An Index-Based Method for Efficient Maximizing Range Sum Queries in Road Network

Xiaoling Zhou(✉) and Wei Wang

University of New South Wales, Sydney, Australia
{xiaolingz,weiw}@cse.unsw.edu.au

Abstract. Given a set of positive weighted points, the Maximizing Range Sum (maxRS) problem finds the placement of a query region r of given size such that the weight sum of points covered by r is maximized. This problem has long been studied since its wide application in spatial data mining, facility locating, and clustering problems. However, most of the existing work focus on Euclidean space, which is not applicable in many real-life cases. For example, in location-based services, the spatial data points can only be accessed by following certain underlying (road) network, rather than straight-line access. Thus in this paper, we study the maxRS problem with road network constraint, and propose an index-based method that solves the online queries highly efficiently.

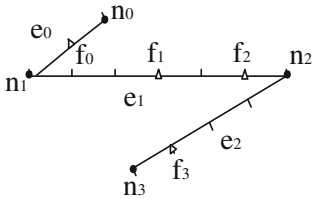
Keywords: Maximizing range sum · Road network · Query processing

1 Introduction

In recent years, location-based services that answer queries on spatial databases have drawn much attention, due to the proliferation of mobile computing devices. One of the common queries is the *Maximizing Range Sum* (maxRS) problem [1–6]. Given a set of positive weighted spatial points and a query shape(eg. rectangular or circular) r of user specified size, the maxRS problem finds an optimal placement of r such that the total weight of all points covered by r is maximized. The maxRS problem is widely applied in facility locating problems [8] for finding the best facility location with maximum number of potential clients, spatial data mining for extracting interesting locations from log data [9], and point enclosing problems.

However, most of the existing work adopt Euclidean distance metric in their method. This is not applicable in many real-life location-based services, where the spatial data points can only be accessed by following certain underlying (road) network. For example a tourist service that answers user queries of finding the most attractive places in a city in the sense that it is close to as many sightseeing spots as possible within a given range (eg. 5 km walking distance). In such scenarios, the ways to access scenic spots are constrained by the road

M.A. Cheema et al. (Eds.): ADC 2016, LNCS 9877, pp. 95–109, 2016.
DOI: 10.1007/978-3-319-46922-5_8



(a) Road Network

Edge	SNode	ENode	Length
e_0	n_0	n_1	2
e_1	n_1	n_2	6
e_2	n_2	n_3	4

(b) Edges

Facility	Edge	Position	Weight
f_0	e_0	1	3
f_1	e_1	3	8
f_2	e_1	5	5
f_3	e_2	3	2

(c) Facilities

Fig. 1. Road network example

network, hence the actual distance between two locations is probably significantly different from their Euclidean distance, for example the distance between facilities f_0 and f_3 in Fig. 1(a).

Therefore, in this paper, we study the maxRS problem with a road network constraint, where the distance between two points in the network is determined by the length of the shortest path connecting them (i.e. network distance [17]). That is, given a road network, a set of positive weighted facilities on it, and a network radius r, we find a location p on the network that maximizes the total weight of facilities whose network distance to p is no larger than r. Figure 1 shows an example of a road network containing 4 nodes, 3 edges, and 4 weighted facilities. Given a query radius $r = 2$, any position between f_1 and f_2 can be the answer of the maxRS query, since the total weight of facilities that can reach these positions within distance 2 is the maximum (i.e. 13).

Currently, the only existing work, as far as we know, that deals with the maxRS problem in a road network is [14]. The authors proposed an external-memory algorithm based on segments generation and linesweeping on the road network. However, their method is not efficient for large radius or road network databases, as shown in the experimental results in Sect. 6. They take tens or hundreds of seconds to answer one query with specified radius, which is undesirable for a mobile or web service that copes with the needs of answering millions of concurrent user queries, each with a different radius parameter. Thus, we propose a new solution to this problem that answer online queries much more efficient (around 6–8 orders of magnitude faster) than previous method, by making use of a tiny precomputed index which is of size linear to the facility number, and therefore is superior for dealing with large number of concurrent or batch query workloads.

In the rest of the paper, we formally define the problem in Sect. 2, and introduce the details of our proposed method in Sects. 3 and 4. Two optimizations for index construction are presented in Sect. 5. Experimental results on various parameters demonstrate the superiority of our method compared with existing work in Sect. 6, and finally followed the related work review in Sect. 7 and conclusion in Sect. 8.

