Distance Join Queries on Spatial Networks

Jagan Sankaranarayanan, Houman Alborzi, and Hanan Samet*
Center for Automation Research
Institute for Advanced Computer Studies
Department of Computer Science
University of Maryland, College Park, MD 20742
{jagan,houman,hjs}@cs.umd.edu

ABSTRACT

The result of a distance join operation on two sets of objects R, S on a spatial network G is a set P of object pairs $<p, q>$, $p \in R$, $q \in S$ such that the distance of an object pair $<p, q>$ is the shortest distance from p to q in G. Several variations to the distance join operation such as UNORDERED, INCREMENTAL, TOP-K, SEMI-JOIN impose additional constraints on the distance between the object pairs in P, the ordering of object pairs in P, and on the cardinality of P. A distance join algorithm on spatial networks is proposed that works in conjunction with the SILC framework, which is a new approach to query processing on spatial networks. Experimental results demonstrate up to an order of magnitude speed up when compared with a prominent existing technique.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Spatial Databases and GIS*; E.1 [**Data Structures**]: Graphs and Networks; H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms

Algorithms, Performance, Design

Keywords

Location-based services, Spatial networks, SILC framework, Query processing, Path coherence, Spatial databases

1. INTRODUCTION

The distance join operation computes a subset of the Cartesian product $R \times S$ of two sets R and S of a specified order and is based on the distance [10]. The result of a distance join operation is a set P of ordered pairs of objects $<p, q>$, such that $p \in R$ and $q \in S$. In this paper, we propose distance join operations on a spatial network G, where R and S are sets of objects in G and the distance of an object pair $<p, q>$ in P is the shortest distance $d_N(p, q)$ from p to q in G. Additionally, spatial and non-spatial constraints could be further imposed either on P, on the object pairs in P, or on both. This results in many different variants of the distance join operation, some of which are discussed below.

The first variant deals with the order in which pairs of objects in P are reported. The result of an ORDERED distance join operation is a set P of object pairs that is obtained and reported in an increasing (or decreasing) order of the distance between the pairs *i.e.*, the first pair in P is the closest object pair in $R \times S$, while the last element in P is the farthest object pair in $R \times S$. A distance join operation is said to be UNORDERED if an ordering of P is not specified.

Distance join operations may generate a very large number of object pairs even for sets of objects R, S of a modest size. For example, R and S both containing $50,000$ objects may generate up to 2.5 billion object pairs. However, in practice, we may be only interested in a small number of pairs in the result set. Hence, computing all the possible pairs in $R \times S$ may result in wasted work. The TOP-K distance join operation computes the first k pairs of objects in P of a distance join operation. Note that the TOP-K constraint implicitly assumes that a distance ordering of the objects in P is specified. Hence, we slightly modify the effect of the TOP-K constraint on ORDERED and UNORDERED distance join operations. An UNORDERED TOP-K distance join operation computes the "TOP-K" object pairs of $R \times S$, although the object pairs in P are still unordered, *i.e.*, we do not establish a total ordering of the k object pairs in P. In contrast, the "TOP-K" constraint applied to an ORDERED distance join operation results in an ordered set of object pairs in P containing k object pairs. Such a join operation has interesting applications to GIS. For example, given a set of locations corresponding to exits R on a highway and a set of restaurants S, we may wish to determine the k closest pairs containing an exit on the highway and a restaurant. Incidentally, such a query is a variant of the "in-route" query proposed in [17]. Figure 1c illustrates the top-10 pairs of an ORDERED distance operation on a road network of Washington, DC.

Instead of limiting the cardinality of the result set P using a TOP-K constraint, the distance join operation may be constrained to report only those object pairs that are within a specified minimum d^- and maximum d^+ distance values

*The support of the National Science Foundation under Grants EIA-00-91474 and CCF-05-15241, and Microsoft Research is gratefully acknowledged.

ACM-GIS'06, November 10–11, 2006, Arlington, Virginia, USA.

