IEICE TRANS. INF. & SYST., VOL.E99–D, NO.5 MAY 2016

PAPER

Efficient Evaluation of Maximizing Range Sum Queries in a Road Network

Tien-Khoi PHAN[†], **HaRim JUNG**[†], **Hee Yong YOUN**[†], ***Nonmembers***, ***and*** **Ung-Mo KIM**[†a)], ***Member***

SUMMARY Given a set of positive-weighted points and a query rectangle r (specified by a client) of given extents, the goal of a maximizing range sum (MaxRS) query is to find the optimal location of r such that the total weights of all points covered by r is maximized. In this paper, we address the problem of processing MaxRS queries over road network databases and propose two new external memory methods. Through a set of simulations, we evaluate the performance of the proposed methods.

key words: *facility optimization location, location-based services, maximizing range sum query, road network database, spatial database*

1. Introduction

Location-based services have recently attracted much attention as one of the most promising applications whose main functionality is to process location-based queries on spatial databases. Most traditional studies on spatial databases have focused on finding nearby data objects (e.g., *range queries* and *nearest neighbor queries*), rather than finding the best location to optimize a certain objective. Recently, a *maximizing range sum* (MaxRS) query was introduced [1]. Given a set of positive-weighted points and a query rectangle r of a given size, the goal of a MaxRS query is to find the optimal location of r such that the sum of the weights of all the points covered by r is maximized. A MaxRS query is useful in many location-based applications such as finding the most representative place in a city with a limited reachable range for a tourist or finding the best location with a limited delivery range for a pizza store.

Figure 1 (a) shows an example of the MaxRS query, where the size of the query rectangle r is $a \times b$ and all the points are assumed to have the same weight, 1. The center of the solid-lined rectangle is the optimal location of r because it covers the largest number of points (i.e., 3).

To process MaxRS queries on Euclidean space, Choi et al. [1] proposed an external-memory algorithm, while Imai and Asano [2] proposed an in-memory algorithm. In many real-life location-based services, however, the motion of a client may be constrained by an underlying (spatial) road network. Consider the scenario of a tourist service as an example, where a tourist (i.e., client) tries to find the hotel whose location is close to as many sightseeing spots as possible (e.g., maximum 2.0 km walk from the hotel). In this scenario, a MaxRS query can be applied. However, the existing MaxRS query processing methods cannot be applied because the distance between the hotel and each sightseeing spot is confined by the underlying (spatial) road network, and thus, the actual distance between two locations can differ significantly from their Euclidean distance. With this problem in mind, we study the problem of processing MaxRS queries over road network databases, where the distance between two points is determined by the length of the shortest path connecting them.

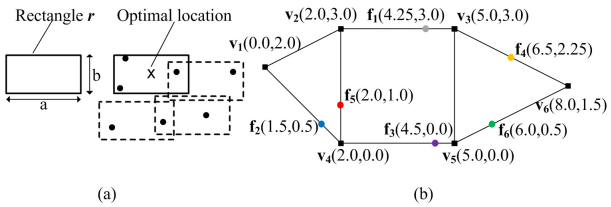

Fig. 1 Examples of the MaxRS query and road network

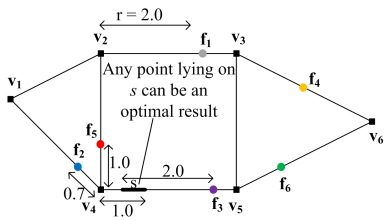

Fig. 2 MaxRS query in road network

Figure 1 (b) shows an example of the road network, which consists of 6 nodes (square vertices) and 8 edges. There are 6 facilities $f_1, f_2, \ldots, f_6$ (a set of weighted points), each of which is associated with a positive weight $w(f_i)$ indicating the relative importance of f_i, where $1 \le i \le 6$. The pairs of values in parentheses indicate the coordinates of nodes and facilities. In this paper, we assume that all facilities are located on the edges of the road network. Then, a MaxRS query in a road network is defined as follows. Given a set of facilities and a radius r, the MaxRS query finds all the locations p (on a road network), which maximizes the total weights of all the facilities whose network distance to p is less than or equal to r.

Based on the road network shown in Fig. 1 (b), Fig. 2 shows an example of MaxRS query with the radius $r = 2.0$. The weight of each facility is assumed to be 1. Consider the stage s, where the distance between each point in s and three facilities f_2, f_3, f_5 is less than or equal to r, while the

Manuscript received October 2, 2015.
Manuscript revised January 7, 2016.
Manuscript publicized February 16, 2016.

[†]The authors are with the College of Information and Communication Engineering, Sungkyunkwan University, 440–746, Korea.
a) E-mail: ukim@skku.edu (Corresponding author)

DOI: 10.1587/transinf.2015EDP7422

Copyright © 2016 The Institute of Electronics, Information and Communication Engineers

total weight of these facilities is 3, which is the maximum in this scenario. Therefore, s becomes the result of the MaxRS query and the query issuer can choose any point lying on s.

In our previous paper [3], we proposed the MaxRS query processing method in a road network, namely the B^+-*Tree seg-file* (*BTSF*) method. In this paper, after discussing the limitations of the BTSF method, we propose two new methods, called the *Hilbert R-Tree seg-file* (*HRSF*) method and the Edge-Area (*EA*) method, both of which make use of spatial characteristics of the road network. The main contributions of our study is to discuss the limitations of the BTSF method and to propose the HRSF and EA methods, and to verify the scalability of the proposed methods.

The reminder of this paper is organized as follows. In Sect. 2, the problem is formally defined and the preliminaries are provided. In Sect. 3, the drawbacks of BTSF method are discussed. In Sect. 4 and Sect. 5, the details of the HRSF method and the EA method, respectively, are described. In Sect. 6 the results of simulation experiments are presented. In Sect. 7, some related work is reviewed. Finally, Sect. 8 concludes the paper.

2. Problem Formulation and Preliminaries

2.1 Problem Formulation

A road network is represented by an undirected graph $G = (V, E)$, where V is a set of vertices (i.e., nodes) and E is a set of edges. Let F be a set of facilities, each of which, denoted by f, is located on an edge (in E) and is associated with a positive weight $w(f)$.

Definition 1: (*Network range and network radius*). Network range $p(r)$ of a point p in a road network consists of all points (in the road network) whose network distance to p is less than or equal to the value r, where r is the network radius of p.

Definition 2: (*A MaxRS query in a road network*). Given G, F and a network radius value r, let $p(r)$ be the network range of a point p in the road network, and $F_{p(r)}$ be the set of facilities covered by $p(r)$. Then, a Maximizing Range Sum (MaxRS) query in a road network finds all points p (in G) that maximizes $\sum_{f \in F_{p(r)}} w(f)$.

2.2 Preliminaries

In this section, we present the background ideas of solving MaxRS in a road network and the overview of the storage system assumed in this paper. We also summarize the BTSF method [3], the first method for processing MaxRS queries in a road network.