2 Problem Definition

We follow the definition in [14] to formally describe the problem. A road network is represented as an undirected graph $G = (V, E)$, where V is a set of nodes, and E is a set of edges. We use F to denote the set of facilities, each of which, denoted as f, is located on an edge and is associated with a positive weight $w(f)$.

Definition 1 (Network Radius and Network Range). *The network range $p(r)$ of a point p in a road network contains all points in the network whose network distance to p is no greater than r, where r is called the network radius.*

Definition 2 (MaxRS query in road network). *Given G, a set of positive-weighted facilities F, and a network radius value r, let $p(r)$ be the network range of a point p in the network, and $F_{p(r)}$ be the set of facilities covered by $p(r)$. A Maximizing Range Sum (maxRS) query in a road network finds an optimal point (i.e. position) p in G that maximizes: $\sum_{f \in F_{p(r)}} w(f)$.*[1]

3 The Proposed Method

The previous method [14] generates segments from each facility f along the network until distance reaches query radius r, then sorts the segments and uses line-sweeping method to find final answer. This method takes time $O(|E||F| \log |F|)$ in worst case. The performance is undesirable, especially when r is large, which can be observed from the experimental results in Sect. 6.

Therefore, we hope to accelerate online query via some precomputed index. A direct idea is to compute the optimal location p and its weight sum w for each possible query radius r and store $<r, w, p>$ as index entries. This is obviously impossible since there are infinite number of distinct radius values. However, we can substantially reduce the index to a feasible size by the following way: (1) When radius r increases, the maximum weight sum w is non-decreasing. Thus, we can categorize the naive index entries into equivalent classes, where each class contains entries whose weights are the same. (2) For each class, store only the entry with minimum r into index. For example, if for both radiuses r and $r'(r' > r)$, the maximum weight sum can be achieved is w, we store only $<r, w, p>$, and leave out $<r', w, p'>$. The reduced index of previous example (Fig. 1) is shown in Table 1.[2]

The reduced index size is upper bounded by $\sum_{f \in F} w(f)$ since this is the maximum number of distinct weight sum values. Then we have $\sum_{f \in F} w(f) \leq \max_{f \in F} w(f) \cdot |F| = O(|F|)$ when $\max_{f \in F} w(f)$ is constant, which is usually the case.

The above index structure immediately leads to a simple binary search query processing method with complexity $O(\log |F|)$.

[1] W.l.o.g., we assume r and edge length are real numbers, and $w(f)$ is constant integer.

[2] Note that there may exist multiple optimal locations, but we store only one in the index.

Table 1. Example index

r	MaxWeight	OptPosition
0	8	$[e_1, 3]$
1	13	$[e_1, 4]$
3	16	$[e_1, 2]$
5	18	$[e_1, 4]$

Theorem 1 *Given query radius r, assume r∗ is the largest radius in the index that is smaller than or equal to r, then maxRS(r) = maxRS(r*).*

Therefore, according to Theorem 1, given query r, we use binary search to find $r*$ in the index, and retrieve the index entry as maxRS result for r. For example, if given $r = 4$ and the index in Table 1, we will return the entry $<3, 16, [e_1, 2]>$.

4 Index Construction

In this section, we introduce the details of index construction.

Lemma 1 *Given radius r, if a point p is in the network range of facility f (i.e. f(r)), then f is in $F_{p(r)}$, and vice versa.*

Using Lemma 1, we can transfer the problem of finding an optimal location in the network that can *reach* facilities with maximum total weight within radius r to the problem of finding a position that can *be reached* by facilities with maximum total weight within radius r.

Unlike the previous method [14] where r is known in advance, we do not assume r during index construction, thus the aggressively generating segments until reach length r is not applicable to our case. Therefore, we design an event-driven algorithm which is essentially a simulation of the following process.

From each facility f, we generate directed cursors carrying weight $w(f)$, and go to every possible way along the network. We move all the cursors simultaneously. Whenever there are cursors meet together, we check at the meet position (p) the total weight (w) of cursors have passed this position (we call it **Location Weight** of p). If w is larger than the last indexed entry, we add new index entry $<R, w, p>$, where R is the current total moving distance for each cursor.

The above moving process is driven by the following two kinds of events:

- **Meet Event:** There are cursors meet, i.e. two or more cursors having different directions reach the same position on an edge.
- **End Event:** There are cursors reach the edge ends, i.e. nodes in the network.