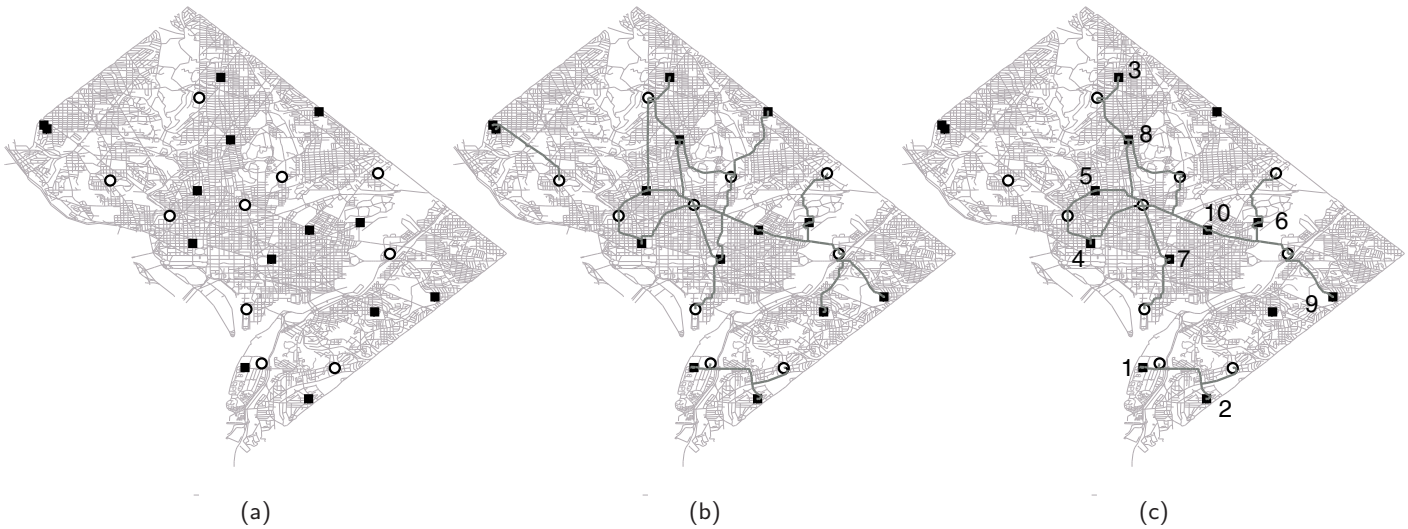

Figure 1: Example of a distance join operation on a road network of Washington, DC. a) Objects in R are shown using square icons, and objects in S are shown using circular icons. A subset of the result of a distance join operation, such that b) object pairs at a distance of less than 2.5 miles, and c) the top 10 object pairs in the result.

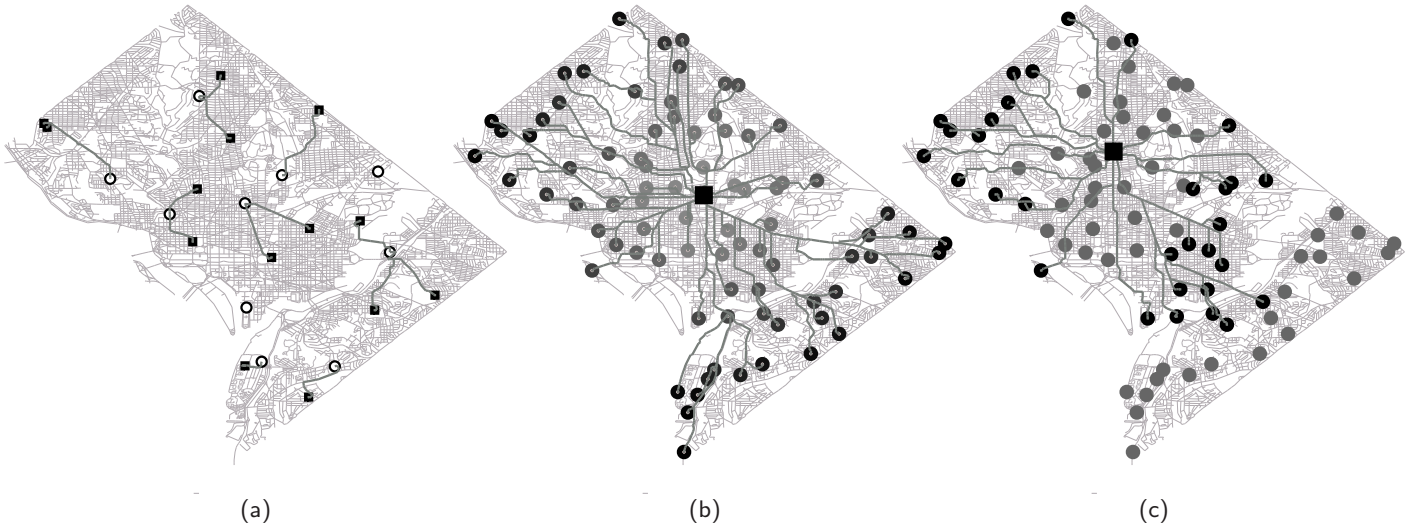

Figure 2: a) A subset of the result of a distance semi-join operation on the sets of objects R and S shown in Figure 1a. When R is an object, b) an ORDERED distance operation is an incremental nearest neighbor search on S. c) an UNORDERED distance join with a distance restriction is a range search on S.

i.e., $\forall <p,q> \in P$, such that $d^- \leq d_N(p,q) \leq d^+$. For example, given a set of locations R on a road network corresponding to where employees of a chain store reside, and the set of stores S, we can find the set of stores that each employee can reach within ϵ distance, or alternatively within ϵ time when the time taken to travel an edges of the road network is provided. Figure 1b is an illustration of a distance join operation when the minimum and maximum distance between the object pairs is specified.

An interesting variant of the distance join is the distance SEMI-JOIN, which restricts the number of occurrences of any object $p \in R$ in P. The result of a distance SEMI-JOIN operation computes a set P of object pairs, such that for all $p \in R$, there exists exactly one pair $<p,q_i>, q_i \in S$ in P. The result of an ORDERED distance semi-join operation pairs is an ordered set of object pairs, such that each object in R is paired with its closest object in S. Incidentally, the result of an ORDERED distance semi-join is equivalent to the ANN join [5]. For example, given a set of stores R and a set of warehouses S on a spatial network, the ORDERED distance semi-join associates each store with its closest warehouse. Figure 2a is an illustration of a distance SEMI-JOIN operation on a road network.

Incidentally, an ORDERED distance join operation can also be INCREMENTAL – that is, each invocation of the join only computes the next object pair in the result set. For simplicity sake, we assume that all ORDERED distance join operations are also INCREMENTAL, although this need not always be the case.

In this paper, we present an algorithm that can perform a variety of distance join operations on spatial networks. The inputs to the algorithm determine the nature of the distance join, *i.e.*, the user can choose between ORDERED versus UNORDERED, JOIN versus SEMI-JOIN, and distance pruning versus TOP-K. However, it should be clear that not all combinations of these operations yield meaningful results.

