Efficient Algorithms for Optimal Location Queries in Road Networks

Zitong Chen†, Yubao Liu†, Raymond Chi-Wing Wong‡, Jiamin Xiong†, Ganlin Mai†, Cheng Long‡
†Sun Yat-Sen University, China
‡The Hong Kong University of Science and Technology, Hong Kong, China

ABSTRACT

In this paper, we study the *optimal location query* problem based on road networks. Specifically, we have a road network on which some clients and servers are located. Each client finds the server that is closest to her for service and her cost of getting served is equal to the (network) distance between the client and the server serving her multiplied by her weight or importance. The optimal location query problem is to find a location for setting up a new server such that the maximum cost of clients being served by the servers (including the new server) is minimized. This problem has been studied before, but the state-of-the-art is still not efficient enough. In this paper, we propose an efficient algorithm for the optimal location query problem, which is based on a novel idea of *nearest location component*. We also discuss three extensions of the optimal location query problem, namely the optimal multiple-location query problem, the optimal location query problem on 3D road networks, and the optimal location query problem with another objective. Extensive experiments were conducted which showed that our algorithms are faster than the state-of-the-art by at least an order of magnitude on large real benchmark datasets. For example, on our largest real datasets, the state-of-the-art ran for more than 10 hours but our algorithm ran within 3 minutes only (i.e., >200 times faster).

1. INTRODUCTION

Location-based analysis is very important and prevalent nowadays. At present, there are tools for location-based analysis on road networks (`http://www.esri.com/software/arcgis/extensions/networkanalyst`). A fast query response is expected in an interactive setting of these tools. At the same time, many mobile devices with limited memory are installed with various mobile applications for location-based analysis. In this paper, we study one type of location-based analysis, namely the *optimal location query* (OLQ) problem.

Given a set C of clients and a set S of servers in a road network $G = (V, E)$ where V is a vertex set and E is an edge set, an optimal location query (OLQ) is to find a location such that when a new server is set up at this location, a certain *cost* function computed based on the clients and servers (including the new server) is optimized. This optimal location query is very important since it is used as a basic operation in many real applications such as location planning, location based service and profile-based marketing [1, 17, 5].

In [17], the optimal location query problem with the following cost function is studied. Given a road network, a set of clients and a set of servers, the cost of serving the the clients is defined to be the *maximum* (network) distance from a client to its closest server. Then, the optimal location query problem which minimizes the above cost function is called the *MinMax query*. The intuition of MinMax query is to optimize the *worst-case* cost of a client, which has many applications in real life. For example, in an emergency scenario, it is often required to optimize the worst-case cost (e.g., the worst-case time of getting the ambulance/fire-station/police service). An algorithm was designed for the MinMax query in [17], whose major idea is to first augment the road network by creating a vertex for each client and each server in the road network and then partition the augmented road network into sub-networks/subgraphs for solving the problem.

However, the algorithm in [17] has several shortcomings as follows. First, the algorithm relies on an augmented road network which could be prohibitively large. Specifically, the augmented road network has the number of its vertices as large as $|V| + |S| + |C|$ and the number of its edges as large as $|E| + |S| + |C|$, both of which become very large when there are a large number of servers and/or clients. Second, the algorithm has its time complexity of $O((|V| + |S| + |C|)^2 \log(|V| + |S| + |C|))$ which is prohibitively expensive. Third, the algorithm involves a partitioning procedure which heavily depends on the quality of a partition parameter and an improper setting of the parameter would result in a very long running time. For example, the experimental results in [17] show that the running time of the algorithm with an "improper" setting could be three times longer than that with the "best" setting.

Motivated by the shortcomings of the existing approach in [17], in this paper, we design an efficient algorithm called *MinMax-Alg* which avoids the above shortcomings. Specifically, we make the following contributions.

Firstly, we propose an efficient algorithm for the optimal location query in road networks. The proposed query algorithm is executed on the original road network without generating any new road network where the number of the vertices to be examined is equal to $|V|$. We also present several new pruning techniques based on the idea of *nearest location component (NLC)* of the clients, which can dramatically reduce the algorithm search space. In this paper, we focus on the query algorithm MinMax-Alg using these new pruning techniques. The time complexity of our query algorithm is significantly smaller than that of the best-known algorithm [17]. In particular, the time complexity of our MinMax-Alg

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD '14, June 22 - 27 2014, Snowbird, UT, USA
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.

is $O(\gamma \cdot |V| \log |V| + |V| \cdot |C| \log |C|)$ where γ is at most $|C|$ and is usually much smaller than $|C|$ in practice. In our experiments with the default setting on the *SF* (San Francisco) real dataset [17] where $|C|$ is 300k and $|E|$ is 223k, γ is equal to 27.

Secondly, we discuss three extensions to our problem. (1) We study to find multiple locations (instead of a single location) for the optimal location query, which has not been studied in the literature. We show that this problem is NP-hard and propose a greedy algorithm (GA) for this general problem. (2) We discuss the optimal location query on a three-dimensional (3D) road network [10]. (3) We extend the techniques based on NLC to handle the optimal location query with another objective.

Thirdly, we conducted extensive experiments to verify the efficiency of our algorithm. Our algorithm is significantly faster than the best-known algorithm by at least an order of magnitude on the datasets with large sizes. For example, on our largest datasets, the best-known algorithm in [17] runs for more than 10 hours but our algorithm (i.e., MinMax-Alg) runs within 3 minutes only (i.e., >200 times faster).

The rest of this paper is organized as follows. Section 2 gives the problem definition and Section 3 reviews the related work. Section 4 introduces our method of building the NLCs of the clients and Sections 5 introduces our algorithm MinMax-Alg. Sections 6 discusses three extensions of our problem. Section 7 gives the empirical study and Section 8 concludes the paper.

2. PROBLEM DEFINITION

Let $G = (V, E)$ be a road network, and C (S) be a set of clients (servers) on G. For any edge $e = (v_l, v_r)$ of G, v_l (v_r) is the left (right) vertex of e.

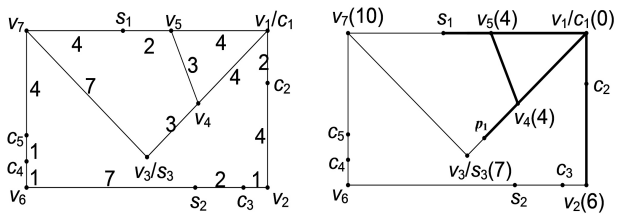
We adopt the network distance metric to define the distance between two locations on the road network, denoted by $d(\cdot, \cdot)$. Let c be a client in C. We denote c's closest server in S by $NN_S(c)$. Besides, we denote the distance between c and its closest server in S by $c.dist$, i.e., $c.dist = d(c, NN_S(c))$. Each client $c \in C$ is associated with a positive weight, denoted by $w(c)$, which denotes the importance of the client. For example, if c is a residential estate, then $w(c)$ could be set to the number of residents living at c. We define the *cost value* of c, denoted by $Cost(c)$, to be $w(c) \cdot c.dist$.

The optimal location query (OLQ) [17] problem is to find a location such that once a new server is set up at this location, the maximum cost value of the clients based on the servers (including the new server) is minimized. Formally, the OLQ problem is defined in Problem 1.

PROBLEM 1. *Given a road network $G = (V, E)$, a set C (S) of clients (servers) on G, the optimal location query problem is to find a location p which minimizes* $\max_{c \in C}\{w(c) \cdot d(c, NN_{S \cup s(p)}(c))\}$, *where $s(p)$ denotes the new server located at p. We also call this problem the* MinMax query.

Consider an example of a road network G in Figure 1(a). In this figure, each line segment corresponds to an edge and each dot corresponds to a vertex or a client or a server in the road network. In this example, there are 7 vertices, namely $v_1, v_2, ..., v_7$, 3 servers, namely s_1, s_2 and s_3, and 5 clients, namely $c_1, c_2, ..., c_5$. The number near to each line segment in the figure denotes the distance between the two end-points of the line segment. Since c_1 (s_3) has the same location as vertex v_1 (v_3) in the network, we write "v_1/c_1" ("v_3/s_3") in the figure.

Given two points p_1 and p_2 on an edge $e = (v_l, v_r)$, we define a *point interval* on e in the form of $[p_1, p_2]$ and p_1 (p_2) is said to be the *start point* (*end point*) of this interval. Note that a point interval is a portion (or a whole portion) of an edge. Suppose that the



(a) $G(V, E)$ (b) $NLC(c_1)$

Figure 1: A Running Example

start point p_1 is nearer to the left vertex v_l than the end point p_2. We use a *number interval* to denote this point interval. Specifically, the number interval for a point interval $[p_1, p_2]$ is defined to be the closed real interval $[a, b]$, where $a = d(v_l, p_1)$ and $b = d(v_l, p_2)$. For example, the point interval $[p_1, v_4]$ on edge (v_3, v_4) in Figure 1(b) could be represented by $[1, 3]$ if $d(v_3, p_1) = 1$ and $d(v_3, v_4) = 3$. It is easy to verify that given a point interval, we can derive its corresponding number interval in $O(1)$ time. In the following, we write these two terms, the point interval and the number interval, interchangeably.

Next, we introduce a key concept used in this paper, which is called *nearest location component* (NLC).

DEFINITION 1. *For each client $c \in C$, the* nearest location component *of c, denoted by $NLC(c)$, is defined to be a set of all points on the edges in G such that each of these points has its distance to c at most $c.dist$. Formally, $NLC(c) = \{p | d(c, p) \leq c.dist$ and p is a point on the edges of $G\}$.*

For example, in Figure 1(b), all bold lines correspond to $NLC(c_1)$, where each point along one of these lines has its distance to c_1 at most $c_1.dist = 6$ (note that $NN_S(c_1)$ is s_1 and thus $c_1.dist = d(c_1, s_1) = 6$). The number in the bracket near to the vertex denotes the distance between the vertex and c_1. An edge e is *covered* by $NLC(c)$ if there exists such a point p along edge e such that $p \in NLC(c)$. For example, $NLC(c_1)$ covers the edges (v_1, v_2), (v_1, v_4), (v_1, v_5), (v_3, v_4), (v_4, v_5) and (v_5, v_7). Besides, an edge is *completely covered* by $NLC(c)$ if all points along the edge are included in $NLC(c)$. For example, the edges (v_1, v_2), (v_1, v_4), (v_1, v_5), and (v_4, v_5) are completely covered by $NLC(c_1)$.

Given any point p on an edge in G, if $p \in NLC(c)$, then we say that the client c is *attracted* by a server to be built at the location p. For simplicity, we say that c is *attracted* by p.

