Skyline Trip Planning Queries

Htoo Htoo, Yutaka Ohsawa
Graduate School of Science and Engineering, Saitama University, Japan
htoohtoo@mail.saitama-u.ac.jp ohsawa@mail.saitama-u.ac.jp

Abstract

Given a current point and a final destination, and visiting data point categories on a trip, a sequenced route query finds the shortest route which meets the specific query condition. When this query is used in real applications, the popularity of the visiting points (for example, restaurant, shopping center, etc.) in a trip is also important besides the length of the route. Therefore, a recommendation query method that evaluates the goodness by multiple criteria, such as the length and the popularity, is requested. There are some methods to cope with recommendation queries in multiple criteria. This paper proposes skyline query method for trip planning queries. The skyline query presents distinctive sequenced routes based on two criteria, the length and the popularity.

1. Introduction

Suppose that you are visiting to a strange city and you have free time for a day. Then, you make a plan to visit to a sightseeing spot, a museum, to have dinner at a restaurant, and then return back to the hotel. This type of route query is called a sequenced route (SR) query. In this case, there are many restaurants, sightseeing spots, and museums in the area, therefore, there can be a large number of candidate tour routes. On the WEB, several popular places including restaurants and museums are ranked by visitors. In such case, you may want to find an optimal route satisfying two conditions; (1) total ranking scores are high, and (2) the route length is short. The purpose of the skyline query is to select highly recommended routes for users based on their selections.

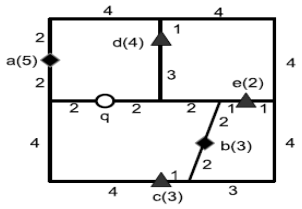

Figure 1. Example of road network distance

Figure 1 shows very simple road network and arrangement of two kinds of data points on the network. The trip assuming here is starting from q and visiting two kinds of data points marked by ♦ and ▲ in this order, and then returning to q. Each data point has popularity value attached to each data point in the parentheses. Here, popularity is evaluated in five ranks, from 1 (very poor) to 5 (very good). The numbers on each link shows the length of the link. The total trip route length can be obtained by the total of each route length for a whole trip. In this simple example, there are two data points (*a*,*b*) in the first visiting category (♦), and there (*c*,*d*,*e*) in the second category (▲). Then, there are total of six different sequenced routes. Table 1 shows them.

Table 1. Example of routes evaluation

rank	route	length	total popularity
1	$q \to a \to d \to q$	16	9
2	$q \to b \to e \to q$	14	5
3	$q \to b \to d \to q$	18	7
3	$q \to a \to e \to q$	18	7
5	$q \to a \to c \to q$	23	8
6	$q \to b \to c \to q$	18	6

Figure 2 plots the six routes in Table 1, the horizontal axis shows the total trip length and the vertical axis shows the total of the popularity values. In this figure, route 3 and 4 have the same length and the total value, therefore, they are plotted at the same point. When we compare route 5 and 1, route 5 has lower value and longer length. Therefore, when we evaluate the routes only by the value and the length, route 5 is not attractive in comparing with route 1.

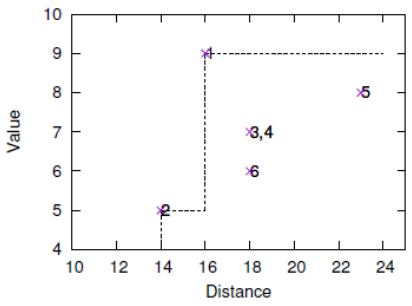

Figure 2. Length and total score

In skyline query, when route *A* is not attractive in comparing with route *B* in any evaluation criteria, we call route *A* is dominated by route *B*. In Figure 2, route 3 and 4 are dominated by route 1, and route 6 is also dominated by route 1. On the other hand, there is no domination between route 1 and 2. Because, route 2 is superior to route 1 in the length, and route 1 is superior to route 2 in the value. Skyline query presents the results only the routes that are not dominated by any other route. Therefore, in this example, route 1 and 2 are presented to the user as the result of the query.

