

Flexible User-Friendly Trip Planning Queries

Bachelorarbeit
von

cand. inform. Violina Zhekova

an der Fakultät für Informatik

Erstgutachter:	Dr. Ing. Martin Schäler
Zweitgutachter:	
Betreuender Mitarbeiter:	M.-Sc. Saeed Taghizadeh

Bearbeitungszeit: 01. November 2018 – 14. April 2019

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, den 20. April 2019

Abstract

Trip planning queries are often from the type Sequenced route Queries (SRQ), a form of nearest neighbor queries, which define a starting point and a list of categories, given by the user. This type of queries are gaining significant interest, because of advances in location based mobile services and they are also of great importance in developing robust systems, where crisis management is of utter importance.

Existing approaches strive to find a best route, based on length, duration or other prime factors, passing through multiple location, called points of interest (PoIs), and they match the route perfectly. However, users may be also interested in other qualities of the route, such as the relationship among sequence points, hierarchy, order and priority of the PoIs. Therefore, in this thesis I introduce a set of operators, which the users may be interested in applying to SRQ, and propose approaches to designing and implementing some of the operators. The implementation considers metric spaces, as these are mostly relevant to the user, when working with road networks in real-life maps.

Todo: main
conclusion

Contents

Abstract	iv
Contents	1
1 Introduction	3
1.1 Motivation	3
1.2 Problem definition	4
1.3 Challenges	4
1.4 Contributions	4
2 Notations and Preliminaries	6
3 Related Work	8
4 Operators	10
4.1 Equality operator	11
4.1.1 Problem definition	11
4.1.2 Precomputations	11
4.1.3 Proposed approach	11
4.1.4 Baseline approach	19
4.1.5 Experimental study	21
4.2 Not-Equality operator	22
4.2.1 Problem definition	22
4.2.2 Precomputations	22
4.2.3 Proposed approach	22
4.2.4 Experimental study	24
4.3 Or operator	25
4.3.1 Problem definition	25
4.3.2 Precomputations	25
4.3.3 Proposed approach	25
4.3.4 Experimental study	31

4.4	Order operator	32
4.4.1	Problem definition	32
4.4.2	Precomputations	32
4.4.3	Proposed approach	32
4.4.4	Experimental study	36
5	Evaluation	37
6	Conclusion and Future work	38
	Bibliography	39

Chapter 1

Introduction

1.1 Motivation

A sequenced route query is defined as finding the shortest path from a starting point towards a possible destination, passing through multiple locations, defined by their category type. There has been significant research and proposed approaches on the topic, but there is not a developed query language to answer this types of queries. The work in this thesis has been focused on researching the topic of sequenced route queries and designing a language to enable the user to express his need in the form of a user query in a flexible manner, such as applying different constraints on the route to be found.

Example: Suppose that a user is planning a trip to town: he first wants to go to a restaurant for lunch, then he wants to stop by a bank, then he meets a friend in the shopping mall and after that he plans to have a dinner at a restaurant. In this specific scenario, the user wants to express his wish for the restaurant to be the same, because he may prefer a route where the equality of the two restaurant PoIs is more important to him than the length of the route.

With existing approaches, the user may get the shortest route [5] or all routes that satisfy the semantic similarity and length conditions equally [6], but that does not guarantee the equality of the two restaurant PoIs. Also finding k optimal routes answering the user's SRQ and then filtering out the routes where the two PoIs of type restaurant are equal has proven to not always generate a result, which is why in this thesis an optimal approach is presented.

Specific constrains such as the equality in the given example above are proposed in the thesis as operators on the query. Existing approaches have been used to transform the complex user query and changes to the approaches have been made in order to retrieve a desired result.

1.2 Problem definition

We have a starting point sp and a category sequence $M = (c_1, c_2, \dots, c_n)$, which constitutes the query, defined by the user. The constraints for this query can be applied as operators. For this query a route (r_1, r_2, \dots, r_n) , defined as a sequence of PoIs, is calculated.

Graph model: The road network or also known as spatial network is modeled as weighted graph where the crossroads are represented by nodes and roads are represented by the edges connecting the nodes. The weights on the edges in this specific research problem are the distances between the nodes on the edges. The distance between any two points can be found by summing up the lengths of the edges that belong to the shortest path between the two points.

The graph used in this paper is constructed using Berlin's spatial datasets from [1], structured in separate CSV files for the crossroads, roads and points of interest. For the implementation of the operators the datasets are imported into a graph structure of nodes and edges, where each node has a unique id, its latitude and longitude and a list of PoIs that have been mapped to it and each edge has a source and destination node and the distance between the two nodes in kilometers as parameters. Each PoI is mapped to the nearest crossroad and has a unique id, a type, its latitude and longitude and the distance to the node it is mapped to.

The map used for implementation and testing is the road network of Berlin, with 428769 crossroads, 504229 roads, 5548 PoIs and 7 category types: restaurant (2081), coffee shop (1002), atms/banks (597), movie theaters (141), pharmacies (589), pubs/bars (958), gas stations (180).

1.3 Challenges

Todo: Chal-
lenges

1.4 Contributions

Todo: Contri-
butions

The remainder of the thesis is organized as follows: First, I review the related work that has been done on the topic of SRQ in Section 2. In Section 3 I cover the proposed operators and go into details on some of them in three separate sections

for each of them: Design, Implementation and Evaluation. Finally, I conclude the thesis by summing up the progress made on the subject and discuss future work.

Chapter 2

Notations and Preliminaries

In this chapter, I would like to introduce some terms, notations and definitions that are used throughout the thesis, such as the definition for a sequenced route query (SRQ), which we need in order to define the operators.

PoIs sets: We assume that we have n sets U_1, U_2, \dots, U_n , which contain points in a 2-dimensional space \mathbb{R}^2 and $dist(.,.)$ is a distance function, which obtains the distance between two points in a two dimensional road network. The sets U_i represent the data sets for the different categories of points of interest, e.g. restaurants, gas stations etc..

Category sequence: $M = (c_1, c_2, \dots, c_l)$ is a sequence of categories, if $1 \leq c_i \leq n$ for $1 \leq i \leq l$, where n is the number of points sets U_i . The user is only allowed to ask for existing location types.