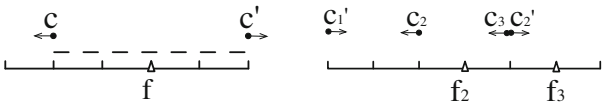
Algorithm 1 shows the main construction process. Here we assume all facilities are on edges but not on nodes, for ease of presentation. Thus we generate two cursors for each f, one goes left (towards startnode) and another goes right

(towards endnode), we call them **reverse cursor** of each other. A cursor c is denoted as $[facilityId, direction, position]$. After generating all cursors, we organize them by the edge they belong to, and sort them according to their positions (Lines 4–5). We call all the edges who have cursors on it as *active edges* ($\mathcal{A}$). Then we compute for each active edge the minimum distance required to reach one of the above events (denoted as **dmin**), and choose among all the *dmin*s the global minimum one as the moving distance d (Lines 6–8) to move all the cursors.

The while loop simulates the continuous moving process. During each iteration, an optimization applied here is that (1) if an edge entry ee's $dmin(ee) > d$, we say its a general case edge. We just accumulate d to its *need move distance* m, rather than actually move all the cursors on it. The new $dmin(ee)$ is computed as $dmin(ee) - d$ (Lines 13–14); (2) if the edge's $dmin(ee) = d$, we say it's an exact case edge. We know there must be at least one event occur on this edge after moving all the cursors. Therefore, we check after moving:

- All the meet positions to see if a new index entry has to be added (Lines 17–19).
- All the cursors that reach edge ends, to see if they have fully passed through this edge. If so, we add the facility this cursor belongs to to the edge's fully passed list FP (Line 22). We also need to erase the end cursors, and add new ones onto the adjacent edges (Lines 23).
- Remove this edge from the active edges list if there's no cursors on it.

Lemma 2 *We say a facility f has fully passed edge e only if the total track of all cursors generated from f has fully covered edge e. For example in Fig. 2(a), c and c' are the two cursors generated from f, and the dotted line represents the total track of the two cursors. Although c' has reached the edge end, the track has not fully covered edge e yet, so we do not add f to e's fully passed list until c reaches edge end.*



(a) Not Fully Pass (b) Compute dmin

Fig. 2. Examples

Lemma 3 *To avoid redundant edge visiting, we only add a facility f's cursor c to an edge e if the following conditions meet:*

1. *f is not in e's fully passed list $FP[e]$.*
2. *There is no other cursors of f that has same direction with c on e.*

Algorithm 1. IndexConstruction(E, V, F)

```
   Input  : The list of edges E, vertices V, and facilities F.
   Output: The list of index entries I.
1  R ← 0; I, A, FP ← ∅ ;
2  d ← ∞;                                  /* the global dmin */;
3  I ← I ∪ <0, fmax.w, fmax.pos>;          /* fmax:the facility with max weight */;
4  for each f in F do
5      A ← A ∪ <f.eid, ∞, 0, [f.id, left, f.pos], [f.id, right, f.pos]> ;
6  for each edge entry ee in A do
7      ee.dmin ← GetDmin(ee) ;
8      d ← ee.dmin < d ? ee.dmin : d;          /* update global dmin */;
9  while I.back.weight < Σ_{f∈F} w(f) do
10     R ← R + d; d ← ∞ ;
11     for each ee in A do
12         if ee.dmin > d then
13             ee.dmin ← ee.dmin − d ;
14             ee.m ← ee.m + d;            /* accumulate need move dist */;
15         else                                                /* exact case edge */
16             move all cursors on ee with distance (ee.m + d); ee.m ← 0 ;
17             for each meet position p on ee do
18                 if GetLocationWeight(p, ee) > I.back.weight then
19                     I ← I ∪ <R, p.weight, p.pos> ;
20                 destroy meet cursors at p belonging to common facility ;
21             for each end cursor c on ee do
22                 add c.fid to ee's fully passed list FP[ee] with precheck ;
23                 add c to adjacent edges with precheck; destroy c ;
24             destroy ee if no cursor on it; ee.dmin ← GetDmin(ee) ;
25         d ← ee.dmin < d ? ee.dmin : d;      /* update global dmin */;
26 return I;
```

In the above processes, there are two technical problems:

- Given an edge entry ee with a list of cursors on it, how to compute $dmin(ee)$.
- Given a position in the road network, how to compute its current location weight.

We solve these two problems in the following sections.

4.1 Compute *dmin* for Each Edge

Algorithm 2 shows the process to compute $dmin$ for an active edge. Recall that $dmin$ is the minimum distance required to reach one of the above events. We first consider the distances that will trigger End Event: (1) the distance between the leftmost go-left cursor and startnode (Line 4); (2) the distance between the rightmost go-right cursor and endnode (Line 5).