2.2.1 Background Idea

We first remind the idea of transforming the *max-enclosing rectangle query* into the *rectangle intersection query* [4],

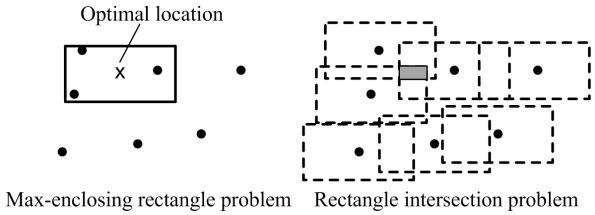

Fig. 3 Example of transformation

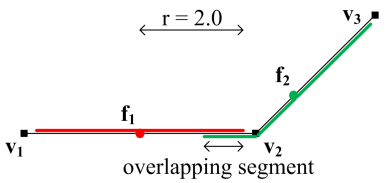

Fig. 4 Max-Segment in MaxRS query

which is the fundamental idea for processing MaxRS queries in Euclidean space [1].

Definition 3: (*A max-enclosing rectangle query*). Given a set of points O, a rectangle r with a given size, a max-enclosing rectangle query finds the location of r such that r encloses the maximum number of points in O.

The MaxRS query calculates the total weight of points, while the max-enclosing rectangle query counts the number of points in rectangle. Note that when assuming all points have the weight being equal to 1, the result of the MaxRS query equals that of the max-enclosing rectangle query.

Definition 4: (*A rectangle intersection query*). Given a set of rectangles R, a rectangle intersection query finds the most overlapped area among the rectangles in R.

It can be observed from the Fig. 3 that the optimal location in the max-enclosing rectangle query can be any point in the most overlapped area (i.e., the gray area, where 3 rectangles overlap), which is the result of the rectangle intersection query.

Consider an example of the MaxRS query in a road network shown in Fig. 4. To simplify our discussion, we use a simple road network that consists of two edges (i.e., $\langle v_1, v_2 \rangle$ and $\langle v_2, v_3 \rangle$) and two facilities (i.e., f_1 and f_2) lying on these two edges. We assume that the weight of each facility is 1 and the network radius r is 2.0. The red segment indicates the network range $f_1(r)$ of f_1, while the green segments in the figure indicate the network range $f_2(r)$ of f_2. Let S be a set of all segments presented in the network range of all facilities in the road network. Then, we define two important notions for the MaxRS query in the road network.

Definition 5: (*Location-weight*). Let p be the location in road network. Then, the location-weight of p with regard to S equals the total weights of all the segments (in S) that contain p.

Definition 6: (*Max-segment*). The max-segment M with regard to S is a segment such that every point in M has the

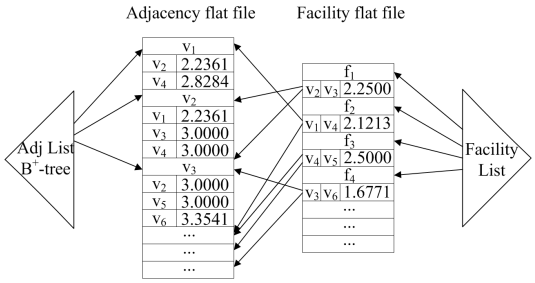

Fig. 5 Disk-based storage model

same location-weight W, and no point in the road network has a location-weight higher than W.

It is easy to observe that the overlapping segment in Fig. 4 is a max-segment. Because all max-segments in the road network contain all the optimal locations (i.e., the result of the MaxRS query in the road network), we need to find all max-segments to evaluate the MaxRS query.

2.2.2 Storage System Overview

Similarly to the disk-based storage model was proposed by Yiu et al. [5], the road network and the facility set are assumed to be stored in a secondary storage. Figure 5 shows the files and indexes for the road network and facility set. In this storage model, the road network (adjacency list) is stored in a flat file, which is indexed by the B$^+$-Tree. For each node v (e.g., v_1), beside the information of v (i.e., identifier and coordinates), we also store two information of all adjacent nodes (identifier and Euclidean distance to v (e.g., length of edge $\langle v_1, v_2 \rangle$)). Similarly, the facility list is also stored in a flat file and indexed by an index structure (e.g., Hilbert R-Tree). To support the MaxRS query processing algorithm efficiently, beside the information of each facility f (i.e., identifier, coordinates, weight), we store the information of the edge that contains f (identifiers of start node and end node, and the distance (offset) between start node and f (e.g., start node of f_1 is v_2, end node of f_1 is v_3 and length of segment $\langle v_2, f_1 \rangle$ is 2.25)).

2.2.3 The B$^+$-Tree Seg-File Method

We summarize our previous method, namely the B$^+$-Tree seg-file (BTSF) method. This method consists of three main steps for processing the MaxRS query. Firstly, for each facility f stored in the B$^+$-Tree facility flat file, the BTSF method generates the segments that cover the overall network range $f(r)$ of f. The segments generated for f have the weight being equal to $w(f)$. Secondly, the BTSF method inserts generated segments into the seg-file. One important point of the seg-file is that all segments on the same edge is grouped into one record, which is called the edge-record. Each edge-record in the seg-file, which is indexed by B$^+$-Tree, has the format of the form $\langle edge, (segment1, segment2, \ldots) \rangle$. This structure of the seg-file helps find max-segments effectively. Finally, the BTSF method processes the seg-file to find the max-segments. The BTSF method scans each edge-record in the seg-file to find the local optimal segments. The final result includes the local optimal segments with the maximum weight.

3. Drawbacks of B$^+$-Tree Seg-File Method

In the BTSF method, the segments are generated from each facility, after which the segments in the same edge are grouped into one record. This has the following limitations:

- First, the facility flat file is indexed by B$^+$-Tree and facilities are organized based on their identifiers without considering spatial nearness among the facilities. As a result, the order of selecting facilities is not based on their locations. This may cause the inefficient access to the adjacency flat file. For example, when the BTSF method processes a facility f_1 to generate segments of f_1, it needs to access nodes (in the adjacency flat file) that are located within the network range $f_1(r)$ (See Appendix - Algorithm 4). These nodes can be stored in a buffer to improve the performance if the next processed facility f_2 is nearby f_1. In this case, the possibility of the accessed nodes being within the network range $f_2(r)$ is higher than other network ranges of the farther facilities. In addition, the edge-records can also be stored in the buffer. Similarly to the case of accessed node in the adjacency flat file, when the BTSF method processes f_2, it can access the edge-records from the buffer, which were stored when the BTSF method processed f_1.
- Second, when the BTSF method inserts a segment into the seg-file (See Algorithm 3 - insertSegment [3]), if the edge-record of this segment has already existed in the seg-file, the BTSF method needs to (i) read the edge-record (exact-match query), (ii) update the segment list of the edge-record and (iii) finally write back the edge-record to the seg-file. From the results of our performance evaluation, this insertion takes a large portion of total I/O's in the BTSF method.