For the sake of convenience, we summarize the notations used in the paper in Table 1.

3. RELATED WORK

We classify the related work into two types, namely optimal location queries with the non-road network setting (Section 3.1) and optimal location queries with the road network setting (Section 3.2).

3.1 Optimal Location Queries with Non-Road Network Setting

There are a lot of existing studies on optimal location queries with the non-road network setting [1, 2, 11, 14, 15, 16, 12, 20, 18] due to the importance of optimal location queries in real-life applications. In general, they find a location which optimizes an objective function in the L_p-norm space. One objective is to maximize the number of clients attracted. Another objective is to minimize the average distance between a client and its closest server.

Table 1: Notations

Notation	Description
G	a road network
V / v	the set of vertices/a vertex
E / e	the set of edges / an edge
C / c	the set of clients /a client
S / s	the set of servers / a server
$w(c)$	importance of client c
p	a location on the road network
$s(p)$	a server at location p
$c.dist$	the distance between c and its nearest server
$Cost(c)$	the cost value of c
$NN_{S'}(c)$	the server in S' nearest to client c
$[p_1, p_2]$	a point interval on a single edge where p_1 and p_2 are two points on this edge
$NLC(c)$	the nearest location component of c
$v.esd$	the edge server distance of v
VN	the virtual node
n	number of clients in C
$NewCost(c, p)$	the cost of client c after the new server is built at location p
$MaxNewCost(p)$	the greatest cost of a client after the new server is built at p
$cost_o$	the cost of the optimal solution for the MinMax query
p_o	the optimal location
$NLC(c, d)$	the shrinking NLC
m_o	the critical number (i.e., the greatest integer in $[1, n]$ such that the $(m_o, Cost(c_{m_o}))$-critical intersection is non-empty)
$(m, \mathcal{C})$-critical intersection	the intersection $\cap_{j=1}^{m} NLC(c_j, d_j)$ where $d_j = \mathcal{C}/w(c_j)$ for each $j \in [1, m]$
Θ	a set of point intervals representing the $(m_o, Cost(c_{m_o}))$-critical intersection
$\mathcal{I}$	a point interval in Θ
$\mathcal{I}'(= (p'_s, p'_e))$	a sub-interval of $\mathcal{I}$ whose interior contains no client
Ω	a set of piecewise linear functions each of which corresponds to a client
$\mathcal{R}$	the $(m_o, Cost(c_{m_o+1}))$-critical intersection
$\lvert C' \rvert$	the greatest number of NLCs covering a point interval in Θ
l	the number of clients in the point interval $\mathcal{I}$ (which is a portion of an edge)
l_s	the greatest number of servers along an edge
l_c	the greatest number of clients along an edge
α	the time complexity of constructing the intersection $\cap_{j=1}^{m_o} NLC(c_j, d_j)$ and checking the emptiness of the intersection
γ	the number of clients examined in MinMax-Alg
$Inf(p)$	the influence value of p

The optimal location query, originated from the *facility location problem*, also known as *location analysis* [1, 2, 11, 14], has been extensively studied in past years. The facility location problem is to locate the preferred facilities with respect to a given set of clients, and is shown to be NP-hard. A number of approximation algorithms were developed for the facility location problem. Different from the facility location problem where the number of all optimal locations is usually limited, in the optimal location query, the number of all optimal locations could be infinite. This is because usually, in the facility location problem, a set of a *limited* number of possible locations is given but in the optimal location query, this set can be the whole space (i.e., the set of all possible points on the road network). Recently, the researchers in the database community pay attention to this problem because of its broad applications.

The MaxBRNN problem [1] is to find an optimal *region* such that the total number of clients attracted by a new server to be set up is maximized. An infinite number of optimal points are contained in the optimal region. A solution with an exponential-time complexity was presented for the MaxBRNN problem in [1]. The MaxBRNN problem was also studied in [15] in which the first polynomial-time complexity algorithm, *MaxOverlap*, was introduced. Some variations, such as the extension of the *MaxOverlap* algorithm in a three-dimensional space and other L_p-norm metric spaces, were studied in [16]. Recently, the *MaxSegment* algorithm, an improved algorithm for the MaxBRNN problem, was given in [12]. Both the running time and the storage cost of the *MaxSegment* algorithm are significantly smaller than the *MaxOverlap* algorithm. A generalized MaxBRkNN problem [20] was studied in which a client may have different probabilities to visit different servers and at the same time, a server is assumed to have different target sets of clients. Moreover, an approximate method was recently presented for the MaxBRNN problem in [18].

Besides, the algorithm in [7] finds an optimal location instead of an optimal region for the L_1-norm space. The algorithm in [19] finds a location which minimizes the average distance from each client to its closest server when a new server is built at this location. The algorithm in [2] locates a place for a new server and this location can minimize the maximum distance between this new server and any client. The algorithm in [13] selects a location from a given set of potential locations for a new server so that the average distance between a client and its nearest server is minimized. The algorithm in [4] searches the location of a rectangular region with a given size such that the sum of the weights of all the points covered by this region is maximized.

3.2 Optimal Location Queries with Road Network Setting

Recently, Xiao el al. [17] first studied the OLQ problem with the road network setting and presented an algorithm which is the state-of-the-art algorithm. Specifically, the algorithm involves the following five major steps. The first step is to generate a vertex for each client and a vertex for each server, and include all generated vertices in the network/graph, resulting in a network with more vertices. The second step is to split each edge into a number of sub-edges via all original vertices and all newly generated vertices which are not located at the end-points of the edges in the original network, resulting in a network with more edges. As a result, a larger network is generated. Both the time and space complexities for generating the new road network are $O(M)$ where $M = \max\{|V|, |S|, |C|\}$. In particular, when each client and each server are not located at the end-points of edges in the original network, the resulting network generated by these algorithms contains $|V|+|S|+|C|$ vertices and $|E|+|S|+|C|$ edges. The third step is to partition the large network into a number of smaller sub-networks. The time complexity of this step is $O(M \log M)$. The fourth step is to execute a search algorithm based on each of the sub-networks in order to find the *local* optimal locations within each of these sub-networks. Note that the time complexity of the search algorithm on a sub-network is $O(|E'| \cdot |V''| \log |V''|)$ where $|E'|$ is the number of edges and $|V''|$ is the total number of vertices visited by the search algorithm. Since the search algorithm on a sub-network sometimes requires to search some vertices in other sub-networks "close" to this sub-network, $|V''|$ can be larger than the number of vertices in this sub-network. Note that $|V''|$ is at most the number of vertices in the resulting network (i.e., $|V| + |S| + |C|$). In our experiments with the default setting on the *SF* real dataset where $|C|$ is 300k, $|S|$ is 1k and $|V|$ is 174k, $|V''|$ is equal to 475k (which is exactly equal to $|V| + |S| + |C|$, the worst-case scenario). It is easy to verify that the overall time complexity of this step is $O((|E|+|S|+|C|) \cdot |V''| \log |V''|)$ after we execute this search algorithm on all sub-networks (since there are $(|E|+|S|+|C|)$ edges in the resulting network). Note that $|V''| = O(|V| + |S| + |C|)$ and $|E| = O(|V|)$ in the road network setting (e.g., a vertex is adjacent to at most 8 edges (3 edges on average) in the real road network *SF* used in our experiments). The time complexity of this step is $O((|V| + |S| + |C|)^2 \log(|V| + |S| + |C|))$. The fifth step is to scan through all local optimal locations obtained in each sub-network (from the previous step) and find the *global* optimal locations in the whole network. In conclusion, it is easy to ver-

ify that the overall time complexity of the best-known algorithm is $O((|V|+|S|+|C|)^2 \log(|V|+|S|+|C|))$ in the worst case, which is prohibitively expensive.

Besides, as described in Section 1, this algorithm has some shortcomings. In this paper, we propose a new algorithm framework with significant improvements for the OLQ problem. In particular, the time complexity of our algorithm is $O(\gamma \cdot |V| \log |V| + |V| \cdot |C| \log |C|)$ where γ is at most $|C|$.

4. BUILDING NLCS

This section introduces a method of building $NLC(c)$ for a client c which involves three steps: (i) determine the shortest distance from each *vertex* v to its nearest server, denoted by $v.dist$ (Section 4.1), (ii) determine the shortest distance from each *client* c to its nearest server, denoted by $c.dist$ (Section 4.2), and (iii) build $NLC(c)$ based on $c.dist$ (Section 4.3).

4.1 Finding Shortest Distance Between a Vertex and Its Closest Server

In this section, we describe how we determine the shortest distance from each vertex v to its nearest server in the network. A naive method is to perform a search (e.g., the Dijikstra's algorithm) from *each* vertex v and find the closest server to v in the network. This method is time-consuming since it has to execute the search process $|V|$ times independently. Thus, the total time complexity of this method is $O(|V|^2 \log |V|)$ since the Dijikstra's algorithm takes $O(|V| \log |V|)$ time [6]. In the following, we introduce an efficient method which runs the search process (i.e., the Dijkstra's algorithm) once only, and thus it has the overall time complexity of $O(|V| \log |V|)$.

We construct a new node called the *virtual node*, denoted by VN, in the network G, and then execute the Dijkstra's algorithm starting from this virtual node VN once only. We guarantee that the distance between each vertex v and its closest server (i.e., $v.dist$) is exactly equal to the shortest distance between VN and v.

Before we describe how we construct this virtual node, we introduce a concept called "edge server distance". For each vertex v, the *edge server distance* of v, denoted by $v.esd$, is defined to be the distance from v to the server closest to v along one of the edges adjacent to v if there is a server on one of these edges, and ∞ otherwise.

Consider v_7 in our example as shown in Figure 1(a) for illustration. There are 3 edges adjacent to v_7 (i.e., (v_7, v_5), (v_7, v_3) and (v_7, v_6)). Only edges (v_7, v_5) and (v_7, v_3) contain servers, namely s_1 and s_3, respectively. Thus, the edge server distance of v_7 (i.e., $v_7.esd$) is equal to $\min\{d(v_7, s_1), d(v_7, s_3)\} = \min\{4, 7\} = 4$. Similarly, we can compute the edge server distances of the other vertices. We have $v_2.esd = 3$, $v_3.esd = 0$, $v_4.esd = 3$, $v_5.esd = 2$ and $v_6.esd = 7$. Since there is no server on the edges adjacent to v_1, $v_1.esd$ is equal to ∞.

Now, we are ready to describe how to construct the virtual node VN. We create a virtual node VN in the network. For each vertex v where $v.esd \neq \infty$, we create an edge (VN, v) and set the length of this edge to be $v.esd$. Next, we take the road network in Figure 1(a) as an example to illustrate the detailed construction.