Route: $R = (r_1, r_2, \dots, r_r)$ is a route, if $r_i \in \mathbb{R}^2$ for each $1 \leq i \leq r$. R_{sp} is a route that starts from the starting point sp : $R_{sp} = (sp, r_1, r_2, \dots, r_r)$

Route length: The length of a route $R = (r_1, r_2, \dots, r_r)$ is defined as:

$$length(R) = \sum_{i=1}^{r-1} dist(P_i, P_{i+1}) \quad (2.1)$$

For $r = 1$ $length(R) = 0$.

Sequenced route: Let $M = (c_1, c_2, \dots, c_l)$ be a sequence of point of interest categories. $R = (r_1, r_2, \dots, r_l)$ is a sequenced route that follows the category sequence M , if $P_i \in U_{M_i}$ where $1 \leq i \leq l$. The points of interest in the route should belong to the corresponding category sets, defined in the category sequence.

Optimal sequenced route (OSR) query: Given a sequence of categories $M = (c_1, c_2, \dots, c_l)$ and a starting point sp in \mathbb{R}^2 , $Q(sp, M)$ is the Optimal Sequenced Route (OSR) Query, which searches for the shortest (in terms of function $length$) sequenced route R that follows M .

$$length(sp, R) = dist(sp, P_1) + length(R) \quad (2.2)$$

All other sequenced routes that follow M are referred to as candidate sequenced routes (SR).

Table 2.1 summarizes all used notations.

<i>Symbol</i>	<i>Meaning</i>
U_i	a point set for a category in \mathbb{R}^2
$ U_i $	cardinality of the set U_i
n	number of point sets U_i
$dist(., .)$	distance function in \mathbb{R}^2
M	category sequence, $= (c_1, c_2, \dots, c_l)$
$ M $	l , size of sequence M = number of items in M
c_i	i th member of M
R	route, $= (r_1, r_2, \dots, r_r)$
$ R $	r , size of route R = number of points in R
r_i	i th point in R
$length(R)$	length of R
$length(sp, R)$	length of $R_{sp} = (sp, r_1, r_2, \dots, r_r)$, $= length(R_{sp})$
$Q(sp, M)$	sequenced route query

Table 2.1: Notations

Chapter 3

Related Work

In this section I would like to review some existing research, related to the topic of this thesis. Sequenced route queries have been extensively researched and different algorithms that optimize the problem and address different use scenarios have been developed. Usually, existing approaches differentiate between vector and metric spaces, considering the Euclidean distance between geographic points or the real-life road-network-based distances accordingly. Some algorithms are focused on returning a single optimal route, where the PoIs match the given categories in the category sequence perfectly, whereas others consider semantic hierarchy or multiple route factors such as rating, distance and category weights.

In *The Optimal Sequenced Route* the researchers propose two effective algorithms for solving the sequenced route query problem. They first elaborate on why a classic shortest path algorithm such as Dijkstra would be impractical for real-life scenarios and then go on to propose the LORD (Light Optimal Route Discoverer) and R-LORD algorithm, which uses a R-tree, which are Dijkstra-based and made for vector spaces and the PNE (Progressive Neighbor Exploration) algorithm, which employs the nearest neighbour search and is designed specifically for metric spaces. Both of their proposed algorithms calculate a perfect route and only return one optimal route (while modification of the PNE algorithm also allow for finding k optimal routes), significantly outperforming Dijkstra's algorithm. [5]

A different approach to the SRQ, designed for metric spaces, is proposed in *Sequenced Route Query with Semantic Hierarchy*. The authors suggest a Skyline based algorithm, called bulk SkySR (BSSR), which searches for all preferred routes to users by extending the shortest route search with the semantic similarity of PoIs' categories. This approach expects a category tree, representing the semantic hierarchy of categories, and applies the Skyline concept, which is searching for routes that are not worse than any other routes in terms of their

scores, to the route length and semantic similarity, also known as the route scores. The BSSR algorithm also exploits the branch-and-bound concept by searching for routes simultaneously to reduce the search space. [6]

Another research article proposes the Personalized and Sequenced Route (PSR) Query, which considers both personalization and sequenced constraints. The approach takes into account multiple factors of a route, such as distance rating and associates different weight with each PoI category and a distance weight. The framework designed to obtain one optimal route consists of three phases: guessing, crossover and refinement, and is focused on spatial databases. [3]

In *In-Route Skyline Querying for Location-Based Services* queries are issued by user moving along a routes towards destinations (PoIs), also defined as query points. The movement of the user is constrained to a road network and the travel distance is considered. In-route queries know the destination and current location of the user, which dynamically changes, and the anticipated route towards the endpoint. Users can apply weights to several spatially-related criteria, when deciding on PoIs to visit next, such as the total distance difference, known as detour, and the relative distance of the current data point. [4]

An article *Sequenced Route Queries: Getting Things Done on the Way Back Home* suggest speedup techniques for sequenced route queries. A contraction hierarchy is proposed for preprocessing results for faster retrieval of answers by shortest path queries in road networks. The second technique uses the distance sensitivity of routes ("most queries are of a local kind"), which it bases on users' typical behavior. In this approach, one optimal route is returned, but queries where the order of PoIs is not necessarily fixed are possible as long as the number of PoIs remains moderate. Also, constraints on the order of visited PoIs can be made, e.g. visiting a restaurant before a shopping center. [2]

Chapter 4

Operators

In this chapter the proposed operators are covered in terms of their design, implementation and evaluation.

Conjunction, disjunction and negation: applied to some of the user-specified categories in the sequence

Order: some of the PoIs in the route must be in the given order

Hops between PoIs: defined number of PoIs between the given categories

Necessity: some of the PoIs in the route can be missing

Perfection: some of the PoIs in the route must match the user-specified category perfectly

4.1 Equality operator

The equality operator is based on the need to express that some PoIs in the SRQ of the same category can or should be equal, as given in the example in Chapter 1.

4.1.1 Problem definition

The equality operator is defined as follows:

Equality operator: Given a sequence of categories $M = (c_1, c_2, \dots, c_l)$, a starting point sp in \mathbb{R}^2 and indices i and j , where $r_i \in U_{M_i}$, $r_j \in U_{M_j}$ and $M_i = M_j$, $EQUAL(i, j)$ is an equality operator, which states that r_i and r_j in the found route $R = (r_1, r_2, \dots, r_l)$ should be the same points of interest. $Q(sp, M, EQUAL(i, j))$ is a Sequenced Route (SR) Query, which searches for the shortest (in terms of function *length*) sequenced route R that follows M and where $r_i = r_j$.