The rest of the paper is organized as follows. Section 2 discusses related work. A brief introduction to the SILC framework is provided in Section 3. Our join algorithm is presented in Section 4. Experimental results are discussed in Section 5, while concluding remarks are drawn in Section 6.

2. RELATED WORK

Performing spatial queries on *transportation networks*, also known as spatial networks, is an application that is of great interest to the GIS community [15, 16, 17]. Transportation networks form an integral part of GIS applications like *location-based services* and *locational analysis*. Moving object databases [16, 20] and trip-planning [17] are closely related to location-based services. The join algorithm proposed in this paper is based on the SILC framework [15] which is a new approach to query processing on spatial networks.

In our earlier work [15], we proposed an incremental nearest neighbor algorithm using the SILC framework. Given a query object q and a set of objects P on a spatial network, our algorithm incrementally obtains the nearest neighbors to q from objects in P. On the other hand, the related IER and INE techniques of Papadias *et al.* [13], and the subsequent improvements to the INE technique by Cho and Chung [4], use a technique that is based on Dijkstra's shortest path algorithm to find the k-nearest neighbors to a query object q on a spatial network. However, their methods are not incremental. Kolahdouzan and Shahabi [12] speed-up the process of finding the k-nearest neighbors by precomputing the Voronoi diagram for a set of locations P on a spatial network from which the k neighbors to q are drawn. The *landmark* approach is another method which is used by Jing *et al.* [11], and Goldberg and Harrelson [7] to speed up Dijkstra's algorithm.

There are two key differences between our approach which is based on the SILC framework and the above approaches. First, our technique requires precomputation of the shortest path between every pair of vertices in a spatial network. Although this can be quite expensive for large spatial networks, it can be achieved with a sufficient investment of time and hardware resources. Our method enables storing the shortest paths compactly and makes provision for retrieving them efficiently. A compact representation of the shortest paths of a spatial network G is termed the *path encoding* of G.

The second difference is that our method has an advantage when multiple datasets share the path encoding to perform queries on spatial networks. For example, a path encoding of Manhattan, NY can potentially be shared by several datasets of landmarks pertaining to postal addresses in Manhattan for query processing. Moreover, spatial networks are usually static structures, while datasets of objects may be updated frequently. For example, when dealing with a set of mobile hosts on a road network, the current positions of the objects are frequently updated, while the road network in itself would largely remain static. Moreover, the datasets and the network can be updated independently of each other. In effect, what we have done is to *decouple* the *data* from the *underlying domain*. In this respect, our work is distinguished from the work of Papadias *et al.* [13] and Kolahdouzan *et al.* [12] who perform path and distance queries at run-time, while making no provisions to reuse such computations across queries and across datasets. For example, the network Voronoi diagram in [12] computed for a set of restaurants in Manhattan cannot be used for another dataset of post offices in Manhattan.

In this paper, we introduce several distance join operations on a spatial network. Our algorithm is based on earlier work in [2, 10, 14, 18] which has now been applied to spatial networks. The "TOP-K" operator is based on the work of Carey and Kossmann in [3]. The distance semi-join operation of R and S, computes the first nearest neighbor to each object in R with neighbors drawn from S. Hence, it is related to the all nearest neighbor join (ANN join) query in [5, 19], although both [5, 19] deal with the problem in an Euclidean space. Moreover, as the first object pair in the result of an incremental distance join is the closest (or farthest) object pair drawn from $R \times S$, our work is also related to the "closest pair" queries introduced in [1].

The only techniques that directly compete with our work are the Join Euclidean Restriction (JER) and Join Network Expansion (JNE) techniques of Papadias *et al.* [13], which are both based on Dijkstra's algorithm. Given two sets of objects R and S on a spatial network and a distance ϵ, the JER and JRE algorithms compute pairs of object pairs $<p, q>$, $p \in S$, $q \in R$, such that the network distance between p and q is less than or equal to ϵ. Note however that both these algorithms are limited in their capabilities. Neither JER nor JRE can compute distance joins incrementally. Moreover, neither JER nor JRE can perform SEMI-JOIN operations. Finally, a distance join operation that specifies both a lower and an upper limit on distance between the object pairs that are reported cannot be handled efficiently by either the JER or JRE techniques. To perform such a query using JER or JRE would require obtaining all the object pairs that satisfy the upper bound on the distance between the object pairs and then pruning the result against the lower bound distance.

3. SILC FRAMEWORK

A spatial network can be abstracted to form an equivalent graph representation $G = (V, E)$, where V is the set of vertices, E is the set of edges, $n = |V|$, and $m = |E|$. Given $e \in E$, $w(e)$ denotes the distance along that edge. In addition, for every $v \in V$, $p(v)$ denotes the spatial position of v with respect to a reference coordinate system. We define the *spatial distance* between u and v, $d_S(u, v)$, as the shortest distance from $p(u)$ to $p(v)$ in the embedding space. For vertices $u, v \in V$, we define $d_N(u, v)$ to be the shortest distance from u to v with respect to the network $G(V, E)$. We use $\pi_N(u, v) \subset V$ to denote the shortest path from u to v. Note that $|\pi(u, v)|$ denotes the number of vertices in the shortest path from u to v. We also define $l_u(v)$ to be the next vertex visited (after u) on the shortest path from u to v.

