Efficient Query Processing 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

A framework for determining the shortest path and the distance between every pair of vertices on a spatial network is presented. The framework, termed SILC, uses *path coherence* between the shortest path and the spatial positions of vertices on the spatial network, thereby, resulting in an encoding that is compact in representation and fast in path and distance retrievals. Using this framework, a wide variety of spatial queries such as incremental nearest neighbor searches and spatial distance joins can be shown to work on datasets of locations residing on a spatial network of sufficiently large size. The suggested framework is suitable for both main memory and disk-resident datasets.

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*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Path and Circuit Problems*; E.2 [**Data Storage Representations**]: Linked Representations

General Terms

Algorithms, Performance, Design

Keywords

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

1. INTRODUCTION

Spatial databases are being deployed in GIS applications with desirable outcomes. Some of the recent advances in spatial database techniques have directly resulted in the ability to handle larger volumes of GIS data and to perform queries of increasing complexity. Although the main research focus in spatial databases is the efficient storage and query processing on data of arbitrary dimensions, GIS applications have been the main driving force. Performing spatial queries on *transportation networks* is an application that is of immense interest to the GIS community [27, 29, 31]. Transportation networks form an integral part of GIS applications like location-based services [23] and locational analysis [1]. Location-based services deal with queries generated by a mobile host. Moving object databases [23, 29, 30, 38] and trip-planning [31] are closely related to location-based services. Locational analysis [1] involves performing a series of sophisticated spatial queries in order to derive useful inferences. For example, urban planners wishing to find an 'optimal' location for a new hospital in an urban setting would issue a series of spatial queries to find a suitable location that is accessible to the general populace within a reasonable time.

The first contribution of our work is a novel framework that allows efficient processing of spatial queries on spatial networks. The proposed framework is sufficiently resilient to allow real time processing of both approximate and exact spatial queries on spatial networks.

Precomputing the shortest path and distance between all pairs of vertices in a spatial network is perceived to be prohibitively expensive. The second contribution of our work is to provide firm evidence to the contrary and to show that precomputing and storing the path-distance information is in fact feasible. In particular, we claim that the additional storage in the case of a road network is almost linear in the size of the network.

The third contribution of the paper is that we introduce the concept of progressive refinement of the inter-object distances between objects in a spatial network. This is in contrast to methods that require fixed computational cost in computing inter-object distances. This flexibility allows us to expend only a fraction of the cost in computing inter-object distances.

1.1 Spatial Networks

In this paper we describe a framework that enables a wide variety of spatial queries on a transportation network. To make the discussion more general, we introduce the concept of a *spatial network*, an extension to a network model. Classically, networks are modeled as a graph $G(V, E)$, where V denotes the set of vertices (or nodes) and E denotes the set of edges (or arcs) of the network. The set E represents the connectivity information of the graph; two vertices u and

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

GIS'05, November 4-5, 2005, Bremen, Germany.

v are *directly connected*, if and only if edge $(u, v) \in E$. Of particular interest is a weighted graph, where a weight is associated with each edge. A spatial network is an extension of a network such that additional spatial components are associated with the elements (vertices and, or edges) of the graph.

A road network is an example of a spatial network. A road network can be viewed as a weighted graph $G(V, E)$, such that each vertex represents a road intersection, and each edge represents a road segment. The spatial position of each vertex with respect to a reference coordinate system is also given, usually in terms of geographical coordinates (*i.e.*, latitude and longitude). Moreover, the weight of an edge represents the length of the associated road segment (or alternatively, the time required to travel the road segment).

1.2 Preliminaries

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 $d_u(v)$ to be the shortest distance from u to v with respect to the network $G(V, E)$. We may interchangeably use the notation $d_N(u, v)$ termed *network distance* to refer to $d_u(v)$. We use $\pi(u, v) \subset V$ 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. Note that the first link on the shortest path from u to v is $(u, l_u(v))$. We define the *path-distance* mapping $M : V \times V \rightarrow \mathbb{R}^+ \times V$ such that $M(u, v) = (d_u(v), l_u(v))$. For every pair of vertices $u, v \in V$, $M(u, v)$ provides the distance from u to v and the first link in the shortest path from u to v. In the above representation, the shortest path from u to v can be obtained by the repeated invocation of M, till v is obtained. Given p_i, the ith vertex on the shortest path from u to v, $M(p_i, v)$ provides p_{i+1}, the next vertex in the shortest path.

1.3 The SILC framework

Assume that a data structure S exists that efficiently computes $M(u, v)$ for any pair of vertices $u, v \in V$ in the spatial network. Given S, we claim that most spatial network queries can be efficiently processed. For instance, the distance between any two vertices u and v can be trivially obtained using S, while computing the path between them may need up to $k = |\pi(u, v)|$ queries on S.