Next, we consider the distances that will trigger Meet Event: Choose the shorter list between go-left cursors list and go-right cursors list as S. For each

cursor c in S, find its closest but not yet meet cursor c' in another list. The meet distance is computed as $dist(c, c')/2$ (Lines 7–9).

From all the distances considered above, we choose the minimum one as $dmin(ee)$ and return it. Consider an illustrative example in Fig. 2(b). To compute its $dmin$, we look at:

(1) the distance to reach left end: $dist(startnode, c_2) = 2$.
(2) the distance to reach right end: $dist(c'_2, endnode) = 2$.
(3) the distance for meet event(assume S is go-right cursors): $dist(c'_1, c_2)/2 = 1$.

Thus, the final $dmin$ is 1.

Algorithm 2. GetDmin(ee)

Input : The edge entry ee.
Output: The minimum moving distance $dmin$ to trigger an event.

```
1  d ← ∞ ;
2  GL ← sorted list of go-left cursors on ee ;
3  GR ← sorted list of go-right cursors on ee ;
4  d ← GL[first].dist < d ? GL[first].dist : d;      /* trigger End Event */;
5  d' ← ee.length − GR[last].dist; d ← d' < d ? d' : d;  /* trigger End Event */;
6  S ← the shorter list between GL and GR ;
7  for each cursor c in S do
8      c' ← the closest but not meet yet cursor in the other list ;
9      d' ← dist(c, c')/2; d ← d' < d ? d' : d;      /* trigger Meet Event */;
10 return d;
```

4.2 Compute Location Weight

Algorithm 3 addresses the problem of retrieving the current location weight of a given position in the network. Given a position p on edge entry ee, its location weight is composed by two parts:

1. The weights of facilities that fully passed edge ee. This can be obtained by checking ee's fully passed list $FP[ee]$ (Line 1).
2. The weights of facilities that passed p, but not in ee's fully passed list, which means they must have active cursors on ee. Thus, we retrieve them by looking at all the go-left cursors on the left of p, and all the go-right cursors on the right of p. For each such cursor c, assume its facility is f, we check if they actually passed p in the following way:
   - If f is not on edge ee, then we know c must get on ee from one end and moving towards another end, so c must have passed p.
   - If f is on edge ee: (1) if p is in between the positions of f and c, then c must have passed p; (2) If p is not in between, then the possible positions of p must be in case p_1 or p_2 shown in Fig. 3(a). We can judge whether c or

its *reverse cursor* c' passed p by checking whether $dist(c, f) \geq dist(f, p)$, if so, we add f as a passed facility. (This comparison is to address the special case in Lemma 2 where f is added to fully passed list only if its cursors' moving track fully covers the edge.)

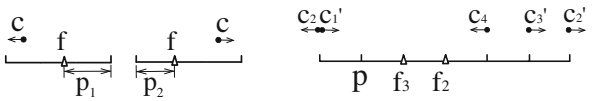
We use set F' to store passed facilities to avoid redundant weight accumulation.

For example, the location weight of p in Fig. 3(b) is computed as $w(f_2) + w(f_3)$.

Algorithm 3. GetLocationWeight(p, ee)

Input : The edge entry ee, and a position p on it.
Output: The current location weight of p.

```
1  F' ← ∅; w ← Σ_{f∈FP[ee]} w(f) ;
2  for each cursor c on ee do
3      if c.pos ≤ p.pos and c.dir = left then
4          if f not on ee or f.pos ≥ p.pos or dist(c, f) ≥ dist(f, p) then
5              F' ← F' ∪ f;          /* f: the facility c belongs to */;
6      else if c.pos ≥ p.pos and c.dir = right then
7          if f not on ee or f.pos ≤ p.pos or dist(c, f) ≥ dist(f, p) then
8              F' ← F' ∪ f ;
9  w ← w + Σ_{f∈F'} w(f) ;
10 return w;
```



(a) Possible Positions (b) Compute Location Weight

Fig. 3. Examples

4.3 Running Example

The running of Algorithm 1 on the road network in Fig. 1 is shown in Table 2. At the beginning (Round 0), the total moving distance R is 0, and the initial index entry $<0, 8, [e_1, 3]>$ is added into index I. Two opposite cursors from each facility are generated and organized by edges (shown in column 2). For each edge entry, its need move distance m is initialized as 0, and *dmin* is computed by Algorithm 2 (equal to 1 for all edges in this example). Thus, the global dmin d is 1.