The SILC framework precomputes the shortest path between all pairs of vertices in a spatial network. For each vertex $s \in V$, we first compute the next link in the shortest path from s to all other vertices in G. Then a region

Figure 3: Example illustrating the coloring process of vertices for Silver Spring, MD. a) Sample vertex u highlighted denoted by "X". b) Remaining vertices are assigned colors based on their shortest path to u through one of the six adjacent vertices of u. c) Morton blocks corresponding to the colored regions in (b).

quadtree, termed the SILC map of s is built on the spatial position of vertices, such that all vertices contained in a block have the same first link in the shortest path from s. We represent the blocks in the region quadtree as a set of Morton blocks [6] and store them on disk. Figure 3 illustrates the process of computing the SILC map of a vertex X in a road network of Silver Spring, Maryland.

Moreover, along with each block B in the SILC map of s, we also record the minimum (λ^-) and the maximum (λ^+) ratio of the difference between the network and the spatial distance between s and all vertices contained in B to the spatial distance between s and all vertices in B. Using the two stored values λ^-, λ^+, we are able to quickly compute the distances between two objects and between an object and a region on a spatial network, although now the distances are obtained as intervals. Distance $d = (\delta^-, \delta^+)$ has a lower bound δ^- and an upper bound δ^+.

Given two objects u and v on a spatial network, $d_I(u, v)$ is an interval $[\delta^-, \delta^+]$, such that $\delta^- \leq d_N(u, v) \leq \delta^+$. Furthermore, the distance interval between u and v can be improved using the REFINE operator, which improves the interval by expending work. A distance interval can be refined at most $|\pi_N(u, v)|$ times, after which $\delta^- = \delta^+ = d_N(u, v)$.

The network distance interval $d_I(u, R)$ between an object v and a block R is a distance interval $[\delta^-, \delta^+]$ that CONTAINS the network distance from v to every object contained in R. This corresponds to the MINDIST (MAXDIST) operators commonly used in spatial query processing.

One of the drawbacks of the SILC framework is that the network distance interval between a block pair $<b_1, b_2>$, or pair $<b, o>$ containing a block b and an object o, cannot be easily computed. We remedy the first problem by assuming that the spatial distance between two vertices in a spatial network is always a lower bound on the network distance between them. Thus, we approximate the network distance $d_I(b_1, b_2)$ between the block pair $<b_1, b_2>$ with the spatial distance $d_S(b_1, b_2)$ between them, which is cheaper to compute. We will avoid generating pairs of the second kind in our algorithm, thus circumventing the problem.

4. SILC DISTANCE JOIN

In this section, we describe our distance join algorithm. We first assume that the SILC encoding of a spatial network has been precomputed and stored on disk. Furthermore, we assume that we are provided with two sets of objects S and R on the spatial network. For the sake of simplicity, we assume that each object in R and S is associated with a vertex in the spatial network, although the algorithm can be easily modified to allow objects to be associated with the edges of the spatial network as in [13]. Moreover, we also assume that hierarchical disk-based data structures (*e.g.*, PMR quadtree [14] or R-tree [8]) over the spatial positions of objects in both R and S is available to the algorithm.

Algorithm 1
Procedure DISTJOIN[T, U, d^+, d^-, k, OPTION]
Input: OPTION can be ORDERED or UNORDERED, JOIN or SEMI-JOIN, TOP-K or distance restriction
Input: $T, U \leftarrow$root node of a spatial structure on R and S
Input: d^-, $d^+$are distance restrictions on the pairs, initially set to $-\infty$ and ∞, respectively.
Input: $S_o \leftarrow$set of objects o_i whose pairs $< o_i, o_j >$ have already been reported. It is initially empty and used for a SEMI-JOIN operation. If OPTION $\neq$ SEMI-JOIN, then the condition $p \notin S_0$ in lines 36 and 40 is always *true*
Input: $k \leftarrow$number of object pairs in result, default value 1
($* d_k \leftarrow \infty$, distance estimate to the k^{th} object pair $*$)
Output: P: pairs of object $<p, q>$, $p \in R$ and $q \in S$
1. INIT: $[\delta^-, \delta^+] \leftarrow d_I(T, U)$
2. **if** (OPTION = ORDERED) **then**
3. $Q \leftarrow$is a priority queue on keyed tuples
4. **else**
5. $Q \leftarrow$is a list
6. **end-if**
7. Q.insert([KEY=δ^-], $T, U, [\delta^-, \delta^+]$)
8. END-INIT
9. **while** not (Q.empty()) **do**
10. $(p, q, [\delta^-, \delta^+]) \leftarrow Q$.pop()
11. **if** (OPTION = TOP-K) **then**
12. **if** ($\delta^- > d_k$) **then**
13. continue **while-loop**
14. **end-if**
15. Update d_k using $[\delta^-, \delta^+]$ as in [10]
16. **end-if**
17. **if** p is a BLOCK and q is a NON-LEAF BLOCK **then**
18. **for** each child block or object b_p in p **do**
19. **for** each child block or object b_q in q **do**
20. $(\delta^-_{pq}, \delta^+_{pq}) \leftarrow d_I(b_p, b_q)$
21. **if** INTERSECTS($[[\delta^-_{pq}, \delta^+_{pq}], [d^-, d^+]$) **then**
22. Q.insert([KEY=δ^-_{pq}], $b_p, b_q, [\delta^-_{pq}, \delta^+_{pq}]$)
23. **end-if**
24. **end-for**
25. **end-for**
26. **else if** p is a BLOCK and q is a LEAF BLOCK **then**

```
27.         for each child block or object b_p in p do
28.             (δ⁻_pq, δ⁺_pq) ←d_I(b_p, q)
29.             if INTERSECTS([[δ⁻_pq, δ⁺_pq], [d⁻, d⁺]) then
30.                 Q.insert([KEY=δ⁻_pq], b_p, q, [δ⁻_pq, δ⁺_pq])
31.             end-if
32.         end-for
33.     else if p is an OBJECT and q is a BLOCK then
34.         for each child block or object b_q in q do
35.             (δ⁻_pq, δ⁺_pq) ←d_I(p, b_q)
36.             if INTERSECTS([[δ⁻_pq, δ⁺_pq], [d⁻, d⁺]) and p ∉
                S_o then
37.                 Q.insert([KEY=δ⁻_pq], p, b_q, [δ⁻_pq, δ⁺_pq])
38.             end-if
39.         end-for
40.     else if p ∉ S_o then   (* p and q are objects *)
41.         if (OPTION = UNORDERED) then
42.             if ([d⁻, d⁺]) CONTAINS [δ⁻, δ⁺] then
43.                 if (OPTION = SEMI-JOIN) then
44.                     add p to S_o
45.                 end-if
46.                 report <p, q>
47.             else if INTERSECTS([δ⁻_t, δ⁺_t], [δ⁻, δ⁺]) then
48.                 [δ⁻, δ⁺].REFINE()
49.                 Q.insert([KEY=δ⁻], p, q, [δ⁻, δ⁺])
50.             end-if
51.         else  (* OPTION = ORDERED *)
52.             (_, [δ⁻_t, δ⁺_t]) ←Q.top()
53.             if INTERSECTS([δ⁻_t, δ⁺_t], [δ⁻, δ⁺]) or not ([d⁻, d⁺]
                CONTAINS [δ⁻, δ⁺]) then
54.                 [δ⁻, δ⁺].REFINE()
55.                 Q.insert([KEY=δ⁻], p, q, [δ⁻, δ⁺])
56.             else
57.                 if (OPTION = SEMI-JOIN) then
58.                     add p to S_o
59.                 end-if
60.                 report <p, q> (and return)
61.             end-if
62.         end-if
63.     end-if
64. end-while
```