A brute force implementation of S stores two values, $d_u(v)$ and $l_u(v)$, for each pair $u, v \in V$. This representation requires $O(n^2)$ storage and can compute queries on M in $O(1)$ time. The expensive storage costs involved with such an implementation have led previous researchers [29] to reject the brute force method in favor of alternative methods that approximate $d_u(.)$. We propose the Spatially Induced Linkage Cognizance (SILC) [1] framework as an efficient implementation of S. For any given vertex u, SILC stores an efficient representation of $d_u(.)$ and $l_u(.)$, that captures the path and distance information from u to all other vertices. The resulting representation is both computationally efficient (*i.e.*, provides efficient path and distance retrievals) and storage efficient (*i.e.*, less storage per vertex).

The SILC framework precomputes the shortest path between all pairs of vertices in a spatial network. We argue that such an approach is feasible and that the results can be *stored* given a reasonably large storage space. The Dijkstra's algorithm using a Fibonacci heap [5] takes $O(n^2 \log n + nm)$ time to compute the shortest path between all pairs of vertices in a spatial network. When $m = O(n)$, as in road networks, the time complexity of the Dijkstra's algorithm would be $O(n^2 \log n)$. Empirical studies [39] have indicated that the Dijkstra's algorithm may not be the fastest algorithm for computing the all pairs shortest paths on road networks. Moreover, recent developments in the shortest paths algorithm literature have shown better theoretical bounds on the computational time. In particular, Henzinger *et. al.* [12] present a linear time shortest path algorithm for planar graphs, while Thorup [33] provides a linear time shortest path algorithm for general graphs with integer edge weights. Using any of the above mentioned techniques, achieving a complexity bound of $O(n^2)$ for computing all pairs shortest paths is now possible. This bound is not unreasonable considering that a sorting operation on n integers takes $O(n \log n)$ time. A host of other techniques like parallel processing and the use of sophisticated hardware such as Graphical Processing Units (GPU) [19] could further speedup the precomputation of all shortest paths of a graph.

Our work is the first, to the best of our knowledge, that efficiently encodes both the path and distance information accurately. In this paper, we define the concept of *path coherence*, that identifies the underlying coherence between the shortest paths and the spatial positions of vertices on a spatial network. Path coherence in spatial networks allows us to subdivide the space into coherent spatial regions. All vertices contained in a coherent region share the first segment of their shortest path from a fixed vertex. Subsequently, the spatial regions are compactly represented as a collection of Morton blocks [7]. Using the above formulation, we show that most spatial query processing techniques that were originally developed for traditional spatial databases, can be applied on spatial networks.

The rest of the paper is organized as follows. Section 2 introduces the concept of path coherence and further discusses strategies that take advantage of path coherence in spatial networks in order to encode the path and distance information between vertices compactly. Section 3 explains a method for storing an approximate distance range between each vertex pair. In Section 4, we show how traditional spatial techniques can be applied to the SILC framework. Section 5 presents the experimental results, and Section 6 compares our work with other competing techniques. Concluding remarks are drawn in Section 7.

2. COLOR CODING THE MAP

We treat spatial networks as general graphs whose vertices have fixed spatial positions. We observe that vertices that are spatially close to one another share a number of common properties. In particular, often, two vertices u, s that are spatially close to each other share large common segments of their shortest path to two other vertices v, t

[1]The *silk road* is an ancient trade route that connected people from different cultures. The SILC framework, although less ambitious, connects locations on a spatial network.

Figure 1: Example illustrates 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).

that are spatially close to each other, but far from u, s. To illustrate the above claim with a day-to-day example, commuters who live in the same neighborhood (live spatially close to one another) mostly use the same roads when traveling to nearby destinations. We call the coherence between the shortest path of nearby sources to nearby destinations as *path coherence*. Path coherence occurs naturally in most spatial networks that are of interest to the GIS community. Wagner and Willhalm [37] discuss the reverse problem of assigning spatial positions to vertices in a general graph so that path coherence is *induced* into the the resulting spatial network.

