

Christoph Niemz
ETH Zurich
D-INFK
chniemz@student.ethz.ch
6th Semester BSc
07-925-670

Query Processing in Spatial Network Databases

written by D. Papadias, J. Zhang, N. Mamoulis and Y. Tao

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

Algorithms for Database Systems 2010
Supervisor: Prof. P. Widmayer

Contents

1. Definitions
2. Introduction
3. Background
 - 3.1 R-Tree
 - 3.2 Adjacency list
4. Architecture for SNDB
 - 4.1 Storing the entities
 - 4.2 Storing the network
 - 4.3 Primitive operations
5. kNN-Algorithms
 - 5.1 Incremental Euclidian Restriction (IER)
 - 5.2 Incremental Network Expansion (INE)
6. Performance
 - 6.1 IER vs. INE
7. Conclusion
8. References

1. Definitions

In this section, the definitions which we will need throughout the report will be given.

DEFINITION 1. A network is an undirected weighted graph $G = (V; E; W)$ where V is the set of vertices (i.e., nodes), E is the set of segments (i.e. Edges). Each segment has its own polyline representation. $W: E \rightarrow \mathbb{R}^+$ associates each edge to a positive real number, which is called a segment's length. The length W of a segment is the Euclidian length of its polyline representation.

In the following, the term „point“ will be used both for a node and an entity.

In Figure 1(a), for instance, an example network is shown. The nodes of the network are depicted by the orange circles. The entities are represented as blue squares. The lines represent segments and for every segment, the number near to it is its length. Entity a lies on segment (n_1, n_2) which has length 2.

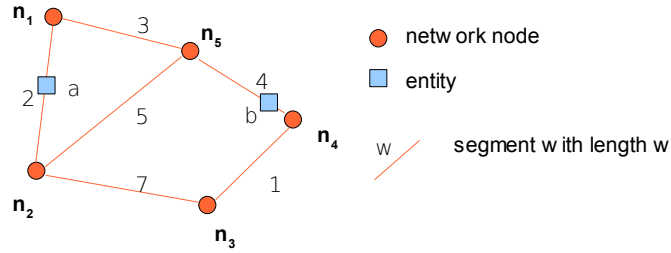


Figure 1(a): An example network.

DEFINITION 2. The polyline representation P_s with $|P_s| =: n$, $n \in \mathbb{N}, n \geq 2$, of a segment $s = (n_i, n_j)$ is an ordered set of two-dimensional coordinates. P_s defines a unique polyline with ending points n_i and n_j , which consists of $n-1$ lines that are connected by the coordinates in P_s in order of their appearance in P_s . The length of a polyline is the sum of the Euclidian lengths of all its lines.

Figure 1(b) shows the segment $n_2 n_3$ from the example graph and its polyline representation.

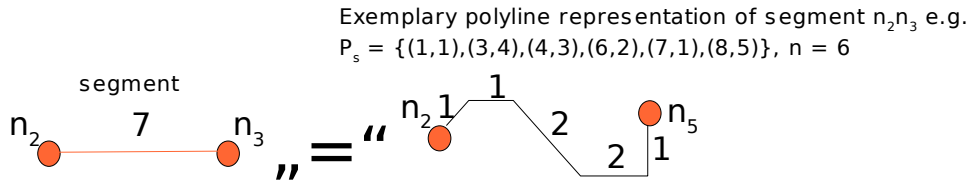


Figure 1(b): Exemplary polyline representation of segment $n_2 n_3$.

One number on each line of the polyline corresponds to the line's Euclidian length. E.g. the first line (starting from n_2) has length 1. The length of segment $n_2 n_3$ is the sum of the lengths of the lines, i.e. $1+1+2+2+1=7$.

DEFINITION 3. The Euclidian distance $d_E(n_i, n_j)$ of two distinct points n_i, n_j is $d_E(n_i, n_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$ with (x_k, y_k) being the x- and y-coordinates of node n_k in the two-dimensional Euclidian space.

DEFINITION 4. The residual distance $d_R(p, n)$ of an entity p to a node n is

- (i) if p is on an edge that goes out from n : the Euclidian length of the polyline representation with ending points p and n .
- (ii) else: ∞

DEFINITION 5. The network distance $d_N(n_i, n_j)$ of two distinct nodes n_i, n_j is the length of the shortest path from n_i to n_j .

DEFINITION 6. The network distance $d_N(p, q)$ of two distinct entities p, q , where p lies on edge (n_a, n_b) and q lies on edge (n_c, n_d) is the distance of the shortest path from p to q . The shortest path from p to q is $\min_{x \in \{a, b\}, y \in \{c, d\}} (d_R(p, n_x) + d_N(n_x, n_y) + d_R(n_y, q))$.

COROLLARY 1. From Definitions 3, 4, 5 and 6 it follows that $d_E(n_i, n_j) \leq d_N(n_i, n_j)$ for two distinct points n_i, n_j , i.e. the Euclidian distance lower bounds the network distance. We refer to this as the *Euclidian lower-bound property*.

DEFINITION 7. Given a source point q and an entity dataset S , a k nearest neighbor (kNN) query retrieves the $k \geq 1$ objects of S closest to q according to the network distance.