The rest of this paper is organized as follows. Related work is presented in Section 2. In Section 3, we propose two algorithms for skyline SR queries. In Section 4, two algorithms are evaluated experimentally. Finally, this paper is concluded in Section 5.

2. Related Work

In this section, two mainly related topics, skyline queries and trip planning route queries are described.

In existing works, skyline queries have been actively researched firstly for snapshot skyline queries in two dimensions [3] and in multi-dimensions [1], [2] based on Euclidean distance. And then skyline queries for continuity [4], [5], [7] were also introduced. Sharifzadeh et al. [6] first proposed the concept of spatial skyline queries SSQ. They proposed two methods Voronoi based SSQ for static query and Voronoi based continuous SSQ. However, both methods were only worked in Euclidean distance. They then proposed SSQ for metric space [8]. As an alternative, time based skyline queries were studied by Chen et al. [9]. They proposed a framework PRISMO for processing predictive skyline queries for moving objects. Their work focused on point, range and subspace, and predictive skyline query returns the skyline of moving objects at some future time. However, prediction for time-parameterized in road network is not an easy task, and their work is not applicable in road network distances.

On the other hand, several types of trip route planning query algorithms have been proposed actively. Trip planning query (TPQ) was first proposed [13]. In their method, a TPQ finds the shortest route from a starting point to a destination by visiting each data point selected from specified data categories sets sequentially. The visiting order is not specified in this query. They proposed the minimum distance query (MDQ) algorithm for this type of query, which gives the optimal route and is adaptable to the road network distance. Lack of any restriction on the visiting order of data points categories, it requires enormous processing time when the number of data point categories to be visited is increased. The similar approach called optimal sequenced route (OSR) queries was proposed [14]. In OSR queries, the visiting order is uniquely specified, and the processing time for OSR becomes obviously lesser than the TPQ. Alternatively, Chen et al. [15] proposed a multi-rule partial sequenced route (MRPSR) query. In their query, the visiting order of data point categories is specified by a set of rules, and the computational complexity lies between TPQ and OSR queries.

Since these types of queries require long processing time, a precomputation method was proposed [16] to shorten the processing time. Moreover, skyline queries for trip planning queries have been proposed [10], [11], [12] with different aspects. In [10], they introduced path finding problem called the skyline trips of multiple POIs categories (STMPC) query which is based on two dimensions; trip total length and trip aggregates cost. Their approach is based on precomputing and storing the distance between POIs and some geographical regions in the network. Their approach is not suitable for moving objects in dynamic environments. Huang et al. [11], studied the problem of finding locations of interest which are not dominated with respect to only two attributes. Their proposed approaches to find a candidate set which is a set of skyline POIs from same category relying on nearest neighbor and range query. In [12], they proposed a method to compute skyline on routes in a road network distance for multiple preferences. However, if the road network is large, their approach degrades in efficiency. In this paper, to overcome some weak points in existing works, we propose skyline trip planning queries which are efficient in processing time, and adaptable to any size of road networks.

3. Skyline TPQ

The incremental SR query used in the skyline kSR query performed by a best first search. The basic idea of this algorithm have been proposed in [17].

Subsection 3.1 summarizes the algorithm. Then in subsection 3.2, we directly apply the algorithm to skyline queries. However, it is very time consuming especially when the search area is wide. Therefore, we modify the incremental SR search algorithm suitable for road network distance, and then apply it to skyline queries in subsection 3.3.

3.1 Incremental sequenced route query

Sets of data points are assumed to be managed by R-trees [18]. Each R-tree is indexed for each one category of data points, and therefore, M R-trees are referred when a SR query finds a route visiting M categories.

