Query Algorithms for Location Based Services in Road Network Distance

Htoo Htoo

A Thesis Submitted to the Graduate School of Science and Engineering, Saitama University in Partial Fulfillment of the Requirements for the Degree of DOCTOR OF ENGINEERING in Mathematics, Electronics and Informatics

Supervisor: Professor Yutaka OHSAWA

Saitama University

2013, March
Abstract

In this thesis, we study query algorithms for location-based services (LBS) in road network distance. Nowadays, spatio-temporal database research and technology have promoted a diversion of LBS applications in the real world. For example, car navigation systems, pedestrian navigation systems, logistic systems, travel information systems are some well-known LBS applications. Additionally, various spatial query methods have recently been proposed for LBS applications. Some typical queries are nearest neighbor queries, range queries, spatial join queries, and trip planning queries (TPQ). For these queries, the target points of a query are points of interest (POIs) which are, for instance, restaurants, convenience stores, and gas stations. Moreover, because spatial data are multidimensional data, data access methods and algorithms for spatial queries become an important role. Therefore, the efficient indexing methods, R-tree indexing structure, for spatial query processing are basically applied to spatial databases.

Spatial query methods were mainly based on Euclidean distance in the beginning, and then their focus shifted to the road-network distance. The difference between using Euclidean distance and road-network distance is in the calculation cost. For example, the Euclidean distance between two arbitrary points can be computed easily, however, in the road-network distance, it takes longer processing time. In the existing literature, to rapidly obtain the distance between two points, we can use some well-known shortest path finding algorithms such as Dijkstra's algorithm and A* algorithm for spatial queries in LBS application. These algorithms are well
applicable for the shortest path queries between two points. However, when users want to get the shortest paths to multiple targets in parallel by using usual existing algorithms, the processing time becomes very long. Therefore, the simultaneous searching algorithms from a single source to multiple target points are practically necessary.

In this thesis, we proposed a single source multi-targets A* (SSMTA*) algorithm to find the shortest paths to multi-target points simultaneously. This algorithm can control the unnecessary duplicated node expansions in spatial query than in the usual A* algorithm because the nodes are expanded at most once. We focused on the incremental Euclidean restriction (IER) strategy which rapidly generates a set of candidate POIs based on their Euclidean distance from a query point. And then, the Euclidean distance for each candidate is verified by computing their road network distance. At this road network distance computation, we applied the proposed SSMTA* algorithm.

For trip planning queries such as optimal sequenced route (OSR) and trip planning query (TPQ), we proposed a fast incremental algorithm to find OSR candidates following IER framework. There is no previous study on the incremental search for OSR with IER framework. When IER framework is applied, OSR candidates in Euclidean distance are incrementally generates first. Let the shortest OSR given by searches in Euclidean space be $S_r$ and its verified length in the road network be $L_N(S_r)$. The shortest OSR in the Euclidean space is not always the shortest OSR in the road network distance. Therefore, all OSRs whose lengths are less than $L_N(S_r)$ also have the potential to be the shortest route in the road network. All OSRs less than $L_N(S_r)$ must be searched in the Euclidean space, and then the results must be verified in the road network. Finally, the shortest OSR in the road network is returned as the result. These are the essential steps of an OSR query based on the IER framework. We also proposed an efficient verification method in the road network distance. Besides OSR with IER framework, we proposed algorithms based on the incremental network expansion (INE) for OSR queries.
To evaluate the performance of the proposed SSMTA* algorithm, we applied it to several spatial queries; $k$ nearest neighbor $k$-NN, aggregate nearest neighbor (ANN) queries. Besides, SSMTA* algorithm is applicable to other spatial queries in LBS applications.

Firstly, the SSMTA* algorithm is applied to $k$-NN query in which $k$-NN candidates are incrementally generated in Euclidean distance. Then, these are verified in road network distance by SSMTA* algorithm. Through the performance evaluation, the proposed method compared with existing works; the INE using Dijkstra’s algorithm, the pairwise A* algorithm and the lower bound constraint (LBC). The proposed method outperformed existing works in processing time and in expanded node number. The processing times of the pairwise A* algorithm and the LBC increase rapidly when the density of POIs is low or the number of $k$ is large. On the pairwise A* algorithm, nodes can be expanded several times when $k$ is large, which increases the hard-disk access times. On the LBC, although the number of node expansions remains low, the cost of PQ scanning increases in proportion to $k$ and the number of node expansions. This performance deterioration is serious when the density of the POI is low. The expanded node number and processing time of Dijkstra’s algorithm remains twice that of the proposed method. From the viewpoint of database systems, however, the number of disk accesses dominates the total calculation time. Accordingly, the proposed method is at least twice as efficient as the other methods.

In applications to ANN queries, a set of $k$-number of ANN candidates are incrementally generated with Euclidean distances by using the minimum bounding method (MBM) method with the R-tree index. By applying the SSMTA* algorithm, the road-network distances from each ANN candidate point in the set to all query points are verified. We proposed three methods; ANNPQ, ANNQP and ANNQPLB for ANN queries and compared them with existing works. The intensive experiments show that SSMTA* algorithm outperformed the existing work in terms of processing time and expanded nodes number.
For the simple trip planning (STP) query based on IER framework, the proposed the bi-directional distance constraint (BDDC) algorithm which based on the SSMTA* algorithm is compared with existing works. With various experimental results, we examined BDDC substantially outperformed previous methods in terms of both the expanded node number and the processing time.

For OSR queries, we used two frameworks INE and IER. We proposed algorithms USVPG, BSVPG based on INE framework and evaluated. Comparing with the existing progressive neighbor exploration (PNE) method, the proposed methods is almost 100 times faster than PNE by thorough experiments. With IER framework, the proposed method EOSR query significantly outperformed PNE, particularly when POIs are densely distributed or the number of POI categories to be visited during the trip is large. We also proposed an algorithm to determine only one shortest route; however, the top $k$ shortest routes are sometimes required to facilitate users’ choices. The algorithm proposed in this study can be easily adopted for this requirement because the EOSR generates candidates incrementally and the algorithm for verifying the road network distance can be easily applied to $k$ OSR queries.
Contents

Abstract 1

1 Introduction 9

1.1 Location Based Services ......................... 9

1.1.1 Incremental Network Expansion Approach .......... 10

1.1.2 Incremental Euclidean Restriction Approach ......... 11

1.1.3 Node Expansions .................................. 11

1.1.4 Type of Queries in LBS .......................... 12

1.2 Contributions ......................................... 14

1.3 System Environment for Experiments .................. 15

1.4 Outline ............................................... 15

2 Related Work 17

2.1 Basic Concepts of Query in LBS application ............ 17

2.2 Spatial Access Method ............................... 17

2.2.1 R-tree ............................................. 18
2.3 Shortest Path Searching Algorithms .......................... 20
  2.3.1 Dijkstra’s Algorithm .................................. 21
  2.3.2 A* Algorithm ....................................... 23

2.4 Typical Spatial Queries ..................................... 25
  2.4.1 Nearest Neighbor Query and Variants .................. 26

2.5 Shortest Path Searching Queries ............................ 28

2.6 IER Framework and the Most Related Queries ............. 29

3 Single Source Multi-target A* (SSMTA*) Algorithm for Queries on Road Network 33
  3.1 SSMTA* Algorithm ....................................... 33
    3.1.1 Related Work ...................................... 34
    3.1.2 Preliminaries ..................................... 35
    3.1.3 Basic Algorithm .................................. 35
    3.1.4 Properties of the SSMTA* algorithm ............... 39

3.2 Incremental Single-source Multi-target A* Algorithm ........ 41
    3.2.1 $k$-NN search by IER .............................. 43
    3.2.2 Incremental SSMTA* Algorithm for $k$NN Query .... 45
    3.2.3 Performance Evaluations of Incremental SSMTA* Algorithm to $k$NN Query ...................... 46

3.3 Summary .................................................. 51

4 Aggregate Nearest Neighbor Query Based on IER Framework 55
4.1 Application to ANN queries ........................................ 55
  4.1.1 Performance Evaluation for ANN Queries ................. 57
4.2 Modified SSMTA* Algorithm ........................................ 60
  4.2.1 ANN Queries with Modified SSMTA* Algorithm ............ 61
  4.2.2 Types of ANN Queries ......................................... 63
  4.2.3 Experimental Results for Modified SSMTA* Algorithm to
        ANN Queries ....................................................... 65
4.3 Summary .................................................................. 68

5 Simple Trip Planning Query Based on IER Framework .... 71
  5.1 Introduction ........................................................... 71
  5.2 Preliminaries .......................................................... 73
    5.2.1 Simple Trip Planning Query .................................. 73
    5.2.2 Basic Concept for Path Finding ............................... 74
    5.2.3 Existing Algorithms for STPQ ................................. 75
  5.3 STP Query Method Based on IER ................................. 77
    5.3.1 STP Query in the Euclidean Distance ....................... 77
    5.3.2 Verification on Road Network Distance ..................... 82
    5.3.3 Bi-directional Distance Constraint Algorithm ............. 83
  5.4 Performance Evaluations .......................................... 88
  5.5 Summary .................................................................. 90

6 Optimal Sequenced Route Query Based on INE and IER Frame-
work

6.1 Optimal Sequenced Route (OSR) Query .................................. 91
6.2 OSR Query Applying A* Algorithm ......................................... 92
   6.2.1 OSR Query by A* Algorithm ............................................ 93
   6.2.2 Bidirectional Search ..................................................... 96
6.3 Suppressing Duplicated Node Expansion ................................. 97
6.4 Experimental Results ......................................................... 101
6.5 Overview on USVPG and BSVPG algorithms ............................. 103
6.6 Incremental Queries in the Euclidean Distance ......................... 105
   6.6.1 OSR Queries in the IER Framework ................................. 105
   6.6.2 Simple Trip Path Query in the Euclidean Distance ............. 108
   6.6.3 Application to Multiple POI Categories ............................ 110
6.7 Experimental Result .......................................................... 113
6.8 Summary ................................------------------------------ 119

7 Conclusion ............................................................ 121

Acknowledgement .......................................................... 126
CHAPTER 1

Introduction

1.1 Location Based Services

Spatio-temporal database research and technology have promoted a diversion of location-based service (LBS) applications in the real world. Advanced transportation systems such as car navigation system, pedestrian navigation system, logistic system, travel information systems are some well-known LBS applications in our daily world. In LBS applications, diversified query methods have recently been proposed to meet the needs of users in their daily life. The target points of a query are points of interest (POIs) which are, for instance, restaurants, convenience stores, and gas station. These query methods were mainly based on Euclidean distance in the beginning, and then, their focus shifted to the use of road-network distance. The remarkable difference between using Euclidean distance and road-network distance is in the calculation cost. For example, the Euclidean distance between two arbitrary points can be calculated easily, while calculating the road-network distance between the same points requires much calculation. Moreover, the calculation of the shortest path and the road network distance between two points can be considered
as a basic operation in LBS. To offer an accurate and precise query result in powerful LBS applications, this basic operation should be taken by the cost effective way. In existing literature, to rapidly obtain the distance between two points, we can use some well-known shortest path finding algorithms such as Dijkstra's algorithm [1] and the A* algorithm [2]. On the other hand, query algorithms based on road network distance have been investigated since Papadias et al.'s [3] pioneering study. They proposed two methods, the incremental Euclidean restriction (IER) and the incremental network expansion (INE).

1.1.1 Incremental Network Expansion Approach

Given a query point $q$ and a set of POIs $P$, a $k$-NN query searches a specified number of ($k$) POIs, which are located closest to $q$. Papadias et al. [3] proposed two types of algorithms for this query. One is incremental network expansion (INE), which searches neighbor POIs by gradually enlarging the search area centered at $q$ using Dijkstra’s algorithm. The blue shaded area in Figure 1.1 indicates the search area on the road network of a $k$-NN query. Here, enlarging the search area starting from $q$, POIs on a road network are searched, and the search is terminated when a specified number $k$ (in this figure $k=10$) of POIs have been found. Although

![Figure 1.1: kNN query with INE approach applying Dijkstra’s algorithm](image)

incremental network expansion is still applicable, it has limited pruning capability on queries with selective spatial constraints. It is suggested that INE is better
performance for range query and nearest neighbor query.

1.1.2 Incremental Euclidean Restriction Approach

The other type of method is incremental Euclidean restriction (IER). The basic idea of this approach is to first search a set of $k$-NN points $C$ on the Euclidean distance using the R-tree [4], and to then confirm that points in $C$ are truly $k$-NN points on the road network distance. Generally, $k$-NN POIs on the Euclidean distance are not always $k$-NNs on the road network distance. Thus, adding the next neighbor on the Euclidean distance to the candidate set, the road network distances of the candidate points are calculated until no more candidates can be a member of the $k$-NN result. The distance between a pair of points, namely, a query point and a candidate POI, can be calculated using the A* algorithm, which is usually more efficient than Dijkstra’s algorithm. We hereinafter refer to this A* algorithm as the pair-wise A* algorithm.

However, while applying A* algorithm, when POIs trend to one side of the query point, and the search areas can be overlapped each other. This means that a node is visited several times during a $k$-NN query. Then, the calculation time of the pair-wise A* algorithm sometimes exceeds that of INE (based on Dijkstra’s algorithm). In particular, for a high POI density and a large $k$ value, multiple areas are overlapped, and the efficiency of the pair-wise A* algorithm deteriorates. Therefore, to avoid multiple overlapped area or, in other words, duplicated node expansion becomes a critical issue. Hereafter, steps of node expansion is discussed in detail.

1.1.3 Node Expansions

The basic operation in the shortest path search is node expansion, and the areas indicated by the dotted lines in Figure 1.1 are the expanded nodes area. Node expansion consists of the following steps:
(1) get the node \( n \), which has the minimum cost from the priority queue,

(2) get all nodes adjacent to \( n \) using the adjacency list,

(3) calculate the cost of each adjacent node,

(4) compose a record for each node then add the records to the priority queue.

Both Dijkstra’s algorithm and the A* algorithm perform node expansion.

Ordinarily, since the size of the adjacency list is large, the list is divided into several small blocks. Using this adjacency list, when a record of a node in a block must be referred, the block containing the referring node is read into a least recent used (LRU) buffer, then adjacent nodes are investigated. Although the processing time depends on the hit ratio of the LRU buffer, the processing time of the shortest path search increase almost linearly with the number of expanded nodes. The more the processing time is reduced for spatial queries, much better performance can be expected. Therefore, efficient query algorithms are vital especially in spatial queries.

### 1.1.4 Type of Queries in LBS

Hereafter, some kinds of queries with visible examples are explained in detail to better understand and know the diversity of spatial queries. At this point, various types of queries can be mainly classified as spatial join queries and trip planning queries depend on the purpose of users’ choices. In LBS, target points of a query (POIs) which include, for instance, restaurants, convenience stores, and gas stations are specified to be searched. Then POIs which satisfy the search conditions, are searched. \( k \) nearest neighbor (\( k \)-NN) queries, aggregate nearest neighbor (ANN) queries, spatial skyline queries, and trip planning queries are examples of these queries.

**kNN query** : Given a query point \( q \) and a set of POIs \( P \), a \( k \)-NN query searches
a specified number of \((k)\) POIs, which are located closest to \(q\). For example, if an on site engineer has got an outdoor work, and then at lunch time, he wants to find one of nearest restaurants to have his lunch. At this situation, \(k\)NN query can find a nearest restaurant and up to \(k\) nearest restaurant from his working area based on the distance or time as weights.

**Aggregate nearest neighbor (ANN) query**: ANN queries determine points of interest (POI), which are minimal, from a query point set \(Q\) according to specified aggregate functions (e.g., \(\text{sum, min and max}\)) used for calculating the users’ traveling distance. For example, three friends who are staying in different locations want to have dinner together at a restaurant. Therefore, using an appropriate aggregate function, the aggregate distance to a suitable restaurant can be determined. If the \(\text{sum}\) aggregate function is applied, a query result will be provided identifying a restaurant that requires the minimum of the total travel distance from three friends.

**Optimal Sequenced Route (OSR) Query**: The current position, the final destination, and some number of point of interest (POI) categories visited during the trip are specified in advance. Then, the query searches the shortest route from the current position with stops at one of each specified POI category from the visiting sequence before reaching the final destination. For example, a restaurant, a department store, and a movie theater may be visited before reaching the final destination; however, the visiting order is already specified. The department store should be visited first, next the restaurant, and finally the movie theater.

The calculation cost for OSR query is

\[
\prod_{i=1}^{m} N_i
\]

where, \(m\) is the number of visiting POI categories, \(N_i(1 \leq i \leq m)\) is the
number of POIs in category $i$. The calculation cost increases according to $m$.

**Trip planning query (TPQ):** When the starting and final destination points and POI categories to visit during the trip are given, TPQ find the shortest route from the starting point to the destination visiting POI one-by-one from the given POI categories during the trip. The visiting order among the given POI categories is not considered in this query. Because of this free visiting order, the complexity of the TPQ is NP-hard, as in the traveling salesman problem. The calculation cost for TPQ query is

$$m! \prod_{i=1}^{m} N_i$$

As we mentioned earlier, both Dijkstra’s algorithm and A* algorithm can be used to perform the network distance computation in spatial queries. However, applying these algorithms to very large spatial database and more complicated queries has high cost. Therefore, in this thesis, we study query algorithms for location based services in road network distance to alleviate the unwanted node expansions in searching and long processing time problems in former algorithms.

## 1.2 Contributions

In this thesis, we studied a single source multi-targets A* (SSMTA*) algorithm to find the shortest paths to multi-target points simultaneously. This algorithm overcomes the critical problem of unnecessary duplicated node expansions in usual pair-wised A* algorithm because by using SSMTA* algorithm, it is certainly controlled to happen one node is expanded exactly once.

More importantly, in our study, we focused on IER strategy which uses a filtering mechanism to rapidly generate a set of candidate POIs based on their Euclidean distance from a query point. And then, the Euclidean distance for each candidate is verified by computing their road network distance. At this road network distance
computation, we applied SSMTA* algorithm and evaluated the efficiency of our proposed algorithm. Hence, we applied SSMTA* algorithm to different spatial join queries such as $k$NN, ANN and the simple trip planning query which is a query type with most simplicity for trip planning queries based on IER approach.

Additionally, to cope with more complicated trip planning queries such as OSR and TPQ, we proposed a fast incremental algorithm to find OSR candidates following IER framework. Furthermore, an efficient verification method for the road network distance is also introduced. Especially in OSR query, we presented another discussion based on INE framework, and we compared our proposed methods with the existing works to prove the efficiency and stability of proposed algorithms.

1.3 System Environment for Experiments

In this thesis, we implemented intensive experiments. We applied Java programming language in implementation. We used a real road network and randomly generated POIs. The road networks used in the experiments are the area of Saitama City, Japan, which has 25,586 road segments (hereafter denoted as “Road-1”), and the area of Saitama Prefecture, which has 468,666 road segments (denoted as “Road-2”). The positions of POIs are generated by a pseudo-random sequence with a specified probability ($Prob$). For example, $Prob = 10^{-3}$ indicates a POI on one thousand road segments. All algorithms are implemented by Java and are evaluated on a PC with an Intel Core i7 CPU 960 (3.2 GHz), 9 GB memory.

1.4 Outline

Our main interest is query algorithms for LBS, and background knowledge is described in Chapter (1). In Chapter (2), some related work are explored. Proposed methodologies and applications are presented in Chapter (3). To measure the efficiency of our proposed methods, we applied our proposed algorithm in different queries in LBS application, and each experimental evaluation for respective query
are discussed in Chapter (4). In Chapter (5), trip planning queries are presented. The thesis is concluded in Chapter(6).
CHAPTER 2

Related Work

2.1 Basic Concepts of Query in LBS application

In this chapter, we introduced two main topics (1) some background information with respect to the critical mechanism for spatial queries in LBS application including spatial access method, well-known shortest path searching algorithm (2) the most related topic to our study, spatial queries in LBS application.

2.2 Spatial Access Method

LBS applications have higher demand and advanced spatial databases play in the main role for modern applications environment. Spatial databases store various kinds of multidimensional data represented by points, line segments, polygons, other kinds of geometric entities. Spatial databases include specialized systems like Geographical Information Systems [[5][6][7][8][9]] multimedia databases and so on. However, the role of spatial databases is continuously and importantly changing. Besides, new types of data such as spatiotemporal data and data handling become critical for spatial databases. According to well handling approach, we can control
query processing time in LBSs application. Here, the main thing should be taken into consideration is that spatio-temporal data is dynamic nature and it is not sufficient to simply store and retrieve like in relational databases. The key characteristic is manipulation for spatial data. Such manipulation mainly consists of queries processing related to the spatial data. Some typical queries are range queries \cite{10,11}, nearest neighbor queries \cite{12,13,14,15,16,17} and another types are more complicated queries such as trip planning queries, optimal sequence route query \cite{18}, spatial join queries such as $k$ nearest neighbor queries \cite{19,20,21} skyline queries \cite{22,23,24} and so on. To support such queries efficiently, the most specific data structures are necessary. At this point, traditional data structures like (B-trees \cite{25}, Hashing methods) are not adequate for spatial indexing due to multi-dimensional purpose. Several extensions of B-trees, such as the $B^+$ tree, the R-tree structure and other variants \cite{26,27,28,29} are also proposed over last years. Among the most popular indexing methods for spatial query processing, R-tree is basically applied to spatial databases. In the next section, we will present R-tree detail.