2. Introduction

By the time this paper [1] was written (2003), most of the work spent on spatial databases did focus on Euclidian information when processing spatial queries.

In the following, we will however discuss query processing techniques that also take into account the information of a pre-defined underlying spatial network.

Firstly, a motivating example will be shown which illustrates the semantic difference of a spatial query depending on which distance measure (Euclidian or network distance) is used. In Figure 2, a , b and c denote entities and q is a query point. The numbers in the figure correspond to the network distances, e.g. node b has network distance 10km from node q . For the query point q , whose movement is constrained by a network, we want to know the nearest entity i.e. we want to answer a Nearest-Neighbor(NN)-query for the point q . Now, depending on which measure we use, either d will be the nearest neighbor of q (using the Euclidian distance) or b will be the nearest neighbor of q (using the network distance). Since q 's movement is constrained to the network, which means that q has to follow a path on the network when moving and might not just move freely to point d , which actually is the entity with highest network distance to q , entity b would be the appropriate answer to this NN-query.

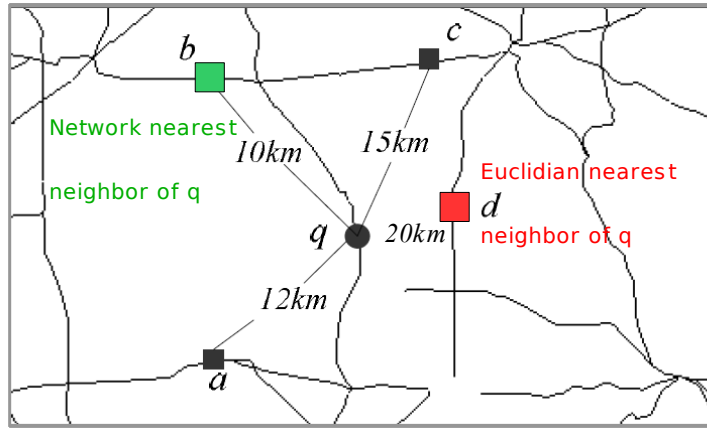


Figure 2: Euclidian vs. Network distance.

Similar motivating examples may also be found for all the other most popular spatial queries, namely Range queries, Closest Pair and e-Distance join.

The purpose of a Spatial Network Database (SNDB) is to do the processing of these popular spatial queries, while taking into account the properties of the underlying network.

The logical structure of the report will be the following. Our goal is that for a given network and query point q , we can answer a kNN-query. To achieve this goal, we will build up an architecture step-by-step that allows us to do network kNN (and other) queries.

At first, a short review of the data structures we need for the architecture will be given. We will review one well-known data structure that stores location information, the R-Tree, and we will also give a short review on a well-understood data structure for storing network

connectivity information, the adjacency list.

By combining and slightly modifying these two well-known data structures, an architecture that allows us to process spatial network queries is constructed.

Based on this architecture, we will then develop two algorithms for the most popular spatial query, namely the k-Nearest-Neighbors-query (kNN), and compare these two algorithms, which will be called Incremental Euclidian Restriction (IER) and Incremental Network Expansion (INE). As the focus of this report is to present the architecture and explain two NN-algorithms, we will not provide algorithms for range queries, closest-pairs and e-distance-Joins, as the authors of the original paper did. For each of these 3 queries, they propose 2 algorithms, which are described in a detailed way in [1] in sections 5,6 and 7.

In the Conclusion, we finally come to an outlook on future research in the area of SNDB and provide some criticism and discuss possible improvements of the techniques presented.

3. Background

In this section, a very quick review on two data structures which we will need for the architecture of SNDB is given:

The R-Tree, which can be seen as a multidimensional extension of the B-Tree, is a dynamic balanced tree that provides a multidimensional index (for this purpose, a 2D-index should suffice) and allows to efficiently compute range and NN-queries in the Euclidian space.

The adjacency list is a data structure that is used to represent a graph that models a network.

3.1 R-Tree

An R-Tree provides a multidimensional index to store point and region data in the Euclidian space. The R-Tree tries to cluster together each set of points that is close in space into a node which is represented by the Minimum Bounding Rectangle (MBR) of the points it contains. In case we have more points to store, nodes are then recursively grouped together following the same principle that is valid for the leaf nodes until we reach the top level, which consists of a single root.

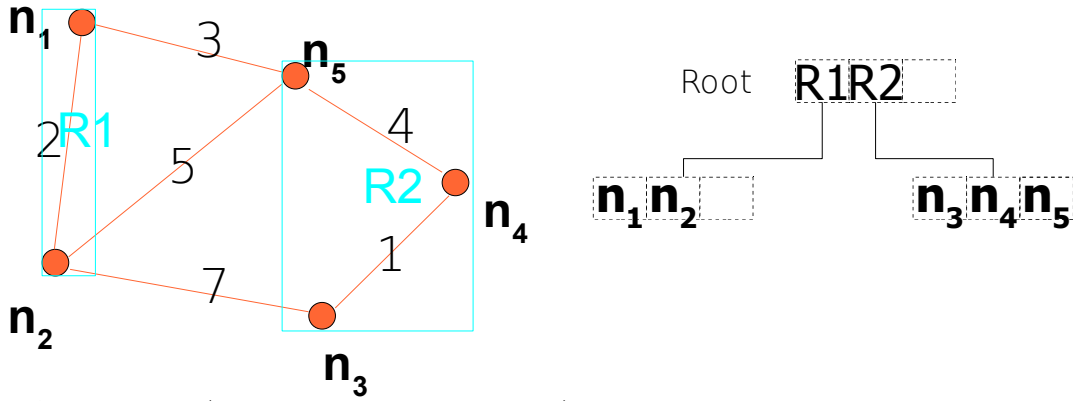


Figure 3. A set of points (the nodes of the example graph) and an exemplary R-Tree representation of it.

