Karlsruher Institut für Technologie

Flexible User-Friendly Trip Planning Queries

Bachelorarbeit
von

cand. inform. Violina Zhekova

an der Fakultät für Informatik

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

Bearbeitungszeit: 01. November 2018 – 13. Mai 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 2. Mai 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 to be visited in a consecutive order, given by the user. This type of queries are gaining significant interest, because of advances in location based services (LBS) and they are also of great importance in developing robust geographic information system (GIS) applications (e.g. logistics and supply chain management), 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 possible operators. The implementation considers metric spaces, as these are mostly relevant to the user, when working with road networks in real-life maps. The query operators this thesis has focused on are the following: equality operator (EO), not-equality operator (NEO), or operator (OR) and order operator (ORDER). The equality operator gives the user the flexibility to build a query, where two PoIs of the same category must be equal. The not-equality answers the opposite problem. The or operator provides the user with the opportunity to give multiple variants of a SRQ, using or operands. With the order operator the user is able to assign fixed positions to one or more categories in the route query, whereas it is then no longer viewed as a sequenced route query and should thus be answered differently than a simple SRQ.

The approaches used to implement the query operators include the Progressive Neighbor Exploration (PNE) approach, presented in [11] and heuristic methods for shrinking the search space. The extensive experiments, which were done using a real-world dataset verified that the created algorithms perform as expected and outperform the baseline approaches in terms of processing time and required workspace.

Contents

Chapter 1

Abbreviations

In this chapter I introduce terms and abbreviations, often used in the context of this thesis.

Table 1.1 summarizes the used abbreviations.

Abbreviation	*Explanation*
SRQ	Sequenced Route Query
PoI	Point of Interest
OSR	Optimal Sequenced Route
TPQ	Trip Planning Query
GIS	Geographic Information System
LBS	Location Based Services
PNE	Progressive Neighbor Exploration [11]
PSR	Partial Sequenced Route
SR	Sequenced Route (referring to full SR)
CRQ	Complex Route Query
EO	Equality Operator
NEO	Not-Equality Operator
OR	Or Operator
ORDER	Order Operator

Table 1.1: Abbreviations

Chapter 2

Introduction

2.1 Motivation

Route queries present an important class of spatial queries for users to request an efficient path by specifying a source and a destination. A sequenced route query (SRQ) is defined as finding the shortest path from a starting point towards a possible destination, passing through multiple locations, defined by their category type. As an essential component, this type of route queries are widely supported by many of today's online map service providers (e.g., Google Maps, Yahoo! Maps, Bing Maps etc). There has been significant research and proposed approaches on the topic of sequenced route queries, but there has not been developed a query language to provide more flexibility to this types of queries. The work in this thesis has been focused on researching the topic of sequenced route queries in road networks 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 and rules on the route to be found.

Example:

Figure 2.1 presents a small example network with four different types of point sets, illustrated with different colors. The colored circles represent the PoIs and the lines between them represent the roads, whereas on each road there is a label with its length. In total there is two restaurants, a movie theater, two banks and two pharmacies. The starting point *sp* is represented by a rhombus.

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 movie theater and after that he plans to have a dinner at a restaurant. In this specific

scenario, the user wants to express his wish for both restaurants 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 itself.

With existing approaches, the user can get the shortest route [11] approach or all routes that satisfy the semantic similarity and length conditions equally [17], but that does not guarantee the equality of the two restaurant PoIs. Also finding k number of optimal routes, which 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 a better optimal approach is presented.

Using Figure 2.1, we see that the user query in the example above, when answered as an *Optimal Sequenced Route* (OSR) query [11], delivers the route (r_1, b_1, mt_1, r_2) with length 11 (shown with red lines in the figure), where both restaurants r_1 and r_2 are different. Our proposed approach, on the other hand, strives to find the optimal route, where the two restaurants are equal. The delivered route will not necessarily be the shortest possible sequenced route, but it will be the shortest route out of all possible routes, where the two PoIs are equal to each other. Such route in our example graph would be (r_1, b_1, mt_1, r_1) with length 12 (shown with dashed lines in the figure).

Specific constrains such as the equality in this example are proposed in the thesis as operators on the query. We modified and extended existing approaches, which answer the Optimal Sequenced Route (OSR) Query such as the *Progressive Neighbor Exploration* (PNE), proposed in [11], in order to transform the complex user query and retrieve a desired result.

2.2 Problem definition

With the rise of location based mobile services the need for a query language to answer complex route queries has emerged. To the best of our knowledge, although different approaches to answer the sequenced route query have been proposed, no one has yet identified the need for a unified query language to answer this complex route query types. What we would like to achieve with this thesis is to identify the problem, which is the need for flexibility in terms of route queries, and to propose a comprehensive solution to the problem. Through scientific research and research on the user's specific needs in this area of route queries, we have come up with several important operators that should be a part of this query language. Since the query language is entirely based on the user's requirements, which in-

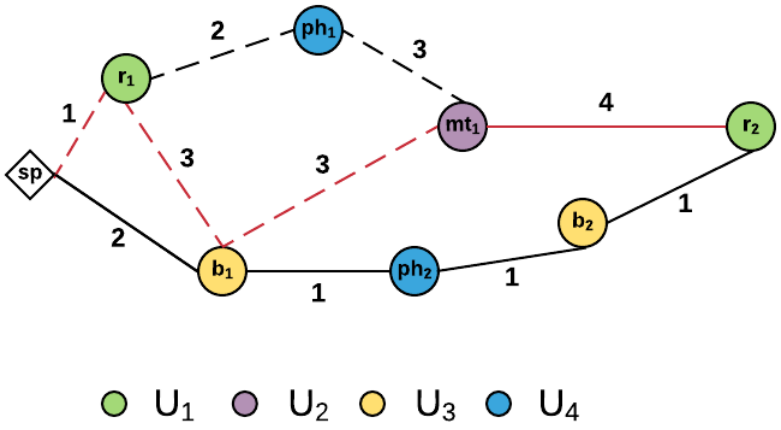

Figure 2.1: A network with four different types of point sets

clude individual personal needs, the query language operators that we propose are not a complete set. We propose four operators, that we deem the most important in terms of the level of flexibility they give to the user, and give an outlook on other ones, which can enrich the language in the future.
One of the most important comparison operators in programming and query languages is the equality operator. Therefore we decided to develop an articulate version of the equality operator for the route query type, and also developed the opposite not-equality operator in addition. Another very important programming and database language operator is the logical or operator, which gives the user the flexibility to define a subset of possible query variants that have to be filtered to obtain the best result. And lastly, we propose the order operator, which transforms the *sequenced* route query in a *partially sequenced* or *not sequenced* route query. This type of operator is closely related to the *Traveling Salesman Problem (TSP)*, which asks for the sequence in which a set of given points from a starting point is visited. The reason for choosing to design this operator is the strive for maximum flexibility. In this respect, the order operator represents the highest level of flexibility when it comes to planning a trip as a tourist with the intention of visiting multiple locations in a short period of time, without much need for order of the different types of locations.

2.3 Challenges

When working with real-life road networks, we are usually posed with the challenge of having big datasets with many PoIs. This makes it difficult to design fast algorithms for route query problems without the need for optimizations techniques. Trivial approaches usually explore the entire dataset in order to find one optimal solution, which especially in the case of route queries can increase the time significantly. If we need to check all neighbors to one PoIs of one category and do this for all categories in order to find the shortest route from a category sequence, the time in the case of the trivial approaches increases exponentially with increase in the size of the category sequence. The *Optimal Sequenced Route Query* [11] presents the PNE approach, which solves the OSR query in metric spaces efficiently. The challenge for our task is to be able to use this designed approach to implement the proposed operators.
As shown in the example in Section 2.1 the Progressive Neighbor Exploration (PNE) approach doesn't always answer the equality operator query, therefore we need to modify the algorithm to serve our needs. In order to make sure that the algorithm delivers an optimal result, all routes must be checked, where two specific PoIs of one category are equal. This means that all PoIs of this category must be somehow inspected, which increases the work space significantly, which in turn spikes the processing time of the query. This poses the need for an optimization technique, such as the heuristic approach, presented in Chapter 5.1, to narrow the search space.
The not-equality operator is an extension of the PNE algorithm, because in comparison to the equality operator, there is a greater possibility that the PNE approach finds a route with different PoIs from the same category. With a little tweak of the algorithm, it is solved by PNE without the need for an optimization technique.
Furthermore, the or operator and the order operator are faced with the challenge for exponentially growing permutations, depending on the size of the query. Therefore the trivial approaches, which simply build all permutations and perfomr the PNE approach on them, can generate a big work space and increase the time exponentially with the increase in the number of permutations. In our proposed approaches for these two operators we modify the PNE algorithm to be able to work on the possible permutations simultaneously, while pruning longer routes, which may be generated when the algorithm operates on the permutations separately. In this way we shrink the search space, while also making sure that no necessary routes are missed in the process.

2.4 Contributions

In this thesis, we introduce four different operators, which can be applied to a route query, and propose alternative approaches for solving them, which outperform the trivial solutions. The presented algorithms relate to the following operators: equality operator (EO), not-equality operator (NEO), or operator (OR) and order operator (ORDER). The equality operator addresses the need for flexibility, introduced in the example in Section 2.1, and the not-equality solves the opposite problem, in which the user specifically wants two different PoIs of the same category type in the query. Furthermore, the or operator represents the disjunction operator, usually present in query languages, and it gives the user the possibility to apply multiple category options to one or more route query points. Lastly, the order operator provides the user with the option to have a partially or fully not sequenced route query, where he can define zero or multiple categories to be at fixed positions in the route query sequence.
The algorithms for solving the operators are largely inspired by the PNE approach, presented in the *Optimal Sequenced Route Query* [11]. For answering the equality operator query, a heuristic approach is explored, as well, for shrinking the search space and reducing the processing time. For the three of the operators (EO, OR and ORDER) baseline approaches are also introduced and compared to the proposed approaches in the experiment results. Our introduced algorithms outperform the trivial solutions in the experiments with a real dataset significantly, especially with increasing the search space with the help of different evaluation parameters, which lets us to believe that they represent a meaningful solution to give more flexibility to the user for building route queries.