4. The Hilbert R-Tree Seg-File Method

In this section, we present the HRSF method, where the facility flat file and seg-file are indexed by the Hilbert R-Tree (HRT) [6]. The HRSF method can apply spatial range queries for the MaxRS query processing algorithm.

4.1 Overview

The order of selecting facilities plays an important role in improving the performance of MaxRS query processing. When using the HRT for indexing the facility flat file, facilities are ordered based on their Hilbert value [6], which ensures facilities being close together in space, are processed in succession.

In Fig. 6, the values in square brackets are the Hilbert

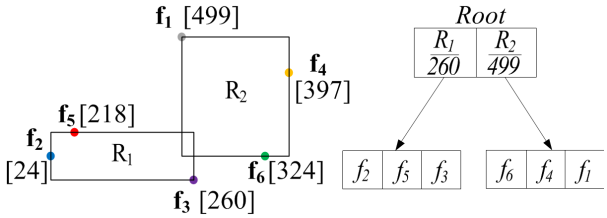

Fig. 6 An example of the HRT that indexes facilities

Table 1 The processed order of facilities

B$^+$-Tree indexing			Hilbert R-tree indexing		
Order	Buffer	I/O's	Order	Buffer	I/O's
f_1	$v_2 - v_3$	2	f_2	$v_1 - v_4$	2
f_2	$v_1 - v_4$	2	f_5	$v_2 - v_4$	1
f_3	$v_5 - v_4$	1	f_3	$v_5 - v_4$	1
f_4	$v_3 - v_6$	2	f_6	$v_5 - v_6$	1
f_5	$v_2 - v_4$	2	f_4	$v_3 - v_6$	1
f_6	$v_5 - v_6$	2	f_1	$v_3 - v_2$	1
Total		11	**Total**		7

values of facilities. R_1 and R_2 are MBRs of two sets of facilities $\{f_2, f_5, f_3\}$ and $\{f_6, f_4, f_1\}$, respectively. A non-leaf node of the HRT contains entries of the form (R, LHV, ptr), where R is the MBR that encloses all the children of that node, LHV is the largest Hilbert values among the Hilbert value of all facilities covered by R, and ptr is a pointer to the child node. A leaf node contains facilities, which are ordered by their Hilbert values.

Table 1 shows the difference of I/O cost for accessing the adjacency flat file between B$^+$-Tree facility flat file and HRT facility flat file for the road network shown in Fig. 1 (b). For simplicity, we assume the network radius is small enough (e.g., $r = 0.5$), and thus only two vertices need to be read from the adjacency flat file when processing each facility. The buffer is assumed to contain at most two vertices, and when reading one vertex from the adjacency flat file, it is assumed to take one I/O operation. In the table, (i) the *Order* columns show the processed order of the facilities, (ii) the *Buffer* columns show the vertices need to be read when processing a facility, and (iii) the *I/O's* columns show the number of I/O's for reading the adjacency flat file. The first processed facility is f_2 and two vertices $\{v_1, v_4\}$ are read from the adjacency flat file, which takes two I/O operations. The second processed facility is f_5, and two vertices $\{v_4, v_2\}$ are read, because v_4 has already been in the buffer, only v_2 is read with one more I/O operation. The HRT indexing enables to access the adjacency flat file more efficiently than the B$^+$-Tree indexing.

To reduce I/O cost, the generated segments can be inserted directly to the seg-file, so that the segments lying on the same edge are not grouped into their edge-records. In the BTSF method, which uses the B$^+$-tree for indexing the seg-file, it takes a large amount of I/O operation to find and group the segments lying on the same edge when finding the max-segments. To remedy this problem, the HRT is used to index the seg-file. With this indexing, the segments lying on the same edge are retrieved by spatial range queries. The structure of the HRT of the seg-file is similar to that of the facility flat file.

Algorithm 1 HRSFMaxRS

Input: AF: adjacency flat file, FF: facility flat file, SF: seg-file, r: network radius
Output: $maxSegs$: list segments with maximum weight
1: Initialize empty lists completedEdges, maxSegs
2: $allFacMBR$ = MBR of all facilities
3: $facCursor = FF$.query($allFacMBR$)
4: **while** ($facCursor$.hasNext()) **do**
5: $fac = facCursor$.next()
6: GenerateSegments(AF, SF, fac, r)
7: $allSegMBR$ = MBR of all segments
8: $segCursor = SF$.query($allSegMBR$)
9: **while** $segCursor$.hasNext() **do**
10: $sS = segCursor$.next()
11: **if** (sS.Edge in $completedEdges$) **then**
12: continue
13: **else**
14: $completedEdges$.add(sS.Edge)
15: $edgeRecord$ = new EdgeRecord(sS.Edge)
16: $edgeMBR$ = MBR of sS.Edge
17: $segsInEdgeCursor = SF$.query($edgeMBR$)
18: **while** ($segsInEdgeCursor$.hasNext()) **do**
19: $sQ = segsInEdgeCursor$.next()
20: **if** (sS.Edge = sQ.Edge) **then**
21: **for** each segment sE in $edgeRecord$ **do**
22: **if** (sE.facId = sQ.facId) **then**
23: $mergeSeg$ = mergeSegment(sE, sQ)
24: **if** ($mergeSeg$ is not null) **then**
25: $edgeRecord$.remove(sE)
26: $sQ = mergeSeg$
27: $edgeRecord$.add(sQ)
28: scan $edgeRecord$ for filling $maxSegs$

4.2 MaxRS Query Processing Algorithm

The detail of the MaxRS query processing algorithm, denoted by *HRSFMaxRS*, is presented in Algorithm 1. Because the facility flat file is indexed by the HRT, HRSFMaxRS issues a spatial range query to get all facilities, where the query range is the MBR of all facilities (lines 2–3). From each facility f, HRSFMaxRS generates the segments that cover the network range $f(r)$ and inserts these segments to the seg-file by calling function $GenerateSegments$ in Algorithm 4 (lines 4–6). Although the generating segments procedure is similar to the BTSF method [3], for convenience, we present the details of $GenerateSegments$ in Appendix. In HRSF method, facilities are, however, processed in the increasing order of their Hilbert values.

In the step of finding the max-segments, HRSFMaxRS groups all segments located on the same edge to one edge-record in order to discover the local optimal segments. HRSFMaxRS uses a spatial range query to find all segments (lines 7–8). For each segment (i.e., sS), HRSFMaxRS checks if the edge contains this segment. If the corresponding edge was processed (i.e., the edge is in the list called $completeEdges$), HRSFMaxRS checks the next segment (lines 11–12). If the edge is not processed, HRSFMaxRS adds the edge to $completeEdges$, so that it will not process the edge in the future (line 14). Then, HRSFMaxRS creates an edge-record of the edge to store all segments on the edge (lines 15–27). Firstly, HRSFMaxRS finds all segments that

located on the edge by a spatial range query (lines 16–17). Then, for each segment (i.e., sQ), if the segment is really located on the edge (note that some segments inside the MBR of the edge may not be located on the edge), HRSFMaxRS inserts the segment to the edge-record (lines 20–27). In case there are some existing segments of the same facility in the edge-record, HRSFMaxRS merges the segment to the existing segments (lines 22–26). Finally, HRSFMaxRS scans the edge-record to find the max-segments for filling the list called $maxSegs$ (See Algorithm 4 - findMaxSegments [3]). Creating edge-record from all segments lying on the same edge, and finding the local optimal max-segments from the edge-record continues until edge-records of all segments are processed.

