Optimal Location Queries in Road Networks

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

In this paper, we study an optimal location query based on a road network. Specifically, given a road network containing clients and servers, an optimal location query is to find a location on the road network 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. Two types of cost functions, namely MinMax and MaxSum, have been used for this query. The optimal location query problem with MinMax as the cost function is called the MinMax query which is to find a location for setting up a new server such that the maximum cost of a client being served by his/her closest server is minimized. The optimal location query problem with MaxSum as the cost function is called the MaxSum query which is to find a location for setting up a new server such that the sum of *weights* of clients attracted by the new server is maximized. The MinMax query and the MaxSum query correspond to two types of optimal location query with the objectives defined from the clients' perspective and from the new server's perspective, respectively. Unfortunately, the existing solutions for the optimal query problem are not efficient. In this paper, we propose an efficient algorithm, namely *MinMax-Alg* (*MaxSum-Alg*), for the MinMax (MaxSum) query, which is based on a novel idea of *nearest location component*. We also discuss two extensions of the optimal location query, namely the optimal multiple-location query and the optimal location query on a 3D road network. Extensive experiments were conducted, showing 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, in our largest real datasets, the state-of-the-art ran for more than 10 (12) hours while our algorithm ran within 3 (2) minutes only for the MinMax (MaxSum) query, i.e., our algorithm ran at least 200 (600) times faster than the state-of-the-art.

Categories and Subject Descriptors: H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms: Algorithms

Additional Key Words and Phrases: Optimal location query, road network, nearest location component

ACM Reference Format:
Zitong Chen, Yubao Liu, Raymond Chi-Wing Wong, Jiamin Xiong, Ganglin Mai, and Cheng Long, 2015. Optimal Location Queries in Road Networks. *ACM Trans. Datab. Syst.* V, N, Article A (January YYYY), 39 pages.
DOI : http://dx.doi.org/10.1145/0000000.0000000

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/

Author's addresses: Zitong Chen, Department of Computer Science, Sun Yat-sen University, China; email: 453140207@qq.com; Yubao Liu (corresponding author), Department of Computer Science, Sun Yat-sen University, China; email: liuyubao@mail.sysu.edu.cn; Raymond Chi-Wing Wong, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong; email: raywong@cse.ust.hk; Jiamin Xiong, Department of Computer Science, Sun Yat-sen University, China; email: 812023634@qq.com; Ganglin Mai, Department of Computer Science, Sun Yat-sen University, China; email: 415279896@qq.com; Cheng Long, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong; email: clong@cse.ust.hk.
Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© YYYY ACM 0362-5915/YYYY/01-ARTA $15.00
DOI : http://dx.doi.org/10.1145/0000000.0000000

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).

Given a set C of clients and a set S of servers on 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 on the road network 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 has been used as a basic operation in many real applications such as location planning, location-based service and profile-based marketing [Cabello et al. 2005; Xiao et al. 2011; de Berg et al. 2000].

In general, in OLQ, each client finds the server that is closest to him/her for service and his/her cost of being served is equal to the (network) distance between the client and the server serving him/her multiplied by his/her weight or importance. In [Xiao et al. 2011], two types of cost function were studied for the optimal location query, namely MinMax and MaxSum. The optimal location query with MinMax as the cost function (called the MinMax query) is to find a location for setting up a new server such that the maximum cost of a client being served by his/her nearest server (including the new server) is minimized. The optimal location query with MaxSum as the cost function (called the MaxSum query) is to find a location for setting up a new server such that the sum of *weights* of clients attracted by the new server is maximized.

The intuition of the MinMax query is to optimize the *worst-case* cost (or the maximum cost) of a client, and it has many applications in real life.

Application Example [The MinMax Query]. In many facility location applications, we sometimes need to set up a new facility (e.g., a hospital, a fire station and a police station) on a road network on which some existing facilities of the same type have been set up already such that the maximum distance between a residential location and its nearest facility is minimized. Besides, the MinMax query could be used in some emergency applications where the problem is to place supply/service centers for rescue or relief jobs.

The intuition of the MaxSum query is to maximize the total sum of the weights of all clients attracted by the new server, and it has many applications.

Application Example 2 [The MaxSum query]. In some business applications, the MaxSum query could be used to find a location for setting up a new business unit (e.g., a convenience store) such that it attracts as many customers as possible (assuming that the customers are more interested in choosing the nearest convenience store to them).

An algorithm was designed for the MinMax (MaxSum) query in [Xiao et al. 2011], and the 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/sub-graphs for solving the problem. As a consequence, it 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 vertices as large as $|V|+|S|+|C|$ and the number of edges as large as $|E| + |S| + |C|$, both of which are 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 [Xiao et al. 2011] 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 algorithms [Xiao et al. 2011], in this paper, we design an efficient algorithm called *MinMax-Alg* (*MaxSum-Alg*) for the MinMax (MaxSum) query which avoids the above shortcomings. Specifically, we have the following contributions.

Firstly, we design efficient algorithms for the optimal location query on a road network, which are based on the original road network but not the larger augmented road network. We develop several

new pruning techniques based on the idea of *nearest location component (NLC)* of the clients, which dramatically reduce the search space of the algorithm. The time complexity of our algorithm is significantly smaller than that of the state-of-the-art [Xiao et al. 2011]. In particular, the time complexity of *MinMax-Alg* 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 [Xiao et al. 2011] where $|C|$ is 300k and $|E|$ is 223k, γ is equal to 27. The time complexity of *MaxSum-Alg* is $O(\xi \cdot |V| \log |V| + |\mathcal{E}| \log |\mathcal{E}| + \epsilon \cdot |V| \log |V|)$, where ξ is at most $|C|$, $|\mathcal{E}|$ is at most $|E| + |S|$ and ϵ is at most $|E|$. Note that ξ and ϵ are usually smaller than $|C|$ and $|E|$ in practice. In our experiments on the *SF* real dataset with the default setting, ξ is 63k and ϵ is 12.

Secondly, we discuss two extensions to our problem, namely the problem of finding multiple locations (instead of a single location) for the optimal location query (this is called the *optimal multiple-location query*) and the optimal location query on a 3D road-network [Kaul et al. 2013]. For the optimal multiple-location query, we prove its NP-hardness and develop a greedy algorithm (GA). For the optimal location query on 3D road networks, we show how it could be solved by adapting the algorithms based on (2D) road networks.

Thirdly, we conducted extensive experiments to verify the efficiency of our algorithm. Our algorithm is significantly faster than the state-of-the-art by at least an order of magnitude on large datasets. For example, in our largest datasets, the state-of-the-art [Xiao et al. 2011] ran for more than 10 (12) hours while our algorithm ran within 3 (2) minutes only for the MinMax (MaxSum) query, i.e., our algorithm ran at least 200 (600) times faster than the state-of-the-art.

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. Sections 5 introduces our algorithm *MinMax-Alg* and Sections 6 introduces our algorithm *MaxSum-Alg*. Sections 7 discusses the extensions of our problem. Section 8 gives the empirical study and Section 9 concludes the paper.

2. PROBLEM DEFINITION

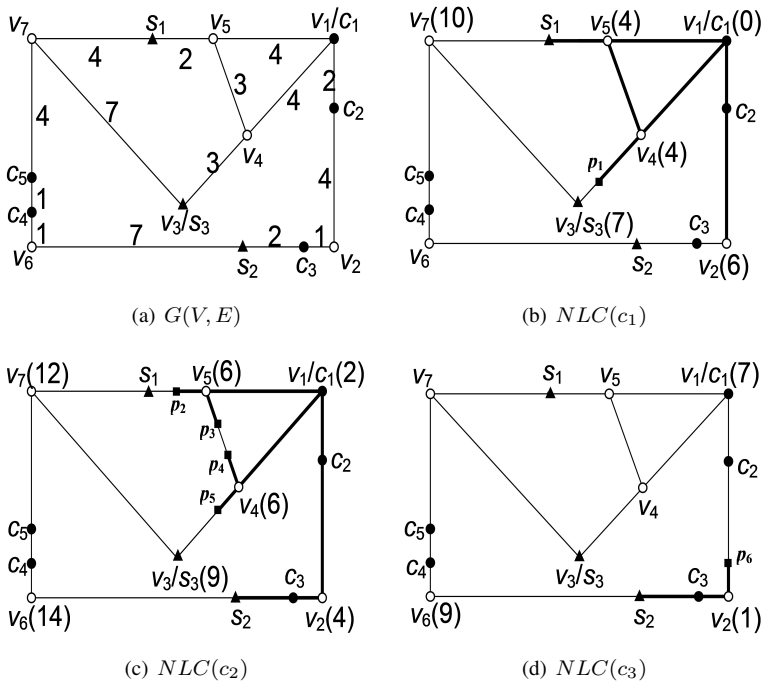
Let $G = (V, E)$ be a road network, and C (S) be a set of clients (servers) on G. For any edge $e = (v_l, v_r)$ of G, v_l (v_r) is the left (right) vertex of e. We adopt the network distance metric to define the distance between two locations on the road network and denote it by $d(\cdot, \cdot)$. Let c be a client in C. We denote c's closest server in S by $NN_S(c)$. 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. In most applications, given a client c, $w(c)$ corresponds to the number of residents (or population) at the location of c. We define the *cost value* of c, denoted by $Cost(c)$, to be $w(c) \cdot c.dist$.

Example 2.1 (*Running example*). *Consider an example of a road network G in Figure 1(a). In this figure, each line segment corresponds to an edge, and each unfilled circle, each filled circle and each triangle corresponds to a vertex, a client and a server in the road network, respectively. 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. In addition, each filled square in Figure 1(b), Figure 1(c) and Figure 1(d) corresponds to a point in the road network.*

For simplicity, in this running example, we assume that the weight of each client is 1 if we do not specify its weight explicitly.

Formally, the optimal location query (OLQ) problem with the MinMax (MaxSum) cost function is defined as follows.

PROBLEM 1 (THE MINMAX QUERY). *Given a road network $G = (V, E)$, a set C (S) of clients (servers) on G, the OLQ problem with the MinMax cost function is to find a location p*



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

(c) $NLC(c_2)$ (d) $NLC(c_3)$

Fig. 1. A Running Example

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.

PROBLEM 2 (THE MAXSUM QUERY). *Given a road network* $G = (V, E)$*, a set* C *(*S*) of clients (servers) on* G*, the OLQ problem with the MaxSum function is to find a location* p *such that* $\sum_{c \in C} w(c) \cdot (d(c, NN_{S \cup \{s(p)\}}(c)) \geq d(c, s(p)))$ *is maximized where* $(\cdot \geq \cdot)$ *returns 1 if it is true and 0 otherwise. We also call this problem the* MaxSum query.

Next, we introduce a key concept called *nearest location component* (NLC), which would be used in our algorithms.

Definition 2.2. 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 $G\}$.

Example 2.3. *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*.* $NLC(c_2)$ *and* $NLC(c_3)$ *are shown in Figure 1(c) and Figure 1(d), respectively.*

An edge e is *covered* by $NLC(c)$ if there exists such a point p along edge e such that $p \in NLC(c)$. There are two types of coverage, namely "complete coverage" and "partial coverage". 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 $NLC(c_1)$. An edge is *partially covered* by $NLC(c)$ if some (but not all) of the points along the edge are included in $NLC(c)$. For example, the edges (v_3, v_4) and (v_5, v_7) are partially covered by $NLC(c_1)$.

Table I.
Basic 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 $j \in [1, m]$
$\mathcal{I}$	a point interval set
I	a point interval
$\mathcal{R}$	the $(m_o, Cost(c_{m_o+1}))$-critical intersection
$Inf(p)$	the influence value of p
$\mathcal{S}$	the set of all clients whose NLCs cover e
$\mathcal{S}'$	the set of all effective clients whose NLCs cover e
$GPI(e)$	the greatest possible influence of e
$GPEI(e)$	the greatest possible effective influence of e

Let p be a location in $NLC(c)$ for a client c. Suppose that a new server is set up at location p. It immediately follows that the new server corresponds to the nearest server to client c and thus client c would be *attracted* by the new server. In this case, for simplicity, we say that client c is attracted by location p.

We introduce another concept called *point interval* which could be used to represent NLCs of clients. Specifically, 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 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.

Note that the NLC of a client can be represented by a set of point intervals. For example, the point interval of $NLC(c_1)$ on edge (v_1, v_2) is $[0, 6]$.

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

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 [Cabello et al. 2005; Cardinal and Langerman 2006; Meyerson 2001; Krarup and Pruzan 1983; Tansel et al. 1983; Wong et al. 2009; Wong et al. 2011; Liu et al. 2013; Zhou et al. 2011; Yan et al. 2011] 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.

The optimal location query, which was proposed based on the *facility location problem* and is also known as *location analysis* [Cabello et al. 2005; Cardinal and Langerman 2006; Meyerson 2001; Krarup and Pruzan 1983; Tansel et al. 1983], 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 in the space). Recently, the researchers in the database community paid attention to the optimal location query because it has a broad range of applications.