2.5 Outline

The remainder of the thesis is organized as follows: First, in Section 3 I introduce the notations used in the thesis, then I review the related work that has been done on the topic of SRQ in Section 4. In Section 5 I cover the proposed operators and go into details on some of them in separate sections for each of them: Motivation, Problem definition and Proposed approach. In Section 6 I present the results from the experimental studies, performed on the implemented operators and I evaluate the results. Finally, in Chapter 7 I conclude the thesis by summing up the progress made on the subject and discuss future work.

Chapter 3

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, ..., 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: $M = (c_1, c_2, ..., c_l)$ 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: $R = (r_1, r_2, ..., r_r)$ 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: $R_{sp} = (sp, r_1, r_2, ..., r_r)$

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

$$length(R) = \sum_{i=1}^{r-1} dist(P_i, P_{i+1}) \tag{3.1}$$

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

Sequenced route: Let $M = (c_1, c_2, ..., c_l)$ be a sequence of point of interest categories. $R = (r_1, r_2, ..., r_l)$ 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 $M = (c_1, c_2, ..., c_l)$ 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.

$$length(sp, R) = dist(sp, P_1) + length(R) \tag{3.2}$$

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

Table 3.1 summarizes all used notations.

Symbol	*Meaning*
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, $= (c_1, c_2, ..., c_l)$
$\|M\|$	l, size of sequence M = number of items in M
c_i	ith member of M
R	route, $= (r_1, r_2, ..., r_r)$
$\|R\|$	r, size of route R = number of points in R
r_i	ith point in R
$length(R)$	length of R
$length(sp, R)$	length of $R_{sp} = (sp, r_1, r_2, ..., r_r)$, $= length(R_{sp})$
$Q(sp, M)$	sequenced route query

Table 3.1: Notations

Chapter 4

Related Work

In this section I would like to review some existing research, related to the topic of this thesis. Route queries have been extensively researched and various algorithms that optimize the problem and address different use scenarios have been developed. Usually, existing approaches differentiate between vector and metric spaces, considering either the Euclidean distance between geographic points or the real-life road network distances. Some algorithms are focused on obtaining 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 and retrieve multiple possible routes. There is also the distinction between Trip Planning Queries (TPQ) [3] and Optimal Sequenced Route (OSR) Queries [11]. While TPQ searches for the optimal route from a starting point towards a destination, passing through a set of Point of Interest (PoI) types without specified order, the OSR query has a starting point and a predefined order (sequence) on the categories of PoIs, which are to be visited. For the obtaining of optimal route, different route parameters may be considered such as route length, travel duration, ranking of the PoIs or dynamic traffic information.

In *The Optimal Sequenced Route* [11] the researchers propose two effective algorithms for solving the Optimal Sequenced Route (OSR) 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 an R-tree structure, for vector spaces and the PNE (Progressive Neighbor Exploration) algorithm, which employs the nearest neighbor search and is designed specifically for metric spaces. Both of their proposed algorithms calculate a perfect route and return one optimal route (while modification of the PNE algorithm also allow for finding

k optimal routes), significantly outperforming Dijkstra's algorithm. The authors of *Processing Optimal Sequenced Route Queries Using Voronoi Diagrams* [9] extended this research and proposed a pre-computation approach in both vector and metric spaces to answer the OSR query by taking advantage of a family of AW(additively weighted)-Voronoi diagrams for different POI types. The approach ultimately outperforms the previous index-based approaches in terms of query response time.

Another solution to solving the OSR query with included destination point is demonstrated in *Sequenced Route Query in Road Network Distance Based on Incremental Euclidean Restriction* [16], where a framework based on the incremental Euclidean restriction (IER) approach is employed, which searches candidates in the Euclidean distance first, and then verifies the results of the road network distance. Compared to PNE, the approach performs better with densely distributed PoIs or when the number of POI categories to be visited during the trip is large.

A different approach to the SRQ, designed for metric spaces, is proposed in *Sequenced Route Query with Semantic Hierarchy* [17]. The authors suggest a Skyline based algorithm, called bulk SkySR (BSSR), which searches for all possible optimal routes 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, proposed in [12], which is based on searching for routes that are not worse than any other routes in terms of their scores - route length and semantic similarity. The BSSR algorithm also exploits the branch-and-bound concept by searching for routes simultaneously to reduce the search space.

Another research work proposes the Personalized and Sequenced Route Query [6], which considers both personalized 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.

In *In-Route Skyline Querying for Location-Based Services* [8] queries are issued by the 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. However, this approach issues continuous skyline queries in search for a single PoI category, and it is

therefore not applicable to SkySR queries, which obtain routes that pass through multiple PoIs at a time.

Furthermore, *A Continuous Query System for Dynamic Route Planning* [10] researches the problem of answering continuous route planning queries in a road network environment, while taking the delay of updates into account, and aim to obtain the shortest path.

An attempt to tackle interactive route search is presented in *An Interactive Approach to Route Search* [14]. The user can move from a starting location towards a destination, while making search queries for points of different types on the way. Upon arrival at venues, the user gives feedback specifying whether the venue satisfies its corresponding query. The proposed heuristic algorithms compute the next object to be visited, based on the feedback.

Another article *Sequenced Route Queries: Getting Things Done on the Way Back Home* [5] suggests 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.

The authors of *On Trip Planning Queries in Spatial Databases* [3] proposed approximation algorithms for Trip Planning Queries (TPQ) in a metric space. With TPQ, the user specifies a set of POI types and asks for the optimal route from her starting location to a specified destination which passes through exactly one POI in each POI type. In comparison to an OSR query, there is no order imposed on the types of POIs to be visited in a TPQ query. A multi-type nearest neighbor (MTNN) query solution was proposed in *Exploiting a Page-Level Upper Bound for Multi-Type Nearest Neighbor Queries* [13]. Given a starting point and a set of location categories, a MTNN query finds the shortest path for the query point such that only one instance of each category is visited during the trip. MTNN is compared against RLORD, proposed in [11], which was designed to solve the spatially constrained OSR. Thus MTNN can be seen as an extended solution of OSR, which uses an R-Tree structure and exploits a page-level upper bound (PLUB) for efficient pruning at the Rtree node level. The approach is only superior to RLORD, when datasets are compact.

The partial sequenced route query with traveling rules in road networks [4] handled the multi-rule partial sequenced route (MRPSR) query, which enables the

user to efficiently plan a trip by defining a number of traveling rules. The MRPSR query provides a unified framework that subsumes the trip planning query (TPQ) and the optimal sequenced route (OSR) query. It proposes three algorithms (Nearest Neighbor-based Partial Sequence Route query (NNPSR) algorithm, NNPSR combined with the Light Optimal Route Discoverer (LORD) algorithm, Advanced A* Search-based Partial Sequence Route query (AASPSR(k))) for solving the problem in road networks with a sub-optimal solution. The paper *Optimal Route Queries with Arbitrary Order Constraints* [7], on the other hand, focuses on efficient, exact methods for the optimal TPQ route problem, while also giving the option for visiting order constraints. The proposed approach uses backward search and forward search and proposes four algorithms (Simple Backward Search (SBS), Batch Backward Search (BBS), Simple Forward Search (SFS), Batch Forward Search (BFS)), whereas the BFS algorithm that combines merits from both backward and forward search achieves the best performance. However, both approaches are based on the Euclidean distance between PoIs and therefore could be difficultly applied to metric spaces.

A linked area of research studies min-cost path queries. In *Monitoring Minimum Cost Paths on Road Networks* [15] the continuous min-cost path query is proposed and a system, PathMon, is presented to monitor min-cost routes in dynamic road networks, attempting to capture the characteristics of ever-changing road networks. Another paper [2] deals with query processing in spatial network databases and proposes an architecture that integrates network and Euclidean information, capturing pragmatic constraints.It designs algorithms based on the Euclidean restriction and a network expansion frameworks that take advantage of location and connectivity to efficiently prune the search space.

However, all of the aforementioned solutions have not yet attempted to design a uniform route query language for the road network's metric space with the ability to provide the user with options to apply different rules to the query and to retrieve an optimal result in the end. This will be the focus of this thesis.

Chapter 5

Operators

We have a starting point and a category sequence, which constitutes the SRQ. In addition to this two query parameters, the user can define other parameters, which are applied to the route query as query operators. Such possible query operators, presented in this thesis, are the equality operator, not-equality operator, or operator and order operator.

We define the *Complex Route Query* (CRQ):

Complex route query (CRQ): Given a starting point sp, a category sequence $M = (c_1, c_2, ..., c_n)$ and an operator $OPERATOR$, $Q(sp, M, OPERATOR)$ is a Complex Route (CR) Query, which searches for the optimal route $R = (r'_1, r'_2, ..., r'_l)$, defined as a sequence of PoIs.

In this chapter the proposed operators are covered in terms of their design, implementation and evaluation. It is divided into for subsections for each operator - EO, NEO, OR, ORDER. In the context of each operator, we first motivate the need for this operator and define it formally, then we go on to present the proposed approach and lastly we discuss the correctness of the solution and compare it with the baseline approach.

5.1 Equality operator

5.1.1 Motivation

The equality operator is based on the need of a user to express that some PoIs of the same category in the SRQ should be equal, as presented in the example in Section 2.1.

5.1.2 Problem definition

The equality operator is defined as follows:

Equality operator: Given a sequence of categories $M = (c_1, c_2, ..., c_l)$, 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 $R = (r_1, r_2, ..., r_l)$ 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 $r_i = r_j$.