\subsection{R-tree}

R-tree originally proposed by \cite{4} is a generalized data structures with the purpose of the multidimensional access method. It is similar with $B^+$ tree \cite{30} which is a standard access method for one dimensional data aimed for the relational database. In R-tree index structure, spatial objects are represented by the minimum bounding rectangles (MBRs). There are various evolutions of R-tree discussed in \cite{31}, however, here, we first introduced about basic concept of original R-tree by Guttman \cite{4}. By definition, R-tree is high balanced tree, and at least two children can be existed at the root. Every node contains between $m$ and $M$ entries, where $m$ is the minimum number of entries of a leaf node and it can be varied, however, $m \leq M/2$ always stands. $M$ is the maximum number of entries in a leaf node. Assume that $a$, $b$, $c$, $d$, $e$, $o$, $p$, $w$, $x$, $y$, $z$ in figure 2.1 are schools in Saitama City. Then, the left side in the figure shows the logical structure of existence of MBRs bounding
spatial objects, and the right side shows the hierarchical structure of the R-tree. Each node of R-tree corresponds to the MBR that bounds its children. The leaf nodes contain pointers to the database objects. An entry in a node represents a spatial area. All leaves appear on the same level. The nodes are implemented as disk pages and are dynamically indexed. In Guttman’s presented R-tree, searching, deletion and insertion of spatial data are discussed. In searching spatial data, MBRs in any levels may be overlapped each other, and hence, many nodes may require to be searched before searching MBR is reached. Instead of searching spatial data directly, fast access to MBRs in R-tree can reduce the processing cost. However, to get efficiency of R-tree indexing, it is also needed to minimize the overlapping area of MBR of nodes. In insertion, it is necessary to split into two nodes when there is no more vacant leaf for new entry because maximum allowed entries have already been occupied. Then, a leaf node in which the new entry can be placed is chosen and the split algorithm tries to find the area whose MBR needs the minimum space for its respective MRBs to accommodate the new entry. If there is a vacant leaf to include the new entry, it is simply inserted into the leaf and the respective path between the inserted leaf to the root is updated. This splitting method is named

Figure 2.1: The Simple R-tree
linear split. The quadratic split and exponential split are also classified to reduce the unwanted overlapped area between MBRs and to keep the MBRs as less enlargement as possible. Searching in R-tree may need to search more than one subtrees under a node as bounding rectangles in each level may be overlapped. To minimize searching, it is also necessary to reduce dead-space. Given a rectangle $S$, all children if their bounding rectangles overlap with $S$ are searched. Searching cost depends on the structure of the tree, and hence, it should be improved by minimizing the area of MBR of nodes. Deletion in R-tree is different with B$^+$ tree in handling of an underflowing node. In B$^+$ tree, an underflowing case can be handled by merging two sibling nodes as its index is one-dimensional data and two siblings nodes will contain consecutive entries. However, R-tree index is for multi-dimensional and this property does not hold. Therefore, sometimes, it is needed to condense a tree due to the elimination of one node. When an entry $E$ has been deleted from the leaf node $L$, $L$ has fewer than $m$ entries and leaf $L$ is entirely removed and all its entries are reinserted. At this point, reinsertion gives the same result as merging for two R-tree nodes that are sorted at the same level. Updates are propagated upwards and the MBRs in the path form root to $L$ are modified. [32]

2.3 Shortest Path Searching Algorithms

When a road network ($G(V, E)$) and two points which are source($s$) and target($t$) are given, the shortest path between $s$ and $t$ on the graph can be easily found, and it is called the single source shortest path search. For instance, a tourist wants to find the quickest route from one sightseeing spot to another on a road map; in this case vertices represent locations and edges represent segments of road and are weighted by the time or distance needed to travel that segment. This kind of query is a basic function in car navigation system, and also in various spatial queries which are applied in LBS application in the road network distance. Shortest path searching algorithms have been studied since 1950’s and various data structures and algorithms for them have also been studied. However, they can
mainly be classified as two ways (1) on-the-fly methods; methods such as applying adjacency list of nodes, (2) precomputation methods; methods such as applying road network distance materialization approaches. Two well-known shortest path searching algorithms are Dijkstra’s algorithm [[1][33]] and A* algorithm [35] for the on-the-fly approach. However, A* algorithm is normally faster than Dijkstra’s algorithm. Moreover, bi-directional search algorithms have also been studied based on these two algorithms. In the next section, we will briefly discuss about some well-known shortest path searching algorithms.

![Figure 2.2: The shortest path search](image)

### 2.3.1 Dijkstra’s Algorithm

Given a source node \( s \), Dijkstra’s algorithm originally proposed by [1] will compute the shortest path from the source node \( s \) to all other reachable nodes by using a best-first search approach when a graph \( G = (E, V) \) and source vertex \( s \in V \) are given. This algorithm works on both directed and undirected graphs, however, all edges must have non-negative weights. The detail algorithm is described as following. Initially, the distance to source vertex is initialized as zero. Then, all other distances are set to infinity. Two sets \( C \) and \( Q \) are prepared. The set \( C \) is the set of visited vertices and it is initially empty. The set \( Q \) initially contains all vertices. While \( Q \) is not empty, the element \( u \) with the minimum distance is selected from \( Q \), and \( u \) is added to \( C \) as a visited node. Then, the distances of all
adjacent nodes to $u$ are computed and updated. This process terminates when $Q$ is empty. In the figure 2.2, to find the shortest path from $S$ to $E$, the following steps occur. First, the set $C$ is initially empty set, and $a, b, f, e$ are adjacent nodes to source node $S$. The distances of adjacent nodes to source node $S$ are computed and cost is updated in $Q$ as follow. Then, the source node $S$ is added into the set $C$.

$$Q = Sa(3), Sb(3), Sf(3), Se(4)$$

Next, the node with the minimum cost is fetched from $Q$, in this case, let $Sa(3)$ is selected from $Q$. Again, adjacent nodes of $a$ are $S, b$ and $f$. Among these nodes, $S$ is already expanded and not needed to expand again. Thus, their distance from $S$ are calculated, and $a$ is added into $C$.

$$Q = Sb(3), Sf(3), Se(4), Sab(8), Saf(9)$$

$$C = S, a$$

Then, in the next loop, $Sb(3)$ is selected from $Q$ and node $b$ is expanded, and added into $C$. Therefore, in $Q$, the cost will be as following.

$$Q = Sf(3), Se(4), Sab(8), Sabc(8), Saf(9)$$

$$C = S, a, b$$

This process is repeated, and let say the shortest path $SedE(10)$ is inserted in $Q$ and if that cost is selected from $Q$ for next expansion, the shortest path between $S$ and $E$ has already determined. By handling to store vertices in an array or linked list, the complexity of this algorithm is $O(|V|^2) + |E|$. However, [35] proposed an approach that can reduce the complexity of the original Dijkstra’s algorithm to $O(|E| + |V|log|V|)$. For sparse graphs, it can be implemented more efficiently by storing the graph in an adjacency list using a heap or priority queue.
2.3.2 A* Algorithm

The A* algorithm searches the shortest path from the origin \( q \) to the destination \( p \) more efficiently than Dijkstra’s algorithm and estimates the cost \( Cost \) from \( q \) to \( p \) via a current node \( n \), by means of the following equation:

\[
Cost = d(q, n) + h(n, p)
\]

where \( d(q, n) \) is the actual cost of moving from \( q \) to \( n \) on a road network, \( h(n, p) \) is the estimated cost between \( n \) and \( p \), and the value must be lower than the actual cost. The value returned by \( h(n, p) \) is the lower bound of the actual cost, then for any two points \( a \) and \( b \), the condition \( d(a, b) \geq h(a, b) \) is satisfied. We can assume several types of cost, such as travel distance, travel time, and travel expense, to be minimized. Among these costs, when we minimize the travel distance, the Euclidean distance can be used as the estimated cost. Moreover, when we minimize the travel time, we use the travel time for moving by the expected fastest speed between two points. In the remainder of the explanation, we use the distance on the road network as the cost. Moreover, targets of the search considered throughout this study are points of interest (POIs). However, these POIs do not always exist on a road network node, for simplicity, we assume that POIs are located on a node throughout the present paper. This restriction, however, can be easily removed [36].

The A* algorithm controls searching by a priority queue \( PQ \). During processing, the following record is composed and inserted into \( PQ \).

\[
< Cost, N_C, N_P, d_N(q, N_C), RLink(N_P, N_C) >
\]  

(2.1)

Figure 2.3 summarizes the symbols in this record. \( Cost \) has been described above. \( N_C \) is the currently intended node, and \( N_P \) is the previously visited node on the path from \( q \) to \( N_C \). \( d_N(q, N_C) \) is the path length from \( q \) to \( N_C \) on the road network,
and $RLink(N_P, N_C)$ is the pointer to the road link connecting $N_P$ and $N_C$. Then, $Cost = d_N(q, N_C) + d_E(N_C, p)$, which is the sum of the road network distance from $q$ to $N_C$ and the Euclidean distance between $N_C$ and $p$.

![Figure 2.3: A* algorithm](image)

At the beginning of the search, the following record is inserted into an empty $PQ$.

$$< d_E(q, p), q, - , 0, - >$$  \hfill (2.2)

Here, $-$ denotes the NULL value. When the search begins, $N_C$ is $q$. Then, no $N_P$ exists. Moreover, $RLink(N_P, N_C)$ does not exist. Then, the NULL value is assigned to these two items. In order to avoid expanding nodes that have already been examined, the once expanded node is inserted into a closed set $CS$, which is initialized by an empty set at the beginning of the search.

The A* algorithm then repeats the following steps until the current node reaches the destination $p$.

(1) Remove the minimum $Cost$ record from $PQ$ and place this record in $CS$.

(2) Obtain all nodes neighboring $N_C$ by means of the adjacency list.

(3) For each neighboring node, calculate $Cost$, compose a record shown by Eq. 3.1, then insert $Cost$ into $PQ$. 

This sequence of steps is a node expansion of the A* algorithm. Among these steps, step (2) needs to access the adjacency list. When the entire adjacency list of the road network is small enough to be stored in the main memory, no disk access occurs during the shortest path search. However, the size of the adjacency list is generally large. Therefore, the adjacency list is divided into several small blocks, some of which are read into the LRU buffer, and is then referred to in order to investigate the adjacency nodes. Hence, the number of expanded nodes directly affects the processing time. The A* algorithm is preferred to Dijkstra’s algorithm because the former expands a much smaller area of nodes than the latter when there is only one destination.

When a record is removed from $PQ$ and the $N_C$ of this record reaches the destination $p$, all nodes on the shortest path from $q$ to $p$ have been included in $CS$. Then, the path can be obtained from $CS$ according to the following steps.

1. $n \leftarrow p$

2. Search the record $r$ in $CS$ for which $N_C$ reaches $n$

3. Obtain the pointer to the road link connecting $n$ and $r.N_P$ (i.e., $r.RLink(N_P, n)$)

4. $n \leftarrow r.N_P$, then repeat the procedure from step (1) until $n$ reaches $q$.

In the abovementioned path restoration, the basic operation is to find the record for which $N_C$ reaches the query node $n$. In order to perform this search quickly, $CS$ is implemented by a hash table or a balanced tree.

### 2.4 Typical Spatial Queries

In spatial databases, the fundamental proximity query types are nearest neighbor (NN) query and range query. These two types of queries have been actively researched in various LBS applications.
The nearest neighbor query is a popular query type aiming to retrieve the closest neighbor to a query point from a set of given objects. Based on different constraint conditions, NN query processing can be classified into three categories, such that in Euclidean spaces (e.g. [37][38][39]), in spatial networks (e.g.[40][41][42][43]) and in higher dimensional spaces ([44][45][46]). With the R-tree variants [47][48] of spatial indices, depth first search [38] and best first search [37] have been the representative branch-and-bound techniques for processing nearest neighbor queries.

Range queries select items that overlap a given query window and range query is, perhaps, the most common, and has been widely used as the subject of analysis in other related studies [[49][50][51][52][53][54][55]] as well. Range queries are used when data items will be selectively retrieved and by measuring number of nodes accessed in the index structure while servicing the query, I/O and CPU processing costs can be controlled.

2.4.1 Nearest Neighbor Query and Variants

Several types of NN queries and indexing structures for spatial query have been proposed in past decade [[56][57][58][59][60][61]]. NN queries in spatial databases can be classified major categories: simple $k$-NN queries [[62][63][37][64][38]], $k$-NN join queries [65], approximate nearest neighbor query [66], constrained $k$-NN queries [67], continuous neighbor queries CNN [[68][69][70][71]] and reverse NN queries [72][73]. Recently nearest neighbor search solutions have been extended to support queries on spatial networks. Regarding simple $k$-NN queries, there have been several approaches. Jensen et al. [42] proposed data models and graph representations for $NN$ queries in road networks and designed corresponding solutions. Shahabi et al. [74] applied graph embedding techniques to $k$-NN search on road networks. They transformed a road network to a high-dimensional Euclidean space where traditional NN search algorithms can be applied. They showed that their method has a good approximation of the $k$-NN in the road network. However, their method involves spatial indexes for high-dimension and it leads to poor performance. Further,
the query result is approximate and the precision very much depends on the POI density and distribution. [75] presented an efficient and flexible query framework, ROAD, based on search space pruning by using shortcuts for accelerating network traversals and object abstracts for guiding traversals. In [76], $k$-NN based on users’ preferences, was studied and this query selects the best spatial location with respect to the quality of facilities in its spatial neighborhood. For instance, a customer may want to rank the rooms with respect to the appropriateness of their location, defined after aggregating the qualities of other features such as restaurants, hospitals, schools, markets, etc within a distance range from them. In this situation, a spatial preference query ranks objects based on the qualities of features in their spatial neighbor objects.

Kollios et al. [66] proposed a method for indexing spaces with arbitrary distance measures for retrieval of approximate nearest neighbor. However, all LBS applications are not only based on distance measures, and when queries for time based is needed to offer, their approach is not adequate.

Another type of nearest neighbor search with rather limited attention is CNN query [77]. CNN query finds the nearest neighbor NN of every point on a line segment (e.g., find all my nearest convenient store during my route from my school, point $(s)$ to my home point $(d)$). The result contains a set of (point, interval) tuples, such that point is the NN of all points in the corresponding interval. The first algorithm for CNN query processing, proposed in [19] uses static branch-and-bound algorithm and query operation is minimized by using information of previous query and pre-fetched results. However, incorrect pre-fetched results leads no accuracy guarantee and there is a significant computational overhead. Cho et al. [71] proposed a unique continuous search algorithm UNICONS for NN queries and CNN queries. In their approach, precomputed information is used. When POI density is low, UNICONS requires a larger portion of network to be retrieved due to increase in the searching area. It degrades its performance compared with VN$^3$ method proposed by [40]. Kolahdouzan et al. [78] continuous $k$-NN queries to spatial network
database where the distance between two objects is defined as the length of the shortest path between them.

2.5 Shortest Path Searching Queries

In many LBS applications (e.g., logistics and supply chain management), users have to plan a trip to a number of locations with several sequence rules and the goal is to find the optimal route that minimizes the total traveling distance. One related query type is named the optimal sequenced route (OSR) query proposed by [79]. The OSR query was first proposed by Sharifzadeh et al.[80]. They proposed several algorithms for an OSR query to operate on the Euclidean distance. Among them, the light optimal route discoverer (LORD) first finds a greedy route which is composed by the successive nearest neighbor search. The greedy route is found by performing a consecutive NN search from the starting point to the last visiting category. The search area is restricted by the length of this greedy route. Then, the LORD finds the optimal route in the reverse order (from the last category to the starting point), by narrowing the search area. The authors also proposed a more efficient algorithm called the R-LORD (R-tree-based LORD). However, these algorithms cannot be adapted directly to the road-network distance. Hence, for road network distance query, they proposed another algorithm named progressive neighbor exploration (PNE).

During almost the same time, Li et al. proposed the trip planning query (TPQ) [81]. The TPQ is similar to an OSR query; however, the visiting order of the POI is not specified in the TPQ. Because of this free visiting order, the complexity of the TPQ is NP-hard, as in the traveling salesman problem. Therefore, Li et al. proposed several types of approximation algorithms. However, these algorithms cannot be directly applied to road networks, because of the heavy burden of the NN search. For TPQ on a road network, Li et al. proposed the minimum distance query (MDQ) algorithm. Basically, the MDQ expands nodes on the road network
successively, finding the NN POI in the same way as by Dijkstra’s algorithm. This causes duplicated node expansion, and the processing time increases, especially when multiple trip-plan routes (k-TPQ) are requested.

Chen et al. [82] proposed another type of route query called the multi-rule partial sequenced route (MRPSR) query. This query generalizes both the OSR and the TPQ. For example, suppose we want to visit a bank, a restaurant, and a movie theater in that visiting order. A user may want to visit a bank before visiting both the restaurant and the movie theater because he needs to withdraw some money. However, the order of visiting to the movie theater and the restaurant can be exchanged. In this case, the visiting order is specified as a semi-ordered set, which can be represented as a directed graph. They called this graph an activity on vertex (AOV) network.

2.6 IER Framework and the Most Related Queries

Compared with calculating the Euclidean distance, calculating the road network distance is computationally heavier. The time for calculating the shortest path between two points increases with the path length. As a result, there has been much research on finding shortest path query methods more efficient and reducing the computation times for road networks. The methods developed have generally taken one of two approaches: hypothesis verification or pre-processing. This section summarizes these methods.

Papadias et al. [3] proposed two approaches to apply to several kinds of queries, including k-NN queries regarding road network distance. They are incremental node expansion (INE) and incremental Euclidean restriction (IER). In the former, Dijkstra’s method is simply used to search k-nearest neighbor points of interest (POIs). In the latter, hypothesis verification is used. IER method finds k-NN points on the basis of the Euclidean distance before verifying the distance on the road network by using Dijkstra’s algorithm. Yiu et al. adapted this approach to
ANN queries [36].

In addition to pre-processing on a road network, several other types of methods have been proposed. Hu et al. [83] proposed a distance signature method which is a kind of materialization method of the road network distance. For a k-NN query, node information is added to an adjacency list which indicates the neighboring node leading to each POI. The shortest path from any node to a target POI on the road network can be found by simply tracing the path from one node to the next until the target POI is reached. Though, this method requires a data amount of $O(nm)$, where the $n$ is number of nodes on the road network and the $m$ is the number of POIs, it works very efficiently for finding the shortest path from a node to a POI. However, if points are added to or removed from the set of POIs, the re-construction to overall elements in the adjacency list is necessary.

Samet et al. [84] generalized this method into one for finding k-NN points in a best-first manner. This method is based on precomputation of the shortest paths between all possible nodes on the road network. The information for the next visited node is compressed using shortest path quad-tree, resulting in $O(n^{1.5})$ storage cost.

Kolahdouzan et al. [40] proposed Voronoi based network nearest neighbor (VN3) algorithm that searches for k-NN points using a network Voronoi diagram. While this method works well for bi-directional road networks, it can provide only an approximate shortest route if uni-directional roads are included. Zhu et al. [85] used the network Voronoi diagram for ANN queries.

Another precomputation method was proposed by Shaw et al. [86]. It uses an M-tree [87], a general-purpose data structure for metric space, like an R-tree for Euclidean space. This structure needs the calculated distance between two points during query processing, and this calculation can become very heavy for a road network because the shortest path calculation cost increases with the distance between two points. To overcome this problem, Zhu et al. proposed an approximation query method. Ioup et al. [88] used an M-tree for ANN queries in metric space.
In order to rapidly obtain the distance between two points on a road network, several precomputation approaches have also been investigated. The basic idea is that the distance between two points are precomputed and are then referenced when requested. However, this requires a large amount of data storage. The space complexity is \(O(|V|^2)\) for a graph \(G(V,E)\), where \(V\) is a set of nodes that consist of intersection points or ends of a road segment, and \(E\) is a set of links that connect nodes. Furthermore, obtain the shortest path between any two points on \(G\) requires a space complexity of \(O(|V|^3)\). For instance, if \(|V| = 10^6\) (1M(mega)) on a road network, then \(|V|^2\) will be \(10^{12}\) (1T(tera)), and \(|V|^3\) will be \(10^{18}\) (1E(exa)). Hence, for a large road network, materialized methods are not practically suitable for distance and path calculations.

Huang et al. [89] proposed a semi-materialization approach. When a node set \(N\) is given, for every pair of \(s\) and \(d\) \((s,d \in N)\), the next visiting node and the distance of the shortest path from \(s\) to \(d\) is recorded. This approach requires a storage complexity of \(O(|V|^2)\). The distance between any pair of nodes can be obtained by \(O(1)\) operations. When a source node \(s\) and the destination node \(d\) are given, the shortest path can be obtained by tracking the next visiting node repeatedly. Then, the shortest path can be obtained by \(O(|V|)\) operations. Jing et al. [90] adopted this approach in the hierarchical encoding path view (HEPV), which hierarchically partitions the adjacency list. This structure constructs \(s-d\) pairs inside the hierarchically divided area and then reduces the total amount of data for the adjacency list.