Like in the B-Tree, the actual information is stored in the leaf nodes and the nodes on the levels above provide only indexing information.

Updates and deletes are handled dynamically by the R-Tree which together with its property to efficiently compute range queries makes it a state-of-the-art data structure for spatial databases such as Geo Information Systems.

In Figure 3, it is shown how the nodes of the example graph might be represented in an R-Tree. Nodes n_1 and n_2 are grouped into rectangle R1 and nodes n_3 , n_4 and n_5 are grouped into rectangle R2. A more detailed explanation of the R-Tree can be found in [3].

3.2 Adjacency list

Under the assumption that there already exists a digitization process that creates a modeling graph of the real-life network we want to represent, we are now interested in a data structure that allows us to store the network connectivity in a space-efficient way so that we are able to do network expansion and Shortest-Path-computations.

As the adjacency matrix of real world networks usually is very sparse and thus wastes a lot of space that is unused, we will choose the adjacency list to represent the network.

Figure 4 shows the example graph and its adjacency list representation. For each node, there is an adjacency list. The adjacency list of a node n contains an entry for each segment that goes out from n . In each entry, the pointer to the adjacency list of the adjacent node and additionally the length of the segment is stored. The last field of an entry is a pointer to the next element in a node n 's adjacency list. E.g. the first entry $[n_1, 3,]$ of the adjacency list of n_5 corresponds to the segment n_5n_1 that has length 3. The first field is a pointer to the adjacency list of n_1 , the second field is the segment's length and the third field is a pointer to the next entry $[n_2, 5,]$ in n_5 's adjacency list.

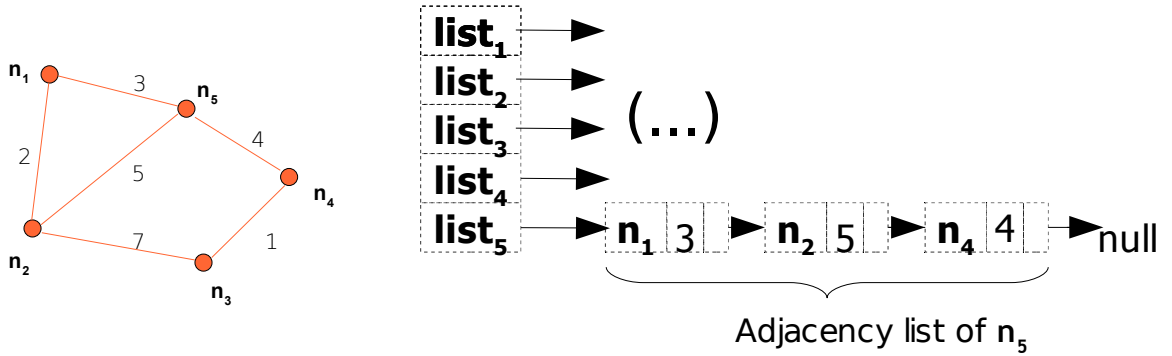


Figure 4. Example graph and its adjacency list.

As we would also like to have a proximity-preserving representation of a graph on disk, the Connectivity-Clustered-Access-Method (CCAM) might be a good approach for our problem. How CCAM works will not be explained here, it is described in detail in [4].

4. An Architecture for SNDB

In the following, we will construct an architecture that allows us to get Euclidian and network information from the network. We first describe how the entities are stored and then explain the storage of the network.

4.1 Storing the entities

Our first design decision is to separate the entities (e.g. hotels, gas stations) completely from the network. To be concrete, this means that we will store each entity dataset in its own R-Tree. This has a lot of advantages:

- (i) Euclidian queries independent of the network can be processed efficiently on each of the datasets.
- (ii) We can combine queries that use both Euclidian and network information. This will be useful for the Euclidian Restriction framework presented later.
- (iii) Each dataset can be updated separately and specific optimizations on each dataset can be done independently of the others.
- (iv) It is easier to add/delete/update entity datasets.

However, this decision also has some disadvantages to the alternative of storing the entities and the network in the same data structure (e.g. R-Tree):

- (i) Two separate data structures have to be entered when processing queries that need both entity and network information (which is true for most queries).
- (ii) The add/delete/update operations for several data structures have to be processed and managed separately.

As the advantages of our approach outbalance its disadvantages, we will stick to our approach.

4.2 Storing the network

As we need to store both Euclidian and network information of the network in order to process spatial network queries, our architecture must be able to store both.