The SILC framework takes advantage of the path coherence between vertices in a spatial network in order to encode the path and distance information between all pairs of vertices. In order to illustrate the working of the path encoding method in the SILC framework, we devise a method of assigning *colors* to vertices in a spatial network. The goal of the coloring process is to arrive at a method that efficiently represents the path-distance information from a vertex u to all other vertices in a spatial network. Note that a brute force method takes $O(n)$ space per vertex to represent the path-distance information. Suppose that u has D adjacent vertices. We use D distinct colors corresponding to each adjacent vertex of u. We assign color i to vertex v, if and only if, $l_u(v)$ is the i-th adjacent vertex of u. In effect, all vertices in V that share the same first link on the shortest path from u are assigned the same color. At the completion of the coloring operation, each vertex is colored with one of the D colors. We also encode vertices that do not have a path from u using a special color. For a given vertex u, we first compute all the shortest paths from u to all other vertices. The coloring operation uses these precomputed paths to proceed with the coloring process which is described in Algorithm 1. In line 1, all the shortest paths from u (SSSP) are computed. In lines 2–3, all vertices $v \in V$ are assigned colors based on which adjacent vertex of u forms the next link in the shortest path from u to v.

Algorithm 1
Procedure ColorizeMap[u]
1. Compute SSSP(u)
2. **for** each $v \in V$ and $v \neq u$ **do**
3. assign $color[v] = color[l_u(v)]$
4. **return**

Figure 1 illustrates the coloring process. Once the coloring operation has been completed, the spatial network has large contiguous colored regions. In particular, v and w belonging to the region denoted by color i means that the first link in the shortest path from u to v and w is the same. The large contiguous colored regions are due to the *path coherence* between the vertices.

2.1 Colored Tiles

The coloring operation on a spatial network creates regions that share the first link in the shortest path. If these spatial regions can be accurately stored using some spatial data structure, then the path information from u to all other vertices in the spatial network are encoded accurately. There are many possible ways to store these regions. We can use object hierarchies such as an R-tree [11] or methods based on a disjoint decomposition such as a region quadtree [25]. We use a disjoint decomposition as we do not want the regions corresponding to different links to overlap which will occur often due to their irregular shape. Having opted for a representation based on a disjoint decomposition, we use a regular decomposition such as a quadtree variant, instead of one based on an irregular decomposition such as an R+-tree [28] due to ease of implementation. In particular, we store these colored regions in a region quadtree. We then represent the regions as a set of Morton blocks [7]. Morton blocks provide an efficient representation of the regions, *i.e.*, they are known to be efficient in handling containment queries and can be stored compactly [25]. A link and a distance interval (explained in Section 3) are associated with each Morton block.

Algorithm 2
Procedure MortonIze[u, T]
Input: $u \in V$. T is a PMR-quadtree on V
Output: MortonList: list of Morton blocks with associated links and distance intervals
1. MortonList ←`empty`
2. **for** each leaf-block $b \in T$ visited in *Morton-order* **do**
3. **if** all points v in b are of same color **then**
4. append b to MortonList
5. **else**
6. recursively split b until S, the resultant set of blocks, is single colored
7. merge S with MortonList
8. **while** Morton blocks can be merged **do**
9. merge sibling blocks if of the same color
10. **for** each Morton block $b \in$ MortonList **do**
11. $\lambda^- = \min_{v \in b} \frac{d_N(u,v)}{d_S(u,v)}$

12. $\lambda^+ = \max_{v \in b} \frac{d_N(u,v)}{d_S(u,v)}$
13. associate (λ^-, λ^+) with b
14. **return** MortonList

Algorithm 2 is an efficient method to construct a list of Morton blocks that represent the colored regions from an initial PMR-quadtree [25] representation of the vertices (line 2). Of course other representations could have been used as the input to the algorithm but we leveraged our existing SAND [26] spatial database system that provided a robust disk based PMR-quadtree implementation. Blocks in the PMR-quadtree are said to be of a uniform color if all the vertices contained within them are of the same color. Thus, adjacent blocks can be combined to form larger blocks, although some blocks may need to be split in order to ensure that all the vertices that they contain are of the same color. The resulting representation is a region quadtree on the colored regions. Now, each leaf-block in the region quadtree is represented as a single Morton block and may be stored on disk. Lines 10–13 are explained in Section 3.

Quadtree representations of regions have been shown to be good dimensionality reducing mechanisms [25], i.e., the storage requirements needed to represent a region R in a region quadtree is $O(p)$, where p is the perimeter of R. Note that the number of regions in a region quadtree corresponding to a vertex is proportional to the outdegree of the vertex, which is relatively small. Our experiments (see Section 5) show that the storage requirements for the SILC framework on road networks are achievable, that is, the storage per vertex is almost independent of the size of the spatial network. This represents a considerable improvement over the brute force encoding.