The MaxBRNN problem [Cabello et al. 2005] 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 [Cabello et al. 2005]. The MaxBRNN problem was also studied in [Wong et al. 2009] 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 [Wong et al. 2011]. Recently, the *MaxSegment* algorithm, an improved algorithm for the MaxBRNN problem, was given in [Liu et al. 2013]. Both the running time and the storage cost of the *MaxSegment* algorithm are significantly smaller than the *MaxOverlap* algorithm. A generalized MaxBRkNN problem [Zhou et al. 2011] 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 [Yan et al. 2011].

In addition, the algorithm in [Du et al. 2005] finds an optimal location instead of an optimal region for the L_1-norm space. The algorithm in [Zhang et al. 2006] 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 [Cardinal and Langerman 2006] 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 [Qi et al. 2012] selects a location from a given set of potential locations for a new server such that the average distance between a client and its nearest server is minimized. The algorithm in [Choi et al. 2012] 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. The algorithm in [Chen et al. 2015] considers the generalized case in which the rectangular region is rotatable.

3.2. Optimal Location Queries with Road Network Setting

Recently, Xiao el al. [Xiao et al. 2011] 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 on the *SF* (San Francisco) real dataset with the default setting where $|C|$ is 300k, $|S|$ is 1k and $|V|$ is 174k, $|V''|$ is equal to 475k (which is exactly equal to $|V| + |S| + |C|$, i.e., 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 $O(|E| + |S| + |C|)$ 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 verify that the overall time complexity of this algorithm is $O((|V|+|S|+|C|)^2 \log(|V|+|S|+|C|))$ in the worst case, which is prohibitively expensive. The space complexity is $O(|V| + |S| + |C|)$ which corresponds to the space complexity of the Dijkstra's algorithm based on the road network. We note that the size of the augmented road network graph increases linearly with the number of clients and the number of servers and an increase in the size of the augmented road network results in a quadratic increase in the query time.

In this paper, we propose a new algorithm framework which has a significantly smaller time complexity compared with the state-of-the-art algorithm. In particular, the time complexity of the proposed *MinMax-Alg* algorithm for the MinMax query is $O(\gamma \cdot |V| \log |V| + |V| \cdot |C| \log |C|)$ where γ is at most $|C|$ and the time complexity of the proposed *MaxSum-Alg* algorithm for the MaxSum query is $O(\xi \cdot |V| \log |V| + |\mathcal{E}| \log |\mathcal{E}| + \epsilon \cdot |V| \log |V|)$, where ξ is at most $|C|$, $|\mathcal{E}|$ is at most $|E| + |S|$ and ϵ is at most $|E|$.

[Ghaemi et al. 2010; 2014] also studied a static version of OLQ with the MaxSum cost function. However, their solutions for the static version of OLQ are different from ours, and the time and space complexities of their solutions are higher than ours. Specifically, their time complexity is $O(|E| \log |E| + |C||V| \log |V| + |C||E| + |E||C|^2)$. The dominant factor in this time complexity is $O(|C||V| \log |V| + |E||C|^2)$. In the *worst case*, our time complexity is $O(|C||V| \log |V| + (|E| + |S|) \log(|E| + |S|) + |E||V| \log |V|)$. The dominant factor in our time complexity is $O(|C||V| \log |V| + |E||V| \log |V|)$. Then, it is easy to know that their time complexity is quadratic to $|C|$ but our time complexity is lower than this quadratic cost. This means that their solutions do not perform well when there are a lot of clients, which is usually the case in most applications. Note that $|V|$ is usually smaller than $|C|$ in a lot of applications. Thus, their time complexity is higher than ours. Furthermore, in the worst case, their space complexity is $O(|C||E|)$ which is higher than ours (i.e., $O(|V| + |C|)$). In addition, they only focus on OLQ with the MaxSum cost function. In addition to the MaxSum cost function, we also consider OLQ with the MinMax cost function in this paper.

Besides, [Ghaemi et al. 2012] studied a dynamic version of OLQ with the MaxSum cost function. Moreover, [Yao et al. 2014], an extension of [Xiao et al. 2011], studied a dynamic version of the OLQ problem with the MinMax and MaxSum cost functions.

Recently, there was a preliminary work [Chen et al. 2014] studying the OLQ problem on road networks, which, however, did not solve the OLQ problem in an adequate way. First, for the MaxSum query, the authors [Chen et al. 2014] provide a brief description of the algorithm only without introducing the details. Second, for the optimal multiple-location query, only the MinMax cost function was studied. In this paper, we solve the OLQ problem adequately by (1) providing the details of the MaxSum-Alg algorithm for the MaxSum query and (2) studying the optimal multiple-location query with the MaxSum cost function. We also develop new pruning techniques for the MaxSum-Alg algorithm which could not be found in [Chen et al. 2014]. Besides, we conducted experiments for the new algorithms.

It is also worth mentioning that in the literature, there is a query type called *isochrone query* in a multimodal network [Bauer et al. 2008; Gamper et al. 2011; Gamper et al. 2012] which is to find all those points each of which could be reached from a given query point within a given time span and by a given arrival time. The concept of NLC newly introduced in this paper differs from an isochrone query in the following aspects. First, an *isochrone* used in the isochrone query corresponds to a subgraph of a given multimodal network in which the *whole* portion of an edge of the network is included while the NLC of a client used in our OLQ query corresponds to a set of the point intervals which could be a whole portion as well as a *partial* portion of an edge of the given road network. Second, an isochrone takes into consideration the time constraints (e.g., a given time span and a given arrival time) while an NLC does not. Third, the algorithm for building NLCs in this paper is different from the existing algorithms for an isochrone query.

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 [Dijkstra 1959]. 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 ∞.

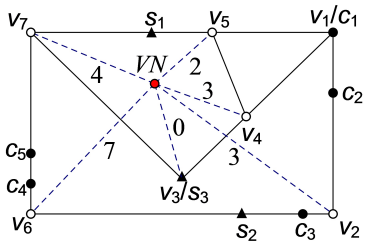


Fig. 2. The road network with a virtual node VN

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 $v.esd$.

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 [Erwig 2000] 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 [Dijkstra 1959], 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

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 4.1. *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.

Example 4.2. Consider an example of the road network in Figure 2. Consider client c_5 on edge (v_6, v_7). Note that there is no server on the edge (v_6, v_7). According Lemma 4.1, $c_5.dist = \min\{d(c_5, v_6) + v_6.dist, d(c_5, v_7) + v_7.dist\}$.

According to Property 1, $v_6.dist = 7$ and $v_7.dist = 4$. From Figure 1(a), $d(c_5, v_6) = 2$ and $d(c_5, v_7) = 4$. Thus, $c_5.dist = \min\{2 + 7, 4 + 4\} = 8$.

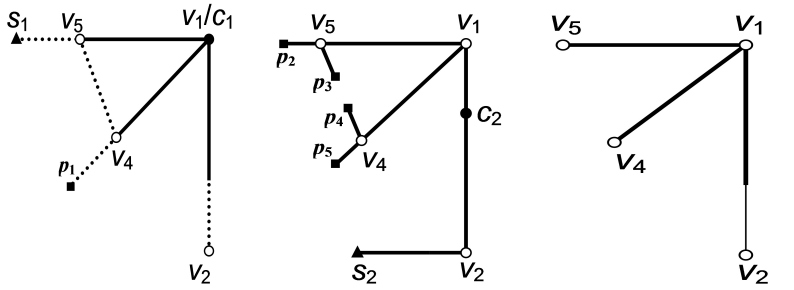
Algorithm 1 Algorithm for Finding the NLC of a Client c (i.e., $NLC(c)$)

1: use Dijkstra's algorithm to traverse the vertices in ascending order of their distances to c
2: $\mathcal{I} \leftarrow \emptyset$
3: **for** each vertex v to be processed in ascending order of its distance to c **do**
4: **if** $d(v, c) \leq c.dist$ **then**
5: **for** each edge e' adjacent to v, in the form of (v, v'), **do**
6: **if** v' is processed before **then**
7: **if** $(c.dist - d(v, c)) + (c.dist - d(v', c)) \geq d(v, v')$ **then**
8: $\mathcal{I} \leftarrow \mathcal{I} \cup \{[v, v']\}$;
9: **else**
10: $\mathcal{I} \leftarrow \mathcal{I} \cup \{[v', p']\}$ and $\mathcal{I} \leftarrow \mathcal{I} \cup \{[v, p]\}$ where p and p' are the points along the edge e' such that $d(v', p') + d(c, v') \leq c.dist$ and $d(v, p) + d(c, v) \leq c.dist$, among which $d(v, p) + d(c, v)$ and $d(v', p') + d(c, v')$ are the largest;
11: **end if**
12: **end if**
13: **end for**
14: **else**
15: // $d(v, c) > c.dist$
16: **for** each edge e'' adjacent to v, in the form of (v, v''), **do**
17: **if** $d(v'', c) \leq c.dist$ **then**
18: **if** c is on edge e'' **then**
19: $\mathcal{I} \leftarrow \mathcal{I} \cup \{[v'', p]\}$ where p is a point along the edge e'' such that $d(c, p) \leq c.dist$ and $d(c, p)$ is the largest.
20: **else**
21: $\mathcal{I} \leftarrow \mathcal{I} \cup \{[v'', p']\}$ where p' is a point along the edge e'' such that $d(c, v'') + d(v'', p') \leq c.dist$ and $d(v'', p')$ is the largest.
22: **end if**
23: **else**
24: **if** c is on edge e'' **then**
25: $\mathcal{I} \leftarrow \mathcal{I} \cup \{[q, q']\}$ where q and q' are two points along the edge e'' such that $d(c, q) = c.dist$, $d(c, q') = c.dist$ and $q \neq q'$ if $c.dist \neq 0$ and $q = q'$ otherwise.
26: **else**
27: regard e'' as deleted
28: **end if**
29: **end if**
30: **end for**
31: **end if**
32: **end for**
33: **return** $\mathcal{I}$

On the other hand, consider client c_3 on edge (v_2, v_6). Note that there is a server s_2 on the edge (v_2, v_6). According to Lemma 4.1, $c_3.dist = \min\{d(c_3, s_2), d(c_3, v_2) + v_2.dist, d(c_3, v_6) + v_6.dist\}$. From Figure 1(a), $d(c_3, s_2) = 2$ and $d(c_3, v_2) = 1$ and $d(c_3, v_6) = 9$. According to Property 1, $v_2.dist = 3$ and $v_6.dist = 7$. Thus, $c_3.dist = d(c_3, s_2) = 2$.

4.3. Building the NLC of a Client

Algorithm 1 shows the algorithm for finding the NLC of a client c (i.e., $NLC(c)$) based on $c.dist$ found in the previous step. 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). 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 $NLC(c)$.



(a) $NLC(c_1, d_1)$ (b) $NLC(c_2, d_2)$ (c) The $(2, Cost(c_2))$-critical intersection

Fig. 3. Example 1 Illustrating Critical Intersection

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 check whether $(c.dist - d(v, c)) + (c.dist - d(v', c)) \geq d(v, v')$ (Line 7), 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 8). Otherwise (Line 9), $[v', p']$ and $[v, p]$ are inserted where p and p' are the points along the edge e' such that $d(v', p') + d(c, v') \leq c.dist$ and $d(v, p) + d(c, v) \leq c.dist$, among which $d(v, p) + d(c, v)$ and $d(v', p') + d(c, v')$ are the largest (Line 10).

Case 2: $d(v, c) > c.dist$ (Line 15). In this case, for each edge e'' adjacent to v, in the form of (v, v'') (Line 16), we check whether $d(v'', c) \leq c.dist$ (Line 17).

— 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 e'' (Line 18). If c is on edge e'' (Line 18), 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 19). 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 21).
— If no, then we check whether c is on the edge e'' (Line 24). 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 25). 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 27).

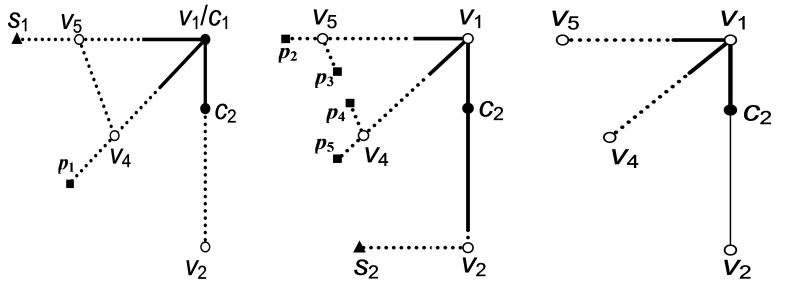
Finally, we return $\mathcal{I}$ as an output, representing $NLC(c)$ (Line 33). Note that if the intervals in $\mathcal{I}$ are overlapping, then they are merged into one interval that covers these intervals accordingly.