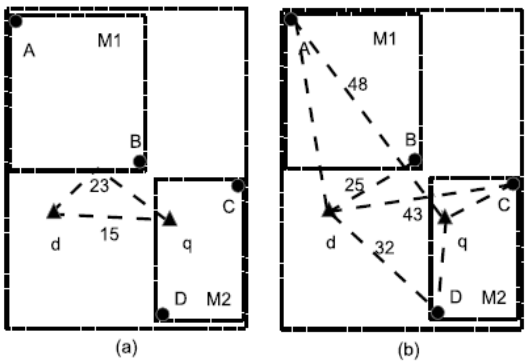

Figure 3. Lower bound route of SRT

Figure 3 shows very simple data partition in an R-tree. This R-tree manages only four data points from A to D. Firstly, a very simple SR query visiting only one category is described. *q* is a query point and *d* is a final destination. In this query, the shortest length route from *q* to *d* visiting only one data point is found. This query is performed by a best first search using a heap that manages the data format <*c*,*e*>, here *c* is the lower bound length of the found route, and *e* is a node or an entry in an R-tree. First, entry <0, Root> is inserted in the heap *H*, here Root is the root entry of the R-tree. Then, the following loop is repeated until the shortest route is found or *H* becomes empty. Initially, a record <0, Root> is deheaped from *H*. In this case, *e* is not a leaf entry. Therefore, by descending the R-tree, two records, <15, M2>, <23, M1>, are composed, and they are inserted into *H*. Here, *q* is included in the MBR (minimum bounding rectangle) of M2, therefore, the *possible lower bound route length* from *q* to *d* (more detail, see [17]). For entry M1, the possible lower bound length becomes the length shown by dash line labeled 23. After these two records are inserted, *H* contains two records <15, M2>, < 23, M1>. Next, <15, M2> is deheaped, then node M2 is visited. This is a leaf node, and there are two data points C and D. Records for these two points are created, and then are inserted into *H*. At this point, *H* contains three records <25, M1>, <32, D>, < 43, C>. Next, <25, M1> is deheaped, then M1 is visited and processed the same way as mentioned for M2. After this operation, the contents of *H* change as follows:

<25, B><32, D><43, C>< 48, A>

Next, <25, B> is deheaped, and *e* of this record is a point. At this time, a trip route $q \rightarrow B \rightarrow d$ is found, and this route is the shortest length route. This algorithm can search routes incrementally by keeping the contents of the heap till the next query.
This algorithm can be extended to general sequenced route queries, for the case when *M* is larger than one.

Algorithm 1 shows the pseudo-code. In this algorithm, the heap manages the following record,

$< c,\ell,e,i,p,v>$

Here, *c* is a lower bound of trip route length, ℓ is the distance from *q* to the current found point. *e* is a searching data point or an MBR of the R-tree managing target point set, *i* is the next category number to be found, and *p* is an array managing found visiting data points, and *v* is the total score of the data points in the constructing route. The records in the heap is ordered by *c* value.

Algorithm 1:IESR

```
1:  Function initialize (s,d)
        Data: s: start position, d: destination
2:      H.add(d_E(s,d),0, Root[1],1,0,0);
3:      return H;
4:  Function next(H,d)
        Data: H: heap, d: destination
        Result: next shortest trip route
5:      while H is not empty do
6:          r ← H.deletemin();
7:          if r.e is point then
8:              if r.i > M then return r;
9:              r.p[r.i]← r.e.p;
10:             r.v ← r.v+ r.e.v;
11:             r.l ← r.l+ d_E(r.p[r.i-1],r.p[r.i]);
12:             if r.i = M then
13:                 r.l ← r.l+ d_E(t,d);
14:                 r.p[M+1] ← d; r.i← r.i+1;
15:                 H.add(r.l, r.l, d, M+1, r.p, r.v);
16:             else
17:                 m← Root[r.i+1];
18:                 l ← r.l+D(t,d,m);
19:                 H.add(l,r.d,m,r.i+1,r.p,r.v);
20:         else
21:             n ← r.e;
22:             if n is Leaf node then
23:                 foreach v in n do
24:                     l ← d_E(r.p[r.i-1],v.e)+d_E(v.e,d);
25:                     H.add(l+r.l, r.l, v, r.i, r.p, r.v);
26:             else
27:                 foreach v in n do
28:                     l ← D(r.p[r.i-1],d,v.MBR);
29:                     H.add(l+r.l, r.l, v, r.i, r.p, r.v);
```