4.1.2 Precomputations

In order to faster calculate the heuristic for the partial routes, the nearest neighbors of all PoIs' categories to each node are precalculated and kept in a 2-dimensional table in memory for easy access. For precalculation a modified Dijkstra is executed for every node, which terminates as soon as it reaches the nearest neighbors of every category to a the given graph vertex.

4.1.3 Proposed approach

The equality operator is designed using the PNE approach, proposed in [5]. It uses the progressive neighbour explorer as its base to upgrade on and extends it with a heuristic approach to shrink the search space.

Heuristic

For generating the routes and deciding which of them are worth further expanding on, the proposed approach uses an initially calculated *upper bound* of an artificially build OSR, which satisfies the equality condition, and compares it to a lower bound of a route, considered by the algorithm. The *lower bound* of a certain route represents the sum of its length and its heuristic. The heuristic of a

certain PSR is the maximum distance out of the distances to PoIs from the set of categories that are yet to be expanded. 1

Procedure heuristic(R)

```

1 // Calculates the heuristic for the given route
   $R = (r_1, r_2, \dots, r_k)$ 
2 // For every route, which already contains  $r_i$ 
   $R = (r_1, r_2, \dots, r_i, \dots, r_k)$  the distance to  $r_j$  is
  calculated as the  $\text{dist}(r_k, r_i)$ 
3 for  $c_{k+1}$  to  $c_n$  do For all direct neighbors to  $r_k$  of every subsequent
  category find the maximum distance
4   | find maximum;
5 end

```

Algorithm

The algorithm for the equality operator as shown in 2 is constructed using multiple procedures.

Algorithm 1: equalityOperator

Input : $Q(sp, M = (c_1, c_2, \dots, c_l)), EQUAL(i, j)$

Output: $R = (r_1, r_2, \dots, r_l)$

```

1 initialize heap;                                // Heap with PSR
2 initialize found;                                // The candidate SR
3 initialize UB;
4 optimalRoute = PNE( $Q$ );
5 if optimalRoute[ $i$ ] = optimalRoute[ $j$ ] then
6   | optimal route has been found;
7   | return optimalRoute;
8 else
9   | dummySR();
10  | modifiedPNE();
11 end

```

In *heap* the PSR which are to be examined by the algorithm are stored. It is sorted by lower bound of the PSR. *found* stores a candidate SR. *UB* is the length of the candidate route and it is updated each time a full SR is found. The update is performed in *trim* 10

First, an optimal sequenced route is found using the PNE algorithm 9. It is checked, if the two PoIs that the user has asked to be equal, are equal in the OSR. If so, the OSR is returned, else the equality operator continues with the creation of an artificial SR *dummySR* and the modified PNE algorithm.

Second, we artificially create a sequenced route from the optimal route, found by PNE, as seen in 3. The optimal route is changed, so that r_j is made to be equal to r_i and the length of the artificially created PSR is the initial upper bound, by which later partial sequenced routes are either kept or discarded.

Procedure *dummySR(optimalRoute)*

```

1 // Creating a dummy SR (partial sequence route)
  from the found optimal route; replacing  $r_j$ 
  with  $r_i$ 
2  $dummySR = (r_1, r_2, \dots, r_{i-1}, r_i, \dots, r_i)$ ; // First part of the
  route
3  $dummySR \leftarrow add\ PNE(r_i, (c_{j+1}, \dots, c_l))$ ;
4  $UB = length(dummySR)$ ;
5 place dummySR on the heap;
```

The modified PNE algorithm 4 begins iterating all r_1 from the category set U_{M_1} , which are subsequent to *sp* in the and it compares the *lower bound*, generated by them, to the global *upper bound*. They are only considered in further steps of the algorithm, if the partial sequenced route has a smaller lower bound than the upper bound.

Algorithm 2: modifiedPNE returns Route

```

1 foreach  $r_1$  in  $U_{M_1}$  do Checking the upper bound for every  $r_1$  neighbor of
    $sp$  in the category set  $U_{M_1}$ 
2   build a new  $PSR$  with  $r_1$ ;
3    $LB = \text{length}(PSR) + \text{heuristic}(PSR)$ ;
4   if  $LB \leq UB$  then
5     place the new  $PSR(r_1)$  on the heap;
6   end
7 end
8 while heap is not empty do
9    $current = \text{fetch a } PSR \text{ from the heap}$ ;
10  switch  $s = \text{size}(current)$  do
11    case  $s \leq j - 1$  do Finding PSRs before  $r_j$ 
12       $\text{case1}()$ ;
13    end
14    case  $s = j$  do Finding PSR containing  $r_j$ 
15       $\text{case2}()$ ;
16    end
17    case  $s = j + 1$  do Finding PSR after/containing  $r_j$ 
18       $\text{case3}()$ ;
19    end
20    case  $s \geq j + 2$  do Finding PSRs after  $r_j$ 
21       $\text{case4}()$ ;
22    end
23  end
24 end
25 return  $found$ 

```

Next, the modified PNE algorithm acts as a PNE algorithm and it fetches partial sequenced routes from the heap and generates new routes. This process is repeated until the *heap* runs out of PSR, which means that all possible candidate routes have been expanded and taken into account.

At each subsequent iteration of the algorithm 4 (line 9 to 26) there are four distinct cases depending on the length of the route. Case one 5 and four 8 follow the original PNE. Case two 6 is focused on finding the travel distance between r_{j-1} and r_i . In each of the cases after fetching the route first the *lower bound* of the fetched PSR is compared to the global *upper bound* to see if the route should be modified or discarded (lines 1 to 3). After that the PSR is modified accordingly and again a length check is performed before finally putting the route on the *heap*. The length check is performed to make sure that the PSR is not longer than the already found SR with a length, which is the upper bound UB .

In case one 5 we find PSR before r_j . Two modifications of the PSR are performed, which follow the PNE algorithm 9. In a) (line 4) the nearest neighbor to the last PoI in the PSR r_k in $U_{M_{k+1}}$ is found and the PSR is updated to contain r_{k+1} and placed back on the heap. In b) (line 11) the k-th nearest neighbor to the second to last PoI r_{k+1} in U_{M_k} is found and the last PoIs in the PSR r_k is replaced with it.