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.

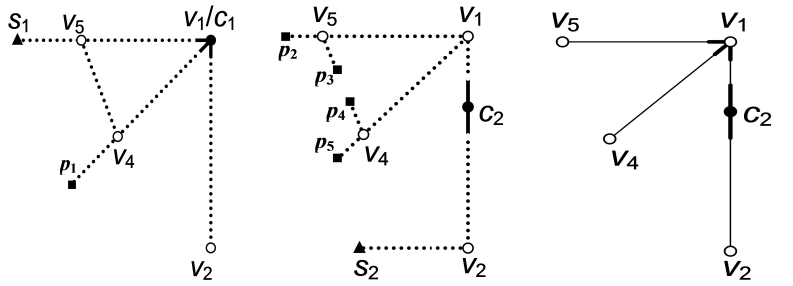
5. ALGORITHM MINMAX-ALG

In this section, we propose an algorithm called *MinMax-Alg* for the MinMax query. In Section 5.1, we introduce the basic concepts. In Section 5.2, we introduce the basic algorithm. In Section 5.3, we



(a) $NLC(c_1, d_1)$ (b) $NLC(c_2, d_2)$ (c) The $(2, Cost(c_3))$-critical intersection

Fig. 4. Example 2 Illustrating Critical Intersection



(a) $NLC(c_1, d_1)$ (b) $NLC(c_2, d_2)$ (c) The $(2, Cost(c_3))$-critical intersection

Fig. 5. Example 3 Illustrating Critical Intersection

introduce some enhancements over the basic algorithm. In Section 5.4, we present the pseudo-code for *MinMax-Alg*. In Section 5.5, we give the complexity analysis.

5.1. Basic Concepts

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". Intuitively, the "critical number" denotes the smallest possible number of clients to be checked such that we can find the optimal location for the MinMax query. 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 5.1 (*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 $p \in NLC(c)\}$.

Intuitively, the shrinking NLC of a client corresponds to a set of point intervals on the road network such that each point in the intervals has its distance to the client at most a specific value d.

Definition 5.2 (*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 $j \in [1, m]$.

Intuitively, "critical intersection" denotes a set of the intervals such that if a new server is set up at a point in the intervals, then the maximum cost value of a client is at most a certain value $\mathcal{C}$.

In the following, we use three examples to describe further the above concepts.

Example 5.3. Consider clients c_1, c_2 and c_3 in our running example as shown in Figure 1. Note that $c_1.dist = 6$, $c_2.dist = 7$, and $c_3.dist = 2$. Supposed that $w(c_1) = 7$, $w(c_2) = 4$ and $w(c_3) = 7$. Then, $Cost(c_1) = 42$, $Cost(c_2) = 28$ and $Cost(c_3) = 14$. Thus, we have $Cost(c_1) > Cost(c_2) > Cost(c_3)$.

Suppose that $m = 2$ and $\mathcal{C} = Cost(c_2)$. Consider client c_1. Let $d_1 = Cost(c_2)/w(c_1) = 4$. Note that $d_1 < c_1.dist$. In Figure 3(a), all the lines (including the dotted lines and the bold lines) denote $NLC(c_1)$. Only all the bold lines denote $NLC(c_1, d_1)$ which is a shrinking NLC of c_1 with respect to d_1. Consider client c_2. Let $d_2 = Cost(c_2)/w(c_2) = c_2.dist = 7$. Note that $d_2 = c_2.dist$. In Figure 3(b), $NLC(c_2, d_2)$ is a shrinking NLC of c_2 respect to d_2, which is identical to $NLC(c_2)$. Then, the $(2, Cost(c_2))$-critical intersection is equal to $NLC(c_1, d_1) \cap NLC(c_2, d_2)$ and corresponds to the bold lines in Figure 3(c).

Example 5.4. The clients and their weights are the same as Example 5.3.

Suppose that $m = 2$ and $\mathcal{C} = Cost(c_3)$. Consider client c_1. Let $d_1 = Cost(c_3)/w(c_1) = 2$. Note that $d_1 < c_1.dist$. In Figure 4(a), all the lines (including the dotted lines and the bold lines) denote $NLC(c_1)$. Only all the bold lines denote $NLC(c_1, d_1)$ which is a shrinking NLC of c_1 with respect to d_1. Consider client c_2. Let $d_2 = Cost(c_3)/w(c_2) = 3.5$. Note that $d_2 < c_2.dist$. In Figure 4(b), all the bold lines denote $NLC(c_2, d_2)$ which is a shrinking NLC of c_2 respect to d_2. Then, the $(2, Cost(c_3))$-critical intersection is equal to $NLC(c_1, d_1) \cap NLC(c_2, d_2)$ and corresponds to the bold lines in Figure 4(c).

Example 5.5. Similar to the above two examples, we consider clients c_1, c_2 and c_3. Suppose that $w(c_1) = 7$, $w(c_2) = 4$ and $w(c_3) = 1.4$. Thus, we have $Cost(c_1) = 42 > Cost(c_2) = 28 > Cost(c_3) = 2.8$.

Suppose that $m = 2$ and $\mathcal{C} = Cost(c_3)$. Consider client c_1. Let $d_1 = Cost(c_3)/w(c_1) = 0.4$. Note that $d_1 < c_1.dist$. In Figure 5(a), all the lines (including the dotted lines and the bold lines) denote $NLC(c_1)$. Only all the bold lines denote $NLC(c_1, d_1)$ which is a shrinking NLC of c_1 with respect to d_1. Consider client c_2. Let $d_2 = Cost(c_3)/w(c_2) = 0.7$. Note that $d_2 < c_2.dist$. In Figure 5(b), the bold lines denote $NLC(c_2, d_2)$ which is a shrinking NLC of c_2 respect to d_2. Then, the $(2, Cost(c_3))$-critical intersection is equal to $NLC(c_1, d_1) \cap NLC(c_2, d_2) = \emptyset$. Thus, in Figure 5(c), the bold lines denote both $NLC(c_1, d_1)$ and $NLC(c_2, d_2)$ do not share any common part.

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

Definition 5.6 (*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.

Intuitively, the "critical number" denotes the smallest possible number of clients to be checked such that we can find the optimal location for our problem.

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 and third examples. Similarly, we deduce that the critical number m_o for both examples is equal to 2.

5.2. Basic Algorithm

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 5.7. *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 based on which we 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.

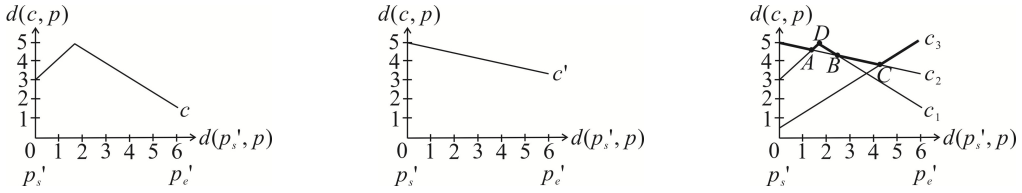
Step 1 and Step 2 will be described in detail in Section 5.2.1 and Section 5.2.2, respectively.

5.2.1. 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 $m - 1$.

5.2.2. Step 2: How to Find Optimal Location in Critical Intersection. 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 $I = [p_s, p_e]$. Note that 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 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 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



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

Fig. 6. Piecewise Linear Function denoting $d(c, p)$ where c is a particular client and p is an arbitrary point

sub-intervals (of the point interval 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 $I' = [p'_s, p'_e]$ of I. Let p be an *arbitrary* point along 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 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 6(a) shows that when p changes (along $I' = [p'_s, p'_e]$ 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, Figure 6(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 I', we have a piecewise linear function on I'. For another client c' where the shrinking NLC of c' completely covers I', we have another piecewise linear function on I'. There may exist multiple clients whose shrinking NLCs completely cover I'. For instance, Figure 6(c) shows the piecewise linear functions on I' for 3 clients, c_1, c_2 and c_3, whose shrinking NLCs completely cover 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 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 6(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 [Hershberger 1989] 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 I' with the smallest cost. In Figure 6(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 I' of a given point interval. Given a point interval I, we can obtain the optimal location with its cost for each of the sub-intervals of I and find the one with the smallest cost as the optimal location for the point interval 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.

5.3. Further Enhancement

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

Consider the second example where $m_o = 2$. Figure 4 shows an example of $\mathcal{R}$ which is non-empty. Consider the third example where $m_o = 2$. Figure 5 shows another example of $\mathcal{R}$ which is empty.

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

LEMMA 5.9. *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 second 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 third 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

Algorithm 2 Algorithm MinMax-Alg

find $c.dist$ for each client c
sort all clients $c_1, c_2, ..., c_n$ in descending order of their cost values
// Step 1 (Finding m_o)
for $m = 1$ to n **do**
  $\mathcal{C} \leftarrow Cost(c_m)$
  for $i = 1$ to m **do**
    $d_i \leftarrow \mathcal{C}/w(c_i)$
  end for
  $\mathcal{I}_{new} \leftarrow \cap_{i=1}^{m} NLC(c_i, d_i)$
  if $\mathcal{I}_{new} = \emptyset$ **then**
    break;
  else
    $\mathcal{I} \leftarrow \mathcal{I}_{new}$
    $m_o \leftarrow m$
  end if
end for
// Step 2 (Checking $\mathcal{R}$)
if $m_o < n$ **then**
  $\mathcal{C}' \leftarrow Cost(c_{m_o+1})$
  for $i = 1$ to $m_o + 1$ **do**
    $d'_i \leftarrow \mathcal{C}'/w(c_i)$
  end for
  $\mathcal{R} \leftarrow \cap_{i=1}^{m_o} NLC(c_i, d'_i)$
  if $\mathcal{R} \neq \emptyset$ **then**
    return $\mathcal{R}$ (or any point in $\mathcal{R}$)
  else
    $p_o \leftarrow$ the location with the smallest cost in the intersection $\mathcal{I}$ (represented by a set of point intervals)
    return $\{p_o\}$
  end if
else
  $p_o \leftarrow$ the location with the smallest cost in the intersection $\mathcal{I}$ (represented by a set of point intervals)
  return $\{p_o\}$
end if

$(m_o, Cost(c_{m_o}))$-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.

5.4. Pseudo-Code

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

Example 5.10. *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 3-16), 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 9). Since $\mathcal{I}_{new}$ is non-empty, variable $\mathcal{I}$ is updated to $\mathcal{I}_{new}$ (Line 13) and variable m_o is updated to $m(= 1)$ (Line 14).

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 9). Since $\mathcal{I}_{new}$ is non-empty, variable $\mathcal{I}$ is updated to $\mathcal{I}_{new}$ (Line 13) and m_o is updated to $m(= 2)$ (Line 14).

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 9). Since $\mathcal{I}_{new} = \emptyset$, we break the iterative step in Step 1 (Line 11) 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 18), 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.

THEOREM 5.11. *MinMax-Alg returns the optimal solution for the MinMax query.*

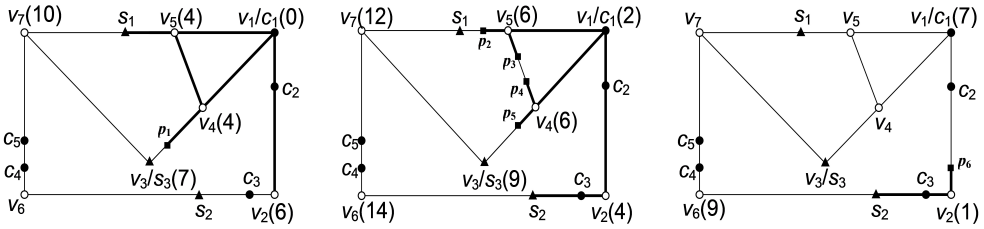
PROOF. It is easy to verify the correctness of MinMax-Alg with Lemma 5.7 and Lemma 5.9. □

5.5. Complexity Analysis

Firstly, we analyze the time complexity of finding the optimal location in the critical intersection (i.e., Step 2). Consider a sub-interval I' of a point interval I in Θ. Let $|C'|$ be the greatest number of (shrinking) NLCs covering a point interval I. Note that $|C'|$ is at most $|C|$ and in practice, it is significantly smaller than $|C|$. Besides, the sub-interval 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 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 $|E|$.

Secondly, we analyze the time complexity for further enhancement (in Section 5.3). 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 $O(\alpha + |\Theta| \cdot l_c \cdot |C'| \log |C'|)$.