A trip route first searches a data point in the first visiting category. This process is the same with the above mentioned simple case. After *i*-th visiting data point has been visited, the target category is advanced to the next category. Function *initialize* (line 1 to 3) initializes the heap by inserting the initial record. Here, Root [1] is the root node of the R-tree managing the first visiting data points, and $d_E(s, d)$ shows the Euclidean distance between *s* and *d*. Function *next* returns SR incrementally by the length of the SR. Lines from 7 to 19 are for the case when *e* is a point. If the point is the destination (*d*), a complete route has been searched, and then the route is returned. Otherwise, if the point belongs to the *M*-th category, a record whose *e* is *d* is created, and inserted into *H* (from 13 to 15). Unless the search has reached to the last category, the target category will be advanced to one more. In this case, the root node of the R-tree managing $i + 1$ category data is set to *e* of the record (line 17). Here, $D(t, d, m)$ shows the *possible lower bound route length* from *t* to *d* via MBR *m*.

From line 20 to the last are the case when *e* is an MBR of an R-tree. When *e* is a leaf node of the R-tree, records for all entries (points) in the node are composed. Otherwise, when the entry is a non-leaf node, the possible minimum distance from *pnt*[*r*.*i*-1] to *d* via the MBR of *v* is calculated to compose a new record, and then inserted into *H*. This process is applied for all entries in the node. After a sequence of visiting points is found by IESR, the real route path and the length on the road network can be obtained by, for example, A* algorithm. Skyline query searches many sequenced routes, and the same point pairs are appeared many times during the skyline search. To avoid duplicated shortest path searches, once obtained shortest path between two neighboring points is recorded in a hash table. When an already existed shortest path in the hash table is requested again, the path is returned immediately. By doing this, the processing time is considerably improved.

3.2 Basic algorithm

We consider a trip starting from q, visiting M-types of data points selected one each from each data points set, and then returning back to q again. Each data point is assigned a popularity value in the range from 1 (poor) to 5 (very good). Let the route length be d and the total popularity scores be v. The purpose of the skyline query is to find all interesting routes.

A simple but time consuming solution for this query is to find all possible trip routes, and to remove dominated routes from the result. However, this method is not practical from the point of view of the processing time. To find even if one trip route is time consuming especially when M is large.

Therefore, the search area must be restricted to reduce the number of candidates. In our proposed query, the skyline routes are determined from the routes whose lengths are under $d_{min} \times \theta$, where d_{min} is the shortest route length, and θ (>1) is predetermined parameter. By this restriction, the routes with impractical long lengths are eliminated from the result.

Algorithm 2: IESR-Skyline: SkylineCand

```
1:  Function SkylineCand(s,e)
        Data: s: start position, e: destination
position
        Result: Skyline routes set
2:      initialize(s,d);
3:      pr ← netDist(next(s,d));
4:      S ← {pr};
5:      d_min ← pr.nDist;
6:      repeat
7:          pr ← netDist(next(s,d));
8:          if pr.nDist < d_min then
9:              d_min ← pr.nDist
10:         S ← S ∪ pr
11:     until pr.eDist ≥ d_min;
12:     maxD ← d_min × θ;
13:     repeat
14:         pr ← netDist(next(s,e));
15:         S ← S ∪ pr
16:     until pr.eDist > maxD;
17:     return S;
```

Algorithm 2 shows the pseudo-code to find all routes whose lengths are under $d_{min} \times \theta$. The function *nextDist* in line 3 determines the SR length in the road network distance. Line 4 adds the found route into set S. Then, line 5 extracts the length and sets it into d_{min}. From line 4 to line 8, the shortest SR is searched in Euclidean distance using IESR. However, the found route, pr, still does not guarantee as the shortest route in road network distance. The loop from line 6 to 11 finds the shortest route in road network distance.