Consider our running example again. Since only v_2, v_3, v_4, v_5, v_6 and v_7 have their edge server distances not equal to ∞, we create a virtual node VN in the road network as shown in Figure 2. Specifically, the virtual node VN is connected with the vertices v_2, v_3, v_4, v_5, v_6 and v_7 only. Their corresponding edge lengths are $3, 0, 3, 2, 7$ and 4, respectively.

It is easy to verify that the time of computing $v.esd$ for all vertices is $O(|V|)$ (remember that $|E| = O(|V|)$ in a road network $G = (V, E)$).

Next, we execute the Dijkstra's algorithm, which takes VN as a source node, to traverse all vertices in G. After each vertex is traversed, the shortest distance from VN to each vertex v, $d(VN, v)$, can be obtained.

The idea of using the concept of "virtual node" was studied in [8] and this concept has the following property.

PROPERTY 1. *Consider the network G with the virtual node VN. Then, for each vertex v in G (except the virtual node), $v.dist = d(VN, v)$.*

By Property 1, we can know that $v.dist = d(VN, v)$. Since the Dijkstra's algorithm takes $O(|V| \log |V|)$ time [6], finding the shortest distances between all vertices and their closest servers (i.e., $v.dist$ for all $v \in V$) takes $O(|V| \log |V|)$ time.

4.2 Finding Shortest Distance Between a Client and Its Closest Server

Now, we know how to compute $v.dist$ efficiently for each *vertex* v. In this section, we compute $c.dist$ efficiently for each *client* c. Similar to Section 4.1, a naive method of computing $c.dist$ for all clients c takes $O(|C| \cdot |V| \log |V|)$ time. In this section, we present an efficient method for this, which takes $O(|C| \cdot l_s)$ time where l_s is a small number at most $|S|$, based on the distance information computed in the previous section with the help of the following lemma.

LEMMA 1. *Consider a client c on an edge $e = (v_l, v_r)$. If there is no server on e, then $c.dist = \min\{d(c, v_l) + v_l.dist, d(c, v_r) + v_r.dist\}$. Otherwise, $c.dist = \min\{d(c, s'), d(c, v_l) + v_l.dist, d(c, v_r) + v_r.dist\}$ where s' is the closest server to c along e.*

It is easy to verify the correctness of the above lemma. Let l_s be the greatest number of servers along an edge. The running time of finding $c.dist$ for one client c takes $O(l_s)$ time, and the overall running time of finding $c.dist$ for all clients c takes $O(|C| \cdot l_s)$ time.

4.3 Building the NLC of a Client

In this section, we discuss how to find the NLC of client c based on $c.dist$ computed in the previous section.

Algorithm 1 shows the algorithm for finding the NLC of a client c (i.e., $NLC(c)$). In this algorithm, we use the Dijkstra's algorithm to traverse the vertices in the network in ascending order of their shortest distances to c (Line 1). Besides, we introduce a variable $\mathcal{I}$ which is used to store $NLC(c)$ and is being updated during the execution of the algorithm. In the algorithm, $\mathcal{I}$ is to store a set of point intervals representing $NLC(c)$.

Initially, $\mathcal{I}$ is set to $\emptyset$ (Line 2). Then, the algorithm processes each vertex v (Line 3) with two different cases (based on the processing ordering of the Dijkstra's algorithm). *Case 1:* $d(v, c) \leq c.dist$ (Line 4). In this case, for each edge e' adjacent to v, in the form of (v, v') (Line 5), we check whether v' is processed before (Line 6). If yes, then we know that the whole edge e' is inside $NLC(c)$. Thus, the point interval with the end points of e' (i.e., v and v'), which is equal to $[v, v']$, is created and is inserted into $\mathcal{I}$ (Line 7).

Case 2: $d(v, c) > c.dist$ (Line 9). In this case, for each edge e'' adjacent to v, in the form of (v, v'') (Line 10), we check whether $d(v'', c) \leq c.dist$ (Line 11).

- If yes, then we know that a portion of the edge e'' containing v'' is inside $NLC(c)$. Next, we check whether c is on edge

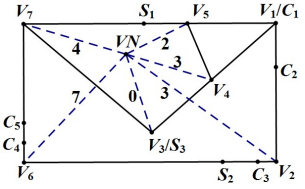
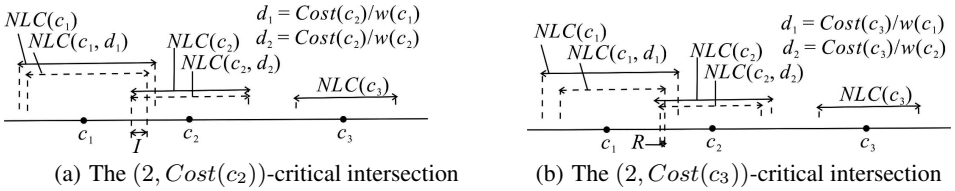


Figure 2: The road network with a virtual node VN



(a) The $(2, Cost(c_2))$-critical intersection (b) The $(2, Cost(c_3))$-critical intersection

Figure 3: Example 1

Algorithm 1 Algorithm for Finding the NLC of a Client c (i.e., $NLC(c)$)

```
1: use Dijkstra's algorithm to traverse the vertices in ascending order of their distances to c
2: I ← ∅
3: for each vertex v to be processed do
4:   if d(v, c) ≤ c.dist then
5:     for each edge e' adjacent to v, in the form of (v, v'), do
6:       if v' is processed before then
7:         I ← I ∪ {[v, v']};
8:   else
9:     // d(v, c) > c.dist
10:    for each edge e'' adjacent to v, in the form of (v, v''), do
11:      if d(v'', c) ≤ c.dist then
12:        if c is on edge e'' then
13:          I ← I ∪ {[v'', p]} where p is a point along the edge e'' such that
             d(c, p) ≤ c.dist and d(c, p) is the largest.
14:        else
15:          I ← I ∪ {[v'', p']} where p' is a point along the edge e'' such
             that d(c, v'') + d(v'', p') ≤ c.dist and d(v'', p') is the largest.
16:      else
17:        if c is on edge e'' then
18:          I ← I ∪ {[q, q']} where q and q' are two points along the edge
             e'' such that d(c, q) = c.dist, d(c, q') = c.dist and q ≠ q' if
             c.dist ≠ 0 and q = q' otherwise.
19:        else
20:          regard e'' as deleted
21: return I
```

e'' (Line 12). If c is on edge e'' (Line 12), then we know that there exists a point p on the edge e'' such that $d(c,p) \leq c.dist$ and $d(c,p)$ is the largest. We create a point interval $\{[v'', p]\}$ and insert it into $\mathcal{I}$ (since each point in this point interval is in $NLC(c)$) (Line 13). If c is not on edge e'', we know that there exists a point p' on the edge e'' such that $d(c,v'') + d(v'',p') \leq c.dist$ and $d(v'',p')$ is the largest. We create a point entry $\{[v'', p']\}$ and insert it into $\mathcal{I}$ (since each point in this point interval is in $NLC(c)$) (Line 15).

- If no, then we check whether c is on the edge e'' (Line 17). If c is on the edge e'', then we know that a portion of the edge e'' is inside $NLC(c)$. We also know that there exist two points along the edge e'', namely q and q', such that $d(c,q) = c.dist$, $d(c,q') = c.dist$ and $q \neq q'$ if $c.dist \neq 0$ and $q = q'$ otherwise. We create a point entry $\{[q, q']\}$ and insert it into $\mathcal{I}$ (since each point in this point interval is in $NLC(c)$) (Line 18). If c is not on the edge e'', we know that no point along the edge e'' is inside $NLC(c)$. we can regard e'' as deleted (Line 20).

Finally, we return $\mathcal{I}$ as an output, representing $NLC(c)$ (Line 21). Note that if the intervals in $\mathcal{I}$ are overlapping, then they are merged into one interval that covers these intervals accordingly.

Consider our running example. All the point intervals included in $NLC(c_1)$ are marked in bold lines in Figure 1(b).

The major time cost of Algorithm 1 comes from Line 1 (i.e., Dijkstra's algorithm). Since Dijkstra's algorithm takes $O(|V| \log |V|)$ time, Algorithm 1 takes $O(|V| \log |V|)$ time and $O(|V|)$ space.

5. ALGORITHM MINMAX-ALG

In this section, we propose an algorithm called *MinMax-Alg* for the MinMax query. The algorithm is developed based on a concept called "critical number". With the concept of "critical number", we can determine the optimal location for the MinMax query efficiently. Let p_o be the optimal solution (position) for the MinMax query. Let $cost_o$ be the cost of the optimal solution for the MinMax query. That is, $cost_o = \min_{p \in G}[\max_{c \in C} w(c) \cdot d(c, NN_{S \cup \{s(p)\}}(c))]$. Here, "$p \in G$" means that p is an arbitrary point along an edge in G.

Before we give the definition of "critical number", we give an assumption and some concepts first. We assume that different clients have different costs. This assumption allows us to avoid several complicated, yet uninteresting, boundary cases. Obviously, when the assumption is not fulfilled, we can always apply an infinitesimal perturbation to the positions of some clients or servers, to break the tie of the costs of two clients. Due to the tininess of perturbation, query results from the perturbed datasets should be as useful as those from the original datasets.

Suppose that there are n clients, namely $c_1, c_2, ..., c_n$, and all clients are ordered in descending order of their costs. Without loss of generality, assume that $Cost(c_1) > Cost(c_2) > ... > Cost(c_n)$.

Next, we introduce two concepts, namely "shrinking NLC" and "critical intersection".

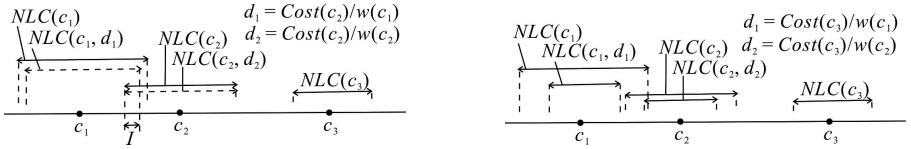
DEFINITION 2 (SHRINKING NLC). *Given a client c and a value d where $0 \leq d \leq c.dist$, the* shrinking NLC *of c with respect to d, denoted by $NLC(c,d)$, is defined to be $\{p | d(c,p) \leq d$ and $p \in NLC(c)\}$.*

Consider two examples. The first example is shown in Figure 3. In this example, there are 3 clients, namely c_1, c_2 and c_3, on a single edge. Here, $Cost(c_1) > Cost(c_2) > Cost(c_3)$. Assume that $w(c_i) = 1$ for each $i \in [1,3]$. For the ease of illustration, we do not show the servers in the figure. In both sub-figures of Figure 3, we show $NLC(c_1), NLC(c_2)$ and $NLC(c_3)$. In Figure 3(a), if $d_1 = Cost(c_2)/w(c_1)$ and $d_2 = Cost(c_2)/w(c_2)$, then $NLC(c_1, d_1)$ is a shrinking NLC of c_1 with respect to d_1 and $NLC(c_2, d_2)$ is a shrinking NLC of c_2 with respect to d_2. Similarly, in Figure 3(b), if $d_1 = Cost(c_3)/w(c_1)$ and $d_2 = Cost(c_3)/w(c_2)$, then $NLC(c_1, d_1)$ is a shrinking NLC of c_1 with respect to d_1 and $NLC(c_2, d_2)$ is a shrinking NLC of c_2 with respect to d_2.