Thirdly, we consider the time complexity of Algorithm 2. The major cost of the operations from Line 1 to Line 16 of this algorithm comes from computing $c.dist$ for all clients (Line 1), sorting (Line 2) and NLC building (Line 9). 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



(a) $NLC(c_1)$ (b) $NLC(c_2)$ (c) $NLC(c_3)$

Fig. 7. The NLCs in our running example

all clients examined takes $O(\gamma \cdot |V| \log |V|)$ time. The overall time complexity of the operations from Line 1 to Line 16 of this algorithm is $O(|C| \cdot l_s + |C| \log |C| + \gamma \cdot |V| \log |V|)$. As described previously, the operations from Line 17 to Line 33 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. ALGORITHM MAXSUM-ALG

In this section, we propose an algorithm called *MaxSum-Alg* for the MaxSum query. In Section 6.1, we introduce a basic algorithm. In Section 6.2, we develop some pruning techniques which can be used to enhance the basic algorithm. In Section 6.3, we introduce the enhanced algorithm and analyze its time and space complexities.

6.1. Basic Algorithm

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

Definition 6.1. Given a point p on G, the *influence value* of p, denoted by $Inf(p)$, is defined to be $\sum_{c \in C s.t. p \in NLC(c)} w(c)$.

Example 6.2. Consider our running example as shown in Figure 7. Figure 7(a), Figure 7(b) and Figure 7(c) show the road networks with $NLC(c_1)$, $NLC(c_2)$ and $NLC(c_3)$ (shown in bold), respectively. Consider a point p_1 in Figure 7(a) and a point p_3 in Figure 7(b). Since p_1 is only covered by $NLC(c_1)$ and p_3 is covered by $NLC(c_1)$ and $NLC(c_2)$, $Inf(p_1) = w(c_1) = 1$ and $Inf(p_2) = w(c_1) + w(c_2) = 2$.

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

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

LEMMA 6.3 (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 greatest influence value. Assume that all the clients attracted by p_o are $c_1, c_2, \ldots, c_m$. Note that they form C_{p_o}. Then, for each $i \in [1, m]$, $p_o \in NLC(c_i)$. Thus, $p_o \in \cap_{i=1}^{m} NLC(c_i)$. This means that any optimal location is in the intersection of these 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 6.3 gives a powerful tool to find an optimal solution. Specifically, according to Lemma 6.3, 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 point intervals with the greatest influence value. It is easy to verify the correctness of this algorithm.

6.2. Pruning Techniques

The basic algorithm can be enhanced with some pruning techniques. Next, we give the following lemma which can be used to prune some edges for processing.

LEMMA 6.4 (PRUNING). *Let e be an edge and $\mathcal{S}$ be the set of all clients whose NLCs cover e. Given a point p on e, we have $Inf(p) \leq \sum_{c \in \mathcal{S}} w(c)$.*

PROOF. Since e is only covered by the NLCs of clients in $\mathcal{S}$, for each of these NLCs, any point p on e is either covered by the NLC or not. Thus, $Inf(p) \leq \sum_{c \in \mathcal{S}} w(c)$ by Definition 6.1. The lemma holds. □

The above lemma suggests that the greatest possible influence value of a point along edge e is upper bounded by $\sum_{c \in \mathcal{S}} w(c)$ which is the *greatest possible influence* (GPI) of e and is denoted by $GPI(e)$.

Lemma 6.4 helps us to design a pruning strategy such that some edges can be pruned for processing. Suppose that we know that the current-best influence value found is IV (i.e., we found that there exists a point on an edge whose influence value is equal to IV). If the GPI of an edge e is smaller than IV, we can safely discard/prune this edge without computing the (exact) influence value of any point on this edge (because the influence value of any point on this edge is smaller than IV according to Lemma 6.4).

Example 6.5. *Consider Figure 7. Consider edge $e = (v_1, v_2)$. Note that e is covered by 3 NLCs, namely $NLC(c_1)$, $NLC(c_2)$ and $NLC(c_3)$. Thus, we have $GPI(e) = 3$. Similarly, we can compute the GPI values of all other edges. Then, we sort all edges in descending order of their GPI values and examine the point interval of NLCs on each edge. In Figure 7, e is completely covered by $NLC(c_1)$ and $NLC(c_2)$ and is partially covered by $NLC(c_3)$. Thus, the point interval of $NLC(c_1)$ on e is $[0, 6]$, the point interval of $NLC(c_2)$ on e is $[0, 6]$ and the point interval of $NLC(c_3)$ on e is $[5, 6]$. Suppose that we have processed edge e and have found that the influence value of a point in the point interval $[5, 6]$ on edge e is 3. Thus, the current best-known influence value is 3.*

Since the GPI of each of the other edges is smaller than 3, each of these edges can be pruned/discarded and need not be examined. This is because according to Lemma 6.4, the influence value of any point on each of these edges will be smaller than the greatest influence value 3.

As shown in the above Example 6.5, this pruning rule based on Lemma 6.4 requires to compute exactly $GPI(e)$ for each edge e, which is a little bit expensive. Specifically, a possible implemen-

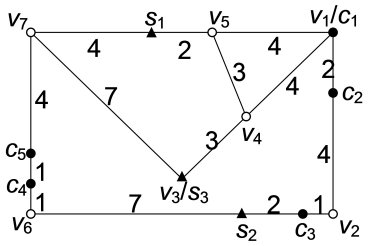


Fig. 8. The road network G in our running example

tation of computing exactly $GPI(e)$ is to examine the NLCs for all clients and determine which NLCs cover e. This implementation takes $O(|C| \cdot |V| \log |V|)$ time since it takes $O(|V| \log |V|)$ time to build the NLC of each client. In the following, we introduce a concept of "*effective clients*" and present an approach computing the upper bound of $GPI(e)$ with this concept which takes $O(\xi \cdot |V| \log |V|)$ time only where ξ is the total number of edges containing clients. Note that ξ is also usually smaller than $|C|$ since some edges contains more than one client. Thus, we improve the efficiency of the proposed algorithm by sacrificing the tightness of the upper bound of GPI.

In the following, we first define "effective clients" and then present a six-step implementation of *MaxSum-Alg* by using the concept of "effective clients".

Effective Client: Firstly, we split each edge containing servers not located at its end-points into a number of sub-edges such that each server is located at an end-point of a sub-edge. Here, in order to illustrate our concept of "effective client" clearly, we describe that we need to split edges into sub-edges *conceptually*. In our real implementation, we do not need to *physically* split edges into sub-edges. We will describe this implementation later in Section 6.3.

Let $\mathcal{E}$ be the set of all sub-edges and all non-split edges. For example, in Figure 8, edge (v_5, v_7) contains s_1 not located at its end-points. Thus, the edge is split into two sub-edges, namely (v_5, s_1) and (s_1, v_7).

Next, for each of edge/sub-edge $e = (v_l, v_r)$ in $\mathcal{E}$ containing clients, we define the *effective client* for e, denoted by EC_e, as follows. We consider two types of edges containing clients.

— The first type of edges are edges containing exactly one client. For each edge e containing only one client, we clone the client to generate a new client called the effective client for e, denoted by EC_e, with the same weight and the same location.
— The second type of edges are edges containing at least two clients. For each edge $e = (v_l, v_r)$ containing at least two clients, say $c_1, c_2, ..., c_m$ where $m \geq 2$, we create a new client called the effective client for e, denoted by EC_e, as follows: (1) the weight of EC_e, denoted by $w(EC_e)$, is set to $\sum_{i=1}^{m} w(c_i)$, and (2) EC_e is placed on edge e such that the distance between v_l and EC_e (denoted by $d(v_l, EC_e)$) is $(d(v_l, v_r)+v_r.dist-v_l.dist)/2$ (Note that $0 \leq d(v_l, EC_e) \leq d(v_l, v_r)$ (to be shown later), which means that EC_e must be on edge e).

Example 6.6. *Consider the road network in Figure 8 (which is the same as Figure 1(a)). Note that* $v_1.dist = 6$, $v_2.dist = 3$ *and* $d(v_1, v_2) = 6$. *Consider edge* $e = (v_1, v_2)$. *Since* e *contains no servers, we do not split* e. *Since* e *contains two clients, namely* c_1 *and* c_2, *the effective client* EC_e *is created where* $w(EC_e) = w(c_1)+w(c_2) = 2$ *and* $d(v_1, EC_e) = (d(v_1, v_2)+v_2.dist-v_1.dist)/2 = (6 + 3 - 6)/2 = 1.5$.

LEMMA 6.7. $0 \leq d(v_l, EC_e) \leq d(v_l, v_r)$

PROOF. There are two cases. *Case 1:* The edge e includes exactly one client. *Case 2:* The edge e includes at least two clients.

For Case 1, EC_e and client c (on edge e) have the same location. Since $0 \leq d(v_l, c) \leq d(v_l, v_r)$, then $0 \leq d(v_l, EC_e) \leq d(v_l, v_r)$.

Next, we discuss Case 2.

Since $v_r.dist \leq d(v_l, v_r) + v_l.dist$, we have $v_r.dist + d(v_l, v_r) - v_l.dist \leq 2 \cdot d(v_l, v_r)$. Since $d(v_l, EC_e) = (d(v_l, v_r) + v_r.dist - v_l.dist)/2$, we have $d(v_l, EC_e) \leq d(v_l, v_r)$.

Since $v_l.dist \leq d(v_l, v_r) + v_r.dist$, we have $d(v_l, v_r) + v_r.dist - v_l.dist \geq 0$. Then, $d(v_l, EC_e) \geq 0$. This lemma holds. □

Based on Lemma 6.7, we know that each effective client is on the edge $e = (v_l, v_r)$.

Note that given a client c, the shortest distance between c and its nearest server is denoted by $c.dist$. Here, we *overload* the notation of $dist$ for an effective client EC_e (i.e., $EC_e.dist$). If the exact value of $EC_e.dist$ is known, we can construct $NLC(EC_e)$ according to Definition 2.2. The remaining question is how to set the exact value of $EC_e.dist$.

Given an effective client EC_e for edge $e = (v_l, v_r)$, we set

$$EC_e.dist = \begin{cases} c.dist & \text{if } e \text{ contains only one client } c \\ (v_l.dist + d(v_l, v_r) + v_r.dist)/2 & \text{if } e \text{ contains multiple clients} \end{cases} \quad (1)$$

Example 6.8. *Consider the road network in Figure 8. Consider edge $e = (v_1, v_2)$ containing two clients, namely c_1 and c_2. We derive that $EC_e.dist = (v_1.dist + d(v_1, v_2) + v_2.dist)/2 = (6 + 6 + 3)/2 = 7.5$.*

Next, we show that after we set $EC_e.dist$ to this value according to (1), for each client c on e, $NLC(EC_e)$ covers $NLC(c)$.

LEMMA 6.9. *Given an edge $e = (v_l, v_r) \in \mathcal{E}$, for each client c on e, $NLC(EC_e)$ covers $NLC(c)$.*

PROOF. Since c is on edge e and there are no servers on edge e, we have $c.dist = \min\{d(v_l, c) + v_l.dist, d(v_r, c) + v_r.dist\} \leq (d(v_l, c) + v_l.dist + d(v_r, c) + v_r.dist)/2 = (v_l.dist + d(v_l, v_r) + v_r.dist)/2 = EC_e.dist$.

Next, we only need to show that for any point $p \in NLC(c)$, $p \in NLC(EC_e)$.

Consider two cases. *Case A:* There is exactly one client on edge e. *Case B:* There are at least two clients on edge e.

Consider *Case A*. There is exactly one client c on edge $e = (v_l, v_r)$. According to (1), we have $EC_e.dist = c.dist$. Since EC_e and c have the same location on e, we deduce that $NLC(EC_e)$ covers $NLC(c)$.

Consider *Case B*. There are at least two clients on edge e. Firstly, we want to show that edge $e = (v_l, v_r)$ is completely covered by EC_e. In order to show this, we want to show that $d(v_l, EC_e) \leq EC_e.dist$ and $d(v_r, EC_e) \leq EC_e.dist$. By Lemma 6.7, EC_e is on edge e. We have $d(v_l, EC_e) = (v_r.dist + d(v_l, v_r) - v_l.dist)/2 \leq (v_r.dist + d(v_l, v_r) + v_l.dist)/2 = EC_e.dist$. Thus, $d(v_l, EC_e \leq EC_e.dist$. Besides, $d(v_r, EC_e) = d(v_l, v_r) - d(v_l, EC_e) = d(v_l, v_r) - (v_r.dist + d(v_l, v_r) - v_l.dist)/2 = (-v_r.dist + d(v_l, v_r) + v_l.dist)/2 \leq (v_r.dist + d(v_l, v_r) + v_l.dist)/2 = EC_e.dist$. Thus, $d(v_r, EC_e) \leq EC_e.dist$. In conclusion, edge e is completely covered by $NLC(EC_e)$.

Secondly, we want to show that for any point $p \in NLC(c)$, $p \in NLC(EC_e)$. Note that e is completely covered by $NLC(EC_e)$. If p is on edge e, then $p \in NLC(EC_e)$. Thus, in the following, we assume that p is not on edge e. Then, we want to show that $p \in NLC(EC_e)$. Consider two cases. *Case 1:* c is in the point interval $[v_l, EC_e]$. *Case 2:* c is in the point interval $[EC_e, v_r]$.