The function *next*(s,d) in IESR returns the route in ascending order of the route lengths. Therefore, if the function is repeatedly called until the (Euclidean) length being smaller than d_{min} altering it by the found shorter length (from line 8 to 9), the shortest route can be determined. Line 12 sets the upper limit length of the searching route to *maxD*. The loop from 13 to 16 finds all routes under the limited length. The returning route set S contains all routes whose lengths are $d_{min} \times \theta$.

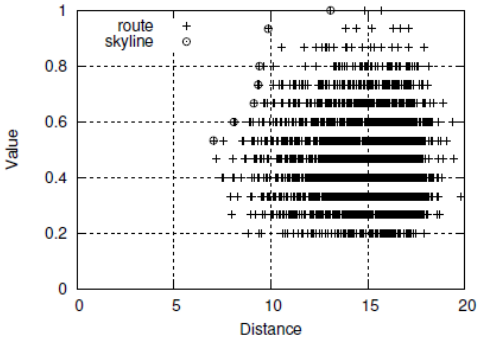

Figure 4. Skyline, θ= 2.0

Figure 4 shows an example of contents in set S where θ = 2. The horizontal axis is the length of SR, and the vertical axis is the sum of the popularity value of the routes. As shown in this figure, Algorithm 2 generates very large number of routes. Especially, when θ is large, the number also becomes huge.

Next procedure is to extract only skyline objects from the candidate set, which are shown by ○'s in Figure 4.

Algorithm 3 shows the pseudo-code. The argument route[] is an array, and the content of the array is the skyline candidates obtained by Algorithm 2. Line 2 and line 3 initialize a Boolean array which has the same size with *route*, by the value **true**. Line 4 sorts the route by the ascending order of the route lengths. As the result, the first element, route[1] is always an element of the skyline. *minPos* in line 8 keeps the position in *routearray* of the previously found skyline. In the loop from line 9 to 11, the routes that have lower score values than route[*minPos*] are removed from the skyline candidate set by altering the *alive* value to **false**. By repeating this, *alive* value of the route that is

dominated by a route is changed to **false**. From line 13 to 15, the routes having **true** value in alive array are selected from *route* array, and inserted the routes into the result set *res*. Hereafter, the procedure mentioned above is called IESR-Skyline. The processing time of the procedure is $O(N^2)$ where N is the number of the candidate routes, and this algorithm is not time efficient.

Algorithm 3: IESR-Skyline: makeSkyline

```
1:  Function MakeSkyline(route[])
        Data: route[]: array of candidate routes
        Result: Skyline routes set
2:      for i=1 to route.size do
3:          alive[i]= true;
4:      sort routes by length;
5:      for j=1 to route.size do
6:          if alive[i]= false then;
7:              continue;
8:          minPos ← j;
9:          for i= minPos +1 to route.size do
10:             if route[i].score≤
route.[minPos].score
                    then
11:                 alive[i]← false;

12:     res ←ϕ;
13:     for i=1 to route.size do
14:         if alive[i]= true then
15:             res ← res ∪ route[i];
16:     return res;
```

However, the most time consuming part in IESR-Skyline is to extract the candidates. Therefore, the ratio of the processing time of this procedure is small.

3.3 Proposed algorithm

The algorithm described in the previous subsection generates lots of candidate routes. This procedure is very time consuming. Skyline generation can make efficient if we can find the sequenced routes incrementally with the route lengths in the road network distances. By the incremental query, the first found sequenced route is guaranteed as the shortest route, therefore it can always be an element of the skyline. When the popularity value of the shortest route is v_1, the routes that give lower popularity values can be eliminated in the sequenced route search because they are dominated by the shortest route. Therefore, the further sequenced route search is performed only to the routes that can be expected to give larger popularity value than v_1.