The proposed architecture is composed of three main parts: The adjacency component, the polyline component and the network R-Tree. Each of these three parts will be explained in the following.

The adjacency component stores the adjacency list l_n of each node n and for each adjacent node m (i.e. for each entry in the adjacency list of n) also stores the distance to m and the MBR of m and n . For each entry it also stores a pointer to the page of the adjacent node m and a pointer to the page of the finer grained polyline representation of the segment.

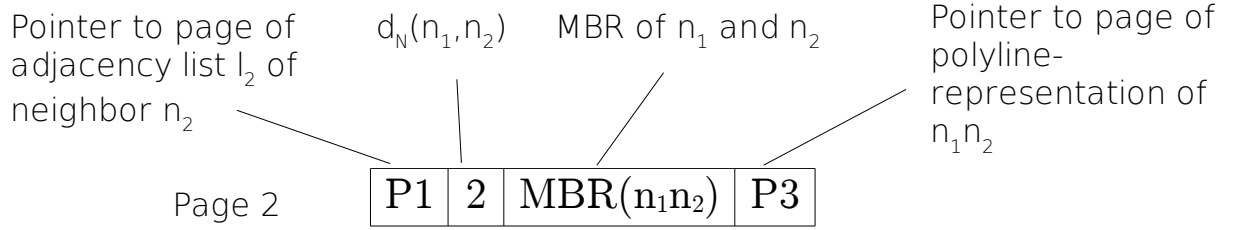


Figure 5. One entry of the adjacency list of n_1 representing the edge n_1n_2

In Figure 5, we can see one entry of the adjacency list of node n_1 . This entry represents the segment n_1n_2 . The first entry P1 is a reference to the page of the adjacency list of the adjacent node n_2 , the second entry „2“ represents the length of the segment n_1n_2 , the third entry MBR(n_1n_2) is the Minimum Bounding Rectangle (MBR) of the nodes n_1 and n_2 and the last entry is a pointer to the page of the polyline representation of the segment n_1n_2 , which is stored in the polyline component.

Having an adjacency component like this, that provides us with information about connectivity (i.e. which nodes are connected) and individual path length for each segment it is now possible to do network expansion and shortest-path-computations on the network.

The polyline component stores the finer grained polyline representation of each network segment and additionally for each polyline representation stores the pages that contain the start- and end-node of the polyline. The polyline component acts as a connector between the connectivity (i.e. graph) information of the network and the location (i.e. the network R-Tree) information of the network, while also storing the polyline for each segment.

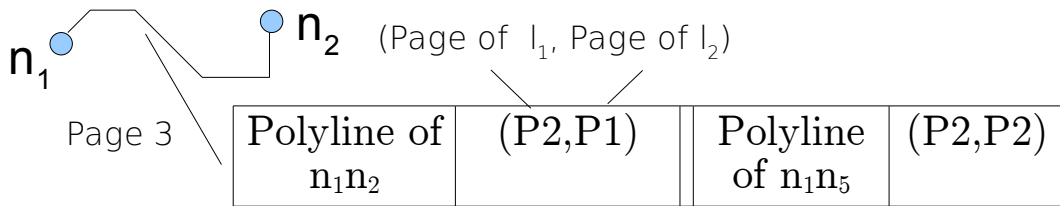


Figure 6. Two exemplary polyline component entries.

In Figure 6, two entries of the polyline component that are stored in page 3 are shown. In the first entry, which stores the polyline of segment n_1n_2 , the first field stores $P_{n_1n_2}$ and the second field stores the pages of the adjacency lists of the segments' nodes n_1 and n_2 . P2 is the pointer to disk page 2, where the adjacency list l_1 is stored and P1 is the pointer to the disk page 1, where the adjacency list l_2 is stored.

The **network R-Tree** enables us to do spatial queries on the network. It stores location information for all the segments of the network; for every segment it stores the MBR of its nodes and a pointer to the page containing the segment's polyline representation.

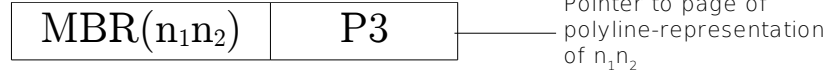


Figure 7. One network R-Tree entry.

Figure 7 shows one leaf entry of the network R-Tree. This entry corresponds to the segment n_1n_2 . The first field stores the MBR of nodes n_1 and n_2 , the second entry is a pointer to the page of the polyline representation of segment n_1n_2 .

Combining these three data structures results in an architecture that can be seen in a bigger picture in Figure 8. The three components are linked in the following way: There is a pointer in each adjacency list entry to the entries' (segment's) polyline representation in the polyline component. There is also a pointer from each polyline component entry back to the adjacency list entry it represents. Each entry (segment) of the network R-Tree stores a pointer to its own polyline component entry. Like this, it is possible to query the R-Tree for a segment and then gain the segment's polyline representation, which gives us more detailed information about the segment's spatial layout, and its adjacency list, which allows us to compute network-related algorithms such as shortest-paths.

