Query Processing in Spatial Network Databases

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clearwater Bay, Hong Kong
dimitris@cs.ust.hk

Jun Zhang
Department of Computer Science
Hong Kong University of Science and Technology
Clearwater Bay, Hong Kong
zhangjun@cs.ust.hk

Nikos Mamoulis
Department of Computer Science and Information Systems
University of Hong Kong
Pokfulam Road, Hong Kong
nikos@csis.hku.hk

Yufei Tao
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
taoyf@cityu.edu.hk

Abstract

Despite the importance of spatial networks in real-life applications, most of the spatial database literature focuses on Euclidean spaces. In this paper we propose an architecture that integrates network and Euclidean information, capturing pragmatic constraints. Based on this architecture, we develop a *Euclidean restriction* and a *network expansion* framework that take advantage of location and connectivity to efficiently prune the search space. These frameworks are successfully applied to the most popular spatial queries, namely *nearest neighbors*, *range search*, *closest pairs* and *e-distance joins*, in the context of spatial network databases.

1. Introduction

Spatial databases have been well studied in the last 20 years resulting in the development of numerous conceptual models, multi-dimensional indexes and query processing techniques [RSV02]. Surprisingly, most of existing work considers Cartesian (typically, Euclidean) spaces, where the distance between two objects is determined solely by their relative position in space. However, in practice, objects can usually move only on a pre-defined set of trajectories as specified by the underlying network (road, railway, river etc.). Thus, the important measure is the network distance, i.e., the length of the shortest trajectory connecting two objects, rather than their Euclidean distance. Previous work on spatial network databases (SNDB) is scarce and too restrictive for emerging applications such as mobile computing and location-based commerce. This necessitates the development of novel and comprehensive query processing methods for SNDB.

Every conventional spatial query type (e.g., *nearest neighbors*, *range search*, *spatial joins* and *closest pairs*) has a counterpart in SNDB. Consider, for instance, the road network of Figure 1.1, where the rectangles correspond to hotels. If a user at location *q* poses the range query "find the hotels within a 15km range", the result will contain *a*, *b* and *c* (the numbers in the figure correspond to network distance). Similarly, a nearest neighbor query will return hotel *b*. Note that the results of the corresponding conventional queries are different (e.g., the Euclidean nearest neighbor is *d*, which is actually the farthest hotel in the network). Furthermore, queries may combine both location and network aspects, such as "find the nearest hotel to the south" (e.g., hotel *a*).

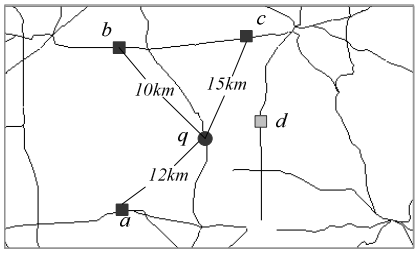


Figure 1.1: Road network query example

A crucial pre-requisite for solving these queries in SNDB is a realistic architecture, which captures *spatial entities* (e.g., hotels) and the underlying network, preserving both Cartesian co-ordinates and connectivity. In addition, this architecture must take into account real-life constraints, such as unidirectional roads, “off-network” (but still reachable) entities, etc. Furthermore, although the network is almost static, the entities may change with high frequencies (e.g., a new/existing hotel opens/closes).

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

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

It is also possible that entire entity sets are added as more information or services become available (e.g., a new "restaurant" dataset is incorporated in the system).

In this paper we propose a flexible architecture for SNDB, by separating the network from the entity datasets. In particular, we employ a disk-based network representation that preserves connectivity and location, while spatial entities are indexed by respective spatial access methods for supporting Euclidean queries and dynamic updates. Using this architecture, we develop two frameworks, *Euclidean restriction* and *network expansion*, for processing the most common spatial queries, namely nearest neighbors, range search, closest pairs and distance joins. The resulting algorithms expand conventional processing techniques by integrating connectivity and location information for efficient pruning of the search space. To the best of our knowledge, this is the first work dealing with the efficient processing of spatial queries in SNDB.

The rest of the paper is organized as follows: Section 2 overviews related work. Section 3 describes our architecture and its application in real-life scenarios. Sections 4 and 5 present algorithms for nearest neighbor and range search queries, respectively, while Sections 6 and 7 discuss closest pairs and spatial joins. Section 8 evaluates the proposed techniques with comprehensive experiments, and Section 9 concludes the paper with a discussion.

2. Related Work

In this section we overview previous work related to location (Section 2.1) and connectivity (Section 2.2) representation and processing in databases.

2.1 Spatial Query Processing in the Euclidean Space

R-trees [G84, SRF87, BKSS90] are the most popular indexes for Euclidean query processing due to their simplicity and efficiency. The R-tree can be viewed as a multi-dimensional extension of the B-tree. Figure 2.1 shows an exemplary R-tree for a set of points $\{a,b,\ldots,j\}$ assuming a capacity of three entries per node. Points that are close in space (e.g., a,b) are clustered in the same leaf node (E_3) represented as a minimum bounding rectangle (MBR). Nodes are then recursively grouped together following the same principle until the top level, which consists of a single root.

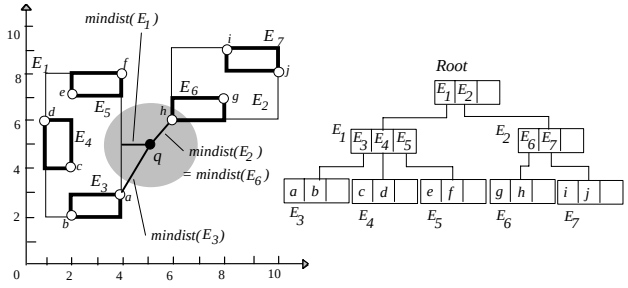


Figure 2.1: An R-tree example

R-trees (like most spatial access methods) were motivated by the need to efficiently process *range queries*, where the range usually corresponds to a rectangular window or a circular area around a query point. The R-tree answers the query q (shaded area) in Figure 2.1 as follows. The root is first retrieved and the entries (e.g., E_1, E_2) that intersect the range are recursively searched because they may contain qualifying points. Non-intersecting entries (e.g., E_3) are skipped. Note that for non-point data (e.g., lines, polygons), the R-tree provides just a *filter* step to prune non-qualifying objects. The output of this phase has to pass through a *refinement* step that examines the actual object representation to determine the actual result. The concept of filter and refinement steps applies to all spatial queries on non-point objects.

A *nearest neighbor* (NN) *query* retrieves the (k≥1) data point(s) closest to a query point q. The R-tree NN algorithm proposed in [HS99] keeps a *heap* with the entries of the nodes visited so far. Initially, the heap contains the entries of the root sorted according to their minimum distance (*mindist*) from q. The entry with the minimum *mindist* in the heap (E_1 in Figure 2.1) is expanded, i.e., it is removed from the heap and its children (E_3, E_4, E_5) are added together with their *mindist*. The next entry visited is E_2 (its *mindist* is currently the minimum in the heap), followed by E_6, where the actual result (h) is found and the algorithm terminates, because the *mindist* of all entries in the heap is greater than the distance of h. The algorithm can be easily extended for the retrieval of k nearest neighbors (kNN). Furthermore, it is optimal (it visits only the nodes necessary for obtaining the nearest neighbors) and *incremental*, i.e., it reports neighbors in ascending order of their distance to the query point, and can be applied when the number k of nearest neighbors to be retrieved is not known in advance.

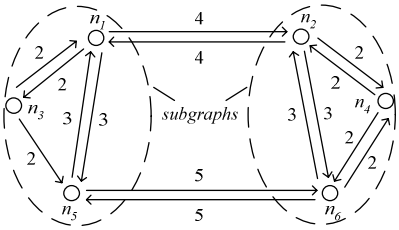
An *intersection join* retrieves all intersecting object pairs (s,t) from two datasets S and T. If both S and T are indexed by R-trees, the *R-tree join* algorithm [BKS93] traverses synchronously the two trees, following entry pairs that overlap; non-intersecting pairs cannot lead to solutions at the lower levels. Several spatial join algorithms have been proposed for the case where only one of the inputs is indexed by an R-tree or no input is indexed [RSV02]. For point datasets, where intersection joins are meaningless, the corresponding problem is the *e-distance join*, which finds all pairs of objects (s,t) $s \in S$, $t \in T$ within (Euclidean) distance e from each other. *R-tree join* can be applied in this case as well, the only difference being that a pair of intermediate entries is followed if their distance is below (or equal to) e. The intersection join can be considered as a special case of the e-distance join, where e=0.