Algorithm 1 describes our distance join algorithm on a spatial network G which is based on the work by Hjaltason and Samet [10] and Shin *et al.* [18]. The algorithm takes two hierarchical data structures T and U, PMR quadtrees in our case, on the spatial positions of the objects in R and S, respectively. Even though we have used PMR quadtrees to describe our algorithm, our discussion is equally applicable to both object hierarchies, such as a R-tree and space hierarchies such as quadtrees and its variants. Moreover, we assume that the SILC decomposition of the G has been precomputed and is available to the algorithm. The SILC decomposition on G enables us to compute the distance interval between two objects, or between an object in T and a block in U. In addition, the algorithm has parameters d^+ and d^- corresponding to the minimum and maximum limits on the distances between object pairs in P. The next parameter k provides an upper bound on the number of object pairs to be computed by the join algorithm. Finally, the OPTION parameter serves to differentiate ORDERED and UNORDERED output of object pairs, and between a JOIN and a SEMI-JOIN. If OPTION is set to ORDERED, then the first invocation of the algorithm returns the the first object pair in the result. Subsequent object pairs in the result set are obtained by repeated invocation of the algorithm, *i.e.*, each invocation computes and returns only the next object pair in the result. If OPTION is set to UNORDERED, then the algorithm returns a set of object pairs corresponding to the result of the distance join operation.

Lines 1–8 initialize the algorithm by first choosing an appropriate data structure depending on the input parameters to the algorithm. If OPTION is set to ORDERED, Q is a *priority queue*. Otherwise, if OPTION is set to UNORDERED, then Q is defined to be a *list*. During the course of the algorithm, three types of pairs are generated by the algorithm – namely, block-block, object-block and object-object pairs– and are stored in Q. Moreover, if Q is a priority queue, then we assume that the pairs stored in Q are ordered in increasing order of the lower bound δ^- of the network distance interval between them which serves as the key. Q is initialized with a pair of blocks $<T, U>$, corresponding to the root of the two corresponding input hierarchical data structures.

In line 10, we obtain the pair $<p, q>$ from the *head* of Q. Let $[\delta^-, \delta^+]$ be the network distance interval between p and q. The algorithm's control structure is such that blocks p and q are split in an asymmetric manner in order to avoid obtaining pairs of the form $<p, q>$, where p is a block and q is an object. This is because computing the network distance interval from a block to an object using SILC is expensive, and hence, we avoid generating such pairs. If p is a block and q is a *non-leaf* block (lines 17–25), then they are split into their c_p and c_q children, respectively. The resulting $c_p \cdot c_q$ pairs of children of p and q are inserted into Q. Such a splitting strategy (termed "Simultaneous" in Section 5), may not be suitable when both p and q have a large outdegree as it may result in an explosion of pairs in Q. Another strategy (termed "Even" in Section 5) would be to split the block pairs more evenly *i.e.*, each time the larger one in p and q is split.

As we approximate the network distance between two blocks with their spatial distance, we may adopt different strategies to break ties between pairs of blocks at the same distance, each resulting in a different traversal of T and U. Choosing the pair with a block at the deepest level, results in a *depth first* traversal of T and U, while choosing the pair with a block at the shallowest level results in a *breadth first* traversal of the tree structures. These two competing strategies (termed "DFS" and "BFS") are further explored in Section 5.

In lines 26–32 of the algorithm, we handle pairs $<p, q>$, such that p is a block and q is a leaf block. Lines 33–39 handle the case when p is an object and q is a block. In the above two cases, the network distance interval of the c_p (c_q) children of p (q) to q (p) are computed and inserted into Q.