For Case 1, since c is in the point interval $[v_l, EC_e]$, we have $d(v_l, c) \leq d(v_l, EC_e)$. Then, we have $d(v_l, c) \leq (d(v_l, v_r) + v_r.dist - v_l.dist)/2$. Then, we derive that $d(v_l, c) + v_l.dist \leq d(v_r, c) + v_r.dist$. Since $c.dist = \min\{d(v_l, c) + v_l.dist, d(v_r, c) + v_r.dist\}$, we have $c.dist = d(v_l, c) + v_l.dist$.

It is easy to know that $d(c, p) = \min\{d(v_l, c) + d(v_l, p), d(v_r, c) + d(v_r, p)\}$.

There are two cases: *Case a:* $d(v_l, c)+d(v_l, p) \leq d(v_r, c)+d(v_r, p)$. *Case b:* $d(v_l, c)+d(v_l, p) > d(v_r, c) + d(v_r, p)$.

For Case a, we have $d(c, p) = d(v_l, c) + d(v_l, p)$. Since $p \in NLC(c), d(c, p) \leq c.dist$. Then, $d(v_l, c) + d(v_l, p) \leq d(v_l, c) + v_l.dist$. Thus, $d(v_l, p) \leq v_l.dist$.

Since $d(EC_e, p) \leq d(v_l, EC_e) + d(v_l, p)$, we derive that $d(EC_e, p) \leq d(v_l, EC_e) + v_l.dist = (v_r.dist + d(v_l, v_r) - v_l.dist)/2 + v_l.dist = (v_r.dist + d(v_l, v_r) + v_l.dist)/2 = EC_e.dist$. Thus, $p \in NLC(EC_e)$.

For Case b, we have $d(c, p) = d(v_r, c) + d(v_r, p)$. Since c is in the point interval $[v_l, EC_e]$, $d(v_r, EC_e) \leq d(v_r, c)$. Thus, since $d(EC_e, p) \leq d(v_r, EC_e) + d(v_r, p)$, we have $d(EC_e, p) \leq d(v_r, c) + d(v_r, p) = d(c, p) \leq c.dist \leq EC_e.dist$. Thus, $p \in NLC(EC_e)$.

For Case 2, similarly, we know that $p \in NLC(EC_e)$.

In conclusion, for any point $p \in NLC(c)$, $p \in NLC(EC_e)$. This lemma holds. □

LEMMA 6.10 (PRUNING). *Let e be an edge in $\mathcal{E}$ and $\mathcal{S}'$ be the set of all effective clients whose NLCs cover e. For each point p on e, we have $Inf(p) \leq \sum_{c' \in \mathcal{S}'} w(c')$.*

PROOF. Let $\mathcal{S}$ be the set of all clients whose NLCs cover e. Given an edge e' in $\mathcal{E}$, we denote $C_{e'}$ to be the set of all the clients on the edge e'.

Then, it is easy to know that $\mathcal{S} = \cup_{e' \in \mathcal{E}}(\mathcal{S} \cap C_{e'})$.

For any edge e', if $\mathcal{S} \cap C_{e'} \neq \emptyset$, then there exist a client c on the edge e' such that the NLC of c covers edge e. Then, according to Lemma 6.9, since c is on the edge e' and $NLC(c)$ covers e, $NLC(EC_{e'})$ covers e. Thus, we have $EC_{e'} \in \mathcal{S}'$.

Next, we have the following derivation. For each point p on e, we have $Inf(p) = \sum_{c \in \mathcal{S}} w(c) = \sum_{e' \in \mathcal{E}} \sum_{c \in \mathcal{S} \cap C_{e'}} w(c) = \sum_{e' \in \mathcal{E} \wedge \mathcal{S} \cap C_{e'} \neq \emptyset} \sum_{c \in \mathcal{S} \cap C_{e'}} w(c) \leq \sum_{e' \in \mathcal{E} \wedge \mathcal{S} \cap C_{e'} \neq \emptyset} w(EC_{e'}) = \sum_{e' \in \mathcal{E} \wedge \mathcal{S} \cap C_{e'} \neq \emptyset \wedge EC_{e'} \in \mathcal{S}'} w(EC_{e'}) \leq \sum_{c' \in \mathcal{S}'} w(c')$. This lemma holds. □

Given an edge e, the *greatest possible effective influence* of e, denoted by $GPEI(e)$, is defined to be $\sum_{c' \in \mathcal{S}'} w(c')$.

LEMMA 6.11. $GPI(e) \leq GPEI(e)$.

PROOF. We adopt the notations used in the proof of Lemma 6.10.

From the proof of Lemma 6.10, we know that $GPI(e) = \sum_{c \in \mathcal{S}} w(c) = \sum_{e' \in \mathcal{E}} \sum_{c \in \mathcal{S} \cap C_{e'}} w(c) \leq \sum_{e' \in \mathcal{E} \wedge \mathcal{S} \cap C_{e'} \neq \emptyset} w(EC_{e'})$.

Let $E_{\mathcal{S}'}$ be the set of all edges containing the effective clients in $\mathcal{S}'$.

For each edge $e' \in \mathcal{E}$ such that $\mathcal{S} \cap C_{e'} \neq \emptyset$, there exists at least one client c on edge e' (which is in $\mathcal{S} \cap C_{e'}$) such that the NLC of c covers edge e. By Lemma 6.9, $NLC(EC_{e'})$ covers $NLC(c)$. Then, $NLC(EC_{e'})$ also covers edge e. Since the NLC of $EC_{e'}$ covers edge e, $EC_{e'} \in \mathcal{S}'$. Note that e' is an edge containing the effective client $EC_{e'}$. Since $EC_{e'} \in \mathcal{S}'$, we have $e' \in E_{\mathcal{S}'}$ and $\sum_{e' \in \mathcal{E} \wedge \mathcal{S} \cap C_{e'} \neq \emptyset} w(EC_{e'}) \leq \sum_{e' \in E_{\mathcal{S}'}} w(EC_{e'}) = GPEI(e)$. Thus, $GPI(e) \leq GPEI(e)$. This lemma holds. □

Thus, $GPEI(e)$ is regarded as an upper bound on $GPI(e)$.

6.3. Enhanced Algorithm

We give a six-step implementation for our enhanced algorithm as follows.

Six-Step Implementation: Specifically, this implementation involves six steps. The first step is to generate $\mathcal{E}$ by splitting each edge containing servers not located at its end-points into a number of sub-edges *conceptually*. In the later part of this section, we will describe how we implement this step without splitting edges into sub-edges in our real implementation.

The second step is to create the effective client for each of edge/sub-edges in $\mathcal{E}$.

The third step is to construct the NLC of each effective client (by calling the algorithm described in Section 4.3). Note that ξ is the total number of edges containing clients. It is also equal to the

Algorithm 3 Algorithm MaxSum-Alg

```
1: IV_o ← 0; I ← ∅ //IV_o is the influence value
2: // Step 1
3: E ← a set of sub-edges, generated from edges split at servers not located at end-points of these
   edges, and non-split edges
4: // Step 2, Step 3 and Step 4
5: for each edge e in E do
6:    construct EC_e and NLC(EC_e)
7:    for each edge e' covered by NLC(EC_e) do
8:       GPEI(e') ← GPEI(e') + w(EC_e)
9:    end for
10: end for
11: // Step 5
12: sort all edges in E in descending order of their GPEI values
13: // Step 6
14: for each edge e in E to be processed in the sorted ordering do
15:    if GPEI(e) < IV_o then
16:       break;
17:    else
18:       find the point interval I on e which has the greatest influence value IV
19:       if IV > IV_o then
20:          IV_o ← IV; I ← {I}
21:       else
22:          if IV = IV_o then
23:             I ← I ∪ {I}
24:          end if
25:       end if
26:    end if
27: end for
28: return I
```

total number of effective clients. Since there are ξ effective clients, constructing the NLC of each effective client takes $O(\xi \cdot |V| \log |V|)$ time.

The fourth step is to compute $GPEI(e)$ for each edge e.

The fifth step is to sort the edges in $\mathcal{E}$ in descending order of their $GPEI$ values.

The sixth step is an iterative process. Let IV_o be a variable, initialized with 0, denoting the best-known influence value found so far, and $\mathcal{I}$ be a variable, initialized with $\emptyset$, denoting the set of point intervals with the corresponding influence value found so far. We process each edge e in $\mathcal{E}$ in the sorted order. If $GPEI(e) \geq IV_o$, then we find the point interval I on e which has the greatest influence value IV. If IV is greater than IV_o, then we update IV_o with IV and $\mathcal{I}$ with I. If IV is equal to IV_o, then we insert I into $\mathcal{I}$.

Algorithm 3 shows the pseudo-code of our proposed algorithm *MaxSum-Alg*.

THEOREM 6.12. *MaxSum-Alg returns the optimal solution for the MaxSum query.*

PROOF. With the help of Lemmas 6.3, 6.4, 6.9, 6.10 and 6.11, it is easy to verify the correctness of MaxSum-Alg. □

Implementation Details: As described before, we do not need to split edges into sub-edges to form $\mathcal{E}$. Note that after we perform the edge splitting operation, we construct a set $\mathcal{E}$ of all sub-edges and all non-split edges, and we just process each edge in $\mathcal{E}$ for processing. This is the only place in our method using $\mathcal{E}$. In our real implementation, we do not split each edge in the road network

containing servers not located at its end-points *physically*. This is because when we *conceptually* process each sub-edge in $\mathcal{E}$, we need to scan each server along the original (non-split) edge. Thus, we do not need to split edges into sub-edges and do not generate any new road network.

Time and Space Complexities: Consider Algorithm 3. Lines 1-10 takes $O(\xi \cdot |V| \log |V|)$ time. Line 12 takes $O(|\mathcal{E}| \log |\mathcal{E}|)$ time. Each iteration (Lines 15-26) takes $O(|V| \log |V|)$ time. Let ϵ be the total number of edges processed in the algorithm iteration. Thus, Lines 14-27 takes $O(\epsilon \cdot |V| \log |V|)$ time. The overall time complexity of Algorithm 3 is $O(\xi \cdot |V| \log |V| + |\mathcal{E}| \log |\mathcal{E}| + \epsilon \cdot |V| \log |V|)$, where ξ is at most $|C|$, $|\mathcal{E}|$ is at most $|E| + |S|$, and ϵ is at most $|\mathcal{E}|$. Note that ξ and ϵ are usually smaller than $|C|$ and $|\mathcal{E}|$ (or $|E| + |S|$) in practice. In our experiments with the default setting on the *SF* real dataset, where $|C|$ is 300k and $|E|$ is 223k, ξ is 63k and ϵ is 12, respectively. Note that the time complexity of the best-known algorithm for the MaxSum query is $O((|V| + |S| + |C|)^2 \log(|V| + |S| + |C|))$ in the worst case.

Since the space consumption of Algorithm 3 includes the storage cost of the Dijkstra's algorithm (i.e., $O(|V|)$) and the storage cost of $c.dist$ for each client c (i.e., $O(|C|)$), the space complexity of Algorithm 3 is $O(|V| + |C|)$.

7. EXTENSIONS

In this section, we discuss the extensions to the optimal location query problem. In Section 7.1, we study finding multiple locations (instead of a single location) for the optimal location query. In Section 7.2, we discuss the optimal location query on a three-dimensional (3D) road network.

7.1. Optimal Multiple-Location Query

In Section 7.1.1, we describe how to find multiple locations for the optimal location query when the cost is MinMax. In Section 7.1.2, we also describe the case when the cost is MaxSum.

7.1.1. Multiple-Location MinMax Query. The existing MinMax query usually finds an optimal location in the road network to set up a *single* server. In practice, we sometimes want to set up *multiple* servers. Specifically, given a positive integer $k \geq 2$, 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 for this problem.

THEOREM 7.1. *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

Algorithm 4 Algorithm GA(MinMax)

1: $\mathcal{IS} \leftarrow \emptyset$
2: **for** $i = 1$ to k **do**
3: execute *MinMax-Alg* (i.e., Algorithm 2) based on the current set S of servers for finding an optimal location for a new server $s(p_o)$
4: insert the new server into S, namely $S \leftarrow S \cup \{s(p_o)\}$
5: $\mathcal{IS} \leftarrow \mathcal{IS} \cup \{(i, s(p_o))\}$
6: **end for**
7: **return** $\mathcal{IS}$

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 1.

Since H is equal to 1 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 called *GA(MinMax)*, 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 locations for the new servers are found. The detailed description for *GA(MinMax)* is shown in Algorithm 4.