Procedure caseOne

```

1  $LB = \text{length}(\text{current}) + \text{heuristic}(\text{current});$ 
2 // Heuristic check
3 if  $LB \leq UB$  then
4   a)  $\text{nearestNeighbour}(r_k, U_{M_{k+1}});$ 
5   update  $PSR$  to contain  $r_{k+1}$ ;
6   // Length check
7   if  $\text{length}(PSR) \leq UB$  then
8     | place  $PSR$  on the heap;
9   end
10 end
11 b)  $\text{kNearestNeighbour}(r_{k-1}, U_{M_k});$ 
12 update  $PSR$ ;
13 if  $\text{length}(PSR) \leq UB$  then
14   | place  $PSR$  on the heap;
15 end

```

In case two 6 r_j is to be found. In a) (line 4) instead of finding the nearest neighbor like the PNE algorithm does, the travel distance between the last PoI in the PSR r_{j-1} and r_i is found, because we want r_j to be equal to r_i in the route. In b) (line 12) the k-th nearest neighbor to the second to last PoI r_{j-2} in $U_{M_{j-1}}$ is found and

the last PoIs in the PSR r_{j-1} is replaced with it.

Procedure caseTwo

```

1  $LB = \text{length}(\text{current}) + \text{heuristic}(\text{current});$ 
2 // Heuristic check
3 if  $LB \leq UB$  then
4   a)  $\text{dist}(r_{j-1}, r_i);$ 
5   update  $PSR$  to contain  $r_i$  in the place  $j$ ;
6   // Length check
7   if  $\text{length}(PSR) \leq UB$  then
8     // Trimming part
9      $\text{trim}(PSR);$ 
10  end
11 end
12 b)  $\text{kNearestNeighbour}(r_{j-2}, U_{M_{j-1}});$ 
13 update  $PSR$ ;
14 if  $\text{length}(PSR) \leq UB$  then
15   place  $PSR$  on the heap;
16 end

```

In case three 7 r_{j+1} is to be found. In a) (line 4) the nearest neighbor to the last PoI in the PSR r_j in $U_{M_{j+1}}$ is found and the PSR is updated to contain r_{j+1} and placed back on the heap. In b) (line 12) the k-th nearest neighbor to the second to last PoI r_{j-1} in $U_{M_{j-2}}$ is expected to be found, but in our case this is r_j and we have already calculated the travel distance between r_{j-1} and r_i in case two 6, so here nothing further needs to be done.

Procedure caseThree

```

1  $LB = \text{length}(\text{current}) + \text{heuristic}(\text{current});$ 
2 // Heuristic check
3 if  $LB < UB$  then
4   a)  $\text{nearestNeighbour}(r_j, U_{M_{j+1}});$ 
5   update  $PSR$  to contain  $r_{j+1}$ ;
6   // Length check
7   if  $\text{length}(PSR) \leq UB$  then
8     // Trimming part
9      $\text{trim}(PSR);$ 
10  end
11 end
12 b) // Found in caseTwo

```

In case four 8 we find PSR after r_j . The case is similar to case one 5, except that in a) instead of directly putting the PSR on the heap, trimming 10 is performed to check if the route is a full SR and if the candidate route *found* and the upper bound *UB* must be updated (line 9).

Procedure caseFour

```

1 // Same procedure as caseOne() + trimming part to
  filter SR and update UB if needed
2  $LB = \text{length}(\text{current}) + \text{heuristic}(\text{current});$ 
3 // Heuristic check
4 if  $LB \leq UB$  then
5   a)  $\text{nearestNeighbour}(r_k, U_{M_{k+1}});$ 
6   update PSR to contain  $r_{k+1}$ ;
7   // Length check
8   if  $\text{length}(PSR) \leq UB$  then
9     // Trimming part
10     $\text{trim}(PSR);$ 
11  end
12 end
13 b)  $\text{kNearestNeighbour}(r_{k-1}, U_{M_k});$ 
14 update PSR;
15 if  $\text{length}(PSR) \leq UB$  then
16   place PSR on the heap;
17 end

```

Algorithm 3: PNE [5]

```

1 // Incrementally create the set of candidate
  routes for  $Q(sp, M)$  from starting point  $sp$ 
  towards PoI set  $U_{M_l}$ 
2 // Candidate routes are stored in a heap sorted
  by length of the routes
3 // At each iteration of PNE a  $PSR$  (partial
  sequenced route) is fetched and examined based
  on its length
4 // Trimming: There must be only one candidate SR
  on the heap
5 switch  $s = size(PSR)$  do
6   case  $s == l$  do
7      $PSR$  is the optimal route;
8     return  $PSR$ ;
9   end
10  case  $s \neq l$  do
11    a) nearestNeighbour( $r_{|PSR|}, U_{M_{|PSR|+1}}$ );
12    update  $PSR$  and put it back on the heap;
13    b) kNearestNeighbour( $r_{|PSR|-1}, U_{M_{|PSR|}}$ );
14    generate a new  $PSR$  and place it on the heap;
15  end
16 end

```

Procedure trim(PSR)

```

1 if  $size(PSR) = l$  then
2   if  $length(PSR) \leq UB$  then
3     update  $UB$ ;
4     update  $found$ ;
5   end
6 else
7   place  $PSR$  on the heap;
8 end

```

Correctness

Todo: Correct-
ness

4.1.4 Baseline approach

The baseline approach on the equality operator is entirely based on PNE by simply forcing r_i and r_j to be equal in the process of modifying the routes. In this variant, there is no heuristic and also no length checks.

Algorithm

The algorithm (shown in 11) starts by finding an optimal sequenced route with PNE, as mentioned in the proposed approach 4.1.3 and checks if r_i and r_j are already equal. If this is the case, it returns the found optimal route, otherwise it continues with the modified PNE 12.

Algorithm 4: equalityOperator-baseline

Input : $Q(sp, M = (c_1, c_2, \dots, c_l)), EQUAL(i, j)$

Output: $R = (r_1, r_2, \dots, r_l)$

```

1 initialize heap;                                // Heap with PSR
2 optimalRoute = PNE (Q);
3 if optimalRoute[i] = optimalRoute[j] then
4   | optimal route has been found;
5   | return optimalRoute;
6 else
7   | modifiedPNE-baseline;
8 end

```

Modified PNE proceeds with examining the routes on the *heap*, ordered by length, by size and modifying them according to PNE. When the current route on the heap is a full SR, then the optimal route has been found. The four cases (line 5, 13, 21 and 27) correspond to the cases in algorithm of the proposed approach, with the only difference being that no heuristic and length checks are performed.