A number of semi-materialization approaches have recently been proposed. Hu et al. [83] proposed distance indexing \(k\)-NN on a road network. In this method, the distance signature was introduced and was used to maintain the approximate network distance between nodes and objects. However, this method requires re-calculation when a new POI is added or an element in the POI set is deleted, the re-calculation extends over the entire network. Samet et al. [84] generalized the method and reduced the space complexity to \(O(|V|^{1.5})\). However, this also
requires a complicated data structure in order to compress the data of the next visiting node.

The network Voronoi diagram is another precomputation method for $k$-NN query. Kolahdouzan et al. [40] proposed a $k$-NN query algorithm called VN$^3$ on the network Voronoi diagrams. Although this method is suitable for bi-directional road networks, it can only provide an approximate shortest route when uni-directional road segments are included.

Ciaccia et al. [87] proposed a general-purpose M-tree (metric-tree) data structure for similarity search in metric space. Shaw et al. [86] adopted the M-tree for $k$-NN query on the road network distance. However, they used a high-dimensional data structure to obtain a distance between two points on the road network, and the structure gives only an approximation distance. Moreover, in general, the road network distance is quasi-metric when uni-directional edges are included, i.e., the symmetry property required in metric space is not always true. Hence, in practice, M-tree is not suitable for use as an index structure in the road network distance.

In this study, we proposed a single-source multi-target A* (SSMTA*) algorithm which provides shortest paths concurrently for multiple target points in [91]. However, it had the following deficits. (1) The lower bound distance to the destination cannot be determined during searching. (2) Then, it cannot terminate the searching when the lower bound distance which is obtained from the priority queue exceeds a limit value. (3) It needs to expand nodes more than LBC-KNN proposed by Deng et al. [92]. Therefore, our study presented the improvement of SSMTA* algorithm overcoming the deficits, and then it is applied to spatial queries.
CHAPTER 3

Single Source Multi-target A* (SSMTA*)
Algorithm for Queries on Road Network

3.1 SSMTA* Algorithm

Route searching on a road network is an essential operation in location based services, a search for the shortest path between two points on the network is necessary for various spatial query applications. Dijkstra’s algorithm [1] and the A* algorithm [2] are representative algorithms used for such searches. Another frequent search is to find the shortest paths between starting point $q$ and multiple target points $p(\in P)$. Such searches are needed, for example, in ANN (aggregate nearest neighbor) queries [36], skyline queries [23], k-NN queries [3], and several kinds of trip planning queries [80][81].

While Dijkstra’s algorithm can be directly applied to these queries, its performance is considerably degraded when the target points are distributed with bias or when some are distant from the starting point (see Figure 3.1(a)). A* algorithms can also solve these queries by iterating each pair-wise query $|P|$ times. Its performance is also degraded when $|P|$ is large, because neighboring road network paths
to \( q \) are repeatedly processed (see Figure 3.1(b)). The inefficiency of this method increases with the size of \(|P|\).

In this chapter, we described an algorithm that can efficiently find the individual distances to several target points. This single-source multi-target A* (SSMTA*) can find the shortest paths in ascending order of the distance from \( q \), and it remains efficient even for very distant target points. The basic operation in the shortest path search is to find neighboring nodes by using an adjacency list and to calculate the cost in terms of distance or time at each node. This operation is called node expansion. SSMTA* expands a road network node at most once, the same as with Dijkstra’s algorithm and in contrast to the conventional A* algorithm. Figure 3.1 shows the expanded node area for (a) Dijkstra’s algorithm, (b) the A* algorithm, and (c) our SSMTA* algorithm.

The contributions are the followings.

- This is the first attempt of the A* algorithm to single-source multi-target situation.
- The SSMTA* algorithm was applied to an ANN as an example. It can be adapted for use with a wide range of location based service queries.
- Its efficiency was demonstrated experimentally.

### 3.1.1 Related Work

The most closely related to the present study is the lower bound constraint (LBC) approach proposed by Deng et al. [92]. The LBC is another single-source multi-target A* approach. They adopted the LBC for \( k \)-NN query, range query, closest pair query, and spatial skyline query. The LBC outperformed the INE and pair-wise A* for these queries in terms of expanded nodes number. However, the LBC uses multiple priority queues (PQs), and the contents of these PQs must be synchronized. This synchronization requires a great deal of computation.
3.1.2 Preliminaries

In location based services, several types of POIs can be query targets. They can include nearby gas stations, hotels, and restaurants. Such facilities are POIs. Although a POI may not reside on a node in the road network, in the following we assume that every POI is on a network node. To resolve this restriction is straightforward [36].

Let set $P$ contain $k$ POIs, and let a starting point $q$ be given. How to construct $P$ from a large number of POIs is application dependent. We first consider the process for finding all paths from $q$ to each $p(\in P)$ and calculating their distance on the road network. This can be solved using by Dijkstra’s algorithm starting from $q$, or by using the A* algorithm in a repetitive manner between $q$ and each $p$. In this section, we describe our more efficient method for doing this, the single-source multi-target A* (SSMTA*) algorithm.

3.1.3 Basic Algorithm

Here, we described our SSMTA* algorithm to search for the shortest paths from a specified starting point to several target points on the road network. Table 3.1 summarizes the symbols used.
Table 3.1: Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P)</td>
<td>Target point set selected from POIs</td>
</tr>
<tr>
<td>(q)</td>
<td>Starting point, called query point in ANN query</td>
</tr>
<tr>
<td>(Q)</td>
<td>Query point set in ANN query</td>
</tr>
<tr>
<td>(PQ)</td>
<td>Priority queue</td>
</tr>
<tr>
<td>(CS)</td>
<td>Closed set</td>
</tr>
<tr>
<td>(d_E(x, y))</td>
<td>Euclidean distance between (x) and (y)</td>
</tr>
<tr>
<td>(d_{E}^{\min}(x, P))</td>
<td>Minimum road network distance between (x) and a point in (P)</td>
</tr>
<tr>
<td>(d_N(x, y))</td>
<td>Road network distance between (x) and (y)</td>
</tr>
<tr>
<td>(RLink(x, y))</td>
<td>Pointer to road segment with edges (x) and (y)</td>
</tr>
</tbody>
</table>

Figure 3.2: SSMTA* Algorithm

In Figure 3.2, \(q\) is the starting point, and \(p_1\) to \(p_4\) are the target POIs in set \(P\). The purpose of the query is to determine the shortest path from \(q\) to each point in \(P\). Suppose that the path on the road network from \(q\) to \(n\) has already been determined, then \(n\) is currently noticed as a current node. \(n\) has three directly neighboring nodes (\(n_a\) to \(n_c\)). We first explain using \(n_a\). The path length from \(q\) to \(n_a\) via \(n\) is \(d_N(q, n) + d_N(n, n_a)\). Let \(d_{E}^{\min}(n_a, P)\) denote the distance between \(n_a\) and the point \(p(\in P)\) that gives the minimum Euclidean distance. In the figure, \(p_1\) is the closest POI to \(n_a\), so \(d_{E}^{\min}(n_a, P) = d_E(n_a, p_1)\). Then, \(Cost = d_N(q, n) + d_N(n, n_a) + d_{E}^{\min}(n_a, P)\) is used to determine for the next expanded node. For each node neighboring \(n\), this cost is calculated and then inserted into the priority queue \((PQ)\), which controls the search order. We call this basic operation node expansion.
The records in the priority queue have the following format.

\[
< \text{Cost}, N_C, N_P, d_N(q, N_C), RLink(N_P, N_C) >
\]  

(3.1)

The \( N_C \) represents the node of current interest (the *current node*). The \( N_P \) is the previous node of \( N_C \) on the path from \( q \) to \( N_C \). The \( d_N(q, N_C) \) is the road network distance from \( q \) to \( N_C \), and \( RLink(N_P, N_C) \) is the (pointer for the) road segment connecting \( N_P \) to \( N_C \). For example, in Figure 3.2, \( N_C \) corresponds to \( n_a \), \( N_P \) to \( n \), and \( d_N(q, N_C) \) to \( d_N(q, n_a) \).

At the beginning of the SSMTA* algorithm, the record \( < d_{\text{in}}^E(q, P), q, -, 0, - > \) is inserted into the \( PQ \). The ‘\(-\)’ of the third item shows the *null* value, because \( q \) does not have a previous node. In the same way, \( q \) does not have a previous link, so the *null* value is also assigned to the fifth item.

The steps in the SSMTA* algorithm are similar to those in the conventional A* algorithm. (1) Get the record having the minimum cost from the \( PQ \); (2) check the current node (\( N_C \)) of the record to see whether it has been expanded (if the record has already been expanded, ignore it); (3) make a new record for each neighboring node and insert it into the \( PQ \). Repeat (1) to (3) until all POIs have been visited. For the check in step (2), a once expanded node is inserted into the *closed set* (CS). The CS keeps all records that were obtained in step (1) indexed by the current node \( N_C \). If the \( N_C \) of the record extracted in step (1) is already in the CS, the node has already been expanded, so the node is ignored.

Let the record obtained from the \( PQ \) be \( e \). If \( e.N_C \) matches \( p(\in P) \), one POI has been found. The corresponding node is then removed from \( P \). The number of points in \( P \) is reduced each time a POI is found. The CS is used to retrieve the route from the found POIs (target points) to \( q \) (the starting point) in a reverse manner by tracing the \( N_P \), the previous node. It is the same way as with Dijkstra’s algorithm or the A* algorithm. First, \( e.N_C \) is searched for in the CS, then the \( N_P \)
of this record is searched for in the CS. This searching is repeated until \( N_P \) meets \( q \). When \( N_P \) meets \( q \), the route can be obtained. Algorithm 3 shows the pseudo code for the SSMTA* algorithm.

**Algorithm 1 SSMTA**

1. \( R \leftarrow \emptyset \)
2. \( d_{min} \leftarrow \min(d_E(q, p_i), p_i \in P) \)
3. \( enqueue(<d_{min}, q, -0, ->) \)
4. **loop**
5. \( e \leftarrow deleteMin() \)
6. **if** \( CS.Contains(e.N_c) \) **then**
7. \( CS.renew(e.d_N(q, e.N_P)) \)
8. **end if**
10. **if** \( e.N_C \in P \) **then**
11. \( R \leftarrow R \cup <e.N_C, getPath(e.N_C) > \)
12. \( P \leftarrow P - e.N_C \)
13. **if** \( P = \emptyset \) **then**
14. \( \text{return } R \)
15. **end if**
16. **end if**
17. **for all** \( nn \in neighbor(e.N_C) \) **do**
18. \( \text{decide } p(p \in P) \text{ which gives } d_E^{min}(nn, P) \)
19. \( d_N \leftarrow d_N(q, e.N_C) + d_N(e.N_C, nn) \)
20. \( enqueue(<d_N + d_E^{min}(N_C, P), nn, e.N_C, RLink(N_c, nn)>) \)
21. **end for**
22. **end loop**

In Dijkstra’s algorithm and the conventional A* algorithm, after a record is obtained once from the \( PQ \) and inserted into the CS, the distance from \( q \) to current node \( N_C \) in the record is fixed. In contrast, in the SSMTA* algorithm, the value \( (d_N(q, N_C)) \) can be changed as illustrated in Figure 3.3. The numbers in parentheses show the order in which the records were obtained from the \( PQ \).

Now, suppose node \( m \) has been obtained from the \( PQ \) (1), and that \( Cost_a = d_N(q, a) + d_E(a, p_2) \) for node \( a \), \( Cost_b = d_N(q, b) + d_E(b, p_1) \) for node \( b \), and \( Cost_n = d_N(q, n) + d_E(n, p_1) \) for \( n \) have been calculated. This means that the nearest \( p(\in P) \) for \( b \) and \( n \) is \( p_1 \), and for \( a \) is \( p_2 \). These three records are created and inserted into
the PQ. If $\text{Cost}_a > \text{Cost}_n > \text{Cost}_b$, the record for $b$ is obtained from the PQ first (2). $b$ is expanded and somehow the search path reaches $p_1$, which is then removed from $P$. In this situation, $P$ remains only $p_2$ in the example.

Next, $n$ is obtained from the $PQ$ (3) and expanded. Through this node expansion, records for $a$ and $o$ are created and $n$ is inserted into the $CS$. This record contains the network distance for the $q \rightarrow m \rightarrow n$ route.

Suppose $a$ is obtained from $PQ$ (4) next. It is expanded and its record, $d_N(q, n) = d_N(q, a) + d_N(a, n)$, is inserted into the $PQ$. Later, the record for $n$ is obtained from the $PQ$ (5). However, $n$ had already been placed in the $CS$ with $d_N(q, n) = d_N(q, m) + d_N(m, n)$. Now, suppose that if $d_N(q, a) + d_N(a, n) < d_N(q, m) + d_N(m, n)$, the $d_N(q, n)$ in the $CS$ should be replaced with a smaller value, $d_N(q, a) + d_N(a, n)$. This is why the distance for a record in the $CS$ needs to be altered in the SSMTA* algorithm.

Figure 3.3: Cost change in CS

### 3.1.4 Properties of the SSMTA* algorithm

The following Lemma 1 is the basis for restoring the shortest path from $CS$ entries.

**Lemma 1.** For each node $N_C$ on the shortest path from $q$ to $p$, the value $d_N(q, N_C)$ of a record in $CS$ is the correct shortest path distance from $q$ to $N_C$. 
Proof. Each record in CS is assigned a provisional network distance \( d_{N}(q, N_C) \) from \( q \) to the current node of the record \( N_C \). This means that another shorter path to \( N_C \) may exist. However, after once a target POI \( p \) is removed from \( PQ \), its Cost value is fixed to \( d_{N}(q, p) \), which is the minimum Cost value in the \( PQ \). In other words, no shorter path can exist. The shortest path between \( q \) and \( p \) is then fixed.

Let \( p_P \) be the previous neighboring node to \( p \) on the shortest path. \( d_{N}(q, p) \) was calculated by the equation \( d_{N}(q, p) = d_{N}(q, p_{P}) + d_{N}(p_{P}, p) \). Here, \( d_{N}(q, p) \) is the shortest distance on the road network. Then, \( d_{N}(q, p_{P}) \) is also the shortest distance from \( q \) to \( p_{P} \). By repeating this until \( p_{P} \) meets \( q \), \( d_{N}(q, N_C) \) in all records in \( CS \) along the shortest path to \( p \) is the shortest path length between \( q \) and \( N_C \). \( \square \)

Lemma 2. Given a destination point set \( C \), the SSMTA* algorithm finds \( p(\in C) \) in ascending order of the road network distance.

Proof. Let two points \( p \) and \( p'(\in C) \) be considered and let \( d_{N}(q, p) < d_{N}(q, p') \) be satisfied. Assume that \( p' \) is reached in advance of \( p \). Then, just before \( p' \) is reached, \( PQ \) contains the following two records as the cost value, \( d_{N}(q, n_a) + d_{E}(n_a, p) \) and \( d_{N}(q, n_b) + d_{N}(n_b, p') \) (see Figure 3.4). Here, \( n_b \) is a directly neighboring node to \( p' \). Since \( p' \) is visited before \( p \), the condition

\[
d_{N}(q, n_b) + d_{N}(n_b, p') < d_{N}(q, n_a) + d_{E}(n_a, p)
\]

holds. However, by the premise, \( d_{N}(q, p) < d_{N}(q, p') \), and \( d_{E}(n_a, p) \leq d_{N}(n_a, p) \), then

\[
d_{N}(q, n_b) + d_{N}(n_b, p') < d_{N}(q, p).
\]

This contradicts the hypothesis. Then, \( p \) should be reached before \( p' \). This means that the SSMTA* algorithm finds the POI in \( C \) in ascending order of the road network distance. \( \square \)
3.2 Incremental Single-source Multi-target A* Algorithm

The SSMTA* algorithm requires that all elements of the POI set \((P)\) be given in advance of starting the query. If a new POI \((r)\) is added to \(P\) after the search has started, the path found from \(q\) to point \(r\) is not necessarily granted the shortest path. A node on the shortest path from \(q\) to \(r\) may not have been expanded, because the \(Cost\) of a record in the \(PQ\) had already been determined when it was inserted, and thus was not changed even though a new point had been added.

After all the shortest paths to \(P\) have been found, each node in the \(CS\) has an exact shortest path distance from \(q\). Then, when a distance \(q\) to any point in the \(CS\) is requested to retrieve, the correct distance can be easily obtained by referring to the \(d_N(q, r)\). The shortest route from \(r\) to \(q\) can be obtained by using the basic algorithm described in 3.1.3.

Papadias et al.[3] proposed an incremental algorithm for \(k\)-NN queries, and Yiu et al.[36] proposed an incremental algorithm for ANN queries. In these algorithms, a new target point is incrementally added to set \(P\). The SSMTA* algorithm can handle this situation in one of two ways.
The first way is to simply search for enough POI candidates using a Euclidean distance query and then to validate the road network distance. This process is repeated until the necessary number of POIs has been found. This approach, however, is problematic. The calculation of $d_{\text{min}}^E(n, P)$ becomes heavier as the number of points in $P$ increases because the number of Euclidean distance calculations is proportional to the number of points in $P$. An even bigger problem is deciding on a sufficient number of points in $P$ before applying the SSMTA* algorithm. The node expansion must also be considered. However, the increase in the total number of node expansions is not proportional to the increase in the number of points in $P$, because POIs distant from $q$ do not become target until the node expansion approaches the node.

The second way is to re-calculate the $PQ$. Suppose that the SSMTA* algorithm has already found all the points in $P$ and that the $PQ$ still contains several unextracted records. If a new point ($r$) is added to $P$, $\text{NewCost} = d_N(q, N_C) + d_E(N_C, r)$ for all $PQ$ entries is calculated, and Cost is replaced with the NewCost value. In Dijkstra’s algorithm, the number of records in the $PQ$ is proportional to the distance to the farthest point in $P$. In the SSMTA* algorithm, determining the size of the $PQ$ is not so simple. However, the $PQ$ still contains the nodes surrounding the expanded nodes (i.e. the nodes in the CS), so the number of records in the $PQ$ should be proportional to the distance from $q$ to the farthest POI in $P$. Additionally, the NewCost calculation can be carried out in main memory, so disk access is not necessary. Moreover, this calculation does not affect the total calculation time very much. Besides the number of points to be added is not restricted, so one or more points can be added simultaneously.

Hereafter, we briefly described the $k$-NN search based on IER as the basis of the proposed method.
3.2.1 \textit{k-NN search by IER}

The incremental Euclidean restriction (IER) proposed by Papadias et al. \cite{Papadias2007} searches \(k\)-NN by the following steps. Here, the set of POIs is supposed to be indexed by an R-tree \cite{Guttman1984}.

1. Search \(k\)-NN in the Euclidean distance using the R-tree, and then store the results in a candidate set \(C\). Subsequently, search \((k+1)\)th NN \(p_n\) incrementally.

2. For each point \((p_i \in C)\), calculate the road network distance \(d_N(q, p_i)\).

3. If \(d_N^k(q, C) \leq d_E(q, p_n)\), then return the top \(k\) POIs in \(C\) and then stop. Here, \(d_N^k(q, C)\) is the road network distance between \(q\) and the \(k\)-th NN in \(C\).

4. Calculate \(d_N(q, p_n)\). Then, if \(d_N^k(q, C) > d_N(q, p_n)\), add \(p_n\) to \(C\). Find the next \(p_n\) by an incremental search on the R-tree.

5. Goto Step (3).

In the above algorithm, an incremental nearest neighbor search is required. This search can be performed on an R-tree by a best-first search using a priority queue \cite{Papadias2007}.

Since the \textit{Cost} used in the A* algorithm is the lower bound of the shortest path length from \(q\) to \(p\), when a record that has the minimum value is obtained from the \(PQ\) and \(d_E(q, p_n) \leq \text{Cost}\) stands, the reserved point \(p_n\) can become a candidate of \(k\)-NN. Then, if multiple A* algorithms can run in parallel, joining \(p_n\) to the candidate set \(C\), the searching can be started targeting \(p_n\) immediately. In contrast, a pairwise A* algorithm will not finish until the search reaches the target. Then, if the target is locates very far away with respect to the road network distance, the search expends a great deal of calculation time and the resulting path may be useless.