Finally, a *closest-pairs* query outputs the (k≥1) pairs of objects (s,t) $s \in S$, $t \in T$ with the smallest (Euclidean) distance. The algorithms for processing such queries [CMTV00] combine spatial joins with nearest neighbor

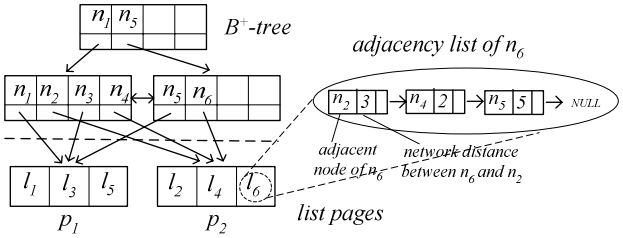
search. In particular, assuming that both datasets are indexed by R-trees, the trees are traversed synchronously, following the entry pairs with the minimum distance. Pruning is based on the *mindist* metric, but this time defined between entry MBRs. As all these algorithms apply only location-based metrics to prune the search space, they are inapplicable for SNDB.

2.2 Disk-based Graph Representations and Algorithms

A graph is usually represented either as a 2D matrix (where each entry corresponds to an edge between a pair of nodes), or an adjacency list. Adjacency lists are preferable for applications, such as road networks, where the graphs are sparse. The main issue for adapting this representation to secondary memory is how to cluster lists of adjacent nodes in the same disk page, in order to take advantage of the access locality and minimize the I/O. The connectivity-clustered access method (CCAM) [SL97] generates a single-dimensional ordering of the nodes (using Z-ordering) and stores the lists of neighbor nodes together. Figure 2.2 shows a graph example and its CCAM structure, assuming that three adjacency lists fit in one page. The lists of the neighboring (in the Z-order) graph nodes n_1, n_3 and n_5 are stored in page p_1, and the lists of the remaining ones in page p_2. Each entry in the list (e.g., l_6) of a node (n_6) contains an adjacent node (n_2) and the corresponding network distance (3). In order to efficiently retrieve the adjacency list l_i of a node n_i, the list pages are indexed by a B^+-tree on the node id. An alternative technique for clustering graph nodes according to their proximity in space is proposed in [HJR96].



(a) A graph example



(b) The CCAM structure

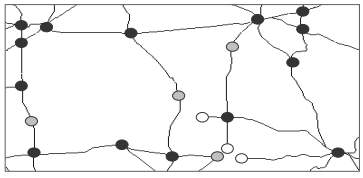
Figure 2.2: A graph example and its CCAM structure

CCAM and similar architectures only preserve connectivity (but not location) information; thus, they are applicable only to *shortest path* and other graph traversal algorithms (but not conventional spatial query processing). The most popular shortest path algorithm, proposed by Dijkstra [D59], starts from the source and expands the route towards the destination, using a priority queue to store visited nodes (sorted according to their distance from the source node). Several variants of this algorithm [CLR90] differ on how they manage the priority queue. The A* algorithm [KHI+86] applies heuristics to prune the search space and direct the graph expansion. Materialization techniques accelerate shortest path processing (at the expense of space requirements) by using pre-computed results stored in materialized views [ADJ90, IRW93, JP96, JHR98]. The performance of secondary-memory adaptations of shortest path algorithms has been analyzed in [J92, SKC93].

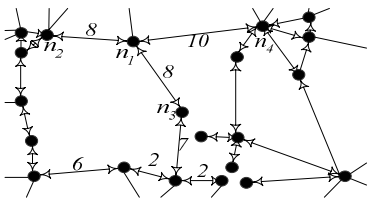
The only existing approach that integrates spatial and connectivity information [HJR97], uses thematic spatial constraints to restrict the permitted paths (e.g., "find the shortest path that passes only through rural areas") and is inapplicable to general query processing. Finally, [SKS02] deal with nearest neighbor queries in road networks by transforming the problem to high dimensional space. Their solution is approximate and specific to this problem. On the other hand, the architecture presented in the next section supports all the counterparts of conventional queries in the SNDB context.

3. Architecture

We assume a digitization process that generates a *modeling graph* from an input spatial network. Considering the road network in the introduction, the graph nodes generated by this process are: (i) the network junctions (e.g., the black points in Figure 3.1a), (ii) the starting/ending point of a road segment (white), and (iii) depending on the application, additional points (gray) such as the ones where the curvature or speed limit changes. The graph edges preserve the connectivity in the original network. Figure 3.1b shows the (modeling) graph for the network of Figure 3.1a; nodes at the boundary of the data space and the network distance of most edges are omitted for clarity.



(a) A road network

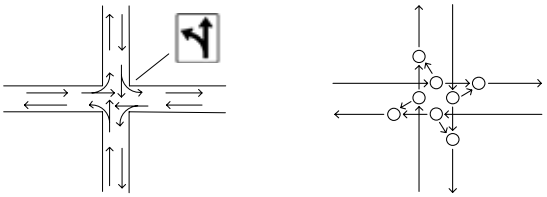


(b) The modeling graph

Figure 3.1: Graph modeling of the road network

In the sequel we use the term edge/segment to denote a direct link in the graph/network. Each edge connecting nodes n_i, n_j stores the *network distance* $d_N(n_i,n_j)$. For nodes that are not directly connected, $d_N(n_i,n_j)$ equals the length of the shortest path from n_i to n_j. If unidirectional traffic is allowed (e.g., one-way road segments), $d_N(n_i,n_j)$ is asymmetric (i.e., it is possible $d_N(n_i,n_j) \neq d_N(n_j,n_i)$). Furthermore, $d_E(n_i,n_j) \leq d_N(n_i,n_j)$, i.e., the corresponding Euclidean distance $d_E(n_i,n_j)$ *lower bounds* $d_N(n_i,n_j)$ (equality holds only if n_i, n_j are connected by a straight segment). We refer to this fact as the *Euclidean lower-bound property*.

Constraints, such as special traffic controls, can be modeled by including extra nodes to the graph. As an example, consider a road junction in Figure 3.2a, where right turns are not permitted. The corresponding graph is shown in Figure 3.2b, where 8 nodes in the same spatial position are used to capture this behavior. Depending on the application needs, additional information may be kept for nodes (such as the type of the node, e.g., highway junction) or the edges (speed limit, category of road segment e.g., highway).



(a) A road junction (b) The modeling graph

Figure 3.2: Example of pragmatic constraint

In order to simplify the presentation, we describe our architecture for the basic functionality, where nodes have identical types and edges only store network distance. We separate the spatial entities (e.g., hotels) from the underlying network, by indexing each entity dataset using an R-tree. This division has many advantages: (i) all conventional (Euclidean) queries, which do not require the network, can be efficiently processed by the R-trees, (ii) as shown later, queries combining network and Euclidean aspects are supported, (iii) dynamic updates in each dataset are handled independently, (iv) new/existing datasets can be added to/removed from the system easily, and (v) specific optimizations can be applied to each individual (network or entity) dataset.