Lemma 1: Let $S = N_F D^{R/d}$ be the total number of segments in the seg-file, where (i) N_F, (ii) D, (iii) R, and (iv) d are (i) the total number of facilities, (ii) the average degree of all vertices in the road network, (iii) the network radius, and (iv) the average length of edges in the road network, respectively. Then, the I/O complexity in HRSF method is $O(S \log_{B_{BA}} V + S \log_{B_{HRSF}} S)$, where B_{BA} is the number of vertices in one block in B$^+$-Tree adjacency flat file, V is the number of vertices in the road network (or that of objects in B$^+$-Tree adjacency flat file), and B_{HRSF} is that of segments per one block in HRSF.

Proof. The total I/O cost of HRSF method should be computed as (1) the I/O cost for reading the adjacency flat file, (2) the I/O cost for building HRSF, (3) the I/O cost for range query when processing seg-file, and (4) the I/O cost for range query in HRT facility flat file. Because (3) and (4) are very small compared to (1) and (2), we do not consider (3) and (4). In the worst case, the segments of each facility can be considered as a D-ary tree with its height being equal to R/d, and thus the number of segments generated by each facility is $O(D^{R/d})$. For (1), it takes $O(1)$ exact-match query in the B$^+$-Tree adjacency flat file to generate a new segment. Because the I/O complexity of an exact-match query in B$^+$-Tree is $O(\log_B N)$, where B is the number of objects per one block and N is the total number of objects in B$^+$-Tree [7], the I/O complexity to access adjacency flat file is $O(S \log_{B_{BA}} V)$. For (2), since the Hilbert R-tree acts like the B$^+$-Tree for insertions [8], the I/O complexity for building HRSF is $O(S \log_{B_{HRSF}} S)$. □

5. The Edge-Area Method

In this section, we propose the Edge-Area (EA) method, where the generated segments are handled in an in-memory HRT. In case the size of the generated segments is greater than the maximum size of the in-memory HRT, the exceeding segments are stored in a HRSF.

5.1 Overview

In the BTSF and HRSF methods, the segments of all facilities are generated and inserted into seg-file, after which this seg-file are processed to find the max-segments. The seg-file is accessed through the algorithms, and this is not efficient. In this section, we present another new method which minimizes the number of accesses to seg-file. The edges are processed in the order of their location. In this way, the segments of all facilities are not generated and inserted into the seg-file continuously. Particularly, only segments of some potential facilities, which is enough to find the local max-segments on a specified edge, are generated. In this situation, because the number of generated segments is not much, these segments can be stored in memory. If the size of generated segments is greater than a maximum given size, the exceed segments are stored in the HRSF.

A notion of finding all potential facilities of an edge which may have the network range overlapping the edge is found.

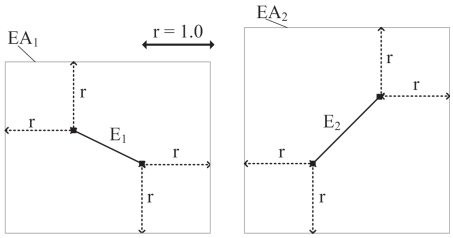

Fig. 7 Edge-Areas of edges

Definition 7: (*Edge-Area*). The edge-area EA of edge E is a rectangle such that every point p in this rectangle may have network range $p(r)$ that overlaps edge E.

Figure 7 shows two edge-areas EA_1 and EA_2 of two edges E_1 and E_2 (the gray solid rectangles). It is easy to observe that any facility f that outside of the edge-area EA_1 cannot have the network range $f(r)$ that overlaps E_1. From this idea, for getting all segments on E_1, only segments of facilities, which are stored in EA_1, are generated. This will limit the number of generated segments, so these segments are stored in memory to process. If the network radius is not long, the HRSF may not be used, and this helps to reduce the number of I/O's a lot.

5.2 The Edge-Area Based Algorithm

Algorithm 2 exhibits in details the MaxRS query processing algorithm according to the edge-area concept, denoted by *EAMaxRS*. The first processed edge is the edge that contains the most left facility in the HRT of the facility flat file (lines 3–5). For each processed edge *currEdge*, EAMaxRS gets all facilities that are located in the edge-area *edgeArea* of *currEdge* (lines 7–8). Then, EAMaxRS generates segments of these facilities and stores the segments (lines 10–13). EAMaxRS only generates segments of facilities which have not been processed (i.e., the facilities are not in the list called $finishedFacs$) (lines 11–12). The generated segments lying on the same edge are grouped into one edge-record, and the edge-record is inserted into the in-memory HRT or the

Algorithm 2 EAMaxRS

Input: AF: adjacency flat file, FF: facility flat file, $HRSF$: an empty HRSF, r: network radius
Output: $maxSegs$: list segments with maximum weight
1: Initialize an empty Hilbert R-Tree $memoryHRT$
2: Initialize empty lists $finishedFacs$, $maxSegs$
3: $facMBR$ = MBR of all facilities
4: $facCursor$ = FF.query($facMBR$)
5: $currEdge$ = $facCursor$.next().Edge
6: **while** ($finishedFacs$.size() < number of facilities) **do**
7: $edgeArea$ = getEdgeArea($currEdge$)
8: $facEACur$ = FF.query($edgeArea$)
9: **while** ($facEACur$.hasNext()) **do**
10: fac = $facEACur$.next()
11: **if** (fac is not in $finishedFacs$) **then**
12: generating segments and insert to $memoryHRT$ or $HRSF$
13: $finishedFacs$.add(fac);
14: $ProcessEdge(currEdge, memoryHRT, SF, maxSegs)$
15: **if** ($memoryHRT$ is not empty) **then**
16: $currEdge$ = the edge of first edge-record in $memoryHRT$
17: **else**
18: **while** ($facCursor$.hasNext()) **do**
19: $newFac$ = $facCursor$.next()
20: **if** ($newFac$ is not in $finishedFacs$) **then**
21: $currEdge$ = $newFac$.Edge
22: break
23: **for** each $edge\text{-}record$ in $memoryHRT$ **do**
24: scan $edge\text{-}record$ for filling $maxSegs$
25: **for** each $edge\text{-}record$ in HRSF **do**
26: scan $edge\text{-}record$ for filling $maxSegs$