The second example is shown in Figure 4. This example is exactly the same as the first example except that $c_3.dist$ is smaller and thus both $Cost(c_3)$ and $NLC(c_3)$ are smaller. Similarly, we obtain the shrinking NLCs as shown in Figure 4(a) and Figure 4(b).

DEFINITION 3 (CRITICAL INTERSECTION). *Given an integer $m \in [1,n]$ and a non-negative real number $\mathcal{C}$, the* $(m, \mathcal{C})$-critical intersection *is defined to be $\cap_{j=1}^{m} NLC(c_j, d_j)$ where $d_j = \mathcal{C}/w(c_j)$ for each $j \in [1,m]$.*

Consider the first example. In Figure 3(a), $\mathcal{I}$ is the $(2, Cost(c_2))$-critical intersection. In Figure 3(b), $\mathcal{R}$ is the



(a) The $(2, Cost(c_2))$-critical intersection (b) The $(2, Cost(c_3))$-critical intersection

Figure 4: Example 2

$(2, Cost(c_3))$-critical intersection. Besides, it is easy to verify that the $(3, Cost(c_3))$-critical intersection is empty (since $NLC(c_3)$ does not overlap with $NLC(c_1, d_1)$ and $NLC(c_2, d_2)$). Consider the second example. in Figure 4(a), $\mathcal{I}$ is the $(2, Cost(c_2))$-critical intersection. In Figure 4(b), the $(2, Cost(c_3))$-critical intersection is empty. Besides, it is easy to verify that the $(3, Cost(c_3))$-critical intersection is empty.

Now, we are ready to define the critical number as follows.

DEFINITION 4 (CRITICAL NUMBER). *The* critical number *denoted by m_o is defined to be the greatest integer $\in [1, n]$ such that the $(m_o, Cost(c_{m_o}))$-critical intersection is non-empty.*

Consider the first example. We know that both the $(1, Cost(c_1))$-critical intersection and the $(2, Cost(c_2))$-critical intersection are non-empty, but the $(3, Cost(c_3))$-critical intersection is empty. Thus, the critical number m_o is equal to 2 in the first example. Consider the second example. Similarly, we deduce that the critical number m_o is equal to 2.

Now, we are ready to give the following lemma which gives us hints of how to find the optimal location p_o with the help of the critical number m_o.

LEMMA 2. *There exists an optimal location p_o in the $(m_o, Cost(c_{m_o}))$-critical intersection.*

PROOF. Given a client c and a location p in the $(m_o, Cost(c_{m_o}))$-critical intersection, we denote $NewCost(c, p)$ to be the cost of client c after a new server is set up at p. Given a location p in the $(m_o, Cost(c_{m_o}))$-critical intersection, we denote $MaxNewCost(p) = \max_{c \in C} NewCost(c, p)$. Since if we set up a new server at any location in $\cap_{j=1}^{m_o} NLC(c_j, Cost(c_{m_o})/w_j)$ (which is non-empty), the greatest cost of a client is at most $Cost(c_{m_o})$ (i.e., $MaxNewCost(p) \leq Cost(c_{m_o})$). Thus, for any location p in the $(m_o, Cost(c_{m_o}))$-critical intersection, since $cost_o \leq MaxNewCost(p)$, we conclude that $cost_o \leq MaxNewCost(p) \leq Cost(c_{m_o})$.

Consider two cases. *Case 1:* $cost_o = Cost(c_{m_o})$. For any location p in the $(m_o, Cost(c_{m_o}))$-critical intersection, since $cost_o \leq MaxNewCost(p) \leq Cost(c_{m_o})$, we deduce that $MaxNewCost(p) = Cost(c_{m_o})$. Thus, there exists an optimal location in the $(m_o, Cost(c_{m_o}))$-critical intersection.

Case 2: $cost_o < Cost(c_{m_o})$. Since $cost_o = MaxNewCost(p_o)$, we have $MaxNewCost(p_o) < Cost(c_{m_o})$. Next, we show that p_o is in the $(m_o, Cost(c_{m_o}))$-critical intersection. We prove by contradiction. Suppose that p_o is not in the $(m_o, Cost(c_{m_o}))$-critical intersection. There exists an integer $j_o \in [1, m_o]$ such that p_o is outside $NLC(c_{j_o}, Cost(c_{m_o})/w(c_{j_o}))$. Thus, $d(p_o, c_{j_o}) > Cost(c_{m_o})/w(c_{j_o})$. That is, $d(p_o, c_{j_o}) \cdot w(c_{j_o}) > Cost(c_{m_o})$. Then, we have $cost_o = MaxNewCost(p_o) \geq NewCost(c_{j_o}, p_o) = \min\{Cost(c_{j_o}), d(p_o, c_{j_o}) \cdot w(c_{j_o})\}$. Since $Cost(c_{j_o}) \geq Cost(c_{m_o})$ and $d(p_o, c_{j_o}) \cdot w(c_{j_o}) > Cost(c_{m_o})$, we deduce that $cost_o \geq Cost(c_{m_o})$. That leads to a contradiction that $cost_o < Cost(c_{m_o})$. This lemma also holds. □

The above lemma is a powerful tool which helps us design a two-step algorithm called *MinMax-Alg* for the MinMax query as follows.

- **Step 1 (Finding Critical Number m_o):** The first step is to find the critical number m_o.
- **Step 2 (Finding Optimal Solution):** The second step is to find the optimal solution in the $(m_o, Cost(c_{m_o}))$-critical intersection.

Step 1 and Step 2 will be described in detail in Section 5.1 and Section 5.2, respectively.

5.1 Step 1: How to Find Critical Number m_o

Finding m_o involves the following steps. First, we initialize a variable m to 2. Then, we can check whether the $(m, Cost(c_m))$-critical intersection is non-empty. If yes, we increment m by 1 and continue the above process. If no, we can terminate the process and know that m_o is equal to $m - 1$.

5.2 Step 2: How to Find Optimal Location in Critical Intersection

In Section 5.2.1, we introduce a basic method to find the optimal location in the critical intersection, and in Section 5.2.2, we introduce an enhancement on the basic method.

5.2.1 Basic Algorithm

We want to find the optimal location in the $(m_o, Cost(c_{m_o}))$-critical intersection (which is an intersection among multiple shrinking NLCs of clients, namely $c_1, c_2, ..., c_{m_o}$). Note that this intersection can be represented by a set Θ of point intervals. Next, we describe how to find the particular location in a *single* point interval in Θ with the smallest cost.

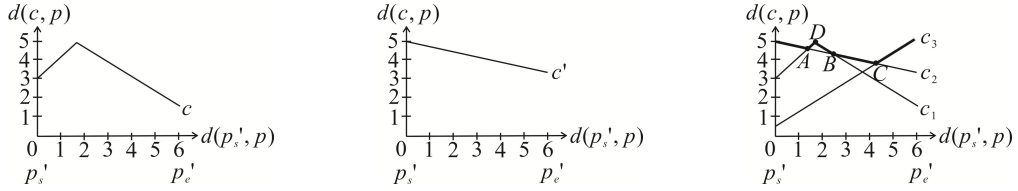
Consider a point interval $\mathcal{I} = [p_s, p_e]$. Note that $\mathcal{I}$ is completely covered by the shrinking NLCs of clients, namely $c_1, c_2, ..., c_{m_o}$. Consider the shrinking NLC of a *particular* client c.

Note that point interval $\mathcal{I}$ is a portion (or the whole portion) of a single edge and thus is on a single edge. This edge may contain l clients. Suppose that there are multiple clients, namely $c'_1, c'_2, ..., c'_l$, in the point interval $\mathcal{I}$, where c'_i is the i-th closest client to p_s for each $i \in [1, l]$. Thus, we split the whole interval into a number of sub-intervals, $[p_s, c'_1], [c'_1, c'_2], ..., [c'_l, p_e]$. Note that each of the sub-intervals (of the point interval $\mathcal{I}$ on this single edge) is also completely covered by the shrinking NLC of c. Besides, there is no client on the interior of each sub-interval (on this single edge).

Consider a sub-interval $\mathcal{I}' = [p'_s, p'_e]$ of $\mathcal{I}$. Let p be an *arbitrary* point along $\mathcal{I}'$. We know that $d(c, p)$ can be expressed as follows.

$$d(c, p) = min\{d(c, p'_s) + d(p'_s, p), d(c, p'_e) + d(p'_e, p)\}$$

Note that when p changes along $\mathcal{I}'$, $d(c, p)$ changes linearly. We can regard that the form of the above equation is a *piecewise linear function*, containing two components (one is the linear equation "$d(c, p'_s) + d(p'_s, p)$" and the other is the linear equation "$d(c, p'_e) + d(p'_e, p)$" where p'_s, p'_e and c are fixed). For example, Figure 5(a) shows that when p changes (along $\mathcal{I}' = [p'_s, p'_e]$ where $d(p'_s, p'_e) = 6$), $d(c, p)$ varies. In the figure, the x-axis denotes $d(p'_s, p)$ where p'_s is fixed and p is varying, and the y-axis denotes $d(c, p)$ where c is fixed. There are two line segments in the figure. Similarly,



(a) One client c (b) One client c' (c) Multiple clients

Figure 5: Piecewise Linear Function denoting $d(c, p)$ where c is a particular client and p is an arbitrary point

Figure 5(b) shows the piecewise linear function for another client c' (which contains one line segment in the figure only).

We conclude that for a *particular* client c where the shrinking NLC of c completely covers the sub-interval $\mathcal{I}'$, we have a piecewise linear function on $\mathcal{I}'$. For another client c' where the shrinking NLC of c' completely covers $\mathcal{I}'$, we have another piecewise linear function on $\mathcal{I}'$. There may exist multiple clients whose shrinking NLCs completely cover $\mathcal{I}'$. For instance, Figure 5(c) shows the piecewise linear functions on $\mathcal{I}'$ for 3 clients, c_1, c_2 and c_3, whose shrinking NLCs completely cover $\mathcal{I}'$. In this figure, the piecewise functions for c_1, c_2 and c_3 involve 2 line segments, 1 line segment and 1 line segment, respectively.