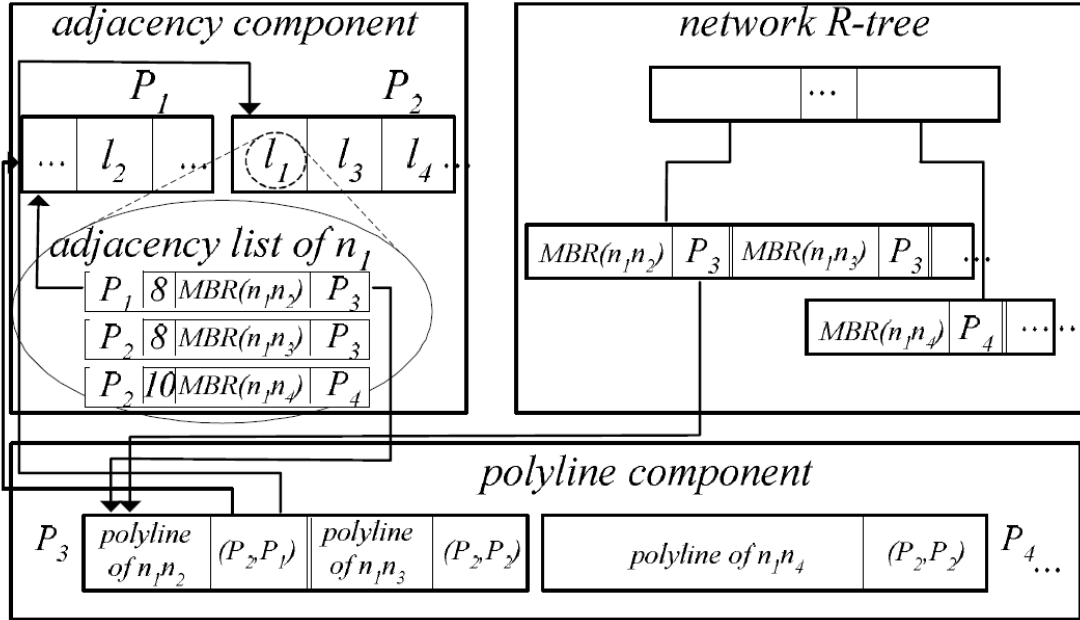


Figure 8. The proposed SNDB architecture with exemplary entry values.

4.3 Primitive operations

We can now use this architecture to implement four useful primitive operations, that allow us to get location information from the network. For each of these operations its semantics and a short description of its implementation will be given.

check_entity(seg,p) returns true iff entity p lies on segment seg . The result is computed by querying the network R-Tree for segment seg and obtaining the polyline representation P_{seg} . Due to approximation or digitization errors it is possible that although entity p actually lies on seg , its stored coordinates may deviate from the segment. This case can be handled by defining a threshold dT , such that if p is within distance dT from seg , we can assume that it lies on it.

find_segment(p) returns the segment s on which entity p is located (we say that s covers p). This is implemented by performing a point location query on the network R-Tree. The first segment that covers p is returned.

find_entities(seg) returns all entities located on segment seg. This is implemented by first finding all candidate entities that lie in the MBR of seg using the network R-Tree. P_{seg} is then used to filter out the false hits, i.e. the entities that are in seg's MBR but which lie not on seg

compute_ND(p_1, p_2) computes the network distance $d_N(p_1, p_2)$ using Dijkstra's Shortest-Path-algorithm which is explained in [9]. Dijkstra's algorithm assumes that the two points whose shortest path we wish to compute are both nodes, which is not necessarily the case in our architecture (entities may lie on a segment).

In this case, if our start point p_1 is not a node, we have to compute the residual distances $d_R(p_1, n_1)$, $d_R(p_1, n_2)$ to the two nodes n_1, n_2 of the segment. We then initialize the Queue Q for Dijkstras algorithm with $Q = [(n_1, d_R(p_1, n_1)), (n_2, d_R(p_1, n_2))]$ and then continue Dijkstra's algorithm as usual. If the destination point also is not a node, we repeat the procedure used for the start point.

5. 2 kNN-Algorithms

In the paper [1], for each of the 4 most popular spatial queries 2 algorithms are proposed, one for each that is based on the *Euclidian restriction* framework and another one that is based on *Network Expansion*. The idea of Euclidian restriction is to use the Euclidian lower-bound property to efficiently prune the search space, while the approach of Network expansion is to perform a straight-forward expansion of the network graph.

In this report, we will focus on the proposed kNN-algorithms. In the following, at first the IER variant of kNN is proposed and after this the INE version of kNN is developed.

5.1 Incremental Euclidian Restriction (IER)

IER applies the multistep-kNN method [5,6], which will not be explained here.

We will first look at the case where $k=1$ and then explain how to retrieve k nearest neighbors.

We will use an example that is shown in Figure 9. q is the query point for which we want to compute the first NN. We proceed in the following way.

Firstly, the Euclidian NN p_{E1} of q is computed using the entity R-Tree in which p_{E1} is stored. This sets the boundary $d_{E_{max}} = d_N(q, p_{E1})$ because: All other potential NNs of q must be within the range $d_{E_{max}}$ from q because of the Euclidian lower-bound property. This range is shown in the gray shaded area in Figure 9.