In Round 1, we move all the cursors with distance 1 and show the current active edges. The underlined cursors represent a meet event, and a new index

Table 2. Index construction: running Algorithm 1 on example in Fig. 1

Round#	Active Edges ($\mathcal{A}$)	FP	Index(I)	R	d
0	$< e_0, 1, 0, [f_0, \mathsf{l}, 1], [f_0, \mathsf{r}, 1] >$ $< e_1, 1, 0, [f_1, \mathsf{l}, 3], [f_1, \mathsf{r}, 3], [f_2, \mathsf{l}, 5], [f_2, \mathsf{r}, 5] >$ $< e_2, 1, 0, [f_3, \mathsf{l}, 3], [f_3, \mathsf{r}, 3] >$	$\emptyset$	$0,\ 8, [e_1, 3]$	0	1
1	$< e_1, 1, 0, [f_0, \mathsf{r}, 0], [f_1, \mathsf{l}, 2], \underline{[f_1, \mathsf{r}, 4], [f_2, \mathsf{l}, 4]} >$ $< e_2, 1, 0, [f_2, \mathsf{r}, 0], [f_3, \mathsf{l}, 2] >$	$e_0 : f_0$	$0,\ 8, [e_1, 3]$ $1, 13, [e_1, 4]$	1	1
2	$< e_1, 1, 0, \underline{[f_0, \mathsf{r}, 1]}, [f_1, \mathsf{l}, 1], [f_2, \mathsf{l}, 3], [f_1, \mathsf{r}, 5] >$ $< e_2, 1, 0, \underline{[f_2, \mathsf{r}, 1], [f_3, \mathsf{l}, 1]} >$	$e_0 : f_0$	$0,\ 8, [e_1, 3]$ $1, 13, [e_1, 4]$	2	1
3	$< e_0, 2, 0, [f_1, \mathsf{l}, 2] >$ $< e_1, 2, 0, \underline{[f_0, \mathsf{r}, 2], [f_2, \mathsf{l}, 2]}, [f_3, \mathsf{l}, 6] >$ $< e_2, 2, 0, [f_1, \mathsf{r}, 0], [f_2, \mathsf{r}, 2] >$	$e_0 : f_0$ $e_1 : f_1$ $e_2 : f_3$	$0,\ 8, [e_1, 3]$ $1, 13, [e_1, 4]$ $3, 16, [e_1, 2]$	3	2
4	$< e_0, 2, 0, [f_2, \mathsf{l}, 2] >$ $< e_1, 2, 0, \underline{[f_0, \mathsf{r}, 4], [f_3, \mathsf{l}, 4]} >$ $< e_2, 2, 0, [f_1, \mathsf{r}, 2] >$	$e_0 : f_0$ $e_1 : f_1$ $e_2 : f_3$	$0,\ 8, [e_1, 3]$ $1, 13, [e_1, 4]$ $3, 16, [e_1, 2]$ $5, 18, [e_1, 4]$	5	-

entry $<1, 13, [e_1, 4]>$ is added. f_0 as a fully passed facility of e_0, is added to FP. The new global $dmin$ is computed as 1.

We repeat the above moving process, until the weight of last indexed entry equal to $\sum_{f \in F} w(f)$. After 4 rounds, the final index is shown in column 4.

5 Optimizations

Next, we present optimization techniques to improve index construction.

The Upper Bound Filter (Opt1). Given the network G and the set of facilities F, the index entries (i.e. the radius r required to reach each distinct weight sum) are fixed. To obtain the r values, in each while round in Algorithm 1, we choose the global $dmin$ as the moving distance. Therefore, the total indexing time is closely related to $dmin$ in each round. The larger moving distance in each round, the less number of rounds to go through, and the earlier we finish the index construction. So the goal is to move as long as possible in each round. Another observation is that around 90 % of edge entries (statistics from experimental results) achieve $dmin$ at meet events.

Based on these observations, we propose a simple filtering method used during computing $dmin$ for each edge e: before checking the meet cases from Line 6 in Algorithm 2, we compute the total weight of all cursors on edge e and all facilities that have fully passed e. If this weight is no larger than the last indexed entry, we know even there are meet cases occur in this edge, there will be no new index entries to be added. Therefore, we can safely skip the Lines 6–10. To

be more specific, we add the following line before Line 6 in Algorithm 2: return d if $(\sum_{c \in e} w(c) + \sum_{f \in FP[e]} w(f) \leq I.back.weight)$.