Now, we are ready to describe how we find the particular location on $\mathcal{I}'$ with the smallest cost. This involves two sub-steps.

- The first sub-step is to find the *upper envelope* of all piecewise linear functions involved. Given a set Ω of piecewise linear functions, the *upper envelope* of Ω is defined to be the function which takes x as input and outputs the maximum value of all piecewise linear functions taking this value x as input. For example, in Figure 5(c) with 3 piecewise linear functions, the bolded line corresponds to the upper envelope of the set of these 3 functions. Finding the upper envelope can be done in $O(m \log m)$ time [9] where m is the total number of piecewise linear functions involved.
- The second sub-step is to find the minimum value in this upper envelope, which denotes the particular location on $\mathcal{I}'$ with the smallest cost. In Figure 5(c), point C corresponds to the minimum value in this upper envelope. This sub-step can be done together with the first sub-step when we construct the upper envelope.

We have just described how to find the optimal location for a *single* sub-interval $\mathcal{I}'$ of a given point interval. Given a point interval $\mathcal{I}$, we can obtain the optimal location with its cost for each of the sub-intervals of $\mathcal{I}$ and find the one with the smallest cost as the optimal location for the point interval $\mathcal{I}$. Since we have a number of point intervals in Θ, we can obtain the optimal location with its cost for each point interval in Θ and find the one with the smallest cost as the optimal location in the critical intersection.

Time Complexity Analysis: Next, we analyze the time complexity of finding the optimal location in the critical intersection. Consider a sub-interval $\mathcal{I}'$ of a point interval $\mathcal{I}$ in Θ. Let $|C'|$ be the greatest number of (shrinking) NLCs covering a point interval $\mathcal{I}$. Note that $|C'|$ is at most $|C|$ and in practice, it is significantly smaller than $|C|$. Besides, the sub-interval $\mathcal{I}'$ is associated with at most $|C'|$ piecewise linear functions. The time complexity for both the first sub-step and the second sub-step for a single sub-interval $\mathcal{I}'$ is $O(|C'| \log |C'|)$. Let l_c be the greatest number of clients along an edge. Similarly, l_c is at most $|C|$ and is usually much smaller than $|C|$. Since there are $O(l_c)$ sub-intervals of a single point interval, the time complexity of processing a single point interval is $O(l_c \cdot |C'| \log |C'|)$. Since there are $|\Theta|$ point intervals, the time complexity of the method of finding the optimal location in the critical intersection is $O(|\Theta| \cdot l_c \cdot |C'| \log |C'|)$. Note that $|\Theta|$ is at most $|E|$ and is usually much smaller than $|E|$.

5.2.2 Further Enhancement

In the previous section, we described a method finding the optimal location in the critical intersection, which takes $O(|\Theta| \cdot l_c \cdot |C'| \log |C'|)$ time. Although it is an efficient method using the concept of "piecewise linear functions", in this section, we want to introduce an additional step to find a special region $\mathcal{R}$ and check whether $\mathcal{R}$ is empty, which can be done in $O(\alpha)$ time where α is a positive integer extremely smaller than $|E|$. In practice, in our *SF* real dataset, α is at most 390 but $|E|$ is 223k. If $\mathcal{R}$ is non-empty, then we return any location in $\mathcal{R}$ (or simply the whole region $\mathcal{R}$) as the optimal location, which can be done in $O(\alpha)$ time. Otherwise, we keep executing the method introduced in the previous method to find the optimal location in the critical intersection.

We first give some concepts and a lemma, and then introduce the additional step.

DEFINITION 5. *Suppose that $m_o < n$. We define $\mathcal{R}$ to be the $(m_o, Cost(c_{m_o+1}))$-critical intersection.*

Consider the first example where $m_o = 2$. Figure 3(b) shows an example of $\mathcal{R}$ which is non-empty. Consider the second example where $m_o = 2$. Figure 4(b) shows another example of $\mathcal{R}$ which is empty.

Next, we give a lemma based on $\mathcal{R}$.

LEMMA 3. *Suppose that $m_o < n$. If $\mathcal{R} \neq \emptyset$, then (1) $cost_o = Cost(c_{m_o+1})$ and (2) when a new server is set up at any location in $\mathcal{R}$, the maximum cost of a client is equal to $cost_o$. If $\mathcal{R} = \emptyset$, then $Cost(c_{m_o+1}) < cost_o \leq Cost(c_{m_o})$.*

PROOF. We prove the lemma with three parts.

First, we prove that in both cases of $\mathcal{R}$, we have $Cost(c_{m_o+1}) \leq cost_o \leq Cost(c_{m_o})$. Consider $cost_o \geq Cost(c_{m_o+1})$. We prove by contradiction. Suppose that $cost_o < Cost(c_{m_o+1})$ which further implies that $cost_o < Cost(c_j)$ for $j = m_o + 1, m_o, ..., 1$. It could be verified that $p_o \in NLC(c_j, cost_o/w_j)$ for $j = m_o + 1, m_o, ..., 1$ since the cost of c_j would be updated from $Cost(c_j)$ to $cost_o$ by the definition of p_o and $cost_o$. Besides, we know that $NLC(c_j, cost_o/w_j)$ is covered by $NLC(c_j, Cost(c_{m_o+1})/w_j)$ for $j = m_o + 1, m_o, ..., 1$ since $cost_o < Cost(c_{m_o+1})$. Therefore, we know that $p_o \in NLC(c_j, Cost(c_{m_o+1})/w_j)$ for $j = m_o + 1, m_o, ..., 1$ which leads to a contradiction that m_o is the critical number.

Consider $cost_o \leq Cost(c_{m_o})$. This is obvious since if we set up a new server at any location in $\cap_{j=1}^{m_o} NLC(c_j, Cost(c_{m_o})/w_j)$ (which is not empty), the maximum cost of a client is at most $Cost(c_{m_o})$ (the cost of c_j for $j = 1, 2, ..., m_o$ would be updated from $Cost(c_j)$ to a value at most $Cost(c_{m_o})$ and the cost of c_j for $j = m_o + 1, m_o + 2, ..., n$ which is originally smaller than $Cost(c_{m_o})$ would not be increased).

Second, we show that in the case of $\mathcal{R} \neq \emptyset$, we have $cost_o \leq Cost(c_{m_o+1})$ which further implies that $cost_o = Cost(c_{m_o+1})$ (by using the results in the first part of the proof). This is obvious since if we set up a new server at any location in $\mathcal{R}$, the maximum cost of a client is at most $Cost(c_{m_o+1})$ (the cost of c_j for $j =$

$1, 2, ..., m_o$ would be a value at most $Cost(c_{m_o+1})$ and the cost of c_j for $j = m_o + 1, m_o + 2, ..., n$ which is originally at most $Cost(c_{m_o+1})$ would not be increased).

Third, we show that in the case of $\mathcal{R} = \emptyset$, we have $cost_o > Cost(c_{m_o+1})$ by contradiction. Suppose $cost_o \leq Cost(c_{m_o+1})$. We have two cases. Case 1: $cost_o < Cost(c_{m_o+1})$. This leads to a contradiction according to the results in the first part of this proof. Case 2: $cost_o = Cost(c_{m_o+1})$. In this case, by using the fact that $p_o \in NLC(c_j, cost_o/w_j)$ for $j = 1, 2, ..., m_o$, we know that $p_o \in \cap_{j=1}^{m_o} NLC(c_j, Cost(c_{m_o+1})/w_j)$ which leads to a contradiction since $\mathcal{R} = \emptyset$. □

Consider the first example where $m_o = 2$. Since $\mathcal{R}$ is non-empty, by the above lemma, we know that the optimal cost $cost_o$ is equal to $Cost(c_3)$ and when a new server is set up at any location in $\mathcal{R}$, the maximum cost of a client is equal to $cost_o$. Consider the second example where $m_o = 2$. Since $\mathcal{R}$ is empty, by the above lemma, we know that $Cost(c_3) < cost_o \leq Cost(c_2)$.

The above lemma suggests that if we know that $\mathcal{R}$ is non-empty, then we immediately return the whole region $\mathcal{R}$ as the final answer; otherwise, we proceed to find a particular point in the $(m_o, Cost(c_{m_o}))$-critical intersection so that the cost of the final solution with the new server set up at this location is equal to $cost_o$.

Now, we are ready to give the description of Step 2 based on the above lemma.

- **Step 2(a):** If $m_o < n$, then we do the following. We check whether region $\mathcal{R}$ is empty or not. If not, we immediately return region $\mathcal{R}$. If yes, we find the optimal location in the $(m_o, Cost(c_{m_o}))$-critical intersection.
- **Step 2(b):** If $m_o = n$, then we find the optimal location in the $(m_o, Cost(c_{m_o}))$-critical intersection.

Time Complexity Analysis: Consider Step 2(a). Let α be the time complexity of constructing the intersection and checking the emptiness of the intersection. In general, α can be measured by the number of disconnected components for an intersection between two NLCs. In our experiments, this number is at most $390 \ll |E|$ under the default setting on the *SF* real dataset where $|V|$ is 174k, $|E|$ is 223k, $|C|$ is 300k and $|S|$ is 1k. After the emptiness of the intersection is checked, we perform different steps with two different sub-cases. The first sub-case is that $\mathcal{R}$ is non-empty. This sub-case can be handled easily since we just need to return $\mathcal{R}$. The second sub-case is that $\mathcal{R}$ is empty. The steps in this sub-case take $O(|\Theta| \cdot l_c \cdot |C'| \log |C'|)$ time. In conclusion, the time complexity of Step 2(a) is $O(\alpha + |\Theta| \cdot l_c \cdot |C'| \log |C'|)$. Similarly, the time complexity of Step 2(b) is $O(|\Theta| \cdot l_c \cdot |C'| \log |C'|)$. Thus, the overall time complexity of the two-step algorithm is $O(\alpha + |\Theta| \cdot l_c \cdot |C'| \log |C'|)$.

5.3 Pseudo-Code

We present the pseudo-code of MinMax-Alg in Algorithm 2.