In a second step, the second Euclidian NN p_{E2} of q is computed again using its entity R-Tree. p_{E2} is still in the valid range (the grey-shaded area). Because of this, it becomes a new candidate for q 's NN. As $d_N(q, p_{E2}) < d_N(q, p_{E1})$, it promotes from a NN candidate to the current NN of q . The Euclidian lower-bound is reset from $d_E(q, p_{E1})$ to $d_E(q, p_{E2})$ which corresponds to the reduction of the white area in Figure 9. The next Euclidian NN p_{E3} is already outside the boundary $d_{E_{max}}$. Thus the program can terminate with p_{E2} as q 's NN, because the Euclidian lower-bound property assures that there are no further NNs of q .

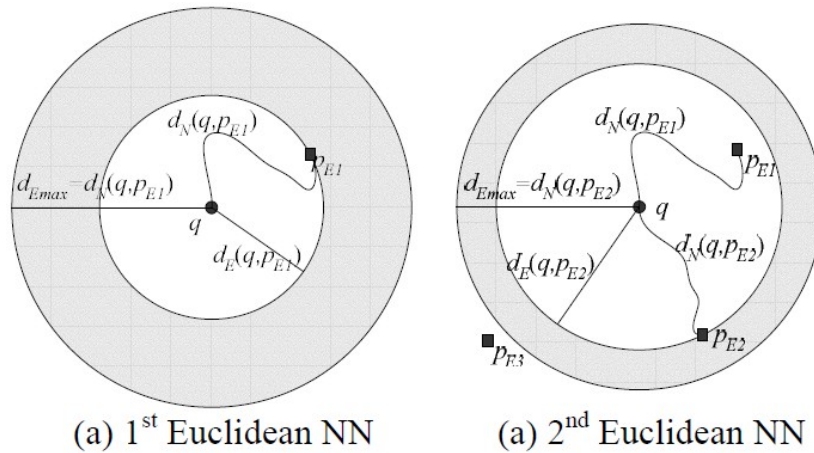


Figure 9. The first two steps of IER if $k=1$.

The extension of IER to k nearest neighbors is straight-forward. Using the entity R-tree the k Euclidian nearest neighbors of q are retrieved and then sorted (and kept sorted throughout the algorithm) in ascending order of their network distance to q . Similar to the case where $k=1$, we set the boundary $d_{E_{\max}}$ to the network distance of the farthest away nearest neighbor (i.e. the k^{th} entry of the list of NNs), because any other NN has to be in range $d_{E_{\max}}$. We now repeatedly compute the next Euclidian NN, check if its network distance to q is smaller than the one of the k^{th} current NN (which makes it a network nearest neighbor candidate) and insert it into the current list of NNs so that the list remains sorted in ascending order of network distances to q while updating $d_{E_{\max}}$ updated, until the next Euclidian NN is outside the range of $d_{E_{\max}}$. When the program stops, the list of current NNs is the set of q 's k Nearest Neighbors. The problem of IER is that it performs well if the ranking of the data points by their Euclidian distance is similar to the ranking of the data points by their network distance. If this is not the case, however, we will visit a lot of Euclidian NNs that are no candidates for a network NN resulting in a high number of redundant computations. The detailed pseudocode of IER can be found in [1], section 4.1.

5.2 Incremental Network Expansion (INE)

INE tries to avoid the above mentioned problem of IER by performing a network expansion with starting point q . It then visits all the entities in the order they are encountered along the network.

How INE works for $k=1$ will now be illustrated using an example that is shown in Figure 10. Again, q is our query point. Expanded nodes will be shown in orange, not-expanded nodes are shown in black. INE proceeds as follows.

At first, the segment n_1n_2 that covers q is located with `find_segment( $q$ )`. Then we retrieve all entities that are on seg with `find_entities( $n_1n_2$ )`. As there are no entities on n_1n_2 , we start the network expansion. First, we visit the node n with smallest $d_R(q, n)$, which is n_1 and expand it. Expanding a node means that we retrieve all segments that go out from it and look for entities on them with `find_entities(seg)`. As there are no entities on n_1n_7 , we expand the next nearest node to q , which is n_2 . The inspection of n_2 's outgoing segments yields p_5 . p_5 must be q 's NN since $d_N(q, p_5)$ provides a bound $d_{N_{\max}}$ that restricts the search space. All other possible NNs have to be more far away.

As for IER, the extension of INE to k NNs is straight-forward. We use a list of k current NNs and update it each time when expanding a node. When implementing INE, similarly to Dijkstra's algorithm, we use a queue Q of nodes that is sorted in ascending order of the nodes' network distance to q . The entries of Q represent the nodes which we still have to expand. As long as the network distance of the first node (i.e. the node p with $\min(d_N(q, p))$) is below a threshold $d_{N_{\max}}$ which is determined by the network distance of the k -th element of the list of the current k NNs, we continue expanding the network and updating the list of nearest neighbors. We also omit pseudocode for INE, but it can be retrieved from [1], section 4.2.