Algorithm 5: modifiedPNE-baseline returns Route

```

1  firstPSR = nearestNeighbour (sp,  $U_{M_1}$ );
2  place firstPSR on heap;
3  current = fetch a PSR from the heap;
4  switch  $s = \text{size}(\text{current})$  do
5      case  $s \leq j - 1$  do Finding PSRs before  $r_j$ 
6          a) nearestNeighbour ( $r_k$ ,  $U_{M_{k+1}}$ );
7          update PSR to contain  $r_{k+1}$ ;
8          place PSR on the heap;
9          b) kNearestNeighbour ( $r_{k-1}$ ,  $U_{M_k}$ );
10         update PSR;
11         place PSR on the heap;
12     end
13     case  $s = j$  do Finding PSR containing  $r_j$ 
14         a) dist ( $r_{j-1}$ ,  $r_i$ );
15         update PSR to contain  $r_i$  in the place  $j$ ;
16         trim(PSR) ; // Trimming part
17         b) kNearestNeighbour ( $r_{j-2}$ ,  $U_{M_{j-1}}$ );
18         update PSR;
19         place PSR on the heap;
20     end
21     case  $s = j + 1$  do Finding PSR after/containing  $r_j$ 
22         a) nearestNeighbour ( $r_j$ ,  $U_{M_{j+1}}$ );
23         update PSR to contain  $r_{j+1}$ ;
24         trim(PSR) ; // Trimming part
25         b) // Found in caseTwo
26     end
27     case  $s \geq j + 2$  do Finding PSRs after  $r_j$ 
28         a) nearestNeighbour ( $r_k$ ,  $U_{M_{k+1}}$ );
29         update PSR to contain  $r_{k+1}$ ;
30         trim(PSR) ; // Trimming part
31         b) kNearestNeighbour ( $r_{k-1}$ ,  $U_{M_k}$ );
32         update PSR;
33         place PSR on the heap;
34     end
35     case  $s == l$  do Optimal route with equal PoIs at  $i$  and  $j$  has been
        found
36         return current;
37     end
38 end
39 return found;

```

Correctness

Todo: Correctness of baseline

4.1.5 Experimental study

Todo: Experiments

4.2 Not-Equality operator

The not-equality operator is based on the need to express that some PoIs in the SRQ of the same category shouldn't be equal.

4.2.1 Problem definition

The not-equality operator is defined as follows:

Not-Equality operator: Given a sequence of categories $M = (c_1, c_2, \dots, c_l)$, a starting point sp in \mathbb{R}^2 and indices i and j , where $r_i \in U_{M_i}$, $r_j \in U_{M_j}$ and $M_i \neq M_j$, $NOTEQUAL(i, j)$ is an equality operator, which states that r_i and r_j in the found route $R = (r_1, r_2, \dots, r_l)$ should be different points of interest. $Q(sp, M, NOTEQUAL(i, j))$ is a Sequenced Route (SR) Query, which searches for the shortest (in terms of function *length*) sequenced route R that follows M and where $r_i \neq r_j$.

4.2.2 Precomputations

The precomputations are the same as for the not equality operator in Section 4.1.2.

4.2.3 Proposed approach

The not-equality operator is designed using the PNE approach, proposed in [5]. It uses the progressive neighbour explorer as its base to upgrade on and explore all the possible optimal routes until it finds an optimal route, in which the given PoIs are different.

Algorithm

The algorithm (shown in 13) starts by initializing the heap, ordered by the length of the routes, and the first PSR (line 1, 2, 3). It then proceeds to inspect and modify the routes on the heap based on their length, until a full SR is found (line 7 to 20). Each full SR (line 8 to 11) is checked if it is a full SR. If this is the case, the found optimal SR is returned, otherwise the next PSR on the heap is fetched. If the fetched PSR is not a full SR but a partial route (line 12 to 19), then the algorithm performs a) and b) as in the PNE algorithm 9.

Algorithm 6: notEqualityOperator

Input : $(sp, M = (c_1, c_2, \dots, c_l)), NOTEQUAL(i, j)$
Output: $R = (r_1, r_2, \dots, r_l)$

```

1 initialize heap;
2 initialize candidate;
3 firstPSR = nearestNeighbour(sp,  $U_{M_1}$ );
4 place firstPSR on heap;

5 // At each iteration of PNE a PSR (partial
   sequenced route) is fetched and examined based
   on its length and it is checked
6 fetch a PSR from the heap;
7 switch s = size(PSR) do
8   case s == l do
9     PSR is the optimal route;
10    return PSR;
11  end
12  case s ≠ l do
13    a) nearestNeighbour( $r_{|PSR|}, U_{M_{|PSR|+1}}$ );
14    update PSR;
15    trim(PSR);
16    b) kNearestNeighbour( $r_{|PSR|-1}, U_{M_{|PSR|}}$ );
17    generate a new PSR;
18    trim(PSR);
19  end
20 end

```

Each time a *PSR* is generated the trim procedure 14 is performed. It is checked if the generated *PSR* is a full SR (line 1) and if that is the case (line 1 to 16) the routes gets further examined. In the case that the route satisfies the requirement for i and j to not be equal (line 2 to 8), its length is compared of that of the candidate route and if it is shorter or equal, the candidate route is updated and the sequenced route is placed on the *heap*. Otherwise (line 8 to 15) we check if j is the last index in the route and in this case (line 10 to 14) the k th neighbor of the previous PoI to the last one is found and a new *PSR* is generated. If the route doesn't satisfy the requirements for i and j to not be equal and j to be last index, it is simply discarded as not possible SR. If the generated route is still not a full SR (line 16 to 21) the *PSR* is placed on the *heap*.

Procedure trim(PSR)

```

1 if size( $PSR$ ) =  $l$  then
2   if  $PSR[i] \neq PSR[j]$  then
3     // Optimization: length check
4     if length( $PSR$ ) <= length(candidate) then
5       update candidate;
6       place  $PSR$  on the heap;
7     end
8   else
9     // In case  $j$  is the last index in the route,
       we find the  $k$ th neighbor of the previous
       PoI to the last one
10    if size( $PSR$ ) =  $j + 1$  then
11      kNearestNeighbour( $r_{|PSR|-1}, U_{M_{|PSR|}}$ );
12      generate a new  $PSR$ ;
13      trim( $PSR$ );
14    end
15  end
16 else
17   // Optimization: length check
18   if length( $PSR$ ) <= length(candidate) then
19     place  $PSR$  on the heap;
20   end
21 end

```

Correctness

Todo: Correct-
ness

4.2.4 Experimental study

Todo: Experi-
ments

4.3 Or operator

The or operator gives the user flexibility to express his need for different route alternatives. It represents the disjunction operator, usually present in query languages. In the context of a sequenced route query, the user can specify multiple categories or a list of categories, which can be disjoint. Then the query is responsible for finding the best route out of all the options that the user has specified. The best route is qualified based on length.