The network storage scheme consists of three components. The *adjacency component* captures the network connectivity. The adjacency lists of the nodes close in space (according to their Hilbert[1] values) are placed in the same disk page. In Figure 3.3 (based on Figure 3.1b), the list l_1 of n_1 consists of 3 entries, one for each of its connected nodes (n_2, n_3, n_4) (ignoring nodes outside the boundary). The first entry (for edge n_1n_2) has the form <NBptr(n_2), 8, MBR(n_1n_2), PLptr(n_1n_2)>, where NBptr(n_2) points to the disk page (i.e., P_1) containing the adjacency list l_2 of n_2. NBptr(n_2) enables fast access to the neighboring node n_2, without any additional look-up (while CCAM, as reviewed in Section 2.2, requires B-tree accesses). The next field (8) is the network distance of edge n_1n_2. MBR(n_1n_2) records the minimum bounding rectangle of the actual poly-line n_1n_2 in the original network, which is stored in the disk page (=P_3) specified by PLptr(n_1n_2). The other adjacency entries (for n_3, n_4) have the same format.

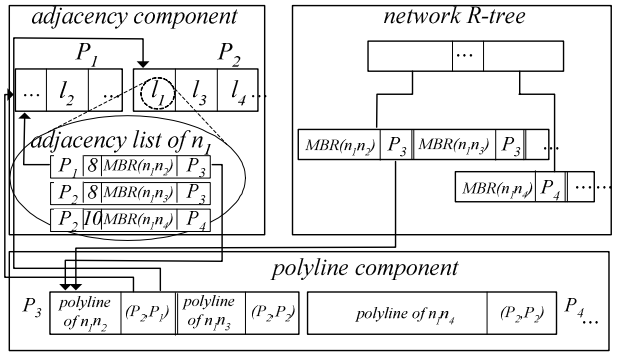


Figure 3.3: Example of the proposed architecture

The *poly-line component*, stores the detailed poly-line representation of each segment in the network. A poly-line entry n_in_j also includes a pair of pointers to the disk pages containing the adjacency lists of its endpoints n_i, n_j. The last component is a *network R-tree* that indexes the poly-lines' MBRs and supports queries exploring the spatial properties of the network. Each leaf entry contains a pointer to the disk page storing the corresponding poly-line. The architecture supports the following primitive operations for SNDB:

(i) *check_entity*(*seg*, *p*) is a Boolean function that returns true if point (entity) *p* lies on the network segment *seg* (we say that *seg covers p*). In accordance with the conventional spatial databases methodology, the MBR of *seg* is used for filtering and its poly-line representation for refinement. Due to approximation or digitization errors it is possible that, although point *p* actually lies on *seg*, its stored co-ordinates may deviate from the segment. This situation can be handled by defining an (application-dependent) threshold *dT*, such that if *p* is within distance *dT* from *seg*, it is assumed to lie on it.

(ii) *find_segment*(*p*): outputs the segment that covers point *p* by performing a point location query on the network R-tree. If multiple segments cover *p*, the first one found is returned. This function is applied whenever a query is issued, to locate the segment on which the query point lies. If the query point does not lie on any segment, we can "snap" it to the closest one assuming incomplete information (e.g., an un-recorded alley), or we can consider it unreachable depending on the application specifications. Although uncertainty handling in SNDB is an interesting topic, for the sake of simplicity, in the

[1] We apply Hilbert ordering, instead of the Z-ordering used by CCAM, because it achieves better locality.

following discussion we assume that each entity and query point falls on at least one network segment.

(iii) *find_entities*(*seg*): returns entities covered by segment *seg*. Specifically it first finds all the candidate entities that lie in the MBR of *seg*, and then eliminates the false hits using the poly-line of *seg*.

(iv) *compute_ND*(p_1,p_2): returns the network distance $d_N(p_1,p_2)$ of two arbitrary points p_1, p_2 in the network, by applying a (secondary-memory) algorithm to compute the shortest path from p_1 to p_2. Specifically, we chose to adapt Dijkstra's algorithm because it is simple, efficient and exhibits access locality, reducing the number of page faults during the retrieval of adjacency lists. However, Dijkstra's algorithm assumes that the source (i.e., query) and the destination (i.e., data point) fall on network nodes, while in our scenario points may fall on edges (or assigned to edges if approximation is used).

Consider, for example, the computation of $d_N(p_1,p_2)$ in the modeling graph of Figure 3.4, where $n_1,...,n_6$ denote the nodes. The algorithm first invokes *find_segment* to return the segments n_1n_2 covering p_1, and n_5n_6 covering p_2. Then, it calculates the distance from p_1 to n_1 and n_2 (5 and 12) using the poly-line n_1n_2, and initiates a priority queue Q=<(n_1,5),(n_2,12)>. The first entry n_1 is de-queued and its adjacent nodes (n_3, n_4) are inserted into Q, together with their accumulated distance from p_1, i.e., Q=<(n_2,12), (n_3,13), (n_4,30)>. After the expansion of n_2, the queue becomes Q=<(n_3,13),(n_4,25)> (the distance to n_4 decreases), and after the expansion of n_3, Q=<(n_5,23), (n_4,25)>. Now the next node to expand is n_5. Since p_2 can be reached from n_5 with cost 3, we insert it, and the queue becomes Q=<(n_4,25), (p_2,26)>. Similarly, after the expansion of n_4, Q=<(p_2,26), (n_6,29)> and the algorithm terminates with $d_N(p_1,p_2)$=26. If p_1 or p_2 fall on multiple segments, then by the definition of the graph they correspond to graph nodes, in which case the algorithm is still applicable. The same is true for networks containing unidirectional segments.

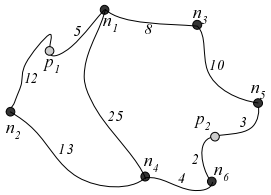


Figure 3.4: Illustration of fundamental operations

Network distance computations can be facilitated by materialization of pre-computed results (e.g., [ADJ90, JHR98]). In Figure 3.4, for example, $d_N(p_1,p_2)$ can be obtained by fetching from the materialized view $d_N(n_1,n_5)$=18, $d_N(n_1,n_6)$=23, $d_N(n_2,n_5)$=22, $d_N(n_2,n_6)$=17, with four disk accesses using a hash function. Although materialization can be incorporated as an additional module in our architecture, we chose not to include it in the basic functionality due to the huge space requirements for large spatial networks.

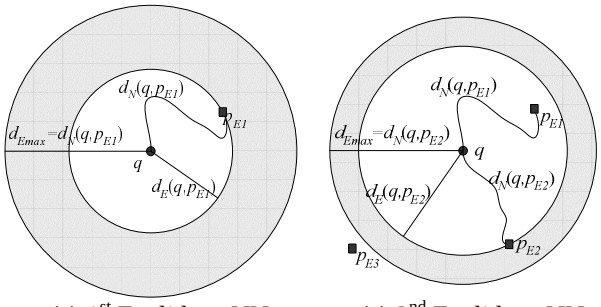
Next, we discuss all common spatial queries using this architecture. For each query type we propose two algorithms based on the *Euclidean restriction* and *network expansion* frameworks, respectively. Euclidean restriction takes advantage of the Euclidean lower-bound property to prune the search space. On the other hand, the network expansion framework performs query processing directly on the network. Since our aim is to illustrate the general methodology, we intentionally keep the algorithms simple and only present essential optimization techniques wherever necessary. Furthermore, we only describe the basic query forms; as discussed in Section 9, variations such as "find the nearest hotels to the south", can be easily processed by the proposed techniques.

4. Nearest Neighbors in SNDB

Given a source point q and an entity dataset S, a k nearest neighbor (kNN) query retrieves the k (≥1) objects of S closest to q according to the network distance (e.g., find the hotel within the shortest driving distance). Sections 4.1 and 4.2 present two algorithms for nearest neighbour queries in SNDB, based on the Euclidean restriction and network expansion frameworks, respectively.