For example, at the Round 1 in Table 2, after checking end event distances on edge e_2, we compute $w(f_2) + w(f_3) = 7 < 13$, we know any meet event occur on e_2 will not generate new index entry, so we directly return 2 rather than 1 as $dmin(e_2)$.

The Exact Filter (Opt2). The above filtering method uses an upper bound to estimate the location weight when meet event occurs. Although it already achieves good performance improvement (saves 21 % construction time), the bound is not tight and can be further improved with a bit more expenses.

Using the same example in Table 2, after checking end event distances on edge e_1 in Round 1, if the upper bound filter is applied, we get $w(f_0) + w(f_1) + w(f_2) = 16 > 13$, therefore the filter has no effect. However, when f_0's go-right cursor and f_1's go-left cursor meet, the actual location weight at meet position is $w(f_0) + w(f_1) = 11 < 13$, so we can still skip this meet distance 1.

Thus, instead of using total weight of cursors and fully passed facilities as an estimation, we compute the exact location weight at future meet positions, and compare it with last indexed entry to decide whether to use this *dmin* or not. Algorithm 4 presents the details. It is similar to Algorithm 3 with the difference that Algorithm 3 computes the current location weight given a position p, while Algorithm 4 computes the future location weight at the meet position given two will meet cursors.

Continue with the above example, if exact filter is applied, we get $dmin(e_1) = 2$, therefore, the global *dmin* after Round 1 is increased to 2, which effectively reduces one round. This optimization technique reduces nearly half of the total round number and index time, and if used together with upper bound filter, achieves further speed-up as shown in Sect. 6.

Algorithm 4. ExactFutureLocationWeight(ee, c_1, c_2)

Input : The edge entry ee, and two will meet cursors c_1 and c_2 on it.
Output: The location weight of middle position p between c_1 and c_2 after meet.

```
1  F' ← ∅; w ← Σ_{f∈FP[ee]} w(f) ;
2  for each cursor c on ee do
3      if c.pos < c1.pos and c.dir = left then
4          if f not on ee or f.pos ≥ c1.pos or dist(c, f) ≥ dist(f, c1) then
5              F' ← F' ∪ f;        /* f: the facility c belongs to */;
6      else if c1.pos ≤ c.pos and c.pos ≤ c2.pos then /* cursors in between */
7          F' ← F' ∪ f ;
8      else if c.pos > c2.pos and c.dir = right then
9          if f not on ee or f.pos ≤ c2.pos or dist(c, f) ≥ dist(f, c2) then
10             F' ← F' ∪ f ;
11 w ← w + Σ_{f∈F'} w(f) ;
12 return w;
```

6 Experiment

In this section, we perform empirical experiments to confirm the substantial query performance improvement by our proposed method in practice.

Experiment Setting. In the experiment, we use two real network datasets, the North American (NA) and San Francisco (SF) road network, same as the previous work [14]. The NA dataset is obtained from [15] and SF from [16]. The facilities are generated randomly in the network with uniform distribution in terms of road network distance, and their weights are within range $(0, 50]$.[3] The cardinalities of datasets are shown in Table 3. The default facility number is 12500 if not explicitly mentioned. The maximum moving radius is set to 1000 while index construction.

We compare our method with the segment generation based algorithm [14] (denoted as SEG) mentioned before; our index-based algorithm is denoted as IND. Both methods are implemented in C++, and experiments are conducted on a server with Quad-Core 2.4 GHz Processor and 96 GB RAM. Although the method described in [14] generates segments in a DFS fashion, we implemented both the DFS and BFS versions. The results show that the DFS version is much slower than BFS due to the significant amount of redundant segments generation, therefore, in the following demonstrations, we compare our method with the BFS version result.

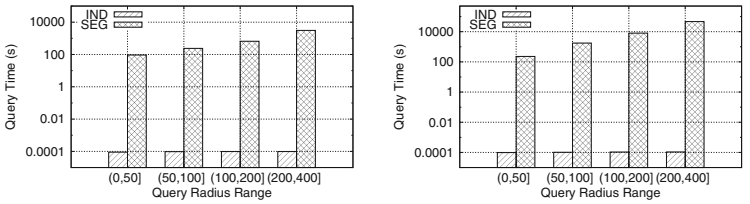
Table 3. Dataset statistics

Dataset	Nodes	Edges	Avg.EdgeLength	Facilities
NA	175813	179179	4.028	12500, 25000, 50000, 100000
SF	174956	223001	8.782	12500, 25000, 50000, 100000

Varying Query Radius Size. The first set of experiments compares the performance of both methods on various query radiuses. The four groups of radius ranges are $(0, 50]$, $(50, 100]$, $(100, 200]$ and $(200, 400]$. We produce 100 queries with lengths generated uniformly from each group, and show the *total query time* for each group in Fig. 4.