To improve upon the storage requirements even further, a number of alternate representations are suggested. We may choose not to store the colored region with the highest number of Morton blocks. This may result in some savings in storage, although path retrievals may become slightly more expensive. The colored regions, as seen in Figure 1b, have radial structures. A simple transformation of the space to polar coordinates may help improve the storage costs. *Chain Code* techniques [6] or variations of *Medial Axis Transformation* (MAT) techniques like *Corner MAT* [25] and *Quadtree MAT* [24] could be used instead of representing regions as a list of Morton blocks. However, unlike the Morton blocks, they may not allow efficient computations.

2.2 Retrieving the shortest path

Given a source vertex s, a destination vertex v, and an intermediate vertex u in the shortest path between s and v, the next link in the shortest path is obtained by performing a simple binary search, for a Morton block containing v, on the Morton list stored with u as described in Algorithm 3. Note that SILC only stores t, the next link after u in the shortest path from s to v. To retrieve the complete path, subsequent invocations need to be performed by replacing u with t, until t equals v. Note that retrieving the shortest path between s and v needs exactly $k = |\pi(s,v)|$ invocations of the NextinPath routine, resulting in k disk accesses. Also, the distance between s and v can be obtained by maintaining a variable and adding up the distances along each individual link comprising the path. Thus, we see that the SILC framework explicitly encodes the path information, while the distance information is implicitly recorded.

Algorithm 3
Procedure NextinPath[s, u, v, d]
Input: s is the source vertex, and v is the destination
Input: u is an intermediate vertex
Input: d holds the network distance from s to u
Output: t is the next vertex in the shortest path
Output: b is the Morton block containing v
Output: d is the distance from s to t
1. retrieve the path encoding for u into MortonList
2. binary search on MortonList for block b containing v
3. $t \leftarrow b.link$
4. $d \leftarrow d + w(u,t)$
5. **return** t, b, d

3. DISTANCE ENCODING

For most spatial applications, an approximate estimate of the distance between two vertices u and v on a spatial network would suffice. Depending on the nature of the spatial network and the space in which the vertices are embedded, it may not be difficult to arrive at a distance function that approximates the network distance. For example, in the case of a road network, the *geodesic* distance between two locations always lower bounds the distance along the road segments. We construct an approximate distance function as part of the SILC framework by storing two values λ^- and λ^+ with each Morton block in the representation. For a Morton block associated with a source u, λ^- (λ^+) is the minimum (maximum) ratio of network distance to the spatial distance from u to all destination vertices in the Morton block. Given a source u, a destination v, and the values λ^- and λ^+ associated with the Morton block containing v, we have $\lambda^- d_S(u,v) \leq d_N(u,v) \leq \lambda^+ d_S(u,v)$. In other words, SILC can efficiently compute an interval on $d_N(u,v)$ using $d_S(u,v)$, λ^- and λ^+. Minor modifications to the Mortonize algorithm (Algorithm 2, lines 10–13) allow λ^- and λ^+ to be incorporated in the SILC framework.

Given any two vertices, u and v in a spatial network, an initial interval on $d_N(u,v)$ is made available by the SILC framework. In addition, we provide the RefineDist (Algorithm 4) operator in order to *tighten* the interval by expending some work. The operator incurs exactly one disk access to identify t, the next link after an intermediate vertex u in the shortest path from s to v. The distance interval is improved by taking the intersection of the initial interval between s and v, with the interval obtained using t. Subsequent refinements are possible by identifying the next link in the shortest path between s and v, and so on. It is clear that this interval converges to a single value after at most $k = |\pi(s,v)|$ invocations of the RefineDist operator, after which the network distance between s and v is known.

Algorithm 4
Procedure RefineDist[s, u, v, d, δ^-, δ^+]
Input: s is the source vertex, and v is the destination
Input: u is an intermediate vertex
Input: d holds the network distance from s to u
Input: (δ^-, δ^+) holds an interval $d_N(s,v)$
Output: t is the next vertex in the shortest path
Output: d holds $d_N(s,t)$
Output: (δ^-, δ^+) are the updated interval on $d_N(s,v)$
1. $(t, b, d) \leftarrow$NextinPath(s, u, v, d)
2. retrieve λ^- and λ^+ from b
3. $\delta^- \leftarrow \max(\delta^-, \lambda^- \times d_S(t,v) + d)$