4.1 Incremental Euclidean Restriction

The Incremental Euclidean Restriction (IER) algorithm applies the multi-step kNN methodology [FRM94, SK98], traditionally used for high-dimensional similarity retrieval. Specifically, assuming that only one NN is required, IER first retrieves the Euclidean nearest neighbor p_{E1} of q, using an incremental kNN algorithm (e.g., [HS99], see Section 2.1) on the entity R-tree of S. Then, the network distance $d_N(q,p_{E1})$ of p_{E1} is computed (by *compute_ND*(q,p_{E1})). Due to the Euclidean lower-bound property, objects closer (to q) than p_{E1} in the network, should be within Euclidean distance $d_{Emax}=d_N(q,p_{E1})$ from q, i.e., they should lie in the shaded area of Figure 4.1a. In Figure 4.1b, the second Euclidean NN p_{E2} is then retrieved (within the d_{Emax} range). Since $d_N(q,p_{E2})<d_N(q,p_{E1})$, p_{E2} becomes the current NN and d_{Emax} is updated to $d_N(q,p_{E2})$, after which the search region (for potential results) becomes smaller (the shaded area in Figure 4.1b). Since the next Euclidean NN p_{E3} falls out of the search region, the algorithm terminates with p_{E2} as the final result.



(a) 1st Euclidean NN (a) 2nd Euclidean NN

Figure 4.1: Finding the NN p_{E2}

The extension to k nearest neighbors is straightforward. The k Euclidean NNs are first obtained using the entity R-tree, sorted in ascending order of their network distance to q, and d_{Emax} is set to the distance of the k^{th} point. Similar to the single NN case, the subsequent Euclidean neighbors are retrieved incrementally, while maintaining the k (network) NNs and d_{Emax} (except that d_{Emax} equals the network distance of the k-th neighbor), until the next Euclidean NN has larger Euclidean distance than d_{Emax}. Figure 4.2 illustrates the pseudo-code of IER.

Algorithm IER (q, k)
/* q is the query point */
1. $\{p_1,...,p_k\}$=Euclidean_NN(q,k);
2. for each entity p_i
3. $d_N(q,p_i)$=*compute_ND*(q,p_i)
4. sort $\{p_1,...,p_k\}$ in ascending order of $d_N(q,p_i)$
5. d_{Emax}= $d_N(q, p_k)$
6. repeat
7. (p,$d_E(q, p)$)=next_Euclidean_NN(q);
8. if ($d_N(q,p)<d_N(q,p_k)$) // p closer than the k^{th} NN
9. insert p in $\{p_1,...,p_k\}$ // remove ex-k^{th} NN
10. d_{Emax} = $d_N(q, p_k)$
11. until $d_E(q,p)>d_{Emax}$

End IER

Figure 4.2: Incremental Euclidean Restriction

4.2 Incremental Network Expansion

IER (and the Euclidean restriction framework in general) performs well if the ranking of the data points by their Euclidean distance is similar to that with respect to the network distance. Otherwise, a large number of Euclidean NNs may be inspected before the network NN is found. Figure 4.3 shows an example where the black points represent the nodes in the modeling graph and rectangles denote entities. The nearest entity to the query q (white point) is p_5. The subscripts of the entities (p_1,p_2,...,p_5) are in ascending order of their Euclidean distance to q. Since p_5 has the largest Euclidean distance, it will be examined after all other entities, i.e., p_1 to p_4 correspond to *false hits*, for which the network distance computations are redundant.

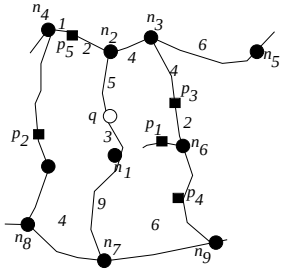


Figure 4.3: Finding the NN p_5

To remedy this problem, the Incremental Network Expansion (INE) algorithm performs network expansion (starting from q), and examines entities in the order they are encountered. Specifically, INE first locates the segment n_1n_2 that covers q, and retrieves all entities on n_1n_2 (using the primitive operation *find_entities*). Since no point is covered by n_1n_2, the node (n_1) closest to the query is expanded (while, the second endpoint n_2 of n_1n_2 is placed in a queue Q). No data point is found in n_1n_7 and n_7 is inserted to Q=<(n_2,5), (n_7,12)>. The expansion of n_2 reaches n_4 and n_3, after which Q=<(n_4,7), (n_3,9), (n_7,12)> and point p_5 is discovered on n_2n_4 (while no point is found on n_2n_3). The distance $d_N(q,p_5)$ =6 provides a bound d_{Nmax} to restrict the search space. The algorithm terminates now since the next entry n_4 in Q has larger distance (i.e., 7) than d_{Nmax}. Figure 4.4 shows the pseudocode of INE.

Algorithm INE (q, k)
1. n_in_j=*find_segment*(q)
2. S_{cover}=*find_entities*(n_in_j); // S_{cover} is the set of entities covered by n_in_j
3. $\{p_1,...,p_k\}$=the k (network) nearest entities in S_{cover} sorted in ascending order of their network distance (p_m, p_{m+1}...,p_k may be ∅ if S_{cover} contains < k points)
4. d_{Nmax}=$d_N(q,p_k)$ // if p_k=∅, d_{Nmax}=∞
5. Q = <(n_i, $d_N(q,n_i)$), (n_j, $d_N(q,n_j)$)> //sorted on d_N
6. de-queue the node n in Q with the smallest $d_N(q,n)$
7. while ($d_N(q,n)<d_{Nmax}$)
8. for each non-visited adjacent node n_x of n
9. S_{cover}=*find_entities*(n_xn);
10. update $\{p_1,...,p_k\}$ from $\{p_1,...,p_k\}\cup S_{cover}$
11. d_{Nmax}=$d_N(q,p_k)$
12. en-queue (n_x,$d_N(q,n_x)$)
13. de-queue the next node n in Q

End INE

Figure 4.4: Incremental Network Expansion

5. Range Queries in SNDB

Given a source point q, a value e and a spatial dataset S, a range query retrieves all objects of S that are within network distance e from q. Section 5 applies the Euclidean restriction and network expansion paradigms for processing such queries.

5.1 Range Euclidean Restriction

The Range Euclidean Restriction (RER) method first performs a range query at the entity dataset and returns the set of objects S' within (Euclidean) distance e from q. Assuming the Euclidean lower bound property, S' is guaranteed to avoid false misses (i.e., $d_N(q,p)\leq e \Rightarrow d_E(q,p)\leq e$), but it may contain a large number of false hits. In order to reduce the number of network distance computations, RER performs network expansion only once, examining all segments within network distance e from q. Points of S' that fall on some segment, are removed from S' and returned to the user. The process terminates when all the segments in the range are exhausted, or when S' becomes empty.

Figure 5.1 illustrates the pseudo-code of the algorithm. S' contains the results of the Euclidean range query sorted on some dimension. When a new segment is encountered, the

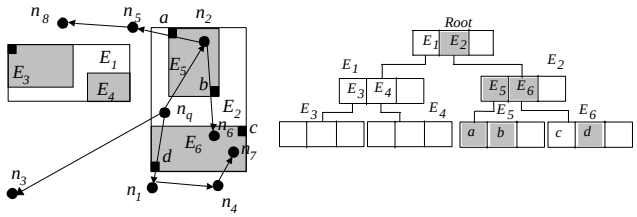
sorted list is used to efficiently check if any point falls inside its MBR (filter step). Such points are then compared with the poly-line representation of the segment to determine whether they belong to the actual result (refinement step). Part of some segments at the boundary may exceed the query threshold *e*, but these segments must be considered anyway since they may contain data points that satisfy the query.

Algorithm RER(*q, e*)
/* *q*: *query point, e*: *the network distance threshold* */
1. *result*=∅
2. *S'* = Euclidean-range(*q, e*)
3. n_in_j=*find_segment*(*q*)
4. Q=<(n_i,$d_N(q,n_i)$), (n_j,$d_N(q,n_j)$)>
5. de-queue the node *n* in *Q* with the smallest $d_N(q,n)$
6. while ($d_N(q,n)\leq e$ and $S' \neq \emptyset$)
7. for each non-visited adjacent node n_x of *n*
8. for each point *s* of *S'*
9. if *check_entity*(n_x,*n*,*s*)
10. *result*=*result*∪{*s*}; *S'* =*S'* −{*s*}
11. en-queue (n_x,$d_N(q,n_x)$)
12. de-queue the next node *n* in *Q*
13. end while
End RER

Figure 5.1: Range Euclidean Restriction

5.2 Range Network Expansion

The Range Network Expansion (RNE) algorithm first computes the set *QS* of qualifying segments within network range *e* from *q* and then retrieves the data entities falling on these segments. The methodology is similar to INE, but now numerous queries, one for each qualifying segment, are performed simultaneously (i.e., an intersection join as discussed in Section 2.1). To illustrate RNE, assume that *QS* contains the segments shown in Figure 5.2a. Starting from the root of the object R-tree, RNE visits nodes that intersect the MBR of at least one segment in *QS*. Figure 5.2b illustrates the visited nodes and the qualifying objects in gray.