It can be seen that our method is around 6+ orders of magnitude faster than SEG. This demonstrates the substantial advantage of our index-based method to efficiently support large or batch query workloads. When query range increases, the running time of our method stays steady due to the high performance of simple binary search, while SEG takes much more time when query radius rises since they need to generate and process more segments with longer radius. This is more obvious on the SF dataset as the density of edges in SF is higher than NA.

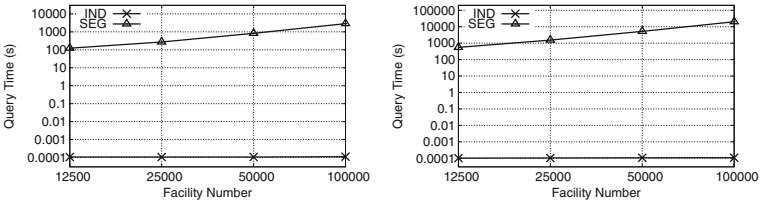
[3] We also test on another popular real dataset(California road network) from [15], where the 87635 facilities are carefully generated using the real-life facility distribution. The result trend is similar with other datasets hence omitted due to space limitation.



(a) NA (b) SF

Fig. 4. Vary query radius

Vary Facility Number. The second group tests method scalability. We start with a set of 12500 facilities, and increase its size till 4x, i.e. 100000, and measure the total query time of both methods on 100 queries randomly generated within length (0,100]. The result is plotted in Fig. 5. Clearly, our method has much better scalability than SEG as SEG's query time climbs up quickly when the data size increases. This is expected as our query time increases only logarithmically with facility number, while SEG's increases superlinearly.



(a) NA (b) SF

Fig. 5. Vary facility number

Index Construction. We show statistics about index construction in Table 4 to demonstrate the performance of different optimizations introduced in Sect. 5. The results are obtained on NA dataset with 12500 facilities. Results for other settings are similar and therefore omitted. In the table, Original means no optimization, and Opt12 means both optimizations are adopted. It is shown that Opt1 itself reduces 25 % of the Original round number, and achieves a 21 % acceleration of the index construction. While Opt2, if used alone, leads to 50 % reduction of the round number and 46 % of the total construction time. This is expected since Opt2 uses a tighter bound and thus achieves a larger global dmin in each round compared with Opt1. Finally, if both optimizations are applied, although the round number is still decreased by 50 % (as it depends on the tighter bound), the total time is further reduced to 51 % of the Original time. This is because in the cases where both Opt1 and Opt2 take effects, Opt12 uses Opt1 which requires less processing time than Opt2.

Table 4. Index construction optimizations

	Original	Opt1	Opt1/orig	Opt2	Opt2/orig	Opt12	Opt12/orig
Round#	199928	150421	75 %	101246	50 %	101246	50 %
Time(s)	37661	29893	79 %	20404	54 %	19274	51 %
Size	76 KB						

The index size is 76 KB despite what optimizations are used to construct it. The small index size further confirms the superiority and practicability of our method.

7 Related Work

Facility location optimization problem finds the optimal location by maximizing/minimizing some objective functions such as location influence (i.e. total weight of its RNNs) [10], average min-dist (distance from each object to its nearest facility) [11], total weighted distance to RNNs [12], and so on (see [13] for survey).

The maxRS problem can been seen as another instance of the facility location problem. Considering the axis-parallel rectangular query range, Nandy et al. [2] proposed an $O(n \log n)$ time algorithm to solve it using the plane-sweeping technique [1] with interval trees. Choi et al. [3] proposed an external memory solution following the distribution sweep paradigm [7], and the work was further extended in [5] by providing solutions to the AllMaxRS problem, which retrieves all optimal locations achieving the maximum total covered weight. Tao et al. [4] studied the approximate maxRS problem, and obtained a $(1-\epsilon)$-approximate answer with high confidence in time $O(n \log \frac{1}{\epsilon} + n \log \log n)$ via grid sampling.