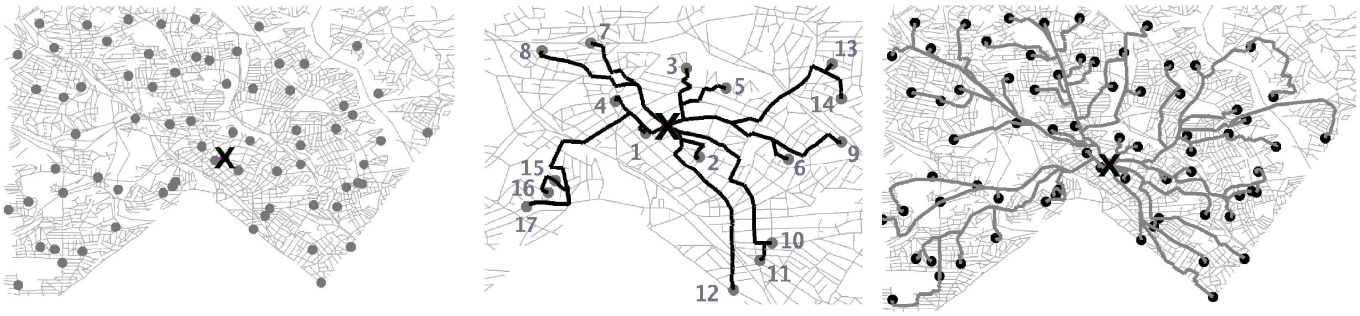


Figure 2: Mechanics of a nearest neighbor search [15] on a road network. a) Initial configuration: A query point (denoted by "X") and a set of locations filled circles. b) Query progression: Partial result of ranking the dataset of points based on the length of their shortest path from the query point. Notice that location "3" is reported as a closer neighbor to the query point than "4", even though the spatial distance between location "4" and "X" is lesser than the spatial distance between location "3" and "X". c) Final result: All points have been ranked by their network distance to the query point.

4. $\delta^+ \leftarrow \min(\delta^+, \lambda^+ \times d_S(t,v) + d)$
5. **return** t, d, δ^-, δ^+

3.1 Distance Functions

The SILC framework explicitly encodes the path and an approximate distance between every pair of vertices, while the network distance is implicitly recorded. The approximate distance is recorded as a distance interval that consists of a lower bound, and an upper bound. Determining the distance interval between a pair of vertices costs almost nothing, while computing the network distance between vertices takes extra work and should be avoided if possible. All distance measures used in the SILC framework are distance intervals. We point out that the distance interval is sufficient in most cases where only the relative positions of objects need to be determined. For example, the nearest neighbor to a query object q, is an object p which is closer to q than any other object in the dataset. p is determined to be the closest neighbor to q, if the upper distance bound provided by the distance interval of p from q is less than the lower distance bound of all other objects in the dataset. In other words, if the distance interval of p is less than and non-intersecting with that of all other objects in the dataset, there is no ambiguity that p is the closest neighbor of q. Each invocation of the RefineDist operator discussed in Algorithm 4 causes the interval to become tighter. When the interval converges to a single value, it corresponds to the network distance between the vertex pair. We point out that the network distance between a vertex pair can be achieved with at most $k = |\pi(p,q)|$ disk accesses, by the repeated invocation of the RefineDist operator.

We now redefine some of the concepts related to distances commonly used in spatial algorithms. A distance $d = (\delta^-, \delta^+)$ is recorded as an interval with a lower bound δ^- and an upper bound δ^+. When δ^- equals δ^+, d is the network distance. Union of two distance intervals d_1, d_2 is the tightest interval that contains both d_1 and d_2. The Intersects operator of two distance intervals d_1, d_2 determines if the two intervals share a common value.

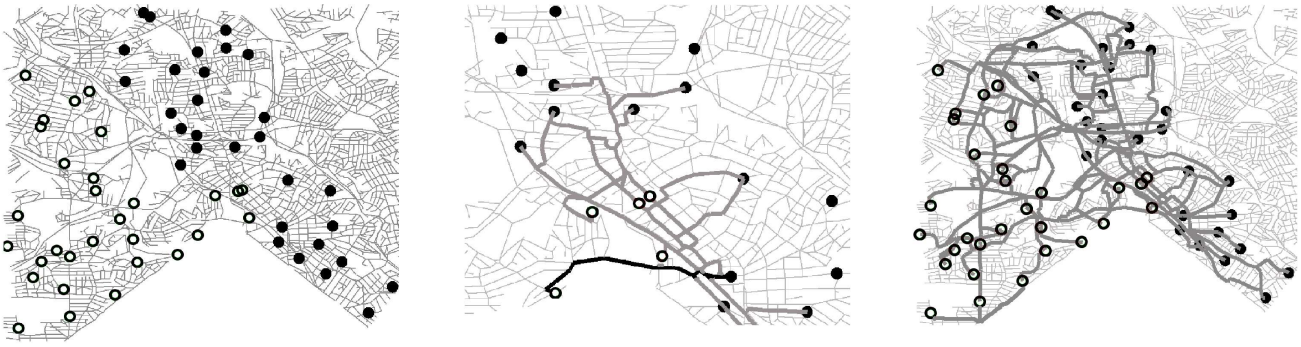
The IntrvlDist_N operator of a vertex v and a region R finds a distance interval such that the interval contains the network distance from v to each vertex contained in R. In spatial algorithms MinDist_S (MaxDist_S) between a point v and a region R is the minimum (maximum) possible spatial distance between v and any point contained in R. IntrvlDist_N function returns a distance interval $d = (\delta^-, \delta^+)$ such that for any vertex t contained in R, $\delta^- \leq d_N(v,t) \leq \delta^+$. We are able to compute suitable values for δ^- and δ^+ by using the MinDist_S and MaxDist_S distances between v and R and the path-distance map of v as shown in Algorithm 5