(a) Network and objects (b) The object R-tree

Figure 5.2: Example of RNE

In order to avoid joining the entire *QS* (which may be large) with every entry, we perform the following optimization. *QS* is divided into (possibly overlapping) sets QS_i, one for each entry E_i in the current R-tree node. A segment is assigned to all entries that intersect its MBR. When the children of E_i are visited, they are only compared against QS_i. Thus, as RNE descends the tree, the number of comparisons performed for each entry drops. In Figure 5.2, the set of qualifying segments $QS_1 = \emptyset$, while for E_2, QS_2 consists of all segments except n_1n_4 and n_5n_8. Similarly, $QS_5 = \{n_qn_2, n_2n_5, n_2n_6\}$ and $QS_6 = \{n_qn_1, n_2n_6, n_4n_7\}$. When the node of E_5 (E_6) is visited, its points will only be checked against the segments of QS_5 (QS_6).

An object can be reported more than once if it lies at the intersections of the segments in *QS*. Such duplicates are easy to remove, by sorting the results at each leaf node before reporting them. Finally, RNE is I/O optimal (since it only accesses R-tree nodes that overlap some qualifying segment, and therefore, may contain results). The pseudo code of RNE is presented in Figure 5.3. The initial parameters of the algorithm are (root of R-tree *S*, *QS*, ∅). To reduce the number of intersection tests, at lines 2 and 7 we apply a plane sweep algorithm [APR+98].

Algorithm RNE(*node_id, QS, result*)
1. if (*node_id* is an intermediate node)
2. compute QS_i for each entry E_i in *node_id* // join
3. for each entry E_i in *node_id*
4. if ($QS_i \neq \emptyset$)
5. RNE(E_i.*node_id*, QS_i, *result*)
6. else // *node* is a leaf node
7. $result_{node_id}$ =plane-sweep(*node_id.entries*, QS_i)
8. sort $result_{node_id}$ to remove duplicates
9. *result* = *result*∪$result_{node_id}$
End RNE

Figure 5.3: Range Network Expansion

An alternative is to use the methodology suggested by [PRS99]. In particular, the MBR of all segments in *QS* is applied as a range query to the object R-tree. When a leaf node is reached, its contents are joined with *QS*, using plane-sweep. This technique performs a simple intersection test at each visited R-tree node; however, if the network range is large and irregular it may visit numerous tree nodes that do not overlap any qualifying segment (e.g., E_1 in Figure 5.2).

Finally, if *QS* does not fit in memory, the join is performed in a block nested loops fashion, i.e., RNE is repeatedly applied for subsets of *QS* that fit in memory and the partial results are materialized. Another approach is to compute all qualifying segments, materialize them and join them with the object R-tree using one of the spatial join algorithms that are applicable in the presence of a single tree [RSV02].

6. Closest-Pairs in SNDB

Given two datasets *S*, *T* and a value *k*, a closest-pairs query retrieves the *k* (≥1) pairs (*s*,*t*) $s \in S$, $t \in T$ that are closest in the network (e.g., find the hotel, restaurant pair within the smallest driving distance). This section describes retrieval of closest pairs in SNDB.

6.1 Closest-Pairs Euclidean Restriction

Like IER, the Closest-Pairs Euclidean Restriction (CPER) algorithm applies the multi-step *k*NN methodology. Assume for instance that only the closest pair is required. CPER performs an incremental closest-pairs query [CMTV00] on the R-trees of S, T and retrieves the Euclidean closest pair (s,t). The network distance $d_N(s,t)$ provides an upper bound d_{Emax} for all candidate pairs in the Euclidean space. Subsequent candidate pairs are retrieved incrementally, continuously updating the result and d_{Emax}, until no candidate pairs can be found within the d_{Emax} bound. The extension to k nearest neighbors is similar to that of IER. Figure 6.1 illustrates the pseudo-code of CPER algorithm.

Algorithm CPER (S,T, k)
/* S and T are two entity data sets; k is the number of closest pairs to be retrieved*/
1. $\{(s_1,t_1),...,(s_k,t_k)\}$=Euclidean_CP($S$,$T$,$k$);
   // find the k Euclidean closest pairs
2. for i=1 to k
3. $d_N(s_i,t_i)$=*compute_ND*(s_i,t_i)
4. sort (s_i,t_i) in ascending order of their $d_N(s_i,t_i)$
5. $d_{Emax}=d_N(s_k,t_k)$
6. repeat
7. (s',t') = next_Euclidean_CP(S,T)
8. $d_N(s',t')$=*compute_ND*(s',t')
9. if $(d_N(s',t')<d_{Emax}$
   // (s',t') is closer in the network than (s_k,t_k)
10. insert (s',t') in $\{(s_1,t_1),...,(s_k,t_k)\}$
11. $d_{Emax}=d_N(s_k,t_k)$
12. until $d_E(s',t')>d_{Emax}$

End CPER

Figure 6.1: Closest-Pairs Euclidean Restriction

6.2 Closest-Pairs Network Expansion

The difference between closest-pairs and the previous query types (range search and NN) is that now there does not exist a query point, which can be used as a source for network expansion. Thus, the only option is to use as sources all the data points of one dataset (the one with the smallest cardinality). The pseudo-code for Closest-Pairs Network Expansion (CPNE) algorithm is shown in Figure 6.2, assuming that the seeds for expansion are provided by S. CPNE calls INE (Section 4.2) to retrieve the k nearest neighbors $t_1,..,\ t_k$ $(\in T)$ of the first object s_1 of S. The distance $d_N(s_1,t_k)$ provides a d_{Nmax} bound for subsequent expansions. As closer pairs are discovered, this bound gradually decreases.

Obviously, in most cases CPNE is expected to be significantly more expensive than CPER. However, it is still useful in some extreme situations (e.g., large cardinality difference between the datasets, very high k). Furthermore, it is the only option if the lower bound property does not hold, in which case CPER is inapplicable. This issue will be discussed further in Section 9.

Algorithm CPNE (S,T,k)
1. $\{t_1, ...,t_k\}$=**INE**(s_1,k)
   // retrieve kNN $t_1,..,\ t_k$ of first entity s_1 in S
2. *result*$=\{(s_1,t_1), ..., (s_1,t_k)\}$
3. $d_{Nmax}=\max\{d_N(s_1,t_i)\}$ // current k^{th} CP distance
4. for each other point $s_i \in S$ $(s_i \neq s_1)$
5. n_in_j=*find_segment*(s_i)
6. T_{cover}=*find_entities*(n_in_j); //in T
7. for every entity t in T_{cover}
8. if $d_N(s_i,t)<d_{Nmax}$ then update *result* and d_{Nmax}
9. $Q =<(n_i,d_N(s_i,n_i)), (n_j,d_N(s_i,n_j))>$
10. de-queue the node n in Q with the smallest $d_N(s_i,n)$
11. while $(d_N(s_i,n)\leq d_{Nmax})$
12. for each non-visited adjacent node n_x of n for s_i
13. T_{cover}=*find_entities*(n_xn)
14. for each entity t in T_{cover}
15. if $d_N(s_i,t)<d_{Nmax}$ then update *result* and d_{Nmax}
16. en-queue $(n_x,d_N(s_i,n_x))$
17. de-queue the next node n in Q
18. end while

End CPNE

Figure 6.2: Closest-Pairs Network Expansion

7. *e*-Distance Joins in SNDB

Given two spatial datasets S, T and a value e, an e-distance join retrieves the pairs (s,t) $s \in S$, $t \in T$ such that $d_N(s,t)\leq e$ (e.g., find the hotel, restaurant pairs within 10km driving distance). Similar to the previous query types, we present algorithms in the Euclidean restriction and network expansion paradigms, respectively.