4.3.1 Problem definition

In order to explain the or query, first we need to define what an OR sequence is:

OR sequence: An or sequence $OR = (M_1, M_2, \dots, M_m)$ represents the disjunction of category sequences, such as $M_1 = (c_1, c_2, \dots, c_l)$. At least one of the category sequences, defined in this or sequence, must be present in the final result of the query.

The or query is defined as follows:

Or query: Given a sequence of OR sequences $S = (OR_1, OR_2, \dots, OR_n)$ and a starting point sp in \mathbb{R}^2 $Q(sp, S)$ is a Sequenced Route (SR) Query, which searches for the shortest (in terms of function *length*) sequenced route R that follows one of the possible permutations of the sequence S . For example $P = (M_a, M_b, \dots, M_z)$ is a permutation of S , where $M_a = (c_{a1}, c_{a2}, \dots, c_{al})$, $M_b = (c_{b1}, c_{b2}, \dots, c_{bl})$ and $M_z = (c_{z1}, c_{z2}, \dots, c_{zl})$ build a category sequence $M = (c_{a1}, c_{a2}, \dots, c_{al}, c_{b1}, c_{b2}, \dots, c_{bl}, c_{z1}, c_{z2}, \dots, c_{zl})$.

4.3.2 Precomputations

The precomputations are the same as for the or operator in Section 4.1.2.

4.3.3 Proposed approach

The or operator is designed using the PNE approach, proposed in [5]. It progressively inspects each option M_i from the or sequences OR_i in $S = (OR_1, OR_2, \dots, OR_n)$, compares them and continues with the best one, based on length, until it reaches a full sequenced route.

Algorithm

The algorithm (shown in 15) starts by initializing the heap, ordered by the length of the routes, and the first PSR (line 1 to 16). It then proceeds to inspect and

modify the routes on the heap based on their length, until a SR is reached. As can be seen in line 2, a PSR is build with each M_i from the first or sequences OR_1 in $S = (OR_1, OR_2, \dots, OR_n)$. The algorithm also checks for each M_i (line 5) if it contains a single category or a sequence of categories. In the first case, it finds the nearest neighbor in U_{M_i} to the starting point as the PNE algorithm would, otherwise it finds the nearest neighbor in $U_{M_i[1]}$ to the starting point and initializes the list of categories $PSR.categories$ for the PSR . From all these generated PSR the shortest one is chosen and put on the heap.

Modified PNE proceeds with examining the routes on the heap by $index(PSR)$ and modifying them according to PNE. $index(PSR)$ indicates the index of the last found or sequence OR_s in $S = (OR_1, OR_2, \dots, OR_n)$ for the fetched PSR . It is a full SR, . When the current route on the heap is a full SR, where $s = n$, which indicates the last or sequence OR_n , then the optimal route has been found (line 21 to 24). Otherwise the algorithm modifies the current PSR according to cases a) and b) as in the PNE algorithm 9 (line 25 to 30).

Algorithm 7: orOperator

Input : $(sp, S = (OR_1, OR_2, \dots, OR_n))$
Output: $R = (r_1, r_2, \dots, r_l)$

```

1 initialize heap;
2 foreach  $M_i$  in  $OR_1$  do Finding all the possible neighbors to the starting
   point
3   initialize shortestPSR;
4   // Building a new PSR with  $M_i$ 
5   if  $|M_i| = 1$  then
6     //  $M_i$  only contains one category
7     nearestNeighbour( $sp, U_{M_i}$ );
8     update shortestPSR;
9   else
10    //  $M_i$  is a sequence of categories
11    // The neighbor from the first category  $M_i[1]$ 
       is found and the rest of the categories
       from  $M_i$  are put into categories for the
       specific PSR
12    nearestNeighbour( $sp, U_{M_i[1]}$ );
13     $PSR.categories \leftarrow M_i[2..l]$ ;
14    update shortestPSR;
15  end
16 end
17  $firstPSR \leftarrow shortestPSR$ ;
18 place firstPSR on heap;

19 // At each iteration of PNE a PSR (partial
   sequenced route) is fetched and examined based
   on its length and it is checked

20 fetch a PSR from the heap;
21 //  $index(PSR)$  indicates the index of the last
   found or sequence  $OR_s$  in  $S = (OR_1, OR_2, \dots, OR_n)$ 
   for the fetched PSR. It is a full SR, when
    $s = n$ , which indicates the last or sequence
    $OR_n$ .

22 switch  $s = index(PSR)$  do
23   case  $s == n$  do
24     PSR is the optimal route;
25     return PSR;
26   end
27   case  $s \neq n$  do
28     // a)
29     modifyRouteA(PSR);
30     // b)
31     modifyRouteB(PSR);
32   end
33 end

```

In case a) 16 it is differentiated between a complete PSR and incomplete PSR. A complete PSR has an empty list of $PSR.categories$, which means that the next OR sequence can be examined (line 1 to 19). A PSR is build with each M_i from the or sequences OR_s in $S = (OR_1, OR_2, \dots, OR_n)$. The algorithm also checks for each M_i (line 6) if it contains a single category or a sequence of categories. In the first case, it finds the nearest neighbor to $r_{|PSR|}$ in U_{M_i} as the PNE algorithm would (line 6 to 10), otherwise it finds the nearest neighbor to $r_{|PSR|}$ in $U_{M_i[1]}$ and initializes the list of categories $PSR.categories$ for the PSR (line 10 to 16). From all these PSR the shortest one is chosen and put on the heap. For an incomplete PSR the next nearest neighbor to $r_{|PSR|}$ in $U_{PSR.categories[1]}$ is found (line 20 to 25). The indicator $PSR.prevPosition$ for the PSR is also updated to indicate that in $modifyRouteB(PSR)$ the k-th neighbor to the second to last PoI should be found. Trimming is also performed (see 10).