Based on this idea, Deng et al. \cite{Deng2011} proposed the LBC, which can run the A* algorithm in parallel. The LBC uses multiple priority queues, each of which is
assigned to a target point. The node to be expanded is obtained from the set of
PQs that has the minimum Cost value among the set. Then, the new records
composed by the node expansion process are inserted into all PQs. However, the
Cost value can differ depending on the target point. Figure 3.5 explains the LBC.
In this figure, q is the query point, and points p1 to p4 are POIs, the shortest paths
from q of which have been determined. The dotted line shows the wave front, which
indicates the positions of the nodes contained in PQs. The left-hand side of this
figure shows the PQ array, and each PQ manages the cost for a different target
point.

\[\text{Figure 3.5: LBC algorithm}\]

The LBC works very efficiently from the viewpoint of the expanded node number.
Intuitively, the expanded areas of pair-wise A* algorithms overlap each other as
shown in Figure 3.1(b). This means that the same node is expanded repeatedly by
the individual pair-wise A* algorithm. On the other hand, for the LBC, the node
expansion is always performed once. The serious shortcoming of the LBC; however,
is in the synchronization of the nodes contained in the priority queues. After a
record that has the minimum cost is removed from a PQ (for example n in Figure
3.5), the record must be searched in all other PQs and is removed. This operation
in PQ has a linear calculation cost and is repeated every time a node is expanded.
As such, the calculation cost becomes very high especially when k is large.
3.2.2 Incremental SSMTA* Algorithm for $k$NN Query

Algorithm 2 ISSMTA*

Require: $k, q$ // $k$: number of POIs to be found, $q$: query point
Ensure: $R$ // Result POI set with the shortest paths

1: $R \leftarrow \emptyset$
2: $j \leftarrow 0$
3: $C \leftarrow E_kNN(k, q)$
4: $p_n \leftarrow \text{NextENN}(q)$
5: $d_{min} \leftarrow \min(d_E(q, p_i), p_i \in C)$
6: $\text{enqueue}(d_{min}, q, -1, 0, -)$
7: loop
8: $\langle \text{deleteMin}() \rangle$
9: if $CS.\text{Contains}(e)$ then
10:     continue;
11: else
12:     $CS.\text{add}(e.CN, e.NP, e.DN, e.RLink >)$
13: end if
14: if $e.CN \in C$ then
15:     $R \leftarrow R \cup e.CN, \text{getPath}(e.CN)$
16:     $C \leftarrow C - e.CN$
17:     $j \leftarrow j + 1$
18:     if $j \geq k$ then
19:         return $R$
20: end if
21: $\text{RenewQueue}(e.CN)$
22: end if
23: if $e.Cost > d_E(q, p_n)$ then
24:     $C \leftarrow C \cup p_n$
25:     $p_n \leftarrow \text{NextENN}(q)$
26:     $\text{RenewQueue}(e.CN)$
27: end if
28: for all $nn \in \text{neighbor}(e.CN)$ do
29:     decide $c_i$ which gives minimum $h(nn, c_i)$
30:     $d_N \leftarrow d_N(q, e.CN) + d_N(e.CN, nn)$
31:     $\text{enqueue}(d_N + h(nn, c_i), nn, e.CN, d_N, RLink(nn, e.CN >)$
32: end for
33: end loop
When the SSMTA* algorithm applies to $k$-NN search by IER, a new target point can be inserted into the candidate set $C$. As with using the pairwise A* algorithm described in 3.2.1, this insertion occurs when the Cost of a record dequeued from PQ exceeds the Euclidean distance from $q$ to the reserved POI ($p_n$) (the reserved candidate is the $k+1$-th POI in the initial state). At that time, the Cost value of each record in $PQ$ is recalculated over the new $C$, and the SSMTA* algorithm continues the search until all $k$-NN have been found. Algorithm 2 shows a pseudo-code of this incremental SSMTA* (ISSMTA*) algorithm, that finds $k$NN in the road network distance. $j$ in line 2 counts the number of POIs found shortest paths (see also line 17). $EkNN(k, p)$ searches $k$NN of $q$ in Euclidean distance, and $NextENN(q)$ in line 4 searches next NN of $q$ in Euclidean distance incrementally. Line 23 checks whether $e.Cost$ exceeds $d_E(q, p_n)$. When the result is true, $p_n$ is added to $C$, and $p_n$ is replaced by the next NN. In line 26, each Cost value of PQ records is recalculated over modified candidate set. By this incremental operation, the SSMTA* algorithm is applicable to other types of spatial queries, such as ANN and spatial skyline queries.

### 3.2.3 Performance Evaluations of Incremental SSMTA* Algorithm to $k$NN Query

This section evaluates the performance of the proposed algorithm by comparison with several conventional methods.

The adjacency list used in this experiment is prepared using Peano-Hilbert order as the same with Papadias et al’s experiment in [3]. We prepared a 16KB block adjacency list. The size of the LRU buffer was set to 1MB (64 blocks). The adapted replacement policy was the popularly used “clock” [93], which acts similar to the LRU. The dividing method of the adjacency list using the Peano-Hilbert order is a simple method. However, the performance is similar to that of Huang’s method [94]. The $k$-NN search results among INE, the pairwise A* algorithm (PWA*),
the LBC-KNN, the SSMTA* proposed in [91] (SSMTA*org), and ISSMTA* are compared in experiments. Fig. 3.6 and Fig. 3.7 are the result obtained using Road-2 under $Prob = 0.005$. Fig. 3.6 shows the expanded node number. The LBC-KNN and the ISSMTA* show almost the same values. Therefore, both lines overlap each other. When the $k$ value is small (less than 5), the expanded node number in PWA* is small. However, the expanded node number increases more rapidly than in the other methods when $k$ increases. This is because one node is expanded several times during a $k$-NN search. The vertical axis in Fig. 3.7

Figure 3.6: Relation between $k$ and expanded node number $Prob = 0.005$

Figure 3.7: Relation between $k$ and processing time $Prob = 0.005$
shows the processing time in seconds. The tendency of the increase is almost the same with the expanded node number, except for the LBC-KNN and the PWA*. The processing time of the LBC-KNN increases rapidly when the \( k \) value is large. This is because the cost of the heap operation, which is executed every time for each node expansion, increases according to the \( k \) value increase. Contrary, the processing time of the PWA* is relatively faster in spite of much expanded nodes number, because the buffer hit ratio of the PWA* is high. The other three methods, INE, SSMTA*org, and ISSMTA*, exhibit stable characteristics. Among them, the ISSMTA* method is the most efficient with respect to both the expanded node number and the processing time. Fig. 3.8 and and Fig. 3.9 show the processing time and the expanded node number in the \( k \)-NN search when \( Prob = 0.05 \), which is ten times denser than that of Fig. 3.6 and Fig. 3.7. The expanded node number shows the same tendency except for the LBC-KNN and the PWA* as same as in Fig. 3.6. In comparing Fig. 3.9 with Fig. 3.7, the increase of LBC-KNN is alleviated. This is because the total expanded node number decreases to a tenth.

Varying the POI density(\( Prob \)), the expanded node number and the processing time are measured. Fig. 3.10 and 3.11 are the result on Road-1, and 3.12 and 3.13 are the result on Road-2. In this experiment, \( k \) value was fixed to 20. The
processing time in all methods decreases according to the increase of the POI density because the size of the search area also decreases. Among these methods, the ISSMTA* algorithm is the most efficient in all POI densities. Deng et al. [92] evaluated the performance of the LBC-KNN among the range of $Prob = 0.05$ to 2.0. Over these very high POI density distributions, the search area remains small, and the inefficiency of the LBC-KNN does not become apparent. However, when schools, hospitals, convenience stores, and gas stations are considered as POIs, their densities are not so high [40], and such kind of POIs does not exist beyond 0.05. Then, the ISSMTA* algorithm is expected to perform with considerably high
efficiency in actual applications in comparison with PWA* and the LBC-KNN. In this experiment, the results on densities lower than 0.002 for Road-1 are omitted because the entire area on the road network is searched for cases in which $k = 20$. However, the result using Road-2 shows that the proposed method considerably outperforms PWA* and the LBC-KNN in such low-density cases. When a set of POI is distributed with bias, the result can be predicted as follows; (1) INE requires to search with area on the road network, (2) the areas of expanded node in PWA* become to overlap each other especially near the query point, (3) the expanded
nodes number in the LBC-KNN and the ISSMTA* are the same; however, the LBC-KNN requires much processing time than the ISSMTA* because the average road network distance becomes longer with biased distribution of POIs. Fig. 3.14 shows the result of the experiment to confirm this prediction. The biased sets of POI are prepared by the following method; (1) determine the query point $q$, (2) generate POIs on the road network nodes that belongs on the right half-plane of $q$ under the same POI density with uniform distribution dealt in the above experiments. Fig. 3.14 compares the expanded node number and the processing time by the ratio against the ISSMTA*. U-time and U-node in this figure show the result on the uniform distribution and B-time and B-node show the result on the biased distribution, respectively. The result meets the above mentioned prediction. As the conclusion, the ISSMTA* algorithm also outperforms the other methods when the set of POI is distributed with bias.

3.3 Summary

In LBC-KNN, it assigns a PQ set in which each PQ is assigned for each element of the candidate set $C$. The contents of all PQs must be synchronized to keep the wave front nodes identical among the PQs for different target points. This synchro-
SSMTA* algorithm achieves multi-targets search using a single PQ. Therefore, it does not require synchronization process that is essential in LBC-KNN. However, the original SSMTA* algorithm has a deficit that it increases number of expanded nodes in comparison with LBC-KNN. To overcome this deficit, a method is introduced to recalculate the Cost value of all records in PQ every time when a target point is deleted from (when the shortest path found) and a new target point is inserted to (in ISSMTA*) the set of target points (C). The correctness of the SSMTA* algorithm controlled by a single PQ is not obvious. Hence, we showed proofs for (1) SSMTA* gives the shortest paths (Lemma 1) and (2) SSMTA* finds the shortest path to the target points in ascending order of the length (Lemma 2). The latter attribute is necessary for kNN search in IER framework. By mentioned improvements, the proposed SSMTA* algorithm achieves multi-targets shortest path search by the same expanded node number with LBC-KNN and faster processing time than LBC-KNN.

Moreover, through the performance evaluation of the proposed method in comparison with the INE using Dijkstra’s algorithm, the pairwise A* algorithm and the LBC, we realized that the proposed method outperforms the existing works in
processing time and in expanded node number.

The processing times of the pairwise A* algorithm and the LBC increase rapidly when the density of POIs is low or the number of $k$ is large. The defects occur for the following reasons. On the pairwise A* algorithm, nodes can be expanded several times when $k$ is large, which increases the hard-disk access times. On the LBC, although the number of node expansions remains low, the cost of PQ scanning increases in proportion to $k$ and the number of node expansions. This performance deterioration is serious when the density of the POI is low.

Although, like the proposed method, Dijkstra’s algorithm exhibits stable performance, the expanded node number and processing time of Dijkstra’s algorithm remains twice that of the proposed method. A disadvantage of Dijkstra’s algorithm is that the performance deteriorates substantially when the distribution of the POIs trend toward one side. This biased distribution is apt to appear in the ANN and spatial skyline queries. The calculation cost of Dijkstra’s algorithm is $O(|E| + |V| \log |V|)$ on the road network $G = (V, E)$. The calculation cost of the A* algorithm varies depending on the heuristic function. However, the cost of searching the shortest path between two specified points is on the same order as Dijkstra’s algorithm. The worst case occurs when the heuristic function always returns 0. The expanded node number of the LBC and SSMTA* are the same. This means that the worst time complexity is the same for all of the algorithms considered in this paper. From the viewpoint of database systems, however, the number of disk accesses dominates the total calculation time. Accordingly, the proposed method is at least twice as efficient as the other methods.

We applied SSMTA* algorithm to ANN queries in [91]. The detail is discussed in the chapter (4).
CHAPTER 4

Aggregate Nearest Neighbor Query Based on IER Framework

4.1 Application to ANN queries

In an ANN query, a set of query points \( Q \) is given and \( k \) POIs are found in ascending order by evaluation value, which is based on the distance from a set of query points \( Q \) to \( p(\in P) \). The proposed distance evaluation functions include \( \text{sum} \), \( \text{max} \), and \( \text{min} \) [95]. When \( \text{sum} \) is used, the total distance from each query point in \( Q \) to \( p \) is calculated and used as the evaluation value.

The initial ANN query method proposed by Papadias et al. [96] was dubbed group nearest neighbor query. Since then, several other ANN query methods [95] have been proposed. Yiu et al. [36] proposed three methods for road network distance queries. Experimental evaluation showed that the IER (incremental Euclidean restriction) method outperformed the rest. The IER method is based on a three-step paradigm: (1) search for candidate ANN POIs on the basis of Euclidean distance, (2) evaluate the results on the basis of road network distance, and (3) repeat both steps until \( k \)ANN POIs have been found, which can be efficiently done using a best-first query.
on the R-tree index.

Figure 4.1 shows a simple example application of an ANN query method using the sum function. The ANN query points are $q_1$ and $q_2$. The POIs are $p_1$ to $p_4$; they are indexed by an R-tree. Initially, the minimum bounding rectangles (MBRs) in the root node of the R-tree are placed in a priority queue (PQ). In this example, $PQ=\{<6, R_1>, <11, R_2>\}$. The first item of this record shows the sum of the supposed minimum distances (MINDISTs) from each query point to an MBR. For example, the MINDIST from $q_1$ to $R_1$ is 4 and from $q_2$ to $R_1$ is 2, so $R_1$ has total cost of 6. Therefore, $<6, R_1>$ is dequeued because its cost is lesser. Descending the R-tree one level results in two points $p_1$ and $p_2$ in a leaf node being enqueued so that the $PQ=\{<10, p_1>, <11, R_2>, <15, p_2>\}$. The dequeuing of $p_1$ means that one ANN POI has been found. The next step is verification in which the sum distance on the road network is calculated. The ANN POIs can thus be found incrementally on the basis of the Euclidean distance. These generation and verification steps are repeated until the minimum sum of the road distance fall below the $n$-th ANN on Euclidean distance. This algorithm can easily be expanded to handle $k$ANN queries, meaning that it can be used to search up to the $k$-th minimum ANN. This algorithm can also be adapted simply to use the min and max functions [95].

Yiu et al. [36] proposed evaluating the distance on a road network using an A* algorithm. They used a pair-wise A* algorithm, so $|Q|$ pairs of calculations were
necessary. The SSMTA* algorithm can be adapted to do the same calculation and it should be able to do it more quickly. Two types of methods using the SSMTA* algorithm can be used for ANN applications. One type is straightforwardly adopting for each candidate of ANN result by Euclidean distance as starting point, and each in \( Q \) as the target. The second type is adapting the SSMTA* algorithm so that one of the \( q \ (\in Q) \) is used as the starting point and the \( k \)-ANN result set is used as the destination.

Given the characteristics of the SSMTA* algorithm, we can say that methods of the first type outperforms those of the second when \( |Q| \) is large, and that those of the second method are more suitable when \( k \) is large.

### 4.1.1 Performance Evaluation for ANN Queries

We experimentally evaluated the efficiency of the two types of SSMTA* algorithm methods for ANN searches. The ANN candidates were obtained using Yiu et al.’s method described above. Query cost was calculated using three methods. One used conventional A* algorithm as is used in the IER method [36], hereafter called ANN0. The second method used the SSMTA* algorithm to calculate the distance from each ANN query candidate point targeting all query points in \( Q \), hereafter called ANN1. The third calculation method used the SSMTA* algorithm to calculate the distance from each query point to all ANN candidate points found in a Euclidean \( k \)-ANN search, hereafter called ANN2.

In the Dijkstra, A*, and SSMTA* algorithms, four data sets are referred to while searching: PQ, CS, adjacency list, and road segments. The PQ and CS are usually small enough to reside in memory, so they can be quickly accessed. In contrast, the adjacency list and road segment ones are usually large to reside in memory so they are stored on disk. Node expansion is thus inevitably needed to access them, meaning that node expansion time dominates processing time. We thus evaluated the performance of the methods on the basis of the number of node expansions.
The first experiment was conducted with $|Q| = 3$ and $|Q| = 7$ and with the number of $k$ in a $k$-ANN search variable from 1 to 20. As shown in Fig. 4.2, the ANN1 calculation method did not show much improvement compared with ANN0, especially for $Q = 3$. In contrast, the ANN2 method showed considerable improvement compared to the other two methods especially when $k$ was large. For example, when $k$ was 20 and $|Q|$ was 3, the expanded node number was about 20% that of the other methods.

Figure 4.2: Relation between $k$ and expanded node number $|Q| = 3$

Figure 4.3: Relation between $k$ and expanded node number $|Q| = 7$
Figure 4.4: Relation between $|Q|$ and expanded node number

Figure 4.5: Relation between $Prob$ and expanded node number

Figure 4.4 shows the result, when $k$ was fixed at 5 and $|Q|$ was varied. The actual expanded node number for this query varies with the size of the $Q$ distribution area on the map. The actual results we obtained experimentally were evaluated using two methods, each as a ratio to the ANN0 result. The ANN2 ratio was fairly stable because the searched $k$ was the same for all $Q$. In contrast, the ANN1 ratio decreased with an increase in $|Q|$.

Figure 4.5 shows the results when $Prob$ (the probability of a POI existing on each road segment) was varied and $|Q| = 3$ and $k=5$. With the ANN0 and ANN1
methods, the number of node expansions increased with the $\text{Prob}$, while, with the ANN2 method, the number was fairly stable and independent of $\text{Prob}$.

These results show that the two types of methods using the SSMTA* algorithm have two intrinsic characteristics: the efficiency of the ANN1 method depends on the number of $k$ in a $k$ANN query while that of the ANN2 method depends on $|Q|$. This is because the SSMTA* algorithm works more efficiently when the number of POIs to be searched for is large. Unlike the conventional A* algorithm, which suffers heavily duplicated node expansions especially around the searched POI, the SSMTA* algorithm avoids duplicate node expansions.

\section*{4.2 Modified SSMTA* Algorithm}

The previously described deficits can be overcome by the following modification. Cost values of the entries in PQ are updated when the search has reached each target $p_i (\in C)$ and the target has been removed from $C$. To update the cost values in PQ, all records in PQ are re-calculated using cost equation, targeting only the rest of the target points in $C$. With this improvement, the number of node expansions to find all target points in $C$ is reduced to the same number of nodes used with LBC-KNN which we discussed in the previous chapter. The calculation cost for this process is directly proportional to the number of elements in PQ. However, the frequency of this process is proportional only to the number of elements in $C$. On the other hand, LBC-KNN uses multiple PQs and the synchronization of the contents in all PQs is necessary for each node expansion, and hence, the duration of the operation is proportional to $|C| \times \text{NumberOfNodeExpansion}$. Therefore, this process takes considerable processing time, and degrades the overall performance of LBC-KNN.

Algorithm 3 shows the pseudo-code of the modified SSMTA* algorithm. In this algorithm, the record of the PQ is composed with the following format.
< Cost, N_C, N_P, d_N(q, N_C), RLink(N_P, N_C) >  \quad (4.1)

In this equation, Cost is the cost value calculated by cost calculation equation. N_C is the current intended node, and N_P is the previously visited node on the path from q to N_C. d_N(q, N_C) is the path length from q to N_C on the road network, and RLink(N_P, N_C) is the pointer to the road link connecting N_P and N_C.

When the origin of the search (q) and a set of points (C) are given, this algorithm returns the set of the shortest path from q to each target point in C.

In Line 3, the initial record is composed and enqueued into PQ. The steps beginning from Line 4 are repeated until all points in C have been reached. When the N_C of the dequeued record e is already in CS (closed set), it is ignored (from line 6–8). In Line 9, the node is added to CS.

Line 10 checks whether the search reaches a POI in C. If so, the found POI and the shortest path from q to the POI are registered in the result set R. And, getPath(p) is the function for restoring the shortest path route using CS. The found POI is removed from C. As the result, if C becomes empty, the result set is returned and the algorithm is terminated.

The function, RenewQueue(e, N_C) in line 16, recalculates the Cost value in each record in PQ for the renewed C. As previously described, this operation reduces the total number of expanded nodes.

Lines 18 – 22 expand each node nn that is neighboring e.N_C. Neighboring nodes are found by referring to the adjacency list. The POI that satisfies \( d_{E}^{\text{min}}(nn, C) \) is determined, and a PQ entry for nn is then composed and enqueued in the PQ.

### 4.2.1 ANN Queries with Modified SSMTA* Algorithm

In this section, we applied the modified SSMTA* algorithm to ANN queries. In the first step, \( k \) number of ANN candidates in Euclidean distance against a query
Algorithm 3 Modified SSMTA*

Require: $q, C$
Ensure: $R$ (shortest paths set)
1: $R \leftarrow \emptyset$
2: $d_{\min} \leftarrow \min(d_E(q, p_i), p_i \in C)$
3: enqueue($<d_{\min}, q, -, 0, ->$)
4: loop
5:     $e \leftarrow$ deleteMin()
6:     if $CS$.Contain($e$) then
7:         continue;
8:     end if
10:    if $e.N_C \in C$ then
11:        $R \leftarrow R \cup <e.N_C, getPath(e.N_C)>$
12:        $C \leftarrow C - e.N_C$
13:    if $|C| = 0$ then
14:        return $R$
15:    end if
16:    RenewQueue($e.N_C$)
17:    end if
18:    for all $nn \in$ neighbor($e.N_C$) do
19:        decide $c_i$ which gives $d^E_{\min}(nn, C)$
20:        $d_N \leftarrow d_N(q, e.N_C) + d_N(e.N_C, nn)$
21:        enqueue($<d_N + d^E_{\min}(nn, C), nn, e.N_C, d_N, RLink(nn, e.N_C)>)$
22:    end for
23: end loop

point set $Q$ are generated by the MBM algorithm proposed by Papadias et al. [95].

Then, the generated ANN candidates are set into the candidate set $C(|C| = k)$.

In the second step, the cost values for the ANN candidates are verified in road-network distances by applying the SSMTA* algorithm described in Section 4.2. We presented two verification policies, hereafter called ANNPQ and ANNQP. The first policy, ANNPQ, verifies the road-network distance from each ANN candidate point in $C$ to all query points in $Q$. In contrast, ANNQP verifies the road-network distance from each query point in $Q$ to $p_i \in C$.