Algorithm 5
Procedure $\text{IntrvlDist}_N[v, R,$ MortonList]
Input: R is a region, v is a vertex
Input: MortonList is the path encoding for v
Output: $d = (\delta^-, \delta^+)$ forms the distance interval
1. **for** each $b_i \in$ MortonList intersecting R **do**
2. retrieve λ^- and λ^+ from b_i
3. $r_i \leftarrow$intersection of b_i and R
4. $\mu_i^- \leftarrow \lambda^- \times \text{MinDist}_S\ (v, r_i)$
5. $\mu_i^+ \leftarrow \lambda^+ \times \text{MaxDist}_S\ (v, r_i)$
6. **return** Union of all (μ_i^-, μ_i^+)

To summarize the SILC framework, it explicitly encodes the shortest path and a distance interval (approximate distance) between all pairs of vertices, while implicitly recording the network distance. Each vertex stores a set of Morton blocks associated with a link and distance interval λ^+ and λ^-; collectively referred to as the *pilot-data* of a vertex.

4. APPLICATIONS

The SILC framework enables the use of many well known query processing techniques – that were developed for classic spatial databases – on spatial networks. In this section, we briefly describe a set of sample spatial queries that can be implemented using the SILC framework. To begin with, we assume that a set of objects (points) are provided as input to our algorithm. We assume a spatial data structure (*e.g.*, a PMR quadtree [21]) is built over these points based on their spatial positions. Also, a spatial network with a precomputed SILC encoding is made available to the algorithms. For the sake of simplicity we assume that each point in the set is associated with exactly one vertex on the spatial network. An object that lies on the interior of a directed edge (u, v) can be modeled as an object at u, and then



(a) (b) (c)

Figure 3: Mechanics of an incremental distance join [14] on a road network. a) Initial configuration: The road network and two sets of locations denoted by filled and and hollow circles. b) Query progression: At each step, the distance join fetches the next closest pair of points, one drawn from either of the sets of locations. The lines in a darker shade, denote the shortest path between the latest pair of points retrieved by the join algorithm, and the lines in a lighter shade correspond to the shortest paths between previously obtained pairs. c) Final result: All pairs of points obtained by the distance join and the shortest paths between them.

suitably adjusting the distance measures. Using the above formulation, we can perform a variety of spatial queries as demonstrated in this section.

The rest of the section is organized as follows. Section 4.1 describes a set of queries that can be performed using the framework and Section 4.2 describes an incremental best-first search (BFS) algorithm and an incremental distance join variant that works on the proposed framework.

4.1 Queries on a spatial network

Here, we describe a few basic queries that are performed on a spatial network. For each query we construct an application scenario.

- **path and distance queries:** Compute the shortest path and distance between two locations on a spatial network. (*e.g.*, find the distance and the path from the accident scene to the hospital.)
- **range queries:** Find all locations that exist within a distance of r from a specified query point. (*e.g.*, find all hospitals that are within one mile road distance – or equally, can be reached within five minutes of driving – from the accident scene.)
- **incremental nearest neighbors:** Incrementally retrieve the nearest neighbors to a query point. (*e.g.*, find the nearest hospitals to the accident scene in the increasing order of the trip time.)
- **distance join and distance semi-join:** Given two sets of spatial objects, S and R, incrementally retrieve the closest pair of objects. Distance semi-join [14] requires that objects from S appear only once in the output. (*e.g.*, given a set of stores and another set of warehouses, incrementally retrieve the closest pair containing a store and a warehouse, in the increasing order of the trip time. The distance semi-join finds the closest warehouse to each store.)