EXAMPLE 1. Let us take the road network in Figure 1 for illustration. Assume that the client weights are the following: $w(c_1)$=2, $w(c_2)$=3, $w(c_3)$=1.5, $w(c_4)$=1 and $w(c_5)$=0.5. Note that c_1, c_2, c_3, c_4 and c_5 have their nearest servers as s_1, s_2, s_2, s_2 and s_1, respectively. We know that $c_1.dist = 6, c_2.dist = 7, c_3.dist = 2, c_4.dist = 8$ and $c_5.dist = 8$ (Line 1). Thus, $Cost(c_1) = w(c_1) \cdot c_1.dist = 2 \cdot 6 = 12$. Similarly, we have $Cost(c_2) = 21, Cost(c_3) = 3, Cost(c_4) = 8$ and $Cost(c_5) = 4$. Then, we have $Cost(c_2) > Cost(c_1) > Cost(c_4) > Cost(c_5) > Cost(c_3)$. The client ordering is c_2, c_1, c_4, c_5 and c_3 (Line 2).

In Step 1 (Lines 3-13), variable m is updated incrementally. When $m = 1$, we know that the first client in the ordering is c_2 and thus $\mathcal{C} = Cost(c_2) = 21$ (Line 5) and thus $\mathcal{I}_{new} = NLC(c_2, 21/3)$ (Line 8). Since $\mathcal{I}_{new}$ is non-empty, variable $\mathcal{I}$ is updated to $\mathcal{I}_{new}$ (Line 12) and variable m_o is updated to $m(= 1)$ (Line 13).

When $m = 2$, we know that the second client is c_1 and thus $\mathcal{C} = Cost(c_1) = 12$ (Line 5). Similarly, we have $\mathcal{I}_{new} = NLC(c_2, 12/3) \cap NLC(c_1, 12/2)$ (Line 8). Since $\mathcal{I}_{new}$ is non-empty, variable $\mathcal{I}$ is updated to $\mathcal{I}_{new}$ (Line 12) and m_o is updated to $m(= 2)$ (Line 13).

When $m = 3$, we know that the third client is c_4 and thus $\mathcal{C} = Cost(c_4) = 8$ (Line 5). Similarly, we have $\mathcal{I}_{new} = NLC(c_2, 8/3) \cap NLC(c_1, 8/2) \cap NLC(c_4, 8/1)$ (Line 8). Since $\mathcal{I}_{new} = \emptyset$, we break the iterative step in Step 1 (Line 10) and do not update variable $\mathcal{I}$ and m_o. Thus, finally, we have $\mathcal{I} = NLC(c_2, 4) \cap NLC(c_1, 6)$ and $m_o = 2$.

Consider Step 2. We know that $m_o + 1 = 3$. Since $m_o < n$ (Line 15), and the third client in the ordering is c_4, variable $\mathcal{C}'$ is set to $Cost(c_4)(= 8)$. Since the first client and the second client in the ordering are c_2 and c_1, respectively, variable $\mathcal{R}$ is updated to $NLC(c_2, 8/3) \cap NLC(c_1, 8/2)$. Since $\mathcal{R} \neq \emptyset$, $\mathcal{R}$ is returned as the solution by the algorithm. □

Algorithm 2 Algorithm MinMax-Alg

```
1: find c.dist for each client c
2: sort all clients c_1, c_2, ..., c_n in descending order of their cost values
3: // Step 1 (Finding m_o)
4: for m = 1 to n do
5:    C ← Cost(c_m)
6:    for i = 1 to m do
7:       d_i ← C/w(c_i)
8:    I_new ← ∩_{i=1}^{m} NLC(c_i, d_i)
9:    if I_new = ∅ then
10:      break;
11:   else
12:      I ← I_new
13:      m_o ← m
14: // Step 2 (Checking R)
15: if m_o < n then
16:    C' ← Cost(c_{m_o+1})
17:    for i = 1 to m_o + 1 do
18:       d'_i ← C'/w(c_i)
19:    R ← ∩_{i=1}^{m_o} NLC(c_i, d'_i)
20:    if R ≠ ∅ then
21:       return R (or any point in R)
22:    else
23:       p_o ← the location with the smallest cost in the intersection I (represented
          by a set of point intervals)
24:       return {p_o}
25: else
26:    p_o ← the location with the smallest cost in the intersection I (represented by
       a set of point intervals)
27:    return {p_o}
```

THEOREM 1. *MinMax-Alg returns the optimal solution for the MinMax query.*

PROOF. It is easy to verify the correctness of MinMax-Alg with Lemma 2 and Lemma 3. □

Time and Space Complexities: Consider the time complexity of Algorithm 2. The major cost of the operations from line 1 to line 13 of this algorithm comes from computing $c.dist$ for all clients (Line 1), sorting (Line 2) and NLC building (Line 8). Computing $c.dist$ for all clients take $O(|C| \cdot l_s)$ time. The sorting step takes $O(|C| \log |C|)$ time. Let γ be the number of the clients examined in the algorithm. Note that $\gamma \leq |C|$. Since each NLC building takes $O(|V| \log |V|)$ time, the total NLC building for all clients examined takes $O(\gamma \cdot |V| \log |V|)$ time. The overall time complexity of the operations from line 1 to line 13 of this algorithm is $O(|C| \cdot l_s + |C| \log |C| + \gamma \cdot |V| \log |V|)$. As described

previously, the operations from line 14 to line 27 of this algorithm (i.e., how to find optimal location in the critical intersection) takes $O(\alpha + |\Theta| \cdot l_c \cdot |C'| \log |C'|)$ time. Thus, the total time complexity of this algorithm is $O(|C| \cdot l_s + |C| \log |C| + \gamma \cdot |V| \log |V| + \alpha + |\Theta| \cdot l_c \cdot |C'| \log |C'|)$. Since l_s, l_c and α are small constants in practice, $|\Theta| = O(|V|)$ and $|C'| = O(|C|)$, the time complexity can be simplified as $O(\gamma \cdot |V| \log |V| + |V| \cdot |C| \log |C|)$. It is easy to verify that the storage complexity of this algorithm is $O(|V| + |C|)$ (since this algorithm includes the storage space for running the Dijkstra's algorithm (i.e., $O(|V|)$) and the storage space for storing $c.dist$ of all clients c (i.e., $O(|C|)$)).

6. EXTENSION

In this section, we discuss three extensions to the optimal location query problem. In Section 6.1, we study to find multiple locations (instead of a single location) for the optimal location query. In Section 6.2, we discuss the optimal location query on a three-dimensional (3D) road network. In Section 6.2, we extend the techniques based on NLC to handle the optimal location query with a new objective called *MaxSum*.

6.1 Multiple-Location MinMax Query

The existing optimal location query usually finds *one* optimal location in the road network. In practice, we sometimes want to find multiple locations and set up a server at each of these locations. Specifically, given a positive integer k, we want to find k locations, namely p_1, p_2, p_3, ..., p_k, in the road network such that $\max_{c \in C} w(c) \cdot (d(c, NN_{S \cup \{s(p_1), s(p_2), ..., s(p_k)\}}(c))$ is minimized. We call this problem the *multiple-location MinMax query* problem. To the best of our knowledge, this problem has not been studied in the literature.

Next, we show that this problem is NP-hard and then propose a greedy algorithm (GA) for this problem.

THEOREM 2. *The multiple-location MinMax query problem is NP-hard.*

PROOF. We show the NP-hardness of our problem by transforming an existing NP-complete problem called the *exact cover by 3-sets* to our problem.

Exact Cover by 3-Sets (XC3S): Given a positive integer q, a set X of $3q$ elements and a set $\mathcal{C}$ of size 3 subsets of X, does there exist a subset $\mathcal{C}'$ of $\mathcal{C}$ such that each element in X is in exactly one of the set in $\mathcal{C}'$?

In order to show the NP-hardness of our problem, we have to give a decision version of our problem as follows. Given a set Y of points, we define $U(Y)$ to be $\max_{c \in C} w(c) \cdot (d(c, NN_{S \cup \{s(p) | p \in Y\}}(c))$.

Multiple-Location MinMax Query/Problem (MLMMQ): Given (1) a set S of servers, (2) a set C of clients where each client c in C is associated with a weight $w(c)$ which is a positive integer, (3) a road network $G = (V, E)$ where V is a vertex set and E is an edge set, (4) a positive integer k, and (5) a non-negative real number H, does there exist k points, namely $p_1, p_2, ..., p_k$, on edges of G such that $U(\{p_1, p_2, ..., p_k\}) \leq H$?

We transform problem XC3S to our problem MLMMQ as follows. Firstly, we create vertices as follows. For each element $e \in X$, we create a vertex v_e. We initialize a variable $\mathcal{A}$ to $\emptyset$. For each set a in $\mathcal{C}$, we create a vertex v_a and insert it into $\mathcal{A}$. Besides, we create a vertex v_o. All vertices created form the vertex set V. Secondly, we create edges as follows. For each element e in X, we create an edge (v_o, v_e) and set its weight to 2. For each set a in $\mathcal{C}$ and each element e in a, we create an edge (v_e, v_a) and set its weight to 1. All edges created form the edge set E. Thirdly, we create clients and servers as follows. For each element e in X, we create a client c_e with its weight equal to 1 and put it at vertex v_e. All clients created form the client set C. Besides, we create a server s_o and put it at vertex v_o. Only server s_o forms the server set S. Fourthly, we set k to q and H to $3q$.

Since H is equal to $3q$ and k is equal to q, we deduce that the k positions to be found, namely $p_1, p_2, ..., p_k$, must be at the vertices in $\mathcal{A}$ so that $U(\{p_1, p_2, ..., p_k\})$ is at most H. It is easy to see that this transformation can be constructed in polynomial time. It is also easy to verify that when the problem is solved in the transformed MLMMQ, the original XC3S problem is also solved. Since XC3S is an NP-complete problem, the MLMMQ problem is NP-hard. □

Since this problem is NP-hard, we propose a greedy algorithm (GA), a heuristic-based method, for this problem which involves three steps. The first step is to execute MinMax-Alg (i.e., Algorithm 2) based on the current set S of servers for finding an optimal location for a new server. The second step is to insert the new server into S. The third step is an iterative step which executes the first step and the second step iteratively until k new servers are found.

6.2 Optimal Location Query on 3D Road Network

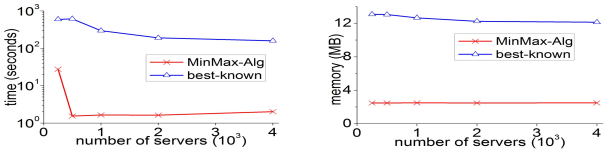
In some cases, location-based analysis can be performed on a three-dimensional (3D) road network such as the road network in a 3D map. In general, a 3D road network [10] can be modeled as an undirected graph $G_{3D} = (V_{3D}, E_{3D})$ with a vertex (edge) set V_{3D} (E_{3D}). A vertex in V_{3D} has the coordinates in a 3D space. An edge in E_{3D} has its left vertex and its right vertex. Clients and servers are located on the edges of the 3D road network. Usually, each location in the 3D road network can correspond to the longitude, latitude and height of this location.