The following explanation focuses on the $sum$ aggregate function; however, the
method can also be applied to other functions. The main difference is the aggregate function used in MBM on Euclidean space.

4.2.2 Types of ANN Queries

ANNPQ

(1) In the first step, set $C$ with $k$-number of ANN candidates is incrementally generated with Euclidean distances by using the MBM method with the R-tree index.

(2) By applying the SSMTA* algorithm, the road-network distances from each ANN candidate point in $C$ to all query points in $Q$ are verified, and the result is then retained in the set $R$. We let the total road-network distance for the $k^{th}$ candidate be $\text{sum}_k^N$.

(3) Next, the candidate $p_m$ (in the first iteration $m = k + 1$) in the Euclidean distance is generated. The total distance is denoted as $\text{sum}_m^E$. If $\text{sum}_m^E > \text{sum}_k^N$, then the result set $R$ is returned and the search process is terminated.

(4) Using the SSMTA* algorithm, the road-network distances from $p_m$ to all $q_i \in Q$ are verified.

(5) When $\text{sum}_m^N < \text{sum}_k^N$, $p_m$ is added to set $R$ and the maximum element is removed from $R$. Then, the process repeats starting from step (3).

In step (3), ANN candidates in Euclidean distance are incrementally generated. While searching, if the total Euclidean distance $\text{sum}_m^E$ of the $m$-th ANN candidate, $p_m (m > k)$, is greater than the current $k$-th total road-network distance $\text{sum}_k^N$, then the total distance of no more candidate can be smaller than that of any member in $R$. Therefore, the searching is terminated. In step (5), if the total road network distance from $p_m$, i.e., $\text{sum}_m^N$, is less than the total road network distance from $p_k$
(\text{sum}_k^N), then the entry that has the maximum value in the result set is replaced with \( p_m \).

Another method ANNQP is described hereafter. In ANNQP, the distances from a query point to ANN candidates are calculated and the results for the same target point are summed to obtain the total distance to the target point. The number of \(|Q|\) searches runs concurrently in ANNQP. Each SSMTA\textsuperscript{*} algorithm has a PQ and a CS. Candidate points are incrementally joined in the candidate set \( C \). At this point in the process, the distance from the query point to the added candidate is not always increased. Instead, the candidate point sometimes might have already been included in the CS. In this case, the shortest path length from the query point to the candidate can be determined by only referring to the distance value of the record in the CS. This is justified by Property 1. The rest of the ANNQP flow can be implemented in the same way as in the ANNPQ.

The ANNQP can be improved using the lower bound distance which can be obtained by the \textit{Cost} value of the PQ record. Hereafter we refer to the modified version of the ANNQP as ANNQPLB. In this method, all PQs assigned to an individual query point are managed by a PQ group. When a dequeuing is requested to the PQ group, the PQ having the minimum \textit{Cost} record in the PQ group is determined, and the record is dequeued from the PQ. Using this improvement, the lower bound distance of the search area is expanded synchronously among the query points set.

Using the previously mentioned control, the lower bound total distance (\( LBTD \)) to a candidate point \((p_i)\) is obtained by the following equation:

\[
LBTD = \sum_{q \in Q} H(p_i, q)
\]  

(4.2)

In the equation, \( H(p_i, q) \) is \( d_N(p_i, q) \) if the road network distance from \( q \) to \( p_i \) is already determined by the search, otherwise, it is the \textit{Cost} value of the dequeued
The following shows the ANNQPLB flow.

**ANNQPLB**

1. $k + 1$ number of ANN candidates are obtained using MBM, the top $k$ of them is set to $C$, and the remaining ones are set to $p_m$.

2. PQ and CS are created for each query point, each PQ initialized, and a PQ group is composed by the PQs.

3. A record $r$ is dequeued from the PQ group.

4. If $LBTD > sum_m^E$, $p_m$ is added to $C$ and a new $p_m$ is obtained by incremental ANN searching using MBM.

5. If $k$ number of ANNs in the road-network distance have already been determined and $LBTD > sum_k^N$, then the result set is returned and the process is terminated.

6. The node $r.N_C$ is expanded and the process repeats starting from step (3).

### 4.2.3 Experimental Results for Modified SSMTA* Algorithm to ANN Queries

To evaluate the efficiency of using an ANN query by applying the SSMTA* algorithm, we experimented four methods. All methods followed the IER framework. First, ANN candidates were incrementally generated in the Euclidean distance by using MBM. Then, by applying each of the four methods, the road network distances were verified. The first method is ANN0. It was a method proposed by Yiu et al. [36]. ANN0 verifies the distance between two points by using a conventional pair-wise A* algorithm. The other three methods ANNPQ, ANNQP and ANNPQLB are evaluated in this experiment.
Figs. 4.6 and 4.7 show the results of the processing time and the expanded node number when the number of elements of the query points set $Q$ is set at 3 and that of ANN candidates, $k$, to be searched is set by varying the value from 1 to 20. Figs. 4.8 and 4.9 show the results when $Q$ is set at 7. When we compared the results of the expanded node number, it was clear that ANNPQ performed better than ANN0. In addition, when the processing time is compared, both methods obtained almost similar results; however, ANN0 was slightly faster than ANNPQ because ANN0 adopted a pair-wise A* algorithm and the searching area was small for each pair. Therefore, the hit ratio in the LRU buffer became higher in ANN0.
When the value of $k$ was large, ANNQP performed better than the other methods except ANNQPLB. On the other hand, ANNQPLB is more efficient than the other methods.

Figs. 4.10 and 4.11 show the expanded node number and processing time, respectively, when the number of elements in $Q$ is set at 3 and $k$ is set at 5. Moreover, the POI density ($Prob$) varies from 0.001 to 0.02. When the density of POI is higher, ANN0 and ANNPQ deteriorate in processing time, because several candidate POIs that have similar cost values are generated when the density of the POIs is high.
Therefore, the number of candidate POIs to be verified also increases. On the other hand, in ANNQP and ANNQPLB, even if the number of candidates is increased, they are located nearby, and the search area of each query point does not widen.

### 4.3 Summary

We have described an algorithm for efficiently finding the shortest paths from one starting point to several targets on a road network, and have demonstrated that it outperforms the conventional A* algorithm when applied to a kANN query espe-
cially when the number of target points is large. We also improved the previous SSMTA* algorithm to get better performance in expanded node numbers. The intensive experiments shows that modified SSMTA* algorithm outperformed the existing work in terms of processing time and expanded nodes number.
Simple Trip Planning Query Based on IER Framework

5.1 Introduction

Several types of trip planning query methods have been proposed for serving LBS applications in recent years. In typical trip planning, some point of interest (POI) categories are given to stop at them before arriving at a final destination. Li et al. [81] proposed trip planning query (TPQ) which is not specified the visiting order of POI categories. For example, a restaurant, a department store, and a movie theater are visited before reaching the final destination, however, a visiting order is not specified. Sharifzadeh et al. [80] proposed optimal sequenced route (OSR) in which the visiting order is strictly given. For example, a department store should be visited first, next is to a restaurant, and finally a movie theater before the final destination.

Although these trip planning queries have been studied mainly in the Euclidean distance, they should be optimized the total travel time or the distance on the mobile road network. Some methods have already been proposed to optimize on
the road network, however, they require some materialization or pre-computation which requires a huge memory space aimed for a large road network.

The current study deals the simplest trip planning query, given only one category of POI to be visited during the trip. We proposed an efficient algorithm to search the optimal route without any materialization nor pre-computation. We called this type of trip planning hereafter as the simple trip planning query (STPQ). Though the target problem is simple, the proposed framework and algorithms can directly be applied to more complicated trip planning queries including TPQ and OSR.

Figure 5.1 shows an example of STPQ. $s$ shows the position of the starting point of the trip, usually it is the current position of the user. $d$ shows the final destination of the trip. Small spots on the road map show locations of the POIs belonging to the specified category to be visited. STP searches pre-specified number ($k$, it is 3 in this example) of shortest routes via a POI from $s$ to $d$. The searched route is shown by thick lines. Sometimes, the result of this type of queries needs to contain plural routes to let users choice.

The framework employed in STP query is based on incremental Euclidean restriction (IER), which searches candidates in the Euclidean distance first, and then verifies the results in the road network distance. This approach is versatile and it has been applied to several kinds of queries based in the road network distance, however, there is few attempt for applying to trip planning queries.

In later sections, we discuss efficient algorithms for both steps in IER framework. For Euclidean distance search, an efficient incremental search algorithm referring MBR on the R-tree index is studied. The incremental search means to report the result one by one from the best. This characteristic is essential for IER framework. This is because all paths in Euclidean distance shorter than the shortest path in the road network distance should be searched; however, the road network distance cannot be known before the verification. For the verification on the road network, an algorithm based on the bi-directional searching with the distance constraint is
proposed, then they are adapted efficiently to STP query.

![Figure 5.1: Simple trip planning query](image)

The main contributions in this chapter are as follows;

- To present a novel incremental STP search algorithm in the Euclidean distance. This algorithm searches simple trip paths by a best-first manner on R-tree.

- To present an efficient verification algorithm in the road network distance, which also reduces duplicate node expansions.

- To reveal the IER framework outperforms all existing algorithms for STP queries in road network distance through experiments.

## 5.2 Preliminaries

### 5.2.1 Simple Trip Planning Query

This section presents a simple trip planning query (STPQ), that is the simplest version of trip planning queries. Other trip planning queries, such as TPQ, OSR, and MRPSR described in the previous section, can set plural POI categories to
be visited. Meanwhile, STPQ can be set only one category of POI during a trip. However, if an efficient algorithm is developed targeting on STPQ, it can be directly applicable to more complicated trip planning queries. This is the meaning to discuss STPQ problem.

**STPQ and kSTPQ** When a start ($s$) and a destination ($d$) points of a trip are given, and a set of POI ($P$) to be visited during the trip is given, a simple trip planning query replies the shortest route starting from $s$, stopping at $p$ ($p \in P$), and arriving $d$. $k$STPQ returns the shortest $k$ routes each of which contains a different POI.

### 5.2.2 Basic Concept for Path Finding

The basis of road network exploration is performed by gradually enlarging search area referring an adjacency list of road network nodes. Each record of the adjacency list is composed a node ($n$) on the road network and it describes directly neighboring nodes to $n$. The nodes correspond intersections or dead-end of a road segment on the road network. The adjacency list is divided into small sized blocks on a hard disk, then its record is referred through LRU (least recently used) buffer. Consequently, the number of times to refer the adjacency list becomes the dominant factor of processing time in the shortest path search on the road network.

Usually, the shortest path search process is controlled by a priority queue (PQ). Each record in PQ has a node $n$ and a cost value, the PQ returns the record in order of the cost value. The cost assignment method is different in each search algorithm. For example, Dijkstra’s algorithm use the network distance from the start point to the currently noticed node $n$, meanwhile A* algorithm use the sum of the network distance from the start point to $n$ and the Euclidean distance between $n$ and the destination. For the node $n$ obtained (dequeued) from PQ, directly neighboring nodes are investigated referring to the adjacency list. To find the neighboring nodes of $n$, cost value is calculated, finally all nodes with the cost value
are enqueued into $PQ$. This sequence of steps are called a node expansion. The node expansion contains the step to refer the adjacency list, which causes disk access. By the reason, the number of node expansions is also directly affected the processing time. Consequently, for efficient shortest path search on the road network, to reduce the number of node expansion is essential. The shortest path is obtained when $n$ dequeued from $PQ$ reaches the destination point $d$.

In the above steps, once expanded nodes are put into a closed set (CS) to avoid the same node being expanded multiple times. Every time a node is dequeued from the PQ, the node is checked whether it is already in the CS or not. If the node is in the CS, it is simply ignored. Otherwise, it is expanded. The CS has another important role, it is used to compose the found shortest path shape. The record in the CS has the current node ($n_c$) and the previous node ($n_p$) to $n_c$. After the search reached the destination ($d$), search the record in the CS whose $n_c$ is $d$. Let the record be $r$. Repeat the process of searching $r.n_p$ in the CS until $r.n_p$ meet the starting point $s$.

### 5.2.3 Existing Algorithms for STPQ

As described in related work, Li et al.’s MDQ and Sharifzadeh et al.’s PNE can directly be applied to STPQ. Both algorithms are almost the same when the number of visiting POI category is limited one as like in STPQ.

Let the starting point be $s$, the destination be $d$, and the set of POI to be visited during the trip be $P$. The basic operation of MDQ and PNE is to search $k$-NN (nearest neighbor) of $s$ from $P$, incrementally. Every time when $i$-th nearest neighbor POI ($p_i \in P, i = 1, \ldots, k$) is found, a new shortest path search whose origin is $p_i$ and the destination is $d$ starts. When a searching targeting $d$ reaches to the destination, one shortest trip path (TP) is fixed. This process is repeated until $k$-th TP is found.

These algorithms should use some pre-computation, for example network Voronoi
diagram[40] or some kinds of materialization, to achieve practical processing time. Contrary, when this searching is requested to do without them, Papadias et al.’s INE can be used for the incremental \(k\)-NN query. Meanwhile, for the latter shortest path searching between \(p_i\) and \(d\), along with Dijkstra’s algorithm, \(A^*\) algorithm can be used because the destination of searching is given in this step. In any case, when these algorithms search the route without any precomputation, they consume huge calculation time especially when POI density on the road network is low or large \(k\) value is requested.

Recently, several STPQ algorithms were proposed by Ohsawa et al. [97]. They are the bi-directional search by Dijkstra’s method, named (BA), the bi-directional search by \(A^*\) algorithm (TNSE), and the incremental search using \(A^*\) algorithm (AIA). By experimental evaluations, it is concluded that AIA outperforms best among them.

AIA is based on \(A^*\) algorithm by the bi-directional searching. Two kinds of searches, one starts from \(s\) targeting \(d\), and the other one starts from \(d\) targeting \(s\) are executed concurrently. Both searching are controlled by one PQ, besides each searching has individual CS. It differs from the usual bi-directional \(A^*\) algorithm, the search does not terminated even when wavefront (which means the nodes contained in PQ) of both search meet. In Figure 5.2(a), \(p_1\) and \(p_2\) are POIs, \(a,b,c\) are road network nodes being included in the wavefront, solid lines show the paths on the road network to reach the nodes, dotted closed curves show the wavefront of each origin, and dotted line segments show the Euclidean distance from each node to the individual destination. The search gradually expands the searching area, then the wavefront reaches a POI as shown \(p_1\) in Figure 5.2(a). At this point, a shortest path originating \(d\) to \(p_1\) is found. The searching continues until the wavefront originating other side reaches the same POI. After a while, both directional searching reach an identical POI, then a simple trip path is determined. In Figure 5.2(b), the searching originated \(s\) reaches \(p_1\), then a TP via \(p_1\) is determined. When \(k\)TP is requested, the above searching is repeated until \(k\) POIs are reached by both
directional searching.

In AIA a node on the road network is expanded at most twice, one by searching originated from $s$ and one by searching originated from $d$. AIA is very efficient when POI density is high. On the other hand, when POI density is low, the search area becomes large, consequently a large number of nodes are expanded. AIA is not given any knowledge about POIs location, then it must gradually expand the searching area. Meanwhile IER strategy adopted in Section 5.3 obtains the searching targets by Euclidean space searching.

5.3 STP Query Method Based on IER

Simple trip planning queries based on IER first search a candidate set in Euclidean distance, and then verify the result set in the road network distance. During the process, simple trip path candidates must be generated incrementally. This section proposes efficient methods for both steps.

5.3.1 STP Query in the Euclidean Distance

Sharifzadeh et al.[79] proposed light optimal route discoverer (LORD) algorithm for finding OSR in the Euclidean distance. Moreover, they proposed an improved version R-LOAD using a spatial index R-tree. The R-LOAD bases on a depth first
search; however, it is not suitable for the incremental search which is required in IER. Therefore, in this section, an incremental simple trip path candidates query method using the R-tree index was presented. This algorithm can straightly adopt to more complex trip planning such as OSR and TPQ. Table 5.1 summerize the symbols appear in the rest of the paper.

### Table 5.1: Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>Starting point of a trip</td>
</tr>
<tr>
<td>$d$</td>
<td>Final destination point of a trip</td>
</tr>
<tr>
<td>$d_E(s, d)$</td>
<td>The Euclidean distance between $s$ and $d$</td>
</tr>
<tr>
<td>$d_N(s, d)$</td>
<td>The road network distance between $s$ and $d$</td>
</tr>
<tr>
<td>$P$</td>
<td>The specified POI set one of which is visited during a trip</td>
</tr>
<tr>
<td>$C$</td>
<td>A set of the candidate POIs to be tested on road network distance</td>
</tr>
<tr>
<td>$k$</td>
<td>The number of paths to be founded</td>
</tr>
<tr>
<td>$e$</td>
<td>An MBR or a POI managed in R-tree</td>
</tr>
<tr>
<td>$L^E(e)$</td>
<td>Lower bound Euclidean path length via $e$</td>
</tr>
<tr>
<td>$L^N_{sd}(e)$</td>
<td>Lower bound road network path length via $e$</td>
</tr>
<tr>
<td>$cost$</td>
<td>Cost value compared in a priority queue</td>
</tr>
</tbody>
</table>

Let POI set be indexed by R-tree[4], and a start and a destination points be $s$ and $d$ respectively. The simple trip path in the Euclidean distance can be obtained by traversing on the R-tree choosing nodes which have high possibility to reach the optimal POI. The possibility can be calculated by referring MBRs of the R-tree.

Figure 5.3 shows typical examples of positional relationship between $s$ and three destinations $d_1, d_2, d_3$ and an MBR $(m)$ in the R-tree. All possible arrangement for two points and an MBR can be categorized into three cases. Our purpose is to calculate the possible minimum path length, in other word, the lower bound path length via a POI located in the MBR from $s$ to $d$, it is denoted as LBPL (lower bound path length).

When the straight line segment connecting $s$ and $d$ overlaps the MBR in the case $d_1$, LBPL is $|L_{sd}|$, because a POI in a descendent node of the MBR can be located
on the line segment $L_{sd}$.

$d2$ and $d3$ are the cases when the line segment connecting to $s$ do not overlap with the MBR. In the case $d2$, LBPL is the sum of the length of the line segments via a nearest vertex of the MBR from $s$ to $d$. In the case $d3$, LBPL is the sum of the length of the path via point $A$ on a edge of the MBR from $s$ to $d$. $A$ is the point where the incident angle from $s$ and the reflection angle to $d3$ are identical.

![Figure 5.3: Minimum bounding distance of STP](image)

The following shows the above mentioned lower bound path length calculation method for three cases. Let the line segment whose end points are $s$ and $d$ be $L_{sd}$, the objective MBR to calculate the LBPL be $m$, and the four vertexes of the MBR be $C_1$ to $C_4$.

**Case1:** In the case $L_{sd}$ intersects $m$, the LBPL is the length of $L_{sd}$, i.e. $|L_{sd}|$. This case corresponds $d1$ in Figure 5.3.

**Case2:** In the case $L_{sd}$ intersects both extended lines of horizontal and vertical sides of $m$, LBPL is the minimum length which through a vertex of $m$ (Figure 5.4(a)), i.e. $\min(|L_{sc_i}| + |L_{C_i d}|) : \{i = 1, \ldots, 4\}$.

**Case3:** Otherwise, $L_{sd}$ is located on one side of an edge $(b)$ of $m$ (Figure 5.4(b)). In this case, the point $d'$ which is symmetry with respect to $d$ across an edge of the MBR $b$ is obtained, then the intersection point $A$ on $b$ and $|L_{sd'}|$ are
calculated. When the point A is located in the range of the edge b, the LBPL is \( |Lsd'| = |LsdA| + |LsdB| \). Otherwise, the LBPL is calculated by the same method with Case 2.

![Figure 5.4: Distance calculation in Case 3](image)

Hereafter, the LBPL obtained from the above mentioned method denotes as \( Lsd^E(e) \), where \( e \) is either an MBR in R-tree or a POI. When \( e \) is an MBR, the value of this equation shows LBPL against the MBR. Meanwhile, when \( e \) is a POI, the value shows the trip path length on Euclidean distance via the POI. The R-tree is traversed by the best first search using a priority queue (PQ). Here, the PQ manages the following records.

\[
< Lsd^E(e), e >
\]  

(5.1)

In the above equation, \( e \) is an MBR or a POI. In the PQ, records are ordered ascending by the value of \( Lsd^E(e) \).
Algorithm 4 Euclidean distance simple trip path query (ESTP)

1: \( n \leftarrow 0, C \leftarrow \emptyset \)
2: \( PQ.\text{poll}(<d_E(s,d),\text{root}>) \)
3: \( \textbf{while } PQ.\text{size()} > 0 \text{ and } n < k \text{ do} \)
4: \( r \leftarrow PQ.\text{poll()} \)
5: \( \textbf{if } r.e \text{ instance of MBR then} \)
6: \( \text{for all } m \in e.c \text{ do} \)
7: \( PQ.\text{offer}(<L_{sd}(m),m>) \)
8: \( \text{end for} \)
9: \( \textbf{else} \)
10: \( C \leftarrow C \cup r.e, n \leftarrow n + 1 \)
11: \( \textbf{end if} \)
12: \( \textbf{end while} \)
13: \( \textbf{return } C \)

Algorithm 4 shows pseudo-code of the simple trip path search based on Euclidean distance. This algorithm finds \( k \) shortest trip paths according to ascending order of their length. In line 1, \( n \) is a counter for the number of the found POIs, and \( C \) is the result set composed by found POI, which is initialized by an empty set. Line 2 calculates the Euclidean distance between \( s \) and \( d \), composes the initial record of equation (5.1), then enqueues the record in \( PQ \). From Line 3 to 12 are repeated until \( k \) TPs are found or \( PQ \) becomes empty. Line 4 dequeued a record \( r \) whose cost is minimum from \( PQ \). In line 5 to 8, when \( r \) is an MBR, make records for all child nodes or POIs and then enqueue them into \( PQ \). In line 10, when \( e \) is a POI, which means that a POI is found, the POI\((r.e)\) is added to the result set \( C \) and increases \( n \).