5.1.3 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.

5.1.4 Proposed approach

The equality operator is designed using the PNE approach, proposed in [11] and briefly presented in algorithm `pne() [11]`. It uses the progressive neighbor explorator 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. The function `heuristic()` calculates the heuristic for a given route R.

Function heuristic(R)

1 `// Calculates the heuristic for the given route` $R = (r_1, r_2, ..., r_k)$
2 `// For every route, which already contains` r_i $R = (r_1, r_2, ..., r_i, ..., r_k)$ `the distance to` r_j `is calculated as 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 $max($`dist(`r_k, r_i`)`$)$;

Algorithm

The algorithm for the equality operator as shown in `equalityOperator()` is constructed using multiple procedures.

Algorithm 1: equalityOperator()

Input : $Q(sp, M = (c_1, c_2, ..., c_l)), EQUAL(i, j)$
Output: $R = (r_1, r_2, ..., r_l)$

```
1  initialize heap ;                          // Heap with PSR
2  initialize foundSR ;                       // 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();
```

$heap$, $found$ and UB are initialized (line 1 to 3). In $heap$ the PSR which are to be examined by the algorithm are stored. It is sorted by the *lower bound* of the PSR. $foundSR$ 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 procedure `trim()`.

First, an optimal sequenced route is found using the PNE algorithm (line 4). It is checked, if the two PoIs that the user has asked to be equal (line 5), are equal in the OSR. If so, the OSR is returned (line 7), else the equality operator continues with the creation of an artificial SR $dummySR$ and the modified PNE algorithm (line 9, 10).

Second, we artificially create a sequenced route from the optimal route, found by PNE, as seen in `dummySR()`. The optimal route is changed, so that r_j is made to be equal to r_i (line 2, 3) and the length of the artificially created PSR is the initial upper bound (line 4), by which later partial sequenced routes are either kept or discarded.

Then the algorithm `modifiedPNE()` begins iterating all r_1 from the category set U_{M_1}, builds PSRs with sp and each r_1 (line 1 to 7) and compares the *lower bound*, generated by them, to the global *upper bound* (line 4). The built PSRs are only considered in further steps of the algorithm, if their *lower bound* is smaller than the *upper bound*.

Next, the modified PNE performs as the original PNE algorithm by fetching partial sequenced routes from the $heap$ and generating new routes (line 8 to 24). This process is repeated until the $heap$ runs out of PSRs, which means that all possible

Procedure dummySR($optimalRoute$)

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

candidate routes have been expanded and taken into consideration.

At each subsequent iteration of the algorithm `modifiedPNE()` (line 9 to 26) there are four distinct cases depending on the length of the route. Case `caseBefore()` (line 11) and `caseAfter()` (line 20) follow the original PNE. Case `caseContaining()` (line 14) 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 immediately (lines 1 to 3). After that the PSR is modified accordingly and again a length check is performed before finally placing the route on the $heap$. The length check is performed to make sure that the PSR is not longer than the already found SR which has a length equal to the upper bound UB.

`caseBefore()` finds PSRs before r_j. Two modifications of the PSR are performed, which follow the PNE algorithm. In a) (line 4 to 9) 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 to 15) 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.

Algorithm 2: modifiedPNE()

```
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 = length(PSR) + heuristic(PSR);
4      if LB <= UB then
5          place the new PSR (r_1) on the heap;
6  while heap is not empty do
7      current ← fetch a PSR from the heap;
8      switch s = size(current) do
9          case s <= j − 1 do Finding PSRs before r_j
10             caseBefore();
11         case s = j do Finding PSR containing r_j
12             caseContaining();
13         case s = j + 1 do Finding PSR after/containing r_j
14             caseAfterOrContaining();
15         case s >= j + 2 do Finding PSRs after r_j
16             caseAfter();
17 return foundSR
```

Procedure caseBefore

```
1  LB = length(current) + heuristic(current);
2  // Heuristic check
3  if LB <= UB then
4      a)
5          nearestNeigbour(r_k, U_{M_{k+1}});
6          update PSR to contain r_{k+1};
7          // Length check
8          if length(PSR) <= UB then
9              place PSR on the heap;
10 b)
11     kNearestNeigbour(r_{k−1}, U_{M_k});
12     update PSR;
13     if length(PSR) <= UB then
14         place PSR on the heap;
```

In `caseContaining()` r_j is to be found. In a) (line 4 to 10) 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 calculated, because we want r_j to be equal to r_i in the route. In b) (line 12 to 16) 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 caseContaining

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

In `caseAfterOrContaining()` r_{j+1} is to be found. In a) (line 4 to 10) 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 usually found (according ot PNE), but in our case this is r_j and we have already calculated the travel distance between r_{j-1} and r_i in `caseContaining()`, so here nothing further needs to be done.

In `caseAfter()` we find PSR after r_j. The case is similar to procedure `caseBefore()`, except that in a) instead of directly putting the PSR on the $heap$, trimming is performed in a) (line 10) to check if the route is a full SR and if the candidate route $foundSR$ and the upper bound UB must be updated.

Procedure caseAfterOrContaining

```
1  LB = length(current) + heuristic(current);
2  // Heuristic check
3  if LB < UB then
4      a)
5          nearestNeigbour(r_j, U_{M_{j+1}});
6          update PSR to contain r_{j+1};
7          // Length check
8          if length(PSR) <= UB then
9              // Trimming part
10             trim(PSR);
11 b)
12     // Already found in caseContaning()
```

Procedure caseAfter

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

Algorithm 3: pne() [11]

```
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      case s ≠ l do
10         a)
11             nearestNeigbour(r_{|PSR|}, U_{M_{|PSR|+1}});
12             update PSR and perform trimming in case it is a candidate
                SR ;
13             put PSR back on the heap ;
14         b)
15             kNearestNeigbour(r_{|PSR|-1}, U_{M_{|PSR|}});
16             generate a new PSR and place it on the heap;
```

Procedure trim(PSR)

```
1  if size(PSR) = l then
2      if length(PSR) <= UB then
3          update UB;
4          update foundSR;
5  else
6      place PSR on the heap;
```

Running example

We describe the algorithm for the equality operator using the example in Section 2.1. The user in the example wanted to visit a restaurant, a bank, a movie theater and a restaurant again, but he wanted both restaurants to be equal ($M = (r, b, mt, r)$, $|M| = l = 4$, $EQUAL(0, 3)$). In Figure 5.1 the partial routes stored in the $heap$ in each step of the algorithm are displayed. First, the $optimalRoute$ is found with PNE: (r_1, b_1, mt_1, r_2). The algorithm checks if the PoIs at indices 0 and 3 are equal and since they are not, it continues with building the $dummySR$: (r_1, b_1, mt_1, r_1). The *upper bound* is initialized with the length of the $dummySR$, which is 12, and also $foundSR$ is initialized with $dummySR$ until a better route is possibly found. In step 2 of the $modifiedPNE()$ all neighbors to the starting point sp from type restaurant (r_1, r_2) are found and if their calculated $lowerbound$ is smaller than the $upperbound$, which is the case for both restaurant, they are put on the $heap$: Partial routes $R_1 = (r_1)$ with length 1 and heuristic 5 and $R_2 = (r_2)$ with length 5 and heuristic 9 are generated and placed on the $heap$. Since the $heap$ is ordered ascending by $lowerbound$, which is calculated as the sum of length and heuristic, R_1 is at the top of the $heap$ and fetched in step 3. Here we have $caseBefore()$, where we first check if the LB of the PSR satisfies the condition to be smaller than the UB, and step a) of the PNE algorithm is performed. The nearest neighbor to r_1 of type bank b_1 is found and the new generated PSR (r_1, b_1) with length 4 and heuristic 3 is calculated and placed on the $heap$. In step 4, (r_1, b_1) is fetched and we enter $caseBefore()$ again and since the condition for the *lower bound* to be smaller than the *upper bound* is fulfilled, steps a) and b) are performed; in a) the nearest neighbor to b_1 from type movie theater is found, which is mt_1, and the PSR is extended, then in b) the next nearest neighbor to r_1 from type bank b_2 is found and a new PSR (r_1, b_2) is generated. For both PSR their length is less the global *upper bound* and they are put on the $heap$. The process is repeated until a route with $size(PSR) = 3$ is reached. This is the case with step 8, where we have fetches PSR (r_1, b_1, mt_1). Here $caseContaining()$ is performed: the next PoI is forcefully set to be the PoI at index 0, which is r_1 and the travel distance from mt_1 to r_1 is calculated. $Trimming$ is performed on new SR (r_1, b_1, mt_1, r_1) with length 12: it is compared with the found route and $foundSR$, but it is the same and $foundSR$ and UB are not updated. In another scenario, if the newly found SR was shorter than $founSRd$, then it would be updated together with UB. In the next step 9, again we have $caseContaining()$ and a new full SR (r_2, b_1, mt_1, r_2) with length 15 is generated, but when compared with $foundSR$, it is longer, so it gets discarded. The same steps are executed in the next two step 10 and 11, until all routes have been developed and checked, whch is the case when $heap$ is empty. Then the optimal sequenced route with

equal PoIs at indices 0 and 3 in $foundSR$ is returned: (r_1, b_1, mt_1, r_1).