Procedure `modifyRouteA(PSR)`

```

1 if  $PSR.categories$  is empty then
2   // We continue with next OR sequence
3   foreach  $M_i$  in  $OR_s$  do
4     initialize shortestPSR;
5     // Building a new  $PSR$  with  $M_i$ 
6     if  $|M_i| = 1$  then
7       //  $M_i$  only contains one category
8        $nearestNeighbour(r_{|PSR|}, U_{M_i});$ 
9       update shortestPSR;
10    else
11      //  $M_i$  is a sequence of categories
12      // The neighbor from the first category
13      //  $M_i[1]$  is found and the rest of the
14      // categories from  $M_i$  are put into categories
15      // for the specific  $PSR$ 
16       $nearestNeighbour(r_{|PSR|}, u_{M_i[1]});$ 
17       $PSR.categories \leftarrow M_i[2..l];$ 
18      update shortestPSR;
19    end
20  end
21   $newPSR \leftarrow shortestPSR;$ 
22 else
23   // We continue to find the PoI of the next
24   // category in the  $PSR.categories$ 
25    $nearestNeighbour(r_{|PSR|}, U_{PSR.categories[1]});$ 
26   remove  $M_i[1]$  from  $PSR.categories$ ;
27   //  $PSR.prevPosition$ , when set to true indicates
28   // that in modifyRouteB( $PSR$ ) all the sequences
29   // in the OR sequence of the previous position
30   //  $OR_s$  should be traversed and checked again,
31   // otherwise the k-th neighbor to the second to
32   // last PoI is found
33    $PSR.prevPosition \leftarrow false;$ 
34   update newPSR;
35 end
36 // Trimming is also done (see 10)
37 place  $newPSR$  on heap;

```

In case b) 17 it is checked if the indicator $PSR.prevPosition$ for the PSR is set to *true* (line 1). If that is the case, it means, that the OR sequence of the previous position OR_{s-1} should be examined again (line 1 to 19). A PSR is build with each M_i from the or sequences OR_{s-1} in $S = (OR_1, OR_2, \dots, OR_n)$. The algorithm also checks for each M_i (line 5) if it contains a single category or a sequence of categories. In the first case, it finds the kth nearest neighbor $r_{|PSR|-1}$ in U_{M_i} as the PNE algorithm would (line 6), otherwise it finds the k-th nearest neighbor to $r_{|PSR|-1}$ in $U_{M_i[1]}$ and initializes the list of categories $PSR.categories$ for the PSR (line 10 to 16). From all these PSR the shortest one is chosen and put on the heap. If the indicator $PSR.prevPosition$ for the PSR is set to *false* it means that the k-th nearest neighbor to $r_{|PSR|}$ in $U_{PSR.categories[1]}$ is found (line 20 to 25). The indicator $PSR.prevPosition$ for the PSR is also updated to indicate that in $modifyRouteB(PSR)$ the k-th neighbor to the second to last PoI should be found.

Procedure modifyRouteB(*PSR*)

```

1 if PSR.prevPosition then
2   // We check the OR sequence of the previous
   position  $OR_{s-1}$ 
3   foreach  $M_i$  in  $OR_{s-1}$  do
4     initialize shortestPSR;
5     // Building a new PSR with  $M_i$ 
6     if  $|M_i| = 1$  then
7       //  $M_i$  only contains one category
8        $kNearestNeighbour(r_{|PSR|-1}, U_{M_i})$ ;
9       update shortestPSR;
10    else
11      //  $M_i$  is a sequence of categories
12      // The neighbor from the first category
        $M_i[1]$  is found and the rest of the
       categories from  $M_i$  are put into categories
       for the specific PSR
13       $kNearestNeighbour(r_{|PSR|-1}, U_{M_i[1]})$ ;
14       $PSR.categories \leftarrow M_i[2..l]$ ;
15      update shortestPSR;
16    end
17  end
18   $newPSR \leftarrow shortestPSR$ ;
19 else
20   // We find the k-th neighbor to the second to
   last PoI
21    $kNearestNeighbour(r_{|PSR|-1}, U_{M_{|PSR|}})$ ;
22   update newPSR;
23 end
24 place newPSR on heap;

```

Correctness

Todo: Correct-
ness

4.3.4 Experimental study

Todo: Experi-
ments

4.4 Order operator

The order operator gives the user the opportunity to provide the algorithm with route alternatives that satisfy his needs, from which the best can be found according to length. In the context of a sequenced route query, the user can specify single, none or multiple categories out of a sequenced category list to be in a specified by him order. Then only these categories are found in the form of a sequenced route query (SRQ) while the others are inspected prioritized and algorithmically. The query is responsible for finding the best route out of all possible permutations of the category sequence.

4.4.1 Problem definition

First we need to define what an ORDER sequence is:

ORDER sequence: An order sequence $ORDER = (i_1, i_2, \dots, i_k)$, with $k \leq l$ and $1 \leq i_i \leq l$ is a sequence of indices in a category sequence $M_1 = (c_1, c_2, \dots, c_l)$. They represent the categories at the given indices and indicate that these categories should remain in the given places in this category sequence M . The PoIs from categories, of which no indices are specified in the ORDER sequence can be placed at any other index from the remaining indices for the not ordered categories.

The order query is then defined as follows:

Order query: Given a sequence of categories $M = (c_1, c_2, \dots, c_l)$, a starting point sp in \mathbb{R}^2 and an ORDER sequence $ORDER = (i_1, i_2, \dots, i_k)$, $Q(sp, M, ORDER)$ is a Route Query, which searches for the shortest (in terms of function *length*) sequenced route R that follows partially M as defined by the ORDER sequence.

4.4.2 Precomputations

The precomputations are the same as for the order operator in Section 4.1.2.

4.4.3 Proposed approach

The order operator is designed using the PNE approach, proposed in [5]. It keeps a sequence of the not ordered categories, which is the complement of the ORDER sequence: $NOTORDERED = \overline{ORDER}$, and inspects progressively each category option for the indices out of the NOTORDERED sequence, compares them and continues with the best one, based on length, until it reaches a full sequenced route. For the categories, the indices of which are in the ordered list, the algorithm

finds them according to PNE. The NOTORDERED sequence is accordingly updated for every PSR, as it is specific to a route. Each time a PSR is build with one of the not ordered categories, it is removed from the NOTORDERED sequence for this specific route.