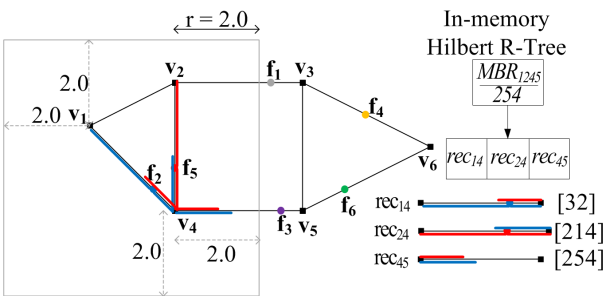

Fig. 8 The processing of the first edge

HRSF. If the size of the in-memory HRT is greater than the maximum given size, the most right edge-record in the tree is moved to the HRSF. After generating segments of a facility, this facility is inserted into $finishedFacs$ (line 13). EAMaxRS finds the max-segments on the edge-record of $currEdge$ by calling the function $ProcessEdge$ in Algorithm 3 (line 14). Then, EAMaxRS chooses the next edge to proceed. If the in-memory HRT is not empty, EAMaxRS chooses the edge of first edge-record in the tree (lines 15–16). In case there is no edge-record in the tree, EAMaxRS finds an unprocessed facility and chooses the edge which contains this facility (lines 18–22). When all facilities are put through, EAMaxRS processes remaining edge-records in the in-memory Hilbert R-tree and the HRSF to find the max-segments (lines 23–26).

Figure 8 describes the processing of the first edge $\langle v_1, v_4\rangle$ shown in Fig. 1 (b). The edge-area of the edge $\langle v_1, v_4\rangle$ is the gray rectangle, which contains two facilities f_2 and f_5. After generating segments of f_2 and f_5, there are three edge-records of three edges: rec_{14} of $\langle v_1, v_4\rangle$, rec_{24} of $\langle v_2, v_4\rangle$, and rec_{45} of $\langle v_4, v_5\rangle$. Each edge-record stores two segments

Algorithm 3 ProcessEdge

Input: E: edge will be processed, $memoryHRT$: in-memory HRT, $HRSF$: HRT segfile, $maxSegs$: current max-segments
1: $edgeMBR$ = MBR of edge E
2: $HRTCursor$ = $memoryHRT$.query($edgeMBR$)
3: **while** ($HRTCursor$.hasNext) **do**
4: $currRec$ = $HRTCursor$.next()
5: **if** ($currRec$.Edge = E) **then**
6: scan $edgeRecord$ for filling $maxSegs$
7: $memoryHRT$.remove($currRec$);
8: **if** ($HRSF$ is not empty) **then**
9: $sfMBR$ = MBR of all record in $HRSF$
10: $HRSFCursor$ = $memoryHRT$.query($sfMBR$)
11: **while** ($HRSFCursor$.hasNext) **do**
12: rec = $HRSFCursor$.next()
13: **if** (size(records $memoryHRT$ + rec) ≤ max size) **then**
14: $memoryHRT$.insert(rec)
15: $HRSF$.remove(rec)
16: **else**
17: break
18: return
19: $HRSFCursor$ = $HRSF$.query($edgeMBR$)
20: **while** ($HRSFCursor$.hasNext) **do**
21: $edgeRecord$ = $HRSFCursor$.next()
22: **if** ($edgeRecord$.Edge = E) **then**
23: scan $edgeRecord$ for filling $maxSegs$
24: $HRSF$.remove($currRec$)
25: break

(Note: two blue segments on $\langle v_1, v_4\rangle$, and two red segments on $\langle v_2, v_4\rangle$ are merged into one segment). These edge-records are organized in the in-memory HRT. In this example, for simplicity, we assume the size of an edge-record is the number of its segments and the in-memory HRT can store up to 6 segments. The structure of the in-memory HRT is similar to that of the facility flat file. In the figure, the values in square brackets on the left of the edge-records are their Hilbert values, MBR_{1245} is the MBR of nodes v_1, v_2, v_4, v_5, which forms $\langle v_1, v_4\rangle$, $\langle v_2, v_4\rangle$, and $\langle v_4, v_5\rangle$. The first edge-record (i.e., rec_{14}) in in-memory HRT is scanned to find the max-segments (returning one max-segment with the weight being equal to 2), then rec_{14} is removed from the tree.

The edge-record processing algorithm is denoted by $ProcessEdge$ in the EA method (Algorithm 3). First, ProcessEdge checks if the edge-record of current processed edge is in the in-memory HRT (lines 1–5). If the edge-record is in the in-memory HRT, ProcessEdge scans the edge-record to find the max-segments, and ProcessEdge removes the edge-record from the in-memory HRT (lines 6–7). Then, ProcessEdge moves edge-records in the HRSF to the in-memory HRT. If the total size of the first edge-record in the HRSF and all records in the in-memory HRT is less than or equal to the maximum given size, ProcessEdge moves the first edge-record into the in-memory HRT. This movement continues until the size of all edge-records in the in-memory HRT reach the maximum given size (lines 8–18). Second, if the edge-record of current processed edge is in the HRSF, ProcessEdge gets and scans the edge-record to find the max-segments, then removes it from the HRSF (lines 19–25).

Figure 9 shows an example of the HRSF in the EA method. From the example in Fig. 8, after processing the first edge $\langle v_1, v_4\rangle$, the second processed edge is $\langle v_2, v_4\rangle$. There is no unprocessed facility in the edge-area of $\langle v_2, v_4\rangle$,

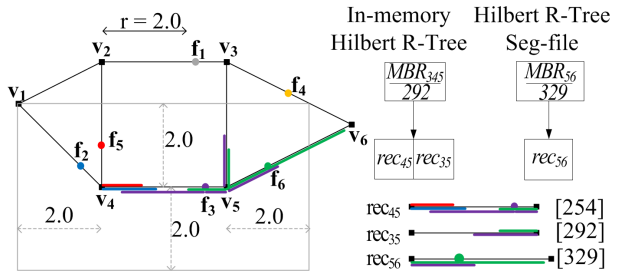

Fig. 9 An example of the HRSF in EA method

its edge-record (i.e., rec_{24}) is scanned to find the max-segments (returning one max-segment with the weight being equal to 2, at this time, there are two temporary max-segments). The third processed edge is $\langle v_4, v_5 \rangle$. Two unprocessed facilities in the edge-area of $\langle v_4, v_5 \rangle$ are f_3 and f_6. After generating segments from f_3 and f_6, there are three edge records: rec_{45} (4 segments), rec_{35} (2 segments) and rec_{56} (2 segments). Because the maximum number of segments of in-memory HRT is 6, the most right edge-record rec_{56} is moved to the HRSF. After processing rec_{45} (returns a new max-segment with the weight being equal to 3), rec_{56} is moved to the in-memory HRT (See Algorithm 3). The next processed edge is $\langle v_3, v_5 \rangle$, and the procedure will continue until all edges are done.