Step	PSR $R : length(R), heuristic(R)$
1	$(r_1 : 1, 5), (r_2 : 5, 4)$
2	$(r_1, b_1 : 4, 3), (r_2 : 5, 4)$
3	$(r_2 : 5, 4), (r_1, b_2 : 6, 5), (r_1, b_1, mt_1 : 7, 5)$
4	$(r_1, b_2 : 6, 5), (r_2, b_2 : 6, 5), (r_1, b_1, mt_1 : 7, 5)$
5	$(r_2, b_2 : 6, 5), (r_1, b_1, mt_1 : 7, 5), (r_1, b_2, mt_1 : 11, 5)$
6	$(r_2, b_1 : 8, 3), (r_1, b_1, mt_1 : 7, 5), (r_2, b_2, mt_1 : 11, 4), (r_1, b_2, mt_1 : 11, 5)$
7	$(r_1, b_1, mt_1 : 7, 5), (r_2, b_1, mt_1 : 11, 4), (r_2, b_2, mt_1 : 11, 4), (r_1, b_2, mt_1 : 11, 5)$
8	$(r_2, b_1, mt_1 : 11, 4), (r_2, b_2, mt_1 : 11, 4), (r_1, b_2, mt_1 : 11, 5)$
9	$(r_2, b_2, mt_1 : 11, 4), (r_1, b_2, mt_1 : 11, 5)$
10	$(r_1, b_2, mt_1 : 11, 5)$
11	$heap$ is empty

Table 5.1: Steps of the algorithm for EO using the road network from Figure 2.1

Correctness

Todo: Correctness

5.1.5 Baseline approach

The baseline approach to 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 are performed.

Algorithm

The algorithm `equalityOperator-baseline()` starts by finding an optimal sequenced route with PNE (line 2), as mentioned in the proposed approach 5.1.4 and checks if r_i and r_j are already equal (line 3). If this is the case, it returns the found optimal route (line 5), otherwise (line 7) it continues with to perform `modifiedPNE-baseline()`.

Algorithm 4: equalityOperator-baseline()

Input : $Q(sp, M = (c_1, c_2, ..., c_l)), EQUAL(i, j)$
Output: $R = (r_1, r_2, ..., r_l)$

```
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;
```

The algorithm `modifiedPNE-baseline()` proceeds with examining the routes on the $heap$, ordered by length, by size and modifying them according to PNE (line 4 to 38). When the current route on the $heap$ is a full SR, then the optimal route has been found (line 34). 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()

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

Correctness

Todo: Correctness of baseline

5.2 Not-Equality operator

5.2.1 Motivation

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. Following a similar example ot the one in Section 2.1 for the road network in Figure 2.1, 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 pharmacy and after that he wants to go to a different restaurant for coffee. In this specific scenario, the user wants to express his wish for the restaurants to be the different, but with the existing PNE [11] approach, which finds an OSR, the user would get the route (r_1, ph_1, r_1) with length 5, where both restaurants are the same. The proposed approach in this chapter strives to find the optimal route, where the two restaurants are different. It will not necessarily be the shortest possible sequenced route, but it will be the shortest route out of all possible routes, where the two PoIs are different. Such route in our example graph would be (r_1, ph_2, r_2) with length 7 (shown with dashed lines in the figure).

5.2.2 Problem definition

The not-equality operator is defined as follows:

Not-Equality operator: Given a sequence of categories $M = (c_1, c_2, ..., c_l)$, 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 $R = (r_1, r_2, ..., r_l)$ 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 $r_i \neq r_j$.

5.2.3 Proposed approach

The not-equality operator is designed using the PNE approach, proposed in [11]. It uses the progressive neighbour explorator 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 `notEqualityOperator()` 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 original PNE algorithm.

Algorithm 6: notEqualityOperator()

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

1 $initialize\ heap$;
2 $initialize\ candidate$;
3 $firstPSR$ =`nearestNeigbour(`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 **case** $s \neq l$ **do**
12 **a)**
13 `nearestNeigbour(`$r_{|PSR|}, U_{M_{|PSR|+1}}$`);`
14 update PSR;
15 `trim(`PSR`);`
16 **b)**
17 `kNearestNeigbour(`$r_{|PSR|-1}, U_{M_{|PSR|}}$`);`
18 generate a new PSR;
19 `trim(`PSR`);`

Each time a PSR is generated the procedure `trim()` 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 route is 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 to 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 a not possible SR. In case that 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] ≠ PSR[j] then
3          // Optimization:  length check
4          if length(PSR) <= length(candidate) then
5              update candidate;
6              place PSR on the heap;
7      else
8          // In case j is the last index in the route,
               we find the kth neighbor of the previous
               PoI to the last one
9          if size(PSR) = j + 1 then
10             kNearestNeigbour(r_{|PSR|-1}, U_{M_{|PSR|}});
11             generate a new PSR;
12             trim(PSR);
13 else
14     // Optimization:  length check
15     if length(PSR) <= length(candidate) then
16         place PSR on the heap;
```

Running example

We describe the algorithm for the not-equality operator using the example in Section 5.2.1. The user in the example wanted to visit a restaurant, a pharmacy and a restaurant again, but he wanted both restaurants to not be equal ($M = (r, ph, r)$,

$|M| = l = 3$, $NOTEQUAL(0, 2)$). In Figure 5.2 the partial routes stored in the $heap$ in each step of the algorithm are displayed. First, the nearest neighbor to the starting point sp, r_1 is found and a new $firstPSR$ is generated and put on the $heap$: (r_1) with length 1. In step 2 route (r_1) is fetched from the $heap$ and modified according to a) and b) as in the PNE algorithm. Similarly, this process is repeated until the route on top of the $heap$ follows the sequence $M = (r, ph, r)$. the difference here from the PNE algorithm comes from the trimming part. For example in step 2, route (r_1, ph_1) is fetched from the $heap$ and in a) the full SR (r_1, ph_1, r_1) is generated. In the trimming function, it is checked if the restaurants at positions 0 and 2 are equal. In case they are not the candidate route is initialized and placed on the $heap$, but in our case, they are equal and j is also the last index 2, which means that the next neighbor to ph_1 is found and the alternate route (r_1, ph_1, r_2) with length 10 is generated and placed on the $heap$. Then some more PSR are generated, until in step 4 (r_1, ph_2) again a full SR (r_1, ph_2, r_2) with length 7 is generated. The route is then trimmed and since the restaurants are not equal and also the route is shorter than the current candidate route, the candidate route is updated and the new SR is place on the $heap$. In step 5 the fetched route (r_1, ph_2, r_2) is a full SR, which satisfies the condition for the restaurant at indices 0 and 2 to not be equal, is returned as the optimal route.

Step	PSR $R : length(R)$
1	$(r_1 : 1)$
2	$(r_1, ph_1 : 3), (r_2 : 5)$
3	$(r_2 : 5), (r_1, ph_2 : 5), (r_1, ph_1, r_2 : 10)$
4	$(r_1, ph_2 : 5), (r_2, ph_2 : 7), (r_1, ph_1, r_2 : 10)$
5	$(r_1, ph_2, r_2 : 7), (r_2, ph_2 : 7), (r_1, ph_1, r_2 : 10)$

Table 5.2: Steps of the algorithm for NEO using the road network from Figure 2.1

Correctness

Todo: Correctness

5.3 Or operator

5.3.1 Motivation

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.

Suppose that a user poses the following query: first he has time to go to either a bank or a pharmacy, then he has a date in a movie theater and after that he plans to go to the restaurant. The proposed approach uses a modified PNE to generate an optimal result for this type of query.

5.3.2 Problem definition

In order to explain the or query, first we need to define what an OR sequence is:
OR sequence: An OR sequence $OR = (M_1, M_2, ..., M_m)$ represents the disjunction of category sequences, such as $M_1 = (c_1, c_2, ..., c_l)$. 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 $S = (OR_1, OR_2, ..., OR_n)$ 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 $P = (M_a, M_b, ..., M_z)$ is a permutation of S, where $M_a = (c_{a1}, c_{a2}, ..., c_{al})$, $M_b = (c_{b1}, c_{b2}, ..., c_{bl})$ and $M_z = (c_{z1}, c_{z2}, ..., c_{zl})$ build a category sequence
$M = (c_{a1}, c_{a2}, ..., c_{al}, c_{b1}, c_{b2}, ..., c_{bl}, c_{z1}, c_{z2}, ..., c_{zl})$.

5.3.3 Proposed approach

The or operator is designed using the PNE approach, proposed in [11]. It progressively inspects each option M_i from the or sequences OR_i in
$S = (OR_1, OR_2, ..., OR_n)$, compares them and continues with the best one, based on length, until it reaches a full sequenced route.

Algorithm

The algorithm `orOperator()` 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 $S = (OR_1, OR_2, ..., OR_n)$. 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 $S = (OR_1, OR_2, ..., OR_n)$ for the fetched PSR. When the current route on the $heap$ is a full SR, where $s = n$, indicating 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 (line 25 to 30).

Algorithm 7: orOperator()

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