This algorithm can incrementally find TP candidates in the Euclidean distance by invoking the algorithm from Line 3 keeping the contents of \( PQ \).

Lemma 3. The algorithm ESTP adds the simple trip path to the result set in ascending order of the path length. When it is invoked incrementally, it also returns the simple trip path in ascending order.

Proof. When \( e \) is a POI, its LBPL is the minimum among all records in the \( PQ \),
since the PQ dequeues the record in ascending order of LBPL value. Then, the POI is added in the result set in ascending order of LBPL. Otherwise, e is a node. In this instance, there are two cases to consider: one is that the child of e is a POI, and the other one is that it is a node. In the former case, LBPL is calculated as the actual path length via the POI, and then the POI is enqueued. In the latter case, the LBPL is calculated targeting for the MBR of the child node. Since the value of the LBPL is a lower bound for all POIs managed in the descendent nodes, it does not exceed the actual LBPL of the POIs in the descendent nodes. Consequently, ESTP returns POIs in ascending order of LBPL.

5.3.2 Verification on Road Network Distance

Next step is the verification step, which tests the candidates generated by ESTP in the road network distance. In this verification step, pair-wise A* algorithm (PWA*) targeting C can be used, because the starting point (s and d) and the destination (c) are given. The SSMTA* algorithm can also be used for this verification by the same reason. In the following explanation, the POIs are located on the road network nodes for the simple explanation. The modification for the case that the POIs are located on road segments is straightforward. It can achieve by using the same data structure described in [3].

The following is the general approach of the verification.

1. Number of k POIs which gives the least \( L_{sd}^E \) are searched by ESTP algorithm. Let the candidate POI set be \( C (c_i \in C, i = 1, \ldots, k) \).

2. Applying PWA* or SSMTA*, calculate the distance \( L_{sd}^N(c_i) = d_N(s, c_i) + d_N(d, c_i) \), for all \( c_i \in C \).

3. Invoking ESTP incrementally, search the POIs whose \( L_{sd}^E \) are less than the k-th \( L_{sd}^N \) obtained in (2). Set the result into \( C \).

4. Calculate \( L_{sd}^N \) for all \( c_i \) obtained in (3) by the same way in (2).
(5) Report the top $k$ POIs ($c_i$) according to the value of $L_{sd}^N(c_i)$.

In step (3), all POIs whose $L_{sd}^E$ are less than $k$-th $L_{sd}^N$ are searched incrementally to compose a new $C$. This is because the POI which is not added into $C$ by the step (1) can also be a member of the result set when its $L_{sd}^N$ value is lower than that of the $k$-th POI in step (2) result. Besides, the other POIs whose $L_{sd}^E$ value is greater than $k$-th $L_{sd}^N$ can be safely ignored.

Both methods perform more efficiently compared with AIA when the density of POI distribution is low. Contrary, when the POI density is high, the number of candidates to be tested increases. Then, both calculation time and expanded node number become considerably larger than AIA. Another weak point of these methods are that $L_{sd}^N(c_i)$ cannot be decided unless both $d_N(s,c_i)$ and $d_N(d,c_i)$ are determined.

To resolve the above mentioned weak points, we proposed another method based on SSMTA* considering the whole trip path length.

### 5.3.3 Bi-directional Distance Constraint Algorithm

The proposed algorithm determines the trip path length by the bi-directional search from $s$ and $d$ simultaneously. In the rest of this chapter, it is called as bi-directional distance constraint, abbreviated as BDDC.

The basic processing is a bi-directional version of the SSMTA* described in Section ???. At the time of the node expansion, the SSMTA* calculates the cost value of a node $n$ by the following equation.

$$cost = d_N(s,n) + d_E(n,c)$$

(5.2)

Here, $s$ is the starting point, and $c$ is a POI in the candidate set. Besides, the
BDDC calculates the cost using the following equation.

\[ \text{cost} = d_N(s, n) + d_E(n, c) + d_L(c, d) \]  \hspace{1cm} (5.3)

The third term in the right side is the Euclidean distance \(d_L(c, d)\) before the searching path from \(d\) does not reach \(c\), and it is changed to the road network distance \(d_N(c, d)\) after the searching path from \(d\) has reached \(c\). This term works to suppress futile node expansion targeting a POI whose \(d_E(n, c)\) is small but the value \(d_L(c, d)\) is large. In the example of Figure 5.5, \(c_1\) is closer to \(na\) than \(c_2\), hence \(d_E(n, c_1) < d_E(n, c_2)\). However, \(d_E(n, c_1) + d_L(c_1, d) > d_E(n, c_2) + d_L(c_2, d)\). Consequently, using Eq. (5.3) as the cost value conducts to more possible candidates faster than Eq. (5.2).

In Figure 5.5, after the searching path from \(s\) has reached \(n\), \(n\) is being expanded. For node \(na\) which is adjacent to \(n\), these two costs are calculated.

\[ \text{cost}_{c_1} = d_N(s, n) + d_N(n, na) + d_E(na, c_1) + d_L(c_1, d) \]  \hspace{1cm} (5.4)

\[ \text{cost}_{c_2} = d_N(s, n) + d_N(n, na) + d_E(na, c_2) + d_L(c_2, d) \]  \hspace{1cm} (5.5)

Then, if \(\text{cost}_{c_2}\) is the smallest among all candidates in \(C\) as shown in Figure 5.5, \(\text{cost}_{c_2}\) is assigned for \(na\) as the cost value. When a node \(n\) is expanded, the cost value is calculated for all adjacent node of \(n\) in the same way.
Until so far, the case in which the searching origin is \( s \) is described. Since BDDC searches the path to the candidate bi-directionally, \( d \) is another searching origin. For the origin \( d \), the cost value is calculated by the above equations exchanging \( s \) and \( d \) contrariwise.

The records in the PQ have the following format.

\[
< \text{Cost, } D_o, N_c, N_p, N_o > \tag{5.6}
\]

Here, \( N_c \) is the current node, \( N_p \) is the previous node on the path to \( N_c \), \( N_o \) is the origin node of the search (\( s \) or \( d \)), and \( D_o \) is the road network distance from the searching origin to \( N_c \) (i.e. \( d_N(N_o, N_c) \)).

BDDC searches the shortest path to \( c \ (\in C) \) simultaneously from \( s \) and \( d \) controlled by a PQ. Therefore, initially these two records are enqueued in PQ.

\[
< d_E(s, d), 0, s, -, s > \tag{5.7}
\]

\[
< d_E(d, s), 0, d, -, d > \tag{5.8}
\]

Here, ‘-’ shows null value. During the search, the minimum cost record is dequeued from the PQ, then its \( N_c \) is expanded.

When the \( N_c \) of the dequeued record meets a candidate \( c_i \) in \( C \), the road network distance between an origin \( N_o \) to \( c_i \) \( (d_N(N_o, c_i)) \) has been determined. At this point, \( c_i \) is removed from \( C \) where the candidates of searching originated \( N_o \). Besides, \( c_i \) is still remaining as a candidate of the searching from the opposite origin.

When BDDC reaches a candidate \( c_i \), the third term in right-hand side in Eq. (5.3) is changed to the road network distance. This operation makes narrower the search area. Algorithm 5 shows the pseudo-code of the BDDC.
Algorithm 5 BDDC

1: $R \leftarrow \emptyset$
2: $C \leftarrow ESTP(k, s, d)$
3: $c_n \leftarrow NextESTP(s, d)$
4: $nextDist \leftarrow d_E(s, c_n) + d_E(c_n, d)$
5: $PQ.offer(d_E(s, d), 0.0, s, null, s)$
6: $PQ.offer(d_E(s, d), 0.0, e, null, e)$
7: while $PQ.size() > 0$ do
8:   $v \leftarrow PQ.poll()$
9:   if $v$ has a POI then
10:      makePartialPath()
11:      modifyPQ
12:   end if
13:   if $v.cost > nextDist$ then
14:      $C \leftarrow C \cup c_n$
15:      $c_n \leftarrow NextESTP(s, d)$
16:      $nextDist \leftarrow d_E(s, c_n) + d_E(c_n, d)$
17:      modifyPQ
18:   end if
19:   if $v.cost > R.maxPL$ then
20:      break
21:   end if
22:   for all $nn \in neighbor(v, N_c)$ do
23:      decide $c_i$ which gives minimum $h(nn, c_i)$
24:      $d_N \leftarrow d_N(q, v.N_c) + d_N(v.N_c, nn)$
25:      $PQ.offer(<d_N + h(nn, c_i), d_N, v.N_c, v.N_c>)$
26:   end for
27: end while
28: return $R$

This algorithm is omitted the operations for the closed set which keeps once expanded nodes for simplicity of the algorithm. This algorithm uses two closed sets, one for the searching from $s$, and the other for from $d$. Every time a node expanded, the node is registered in the corresponding closed set according to the origin. The partial path from an origin to a POI is obtained by tracking back in the closed set as described in Section 5.2.2.

In line 1, the result set $R$ is initialized by an empty set. $R$ keeps the state of
the found path. Every time when the searching reaches a POI, a partial path from either origin to the POI is created by the procedure \textit{makePartialPath}(), and the result is registered to \( R \). When the searching from another origin arrived the POI, \( R \) has a complete trip path. The record in \( R \) are sorted by the path length. \( R.maxPL \) is the \( k \)-th trip path length in \( R \). While \( k \) paths have not been found yet, the value is set to \(+\infty\).

Line 2 obtains \( k \)-STP candidates by using ESTP algorithm. Line 3 finds the next STP candidate \((c_n)\) incrementally. \( nextDist \) in line 4 is the \( L_{sd}(c_n) \). When the \textit{cost} obtained from the PQ exceeds this value, \( c_n \) is added to \( C \) as a candidate. Line 5 and 6 compose initial records starting from \( s \) and \( d \), respectively, then put them into the PQ.

The loop from line 7 to line 27 is repeated while the PQ is not empty. In line 8, a record \( v \) with minimum \textit{cost} is dequeued from the PQ. When \( v \) has a POI on it, a POI has been found. At this time, the partial path from the starting point to \( v \) is retrieved by referring the closed set (line 10). The result of partial path is registered in \( R \). This algorithm needs recalculation of the \textit{cost} value in \( PQ \), every time when a POI is found. This operation recalculates the \textit{cost} value of the records in \( PQ \) except for the found POI from \( C \) (line 11).

When the \textit{cost} value of \( v \) exceeds \( nextDist \), \( c_n \) is joined into \( C \) (line 14), then \( c_n \) is updated by the searching next Euclidean STP (line 15) and alter \( nextDist \) by the new \( c_n \) (line 16). As same with line 11, the \textit{cost} values of all records in \( PQ \) need to be recalculated by the new \( C \) joining \( c_n \) (line 17).

In line 19, \( R.maxPL \) is the \( k \)-th trip path length in the result set \( R \). If the \textit{cost} value of \( v \) exceeds it, no shorter path cannot be found by continuation of the searching, then return the result set and terminate the searching.

Line 22 to 26 are node expansion process, which fetches all connected nodes to \( v.N_C \) (current noticed node) by referring adjacency list. \textit{cost} is calculated for all connected nodes, and the records are created for them, then they are enqueued into
the PQ. Here, $h(nn, c_i)$ in line 25 is $\min\{d_E(nn, c_i) + E(c_i, N_0)\}, c_i \in C$.

## 5.4 Performance Evaluations

Using a real road network (the number of road segments is 25,586) and generated POI data, we conducted intensive experiments over MDQ, MDQ-A*, AIA, PWA*, SSMTA*, and BDDC.

![Expanded Node Number](image1)

**Figure 5.6**: Expanded Node Number($k = 1$)

![Processing Time](image2)

**Figure 5.7**: Processing Time($k = 1$)

Figure 5.6 and 5.7 show the expanded node number and the processing time.
respectively when \( k = 1 \), that is the case to find only one STP which given the shortest route. MDQ and MDQ-A* are omitted in these figures, because they took more than 100 times of the calculation time and the expanded node number of the BDDC.

The processing time and the expanded node number in PWA* increase according to POI density. This is because the number of candidates generated by ESTP based on Euclidean distance increases according to the POI density, then the targets of the verification also increase. The same node is expanded repeatedly several times especially in the high POI density. Besides, the low increasing ratio in calculation time in comparing with the expanded node number is caused by the following reason. The PWA* refers to the adjacency list over relatively small size of region during to find the shortest path between two points, then the hit ratio of the LRU buffer of the adjacency list keeps high. Then the hard disk access times is reduced.

The processing time of AIA increases rapidly around the low POI density. This is because the search area of AIA is enlarged according to the decrease of the POI density. Conversely, BDDC shows good performance in the expanded node number and the processing time independent from the POI density.

Figure 5.8 and 5.9 compare the expanded node number and the calculation time,
respectively, when $k = 5$. Generally, both measurements value increase according to the increase of the number of $k$ value. Among them, the increase of the calculation time in AIA is remarkable. This is because that AIA needs to search a wide area on the road network in according to the increase of $k$ value, it causes the low hit ratio of the LUR buffer.

5.5 Summary

We proposed a fast framework for STP query on road network distance. It bases the incremental Euclidean restriction, in which the candidates are generated by Euclidean distance search, and then verifies them in the road network distance. For the first stage, we proposed an incremental algorithm suitable for IER framework. Besides, for the second stage, we proposed a simultaneous search method named BDDC which is based on the SSMTA* algorithm.

With various experimental results, we examined BDDC substantially outperforms previous methods in terms of both the expanded node number and the processing time. The IER framework discussed in this chapter is also applicable to more complicated trip planning queries such as TPQ and OSR. The detail explanation will be explored in the next chapter(6).
CHAPTER 6

Optimal Sequenced Route Query Based on INE and IER Framework

6.1 Optimal Sequenced Route (OSR) Query

In this chapter, we presented efficient algorithms for an OSR query on road networks. In an usual trip planning, a final destination is normally provided. For example, a home or an office can be a final destination of a trip. In this regard, in our trip planning method, the starting (usually the current position) and the destination positions of the trip are provided explicitly. When the destination is specified explicitly, we can adopt an efficient A* algorithm and the bidirectional search [98] for an OSR query.

The OSR query was first proposed by Sharifzadeh et al.[80]. They proposed several algorithms to find the \( k \)-OSR in both vector (based on the Euclidean distance) and metric (based on the road-network distance) spaces. Among them, the progressive neighbor expansion (PNE) is the only algorithm that can be applied to road networks.
On a road network, the nearest neighbor (NN) object calculated by the Euclidean distance is not always the NN calculated on a road network [3]. The computation cost can drastically differ between these two distance measurements. The Euclidean distance between two points can be easily calculated; however, for the distance on a road network, we need to find the shortest path that connects two points. To find the shortest path, Dijkstra’s algorithm [1] and the A* algorithm [2] are usually used. However, these algorithms consume a large amount of CPU power compared to the Euclidean distance-based search. In addition, a Euclidean distance-based algorithm can use the simple spatial index structures, such as, R-tree [4], to narrow the search space. In fact, the R-LORD algorithm proposed by Sharifzadeh et al. [80] employed R-tree for this purpose. Moreover, a spatial index based on the Euclidean distance is not effective for road network distance-based queries [3].

Therefore, in order to reduce the number of node expansions, duplicated node expansions are inhibited and a visited POI graph (VPG) is used to control duplicated routes. An efficient unidirectional search algorithm for a $k$-OSR query using the VPG is also presented. In addition, to achieve a stable search time, a bidirectional search algorithm to start the search from the current and destination points is described. The proposed method performs 100 times faster than the PNE algorithm [79] by conducting extensive experiments.

### 6.2 OSR Query Applying A* Algorithm

When we plan a trip, the starting position is obvious. In general, the current position acquired by GPS can be taken as the starting position, or the user specifies the starting position explicitly. In most situations, the final destination is also decided. Then, a trip planning query can be invoked with the starting ($S$) and the final destination ($E$) positions. In this case, we can adopt an A* algorithm for the efficient search of the TPQ. We can also use bidirectional search [97] for this purpose. Our algorithm proposed in this section uses both these methodologies, because
they can reduce the calculation cost, which is mainly due to the considerable node expansion on the road network. We first describe an OSR query using the single-source A* algorithm (unidirectional search). We then develop it into a bidirectional search.

### 6.2.1 OSR Query by A* Algorithm

Let $U_i$ be a category of the POI to be visited, and $M$ be a sequence of $U_i$ to specify the visiting order. That is, $M = \{U_1, U_2, \ldots, U_m\}$, and here $m$ is the length of $M$ ($m = |M|$). Our OSR query finds $k$ optimal sequenced routes from the starting point $S$ to the destination $E$, visiting each POI belonging to $U_i (1 \leq i \leq m)$ one after another, according to the given sequence $M$. The partial sequenced route (PSR) is the shortest route from the starting point $S$ to one of the POIs in $U_i$, by passing through the POIs one after another choosing from $U_j (1 \leq j < i)$ on the way according to the given sequence. SR is the total routes from $S$ to $E$, visiting the POIs according to the given sequence $M$. To simplify the algorithm explanation, we assume that the POI is on a road-network node. However, this restriction can be easily resolved [3].

The A* algorithm has been applied to find the shortest route given $S$ and $E$ [2]. Here, we apply it to the $k$-OSR query. The A* algorithm evaluates the favorable node $n$ to be expanded next by the cost $C = d(S, n) + h(n, E)$. Here, $d(x, y)$ is the distance from node $x$ to $y$ moving on the road network, and $h(y, z)$ is a heuristic function between $y$ and $z$. Because we evaluate the cost of the OSR by the route length, we use the Euclidean distance between $y$ and $z$ as the value of $h(y, z)$.

Fig. 6.1 shows an example of a search using the A* algorithm. In this example, the search starts from $S$, and then finds $P_1^1$ belonging to $U_1$. From this POI, a new search targeted at the $U_2$’s POI starts. In parallel, the search starts from $S$, finds $P_2^1$ belonging to $U_1$, and then another new search starts from the POI.

As mentioned above, the A* algorithm decides the next-expanded node on the
road network \((n_a)\) by an extracted record from the priority queue (PQ), which gives the minimum cost \(d(S, n_a) + h(n_a, E)\). For each road segment connected to \(n_a\), the cost is calculated to compose a new record, as shown in Eq. (6.1), and it is then inserted back into the PQ:

\[
C = d(S, n_a) + d(n_a, n_b) + h(n_b, E)
\]

Here, \(n_b\) shows the opposite-side node to \(n_a\) of the road segment.

In the record, \(C\) is the abovementioned cost, \(U_i\) is the next POI category to be visited, and \(L\) is the distance on the road network from \(S\) to \(n_b\) (i.e., \(L = d(S, n_b)\)). \(P_{prev}\) is the last-visited POI that belongs to \(U_i\). \(n_a\) is necessary in the record to restore the PSR by backtracking from \(n_b\) to \(S\). For the backtracking, Eq.(6.1) is recorded in a hash table indexed by \(n_b\), after it is removed from the PQ. The term \(org\) is the origin of the PSR, i.e., \(S\) or \(E\). This term is not always necessary for a unidirectional search where the origin of the search is predetermined; that is, a unidirectional search can start from either \(S\) or \(E\). Both \(S\) and \(E\) can be the origins of a bidirectional search. Hence, both search origins are required for the bidirectional search.

Repeating the node extraction from the PQ and the node expansion, the search
area is gradually enlarged. The search is terminated when the item \( n_b \) of the record extracted from the PQ is \( E \). This terminating condition is the same for the typical A* algorithm.

Every time a POI \( P^i \) belonging to \( U_i \) is found, we start a new search targeting \( U_{i+1} \), and simultaneously, we need to continue the search for another POI that belongs to the same \( U_i \), ignoring \( P^i \). Fig. 6.2 explains this necessity. This figure shows a situation where the POIs that belong to several categories of POI are arranged in a line. The search starting from \( S \) first finds \( P_a \) in \( U_1 \). Then, the next search targeting a POI in \( U_2 \) starts, and the search finds \( P_b \). Next, a search targeting \( U_3 \) starts from \( P_b \), and then finds \( P_f \). Finally, the search reaches \( E \) and is then terminated. By this search, the sequence \( S \rightarrow P_a \rightarrow P_b \rightarrow P_f \rightarrow E \) has been found. However, there are other OSRs that have the same length. For example, the other sequences \( S \rightarrow P_a \rightarrow P_c \rightarrow P_f \rightarrow E \), and \( S \rightarrow P_d \rightarrow P_e \rightarrow P_f \rightarrow E \) have the same length. If we invoke another search ignoring \( P_a \) and continue to search the same category, then the search can subsequently find \( S \), and thus the sequence \( S \rightarrow P_d \rightarrow P_e \rightarrow P_f \rightarrow E \) can be found.