6. Performance Evaluation

In this section, we evaluate and compare the performance of three proposed methods: the B$^+$-Tree seg-file method (BTSF), the Hilbert R-Tree seg-file method (HRSF) and the Edge-Area method (EA). All methods were implemented in Java. We employ the B$^+$-Tree and HRT from *XXL* library [9]. The simulations were conducted on Intel Xeon E5-2520 6-core Processor with 8*GB* RAM running on the Linux system.

6.1 Simulation Setup

We used the real road network of San Francisco [10], which has 174956 nodes and 223001 edges. We generated facilities over this road network. Because the facility locations affect the performance, the facilities were generated under uniform distribution and Gaussian distribution. We normalized the range of coordinates to [0, 1000000].

In our simulations, the performance metric is execution time. Although I/O cost is a main factor for evaluation performance of an index, execution time is more intuitive than I/O cost to understand the performance.

We list the set of used parameters and their default values (stated in boldface) in the simulations in Table 2. In EA method, we fixed the size of the in-memory HRT to 2MB. In each simulation, we evaluated the effect of one parameter while the others were fixed at their default values.

Table 2 Simulation parameters and their values

Parameter	Value used (**Default**)
Cardinality of facilities	5000 – 25000 (**12500**)
Block size	0.5K – 8KB (**4KB**)
Buffer size	128KB – 2048KB (**1024KB**)
Network radius	1000 – 5000 (**2500**)

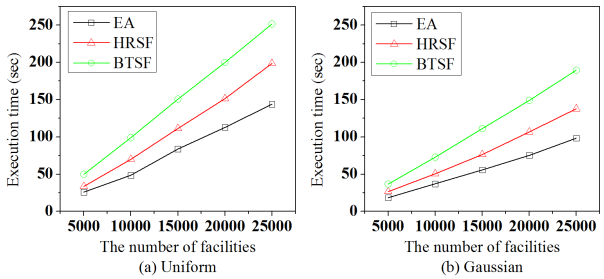

Fig. 10 Effect of the number of facilities

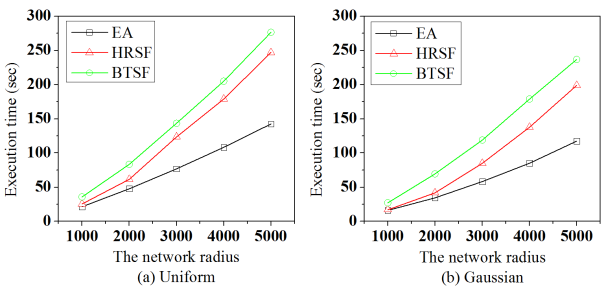

Fig. 11 Effect of the network radius

6.2 Simulation Results

6.2.1 Effect of the Number of Facilities

In this simulation, we varied the cardinality of the facilities from 5000 to 25000. As the number of facilities increases, the execution time increases in all methods (Fig. 10). Obviously, when the number of facilities increases, the number of generated segments also increases, so it takes more time to read the adjacency flat file, build the seg-file and process to find max-segments. Both of the results of Gaussian distribution and uniform distribution suggests that BTSF performs worst, while EA performs best. As shown in the figure, the execution time of all methods in Gaussian distribution of facilities is less than that in uniform distribution. This is because, in Gaussian distribution, the facilities are distributed around a center point, then the methods can utilize the buffer more efficiently than the case of uniform distribution (similar to the ideas in Sect. 3).

6.2.2 Effect of the Network Radius

Figure 11 shows the results of varying network radius (network range) from 1000 to 5000. When the network radius increases, the number of generated segments increases, and

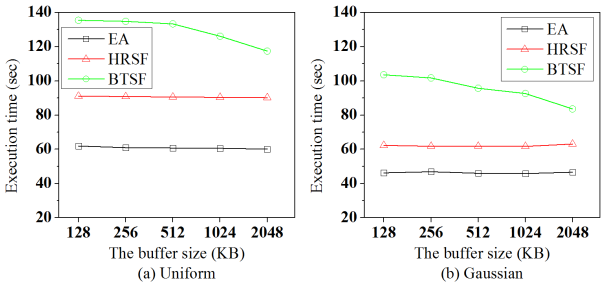

Fig. 12 Effect of the buffer size

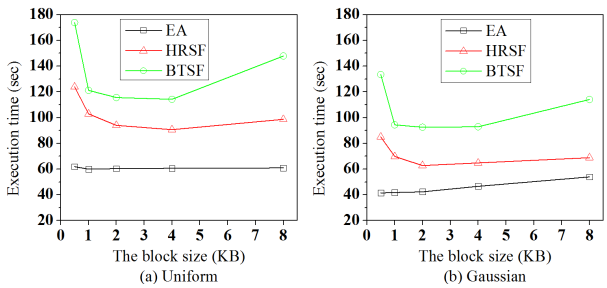

Fig. 13 Effect of the block size

thus the execution time also increases. As shown in the figure, HRSF performs better than BTSF for uniform distribution and Gaussian distribution. This is because, in BTSF, when a segment is inserted into seg-file, BTSF finds the edge-record that contains the segment, updates the segment list of the edge-record and writes back the edge-record to the seg-file. This insertion takes time to read/write the seg-file. However, the performance of HRSF is not as effective as EA. This due to the fact that, the HRSF method accesses the seg-file through the algorithms, while the EA method minimizes the number of accesses to seg-file. This causes the execution time of EA to be less than that of HRSF.

6.2.3 Effect of the Buffer Size

In this simulation, we increased the buffer size from 128KB to 2048KB, then studied the effect of the buffer size on the execution time of BTSF, HRSF and EA (Fig. 12). As the buffer size increases, the number of I/O's decreases, leads to the reduction of the execution time. HRSF and EA perform better and are less sensitive to this parameter than BTSF. It is also clear that the execution time of all methods in Gaussian distribution is less than that in uniform distribution for the reason mentioned in the description of the first simulation.

6.2.4 Effect of the Block Size

Finally, we investigated how the block size affects the performance of BTSF, HRSF, and EA by increasing the block size (from 0.5KB to 8KB). As the block size increases, the number of objects stored in a block also increases, and thus, the number of I/O's obviously decreases. However, when the block size increases, the number of block in the buffer decreases and the time to search an object in a block size also increases. As a result, the execution cost increases when the block size is too large (Fig. 13). It is notable that EA outperforms BTSF and HRSF in uniform distribution and Gaussian distribution. EA is also less sensitive to block size than HRSF and BTSF because of its small seg-file. Similarly to other simulations, the execution time of all methods in Gaussian distribution is less than that in uniform distribution.

7. Related Work

In this section, we review related work on facility optimization location problem in general and MaxRS problem in particular.