7.1.2. Multiple-Location MaxSum Query. Similar to the multiple-location MinMax query, in practice, we sometimes want to find multiple locations and set up a server at each of these locations for the MaxSum query. Specifically, given a positive integer $k \geq 2$, we want to find k locations, namely p_1, p_2, p_3, ..., p_k, in the road network such that $\sum_{c \in C} w(c) \cdot (d(c, NN_S(c)) \geq d(c, NN_{S \cup \{s(p_1), s(p_2), ..., s(p_k)\}}(c))))$ is maximized where $(\cdot \geq \cdot)$ returns 1 if it is true and 0 otherwise. We call this problem the *multiple-location MaxSum query* problem. Note that the new servers built at the k locations should have their collaborative relationship instead of the competitive relationship. Intuitively, these new servers collaborate to attract as many clients as possible in our 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 called *GA(MaxSum)* for this problem.

THEOREM 7.2. *The multiple-location MaxSum query problem is NP-hard.*

PROOF. The proof of this theorem is similar to the proof of Theorem 7.1. Firstly, we give a decision version of our problem as follows. Given a set Y of points, we define $W(Y)$ to be $\sum_{c \in C} w(c) \cdot (d(c, NN_S(c)) \geq d(c, NN_{S \cup \{s(p_1), s(p_2), ..., s(p_k)\}}(c))))$.

Multiple-Location MaxSum Query/Problem (MLMSQ): 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 $W(\{p_1, p_2, ..., p_k\}) \geq H$?

Similar to the proof of Theorem 7.1, we can still transform problem XC3S to our problem MLMSQ except that H is set to $3q$ instead of 1. It is easy to verify that the k positions, namely $p_1, p_2, ..., p_k$ must be at the vertices in $\mathcal{A}$ so that $W(\{p_1, p_2, ..., p_k\})$ is at least 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 transformed MLMSQ, the original XC3S problem is also solved. Since XC3S is an NP-complete problem, the MLMSQ problem is NP-hard. □

The detailed description for *GA(MaxSum)* is shown in Algorithm 5 which is similar to *MaxSum-Alg* (i.e., Algorithm 3). The key differences between these two algorithms are described as follows.

— Firstly, *GA(MaxSum)* is a greedy algorithm and includes an iterative process (i.e., Lines 11-42) involving k iterations for finding k locations.
— Secondly, *GA(MaxSum)* includes an additional step for updating the upper bounds of the edges in the iterative process (i.e., Lines 14-21) so that the clients attracted by the servers found in the previous iterations should not be re-considered when we set up the other servers to be found in the later iterations. Specifically, in each iteration, these steps similar to *MaxSum-Alg* are executed to find the set $\mathcal{I}$ of point intervals with the greatest influence value for a new server. The clients attracted by a new server to be set up in $\mathcal{I}$ are stored in a variable ACS, initialized to $\emptyset$. If ACS is not equal to $\emptyset$, then there are clients attracted by a new server found in the previous iteration. Then, for each edge e, we check if there exists a client c in ACS which is on e. If so, for each edge e' covered by $NLC(EC_e)$, we need to update the upper bound of the edge e' (i.e., $GPEI(e')$) by removing the weighted sum of all clients on e in ACS from $GPEI(e')$. This is because the new servers built at different locations found in the previous iterations have their collaborative relationship, and if a client is attracted by one server found in the previous iteration, then we need not attract this client for another server to be found.

7.2. Optimal Location Query on 3D Road Networks

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 [Kaul et al. 2013] 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 (MaxSum) query algorithm (i.e., Algorithm 2 (Algorithm 3)) originally designed for the original optimal location query on the 2D road network and the greedy algorithm (i.e., Algorithm 4 (Algorithm 5)) originally designed for the multiple-location MinMax (MaxSum) query problem on the 2D road network can be easily extended to handling the original optimal location query and the new multiple-location MinMax (MaxSum) query problem on the 3D road network.

Only the distance computation needs to be adapted for this extension. Specifically, the distance between any two end-points of an edge on the graph representing the 3D road network can be computed by their three-dimensional coordinates (i.e., the longitude, latitude and height of the locations). This computation can be done by [Kaul et al. 2013].

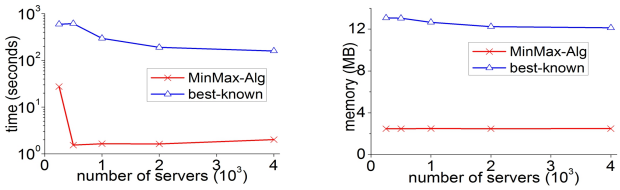
8. EMPIRICAL STUDIES

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/). We believe that the authors of the best-known algorithm tried their best to optimize their own source codes and thus our experimental comparison is fair. 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 three parts. The first part is about the experiments for the MinMax query (Section 8.1). The sec-

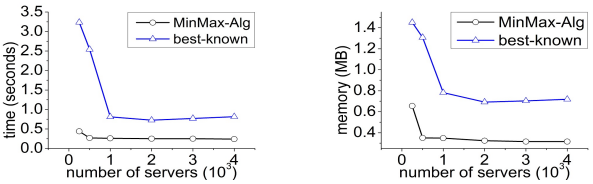
Algorithm 5 Algorithm GA(MaxSum)

$ACS \leftarrow \emptyset; \mathcal{IS} \leftarrow \emptyset$
// Step 1
$\mathcal{E} \leftarrow$ a set of all sub-edges, generated from edges split at servers not located at end-points of these edges, and all non-split edges
// Step 2, Step 3 and Step 4
for each edge e in $\mathcal{E}$ **do**
  construct EC_e and $NLC(EC_e)$
  for each edge e' covered by $NLC(EC_e)$ **do**
    $GPEI(e') \leftarrow GPEI(e') + w(EC_e)$
  end for
end for
for $i = 1$ to k **do**
  $IV_o \leftarrow 0; \mathcal{I} \leftarrow \emptyset$ //IV_o is the influence value
  if $ACS \neq \emptyset$ **then**
    for each edge e in $\mathcal{E}$ **do**
      $CS \leftarrow \{c | c \in ACS$ and c is on $e\}$
      if $CS \neq \emptyset$ **then**
        for each edge e' covered by $NLC(EC_e)$ **do**
          $GPEI(e') \leftarrow GPEI(e') - \sum_{c \in CS} w(c)$
        end for
      end if
    end for
  end if
  // Step 5
  sort all edges in $\mathcal{E}$ in descending order of their $GPEI$ values
  // Step 6
  for each edge e in $\mathcal{E}$ to be processed in the sorted ordering **do**
    if $GPEI(e) < IV_o$ **then**
      break;
    else
      find the point interval I on e which has the greatest influence value IV
      if $IV > IV_o$ **then**
        $IV_o \leftarrow IV; \mathcal{I} \leftarrow \{I\}$
      else
        if $IV = IV_o$ **then**
          $\mathcal{I} \leftarrow \mathcal{I} \cup \{I\}$
        end if
      end if
    end if
  end for
  $ACS \leftarrow$ a set of all clients attracted by a new server to be set up in $\mathcal{I}$
  $\mathcal{IS} \leftarrow \mathcal{IS} \cup \{(i, \mathcal{I})\}$
end for
return $\mathcal{IS}$

ond part is about the experiments for the MaxSum query (Section 8.2). The third part is about the experiments for the extension to the MinMax query and the MaxSum query (Section 8.3).

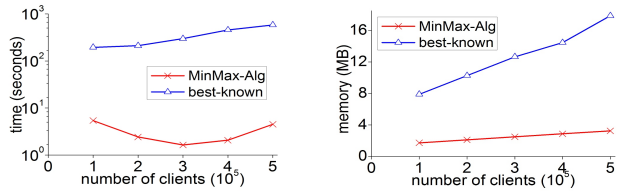


(a) time on SF (b) storage on SF

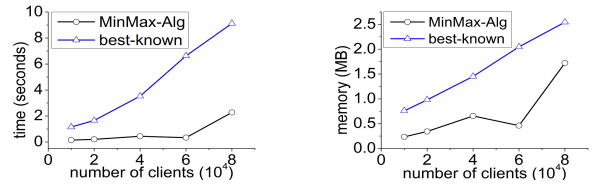


(c) time on CA (d) storage on CA

Fig. 9. Effect of $|S|$ for the MinMax query

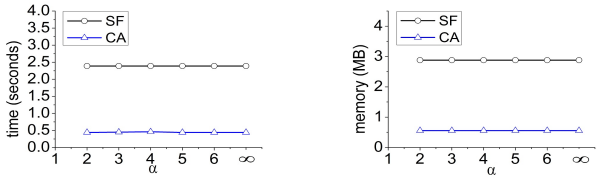


(a) time on SF (b) storage on SF



(c) time on CA (d) storage on CA

Fig. 10. Effect of $|C|$ for the MinMax query



(a) time (b) storage

Fig. 11. Effect of α for the MinMax query

8.1. Experiments for the MinMax Query

In this section, we compared our method, *MinMax-Alg*, with the best-known algorithm [Xiao et al. 2011]. Similar to [Xiao et al. 2011], two real road network datasets, SF and CA, were used in the

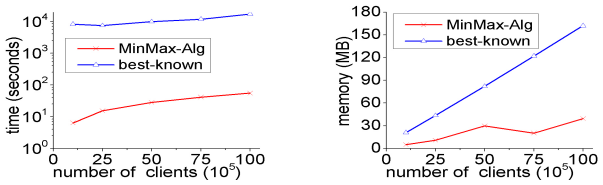
experiments. SF and CA are real datasets for the road networks in San Francisco and California, respectively. SF contains 174,955 vertices and 223,000 edges, and CA includes 21,047 vertices and 21,692 edges. In the real road network datasets, the max/min/avg number of edges adjacent to a vertex is equal 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 [Xiao et al. 2011]. 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 [Xiao et al. 2011] 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 [Xiao et al. 2011], the best-known algorithm has the *best* performance when θ is set to 1‰ which is also the default setting here.

In the experiments, we study the effects of $|S|$, $|C|$ and α. We also study the performance of algorithms on large datasets.

Effect of $|S|$*:* We study the effect of $|S|$ on the *SF* dataset in Figure 9(a) and Figure 9(b). In Figure 9(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. The best-known method has to generate additional vertices for clients and servers in the road network to form an augmented road network, which increases its query time a lot (as described in Section 3.2). 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 9(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 9(c) and Figure 9(d) 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 9(d)) 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 large search space.

Effect of $|C|$*:* We study the effect of $|C|$ on the *SF* dataset in Figure 10(a) and Figure 10(b). As shown in Figure 10(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. Figure 10(b) shows that the memory consumption of MinMax-Alg and the best-known method increases with $|C|$. The results on the *CA* dataset can be found in Figure 10(c) and Figure 10(d).



(a) time (b) storage

Fig. 12. The MinMax query on larger datasets

Effect of α: The experimental results about the effect of α can be found in Figure 11(a) and Figure 11(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.

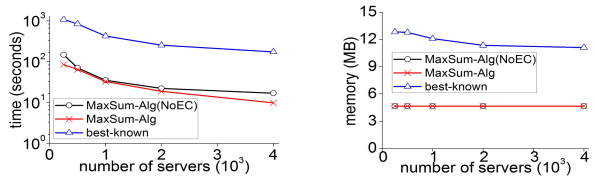
Results on Large Datasets: We observed that the total number of all real buildings in San Francisco in the $OpenStreetMap$ project is more than 10 million. Thus, we also conducted experiments on massive datasets for the real road networks in San Francisco. Specifically, we study the MinMax query performance by varying the number of clients. In the experiments, the number of servers is set to be the default value. The number of clients is varied from 1 million to 10 million. We compare our *MinMax-Alg* with the best-known algorithm in terms of algorithm performance. The results are shown in Figure 12(a) and Figure 12(b). As shown in Figure 12(a), the best-known algorithm performance is bad due to its poor pruning effect. The running time of the best-known algorithm is about 5 hours when the number of clients is equal to 10 million. However, our algorithm is still efficient (within 100s) in this case. In Figure 12(b), the memory consumption of the best-known algorithm is extremely larger than that of our algorithm.

Moreover, we test the running time for the MinMax query with the extreme case, where the number of server (client) is equal to 250 (10 million). In this case, the small number of servers dramatically slows down the best-known algorithm. 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. This is because the algorithm search space for the best-known algorithm becomes larger with the small number of servers.