```
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          nearestNeigbour(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 categries
               from M_i are put into categories for the
               specific PSR
12         nearestNeigbour(sp, U_{M_i[1]});
13         PSR.categories ← M_i[2..l];
14         update shortestPSR;
15 firstPSR ← shortestPSR;
16 place firstPSR on heap;
17 // At each iteration of PNE a PSR (partial
       sequenced route) is fetched and examined based
       on its length and it is checked
18 fetch a PSR from the heap;
19 // index(PSR) indicates the index of the last
       found OR sequence OR_s in S = (OR_1, OR_2, ..., OR_n)
       for the fetched PSR. It is a full SR, when
       s = n, which indicates the last OR sequence
       OR_n.
20 switch s = index(PSR) do
21     case s == n do
22         PSR is the optimal route;
23         return PSR;
24     case s ≠ n do
25         // a)
26         modifyRouteA(PSR);
27         // b)
28         modifyRouteB(PSR);
```

In `modifyRouteA()` it is differentiated between a complete PSR and an 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 $S = (OR_1, OR_2, ..., OR_n)$. 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()` the k-th neighbor to the second to last PoI should be found. Trimming is also performed (see `trim()`).

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 `nearestNeigbour(`$r_{|PSR|}, U_{M_i}$`);`
9 *update* $shortestPSR$;
10 **else**
11 `//` M_i `is a sequence of categories`
12 `// The neighbor from the first category` $M_i[1]$ `is found and the rest of the categries from` M_i `are put into` *categories* `for the specific PSR`
13 `nearestNeigbour(`$r_{|PSR|}, u_{M_i[1]}$`);`
14 $PSR.categories \leftarrow M_i[2..l]$;
15 *update* $shortestPSR$;
16 $newPSR \leftarrow shortestPSR$;
17 **else**
18 `// We continue to find the PoI of the next category in the` $PSR.categories$
19 `nearestNeigbour(`$r_{|PSR|}, U_{PSR.categories[1]}$`);`
20 *remove* $M_i[1]$ from $PSR.categories$;
21 `//` $PSR.prevPosition$`, when set to` *true* `indicates that in modifyRouteB(`PSR`) all the sequences in the OR sequence of the previous position` OR_s `should be traversed and checked again, otherwise the k-th neigbor to the second to last PoI is found`
22 $PSR.prevPosition \leftarrow false$;
23 *update* $newPSR$;
24 `// Trimming is also done (see trim)`
25 place $newPSR$ on *heap*;

In `modifyRouteB()` it is checked if the indicator $PSR.prevPosition$ for the PSR is set ot $true$ (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 $S = (OR_1, OR_2, ..., OR_n)$. 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 k-th 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 PSRs the shortest one is chosen and put on the $heap$. If the indicator $PSR.prevPosition$ for the PSR is set ot $false$ it means that the k-th nearest neighbor to $r_{|PSR|}$ in $U_{PSR.categories[1]}$ is found (line 20 to 25).

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              kNearestNeigbour(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
                  categries from M_i are put into categories
                  for the specific PSR
13             kNearestNeigbour(r_{|PSR|-1}, U_{M_i[1]});
14             PSR.categories ← M_i[2..l];
15             update shortestPSR;
16     newPSR ← shortestPSR;
17 else
18     // We find the k-th neighbor to the second to
          last PoI
19     kNearestNeigbour(r_{|PSR|-1}, U_{M_{|PSR|}});
20     update newPSR;
21 place newPSR on heap;
```

Running example

We describe the algorithm for the or operator using the example in Section 5.3.1, using the road network in Figure 2.1. The user wanted to visit either a bank or a pharmacy, then go to the movies and then to a restaurant (i.e., $S = (OR_1, OR_2, OR_3)$, $|S| = n = 3$, $OR_1 = ((b), (ph))$, $OR_2 = ((mt))$, $OR_2 = ((r))$). In Figure 5.3 the partial routes stored in the $heap$ in each step of the algorithm are displayed. First, all routes from the starting point sp to all possible category sequences in OR sequence OR_1 (b) and (ph) are generated. Since there is only one category the simple case is executed, where the nearest neighbor is found. Routes (b_1) with length 2 and (ph_1) with length 3 are generated and compared and the second one is chosen, based on its length, and placed on the $heap$. In step 1 route (b_1) fetched and the nearest neighbor from the next OR sequence OR_2 is found. OR_2 only contains one category sequence with one category, therefore this is a simple case $modifyRouteA()$, where the nearest neighbor is found. In $modifyRouteB()$ the next nearest neighbor to the starting point from the OR sequence OR_1 is found, which is ph_1 with length 3, because b_2 with length 4 is further apart. In step 5 a full route (b_1, mt_1, r_2) with length 9 is generated and placed on the $heap$ as the candidate route. Then in step 6 another SR (ph_1, mt_1, r_2) with length 10 is generated, but it is longer than the candidate route, previously generated, so it gets discarded. Finally, the optimal route (b_1, mt_1, r_2) with length 9 is returned.

Step	PSR $R : length(r), index(R)$
1	$(b_1 : 2, 1)$
2	$(ph_1 : 3, 1), (b_1, mt_1 : 5, 2)$
3	$(ph_2 : 3, 1), (ph_1, mt_1 : 6, 2), (b_1, mt_1 : 5, 2)$
4	$(b_2 : 4, 1), (b_1, mt_1 : 5, 2), (ph_1, mt_1 : 6, 2), (ph_2, mt_1 : 7, 2)$
5	$(b_1, mt_1 : 5, 2), (ph_1, mt_1 : 6, 2), (ph_2, mt_1 : 7, 2), (b_2, mt_1 : 9, 2)$
6	$(ph_1, mt_1 : 6, 2), (ph_2, mt_1 : 7, 2), (b_1, mt_1, r_2 : 9, 3), (b_2, mt_1 : 9, 2)$
7	$(ph_2, mt_1 : 7, 2), (b_1, mt_1, r_2 : 9, 3), (b_2, mt_1 : 9, 2), \sout{(ph_1, mt_1, r_2 : 10, 3)}$
8	$(b_1, mt_1, r_2 : 9, 3), (b_2, mt_1 : 9, 2), \sout{(ph_2, mt_1, r_2 : 11, 3)}$

Table 5.3: Steps of the algorithm for OR using the road network from Figure 2.1

5.3.4 Baseline approach

A naive solution to find an optimal route with the or operator would be to run the PNE algorithm on all possible permutations of the query and to find the shortest route out of all. This is known as the baseline approach.

Correctness

Todo: Correctness

5.4 Order operator

5.4.1 Motivation

The order operator gives the user the opportunity to build a single query with applied to the category sequence order rules that satisfy his needs. 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 specific fixed positions in the category sequence. Then, only these categories are found in the form of a sequenced route query (SRQ) while the others are inspected and prioritized algorithmically. The approach is responsible for finding the best route, based on length, out of all possible total-order optimal route queries.

Suppose that a user poses the following query: he wants to visit a restaurant, before and after that he wishes to visit a bank and a pharmacy, but it is not important for him to visit one before the other. So the query only knows that restaurant must be visited second. The proposed approach in this chapter uses a modified PNE algorithm to generate an optimal result for this type of query.

5.4.2 Problem definition

First we need to define what an ORDER sequence is:
ORDER sequence: An order sequence $ORDER = (i_1, i_2, ..., i_k)$, with $k \leq l$ and $1 \leq i_i \leq l$ is a sequence of indices in a category sequence $M_1 = (c_1, c_2, ..., c_l)$. 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 $M = (c_1, c_2, ..., c_l)$, a starting point sp in $\mathbb{R}^2$ and an ORDER sequence $ORDER = (i_1, i_2, ..., i_k)$, $Q(sp, M, ORDER)$ 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.

5.4.3 Proposed approach

The order operator is designed using the PNE approach, proposed in [11]. 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 `orderOperator()` starts by initializing the $heap$, ordered by the length of the routes, and the NOTORDERED sequence (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 PSRs 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, 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 (line 22 to 27).

Algorithm 8: orderOperator()

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

```
1  initialize heap;
2  initialize NOTORDERED = ORDER;
3  if ORDER contains 1 then
4      firstPSR =nearestNeigbour(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          nearestNeigbour(sp, U_{M_i});
10         update shortestPSR;
11     firstPSR ← shortestPSR;
12 place firstPSR on heap;
13 // At each iteration of PNE a PSR (partial
       sequenced route) is fetched and examined based
       on its length and it is checked
14 fetch a PSR from the heap;
15 switch s = size(current) do
16     case s == l do
17         PSR is the optimal route;
18         return PSR;
19     case s ≠ l do
20         // a)
21         modifyRouteA(PSR);
22         // b)
23         modifyRouteB(PSR);
```

In `modifyRouteA()` 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 `trim()`).

Procedure modifyRouteA(PSR)

```
1  if ORDER contains s then
2      newPSR = nearestNeigbour(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          nearestNeigbour(r_{s-1}, U_{M_i});
8          update shortestPSR;
9      newPSR ← shortestPSR;
10     update PSR.NOTORDERED;
11 // Trimming is also done (see trim)
12 place newPSR on heap;
```