The final case when both p and q are objects is handled in line 40. If OPTION is set to UNORDERED, then the network distance interval $[\delta^-, \delta^+]$ between the object pairs is first checked against $[d^-, d^+]$ for *containment*. Next, if the intervals do not intersect, then $<p, q>$ cannot belong to the result set and is pruned. If the network distance interval $[\delta^-, \delta^+]$ intersects, but is not contained in $[d^-, d^+]$, then the network distance of $<p, q>$ is *refined* and $<p, q>$ is inserted back into Q. If the distance interval $[\delta^-, \delta^+]$ is contained in $[d^-, d^+]$, then the object pair is added to the result set P.

If OPTION is set to ORDERED, then $< p, q >$ cannot be reported unless it is certain that it is not being reported "out of order". In addition to checking against the distance

constraint $[d^-, d^+]$, we also need to compare it against the next element in the priority queue Q. The network distance interval of $<p, q>$ is compared against the network distance interval of the current top element in line 53 for intersection. If $[\delta^-, \delta^+]$ is both disjoint from the network distance interval $[\delta_t^-, \delta_t^+]$ of the current "top" pair in Q and δ^+ is less than δ_t^-, then $<p, q>$ is reported as the next object pair in the result. The algorithm at this point (line 60) returns the control back to the user. Subsequent pairs can be obtained incrementally by invoking the algorithm as many times as needed. If $[\delta^-, \delta^+]$ intersects $[\delta_t^-, \delta_t^+]$, then it is not clear if $<p, q>$ is the next object pair in the result. Hence, the network distance interval of $<p, q>$ is refined as before and inserted back into Q (as in line 55).

If OPTION is set to SEMI-JOIN, then the algorithm computes the distance semi-join operation. The distance semi-join requires that the algorithm keep track of the pairs that have been already reported –that is, if an object pair $< o_1, o_i >$ has already been reported, subsequent object pairs of the form $< o_1, o_j >$ should be pruned. We achieve this by storing a list S_o of objects in S that have already been reported by the algorithm. In particular, object p is added to S_o in line 59 thereby no subsequent pair containing p would be reported by the algorithm. This facilities pruning of pairs in lines 36 and 40, by comparing each pair with the objects in S_o. More aggressive pruning strategies, described in [10], can be employed here, which may result in further reduction in the number of pairs in Q.

When OPTION is set to TOP-K, we would estimate, as in [10], the upper bound d_k to the network distance of the k^{th} object pair in the result set. This would enable us to prune object pairs (line 11–16) that cannot possibly be present in the top k result. We use a separate priority queue, as described in [10], in order to estimate the value of d_k. We could have also used the technique described in [18] and leave the investigation of its use to a future study. Now, using d_k as the upper bound, we are able to prune object pairs, while constantly improving the estimate as more pairs are examined by the algorithm.

One interesting observation is that when T is an object, and if OPTION is set to ORDERED, Algorithm 1 is identical to the Best First Search (BFS) algorithm [9]. If T is an object, OPTION is set to UNORDERED and a distance constraint is specified, our algorithm performs a *range search*. If T is an object and the TOP-K constraint is specified, the algorithm retrieves the k nearest neighbors to T on the spatial network.

5. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our algorithm, and compare it with two other competing techniques reported in the literature – the JER and JRE techniques of Papadias *et al.* [13]. However, we point out that both of these techniques are limited in their functionalities. The JER and the JRE techniques can only perform simple UNORDERED distance join operations with an upper limit on the network distance between the object pairs. All of the experiments were carried out on a Linux (2.4.2 kernel) quad 2.4 GHz Xeon server with one gigabyte of RAM. We have implemented our algorithms using GNU C++. We tested our algorithms on a large road network dataset corresponding to the important roads in the eastern seaboard states of USA, consisting of 91, 113 vertices and 114, 176 edges. The SILC encoding of this road network was precomputed and stored on disk. The average number of the Morton blocks in the SILC map of a vertex in the dataset is 353. Objects belonging to the sets, R and S, were chosen at random from the vertices of the spatial network. The algorithm uses an LRU based cache that can hold 5% of the disk pages in the main memory.

In our experiments we examined the effect of choosing various block pair splitting strategies, as well as the various tree traversals on the performance of the algorithm. A block pair $<p, q>$ can be split simultaneously (termed "Simul") into children blocks *i.e.*, both p and q are split into children blocks, or only the bigger block among p and q is split (termed "Even"). Also, the algorithm can choose between a *depth first* (DFS) or a *breadth first* (BFS) traversal of T and U. The effect of choosing one of these four variants on an UNORDERED distance join is shown in Figure 4a. From the graph it is clear that depth first traversal ("DFS") with simultaneous ("Simul") splitting of block pairs was found to be slightly better than the other techniques. Figure 4b shows the effect of these variations on an INCREMENTAL distance join operation. Again, we can see that depth first traversal ("DFS") with simultaneous ("Simul") splitting of block pairs was found to be slightly better than the other techniques.

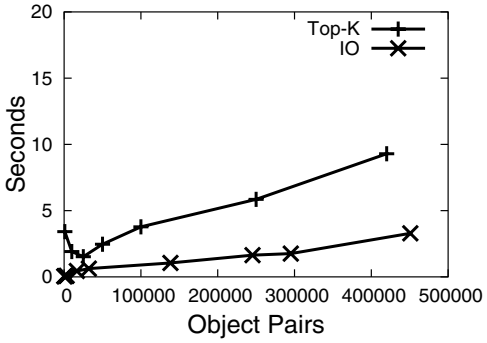

Figure 5: Execution time for a "TOP-K" distance join operation with different values of k.