7.1 Join Euclidean Restriction

A straightforward way to process the e-distance join is to perform an *R-tree join* and find the set of all pairs within Euclidean distance e. Then, for each pair we compute the network distance, filtering out the false hits. The overhead of false hits is especially serious in this case, due to the large output size of the Euclidean join. In order to illustrate how the situation can be improved, consider that the result of *R-tree join* contains six pairs: $(s_1,\ t_1)$, $(s_1,\ t_2)$, $(s_1,\ t_3)$, $(s_2,\ t_1)$, $(s_2,\ t_4)$, $(s_2,\ t_5)$ requiring six network distance computations. On the other hand, since there are only two objects s_1 and s_2 from the first dataset, the actual result may be obtained by expanding only these points.

Based on this observation, the Join Euclidean Restriction (JER) algorithm first applies *R-tree join* and counts the number of distinct points from each dataset that appear in the output. The dataset with the smaller count is used to provide the "seeds" for node expansion. The pseudo-code of the algorithm is shown in Figure 7.1, assuming that the dataset with the smaller number of distinct objects in the result is S. For each such object, the network around it is expanded and the set QS of segments within range e are retrieved. Then, every object $t \in T$ that appears with s in some (Euclidean) join pair is tested (using the primitive operation *check_entity*); if t falls on some segment of QS,

then the pair (s,t) is added to the final result. In order to facilitate fast computation of T_s, and at the same time achieve spatial locality between consecutive expansions, the output of *R-tree join* is sorted on S using the Hilbert space filling curve. For large *Rjoin-res*, the algorithm is repeatedly applied for blocks of pairs that fit in memory.

Algorithm JER (S,T,e)
/* S and T are two entity data sets; *e is the (network) distance threshold* */
1. *Rjoin-res*=R-tree-join(S, T);
2. sort *Rjoin-res* on s
3. for each distinct object $s \in S$ in *Rjoin-res*
4. T_s= set of objects $\in T$ that pair with s in *Rjoin-res*
5. *QS*=*expand_point*(s,e)
6. for each object $t \in T_s$
7. for each segment *seg* in *QS*
8. if *check_entity(seg,t)*
9. *result*= *result*$\cup$(s,t)

End JER

Figure 7.1: Join Euclidean Restriction

7.2 Join Network Expansion

The Join Network Expansion (JNE) algorithm expands the network around points of the smallest dataset (let it be S) to find the matching objects of the second dataset (T). Obviously, as in the case of CPNE, this approach is expected to be very expensive in most situations. In order to reduce the cost, JNE exploits access locality. In particular, the network is expanded around s_1,.., s_n (n depends on the available memory) neighboring points of S, producing corresponding sets of qualifying segments QSs_1,.., QSs_n. Then, the RNE algorithm (Section 5.2) is applied (on the R-tree of T) for all QSs_1,..,QSs_n simultaneously. Every point $t \in T$ that falls on a segment of QSs_i appends a new pair (s_i,t) in the result. The advantage of this approach (with respect to straightforward network expansion) is that it saves disk accesses for segments that appear in multiple QSs_i (which, otherwise, would generate multiple query windows on T). In order to achieve locality of points s_1,.., s_n, we utilize the R-tree structure, i.e., the points are obtained from the same or neighboring leaf nodes in the R-tree of S.

Algorithm JNE (S,T,e)
1. *result*=$\varnothing$
2. while S has not been exhausted
3. get next s_1,.., s_n points
4. for each point s_i
5. QSs_i=*expand_point*(s_i,e)
6. let *QS*=the union of QSs_i (for $1 \le i \le n$)
7. **RNE**(*rootRtree*$_T$, *QS*, *result*)
8. end while

End JNE

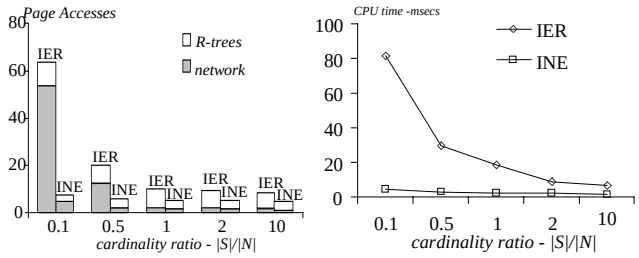
Figure 7.2: Join Network Expansion

8. Experimental Evaluation

In this section we experimentally compare all algorithms in terms of CPU-time and I/O cost using a Pentium III, 700MHz processor, running Windows NT. We set the page size of the data structures to 4K and employ an LRU buffer which accommodates 10% of the road network and 10% of each R-tree participating in an experiment. In order to simulate real-life conditions, we use a spatial network of $|N|$ = 179,000 segments, representing main roads in North America [WWW], "cleaned" to form a connected graph. For simplicity, we consider bidirectional edges; however, this does not affect the interpretability and value of the results. In order to control the density of the entities, we use synthetic datasets with cardinalities in the range $0.01 \cdot |N|$ to $10 \cdot |N|$. The distribution of the entities follows the network distribution. For nearest neighbor and range search, we execute workloads of 200 queries, also following the network distribution.

8.1 Nearest Neighbor Queries

First we compare IER (incremental Euclidean restriction) with INE (incremental network expansion). Figures 8.1a and 8.1b plot the performance of the two methods in terms of I/O accesses and CPU cost, as a function of $|S|/|N|$ (i.e., the ratio of entity to segment cardinality), for k=10. The I/O cost is broken to R-tree and network page accesses.

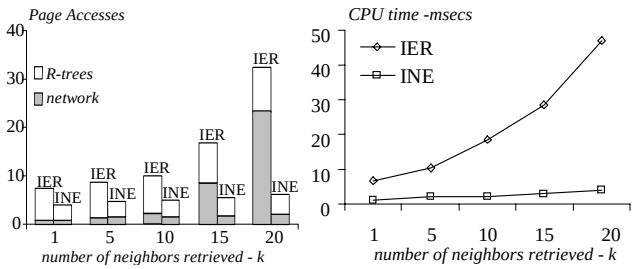


(a) I/O accesses (b) CPU (msec)

Figure 8.1: Cost vs. $|S|/|N|$ (k=10)

When the cardinality $|S|$ of the entity set is small, the Euclidean nearest neighbors are far from the query point. As we see later this increases significantly the number of false hits, and therefore, the unnecessary network distance computations. The problem lessens as $|S|$ increases, and the performance of IER improves. On the other hand, the I/O cost of INE is low because the range queries on the R-tree exhibit high locality and the search path is in the buffer with high probability. Moreover, only the necessary network edges are visited (as ensured by the algorithm).

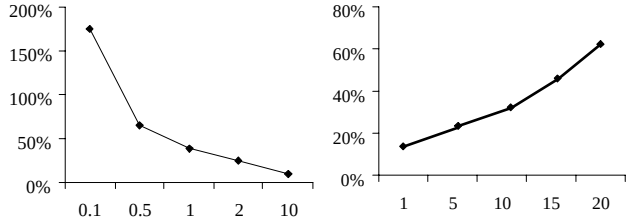
Figure 8.2 shows the performance of the two methods for various values of k, when $|S|=|N|$. INE consistently outperforms IER and the gap increases with k. The explanation is similar to the previous case, i.e., a large k implies a high distance from the query point and, therefore, increases the number of false hits.



(a) I/O accesses (b) CPU (msec)

Figure 8.2: Cost vs. k ($|S|=|N|$)

Figure 8.3 unveils the ratio of false hits retrieved by IER (i.e., the number of Euclidean NN that are not in the query result divided by k) for the two experiments of Figures 8.1 and 8.2. The false hit ratio drops with the cardinality of S, since the entity set becomes denser and increases the probability to find all nearest neighbors on the edges adjacent to the query point or in its vicinity. On the other hand, the false hit ratio increases with k. The effects of false hits are reflected to the processing cost of IER.