4.2 Incremental Neighbor search

We use a variant of the Best First Search(BFS) method by Hjaltason and Samet [13] to compute the nearest neighbors to a query point on a spatial network. Figure 2 is an illustration of the application of our method to a road network dataset. As the inter-distance between objects in the SILC framework are distance intervals, minor modifications are made to the original algorithm. Algorithm 6 depicts the working of a BFS method on a spatial network. The algorithm takes three inputs, a pointer T to the root of a hierarchical spatial data structure containing the set of points (*e.g.*, a set of hospitals) from which neighbors are drawn, a query point q and MortonList, the path-encoding of q. The algorithm uses a priority queue Q of points and blocks, collectively referred to as objects. The distance interval of objects from q are stored. Additionally, a few additional pieces of state information are stored when the object s is a point, *i.e.*, an intermediate vertex u in the shortest path from s to q, and the distance d from s to u. Q retrieves stored objects in an increasing δ^- ordering from q.

Lines 1–3 are executed once for each instance of an incremental nearest neighbor query. The priority queue, Q, is initialized by inserting the root T. At each iteration of the algorithm, the top element in the queue is examined. If the element is a Leaf block, then it is replaced with all the points contained within the block. If a Non-leaf block is retrieved, then all of its children are inserted into the priority queue. If a Point p is found, then the distance interval of p is checked with the top element in the queue for possible *collisions*. A collision takes place when the distance interval of p intersects with the distance interval of the top element in the queue, in which case the distance interval of p is refined by applying the RefineDist operator (described in Algorithm 4) and is re-inserted back into the queue. If the distance interval of p is non-intersecting with the top element of the queue, p is reported (line 16) as the next neighbor to q and the function returns to the caller. More neighbors of q can be retrieved by making subsequent invocations to the routine, – starting at line 4– resulting in an *incremental* retrieval of neighbors.

Algorithm 6
Procedure NetworkBFS[T, q, MortonList]
Input: $T \leftarrow$root node of spatial structure on P
Input: q is the query point

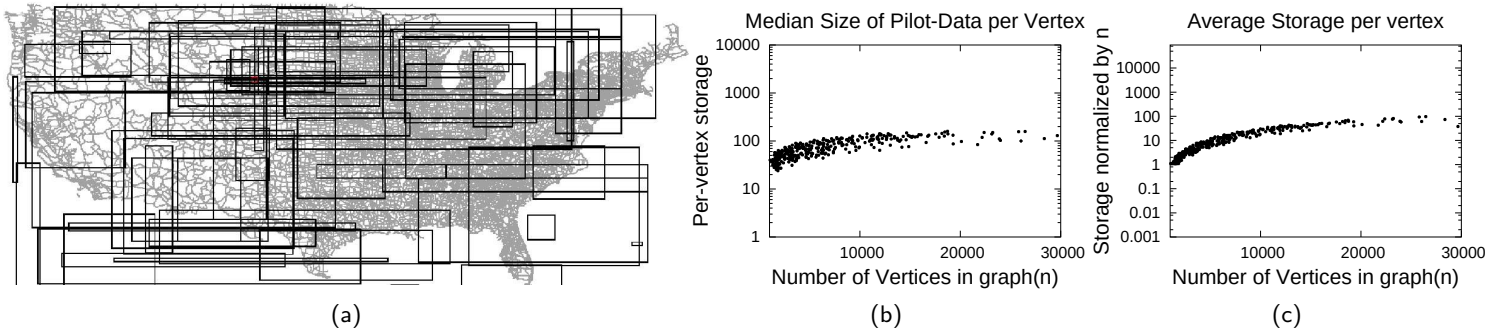


Figure 4: b) Median and c) average sizes of pilot data associated with a) rectangle-shaped regions containing subsets of roads in the USA.

Dataset	Vertices	Edges	average outdegree
Silver Spring (SS)	4400	5800	2.7
Washington (DC)	12400	18000	2.9
Boston (BOS)	17400	24000	2.69
New York City (NYC)	40000	62000	3.1
Major Roads (USA)	380000	400000	2.2

Figure 5: Sample datasets.

Input: MortonList is the path encoding of q
Output: p is the next nearest neighbor to q drawn from P

```
1.  Init: (δ⁻, δ⁺) ←IntrvlDist_N (q, T, MortonList)
2.        Q ←an empty priority queue on tuples
3.        Q.insert(Key=δ⁻, (T, δ⁻, δ⁺, q, 0))
4.  while not (Q.empty()) do
5.      (p, δ⁻, δ⁺, u, d) ←Q.pop() (* Extract top element *)
6.      if p is a Block then
7.          for each child block or point R in p do
8.              (δ⁻, δ⁺) ←IntrvlDist_N (q, R, MortonList)
9.              Q.insert(Key=δ⁻, (R, δ⁻, δ⁺, q, 0))
10.     else (* p is a point *)
11.         (_, μ⁻, μ⁺, _, _) ←Q.top()
12.         if Intersects((μ⁻, μ⁺), (δ⁻, δ⁺)) then
13.             (u, d, δ⁻, δ⁺) ←RefineDist (q, u, p, d, δ⁻, δ⁺)
14.             Q.insert(Key=δ⁻, (p, δ⁻, δ⁺, u, d))
15.         else
16.             report p (and return)
```

