solution for Min-Dist OL under $L_1$ metric that limits the candidate locations and progressively returns the answer. Without the ability to limits the candidate locations under $L_2$ metric, this solution can only be adapted into an approximation algorithm under $L_2$ metric. Nevertheless, the answer can be as accurate as desired. Finally, we proposed a solution framework that can be used to answer both Max-Inf and Min-Dist OL queries. It stores the precomputed results in a very compact way so that both the time efficiency and the space efficiency can be achieved.

There are a few directions to continue the research in this thesis. The first direction is to solve the Min-Dist OL queries under $L_2$ metric accurately. The second is to extend the solutions to the cases where multiple OLs need to be found simultaneously. Another direction is to handle the data with frequent updates.
Bibliography