Figures 5–6 shows the performance of the TOP-K distance join and semi-join operations on a road network dataset. Figure 5 shows the effect of varying the value of k on the performance of TOP-K UNORDERED distance joins between two sets of objects, R and S containing 1000 and 450 objects, respectively. Notice that the TOP-K UNORDERED distance join operation is slightly more expensive than an UNORDERED distance join operation shown in Figure 4a. Figure 6 shows the effect of varying the value of k on the performance of an UNORDERED distance semi-join operation with 18000 objects in R and 4500 objects in S. Notice that a larger proportion of the cost in Figure 6 is attributed to the CPU than in Figure 5, indicating that the algorithm expends a large number of clock cycles to ensure that an object pair is not present in S_o.

Figure 7 is a comparison of the time taken to perform distance join operations with an upper bound restriction on the network distance between object pairs using our approach (termed "SILC") with the JER and the JRE techniques. Our implementation of the JER and the JRE techniques is slightly different from the original formulation of Papadias *et al.* in [13]. In particular, we associate each object is R and

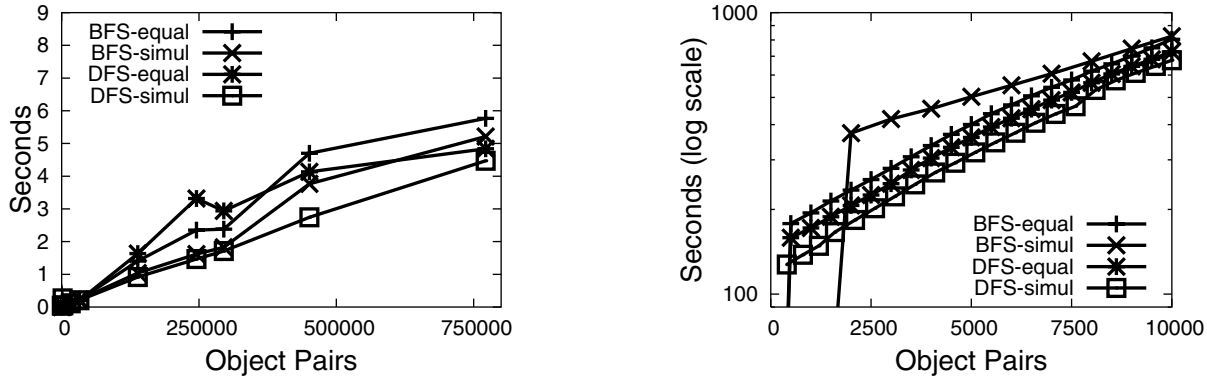

Figure 4: Execution time for list ordering and different block pair splitting strategies for an a) UnOrdered, b) Ordered distance join operation on a road network.

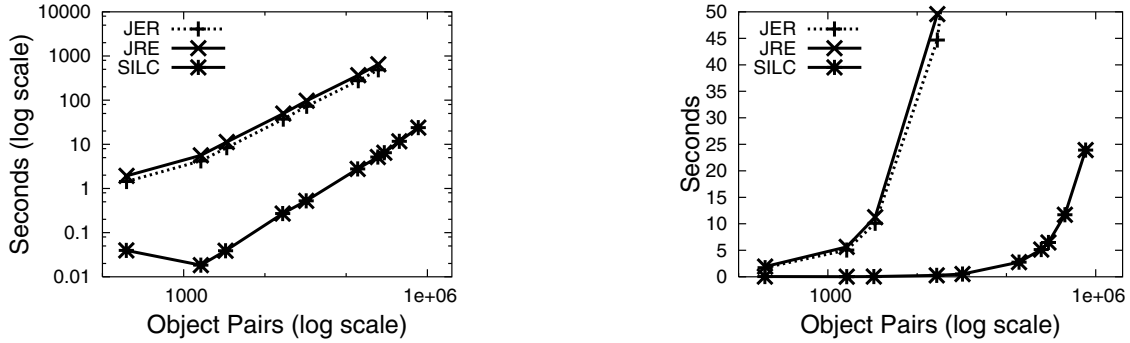

Figure 7: a) Execution times for the JER, JRE and our method ("SILC") is shown for distance join operation with a limit on the maximum distance between the object pairs. b) Figure in (a) has been redrawn (y axis not in logscale) in order to contrast the relative performance of the methods.

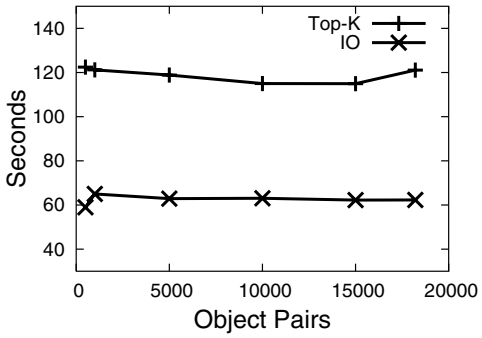

Figure 6: Execution time for a "Top-k" distance semi-join operation with different values of k.

S with a vertex in the spatial network, whereas Papadias *et al.* associate each object with an edge in the spatial network. Consequently, the "find_entities" operation discussed in [13] is no longer performed in our implementation of the JER and the JRE algorithms. Figure 7 shows that our technique, when compared to the JER and the JRE techniques, is at least an order of magnitude faster.

Figure 8 shows the performance of an incremental distance join operation on a road network dataset. As we can see from Figure 8, the bulk of the time taken by the join operation is spent on performing disk IO. Incidentally, a bulk of the IO time resulted from the repeated invocation of the Refine operator in line 48 of Algorithm 1. In effect, as the result of this operation is ordered, it has to establish a total ordering of the pairs in the result set, thereby resulting in several invocations of the Refine operator. One possible explanation is that choosing R and S uniformly has resulted in a large number of objects in S to be at the same distance interval with respect to objects in R thereby requiring many Refine operations.