In `modifyRouteB()` 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 =kNearestNeigbour(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          kNearestNeigbour(r_{s−2}, U_{M_{i−1}});
8          update shortestPSR;
9      newPSR ← shortestPSR;
10     update PSR.NOTORDERED;
11 place newPSR on heap;
```

Running example

We describe the algorithm for the order operator using the example in Section 5.4.1, using the road network in Figure 2.1. The user wanted to visit the restaurant at seond place in his route and also visit a bank and a pharmacy (i.e., $M = (b, r, ph)$, $|M| = l = 3$, $ORDER = (1)$). In Figure 5.4 the partial routes stored in the $heap$ in each step of the algorithm are displayed. First, the $notordered$ list is initiated: $[b, ph]$. Since the first position of the category sequence in the query is not fixed, routes from the starting point sp with both categories in the $notordered$ list are generated: b_1 with length 2 and ph_1 with length 3. Route b_1 is shorter and is therefore placed on the $heap$. In step one, we have the case, where the searched PoI is at a fixed position 1. In this case in a) the nearest neighbor to b_1 from type restaurant r_2 is found, while in b) the next nearest neighbor to the starting point types bank and pharmacy are found and compared again. In step 5 the first SR (ph_1, r_1, b_1) with length 8 is generated and placed as a candidate route on the $heap$. This process continues until in step 11 the optimal route (ph_2, r_2, b_2) with length 6 is returned.

5.4.4 Baseline approach

A naive solution, similar to the one with the or operator in Chapter 5.3.4, for finding an optimal route with the order operator would be to run the PNE algorithm

Step	PSR $R : length(r), r.notordered$
1	$(b_1 : 2, [ph])$
2	$(ph_1 : 3, [b]), (b_1, r_2 : 5, [ph])$
3	$(ph_2 : 3, [b]), (ph_1, r_1 : 5, [b]), (b_1, r_2 : 5, [ph])$
4	$(b_2 : 4, [ph]), (ph_2, r_2 : 5, [b]), (ph_1, r_1 : 5, [b]), (b_1, r_2 : 5, [ph])$
5	$(ph_1, r_1 : 5, [b]), (b_2, r_2 : 5, [ph]), (ph_2, r_2 : 5, [b]), (b_1, r_2 : 5, [ph])$
6	$(b_2, r_2 : 5, [ph]), (ph_2, r_2 : 5, [b]), (b_1, r_2 : 5, [ph]), (ph_1, r_1, b_1 : 8, [])$
7	$(b_1, r_2 : 5, [ph]), (b_2, r_2, ph : 5, [ph]), (ph_2, r_2 : 5, [b]), (b_2, r_2, ph_2 : 7, []),$ ~~$(ph_1, r_1, b_1 : 8, [])$~~ , $(b_2, r_1 : 9, [ph]), (ph_1, r_2 : 10, [b])$
8	$(b_2, r_2 : 5, [ph]), (ph_2, r_2 : 5, [b]), (b_1, r_1 : 5, [ph]), (b_2, r_2, ph_2 : 7, []),$ ~~$(b_1, r_2, ph_2 : 7, [])$~~ , $(b_2, r_1 : 9, [ph]), (ph_1, r_2 : 10, [b])$
9	$(b_1, r_1 : 5, [ph]), (b_2, r_2 : 5, [ph]), (ph_2, r_2, b_2 : 6, [])$, ~~$(b_2, r_2, ph_2 : 7, [])$~~ , $(ph_2, r_1 : 7, [b]), (b_2, r_1 : 9, [ph]), (ph_1, r_2 : 10, [b])$
10	$(b_2, r_2 : 5, [ph]), (ph_2, r_2, b_2 : 6, [])$, ~~$(b_1, r_1, ph_1 : 7, [])$~~ , $(ph_2, r_1 : 7, [b]), (b_2, r_1 : 9, [ph]), (ph_1, r_2 : 10, [b])$
11	$(ph_2, r_2, b_2 : 6, []), (ph_2, r_1 : 7, [b]), (b_2, r_1 : 9, [ph]), (ph_1, r_2 : 10, [b])$

Table 5.4: Steps of the algorithm for ORDER using the road network from Figure 2.1

on all possible permutations of the query and to find the shortest route out of all. This is known as the baseline approach.

Correctness

Todo: Correctness

5.5 Summary

Todo: Summary of the operators

Chapter 6

Experimental studies

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.

Dataset: The graph used to conduct the experiments on 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 the experiments is the road network of Berlin, with 428769 crossroad nodes, 504229 road edges, 5548 PoIs and 7 category types: restaurant, coffee shop, atms/bank, movie theater, pharmaciy, pubs/bar, gas station (see 6.1).

Points	*Size*	*Frequency*
Restaurants	2081	
Coffee shops	1002	High
Pubs and bars	958	
Atms/Banks	597	
Pharmacies	589	Middle
Gas stations	180	
Movie theaters	141	Low

Table 6.1: PoIs in Berlin's dataset

Technical details: The experiments were performed on 2 Linux machines with AMD Opteron Processor 6212 with 2,60 GHz and Intel Xeon E5-2630 processor with 2.40GHz and respectively 16 CPUs and 128 GB RAM. The experiments for each parameter type were executed on 1000 queries with randomly selected starting points and the average of the results is reported.

Several experiments were conducted to evaluate the performance of the proposed algorithms. The evaluation criteria. which relate to all the operators, presented in the thesis are the following: (1) processing time (in milliseconds), (2) total number of heap fetches and (3) maximum heap size, representing the work space (WS) of the method.

6.1 Equality operator

The equality operator was evaluated with respect to the effect of following 3 parameters: (1) query length (cardinality of the category sequence $|M|$), (2) frequency of the categories in place of the equal indices i, j and (3) distance between the equal indices i, j in the category sequence M.

In the first set of experiments, shown in Figure 6.1 a), b), c), the equality operator was evaluated in terms of query length, which varies from 3 to 7. Queries with length less than 3 would be immediately solved with PNE, which is why we do not consider them in these experiments. As we can see in Figure 6.1 a), the query processing time increases proportionately to the query length. Figure 6.1 a) also shows what portion of the total processing time of the proposed approach belongs to executing PNE in the first step of the algorithm. And as expected, both PNE's time and the proposed approach's time is increasing with the query length. Figure 6.1 b), c) follow the same trend of a). As query length increases, number of heap fetches and maximum heap size also increase.

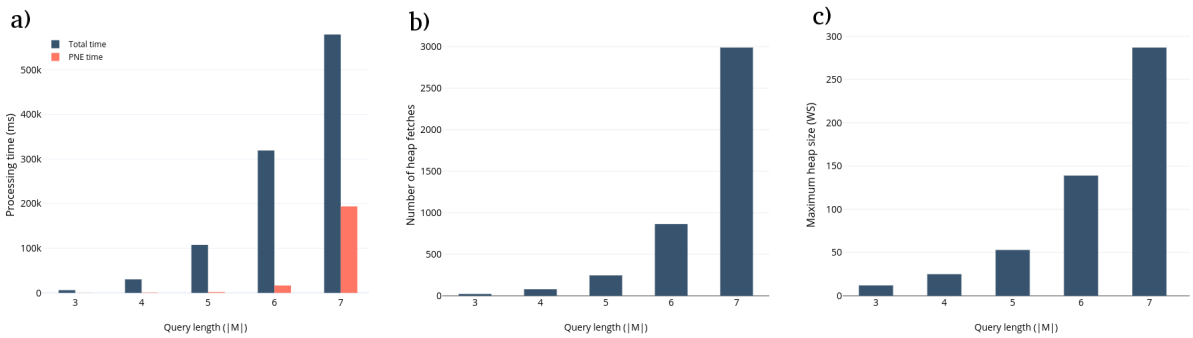

Figure 6.1: Equality operator - query length experiments

In the next set of experiments, shown in Figure 6.2 a), b), c), the equality operator was evaluated in terms of the frequency of the categories in place of the equal indices i, j. Frequency relates to the size of each PoI dataset (see Table 6.1) and is categorized into low, middle and high, for a query length of 5. Figure 6.2 b), c) follow the same trend as the experiments for query length. As frequency of the categories, which are selected to be equal, increases, the processing time, number of heap fetches and heap maximum size increase proportionately. This can be explained with the fact that having more points in the PoI dataset of the equal categories increases the search space of the algorithm and respectively more partial routes are being generated, which increases the heap size, number of heap fetches and logically in turn the processing time.

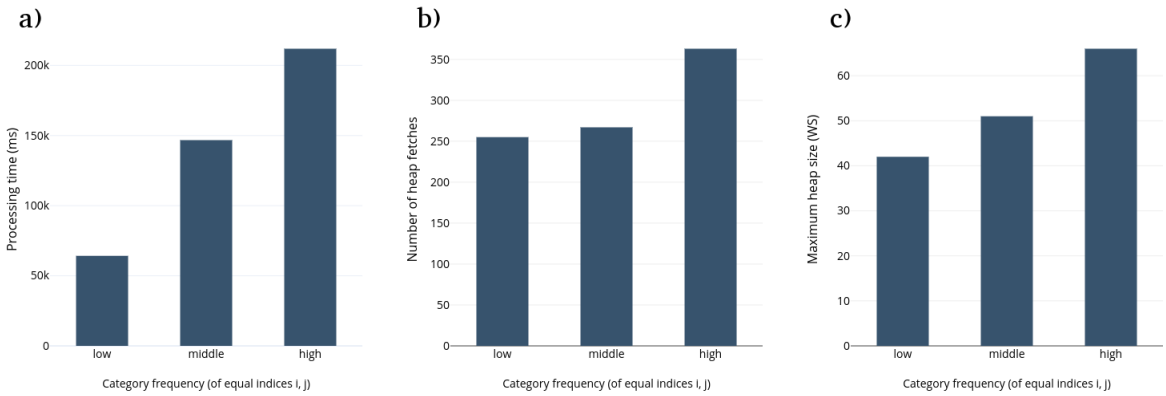

Figure 6.2: Equality operator - category frequency experiments

In the third set of experiments, shown in Figure 6.3 a), b), c), the equality operator

was evaluated in terms of the distance between the equal indices i, j in the category sequence M, which varies from 1 to 3, for a query length of 5. When the distance between the equal indices is 0, the result is always found with PNE, therefore we do not consider distance 1 in the experiments. As distance of the categories, which are selected to be equal, increases, the processing time, number of heap fetches and heap maximum size increase proportionately. This stems from the fact that by increasing the distance, the probability that the PoIs at indices i and j would be equal decreases, therefore less of the routes are found in the first step of the algorithm with PNE. This in turn causes the algorithm to continue with the heuristic approach in order to find an optimal route, which increases the processing time, number of heap fetches and also the work space (maximum heap size).