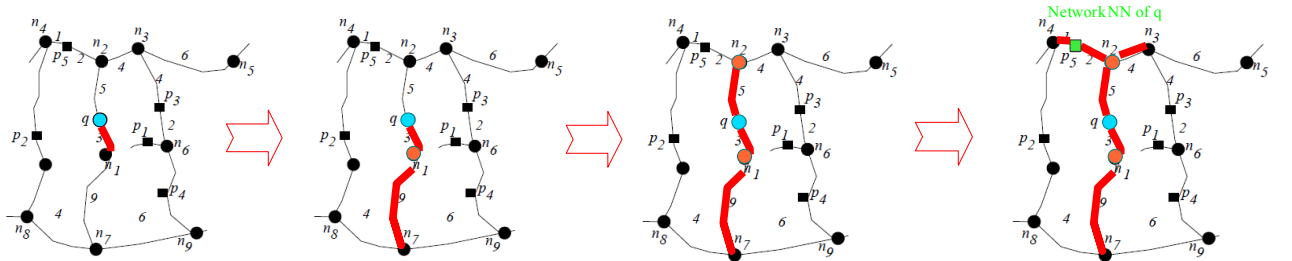


Figure 10. Using INE to find q 's NN

6. Performance of network-kNN

In the following, we will compare the performance of the two proposed kNN-algorithms IER and INE against each other. First, we evaluate their performance for different cardinality ratios, then for different k with constant cardinality ratio and finally we will explore the

False Hit behavior of IER. The cardinality ratio for this setup is defined as $|S|/|N|$, where $|S|$ is the number of entities and $|N|$ is the number of segments in the network.

All experiments were done using a Pentium III with 700 Mhz running Windows NT. The page size of the data structures was set to 4KB and uses an LRU-buffer which has place for 10% of the network and 10% of each R-Tree participating in an experiment. The spatial network represents main roads in North America and has a size of $|N| = 179000$ segments. Different entity datasets with different sizes $|S|$ will be used to control the behavior of the algorithms for different entity densities.

The distribution of the entities corresponds to the distribution of the network, i.e. in locations with high road density there will very likely be more entities.

We execute 200 kNN-queries that also follow the network distribution, i.e. there are more queries in regions with higher segment density.

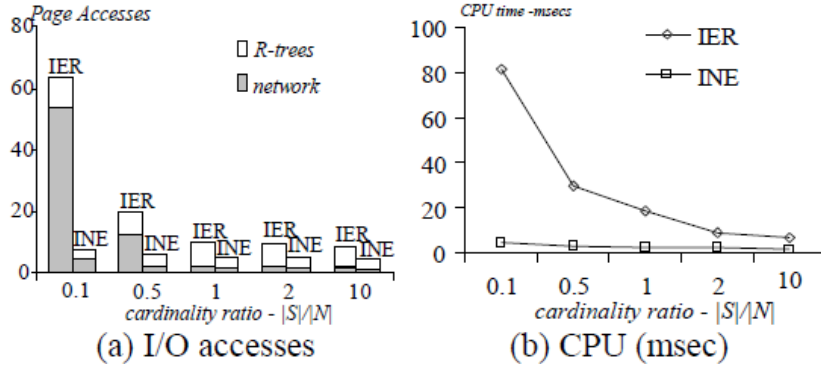


Figure 12. Cost vs $|S|/|N|$ ($k=10$)

As can be seen in Figure 12, in the cases where the cardinality ratio is very low i.e. in cases where there are a lot more network segments than entities IER performs poorly, especially the cost for computing network distances makes up a big part of the time spent. This is because if the entity set $|S|$ is small, the Euclidian Nearest Neighbors are likely to be far away from the query point, increasing the number of false positives (as can be seen in Figure 13 (a)) and leading to a lot of unnecessary network distance (i.e. shortest-path) computations.

The performance of INE is constant for all cardinality ratios, this is because only the necessary network edges are visited, the range queries on the R-Tree exhibit high locality and the search path is in the buffer with high probability.

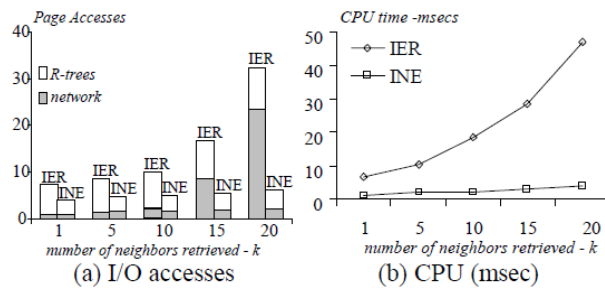


Figure 13 Cost vs k ($|N|=|S|$)

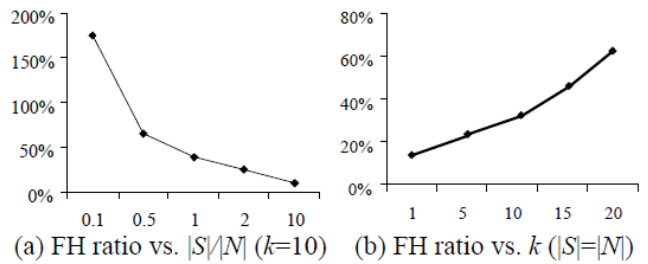


Figure 14. False Hits by IER

The results shown in Figure 12 might be explained in a same way: as k increases, the distance to the query point of other entities involved in the algorithm increases and with it increases the number of false hits, letting IER perform very bad. The False Hit Ratio of IER for varying and constant k can be observed in Figure 13(b) and clearly supports our explanations from before.