Facility optimization location problem. MaxRS problem can be seen as an instance of facility location optimization problem, which has been studied extensively in current years. The aim of this facility location optimization problem is to find an optimal location in order to maximize/minimize an objective function. Cabello et al. [11] introduced and investigated optimization problems according to the *Bichromatic Reverse Nearest Neighbor* (BRNN) rule, while Wong et al. [12] studied a related problem called *MaxBRNN*: finding an optimal region that maximizes the size of BRNNs. These two problems studied in L_2 space. Du et al. [13] proposed the optimal-location query returns a location with maximum influence, where the influence of a location is the total weight of its RNNs. Zhang et al. [14] proposed and solved the min-dist optimal-location query, an extension version of optimal-location query. Xiao et al. [15] have studied about optimal location queries in road network, with the introduction of three important types of optimal location queries: *competitive location query*, *MinSum location query* and *MinMax location query*. However, the goal of these works is to find a location that is far from the competitors and close to customers. This is different from the MaxRS problem, since MaxRS does not consider any competitors, it aims at finding a location with the maximum number of objects around.

MaxRS problem. Imai and Asono proposed an optimal algorithm for the max-enclosing rectangle problem [2] with the time complexity is $O(n \log n)$, n is the number of rectangles. Nandy and Bhattacharya [4] also presented another algorithm which is based on interval tree data structure with the same cost. Those methods are internal memory algorithms. Choi et al. [1] proposed an algorithm for solving MaxRS problem in the case of external memory with optimal I/O's. Choi et al. [16] also extended the MaxRS problem to be more fundamental, namely AllMaxRS, so that all the locations with the same best score can be retrieved. Another version of MaxRS problem is *maximizing circular range sum* (MaxCRS) problem. This is a circle version of MaxRS problem with a circle boundary. As max-enclosing

circle problem is *3SUM-HARD* [17], which the best algorithm takes $O(n^2)$ time. Choi et al. [1] also proposed the MaxCRS problem by a novel reduction that converts the MaxCRS problem to the MaxRS problem. However, all of these studies aim at Euclidean spaces, while our work investigates the MaxRS problem in road networks.

8. Conclusions

The MaxRS problem can be used in location-based applications to find the most profitable service place or the most serviceable place. All of the previous studies are stated in Euclidean distance. However, in many location-based applications, the network distance is used instead of Euclidean distance. This paper introduced two methods of solving the MaxRS problem in a road network database. We proposed external-memory algorithms, which is suitable for a large dataset of road networks.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education(NRF-2013R1A1A2008578) && This research was supported by the ICT R&D program of MSIP/IITP (1391105003).

References

[1] D.-W. Choi, C.-W. Chung, and Y. Tao, "A Scalable Algorithm for Maximizing Range Sum in Spatial Databases," Proc. VLDB, vol.5, no.11, pp.1088–1099, 2012.
[2] H. Imai and T. Asano, "Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane," J. Algorithms, vol.4, no.4, pp.310–323, 1983.
[3] T.K. Phan, H.R. Jung, and U.M. Kim, "An Efficient Algorithm for Maximizing Range Sum Queries in a Road Network," The Scientific World Journal, Volume 2014.
[4] S.C. Nandy and B.B. Bhattacharya, "A unified algorithm for finding maximum and minimum object enclosing rectangles and cuboids," Comput. Math. Appl., vol.29, no.8, pp.45–61, 1995.
[5] M.L. Yiu and N. Mamoulis, "Clustering Objects on Spatial Network," SIGMOD, pp.443–454, 2004.
[6] I. Kamel and C. Faloutsos, "Hilbert R-tree: An improved R-tree using fractals," Proc. VLDB, pp.500–509, Sept. 1994.
[7] D. Zhang, "B Tree," in Handbook of Data Structures and Applications, ed. D.P. Mehta and S. Sahni, pp.15-1–15-21, Chapman and Hall/CRC, 2004.
[8] Y. Manolopoulos, A. Nanopoulos, A.N. Papadopoulos, and Y. Theodoridis, "The Hilbert R-tree," in R-Trees: Theory and Applications, pp.20–22, Springer-Verlag, London, 2006.
[9] J.V.d. Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger, "XXL - A library approach to supporting efficient implementations of advanced database queries," Proc. VLDB, pp.39–49, 2001.
[10] T. Brinkhoff, "A framework for generating network-based moving objects," GeoInformatica, vol.6, no.2, pp.153–180, 2002.
[11] S. Cabello, J.M. Díaz-Báñez, S. Langerman, C. Seara, and I. Ventura, "Reverse facility location problems," CCCG, pp.68–71, 2005.
[12] R.C.-W. Wong, M.T. Özsu, P.S. Yu, A.W.-C. Fu, and L. Liu, "Efficient method for maximizing bichromatic reverse nearest neighbor," Proc. VLDB, vol.2, no.1, pp.1126–1137, 2009.
[13] Y. Du, D. Zhang, and T. Xia, "The optimal-location query," Advances in Spatial and Temporal Databases, Lecture Notes in Computer Science, vol.3633, pp.163–180, Springer Berlin Heidelberg, 2005.
[14] D. Zhang, Y. Du, T. Xia, and Y. Tao, "Progressive computation of the min-dist optimal-location query," PVLDB 2006, pp.643–654.
[15] X. Xiao, B. Yao, and F. Li, "Optimal location queries in road network databases," 2011 IEEE 27th International Conference on Data Engineering, pp.804–815, 2011.
[16] D.-W. Choi, C.-W. Chung, and Y. Tao, "Maximizing Range Sum in External Memory," ACM Trans. Database Syst., vol.39, no.3, pp.1–44, 2014.
[17] A. Gajentaan and M.H. Overmars, "On a class of $O(n^2)$ problems in computational geometry," Computational Geometry, vol.5, no.3, pp.165–185, 1995.

Appendix: Generating Segments Algorithm

The generating segments algorithm is detailed in Algorithm 4, denoted by $GenerateSegments$. For each facility f, GenerateSegments generates the segments, which cover the overall network range $f(r)$. First of all, GenerateSegments retrieves the information of the edge that contains f, start node (i.e., *startN*), and end node (i.e., *endN*) (lines 2–4). Then, GenerateSegments generates the segments at the start node side first (lines 6–15), after which GenerateSegments generates the segments at the end node side (lines 16–23). If the distance between f and the start node is greater than or equal to the network radius r, GenerateSegments only generates one segment with the length being equal to r (lines 7–10). On the contrary, GenerateSegments generates the segment between f and the start node (the length is equal to the offset of facility, lines 12–13) and continuously generate segments from the start node with the remaining network radius by calling the function $RecursiveGenerateSegs$ (line 14), which is described in Algorithm 5. GenerateSegments does the same way to generate segments at the end node side (new offset is the length from f to the end node, line 15). Each new generated segment has the format of the form $\langle start, end, weight, facId, edge \rangle$, where *start* is the start node of the segment, *end* is the end node of the segment, *weight* is the weight of the facility f that generates the segment, $facId$ is the identifier of f, and *edge* is the edge that contains the segment. The identifier $facId$ will help the merging process when there is more than one segment of f generated in one edge. The generated segments are inserted directly to the seg-file (with the HRT indexing). This insertion is different from the BTSF method, where the segments on the same edge are grouped into one edge-record. The insertion of HRSF method helps reduce the number of I/O's when updating edge-records in the seg-file, as discussed in Sect. 4.1. GenerateSegments uses a list, called $finishedEdges$, to store edges, which were processed completely (line 5). The edges in $finishedEdges$ will not be processed during the invocation of the function RecursiveGenerateSegs. After generating the segments of f has finishes, GenerateSegments clears $finishedEdges$ to start generating the segments of a new facility (line 26).