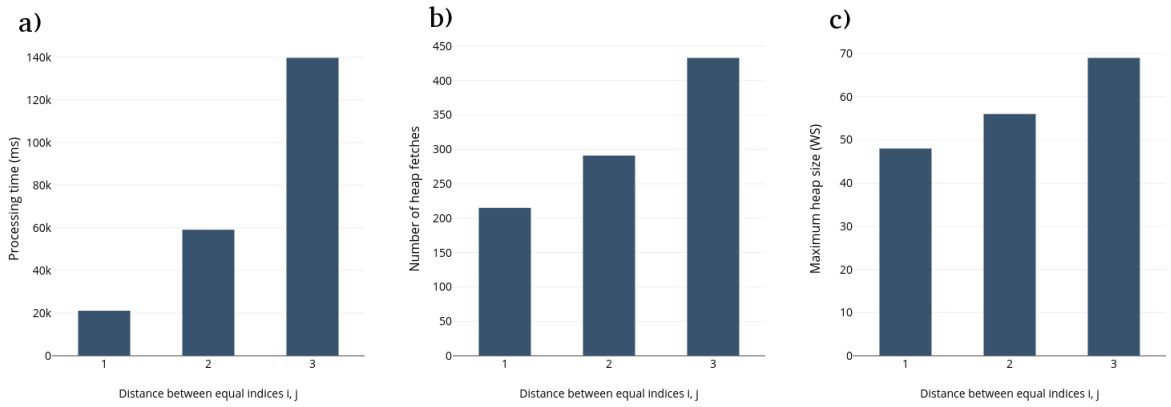

Figure 6.3: Equality operator - distance between equal indices experiments

Finally, the last set of experiments, shown in Figure 6.3 a), b), c), compare the baseline approach with the proposed approach in terms of query length. It can be seen that the proposed approach outperforms the baseline approach for all values of the query length. Also with increase in query length, the processing time , number of heap fetches and maximum heap size of the baseline approach increase with a rate that is more of that of the proposed approach, in 6.3 a) for query length 7 the processing time of the baseline approach is so long that it exceeds the graphic's range and is not fully depicted.

In conclusion, the proposed approach outperforms the baseline approach by a significant amount and it performs as expected in all parameters.

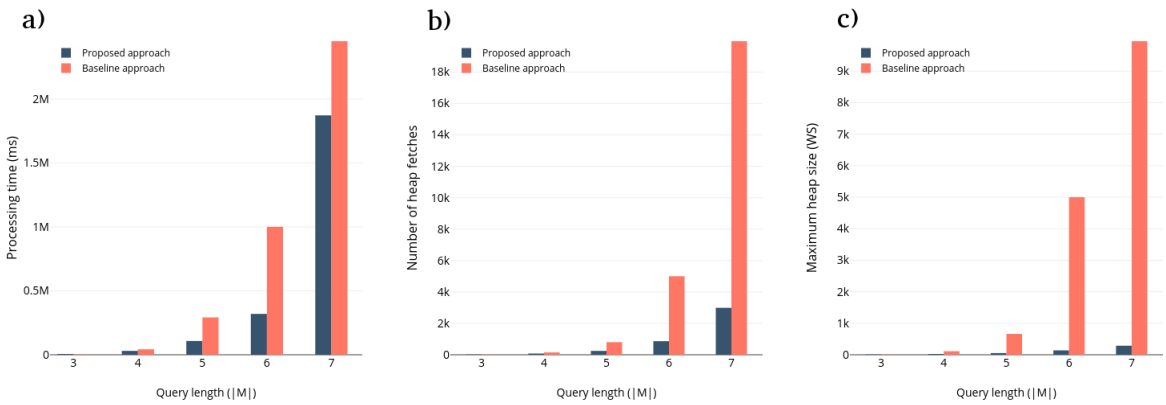

Figure 6.4: Equality operator - comparison between the proposed approach and the baseline approach

6.2 Not-Equality operator

The not-equality operator was also evaluated with respect to the effect of following 3 parameters: (1) query length (cardinality of the category sequence $|M|$), (2) frequency of the categories in place of the equal indices i, j and (3) distance between the equal indices i, j in the category sequence M.

In the first set of experiments, shown in Figure 6.5 a), b), c), the not-equality operator was evaluated in terms of query length, which varies from 3 to 7. As we can see in Figure 6.5 a), the query processing time increases proportionately to the query length. Figure 6.5 b), c) follow the same trend of a). As query length increases, number of heap fetches and maximum heap size also increase.

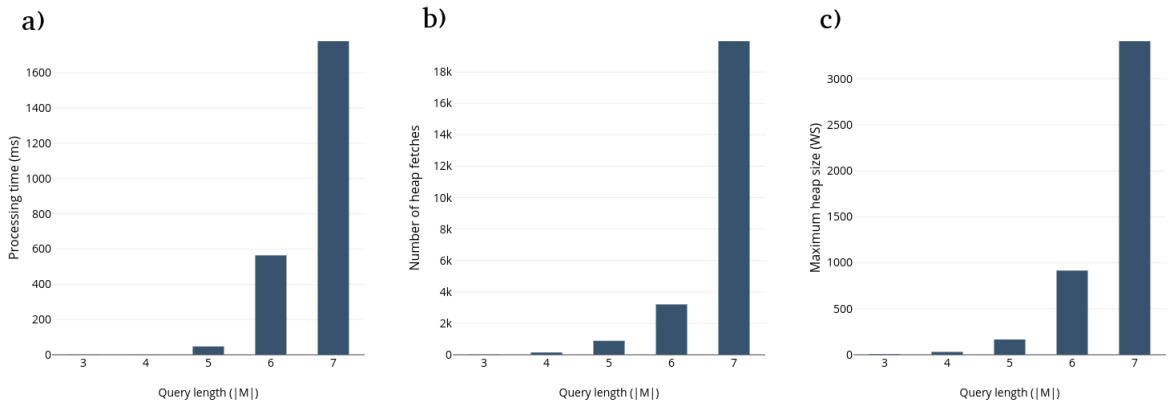

Figure 6.5: Not-equality operator - query length experiments

In the second set of experiments, shown in Figure 6.6 a), b), c), the equality operator was evaluated in terms of the frequency of the categories in place of the equal indices i, j, which can be low, middle and high, for a query length of 5. Here we can see more interesting results than with the equality operator. The low frequency has higher results on all parameters than the middle frequency. This can be explained with the fact that when the categories which have to not be equal are less frequent, then the possibility of PNE reaching a route with different PoIs at places of the not equal categories is lower, therefore the not-equality algorithm searches for longer for not equal PoIs, compared to when the category has a middle frequency. And when the frequency is high, if PNE doesn't directly find a route with equal PoIs then the not-equality algorithm must inspect more points, therefore generates more routes and the time increases significantly. Figure 6.6 b), c) follow the same trend as processing time.

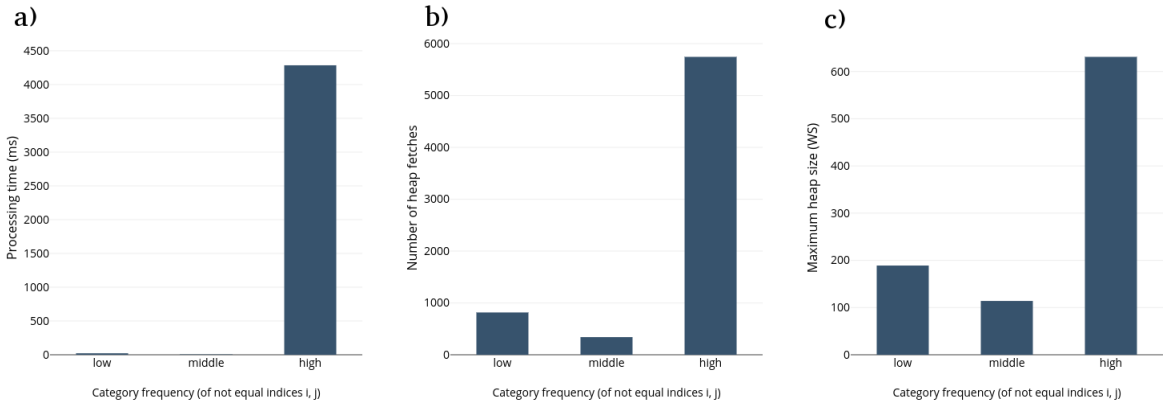

Figure 6.6: Not-equality operator - category frequency experiments

In the third set of experiments, shown in Figure 6.7 a), b), c), the equality operator was evaluated in terms of the distance between the equal indices i, j in the category sequence M, which varies from 0 to 3, , for a query length of 5. Here we can also see more interesting results than with the equality operator. When the distance between the equal indices is 0, the result, usually found with PNE, always delivers equal points at indices i and j. Therefore the algorithm for the not-equality operator has to inspect more points in order to find the optimal route, where the PoIs at indices i and j are not equal to each other. This increases the search space and in turn the number of heap fetches, maximum heap size and the processing time increase as well. For distances 1, 2, 3 no obvious argumentation can be applied, because here we can not judge the possibility for equal PoIs found with PNE objectively. Nevertheless, all three performance parameters - processing time, number of heap fetches and maximum heap size, follow the same trend and increase proportionately to each other.

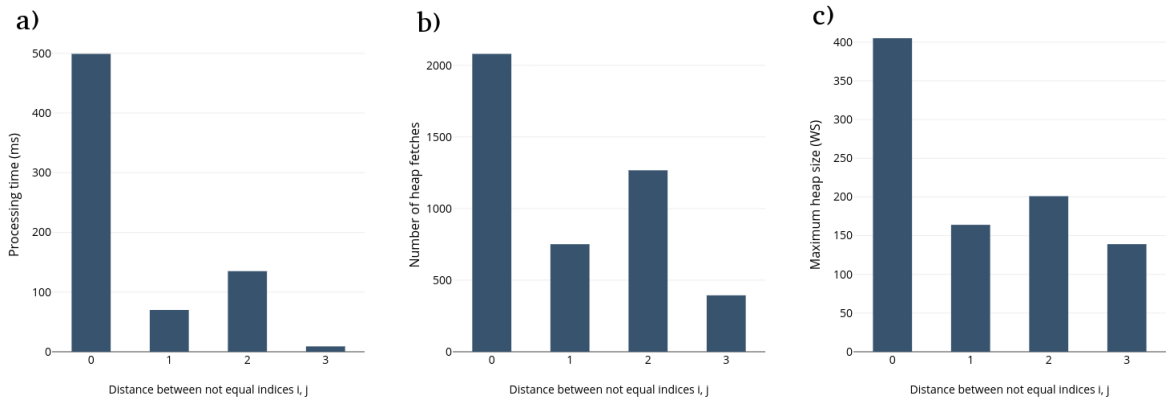

Figure 6.7: Not-equality operator - distance between not equal indices experiments

6.3 Or operator

The or operator was evaluated with respect to the type of number of operands, for a default query length of 5. Three different types of queries were issued, where we changed the type and number of or operands in the first OR sequence OR_1 of the query, while the other four OR sequences only contained one category sequence with a single category. For 2 simple operands the first OR sequence contained two category sequenced with one single category each, for 3 simple operands the first OR sequence contained three category sequenced with one single category each and for 2 complex operands the first OR sequence contained two complex category sequences with 2 categories in each. In this way the complexity of the problem was gradually increased to see how the baseline approach compares to proposed approach of the algorithm. Figure 6.8 a), b), c) shows that the processing time, number of heap fetches and maximum heap size increase proportionately with the complexity of the query. As seen from the experiments the proposed approach is also by magnitudes faster and also more efficient in terms of heap size, as the complexity of the query increases.

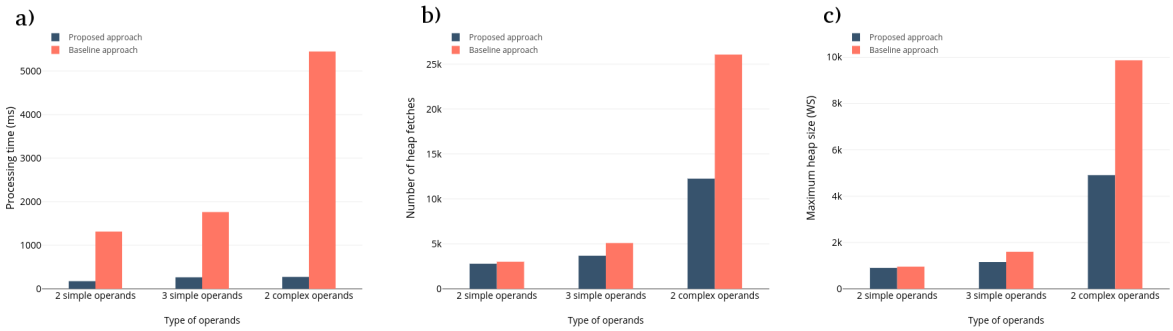

Figure 6.8: Or operator - type and number of operands experiments

6.4 Order operator

The order operator was evaluated with respect to the number of fixed positions in a default query length of 5. The number ranges between 0 and 3. When the number of fixed positions is 4 or 5, the problem can be solved with simple PNE, therefore we do not consider these numbers in the experiments. The complexity of the problem is gradually decreased to see how the baseline approach compares to proposed approach of the algorithm. Figure 6.9 a), b), c) shows that the processing time, number of heap fetches and maximum heap size decrease proportionately with the complexity of the query. As seen from the experiments the proposed approach is also by magnitudes faster and also more efficient in terms of heap size.

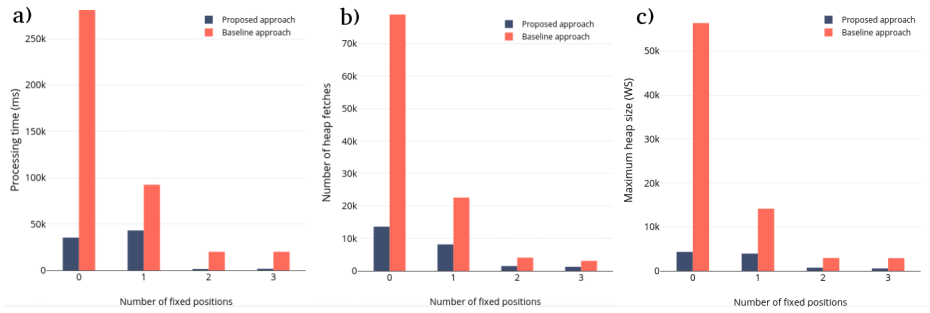

Figure 6.9: Order operator - number of fixed positions experiments

6.5 Evaluation

Todo: Evaluation

Chapter 7

Conclusion and Future work