We call the SR search algorithm proposed here INSR. INSR can be realized by slight modification of IESR. In IESR (Algorithm 1) at line 11, the partial SR length from *s* to the currently found data point is calculated in Euclidean distance. This distance calculation is changed to the road network distance in INSR. By this modification, a new SR can be obtained by ascending order of the length in road network distance. From the currently found data points r.p[r.i], the possibility to be able to obtain higher popularity scores than the currently found SR is checked in the succeeding search. If the possibility is negative, the succeeding search of the SR visiting from the currently found data point is truncated. For this purpose, the highest popularity score of the found skyline route (*lmt*) is given as an argument of *next* function.

The possibility check is performed as the following. In Algorithm 1, when a record obtained from the heap (*H*) is a data point (line 7), the i^{th} - visiting data point is determined. Line 10 adds the popularity value (*r.e.v*) to *r.v*. When the total popularity is MAXS and the total of the score until i^{th} data point is *r.v*, the expecting maximum score of the sequenced routes visiting this data point is $r.v + (M - i) \times MAXS$. If this value is less than *lmt*, the route passing this data point does not become a skyline point. Therefore, the succeeding search can be finished. By this pruning, Skyline search can be improved in processing time.

To summarize the procedure mentioned so far as follows.

- Add an argument *lmt* to function *next.*
- Line 11 of IESR is changed to the following.
  $r.l \leftarrow r.l + d_N(r.p[r.i - 1], r.p[r.i])$;
  if $lmt \geq r.v + (M - i) \times MAXS$ continue;

Here, $d_N(x,y)$ shows the road network distance between two points *x* and *y*.

Algorithm 4 shows the pseudo-code of the skyline search algorithm using above mentioned INSR.

Algorithm 4: INSR-Skyline

```
1: Function SkylineSearch (s,e)
     Data: s: start position, e: destination position
     Result: Skyline routes set
2:   initialize(s,e);
3:   S ← Ø;
4:   pr ← next(s,e,0);
5:   d_min ← pr.nDist;
6:   while pr.nDist ≤ d_min × θ do
7:     S ← S ∪ pr;
8:     pr ← next (s,e,pr.score)
9:   return S;
```

Line 4 searches the shortest SR. d_{min} in line 5 is the length of the shortest route. From line 6 to 8, sequenced routes are incrementally searched while the route length is under $d_{min} \times \theta$, and the route is added to the result set *S*.

4. Experimental results

We conducted intensive experiments to evaluate the proposed algorithm (Prop) described in section 3.3 comparing with (Basic) in section 3.2. The algorithm was implemented by Java language, and experiments were performed on Intel Core i7 4770 (3.4 GHz). We used actual road network data consisting of 16,284 nodes and 24,914 links which covers 168 km^2 area. Point data sets were generated on road network links by pseudo-random sequence with several densities. The density is defined by $p = N/L$, here N is the total number of data points in a set and L is the number of road network links.

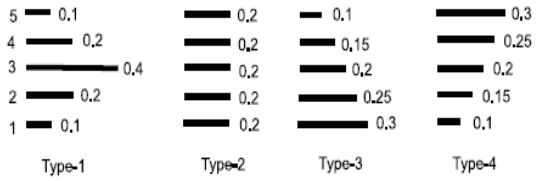

Figure 5. Distribution function of scores

Each data point has integer popularity value generated by pseudo-random sequence between 1 and 5 according to generated four types of the probability distribution functions shown in Figure 5. For example, in Type-1, the density of sore 1 and 5 are both 0.1, score 2 and 4 are both 0.2, and score 3 is 0.4.

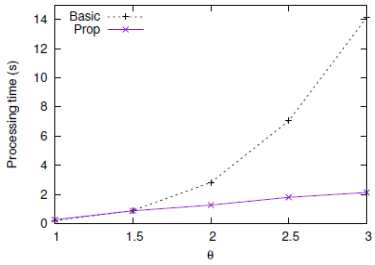

Figure 6. Processing time

Figure 6 shows the processing time of the methods in accordance with changing the value of parameter θ. This result was for Type-1 distribution. When θ is small, both methods run fast. However, the processing time of the Basic method increases rapidly in accordance with the θ increase. On the other hand, the proposed method remained short processing time even when θ = 3.