Algorithm 4 GenerateSegments

Input: AF: adjacency flat file, SF: seg-file, f: facility, r: network radius
1: Initialize the list $finishedEdges$
2: $startN = AF$.getNode(f.startId)
3: $endN = AF$.getNode(f.endId)
4: $edge = \langle startN, endN \rangle$
5: $finishedEdges$.add($edge$)
6: create node fN at facility location
7: **if** (f.offset $\geq r$) **then**
8: create node nN between fN and $startN$, dist(fN, nN) = r
9: $newS$ = new Segment(fN, nN, f.weight, f.Id, $edge$)
10: insert $newS$ into SF
11: **else**
12: $newS$ = new Segment(fN, $startN$, f.weight, f.Id, $edge$)
13: insert $newS$ into SF
14: RecursiveGenerateSegs(SF, $startN$, $r - f$.offset, f)
15: $endOff = edge$.length $- f$.offset
16: **if** ($endOff \geq r$) **then**
17: create node nN between fN and $endN$, dist(fN,nN) = $endOff$
18: $newS$ = new Segment(fN, nN, f.weight, f.Id, $edge$)
19: insert $newS$ into SF
20: **else**
21: $newS$ = new Segment(fN, $endN$, f.weight, f.Id, $edge$)
22: insert $newS$ into SF
23: RecursiveGenerateSegs(SF, $endN$, $r - endOff$, f)
24: $finishedEdges$.clear()

Algorithm 5 RecursiveGenerateSegs

Input: SF: seg-file, $curN$: the node (vertex) is processed, $newR$: new network radius from this node, f: original facility
1: $neighList = curN$.getNeighborList()
2: **for** each $neighN$ in $neighList$ **do**
3: $edge = \langle curN, neighN \rangle$
4: **if** ($edge$ not in $finishedEdges$) **then**
5: **if** ($edge$.length $\geq newR$) **then**
6: create node nN between $curN$ and $neighN$, dist($curN$, nN) = $newR$
7: $newS$ = new Segment($curN$, nN, f.weight, f.Id, $edge$)
8: insert $newS$ into SF
9: **else**
10: $finishedEdge$.add($edge$)
11: $newS$ = new Segment($curN$, $neighN$, f.weight, f.Id, $edge$)
12: insert $newS$ into SF
13: RecursiveGenerateSegs(SF, $neighN$, $newR - edge$.length, f)
14: **else**
15: **if** ($newR - edge$.length > 0) **then**
16: RecursiveGenerateSegs(SF, $neighN$, $newR - edge$.length, f)

Algorithm 5, denoted by RecursiveGenerateSegs, describes the process of generating segments in case the network radius r is greater than the distance between f and the start node or the end node (Algorithm 4 – lines 14 and 23). The procedure is started from the start node or end node (i.e., *curN*) with the new shorten network radius (i.e., *newR*). RecursiveGenerateSegs generates segments on all edges created from the neighbor list of *curN* (line 1). For each edge created by *curN* and its neighbor node (i.e., *neighN*), RecursiveGenerateSegs checks two situations. First, the edge does not exist in $finishedEdges$ (line 4). If the length of the edge is greater than or equal to *newR*, RecursiveGenerateSegs creates a new segment between *curN* and the neighbor node (i.e., *neighN*) with its length being equal to *newR*. Then, RecursiveGenerateSegs inserts the segment into the seg-file (lines 6–8). If the length of the edge is smaller than that of *newR*, RecursiveGenerateSegs creates a new segment between *curN* and *neighN*, and insert the new segment into the seg-file, after which RecursiveGenerateSegs continuously generates segments from *neighN* with the new shorten network radius (line 13). Second, the edge existed in $finishedEdges$ (lines 14–16). If the length of the edge is smaller than that of *newR*, RecursiveGenerateSegs generates segments from *neighN* with the new shorten network radius (line 16). This process continues until the generated segments cover the network range $f(r)$ of the original facility f. Figure A· 1 shows four segments of facility f_2: s_1, s_2, s_3, and s_4. These segments are inserted into the HRT seg-file. In the figure, $MBRs_{123}$ and $MBRs_4$ are the MBR of $\{s_1, s_2, s_3\}$ and s_4, respectively.

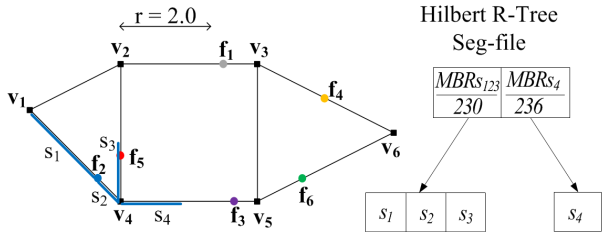

Fig. A· 1 Generating segments of a facility

Tien-Khoi Phan received the B.S. and M.S. degrees in Computer Science and Engineering from Ho Chi Minh City University of Technology, Viet Nam in 2007 and 2011, respectively. He is a Ph.D. candidate in College of Information and Communication Engineering, Sungkyunkwan University, Korea. His research interests include database system, spatial query, GIS and big data.

HaRim Jung received his B.S. degree in Computer Science from Kwangwoon University, Seoul, Korea, in 2004. He received his M.S. and Ph.D. degrees in Computer Science and Engineering from Korea University, Seoul, Korea, in 2007 and 2012, respectively. Currently, he is a research fellow at College of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea. His research interests include location-based services, spatial data management in mobile / pervasive environments and spatial big data management.

Hee Yong Youn received the B.S. and M.S. degrees in electrical engineering from Seoul National University, Seoul, Korea, in 1977 and 1979, respectively, and the Ph.D. degree in computer engineering from the University of Massachusetts at Amherst, in 1988. Currently he is a Professor of College of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea, and the Director of the Ubiquitous Computing Technology Research Institute. His research interests include distributed and ubiquitous computing, system software and middleware, and RFID/USN.

Ung-Mo Kim received the B.E. degree in Mathematics from Sungkyunkwan University, Korea, in 1981 and the M.S. degree in Computer Science form Old Dominion University, U.S.A. in 1986. His Ph.D. degree was received in Computer Science from Northwestern University, U.S.A., in 1990. Currently he is a full professor of College of Information and Communication Engineering, Sungkyunkwan University, Korea. His research interests include data mining, database security, data warehousing, GIS and big data.