![Figure 6.2: OSR query setting border category](image)

Visiting order \( \square (U1) \Rightarrow \star (U2) \Rightarrow \triangle (U3) \)

In a typical shortest-path search, Dijkstra’s algorithm and the A* algorithm use a close set (CS) to avoid multiple node expansions. Once a node is expanded, it is registered to the CS, and the node will not be expanded again. On the other hand, an OSR query requires multiple CSes. Each CS records an expanded node from an individual source of searching (e.g., \( S \) or \( P_f^i \)). This characteristic of an OSR query causes multiple node expansions; that is, a node on a road network is expanded several times. For example, the search paths targeting \( U_1 \) started from \( S \) to find
the POIs $P_1^1$ and $P_1^2$, belonging to $U_1$, then new searches targeting $U_2$ start from both of them. These two searches are executed independently. Therefore, a node that has been expanded by another search can be expanded again, which causes a rapid increase in processing time. This also happens with the PNE when it adopts an incremental $k$-NN on a road network. We will deal with this problem using the bidirectional search in Section 6.3.

### 6.2.2 Bidirectional Search

The unidirectional search described above can be extended to a bidirectional search in a straightforward manner. A bidirectional search starts from $S$ and $E$ simultaneously under the control of one PQ. The record of Eq. (6.1) is put into the same PQ, which is independent of the origin of search. The search starting from $S$ tries to find the POI according to the predetermined POI sequence $M$, and the search starting from $E$ tries to find the POI in the reverse order of $M$; that is, $E \rightarrow U_m \rightarrow U_{m-1} \rightarrow \ldots \rightarrow U_1 \rightarrow S$.

The search is terminated when the search paths from both origins meet at a POI. Every time a search encounters the next POI to be visited, the arrival to the POI from another origin is checked. Suppose the POI is $P_c$ belonging to $U_C$, and both PSRs from $S$ to $P_c$ and from $E$ to $P_c$ have been found; we can then obtain a complete SR by combining these two PSRs at $P_c$. When we need up to the $k$-th-shortest OSRs, we can obtain them by repeating node expansions until the $k$-th-shortest OSR is found. The abovementioned bidirectional search appears simple when only one shortest OSR is requested. However, when multiple $k$-OSRs are requested, there are some problems to be considered. In Section 6.3, we explain the problems and propose a solution.

When one of the categories in $M$ has POIs with a very dense distribution, several independent searches will start from each POI belonging to the category. This causes an enormous number of node expansions, because the node expansions take
place independent of each other such that the computation takes a long time. To
avoid this effect, Fujii et al. [99] proposed a method to set a midway category (MC),
named \textit{bidirectional search with midway category} (BSWMC). The MC is selected
from the POI category that has the highest density. When a search reaches a POI
in the MC, no new search targeted at the next category starts. At this time, a PSR
from $S$ or $E$ to a POI belonging to the MC is found. There is no additional search
to find the next category starting from a POI in MC. Meanwhile, node expansions
from another origin are advanced until one of them reaches the POI from the other
side. At this time, a complete OSR is found.

The BSWMC method is suitable when we know the density of the POI in each
category. In general, however, we cannot know the POI density. Even if we can
conjecture the density, the POI is apt to be distributed with bias. Therefore, we
need to improve the efficiency without setting the MC.

6.3 Suppressing Duplicated Node Expansion

Both the abovementioned approaches start a new node expansion from a POI be-
longing to $U_i$ toward a POI belonging to $U_{i+1}$, every time a POI belonging to $U_i$
is found. In Fig. 6.3, a search for a POI belonging to $U_1$ starts from $S$, and then
the search finds $P^1_1$ and $P^1_2$ in that order. New searches for a POI belonging to
$U_2$ start from $P^1_1$ and $P^1_2$. Then, the search that started from $P^1_1$ finds $P^2_1$ as
the second-visited POI, at which a further new search starts for a POI belonging to $U_3$.
Later, the search starting from $P^1_2$ reaches $P^2_2$, and then another new search for a
POI belonging to $U_3$ starts. However, these two searches will consequently find
the same path (PSR) from $P^2_2$ to $E$, as is shown in Property 1. Therefore, we need to
suppress this redundant node expansion. A similar duplication also happened in
the PNE when it adopted the incremental network expansion (INE).

Property 1. $k$-PSRs starting from a POI belonging to $U_i$ to $E$, obeying a pre-
specified visiting sequence are determined uniquely.
Proof. We deal with a time-invariant road network, and the constellation of the POIs is fixed. If $k$-PSR from a POI position belonging to $U_i$ to $E$ will not be changed at query time, then they are determined uniquely. Therefore, this supports the property.

Despite Property 1, a unidirectional search always starts a new search when a new POI is found. This causes duplicated node expansions, which find the same PSR. By reducing these duplicated node expansions, an efficient $k$-OSR query can be conducted.

Figure 6.3: Search path arrival to the same POI from multiple search paths

Property 2. Consider searching an SR, the search started from $S$, finding POIs in order, then reaches a POI ($P^i_j$) belonging to $U_i$. Let the PSR from $S$ to $P^i_j$ be $R$, and the length be $L_R$. To advance this search, a new search targeted at the next category $U_{i+1}$ starts from $P^i_j$. Consider another search path $R'$ whose length is $L_{R'}$ has reached $P^i_j$. Then, we have the following relation between $L_R$ and $L_{R'}$:

$$L_R \leq L_{R'}$$

Proof. The priority queue (PQ) returns a node to be expanded according to the expected path length in ascending order. When $R$ is returned from the PQ prior to $R'$, the following relation stands:

$$L_R + h(P^i_j, E) \leq L_{R'} + h(P^i_j, E)$$

(6.2)

The heuristic distance value $h(P^i_j, E)$ is the same in both the left and right terms, so we obtain the following relation:
\[ L_R \leq L_{R'} \] (6.3)

Consequently, the equality in Eq.(6.3) is true when \( R \) and \( R' \) have the same length.

From these two properties, we can suppress duplicated node expansions. When \( P_3^3 \) is the nearest POI belonging to \( U_3 \) from \( P_1^2 \), it is determined uniquely independent when the paths reached \( P_1^2 \) from \( S \). Then, when the search already started from \( P_1^2 \), which is the head of the PSR \( S \to P_2^1 \to P_1^2 \), to start a further node expansion from the same POI that is the head of another path is not useful.

In Fig. 6.3, we explained that a late-arrival PSR to \( P_1^2 \) (e.g., \( S \to P_1^1 \to P_1^2 \)) cannot be the shortest path among SRs through \( P_1^2 \), because the first-arrival PSR to \( P_1^2 \) (e.g., \( S \to P_2^1 \to P_1^2 \)) is shorter than the other late-arrival PSR. Then, when we need to find only one shortest OSR, we do not need to consider the late-arrival PSR at any POI. However, when we need to find \( k \)-OSR \((k > 1)\), late arrival PSRs could be a part of the SRs. In this case, we need to consider the late-arrival PSR. Simultaneously, we need to consider suppressing the duplicated node expansions, which gives the same result.

To cope with this problem, we use the visited POI graph (VPG) as shown in Fig. 6.4. In the graph, nodes are visiting POIs (and two terminal points \( S \) and \( E \)) and edges are paths connecting neighboring POIs. An edge in VPG can be shared by plural OSR routes. For example, OSR routes \( R_1 \) and \( R_2 \) are sharing the link \( P_2^1 P_3^3 \) and \( P_3^3 E \). Therefore, if we calculate for a link once, we can avoid the calculation for the same link again. This reduces the calculation cost of road network distance considerably, especially, when \( m \) is large.

Hereafter, we refer to the unidirectional search with the VPG as the USVPG, and the bidirectional search with the VPG as the BSVPG. Algorithm 6 shows the BSVPG algorithm. The USVPG algorithm is almost similar to the BSVPG
algorithm, although it is simpler. Line 1 initializes $PQ$ with $S$ and $E$. $R$ is the result set returning $k$-OSR. The size of $R$ is bounded by $k$, i.e., the number of requested SRs. Then $R.i$, the number of found SRs in $R$, does not exceed $k$. $R.L_{max}$ is the maximum SR length found so far; it has the value $\infty$ while $R.i$ is less than $k$. When $R.i$ reaches $k$, $R.L_{max}$ shows the $R.i$-th shortest SR length. Line 2 initializes the VPG by inserting the two vertices $S$ and $E$.

In Line 4, the entry that has the smallest cost is removed from the $PQ$. When the entry is the next POI to be visited, the $VPG$ is checked as to whether the POI has already registered in it (Line 8). Then, a record ($e.P_{prev} \rightarrow e.nb$) is composed and inserted into the VPG (Line 8). Next, a check is performed as to whether a PSR from another origin has already reached the POI (Line 8). In this case, generated SR is called to make all SRs passing through the POI $e.nb$. The resulting SRs are registered in $R$, and $R.i$ and $R.L_{max}$ are altered by the results (Line 12).

Lines 15 to 18 are always executed when $e.nb$ is the next visiting POI and the PSR from the opposite side has not reached $e.nb$. The record ($e.P_{prev} \rightarrow e.nb$) is inserted into the VPG, and then the POI that belongs to the next visiting POI category is searched (Line 17).

The steps below Line 21 are always executed because of the reason described in Section 6.2.1. Then, when $e$ is a POI, the search advances in two ways, one is to
Algorithm 6 Bi-directional Search with VPG

1: \( PQ \leftarrow \{S, E\}, R \leftarrow \emptyset, R.L_{\text{max}} \leftarrow \infty, R.i \leftarrow 0 \)
2: Initialize \( VPG \).
3: \textbf{loop}
4: \( e \leftarrow \text{deleteMin}(PQ) \)
5: \textbf{if} \( e.C > R.L_{\text{max}} \) \textbf{then}
6: \textbf{return} \( R \)
7: \textbf{end if}
8: \textbf{if} \( e.nb \) is in the next visiting POI category \textbf{then}
9: \textbf{if} \( e.nb \) is an element of \( VPG \) \textbf{then}
10: \( \text{Add the link } (e.P_{\text{prev}} \rightarrow e.nb) \text{ to } VPG. \)
11: \textbf{if} PSRs from the opposite origin have reached \( e.nb \). \textbf{then}
12: \( R \leftarrow \text{generateSR}(e.nb, VPG, R) \)
13: \textbf{end if}
14: \textbf{else}
15: \( \text{Add the link } (e.P_{\text{prev}} \rightarrow e.nb) \text{ to } VPG. \)
16: \textbf{for all} road network node neighboring \( e \) \textbf{do}
17: \text{compose Eq.(6.1) for } U_{\text{next}}, \text{ enqueue it into } PQ. \)
18: \textbf{end for}
19: \textbf{end if}
20: \textbf{end if}
21: \textbf{for all} road network node neighboring \( e.nb \) \textbf{do}
22: \text{compose Eq.(6.1) for } U_i, \text{ enqueue it into } PQ. \)
23: \textbf{end for}
24: \textbf{end loop}
25: \textbf{return} \( R \)

search the POI that belongs the next visiting POI category (Line 17), and the other
is to search the POI that belongs to the same POI category (Line 22). When \( e \) is
not a POI, usual node expansion is continued. This SR search is repeated until the
number of found SRs exceeds the requested number \( k \), and the cost of \( e \) becomes
greater than \( R.L_{\text{max}} \) (Line 23). After the latter condition has been satisfied, no
shorter SR than \( k \)-th shortest SR in \( R \) will be created.

### 6.4 Experimental Results

We generated several sets of POIs in a pseudo-random sequence, with varying dis-
tribution density \((p)\), which means the existence probability of a POI on a road
Algorithm 7 generateSR($e, VPG, R$)

1: Trace $VPG$ from $e$ to the origin composing all PSR, then insert the result into $PSR_1$.
2: Trace $VPG$ from $e$ to the reverse origin composing all PSR, then insert the result into $PSR_2$.
3: Make all SR passing through $e$ by making direct product between $PSR_1$ and $PSR_2$, then add them into $R$.
4: Renew $R\.L_{max}$ and $R\.i$ by the current condition.
5: return $R$

segment (a road segment means a polyline between two intersections, or between an intersection and a dead end). For example, there are approximately 25 POIs in the subject area in the case of $p = 10^{-3}$, and approximately 256 POIs when $p = 10^{-2}$.

Fig. 6.5 shows the results between the unidirectional search (USVPG) and the bidirectional search (BSVPG). (a) compares their processing time among the unidirectional searches that start from $S$ (USVPG-S), $E$ (USVPG-E), and BSVPG. The density of the POI increases from the first POI (0.001) to the last POI (0.01). In these cases, the USVPG-S is the fastest, the USVPG-E is the slowest, and the BSVPG is moderate. When the density distributions of POIs are not equal, the unidirectional search starting from the less-dense side always outperforms the other.

![Figure 6.5: Comparison between USVPG and BSVPG](image-url)
Fig. 6.5(b) shows the result of another pattern. In this case, the middle category has the highest density, and the third category has the second-highest density. The BSVPG outperforms the others in this case. The experiments using the other combinations show the same tendency. Consequently, when we estimate the density of each visiting POI category, we can choose the fastest strategy by starting the search from the less-dense side. In general, however, the estimation is not easy. Even if we can know the statistical density of each POI category, the density may vary depending on its location. Figures 6.6(a) to (f) compare the performance between BSVPG and PNE according to the processing time and the number of expanded nodes. The experiments were conducted under three POI categories in which $p = 0.001$, $p = 0.002$, and $p = 0.01$, with shuffling of the order to be visited. The probability of the sequence is given in the caption of each figure. The left and right columns compare the processing time and the number of the expanded nodes, respectively. As shown in these figures, the BSVPG outperforms the PNE approximately 100 times in both evaluation criteria; (g) and (h) compare the results of all combinational patterns. In these figures, the pattern is shown by the three-digit number that corresponds to the three different POI densities: 1 corresponds to 0.001, 2 to 0.002, and 3 to 0.01. (g) shows $k = 1$ and (h) shows $k = 10$. As indicated by these results, the BSVPG always outperforms the PNE, and the calculation time becomes stable with increasing $k$, independent of the POI density patterns.

6.5 Overview on USVPG and BSVPG algorithms

Two fast algorithms called USVPG and BSVPG have been proposed to search the $k$-OSR for the road-network distance. Both algorithms are controlled by the A* algorithm. The BSVPG searches POIs from $S$ and $E$ simultaneously. In this study, we also proposed the VPG to reduce multiple node expansions, which is unavoidable in trip planning. This fact holds for all the existing trip planning methods that work on road-network distance measurements, for example, the OSR, TPQ, and MRPSR. Therefore, a strategy to reduce them can be the key to the fast trip planning. The
Figure 6.6: Comparison between PNE and BSVPG
VPG can be applied not only to the OSR, but also for several other types of trip planning that use road-network distance measurements.

Experimental results confirm that the presented algorithm can search the OSR approximately 100 times faster than the PNE. The USVPG and BSVPG largely contribute to the improvement in the VPG. The USVPG search, starting from the less-dense POI side, can find the OSR faster than the BSVPG, while the USVPG search starting from the other side will degrade. Therefore, in the case when the POI density cannot be known in advance, the BSVPG can be a good selection.

We also discussed an efficient method based on INE, however, the incremental Euclidean restriction strategy [3] can also be applied for OSR search based on road network distance. On this strategy, two kinds of efficient algorithms are essential; one is an incremental OSR candidates generation method in Euclidean distance, and the other one is the efficient road network distance verification method for those candidates. The detail is describe in the following section.

6.6 Incremental Queries in the Euclidean Distance

6.6.1 OSR Queries in the IER Framework

In this section, we proposed fast trip planning query method in road network distance. In advance of the query, the current position, the final destination, and some number of POI categories visiting during the trip are specified. Then the query searches the shortest route from the current position, visiting one from each specified POI categories before reaching the final destination. Several such kinds of trip planning methods have been proposed. We discussed optimal sequenced query (OSR) which is the simplest because of the strongest restriction on the visiting order. The basic strategy adopted in this research is incremental Euclidean restriction (IER), in which generating candidate route by Euclidean space search and verifying by road network distance. Firstly, we proposed a fast incremental algorithm to find
OSR candidates on Euclidean space. Secondly, efficient verification method in road network distance is proposed.

The OSR query can be defined as follow:

**Definition 1 (OSR query).** Given a current point $s$, a final trip destination $d$, and a visiting order of POI category sets $C_i(1 \leq i \leq m)$, the OSR query finds the minimum distance route starting from $s$, selecting one POI from each $C_i$ according to the visiting sequence, and finally arriving at $d$.

The IER framework generates candidates for OSRs in the Euclidean space, and then verifies those candidates in the road network distance. Let the shortest OSR given by searches in Euclidean space be $Sr$ and its verified length in the road network be $L_N(Sr)$. The shortest OSR in the Euclidean space is not always the shortest OSR in the road network distance. Therefore, all OSRs whose length are less than $L_N(Sr)$ also have the potential to be the shortest route in the road network. Therefore, all OSRs less than $L_N(Sr)$ must be searched in the Euclidean space, and then the results must be verified in the road network. Finally, the shortest OSR in the road network is returned as the result. These are the essential steps of an OSR query based on the IER framework.

When two points $a$ and $b$ are given, $d_E(a, b)$ denotes the Euclidean distance between $a$ and $b$, and $d_N(a, b)$ denotes the road network distance between $a$ and $b$. IER depends on the relationship $d_E(a, b) \leq d_N(a, b)$. Therefore, if an OSR with the length $L_N(Sr)$ is obtained, the OSR candidates in Euclidean distance longer than $L_N(Sr)$ can be safely discarded.

All OSRs whose lengths are less than $L_N(Sr)$ can be determined by an incremental search. In an incremental search, OSR candidates are searched from the shortest up to $k$ OSRs. Therefore, all OSRs shorter than $L_N(Sr)$ can be determined by repeating the incremental search while the length of the determined OSR is shorter than $L_N(Sr)$. As seen above, incremental searching is important in the IER framework, however, few existing algorithm can search OSR incrementally. Sharifzadeh
et al.[79] proposed the light optimal route discoverer (LOAD) algorithm for finding OSR in the Euclidean distance. Moreover, they proposed an improved version R-LOAD using a spatial index R-tree. Since the R-LOAD bases on a depth first search, it is not suitable for the incremental search which is required in IER. One of the exceptions among existing methods is progressive neighbor exploration (PNE) proposed by Sharifzadeh et al. in [79]. Though, MDQ proposed by Li et al.[81] is also incremental algorithm, it is almost the same searching method with PNE when it is adapted to OSR queries.

PNE searches an OSR by gradually expanding search area whose center is the starting point (s). Expanding the searching area, PNE searches a POI from the set of POIs $C_i (1 \leq i \leq m)$ according to the specified order. Hereafter, the visiting POI is determined by the order of the suffix of the categories, as $C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_m$. The searching is controlled by a priority queue (PQ), which consists of the following records.

$$< \text{Cost}, i, p, \text{PSR} >$$

Here, $PSR$ (partial sequenced route) is the found POI sequences starting from s. For example, when a partial route starting from s, visiting $p^1$ in $C_1$ category and $p^2$ in $C_2$ category has already been searched, the $PSR$ is $\{s, p^1, p^2\}$. $i$ in Eq.(6.4) is the category number targeted to the POI search in the previous search, $p$ is the found POI belongs to the category $C_i$. $Cost$ is the total length of the $PSR$ in the above example, $Cost = d_E(s, p^1) + d_E(p^1, p^2)$. The PQ always returns the record having the minimum $Cost$ value.

The following shows the processing flow of the modified version of PNE. Though the final destination is not given in the original PNE, it is given in the following algorithm.

(1) Search the nearest neighbor of s (NN(s)) from $C_1$ category. Let NN(s) be $p$. 
Make the following record, and enqueue it into the PQ.

\[ < d_E(s, p), 1, p, \{s, p\} > \]

(2) Repeat from step (3) while the PQ is not empty.

(3) Dequeue the Cost minimum record \( r \) from the PQ, if \( r_i > m \), it means the searching have reached \( d \), hence returns \( r.PSR \) as the result.

(4-1) Search NN of \( r.p \) in \( C_{i+1} \), make new PSR appended to the result, and calculate the cost of the PSR. Compose a new record, and enqueue it into PQ.

(4-2) Search NN of the last POI in \( r.PSR \), replace the last element in \( r.PSR \) by the searched result. Calculate the length of the PSR, compose a new record, and enqueue it into the PQ.

The searching area of PNE is gradually extended like a concentric circle centered at \( s \), and it is terminated when \( p \) of the dequeued record meets \( d \). Incremental searching can be achieved by invoking PNE from step (2), keeping the contents of the PQ.