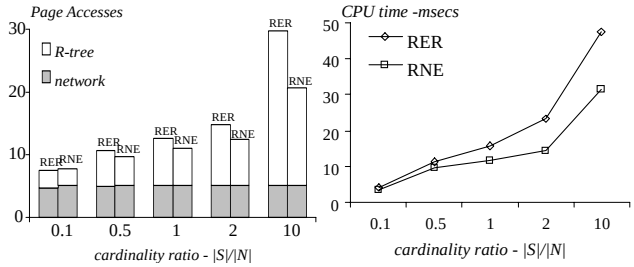
(a) FH ratio vs. $|S|/|N|$ (k=10) (b) FH ratio vs. k ($|S|=|N|$)

Figure 8.3: False hits by IER

Concluding, INE is more efficient and robust than IER, which suffers by the excessive network distance computations due to false hits. Nonetheless IER could perform better in denser, more regular networks (e.g., city blocks), where the Euclidean distance gives a better approximation of the travel cost. Furthermore, its cost will drop significantly if materialization is used (so that network distances can be computed very efficiently).

8.2 Range queries

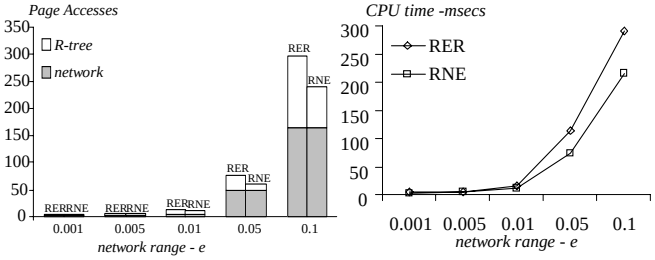
The next set of experiments compares RER (range Euclidean restriction) with RNE (range network expansion). Figure 8.4 shows the cost of the algorithms as a function of $|S|/|N|$, fixing the query range e to 0.01 (1% of the data universe side length). Both algorithms perform a single expansion of the network. Their difference is that (i) RER first retrieves the candidate objects within the Euclidean range e and then expands the network, while (ii) RNE first expands and then performs the query on the data R-tree for the actual results. This explains the fact that the algorithms have the same network cost in all cases. On the other hand, RER also retrieves some false hits (i.e., objects in the Euclidean, but not in the network range), which result in more R-tree node accesses. Although, as shown in Figure 8.6a, the false hit ratio is almost constant, the absolute number of false hits increases with $|S|$, which is reflected in the increasing cost difference of the algorithms as the cardinality of the entity set grows.



(a) I/O accesses (b) CPU (ms)

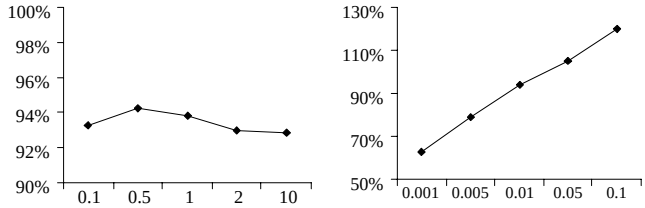
Figure 8.4: Cost vs. $|S|/|N|$ (e=0.01)

Figure 8.5 compares the performance of the methods as a function of e, when $|S|=|N|$. The number of retrieved objects (and the cost of the algorithms) increases proportionally to the area covered by the range, i.e., quadratically with e. As shown in Figure 8.6b the false hits ratio of RER increases linearly with e. Consequently, the relative R-tree cost difference of the algorithms grows faster with e than with $|S|$. Summarizing, RNE is more efficient than RER in the current problem settings, due to the fact that it retrieves only the required R-tree nodes.



(a) I/O accesses (b) CPU (ms)

Figure 8.5: cost vs. e ($|S|=|N|$)

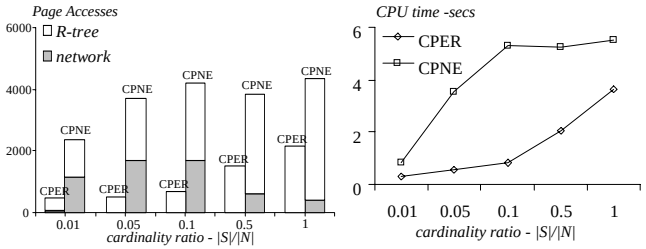


(a) FH ratio vs. $|S|/|N|$ (e=0.01) (b) FH ratio vs. e ($|S|=|N|$)

Figure 8.6: False hits by RER

8.3 Closest pairs

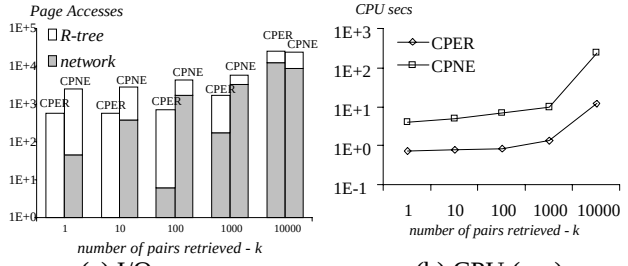
In this section, we compare CPER (closest-pairs Euclidean restriction) with CPNE (closest-pairs network expansion). First, we fix k=100, $|T|$=0.1$|N|$ and vary the cardinality of S. Figure 8.7 plots the costs of the algorithms as a function of $|S|$. CPER outperforms CPNE in all cases, because CPNE expands the network around all points of the smallest dataset, while CPER only expands it incrementally around the Euclidean closest pairs. Note that the I/O cost of CPNE remains almost constant for $|S| \geq 0.1|N|$, because after $|S|$ reaches 0.1$|N|$, the entities of T ($|T| = 0.1|N|$) are used for expansion (i.e., the number of expansions is independent of $|S|$).



(a) I/O accesses (b) CPU (sec)

Figure 8.7: Cost vs. $|S|$ (k=100, $|T|$=0.1$|N|$)

Figure 8.8 shows the relative costs of the algorithms when $|S|=|T|=0.1|N|$ for different values of k. CPER is much faster than CPNE for $k \leq 1000$ for the reasons explained above. For k=10000 the cost of both algorithms explodes for different reasons (Note that the diagrams are in logarithmic scale). CPER now incurs numerous false hits, since there is a huge number of object-pairs with similar Euclidean distances, but diverge network distances. These pairs require many expensive distance computations of long paths, which incur extensive buffer thrashing. CPNE performs extensive expansion, which exceeds the available memory and causes many swaps in the buffer. Summarizing, CPER is much faster than CPNE in our settings, because it can utilize the Euclidean bounds to prune large areas of the search space early.



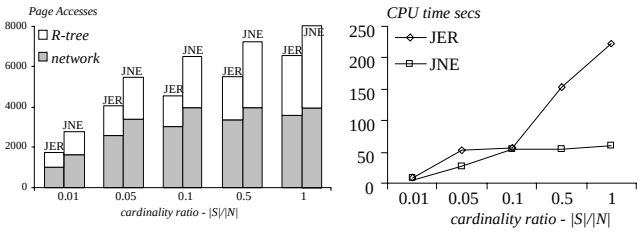
(a) I/O accesses (b) CPU (sec)

Figure 8.8: Cost vs. k ($|S|=|T|=0.1|N|$)

8.4 *e*-Distance joins

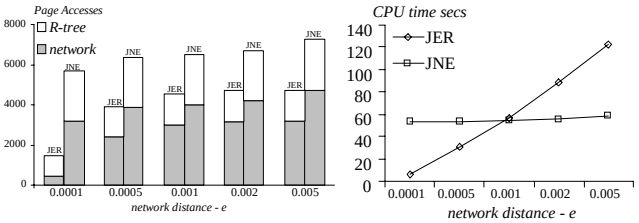
We proceed to compare JER (join Euclidean restriction) and JNE (join network expansion), using $|T|=0.1|N|$ and setting the join distance e to 0.001. Figure 8.9a (8.9b) plots the number of disk accesses (CPU time) as a function of $|S|$ ranging from $0.01|N|$ to $|N|$. JER has better I/O performance, but the difference diminishes as $|S|$ increases. This is because, for large datasets, the number of object pairs qualifying the Euclidean distance join increases considerably, making the subsequent node-expansion (for false hit elimination) expensive. In this case, JER consumes more CPU time, due to the expensive sorting overhead (for selecting the "seed" for node expansion).