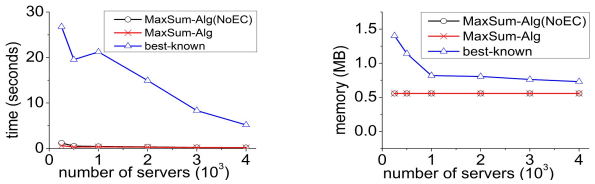
8.2. Experiments for the MaxSum Query

In this section, we study the performance of algorithms for the MaxSum query on the SF dataset and the CA dataset. We compared our *MaxSum-Alg* with the best-known algorithm. The experimental results of the effect of $|S|$ are shown in Figure 13(a) and Figure 13(b). In these figures, there is a notation called "MaxSum-Alg(NoEC)". "MaxSum-Alg(NoEC)" is exactly the same as "MaxSum-Alg" except that in "MaxSum-Alg(NoEC)", we remove the component of "effective client" used in "MaxSum-Alg" such that the pruning mechanism in "MaxSum-Alg(NoEC)" is based on GPI instead of $GPEI$ which relies on "effective client". In the following, we compare *MaxSum-Alg* (using "effective clients" to prune edges based on $GPEI(e)$) with another algorithm called *MaxSum-Alg(NoEC)* which is exactly *MaxSum-Alg* without using "effective clients" and prunes edges based on $GPI(e)$ (instead of $GPEI(e)$).

When the number of servers decreases, for each client c, $c.dist$ will be larger and the time for building $NLC(c)$ will also be larger. Thus, the times of all algorithms become larger. The running time of *MaxSum-Alg* is significantly smaller than that of the best-known algorithm. As shown in our experiments, a lot of edges are pruned by the pruning technique in *MaxSum-Alg* based on $GPEI$. Only 0.025% edges are processed in the algorithm. Besides, in our default setting, *MaxSum-Alg* performs 4.5916 times faster than *MaxSum-Alg(NoEC)*. This is because *MaxSum-Alg* takes less time for building the NLCs based on $GPEI(e)$ compared with *MaxSum-Alg(NoEC)* based on $GPI(e)$. Figure 13(b) shows the memory consumption of *MaxSum-Alg* is not sensitive to the increase in the number of servers. This is because the memory consumption of *MaxSum-Alg* (which is $O(|V| +$

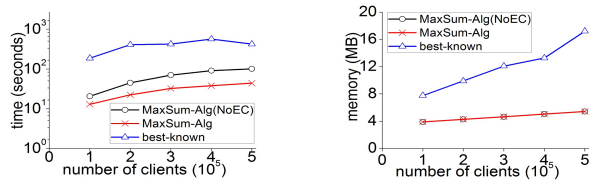


(a) time on SF (b) storage on SF

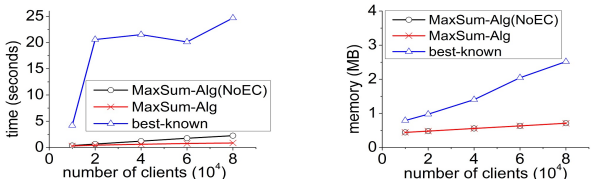


(c) time on CA (d) storage on CA

Fig. 13. Effect of $|S|$ for the MaxSum query



(a) time on SF (b) storage on SF

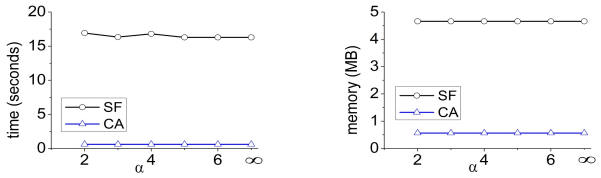


(c) time on CA (d) storage on CA

Fig. 14. Effect of $|C|$ for the MaxSum query

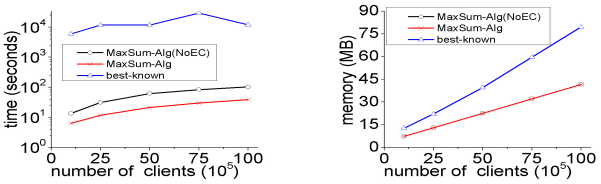
$|C|$)) is not related to the number of servers. In the figure, it is smaller than that of the best-known algorithm. *MaxSum-Alg* and *MaxSum-Alg(NoEC)* have similar memory consumption. The results on the CA dataset are shown in Figure 13(c) and Figure 13(d), which are similar to those on the SF dataset.

The experimental results about the effect of $|C|$ are shown in Figure 14(a) and Figure 14(b). When the number of clients increases, the number of NLCs increases. The time for building all NLCs is also larger. Thus, the running time of each algorithm is larger. The running time of *MaxSum-Alg* is significantly smaller than that of the best-known algorithm. Figure 14(b) shows the memory consumption of *MaxSum-Alg* is smaller than that of the best-known algorithm. This is because the number of edges to be examined in *MaxSum-Alg* is small since a lot of edges can be pruned. The results on the CA dataset are shown in Figure 14(c) and Figure 14(d), which are similar to those on the SF dataset.



(a) time (b) storage

Fig. 15. Effect of α for the MaxSum query



(a) time (b) storage

Fig. 16. The MaxSum query on larger datasets

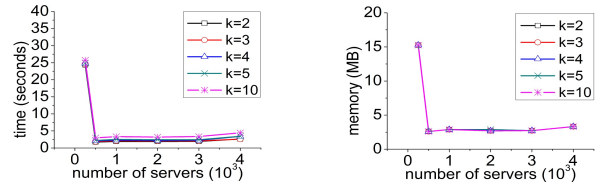
Effect of α: The experimental results about the effect of α can be found in Figure 15. These two figures show that the time and the memory consumption of *MaxSum-Alg* for the *SF* dataset and the *CA* dataset are not sensitive to α.

Results on Large Datasets: We also compared our *MaxSum-Alg* with the best-known algorithm on massive datasets for the real road networks in San Francisco in terms of algorithm performance. The results are shown in Figure 16(a) and Figure 16(b). As shown in the results, the best-known algorithm performance is much worse due to its poor pruning effect. However, our algorithm is still efficient. We also tested the running time for the MaxSum query with the extreme case when the number of servers is very small and the number of clients is very large. In these experiments, the number of servers (clients) is equal to 250 (10 million). the best-known algorithm is very slow in this extreme case as shown in Figure 16(a). The query time of the best-known algorithm is more than 12 hours but the query time of our algorithm *MaxSum-Alg* is within 2 minutes. Similarly, Figure 16(b) shows that the memory consumption of the best-known algorithm is extremely larger than that of our algorithm *MaxSum-Alg*.

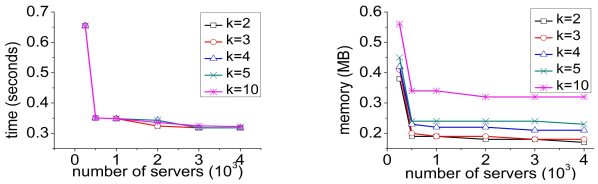
8.3. Experiments for the Extensions

In this section, we give some experimental results about the extensions described in Section 7. Specifically, the experimental result on the multiple-location MinMax query is reported in Section 8.3.1. The experimental result on the multiple-location MaxSum query is reported in Section 8.3.2. The experimental result on the MinMax query on 3D road networks is reported in Section 8.3.3. The experimental result on the MaxSum query on 3D road networks is reported in Section 8.3.4.

8.3.1. Multiple-Location MinMax Query. In this section, we study the performance of the *GA(MinMax)* algorithm for the optimal multiple-location query. The experimental results on the SF dataset and the CA dataset are reported. The experimental results about the effect of $|S|$ on the *SF* dataset with different values of k can be found in Figure 17. In Figure 17(a), similar to the original MinMax query, when $|S|$ is small, the time of *GA(MinMax)* is large. When $|S|$ becomes larger, the time of *GA(MinMax)* increases slightly. Besides, when k is larger, the time of *GA(MinMax)* increases slightly. The reason is explained as follows. *GA(MinMax)* includes two major phases. The first phase is to build the NLCs which are used to find the critical intersection. The second phase is to iteratively find k servers maximizing a certain cost (i.e., the maximum cost of a client attracted by

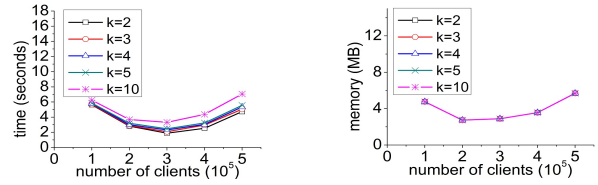


(a) time on SF (b) storage on SF

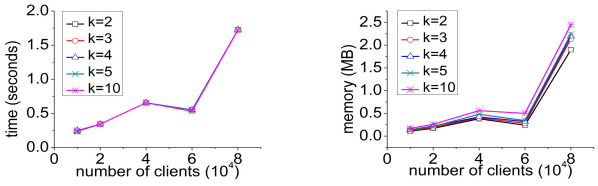


(c) time on CA (d) storage on CA

Fig. 17. Effect of $|S|$ for the multiple-location MinMax query



(a) time on SF (b) storage on SF

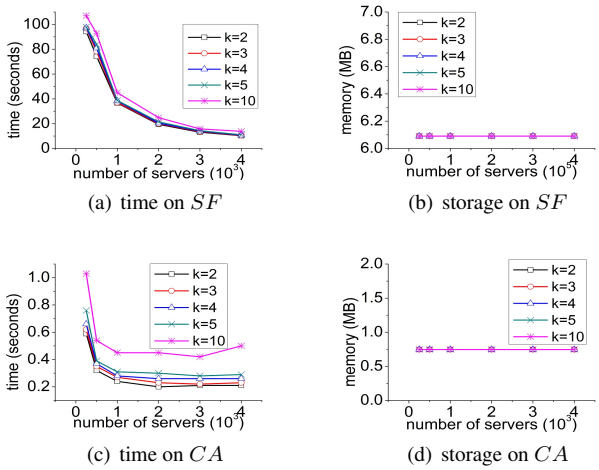


(c) time on CA (d) storage on CA

Fig. 18. Effect of $|C|$ for the multiple-location MinMax query

its nearest server). The second phase involves k iterations. After the first iteration, we find the first server and then we have to update the NLCs based on all servers and this server found. Updating the NLCs from scratch is time-consuming. In our implementation, we developed an incremental algorithm to update the new NLCs based on the NLCs obtained before the first iteration. It is easy to see that this incremental algorithm is very efficient compared with the algorithm finding new NLCs from scratch. We also do a similar operation for the second iteration and later iterations. In our experiments, the first phase spends most of the execution time but the second phase is very fast (since we incrementally update the NLCs efficiently). Thus, the query time of our proposed algorithm, *GA(MinMax)*, increases slightly when k increases.

In Figure 17(b), similarly, we have a larger memory consumption of *GA(MinMax)* when $|S|$ is small. The memory increases slightly with $|S|$. Since the memory consumption of *GA(MinMax)* is independent of k, the memory consumption of *GA(MinMax)* is very close or almost the same for different values of k.



(a) time on SF (b) storage on SF

(c) time on CA (d) storage on CA

Fig. 19. Effect of $|S|$ for the multiple-location MaxSum query

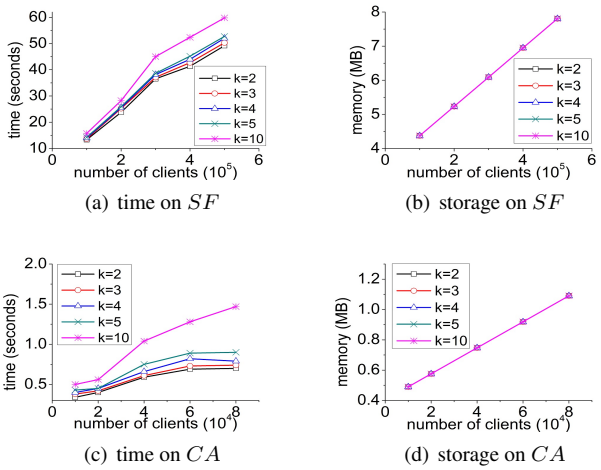
The experimental results about the effect of $|S|$ on the *CA* dataset with different values of k can be found in Figure 17(c) and Figure 17(d).

The experimental results about the effect of $|C|$ on the *SF* dataset can be found in Figure 18(a) and Figure 18(b). The experimental results about the effect of $|C|$ on the *CA* dataset can be found in Figure 18(c) and Figure 18(d).

8.3.2. Multiple-Location MaxSum Query. In this section, we study the performance of the *GA(MaxSum)* algorithm for the optimal multiple-location query. The experimental results on the SF dataset and the CA dataset are reported. The experimental results about the effect of $|S|$ on the *SF* dataset with different values of k can be found in Figure 19. In Figure 19(a), similar to the original MaxSum query, when $|S|$ is small, the time of *GA(MaxSum)* is large. When k is larger, the time of *GA(MaxSum)* increases slightly. We explain the reason as follows. *GA(MaxSum)* includes two major phases. The first phase is to build the NLCs which are used to calculate the upper bound of each edge. The second phase is to iteratively find k servers maximizing a certain cost (i.e., the sum of weights of clients attracted by one of these k servers). The second phase involves k iterations. After the first iteration, we find the first server and then we have to update the NLCs based on all servers and this server found. Updating the NLCs from scratch is time-consuming. In our implementation, we developed an incremental algorithm to update the new NLCs based on the NLCs obtained before the first iteration. It is easy to see that this incremental algorithm is very efficient compared with the algorithm finding new NLCs from scratch. We also do a similar operation for the second iteration and later iterations. In our experiments, the first phase spends most of the execution time but the second phase is very fast (since we incrementally update the NLCs efficiently). Thus, the query time of our proposed algorithm, *GA(MaxSum)*, increases slightly when k increases.