Since this algorithm was proposed to find OSR in metric space, it is directly applicable to Euclidean space. However, when the density of the POI is high or the number of categories to be visited during the trip is large, it requires an enormous amount of calculation. This is because it executes incremental NN queries in step (4-1) and (4-2), and the NN query refers to the R-trees managing POIs.

**6.6.2 Simple Trip Path Query in the Euclidean Distance**

Before describing general OSR queries in which multiple POI categories are specified to be visited, to simplify the fast search, ESTP algorithm in the chapter 5 can be applied. Generally, the simple trip path (STP) query finds the shortest route from a starting point \( (s) \) to a destination \( (d) \) via only one POI belonging a specified
category. Generally, the number of the POIs belonging to the specified category is large, therefore, we assume the POIs are indexed by an R-tree [4]. The basic strategy to find a simple trip path is a best first search calculating the lower bound route length (LBRL) to the MBRs in the R-tree. Fig. 6.7 illustrates the process to find trip route on the R-tree. (a) shows an R-tree, (b) and (c) shows the arrangement of the MBRs (rectangles) and the POIs (black dots). The dashed square in (b) and (c) show the MBR of the root node. The dotted lines illustrate trip routes, and the numbers accompanying show the length of the trip routes.

At the beginning, LBRL is calculated for each MBR in the root node, the record of Eq.(5.1) is composed, then the record is enqueued into the PQ. At this point, the contents of the PQ is the following.

\[< 25, m2 >, < 42, m3 >, < 45, m1 >\]

By dequeuing, \(< 25, m2 >\) is obtained from PQ, hence the child node of \(m2\) is descended one level and reaches the leaf node where POIs, \(C, D,\) and \(E\) are contained. The LBRL is calculated for each POI, and the corresponding records are enqueued. At this point, the PQ contains the following records.

\[< 32, C >, < 38, D >, < 41, E >, < 42, m3 >, < 45, m1 >\]

Dequeuing the PQ again, we get the record \(< 32, C >\), and this time \(e\) of the record is a POI. Herewith the shortest trip path via \(C\) has been gotten. If we continue the search until we get \(k\) shortest trip path, we can get them according to the ascending order of the length.
6.6.3 Application to Multiple POI Categories

OSR queries can be achieved by applying the Euclidean distance simple trip path (ESTP) query algorithm (in the chapter 5) repeatedly and changing the objective POI category. Assume that \( m \) types of POI \( (C_i : 1 \leq i \leq m) \) are visited sequentially during the trip from \( s \) to \( d \). First, a simple trip route visiting a POI in category \( C_1 \) is searched by applying ESTP. We assume that \( p^1c^1 \) is obtained as the result as shown in Fig. 6.8. Next, a POI in category \( C_2 \), which gives the minimum distance during the trip from \( p^1c^1 \) to \( d \), is searched by applying the ESTP again. Repeating this search, we can obtain a route by visiting a number of \( m \) POIs sequentially during the trip from \( s \) to \( d \).

The entire search is controlled by a PQ. The records in the PQ are ordered by the distance of the route from \( s \) to \( d \) by visiting already determined POIs and an MBR, which is searched next. For example, in Fig. 6.8, the cost value is
$d_E(s, p^1 c^1) + L^p_{c} e, d(m)$. The PQ contains records whose categories of targets are different. The PQ record has the following format.

$$<\text{Cost}, \text{prev}, \text{dfs}, \text{tgt}, e, \text{PSR}>$$  \hspace{1cm} (6.5)

Here, \text{prev} is the POI that belongs to the category preceding the current target \text{tgt} category, and its initial value is \text{s}. Furthermore, \text{dfs} is the partial route length from \text{s} to \text{prev}. \text{tgt} is the target POI category number next to be searched, \text{e} is a node in the R-tree managing the POIs in the category \text{C}_{tgt}, and \text{PSR} is a sequenced POI set determined up to this point. The PQ returns records in the ascending order of the Cost value. For example, in Fig. 6.8, \text{prev} is $p^1 c^1$, \text{dfs} is $d_E(s, p^1 c^1)$, \text{tgt} is 2, \text{e} is \text{mbr}, and \text{PSR} is $\{s, p^1\}$.

Let the record dequeued from the PQ be $r$. When \text{e} of $r$ ($r.e$) is an MBR, new records are composed for all child nodes of $r.e$, and then the records are enqueued into the PQ. Otherwise, when $r.e$ is a POI, it is the POI to be visited next. Therefore, the POI category is advanced by one, and then the next target category is changed to $C_{e, tgt+1}$. When the category $C_{e, tgt+1}$ is the final destination \text{d}, a complete route is found, and it is the shortest OSR. Therefore, the result route is
Htoo Htoo

Algorithm 8 Euclidean Optimal Sequenced Route (EOSR)

Require: $s, d, m, T(i : i \leq i \leq m)$
Ensure: Euclidean OSR

1. $PQ.enqueue(<d_E(s, d), s, 0, 1, T(1).root, \{s\}>)$
2. while $PQ.size() > 0$ do
3. \hspace{1em} $r \leftarrow PQ.dequeue()$
4. \hspace{1em} if $r.tgt > m$ then
5. \hspace{2em} return $r.PSR$
6. \hspace{1em} end if
7. \hspace{1em} if $r.e$ instance of POI then
8. \hspace{2em} $i \leftarrow r.i + 1$
9. \hspace{2em} $d \leftarrow r.dfs + d_E(r.prev, r.e)$
10. \hspace{2em} $PQ.enqueue(<d + d_E(r.e, d), r.e, d, i, T(i).root, r.PSR \cup r.e>)$
11. \hspace{1em} else
12. \hspace{2em} for all $ch \in r.e.c$ do
13. \hspace{3em} $PQ.enqueue(<r.dfs + L_E^{r.prev,d}(ch.e), r.prev, r.dfs, r.tgt, ch, r.PSR>)$
14. \hspace{2em} end for
15. \hspace{1em} end if
16. end while

This algorithm can generate OSRs incrementally from the shortest to the next shortest if the function retains the contents of the PQ after the shortest OSR is found. The verification on road network distance requires all OSR candidates whose route lengths are less than $L_{min}$. This search can be achieved by iterating the algorithm while the route length is less than $L_{min}$. Algorithm 8 shows the pseudocode of the OSR search in the Euclidean distance.

The verification on road network distance can be achieved several ways including pair-wise A* algorithm and several materializing methods of shortest path distance on road network. In this approach, the following verification steps based on IER framework are applied.

Step 1 Search the shortest OSR by EOSR.

Step 2 Calculate the road network distance. Let the distance be $L_{min}$. 
**Step 3** Search the next shortest OSR incrementally. If the length \( L_{\text{next}} \) exceeds \( L_{\text{min}} \), an OSR has been found. The process is terminated.

**Step 4** Calculate road network distance of the next candidate. If it is less than \( L_{\text{min}} \), update \( L_{\text{min}} \) by the distance.

**Step 5** Goto Step 3.

This algorithm can be adopted to \( k \)-OSR easily. In this approach, visited POI graph (VPG) (detail in chapter 5) is used to avoid duplicated node expansions which cause long processing time in query searching.

## 6.7 Experimental Result

In this experiment, the heap size was prepared as 1GB to manage big POI data. The distribution of the POI density depends on the distance between the starting point \( s \) and the destination \( d \) because many POIs probably exist on long distance even if the density of POI is spare. Conversely, even if the distance between \( s \) and \( d \) is shorter, when the POI distribution is dense, many POIs may exist on the distance. POI data between 10,000 and 1,000 is generated by pseudo-random sequence. As in Fig. 6.7, space size is set as 100,000. For instance, when \( R=2500 \), the average POI within the circle is about 20.

We evaluated proposed method comparing with the existing work PNE [79]. Fig. 6.10 compares the referred R-tree node numbers of PNE, and the EOSR in OSR queries in the Euclidean distance. In the experiments, the number of visiting POI categories \( m \) is set at 3. The horizontal axis shows POI density and the vertical axis shows the number of referred nodes in R-trees. The size of the R-tree nodes was set to 64 slots (size of a node was 2KB).

In Fig.6.10, PNE-1st and EOSR-1st show the number of visited R-tree nodes when the first (the shortest) result was obtained. PNE-10th and EOSR10-th show
the number of visited R-tree nodes when the tenth shortest result was obtained. As shown in this figure, the referred node number in PNE increases rapidly according to the POI density. In contrast, the increase is lower in the EOSR. For example, the ratio of the visited R-tree node number between two methods reaches 100 times when the POI density is 0.02.

Fig. 6.11 shows the relationship between the referred R-tree node number and the number of the POI categories to be visited \((m)\) during the trip. In this experiment, the POI density was set to 0.01 for all POI categories. The number of nodes increases in accordance with the increase in \(m\) in PNE. The ratio of PNE and the EOSR reaches more than 300 times when \(m = 5\).

Fig. 6.12 compares the processing time between the proposed method EOSR and the existing work PNE. It is measured for the 50th shortest OSR route and 100th shortest OSR route incrementally. The visiting category is set to 3 and R is set to 500 because PNE can only search for the small number of visiting category\((m)\) when the value of R is larger. However, EOSR can search the shortest route without depending on the number of category and the distance between \(s\) and \(d\).

Fig. 6.13 measured the processing time in varying the slot size of R-tree, The
Query Algorithms for LBS in Road Network Distance

Figure 6.10: POI density and visiting node number

Figure 6.11: Relationship between $m$ and visiting node number
Figure 6.12: Relation between $m$ and processing time

Figure 6.13: Relation between slot number in R-tree and processing time
maximum slot size is set to 64 and then, it is varied to 32, 16, and 8. When the number of visiting category is greater than four, 16 has the fastest processing time than other, however, the difference between each other is about four times at most.

Fig. 6.14 shows the comparisons between EOSR and PNE in terms of expanded nodes number and processing time when the number of visiting category varies. The slot size of R-tree is set to 64 (2KB per node) and R=2500. In this result, when \( m \) is greater than five, heap overflow occurred in PNE search, and therefore, results less than 5 are only shown for PNE. EOSR outperformed PNE in the processing time by twice of the number of visiting category. In terms of expanded node number,
EOSR has efficiency 50 to 100 times than PNE. In results described above, the number of POI in each category is set to 10000.

In the experiments so far, the number of POIs in a category was a constant number, fixed 10,000. Next, we conducted an experiment to evaluate the effect of the density of POI distribution by varying the number of POIs (m) for each category. We generated two types of POI sets; one contains 10,000 POIs (denoted ‘H’ set) and the other one contains 1,000 POIs (denoted ‘L’ set) respectively. And then, we measured the processing time for OSR sequences that contained an ‘L’ category in 5 POI categories; the rest 4 categories were ‘H’. We denote this as ‘L among H’ (LAH) pattern. We also prepared reverse patterns; one ‘H’ category was contained among ‘L’ categories; (HAL) pattern. The processing time is shown in Fig. 6.15. The horizontal axis shows the position of the different category (for example when m=1, the position of ‘L’ is at 1 in LAH pattern). As shown in this figure, the processing time is decreased when ‘H’ category is placed in later position in the pattern. On the other hand, in ‘LAH’ pattern, the processing time increased rapidly when ‘L’ pattern is placed later position.

The deficit that the processing time depends on POI density can be resolved by changing the searching order to sparser POI searched first. DFLAH and DFHAL in Fig. 6.15 the result by this searching method started from sparser POIs. As shown in this figure, the processing time becomes independent of the POI density in the sequence and be stabilized over all patterns.

Fig. 6.16 and Fig. 6.17 show the results in road network distance measurement. Fig. 6.16 shows the processing result when m = 3 used Road-1, Saitama City map as road network. POI density D is defined the probability of existing a POI on a road segment. For example, when a POI exist on 100 road segments, D is 0.01. This figure shows the result when D = 0.01. The horizontal axis shows the length of the OSR.

Fig. 6.17 shows the number of OSR routes generated and tested by EOSR nec-
necessary to determine the shortest OSR. The length of OSR has close relation with the distance between $s$ and $d$. When it is large, the number of EOSR routes to be tested increased rapidly.

![Graph](image1)

Figure 6.16: OSR query using Maps $m = 3, D = 0.01$

![Graph](image2)

Figure 6.17: Number of generated EOSR to determine OSR in road network

### 6.8 Summary

We discussed an efficient trip planning method for the road network distance based on IER framework as well as INE. In IER framework, firstly, an incremental search algorithm, the EOSR, for the Euclidean distance is presented. Compared with PNE, which is the only existing incremental algorithm applicable to the OSR in the Euclidean distance, experimental results demonstrate that the EOSR query significantly outperforms PNE, particularly when POIs are densely distributed or the number of POI categories to be visited during the trip is large. We also proposed an algorithm to determine only one shortest route; however, the top $k$ shortest routes are sometimes required to facilitate users’ choices. The algorithm proposed in this study can be easily adopted for this requirement because the EOSR generates candidates incrementally and the algorithm for verifying the road network distance can be easily applied to $k$ OSR queries.
CHAPTER 7

Conclusion

We conclude our study by focusing the following three main issues

1. We proposed an efficient shortest paths algorithm to find the shortest paths to multi-target points simultaneously, named SSMTA* algorithm.

2. We applied SSMTA* algorithm to several spatial queries (e.g. $k$NN, ANN) based on IER strategy.

3. We proposed a novel algorithm for trip planning queries on road network distance based on IER strategy and applied it to OSR query.

For (1) we proposed the SSMTA* algorithm, which searches shortest paths between a query point and each point in a given destination point set. In our study, we showed the stability and efficiency of the algorithm by applying to $k$-NN queries based on the IER strategy. Through the performance evaluation of the proposed method in comparison with the INE using Dijkstra’s algorithm, the pairwise A* algorithm and the LBC, we realized that the proposed method outperforms the existing works in processing time and in expanded node number. We also revealed
that the processing times of the pairwise A* algorithm and the LBC increase rapidly when the density of POIs is low or the number of \( k \) is large. The defects occur for the following reasons. On the pairwise A* algorithm, nodes can be expanded several times when \( k \) is large, which increases the hard-disk access times. On the LBC, although the number of node expansions remains low, the cost of PQ scanning increases in proportion to \( k \) and the number of node expansions. This performance deterioration is serious when the density of the POI is low.

Although, like the proposed method, Dijkstra’s algorithm exhibits stable performance, the expanded node number and calculation time of Dijkstra’s algorithm remains twice that of the proposed method. A disadvantage of Dijkstra’s algorithm is that the performance deteriorates substantially when the distribution of the POIs trend toward one side. This biased distribution is apt to appear in the ANN and spatial skyline queries. Then, the ratio of the expanded node number between the INE and SSMTA* becomes larger for these queries.

The calculation cost of Dijkstra’s algorithm is \( O(|E| + |V| \log |V|) \) on the road network \( G = (V, E) \). The calculation cost of the A* algorithm varies depending on the heuristic function. However, the cost of searching the shortest path between two specified points is on the same order as Dijkstra’s algorithm. The worst case occurs when the heuristic function always returns 0. The expanded node number of the LBC and SSMTA* are the same. This means that the worst time complexity is the same for all of the algorithms considered in this paper. From the viewpoint of database systems, however, the number of disk accesses dominates the total calculation time. Accordingly, the proposed method is at least twice as efficient as the other methods. Regarding (2), we also improved the previous SSMTA* algorithm to get better performance in expanded node numbers almost similar result in LBC. Then, modified SSMTA* algorithm is applied to the ANN query. The intensive experiments shows that modified SSMTA* algorithm outperformed the existing work in terms of processing time and expanded nodes number. We proposed a fast framework for STP query on road network distance based on IER framework. For
the first stage, we proposed an incremental algorithm suitable for IER framework. Besides, for the second stage, we proposed a simultaneous search method named BDDC which is based on the SSMTA* algorithm. With various experimental results, we examined BDDC substantially outperforms previous methods in terms of both the expanded node number and the processing time. The IER framework that are followed in the overall study is also applicable to more complicated trip planning queries such as TPQ and OSR. Hence, we advanced the study to adapted with the complicated queries as issue (3).

For (3), we can classify two topic based on incremental network expansion and incremental Euclidean restriction. While reviewing on INE, we studied two fast algorithms called USVPG and BSVPG to search the $k$-OSR for the road-network distance. Both algorithms are controlled by the A* algorithm. The BSVPG searches POIs from $S$ and $E$ simultaneously. In this study, we also proposed the VPG to reduce multiple node expansions, which is unavoidable in trip planning. This fact holds for all the existing trip planning methods that work on road-network distance measurements, for example, the OSR, TPQ, and MRPSR. Therefore, a strategy to reduce them can be the key to the fast trip planning. The VPG can be applied not only to the OSR, but also for several other types of trip planning that use road-network distance measurements.

Experimental results confirm that the presented algorithm can search the OSR approximately 100 times faster than the PNE. The USVPG and BSVPG largely contribute to the improvement in the VPG. The USVPG search, starting from the less-dense POI side, can find the OSR faster than the BSVPG, while the USVPG search starting from the other side will degrade. Therefore, in the case when the POI density cannot be known in advance, the BSVPG can be a good selection.

For OSR query based on IER framework, two kinds of efficient algorithms are essential; one is an incremental OSR candidates generation method in Euclidean distance, and the other one is the efficient road network distance verification method for
those candidates. We realized by evaluations, an incremental search algorithm, the EOSR, for the Euclidean distance is simply powerful, compared with PNE, which is the only existing incremental algorithm applicable to the OSR in the Euclidean distance, experimental results demonstrate that the EOSR query significantly outperforms PNE, particularly when POIs are densely distributed or the number of POI categories to be visited during the trip is large. Moreover, EOSR generates candidates incrementally and the algorithm for verifying the road network distance can be easily applied to $k$ OSR queries. For the future research trend, if materialization approach can be advanced and worked well in road network distance verification, processing time for queries in spatial database can be upgraded. This will be our further study.
Acknowledgement

Firstly, I would like to express my deepest gratitude to my supervisor Prof. Yutaka Ohsawa. It is not enough to express in words how thankful I am to my supervisor who has taught me so many things for my study and research. His wisdom, valuable guidance and support have made my dream come true. I will never forget his care, aids and encouragement to me whenever I had any difficulty.

I would like to express my grateful thank to Prof. Hitoshi Maekawa whose enthusiastic attention to all presentations and my dissertation in particular and discussions for my research and my future career. I would also like to especially thank my co-supervisor associate Prof. Toshinori Yamada whose diligent comments and suggestions have helped in improving my study. I would like to especially thank Prof. Tetsuya Shimamura and Prof. Takashi Horiyama as member examiners for my defense. I am grateful for their kind attention and thorough discussions. I deeply thank all participants in my defense as well.

I would like to sincerely thank Prof. Masao Sakauchi and Prof. Noboru Sonehara from National Institute of Informatics for continuous help, support, and outlook for research in general during the course of my Ph.D.

There are so many people whose friendship I have ever enjoyed in my student life. In particular, I would like to appreciate an advice from my friend, Hnin Witt Yee, who gave me motivation to be a Ph.D student. She has always been beside me during peaks and valleys for which I am very pleasurable. I would like to thank
seniors, members and friends from Ohsawa Lab who help me improve my Japanese language skill, understand Japanese culture and custom and discuss my research over several Lab meetings.

This study is supported by Rotary Yoneyama scholarship and JASSO scholarship. I would like to express my sincere thank to all Rotarians, especially for all members of my host family; Omiya City Rotary Club and JASSO. Without their priceless aids, I could not have completed my study stably.

I would like to express my deepest thank to my beloved mother and my family members who always love me and trust me and make me strong. I am very grateful to have you all.

At last, I would like to heartily thank my late father who encouraged me with pleasure to study this Ph.D course, and loved me until his last breath.
References


tional symposium on advances in geographic information systems (ACM-GIS), p 1. (2001)


[86] Shaw, K., Ioup, E., Sample, J., Abdelguerfi, M., Tabone, O.: Efficient approximation of spatial network queries using the M-tree with road network...


Author’s Publications


[7] Yutaka Ohsawa, Htoo Htoo, Noboru Sonehara, Masao Sakauchi, Sequenced Route Query in Road Network Distance Based on Incremental Euclidean Restriction, DEXA 2012, LNCS 7446, pp.484-491 (2012.9 Vienna)


[12] 藤野和久, Htoo Htoo, 大沢 裕, 曽根原登: Thin クライアントモデルにおける良く通るルートを用いたモニタリング方式, 第19回地理情報システム学会学術研究発表大会, 3D2, 2010.10(京都)

[13] Htoo Htoo, Yutaka Ohsawa, Noboru Sonehara: Optimal Sequenced Route Query Algorithm Using Visited POI Graph, 第19回地理情報システム学会学術研究発表大会, 1C1, 2010.10(京都)


[15] 橋本知宣, Htoo Htoo, 大沢 裕, 曽根原登: ネットワーク Voronoi 図のマテリアライズ化に関する考察, 第20回地理情報システム学会学術発表大会, F-3-1, 2011.10 (鹿児島）

[16] 西歯耕治, Htoo Htoo, 大沢 裕, 曽根原登: 道路網上におけるC-OSR探索法, 第20回地理情報システム学会学術発表大会, D-5-2, 2011.10（鹿児島）


[20] Sithu Aung, Htoo Htoo, Yutaka Ohsawa: ANN Query Method Based on Road Network Distance, 電子情報通信学会総合全国大会, D-4-5, 2012.3 (岡山)