Algorithm

The algorithm (shown in 18) starts by initializing the heap, ordered by the length of the routes, and the NOTORDERED sequence, which is simply the complement of the ordered category list (line 1, 2). For building the first PSR (line 3 to 14), the algorithm checks if the first index is contained in the ORDER sequence and if that is the case it finds the nearest neighbor to the starting point from the first category, otherwise it builds partial routes with all possible categories for the first position, which are all the categories in the NOTORDERED list. From all these PSR the shortest one is chosen and put on the heap. It then proceeds to inspect and modify the routes on the heap based on their length, until a SR is reached.

The algorithm proceeds with examining the routes on the heap by size and modifying them according to PNE. When the current route on the heap is a full SR, then the optimal route has been found. When the current route on the heap is a full SR, where $s = l$, then the optimal route has been found (line 18 to 21). Otherwise the algorithm modifies the current PSR according to cases a) and b) as in the PNE algorithm 9 (line 22 to 27).

Algorithm 8: orderOperator

Input : $(sp, M = (c_1, c_2, \dots, c_l), ORDER = (i_1, i_2, \dots, i_k))$ **Output:** $R = (r_1, r_2, \dots, r_l)$

```

1 initialize heap;
2 initialize NOTORDERED =  $\overline{ORDER}$ ;
3 if ORDER contains 1 then
4   firstPSR = nearestNeighbour(sp,  $U_{M_1}$ );
5 else
6   foreach i in NOTORDERED do Finding all the possible neighbors
      to the starting point
7     initialize shortestPSR;
8     // Building a new PSR with  $c_i$ 
9     nearestNeighbour(sp,  $U_{M_i}$ );
10    update shortestPSR;
11  end
12  firstPSR  $\leftarrow$  shortestPSR;
13 end
14 place firstPSR on heap;
15 // At each iteration of PNE a PSR (partial
      sequenced route) is fetched and examined based
      on its length and it is checked
16 fetch a PSR from the heap;
17 switch s = size(current) do
18   case s == l do
19     PSR is the optimal route;
20     return PSR;
21   end
22   case s  $\neq$  l do
23     // a)
24     modifyRouteA(PSR);
25     // b)
26     modifyRouteB(PSR);
27   end
28 end

```

In case a) 19 it is checked if the category to be found next is part of the ORDER sequence. If this is the case it finds the nearest neighbor to the last PoI in the current PSR r_{s-1} from the category set U_{M_s} , otherwise it finds all nearest neighbors to r_{s-1} from the categories in the NOTORDERED list M_i and builds multiple partial routes, which are then compared and the shortest one is chosen and put on the heap. Trimming is also performed (see 10).

Procedure modifyRouteA(PSR)

```

1 if ORDER contains  $s$  then
2   |  $newPSR = nearestNeighbour(r_{s-1}, U_{M_s});$ 
3 else
4   | foreach  $i$  in NOTORDERED do Finding all the possible neighbors
      | out of the remaining categories in the not ordered list
5     |  $initialize shortestPSR;$ 
6     | // Building a new  $PSR$  with  $c_i$ 
7     |  $nearestNeighbour(r_{s-1}, U_{M_i});$ 
8     |  $update shortestPSR;$ 
9   | end
10  |  $newPSR \leftarrow shortestPSR;$ 
11  |  $update PSR.NOTORDERED;$ 
12 end
13 // Trimming is also done (see 10)
14 place  $newPSR$  on heap;

```

In case b) 20 an alternative PSR with the k th nearest neighbor to r_{s-2} is found. The algorithm checks again if the category to be found next is part of the ORDER sequence. If this is the case it simply finds the k th nearest neighbor to r_{s-2} from the category set $U_{M_{s-1}}$, otherwise it finds all nearest neighbors to r_{s-2} from the categories in the NOTORDERED list M_i and builds multiple partial routes, which are then compared and the shortest one is chosen and put on the heap.

Procedure `modifyRouteB(PSR)`

```

1 if  $ORDER$  contains  $s - 1$  then
2   |  $newPSR = kNearestNeighbour(r_{s-2}, U_{M_{s-1}});$ 
3 else
4   | foreach  $i$  in  $NOTORDERED$  do Finding all the possible neighbors
      | out of the remaining categories in the not ordered list
5   |    $initialize\ shortestPSR;$ 
6   |   // Building a new  $PSR$  with  $c_{i-1}$ 
7   |    $kNearestNeighbour(r_{s-2}, U_{M_{i-1}});$ 
8   |    $update\ shortestPSR;$ 
9   | end
10  |  $newPSR \leftarrow shortestPSR;$ 
11  | update  $PSR.NOTORDERED;$ 
12 end
13 place  $newPSR$  on heap;

```

Correctness

Todo: Correct-
ness

4.4.4 Experimental study

Todo: Experi-
ments

Chapter 5

Evaluation

The completeness of the operators stems from

Chapter 6

Conclusion and Future work

Bibliography

- [1] E. Ahmadi and M.A. Nascimento. Datasets of roads, public transportation and points-of-interest in Amsterdam, Berlin and Oslo. In: <https://sites.google.com/uAlberta.ca/nascimentodatasets/>, 2017.
- [2] Xuegang Huang and Christian S. Jensen. In-route skyline querying for location-based services. Department of Computer Science, Aalborg University, Denmark.
- [3] Jiajie Xu⁴ Zhiming Ding Jian Dai, Chengfei Liu. On personalized and sequenced route planning. Institute of Software, Chinese Academy of Sciences, Beijing, China, University of Chinese Academy of Sciences, Beijing, China, Department of Computer Science and Software Engineering, School of Software and Electrical Engineering, Faculty of Science, Engineering and Technology, Swinburne University of Technology, Melbourne, Australia, School of Computer Science and Technology, Soochow University, Suzhou, China, School of Computer Science, Beijing University of Technology, Beijing , China.
- [4] Stefan Funke Jochen Eisner. Sequenced route queries: Getting things done on the way back home. Universitat Stuttgart, Institut für Formale Methoden der Informatik.
- [5] Mehdi Sharifzadeh and Cyrus Shahabi Mohammad Kolahdouzan. The optimal sequenced route query. Computer Science Department, University of Southern California.
- [6] Yasuhiro Fujiwara Makoto Onizuka Yuya Sasaki, Yoshiharu Ishikawa. Sequenced route query with semantic hierarchy. Graduate School of Information Science and Technology, Osaka University, Osaka, Japan, Graduate School of Information Science, Nagoya University, Nagoya, Japan, NTT Software Innovation Center, Tokyo, Japan.