Since the memory consumption of *GA(MaxSum)* is independent of k, the memory consumption of *GA(MaxSum)* is the same for different values of k as shown in the figure. The experimental results about the effect of $|S|$ on the *CA* dataset with different values of k can be found in Figure 19(c) and Figure 19(d).

The experimental results about the effect of $|C|$ on the *SF* dataset can be found in Figure 20. In Figure 20(a), when $|C|$ increases, the time of *GA(MaxSum)* also increases. When k is larger, the time of *GA(MaxSum)* increases slightly. The memory consumption of *GA(MaxSum)* is dependent on the number of clients. In Figure 20(b), when $|C|$ increases, the memory consumption of *GA(MaxSum)* increases. The experimental results about the effect of $|C|$ on the *CA* dataset can be found in Figure 20(c) and Figure 20(d).



(a) time on SF (b) storage on SF

(c) time on CA (d) storage on CA

Fig. 20. Effect of $|C|$ for the multiple-location MaxSum query

8.3.3. MinMax Query on 3D Road Networks. 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 network 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.

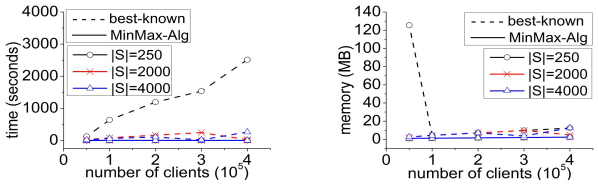
We conducted the experiments for the original MinMax query on the 3D road network. Figure 21(a) and Figure 21(b) show the time and the memory consumption of the methods, respectively, when $|C|$ increases. In Figure 21(a), similarly, *MinMax-Alg* performs more efficiently than the best-known method. Figure 21(b) shows a similar trend as the one based on the 2D road network.

The experimental results about the effect of $|S|$ on the $3D$ dataset are shown in Figure 22. The experimental results about the effect of $|C|$ on the $3D$ dataset are shown in Figure 23.

8.3.4. MaxSum Query on 3D Road Networks. We conducted the experiments for the original MaxSum query on the 3D road network. The 3D road network dataset is the same to that in Section 8.3.3. Figure 24(a) and Figure 24(b) show the time and the memory consumption of the methods when $|C|$ increases. In Figure 24(a), similarly, *MaxSum-Alg* performs more efficiently than the best-known method. When $|C|$ increases, the time of *MaxSum-Alg* decreases, and when $|S|$ decreases, the time of *MaxSum-Alg* increases. Figure 24(b) shows a similar trend as the one based on the 2D road network.

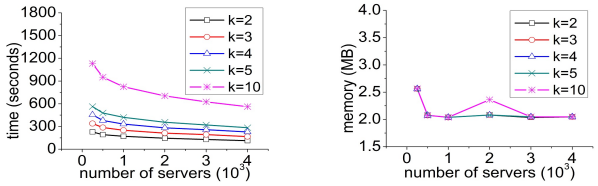
We also conducted experiments for the multiple-location MaxSum query on the 3D road network. The experimental results of the effect of $|S|$ on the $3D$ dataset are shown in Figure 25(a) and Figure 25(b). The experimental results of the effect of $|C|$ on the $3D$ dataset are shown in Figure 26(a) and Figure 26(b).

Summary. For the MinMax query and the MaxSum query, our algorithms, *MinMax-Alg* and *MaxSum-Alg*, are 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 memory consumption of our algorithms is also small and often requires less than 10MB.



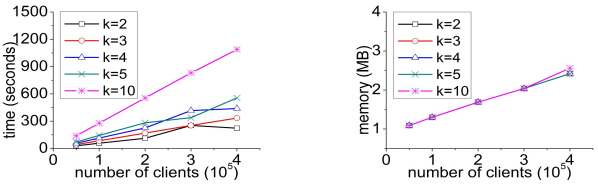
(a) time (b) storage

Fig. 21. Effect of $|C|$ on $3D$ for the MinMax query



(a) time (b) storage

Fig. 22. Effect of $|S|$ on $3D$ for the multiple-location MinMax query



(a) time (b) storage

Fig. 23. Effect of $|C|$ on $3D$ for the multiple-location MinMax query

9. CONCLUSIONS

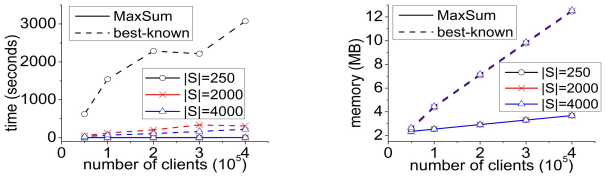
In this paper, we propose a new algorithm framework based on the idea of nearest location component for the optimal location query in the context of road networks. We present an efficient algorithm, namely *MinMax-Alg* (*MaxSum-Alg*), for the optimal location query with the MinMax (MaxSum) cost function. We also discuss the extensions to the optimal location query problem, namely the optimal multiple-location query problem and the optimal location query problem on 3D road networks. We conducted extensive experiments and the 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.

Acknowledgements

We are very thankful to the anonymous reviewers for the very useful comments. The research of Yubao Liu, Zitong Chen, Jiamin Xiong and Ganglin Mai is supported by the National Natural Science Foundation of China (61070005, 61472453, U1401256). The research of Raymond Chi-Wing Wong and Cheng Long is supported by grant FSGRF14EG34.

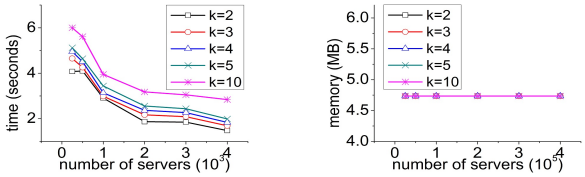
REFERENCES

Veronika Bauer, Johann Gamper, Roberto Loperfido, Sylvia Profanter, Stefan Putzer, and Igor Timko. 2008. Computing isochrones in multi-modal, schedule-based transport networks. In *GIS*. 78.



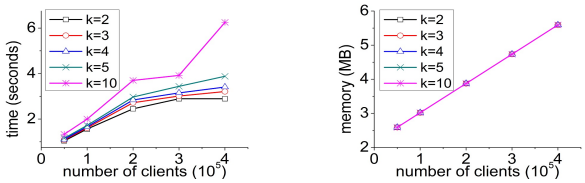
(a) time (b) storage

Fig. 24. Effect of $|C|$ on $3D$ for the MaxSum query



(a) time (b) storage

Fig. 25. Effect of $|S|$ on $3D$ for the multiple-location MaxSum query



(a) time (b) storage

Fig. 26. Effect of $|C|$ on $3D$ for the multiple-location MaxSum query

Sergio Cabello, Jose Miguel Diaz-Banez, Stefan Langerman, Carlos Seara, and Inmaculada Ventura. 2005. Reverse facility location problems. In *CCCG*. 68–71.

Jean Cardinal and Stefan Langerman. 2006. Min-max-min geometric facility location problems. In *EWCG*. 149–152.

Zitong Chen, Yubao Liu, Raymond Chi-Wing Wong, Jiamin Xiong, Xiuyuan Cheng, and Peihuan Chen. 2015. Rotating MaxRS queries. *Information Sciences* 305 (2015), 110–129.

Zitong Chen, Yubao Liu, Raymond Chi-Wing Wong, Jiamin Xiong, Ganglin Mai, and Cheng Long. 2014. Efficient Algorithms for Optimal Location Queries in Road Networks. In *SIGMOD*. 123–134.

Dong-Wan Choi, Chin-Wan Chung, and Yufei Tao. 2012. A Scalable Algorithm for Maximizing Range Sum in Spatial Databases. *PVLDB* 5, 11 (2012), 1088–1099.

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

E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *NUMERISCHE MATHEMATIK* 1, 1 (1959), 269–271.

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

Martin Erwig. 2000. The graph Voronoi diagram with applications. *Networks* 36, 3 (2000), 156–163.

Johann Gamper, Michael H. Böhlen, Willi Cometti, and Markus Innerebner. 2011. Computing isochrones in multi-modal, schedule-based transport networks. In *CIKM*. 2381–2384.

Johann Gamper, Michael H. Böhlen, and Markus Innerebner. 2012. Scalable Computation of Isochrones with Network Expiration. In *SSDBM*. 526–543.

Parisa Ghaemi, Kaveh Shahabi, John P. Wilson, and Farnoush Banaei Kashani. 2010. Optimal network location queries. In *GIS*. 478–481.

Parisa Ghaemi, Kaveh Shahabi, John P. Wilson, and Farnoush Banaei Kashani. 2012. Continuous maximal reverse nearest neighbor query on spatial networks. In *GIS*. 61–70.

Parisa Ghaemi, Kaveh Shahabi, John P. Wilson, and Farnoush Banaei Kashani. 2014. A comparative study of two approaches for supporting optimal network location queries. *GeoInformatica* 18, 2 (2014), 229–251.

John Hershberger. 1989. Finding the Upper Envelope of n Line Segments in O(nlogn) Time. *Inf. Process. Lett.* 4, 33 (1989), 169–174.

Manohar Kaul, Bin Yang, and Christian S. Jensen. 2013. Building Accurate 3D Spatial Networks to Enable Next Generation Intelligent Transportation Systems. In *MDM*. 137–146.

Jakob Krarup and Peter Mark Pruzan. 1983. The simple plant location problem: Survey and synthesis. *European Journal of Operational Research* 12, 1 (1983), 36–57.

Yubao Liu, Raymond Chi-Wing Wong, Ke Wang, Zhijie Li, Cheng Chen, and Zhitong Chen. 2013. A New Approach for Maximizing Bichromatic Reverse Nearest Neighbor Search. *Knowl. Inf. Syst.* 36, 1 (2013), 23–58.

Adam Meyerson. 2001. Online Facility Location. In *FOCS*. 426–431.

Jianzhong Qi, Rui Zhang, Lars Kulik, Dan Lin, and Yuan Xue. 2012. The Min-dist Location Selection Query. In *ICDE*. 366–377.

Barbaros C. Tansel, Richard L. Francis, and Timothy J. Lowe. 1983. Location on networks: A survey. *Management Science* 29, 4 (1983), 498–511.

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

Raymond Chi-Wing Wong, M. Tamer Ozsu, Ada Wai-Chee Fu, Philip S. Yu, Lian Liu, and Yubao Liu. 2011. Maximizing Bichromatic Reverse Nearest Neighbor for Lp-Norm in Two- and Three-Dimensional Spaces. *VLDB J.* 20, 6 (2011), 893–919.

Xiaokui Xiao, Bin Yao, and Feifei Li. 2011. Optimal Location Queries in Road Network Databases. In *ICDE*. 804–815.

Da Yan, Raymond Chi-Wing Wong, and Wilfred Ng. 2011. Efficient methods for finding influential locations with adaptive grids. In *CIKM*. 1475–1484.

Bin Yao, Xiaokui Xiao, Feifei Li, and Yifan Wu. 2014. Dynamic Monitoring of Optimal Locations in Road Network Database. *VLDB J.* 23, 5 (2014), 697–720.

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

Zenan Zhou, Wei Wu, Xiaohui Li, Mong Li Lee, and Wynne Hsu. 2011. MaxFirst for MaxBRkNN. In *ICDE*. 828–839.

Online Appendix to: Optimal Location Queries in Road Networks

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

© YYYY ACM 0362-5915/YYYY/01-ARTA $15.00
DOI : http://dx.doi.org/10.1145/0000000.0000000

Table II.
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 set
I	a point interval
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
$\|C'\|$	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
C_{p_o}	a set of clients attracted by p_o
$\mathcal{S}$	the set of all clients whose NLCs cover e
$\mathcal{S}'$	the set of all effective clients whose NLCs cover e
$GPI(e)$	the greatest possible influence of e
$GPEI(e)$	the greatest possible effective influence of e
ξ	the total number of edges containing clients
$\mathcal{E}$	the set of all sub-edges and all non-split edges
EC_e	the effective client for each of edge/sub-edge e in $\mathcal{E}$ containing clients
$EC_e.dist$	the notation of $dist$ for an effective client EC_e
$NLC(EC_e)$	the nearest location component of an effective client EC_e
$w(EC_e)$	the weight of EC_e
ϵ	the total number of edges processed in the algorithm iteration
IV_o / IV	the greatest influence value
ACS	the set of client attracted by the optimal location
CS	the subset of ACS
$\mathcal{IS}$	the set of optimal multiple-locations