I studied the different types of route queries, that are currently present in the research, and I identified the lack of a query language designed for the user's specific needs for flexibility when it comes to route queries. In order to attempt to fill this gap, I proposed four operators, which are specifically designed for route queries in metric spaces and the prime factor upon which routes are quantified is travel distance. We first presented the equality operator, which was designed using the PNE approach [11] and a heuristic to reduce the search space. We showed through evaluation experiments that the proposed approach outperforms the baseline approach in all of the experiment parameters. Next, we introduced the not-equality operator as extension to the equality operator. For this operator we modified the PNE algorithm in order to get the desired result. Furthermore, we presented the or operator, which strives to give the user maximum flexibility when it comes to building a multiple-option route query. For this operator we extended the PNE approach to work with multiple query options simultaneously and when we compared with the baseline approach, it proved to perform significantly better, especially when many or operands are presents. Last but not least, we designed the order operator, which gives the user the ability to construct partially or even fully not sequenced route query. This approach was also designed similarly to the or operator by extending the PNE approach to handle multiple query options at once. When comparing with the baseline approach, we concluded that the proposed approach performs exponentially better than the trivial solution whenever the number of ordered elements decreases.

With these finding in mind, I consider my work to has successfully delivered four query language operators that could be implemented and used in location based services. In comparison with other attempts at solving the multi-rule route queries [4], our proposed approaches deliver an optimal result. It is important to

note, however, that the mentioned operators are definitely not a complete set of query language operators, but they represent the four ones, that have proved to be the most desired in query language overall and therefore fulfill the user requirements the most. Nevertheless, while researching the topic of route queries, we came up with multiple ideas for operators that could be further researched and implemented. Such operator is for example the hops operator, with which the user can explicitly define how many location visits he wants to make between two categories in the category sequence. This operator is very similar to the order operator and could be designed similarly. Furthermore, an operator could be developed, with the help of which the user can specify necessary categories, which must be present in the final result. Considering this, the query route would only contain the PoIs of unnecessary categories, if these are situated on the way between the necessary points and the route's length does not increase with by adding them. Although still useful as a separate operator, the user could also achieve the same with our already developed or operator.
Another ideas for operators, which are targeted more towards the semantic hierarchy of the categories in the user-specified category sequence, could be the and and not operators. These two operators could be applied to specific categories in the user query's category sequence and could be useful in the case that a single PoI has many category types. For example of the and operator, a user may want to go to a cafe, that is also a restaurant, or in the case of the not operator, he may want to visit a restaurant, which is also specifically not categorized as a coffee shop.
All of these mentioned operators could be also based on the PNE approach that we used for the four operator in this thesis. But another solution could be to expand on the *Skyline Sequenced Route Query* (SkySR) [5]. This method searches for routes based on the Skyline concept, which entails finding routes that are not worse than any other routes in terms of theirs scores - route length and semantic similarity. This could be another alternative to give more flexibility to the user by making use of the semantic similarity of PoIs' categories. An operator that could be useful in this case is the perfection operator, applied to categories in the category sequence. With this operator, a user can specifically force the Skyline algorithm to match the category to which the operator has been applied perfectly in terms of semantic similarity. Finally, there is probably many other operators that would come up with further scientific work and research on the topic of route queries and the ones we summarize here are only our suggestions to the topic. For future work, it would be also interesting to extend the proposed algorithms to also support dynamic road networks, in which traffic information is provided (e.g., travel time, traffic congestion, etc.).

In conclusion, I think that with the development of four operators for the route query language we have successfully contributed to this field of research. We

have also managed to identify linked topics for further study and to make recommendations on how to expand the query language.

List of Figures

List of Tables

List of Algorithms

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] Nikos Mamoulis Yufei Tao Dimitris Papadias, Jun Zhang. Query processing in spatial network databases. Department of Computer Science, Hong Kong University of Science and Technology.

[3] Marios Hadjieleftheriou George Kollios Feifei Li, Dihan Cheng and Shang-Hua Teng. On trip planning queries in spatial databases. Computer Science Department Boston University.

[4] Min-Te Sun Roger Zimmermann Haiquan Chen, Wei-Shinn Ku. The partial sequenced route query with traveling rules in road networks. Geoinformatica (2011) 15: 541.

[5] Xuegang Huang and Christian S. Jensen. In-route skyline querying for location-based services. Department of Computer Science, Aalborg University, Denmark.

[6] 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.

[7] Nikos Mamoulis Jing Li, Yin Yang. Optimal route queries with arbitrary order constraints. IEEE Transactions on Knowledge and Data Engineering (Volume: 25, Issue: 5, May 2013).

[8] Stefan Funke Jochen Eisner. Sequenced route queries: Getting things done on the way back home. Universitat Stuttgart, Institut fur Formale Methoden der Informatik.

[9] Cyrus Shahabi Mehdi Sharifzadeh. Processing optimal sequenced route queries using voronoi diagrams. C. Geoinformatica (2008) 12: 411.

[10] Arnab Bhattacharya Nirmesh Malviya, Samuel Madden. A continuous query system for dynamic route planning. MIT CSAIL, CSE - IIT Kanpur.

[11] Mehdi Sharifzadeh and Cyrus Shahabi Mohammad Kolahdouzan. The optimal sequenced route query. Computer Science Department, University of Southern California.

[12] Konrad Stocker1 Stephan Börzsönyi, Donald Kossmann. The skyline operator. Universität Passau, Technische Universität München.

[13] Hui Xiong Xiaobin Ma, Shashi Shekhar and Pusheng Zhang. Exploiting a page-level upper bound for multi-type nearest neighbor queries. University of Minnesota, Rutgers University, Microsoft Corporation.

[14] Eliyahu Safra-Yehoshua Sagiv Yaron Kanza, Roy Levin. An interactive approach to route search. Technion, Hebrew University.

[15] Wang-Chien Lee Yuan Tian, Ken C. K. Lee. Monitoring minimum cost paths on road networks. Department of Computer Science Engineering The Pennsylvania State University.

[16] Noboru Sonehara Yutaka Ohsawa, Htoo Htoo and Masao Sakauchi. Sequenced route query in road network distance based on incremental euclidean restriction. Graduate School of Science and Engineering, Saitama University, National Institute of Informatics.

[17] 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.