In Figure 8.10a (8.10b), we set $|S|$ to $0.1|N|$ and measure the number of disk accesses (CPU cost) for different values of e. JER is significantly faster in terms of I/O, especially for small join distance in which case very few object pairs satisfy the Euclidean join. Interestingly, the relative CPU performance of JER and JNE changes at e=0.001. Particularly, the cost of JNE is almost independent of e, while JER incurs high CPU cost for large e because, similar to Figure 8.9b, its sorting step needs to process a large number of object pairs (that pass the Euclidean join). Therefore, JER is preferred for selective joins, while JNE should be applied otherwise.



(a) I/O accesses (b) CPU (sec)

Figure 8.9: Cost vs. $|S|$ (e=0.001, $|T|$=0.1$|N|$)



(a) I/O accesses (b) CPU (sec)

Figure 8.10: Cost vs. e ($|S|=|T|=0.1|N|$)

9. Conclusion

This paper presents the first comprehensive approach for query processing in spatial network databases, proposing an architecture that preserves connectivity and location, and several novel algorithms, based on the Euclidean restriction and network expansion frameworks, covering the most common processing tasks.

The Euclidean Restriction framework provides an intuitive way to deal with spatial constraints. If for instance, we want to "find the two nearest hotels to the south", we only need to retrieve the Euclidean neighbors in the area of interest using a constrained NN algorithm [FSA+01]. On the other hand, although network expansion is still applicable, it has limited pruning power on queries with selective spatial conditions. Considering again the example query, the network should be also expanded to the north of the query point, because subsequent nodes may lead to a nearest neighbor to the south.

The Euclidean Restriction framework assumes the lower bounding property, which may not always hold in practice. If, for instance, the edge cost is defined as the expected travel time, the Euclidean distance cannot confine the search space (unless we make additional assumptions, such as maximum speed). On the contrary, network expansion permits a wide variety of costs associated with the edges. It assumes, however, that the cost increases monotonically with the path (i.e., a path

cannot be cheaper than one of its sub-sets), because, otherwise there is no bound in the expansion process. Dijkstra's algorithm is also based on the same assumption, which is realistic for all SNDB applications.

The experimental evaluation suggests that the network expansion framework has superior performance for range search and nearest neighbors, while Euclidean restriction is better for closest pairs and joins. Since, however, our goal was to propose a complete set of algorithms for numerous queries, we did not focus explicitly on optimization of each method. Therefore, further improvements are possible for the proposed algorithms. It will also be interesting to evaluate their relative performance in the presence of materialized network distances.

This paper opens a door to several interesting and practical directions for future work. For instance, a continuous NN query [TPS02] would retrieve the two nearest gas stations (in terms of network distance) during the route from city A to city B. Our framework can also be used in the context of moving object databases to answer: "which is the closest taxi to my present location", "towards which direction should I walk to catch the next (moving) bus", etc. Since most real life objects move on pre-defined spatial networks, the SNDB versions of these queries are much more important than their Euclidean counterparts.

Acknowledgements

This work was supported by grants HKUST 6197/02E, HKUST 6081/01E and HKU 7380/02E from Hong Kong RGC. We would like Qiongmao Shen and Manli Zhu for helping with the implementation.

References

[ADJ90] Agrawal, R., Dar, S., Jagadish, H. Direct Transitive Closure Algorithms: Design and Performance Evaluation. *TODS*, 15(3), 427-458,1990.

[APR+98] Arge, L., Procopiuc, O, Ramaswamy, S., Suel, T., Vitter, J. S. Scalable Sweeping-Based Spatial Join. *VLDB*, 1998.

[BKS+90] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[BKS93] Brinkhoff, T., Kriegel, H., Seeger, B. Efficient Processing of Spatial Joins Using R-trees. *SIGMOD*, 1993.

[CLR90] Corman, T. H., Leiserson, C. E., Riverst. R. L. Introduction to Algorithms. *MIT Press*, 1990.

[CMTV00] Corral, A., Manolopoulos, Y., Theodoridis, Y., Vassilakopoulos, M. Closest Pair Queries in Spatial Databases. *SIGMOD*, 2000.

[D59] Dijkstra, E. W. A Note on Two Problems in Connection with Graphs. *Numeriche Mathematik*, Vol. 1, 269-271, 1959.

[FRM94] Faloutsos, C., Ranganathan, M., Manolopoulos, Y. Fast Subsequence Matching in Time-Series Databases. *SIGMOD*, 1994.

[FSA+01] Ferhatosmanoglu, H., Stanoi, I., Agrawal, D., El Abbadi, A. Constrained Nearest Neighbor Queries. *SSTD*, 2001.

[G84] Guttman, A. R-trees: A Dynamic Index Structure for Spatial Searching, *SIGMOD*, 1984.

[HJR96] Huang, Y., Jing, N., Rundensteiner, E. Effective Graph Clustering for Path Queries in Digital Map Databases. *CIKM*, 1996.

[HJR97] Huang, Y., Jing, N., Rundensteiner, E. Integrated Query Processing Strategies for Spatial Path Queries. *ICDE*, 1997.

[HS99] Hjaltason, G., Samet, H. Distance Browsing in Spatial Databases. *TODS*, 24(2), 265-318, 1999.

[IRW93] Ioannidis, Y., Ramakrishnan, R., Winger, L. Transitive Closure Algorithms Based on Graph Traversal. *TODS*, 18(3), 512-576, 1993.

[J92] Jiang, B. I/O-Efficiency of Shortest Path Algorithms: An Analysis. *ICDE*, 1992.

[JHR98] Jing, N., Huang, Y. W., Rundensteiner, E. Hierarchical Encoded Path Views for Path Query Processing: an Optimal Model and its Performance Evaluation. *TKDE*, 19(1), 102-119, 1997.

[JP96] Jung, S., Pramanik, S. HiTi Graph Model of Topographical Roadmaps in Navigation Systems. *ICDE*, 1996.

[KHI+86] Kung, R., Hanson, E., Ioannidis, Y., Sellis, T., Shapiro, L., Stonebraker, M. Heuristic Search in Data Base Systems. *Expert Database Systems*, 1986.

[PRS99] Papadopoulos A., Rigaux P., Scholl M. A Performance Evaluation of Spatial Join Processing Strategies. *SSD*, 1999.

[RSV02] Rigaux, P. Scholl, M. Voisard, A. *Spatial Databases with Application to GIS*. Morgan Kaufmann, 2002.

[SK98] Seidl, T., Kriegel, H. Optimal Multi-Step k-Nearest Neighbor Search. *SIGMOD*, 1998.

[SKC93] Shekhar, S., Kohli, A., Coyle, M. Path Computation Algorithms for Advanced Traveler Information System (ATIS). *ICDE*, 1993.

[SKS02] Shahabi, C., Kolahdouzan, M., Sharifzadeh, M. A Road Network Embedding Technique for K-Nearest Neighbor Search in Moving Object Databases. *ACM GIS*, 2002.

[SL97] Shekhar, S., Liu, D. CCAM: A Connectivity-Clustered Access Method for Networks and Network Computations. *TKDE*, 19(1), 102-119, 1997.

[SRF87] Sellis, T., Roussopoulos, N. Faloutsos, C.: The R+-tree: a Dynamic Index for Multi-Dimensional Objects, *VLDB*, 1987.

[TPS02] Tao, T., Papadias, D., Shen, Q. Continuous Nearest Search. *VLDB*, 2002.

[WWW] *www.maproom.psu.edu/dcw/*