Similar to the 2D road network discussed before, the 3D road network is also an undirected graph in which each vertex is associated with a small number of edges. Thus, the proposed MinMax query algorithm (i.e., Algorithm 2) originally designed for the original optimal location query on the 2D road network and the *GA* algorithm (the algorithm described in Section 6.1) originally designed for the multiple-location MinMax query problem on the 2D road network can be easily extended to handle the original optimal location query and the new multiple-location MinMax query problem on the 3D road network.

6.3 Optimal Location Query with the MaxSum objective

In this section, we study a new objective *MaxSum* for the optimal location query problem which is to find a location p in the network such that $\sum_{c \in C} w(c) \cdot (d(c, NN_S(c)) \geq d(c, p))$ is maximized where $(\cdot \geq \cdot)$ returns 1 if it is true and 0 otherwise. We call the optimal location query problem with the MaxSum objective the *MaxSum query*. There are many applications for this MaxSum query. Consider that C and S denote a set of residential estates and a set of convenience stores, respectively. Different convenience stores have their competitive relationship. Assume that the customers in a residential estate would be more interested in visiting a convenience store based on their distances. The MaxSum query returns all locations for a new convenience store such that if a new store is built at any of these locations, it would attract the greatest possible number of customers.

The best-known method proposed by [17] for the MinMax query can also be adapted for the MaxSum query. Details can be found in [17]. However, the shortcomings of the adapted method mentioned



(a) time (b) storage

Figure 6: Effect of $|S|$ on SF for the MinMax query

in Section 1 still exist for the MaxSum query. Motivated by this, we propose an efficient method called *MaxSum-Alg* which can also make use of the concept of NLC introduced in Section 4 for the MinMax query.

Before we introduce *MaxSum-Alg*, we give a concept called "influence value" and a lemma, which are used in *MaxSum-Alg*.

DEFINITION 6. *Given a point p on G, the* influence value *of p, denoted by $Inf(p)$, is defined to be $\sum_{c \in C s.t. p \in NLC(c)} w(c)$.*

Based on the above definition, it is easy to verify that the MaxSum query is to find the location whose influence value is maximized.

Next, we generalize the concept of "influence value" from a point to a point interval. Given a point interval $\mathcal{I}$ on an edge of G, $\mathcal{I}$ is said to be *consistent* if for any two points on $\mathcal{I}$, namely p and p', $Inf(p) = Inf(p')$. Given a consistent point interval $\mathcal{I}$ on an edge of G, $\mathcal{I}$ is said to be *maximal* if there does not exist a point interval $\mathcal{I}''$ such that $\mathcal{I} \subseteq \mathcal{I}''$ and $\mathcal{I}''$ is consistent. Given a (maximal) consistent point interval $\mathcal{I}$ on an edge of G, the *influence value* of $\mathcal{I}$ is defined to be $Inf(p)$ where p is a point in $\mathcal{I}$.

Next, we present a lemma which gives hints about where we can find the optimal solution.

LEMMA 4 (OPTIMAL SOLUTION). *Let p_o be the optimal location for the MaxSum query. Let C_{p_o} be a set of clients attracted by p_o. If $|C_{p_o}| \geq 2$, then p_o is in the intersection of the NLCs of all clients in C_{p_o}.*

PROOF. Given any optimal location p_o with the largest influence value. Assume that all the clients attracted by p_o are $c_1, c_2, \ldots, c_m$. Then, we have $p_o \in NLC(c_i)$, where $1 \leq i \leq m$. Thus, $p_o \in \cap_{i=1}^{m} NLC(c_i)$. That means, any optimal location corresponds to the intersection of the NLCs. □

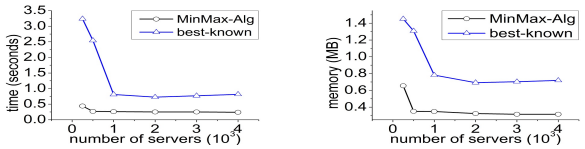
The above lemma describes the case when $|C_{p_o}| \geq 2$. When $|C_{p_o}| = 1$, it is easy to verify that the optimal location is in the NLC of the client with the greatest weight.

Lemma 4 gives a powerful tool to find an optimal solution. Specifically, according to Lemma 4, we design the following two-step algorithm called *MaxSum-Alg*. The first step is that for each edge e in E, we find the set of all clients whose NLCs overlap with e and select the (maximal consistent) point interval on e with the greatest influence value. The second step is to find the set of all sub-intervals with the greatest influence value.

It is easy to verify the correctness of this algorithm. This algorithm can be enhanced with some pruning techniques. For the sake of space, a detailed version of *MaxSum-Alg* can be found in the full version of this paper [3].

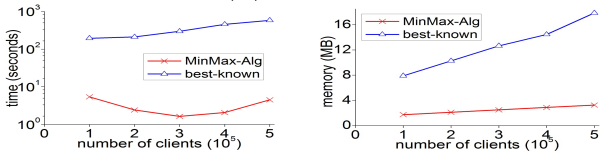
7. EMPIRICAL STUDIES

In this section, we conducted experiments to show that our algorithms are efficient. In the experiments, we compare our algorithms with the best-known algorithms. We implemented our algorithms in C++ and obtained the source code of the best-known algorithms from the authors (`http://www.cs.sjtu.edu.cn/~yaobin/olq/`). All the experiments were performed on a Linux machine with an Intel 3GHz CPU and 4GB memory. The running time and the storage cost were reported in the experiments. The experiments include two parts. One part is about the experiments for the MinMax query (Section 7.1). The other part is about the experiments for the extension to the MinMax query (Section 7.2).



(a) time (b) storage

Figure 7: Effect of $|S|$ on CA for the MinMax query



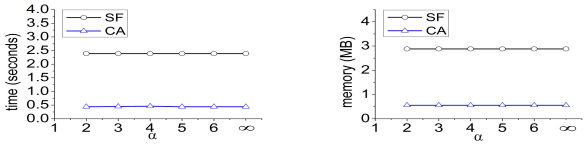
(a) time (b) storage

Figure 8: Effect of $|C|$ on SF for the MinMax query

7.1 Experiments for the MinMax Query

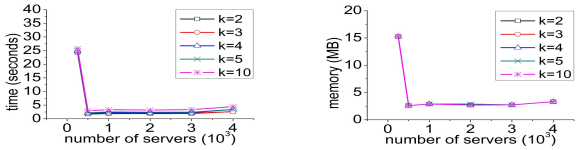
In this section, we compared our method, *MinMax-Alg*, with the best-known algorithm [17]. Similar to [17], two real road network datasets, SF and CA, were used in the experiments. SF and CA are real datasets for the road networks in San Francisco and California, respectively. SF contains 174,955 vertices and 223,000 edges, and CA includes 21,047 vertices and 21,692 edges. In the real road network datasets, the max/min/avg number of edges adjacent to a vertex is equal to 8/1/3, respectively. Most vertices (specifically, 174,231 vertices out of 174,955 vertices) involve at most 4 edges each. There is only one vertex adjacent to 8 edges. The clients and servers were generated in the way similar to [17]. Specifically, we obtained a large number of real building locations in San Francisco (California) from the $OpenStreetMap$ project. Note that the road network of CA has a coarser data granularity and is associated with fewer building locations. Each building location is projected on one of the road network edges nearest to this building. Then, we randomly sampled those locations in SF (CA) as servers and clients. The clients and the servers are stored in two separate lists. Each client is associated with a weight generated randomly from a Zipf distribution with a skewness parameter $\alpha > 1$. By default, α is set to be ∞ and this means that the weight of each client is equal to 1.

The other default settings are as follows. The number of servers and the number of clients for SF (CA) are equal to 1,000 (250) and 300,000 (40,000), respectively. Note that the best-known method [17] has an input parameter called the partition parameter θ which is used to determine a set of vertices randomly picked from the graph to generate a number of subgraphs of G. As shown in [17], the best-known algorithm has the *best* performance when θ is set to



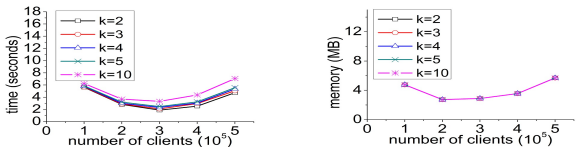
(a) time (b) storage

Figure 9: Effect of α for the MinMax query



(a) time (b) storage

Figure 10: Effect of $|S|$ on *SF* for the multiple-location MinMax query



(a) time (b) storage

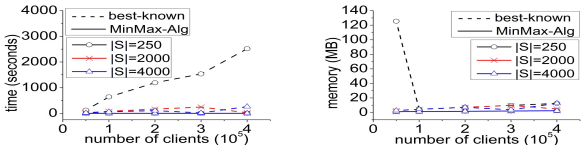
Figure 11: Effect of $|C|$ on *SF* for the multiple-location MinMax query

1‰ which is also the default setting here.

In the experiments, we study the effects of $|S|$, $|C|$ and α.

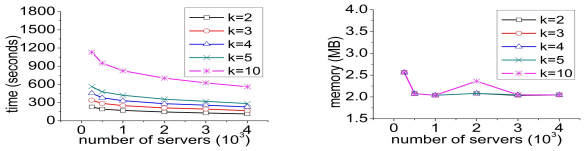
Effect of $|S|$: We study the effect of $|S|$ on the *SF* dataset in Figure 6(a) and Figure 6(b). In Figure 6(a), MinMax-Alg is faster than the best-known method by at least an order of magnitude. In particular, the time of MinMax-Alg is extremely small and is less than 10 seconds in most cases. This is because the number of clients examined by MinMax-Alg is very small and thus MinMax-Alg can find the solution quickly. Besides, the best-known method has to generate additional vertices for clients and servers in the road network, which increases the search space a lot and thus slows down the whole algorithm. Besides, the time of MinMax-Alg decreases and then increases. This is because the time of MinMax-Alg depends on two factors. The first factor is the time of building the NLCs of all clients and the second factor is the time of processing servers/clients. When $|S|$ is very small, the NLC of each client becomes larger and thus, the first factor outweighs the second factor, which increases the overall time of MinMax-Alg. When $|S|$ becomes larger, the second factor dominates the first factor and thus the time of MinMax-Alg increases with $|S|$. Similarly, the time of the best-known method is large when $|S|$ is small. This is because the search space examined by the best-known method is larger when $|S|$ is smaller. In Figure 6(b), the memory consumption of MinMax-Alg is lower than that of the best-known method since the best-known method has to maintain a larger search space compared with MinMax-Alg. Similarly, Figure 7(a) and Figure 7(b) show the experimental results about the effect of $|S|$ on the *CA* dataset. Similar trends can also be found in this dataset but with a shorter time and a lower memory consumption of each algorithm since *CA* is smaller. When $|S|$ is very small, the large memory consumption of MinMax-Alg (in Figure 7(b)) could be explained by the fact that large storage is needed to store the set of point intervals representing the NLCs of all clients and that of the best-known method is because of its the large search space.