When looking at these measurements, it is clearly seen that INE would be a more efficient choice to process network kNN queries for any k and for any cardinality ratio. According to the measurements, IER should not be chosen in any case.

7. Conclusion

In this paper, which can be seen as a first try to create an SNDB we have constructed an architecture that combines Euclidian and network information and allows for the development of algorithms that use both information types.

As this is one of the first publications trying to create a useful architecture for SNDB and one of the first to create algorithms to process the conventional spatial query types kNN, range search, closest-pair and e-distance-Join on a spatial network, one can identify some weak spots in the ideas presented here.

(1) the complete Euclidian restriction framework which makes up four of eight of the proposed algorithms is based on the idea that the Euclidian lower-bound property always holds. However, the Euclidian lower-bound property only holds if the edge cost is defined as the Euclidian length of its polyline. If the edge cost is defined as another measure, e.g. the travel time we cannot use the algorithms proposed in the Euclidian restriction framework, unless making additional assumptions such as maximum speed.

The Network expansion framework on the other hand is independent of the Euclidian Lower bound property. Its only assumption is that edge costs cannot be negative (which is also the assumption of Dijkstra's algorithm), which is a realistic assumption that should hold in most real life networks.

(2) as can be seen in section 6, the performance of Incremental Euclidian Restriction degrades very fast if the cardinality ratio is small, the performance is worst when the cardinality ratio is 0.1, while in real networks the cardinality ratio is usually even smaller than 0.1. This makes the IER algorithm unusable in practice. The INE is not concerned by this however, as it shows constant performance for varying cardinality ratios and varying k.

(3) All algorithms proposed in [1] using Dijkstra's algorithm (such as IER or Closest-Pairs-ER) to compute the Network distance suffer from the following problem. When using a Fibonacci-Heap [10], Dijkstra's algorithm has time complexity $O(n \log(n) + m)$, where n denotes the number of vertices and m the number of edges. This means that in the worst case the algorithm would have to visit every edge and every vertice. The problem is that in real networks in which objects might have a large distance in terms of network hops, Dijkstra will very likely visit a large percentage of all nodes in the network before reaching a node that coincides with or is the object we are interested in. In [7] it is shown that in a real life networks Dijkstra's algorithm might easily visit 75% of the network's nodes in order to compute the shortest path between two nodes that are at two opposite ends of a network. However, in real-life scenarios (e.g. using GoogleMaps to update a travel route on-the-fly) we are interested in real-time or near real-time responses, for which Dijkstra's algorithm will not be suitable. This problem might be tackled by partitioning the network into smaller subnetworks. In [7], an approach is presented that tries to do this and which manages to avoid visiting a big part of the nodes, giving us real-time responses.

(4) The performance evaluation uses the metrics CPU time and I/O-Accesses. Instead of having these rather unexact measures, one might be interested in measuring in other more meaningful metrics like the number of nodes visited or the number of shortest-path computations.

To sum it up, this paper pointed out the importance of taking into account the underlying network when processing queries on spaces where there is a pre-defined real-life network and presented an architecture that captures the Euclidian information of the network as well as its connectivity information, enabling us to develop basic algorithms for the network version of most common spatial queries. The presented algorithms however are very basic and not optimized, as the authors point out themselves. Their intention was to show the general methodology, for this reason only the most basic optimizations have been done and the algorithms only take care of the most basic query forms. However, some special cases such as „find the nearest hotels to the south“ might be computed by adapting the proposed algorithms.

This paper opened a door for several interesting research directions, e.g. we might also be interested in computing continuous Nearest Neighbor (CNN) queries on spatial network databases. For this, some of the authors has proposed a solution in 2006 [8].

Another possibility might be to optimize the proposed algorithms or to think about improving the architecture.

8. References

- [1] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao: Query Processing in Spatial Network Databases. *In Proc. of VLDB*, 2003.
- [2] H.-J. Cho and C.-W. Chung. An efficient and scalable approach to CNN queries in a road network. In *VLDB'05*, pp. 865-876, Trondheim, Norway, Sep. 2005.
- [3] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching, *SIGMOD*, 1984.
- [4] Shekhar, S., Liu, D. CCAM: A Connectivity-Clustered Access Method for Networks and Network Computations. *TKDE*, 19(1), 102-119, 1997.
- [5] Faloutsos, C., Ranganathan, M., Manolopoulos, Y. Fast Subsequence Matching in Time-Series Databases. *SIGMOD*, 1994.
- [6] Seidl, T., Kriegel, H. Optimal Multi-Step k-Nearest Neighbor Search. *SIGMOD*, 1998.
- [7] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–55, 2008.
- [8] Kyriakos Mouratidis, Man Lung Yiu, Dimitris Papadias, Nikos Mamoulis: Continuous Nearest Neighbor Monitoring in Road Networks. *VLDB 2006*: 43-54
- [9] E. Dijkstra. A Note on Two Problems in Connection with Graphs. *Numerische Mathematik*, vol. 1, 1959, pp. 269-270.
- [10] Fredman, M. L. and Tarjan, R. E. 1987. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM* 34, 209–221.