Figure 9 shows the performance of a distance semi-join algorithm on the road dataset. The performance of this algorithm is similar to Figure 8.

6. CONCLUDING REMARKS

An algorithm has been presented that can perform a variety of distance join operations on spatial networks. Some possible directions for future work include the study of other join algorithms (*e.g.*, kNN join [5, 19]) on spatial networks and improvements to the SILC framework. In particular,

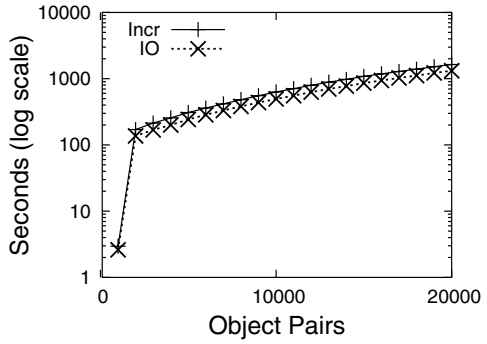

Figure 8: Execution time for incremental distance join algorithm

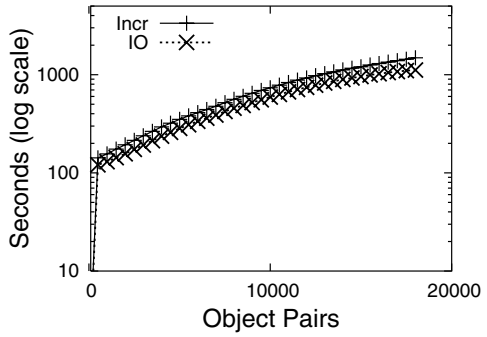

Figure 9: Execution time for incremental distance semi-join algorithm

we plan to extend the capabilities of SILC to be able to efficiently compute the network distance interval between two blocks, and between a block and an object. One possible solution that would allow the computation of the network distance interval between a block and an object is to compute an *inverse* SILC map for each vertex in a spatial network G, where an inverse SILC map of a vertex w is a region quadtree on the spatial positions of the vertices in G, such that the shortest paths from all the vertices contained in a block of the region quadtree to w share the same last link. Computing such an inverse SILC map would enable the computation of the network distance interval between a block and an object, although this does incur a considerable investment of additional disk storage.

7. REFERENCES

[1] S. N. Bespamyatnikh. An optimal algorithm for closest pair maintenance. In *SCG '95: Proceedings of the Annual Symposium on Computational Geometry*, pages 152–161, Vancouver, Canada, June 1995.

[2] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *SIGMOD '93: Proceedings of the ACM SIGMOD Conference*, pages 237–246, Washington, DC, May 1993.

[3] M. J. Carey and D. Kossmann. On saying "enough already!" in SQL. In *SIGMOD '97: Proceedings of the International Conference on Management of Data*, pages 219–230, Tucson, AZ, May 1997.

[4] H.-J. Cho and C.-W. Chung. An efficient and scalable approach to CNN queries in a road network. In *VLDB'05: Proceedings of the 31th International Conference on Very Large Data Bases*, pages 865–876, Trondheim, Norway, Sept. 2005.

[5] K. L. Clarkson. Fast algorithm for the all nearest neighbors problem. In *FOCS '83: Proceedings of the 24th IEEE Annual Symposium on Foundations of Computer Science*, pages 226–232, Tucson, AZ, Nov. 1983.

[6] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, Dec. 1982.

[7] A. V. Goldberg and R. F. Werneck. Computing point-to-point shortest paths from external memory. In *ALENEX '05: Proceedings of the 7th Workshop on Algorithm Engineering and Experiments*, Vancouver, Canada, Jan. 2005.

[8] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD '84: Proceedings of the International Conference on Management of Data*, pages 47–57, Boston, MA, June 1984.

[9] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *SSD '95: Proceedings of the 4th International Symposium on Large Spatial Databases*, pages 83–95, Portland, ME, Aug. 1995.

[10] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *SIGMOD '98: Proceedings of the International Conference on Management of Data*, pages 237–248, Seattle, WA, June 1998.

[11] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: an optimal model and its performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):409–432, May 1998.

[12] M. R. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB'04: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 840–851, Toronto, Canada, Sept. 2004.

[13] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB'03: Proceedings of the 29th International Conference on Very Large Databases*, pages 802–813, Berlin, Germany, Sept. 2003.

[14] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, CA, Aug. 2006.

[15] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *GIS '05: Proceedings of the 13th ACM International Workshop on Geographic Information Systems*, pages 200–209, Bremen, Germany, Nov. 2005.

[16] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. *GeoInformatica*, 7(3):255–273, Sept. 2003.

[17] S. Shekhar and J. S. Yoo. Processing in-route nearest neighbor queries: a comparison of alternative approaches. In *GIS '03: Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems*, pages 9–16, New Orleans, LA, 2003.

[18] H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In *SIGMOD '00: Proceedings of the International Conference on Management of Data*, pages 343–354, Dallas, TX, May 2000.

[19] P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbor problem. *Discrete & Computational Geometry*, 4(2):101–115, 1989.

[20] O. Wolfson, P. Sistla, B. Xu, J. Zhou, and S. Chamberlain. Domino: databases for moving objects tracking. In *SIGMOD '99: Proceedings of the International Conference on Management of Data*, pages 547–549, Philadelphia, PA, June 1999.