Another variation of maxRS problem is maximizing circular range sum(maxCRS) problem, meaning that query range is a circle. Chazelle et al. [18] solved the maxCRS problem in time $O(n^2)$. Aronov et al. [19] proposed a $(1-\epsilon)$-approximate algorithm with complexity $O(n\epsilon^{-2} \log n)$ for unweighted points, and $O(n\epsilon^{-2} \log^2 n)$ for weighted case. Choi et al. [3] solved the maxCRS problem by first converting it to the maxRS problem.

All the above maxRS related work assume Euclidean space. The only existing work, as far as we know, that studied the maxRS problem in road network is [14]. Their proposed method finds answer for a particular query radius r in time $O(|E||F| \log |F|)$, which is very time-consuming when the network or r is large. We devise an index-based method that answers queries in time $O(\log |F|)$ with $O(|F|)$ index size, hence provides significant speed-up to existing method and is beneficial to frequent queries.

8 Conclusion

In this paper, we study the maximizing range sum problem in road network. The only existing work [14] proposed an external-memory algorithm that solves

one specific query with $O(|E||F| \log |F|)$ time. It is not satisfactory for mobile or web services dealing with millions of concurrent user queries or batch queries. We propose an index-based method that results in $O(\log |F|)$ online query time (6+ orders of magnitude faster than existing method in practice), with a tiny index of size linear in facility number $|F|$. Besides, we propose optimization techniques that achieve around 50 % reduction of the offline index construction time. Experiments on various settings verify the efficiency and scalability of our method.

References

1. Imai, H., Asano, T.: Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. J. Algorithms **4**(4), 310–323 (1983)
2. Nandy, S.C., Bhattacharya, B.B.: A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids. Math. Appl. **29**(8), 45–61 (1995)
3. Choi, D.W., Chung, C.W., Tao, Y.: A scalable algorithm for maximizing range sum in spatial databases. Proc. VLDB Endow. **5**(11), 1088–1099 (2012)
4. Tao, Y., Hu, X., Choi, D.W., Chung, C.W.: Approximate MaxRS in spatial databases. PVLDB **6**(13), 1546–1557 (2013)
5. Choi, D.W., Chung, C.W., Tao, Y.: Maximizing range sum in external memory. ACM Trans. Database Syst. **39**(3), 21: 1–21: 44 (2014)
6. Mukherjee, M., Chakraborty, K.: A polynomial time optimization algorithm for a rectilinear partitioning problem with applications in VLSI design automation. Inf. Process. Lett. **83**, 41–48 (2002)
7. Goodrich, M.T., Tsay, J.-J., Vengroff, D.E., Vitter, J.S.: External-memory computational geometry (preliminary version). In: FOCS, pp. 714–723 (1993)
8. Abellanas, M., Hurtado, F., Icking, C., Klein, R., Langetepe, E., Ma, L., Palop, B., Sacristán, V.: Smallest color-spanning objects. In: Meyer auf der Heide, F. (ed.) ESA 2001. LNCS, vol. 2161, p. 278. Springer, Heidelberg (2001)
9. Tiwari, S., Kaushik, H.: Extracting region of interest (roi) details using lbs infrastructure and web databases. In: MDM 2012, pp. 376–379 (2012)
10. Du, Y., Zhang, D., Xia, T.: The optimal-location query. In: Medeiros, C.B., Egenhofer, M., Bertino, E. (eds.) SSTD 2005. LNCS, vol. 3633, pp. 163–180. Springer, Heidelberg (2005)
11. Zhang, D., Du, Y., Xia, T., Tao, Y.: Progressive computation of the min-dist optimal-location query. In: VLDB (2006)
12. Xiao, X., Yao, B., Li, F.: Optimal location queries in road network databases. In: ICDE (2011)
13. Farahani, R.Z., Hekmatfar, M.: Facility Location: Concepts, Models, Algorithms and Case Studies, 1st edn. Physica-Verlag, Heidelberg (2009)
14. Phan, T.K., Jung, H.R., Kim, U.M.: An efficient algorithm for maximizing range sum queries in a road network. Sci. World J. **2014** (2014). Article ID 541602
15. http://www.cs.fsu.edu/lifeifei/SpatialDataset.htm
16. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica **6**(2), 153–180 (2002)
17. Yiu, M.L., Mamoulis, N.: Clustering objects on a spatial network. In: SIGMOD (2004)

18. Chazelle, B.M., Lee, D.T.: On a circle placement problem. Computing **36**, 1–16 (1986)
19. Aronov, B., Har-Peled, S.: On approximating the depth and related problems. In: Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, pp. 886–894 (2005)