Effect of $|C|$: We study the effect of $|C|$ on the *SF* dataset in Figure 8(a) and Figure 8(b). As shown in Figure 8(a), MinMax-Alg is faster than the best-known method. The time of the best-known method increases with $|C|$. The time of MinMax-Alg decreases and then increases when $|C|$ increases. The decrease trend can be explained by the fact that the first factor (the time of building NLCs) dominates the second factor (the time of processing servers/clients) and the increase trend is due to that the second factor outweighs the first factor. For the sake of space, the results about the effect of $|C|$ on the *CA* dataset can be found in the full version of the paper [3]



(a) time (b) storage

Figure 12: Effect of $|C|$ on $3D$ for the MinMax query



(a) time (b) storage

Figure 13: Effect of $|S|$ on $3D$ for the multiple-location MinMax query

and the pattern is similar to that on the *SF* dataset.

Effect of α: The experimental results about the effect of α can be found in Figure 9(a) and Figure 9(b). These two figures show that the time and the memory consumption of MinMax-Alg for the *SF* dataset and the *CA* dataset are not sensitive to α.

Results on Large Datasets: We observed that the total number of all real buildings in San Francisco in the $OpenStreetMap$ project is more than 10 million. Thus, we also conducted experiments on massive datasets for the real road networks in San Francisco where the number of clients (servers) is 10 million (250). The best-known algorithm performance is much worse due to its poor pruning effect. However, our algorithm is still efficient in this case. For example, the query time of the best-known algorithm is more than 10 hours but the query time of our algorithm is within 3 minutes. Due to the limit of space, the details are not shown.

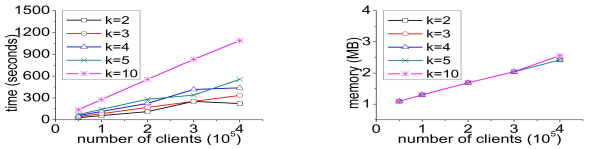
7.2 Experiments for Extensions

In this section, we give some experimental results about the extensions described in Section 6. In the following, for the sake of space, we only show the experimental results on the SF dataset. The results on the CA dataset could be found in the full version of our paper [3].

Multiple-Location MinMax Query: In this section, we study the performance of the *GA* algorithm for the optimal multiple-location query. The experimental results of the effect of $|S|$ on the *SF* dataset with different values of k can be found in Figure 10(a) and Figure 10(b). In Figure 10(a), similar to the original MinMax query when $|S|$ is small, the time of GA is large. When $|S|$ becomes larger, the time of GA increases slightly. Besides, when k is larger, the time of GA increases slightly. In Figure 10(b), similarly, we have a larger memory consumption of GA when $|S|$ is small. Besides, the memory increases slightly with $|S|$. Since the memory consumption of GA is independent of k, the memory consumption of GA is the same for different values of k.

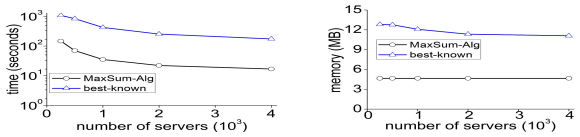
The experimental results about the effect of $|C|$ on the *SF* dataset can be found in Figure 11(a) and Figure 11(b).

MinMax Query on 3D Road Network: In this section, we study our extension on a three-dimensional (3D) road network. The 3D road network adopted in the experiment is the three-dimensional (3D) road network dataset downloaded from `https://archive.ics.uci.edu/ml/datasets/3D+Road+Network+(North+Jutland,+Denmark)`. The 3D road network dataset was generated based on a 2D road network in North Jutland, Denmark. Each location point in the 3D road net-



(a) time (b) storage

Figure 14: Effect of $|C|$ on $3D$ for the multiple-location MinMax query



(a) time (b) storage

Figure 15: Effect of $|S|$ on *SF* for the MaxSum query

work includes its longitude, latitude and height values. The 3D road network includes more than 90,000 edges and 180,000 vertices. In this dataset, there are more than 400,000 *ground points* located on the edges of the 3D road network which can be regarded as clients or servers. We randomly sampled points from these ground points in the 3D road network dataset as servers and clients.

We conducted the experiments for the original MinMax query on the 3D road network. Figure 12(a) and Figure 12(b) show the time and the memory consumption of the methods when $|C|$ increases. In Figure 12(a), similarly, MinMax-Alg performs more efficiently than the best-known method. Besides, when $|C|$ increases, the time of MinMax-Alg decreases. Figure 12(b) shows a similar trend as the one based on the 2D road network.

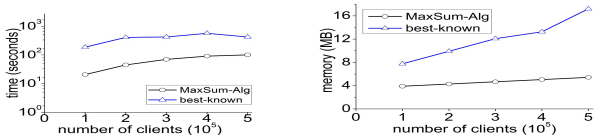
We also conducted experiments for the multiple-location MinMax query on the 3D road network. The experimental results of the effect of $|S|$ on the $3D$ dataset are shown in Figure 13(a) and Figure 13(b). The experimental results of the effect of $|C|$ on the $3D$ dataset are shown in Figure 14(a) and Figure 14(b).

MaxSum Query: In this part, we study the performance of algorithms for the MaxSum query on the SF dataset. We compared our MaxSum-Alg with the best-known algorithm. The experimental results of the effect of $|S|$ are shown in Figure 15(a) and Figure 15(b). In Figure 15(a), MaxSum-Alg perform more efficiently than the best-known method since MaxSum-Alg uses effective NLC-based pruning techniques to speed up the MaxSum query process but the best-known method has to explore a large search space. Besides, similar experimental results of the effect of $|C|$ are shown in Figure 16(a) and Figure 16(b).

Summary. For the MinMax query, our algorithm, MinMax-Alg, is faster than the best-known algorithm by at least one order of magnitude on the benchmark datasets with large sizes. In particular, the MinMax query and the MaxSum query can be answered within several seconds in most cases. The storage of our algorithm is also small and often requires less than 10MB.

8. CONCLUSIONS

In this paper, we propose a new algorithm framework based on the idea of nearest location component for the optimal location query in the context of road networks. We present an efficient algorithm, namely MinMax-Alg, for the optimal location query. We also discuss three extensions to the optimal location query. We conducted extensive experimentst whose results showed that our algorithms are more efficient than the best-known method. In the future, we would like to develop techniques for the optimal location query with moving clients and servers.



(a) time (b) storage

Figure 16: Effect of $|C|$ on *SF* for the MaxSum query

Acknowledgements: We are grateful to the anonymous reviewers for their constructive comments on this paper. The research of Yubao Liu, Zitong Chen, Jiaming Xiong and Ganlin Mai is supported by NSFC project no. 61070005. The research of Raymond Chi-Wing Wong and Cheng Long is supported by grant FSGRF13EG27.

9. REFERENCES

[1] S. Cabello, J. M. Diaz-Banez, S. Langerman, C. Seara, and I. Ventura. Reverse facility location problems. In *CCCG*, 2005.

[2] J. Cardinal and S. Langerman. Min-max-min geometric facility location problems. In *EWCG*, 2006.

[3] Z. Chen, Y. Liu, R. C.-W. Wong, J. Xiong, G. Mai, and C. Long. Efficient algorithms for optimal location queries in road networks (technical report). In *http://www.cse.ust.hk/~raywong/paper/olq-technical.pdf*, 2014.

[4] D.-W. Choi, C.-W. Chung, and Y. Tao. A scalable algorithm for maximizing range sum in spatial databases. *PVLDB*, 2012.

[5] M. de Berg, M. van Krefeld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2000.

[6] E. W. Dijkstra. A note on two problems in connexion with graphs. *NUMERISCHE MATHEMATIK*, 1(1):269–271, 1959.

[7] Y. Du, D. Zhang, and T. Xia. The optimal-location query. In *SSTD*, 2005.

[8] M. Erwig. The graph voronoi diagram with applications. *Networks*, 36(3):156–163, 2000.

[9] J. Hershberger. Finding the upper envelope of n line segments in o(nlogn) time. *Inf. Process. Lett.*, 4(33):169–174, 1989.

[10] M. Kaul, B. Yang, and C. S. Jensen. Building accurate 3d spatial networks to enable next generation intelligent transportation systems. In *MDM*, 2013.

[11] J. Krarup and P. M. Pruzan. The simple plant location problem: Survey and synthesis. *European Journal of Operational Research*, 12(1):36–57, 1983.

[12] Y. Liu, R. C.-W. Wong, K. Wang, Z. Li, C. Chen, and Z. Chen. A new approach for maximizing bichromatic reverse nearest neighbor search. *Knowl. Inf. Syst.*, 36(1):23–58, 2013.

[13] J. Qi, R. Zhang, L. Kulik, D. Lin, and Y. Xue. The min-dist location selection query. In *ICDE*, 2012.

[14] B. C. Tansel, R. L. Francis, and T. J. Lowe. Location on networks: A survey. *Management Science*, 29(4):498–511, 1983.

[15] R. C.-W. Wong, M. T. Ozsu, A. W.-C. Fu, P. S. Yu, and L. Liu. Efficient method for maximizing bichromatic reverse nearest neighbor. *PVLDB*, 2009.

[16] R. C.-W. Wong, M. T. Ozsu, A. W.-C. Fu, P. S. Yu, L. Liu, and Y. Liu. Maximizing bichromatic reverse nearest neighbor for lp-norm in two- and three-dimensional spaces. *VLDB J.*, 2011.

[17] X. Xiao, B. Yao, and F. Li. Optimal location queries in road network databases. In *ICDE*, 2011.

[18] D. Yan, R. C.-W. Wong, and W. Ng. Efficient methods for finding influential locations with adaptive grids. In *CIKM*, 2011.

[19] D. Zhang, Y. Du, T. Xia, and Y. Tao. Progressive computation of the min-dist optimal-location query. In *VLDB*, 2006.

[20] Z. Zhou, W. Wu, X. Li, M. L. Lee, and W. Hsu. Maxfirst for maxbrknn. In *ICDE*, 2011.