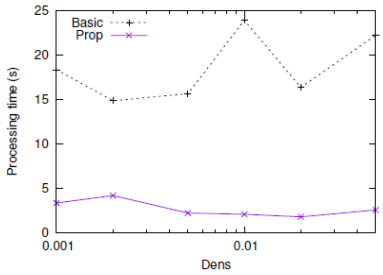

Figure 7. Processing time

Figure 7 shows the relation between the densities of the data point distribution (Dens) and the processing time when θ = 2, and the value distribution pattern is Type-1. The Basic method is not stable in processing time, however, the proposed method is stable.

5. Conclusion

This paper proposed a skyline SR query algorithm that recommends distinctive sequenced routes evaluated by two criteria, one is the route length and the other one is the total popularity score of the visiting points. The origin SR query is drastically time consuming query. In our proposed algorithm, candidate sequenced routes are incrementally searched in road network distance, and the efficiency is stably improved in terms of the processing time.

References

[1] D. Papadias et al., "An optimal and progressive algorithm for skylyine queries," in ACM SIGMOD *2003*, pp. 443–454, 2003

[2] K. Lee et al., "Approaching the skyline in Z order," in VLDB 2007, pp.279-290, 2007

[3] K. Tan et al., "Efficient progressive skyline computation," in Proc. VLDB 2001, pp. 301-310, 2001

[4] Y. Hsueh et al., "Efficient skyline search engine for continuous skyline computation," in IDCE 2011, pp. 1316-1319, 2011

[5] Y. Hsueh et al., "Efficient updates for continuous skyline computations," in DEXA 2008, pp. 419-433, 2008.

[6] M. Sharifzadeh et al., "The spatial skyline queries," in Proc.VLDB 2006, pp. 751-762, 2006.

[7] Z. Huang et al., "Continuous skyline queries for moving objects," in IEEE Trans. Knowledge and Data Engineering, 18(2), pp. 1645-1658, 2006.

[8] M. Sharifzadeh et al., "Processing spatial skyline queries in both vector space and spatial network databases," in ACM Trans. Database System, Article 4, 2009.

[9] N. Chen et al., "PRISMO: predictive skyline query processing over moving objects," in Computer and Electronics 2012, pp. 99-117, 2012.

[10] S. Aljubayrin et al., "Skyline Trips of Multiple POIs Categories", in DASFFA 2015, Part II, LNCS 9050, pp. 189-206, 2015

[11] X. Huang et al., "In-Route Skyline Querying for Location-Based Services" in W2GIS 2004, LNCS 3428, pp. 120–135, 2005

[12] H. Kriegel et al., "Route Skyline Queries: A multi-preference path planning approach," in IDCE 2010, pp.261-272, 2010.

[13] F. Li et al., "On trip planning queries in spatial databases," in *Proc. SSTD 2005*, 2005, pp. 273–290.

[14] M. Sharifzadeh et al., "The optimal sequenced route query," in VLDB, Vol 17, Issue 4, 2005, pp 765-787.

[15] H. Chen et al., "The multi-rule partial sequenced route query," in *ACM GIS '08*, 2008, pp. 65–74.

[16] M. Sharifzadeh et al., "Processing optimal sequenced route queries using Voronoi diagram," *Geoinformatica*, vol. 12, pp. 411–433, 2008.

[17] Y. Ohsawa et al., "Sequenced route query in road network distance based on incremental Euclidean restriction," in *DEXA, LNCS 7446*, 2012, pp. 484 – 491.

[18] A. Guttman ., "R-Trees: a dynamic index structure for spatial searching," in *Proc. ACM SIGMOD*, 1984, pp. 47–57.