The distance join and distance semi-join algorithms of Hjaltason and Samet [14] can be similarly adapted to work on spatial networks. The variant we propose, uses a priority queue similar to the one used in the NetworkBFS (Algorithm 6). The distance join algorithm takes two sets of locations as input, and then constructs pairs of elements (blocks or points), one of which is drawn from either set (see [14] for more details). The priority queue retrieves objects in an increasing IntrvlDist_N ordering of the distance between the element pairs. Figure 3 illustrates the working of our algorithm on a sample road dataset.

5. EXPERIMENTS

The SILC framework presented in this paper provides a compact representation of the path and distance information between any pair of vertices on a spatial network. In this section, we present an experimental evaluation of our technique. The experiments were carried out on a Linux (2.4.2 kernel) quad 2.4 GHz Xeon server with one gigabyte of RAM. We implemented our algorithms using GNU C++. A number of publicly available road network datasets were used in the evaluation. These were obtained from the US Tiger Census [34] and the National Atlas [35] websites. Some of the datasets that we used are described in Figure 5.

The framework presented in the paper can be used for interactive query processing on large spatial network datasets such as road networks. One of the critical requirements for building a *scalable* interactive application is that the size of the input should not significantly affect the performance of the application. The size of a spatial network, denoted by n, is the number of vertices comprising the input. The size of the spatial network has the following effects on the performance of our algorithm: (i) The size of the pilot-data stored with each vertex in the SILC framework depends on n and grows gracefully as n gets larger; (ii) The size of the pilot-data directly affects the time taken to perform the path and distance computations.

We tested our algorithm by taking random samples from a large road-network dataset. In particular, we used a dataset containing all the major roads in the USA (*i.e.*, more than 380,000 vertices and 400,000 edges). Sample random rectangular regions were drawn from the dataset and the road network segments contained completely within them were extracted to serve as inputs to the evaluation of our algorithm (see Figure 4a). By taking the samples at random we were able to account for variations such as rural versus urban, and spatial network configurations that would lead to different pilot data sizes on account of the number of blocks needed for the underlying region quadtree.

From Figure 4b and 4c, we see that the size of the pilot data associated with each vertex grows gradually for smaller graphs until a stable value is reached for larger spatial graph inputs. This seems to suggest that the per-vertex storage requirement of our encoding is almost independent of, or minimally dependent on, n.

In the first set of experiments, we selected pairs of vertices at random from the Silver Spring, MD road data and computed the shortest path between them and their road distance by repeated invocations of Algorithm 3. This algorithm takes k steps for a path of length k. Figure 6a tabulates the CPU and I/O cost (in milliseconds) of this operation as a function of the different path lengths. As expected, the cost of computing the shortest path is directly proportional to the length of the path between the

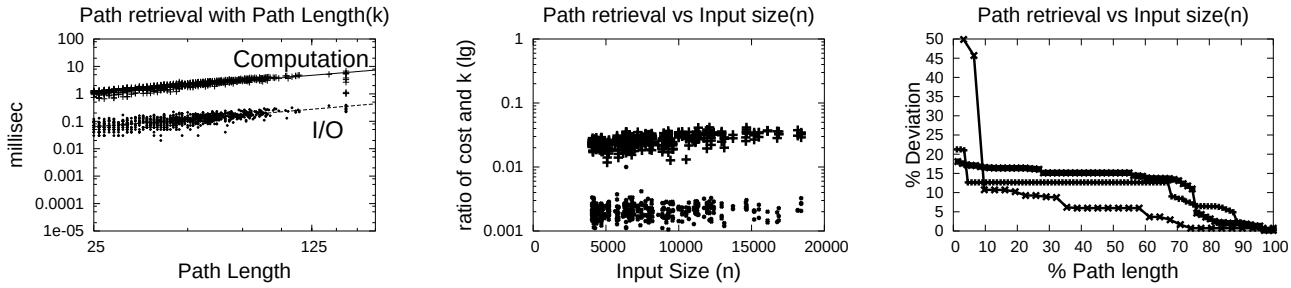


Figure 6: a) CPU time (top) and I/O time (bottom) to retrieve the shortest path between two arbitrary vertices versus the length of the path between them for the Silver Spring, MD map. b) CPU time (top) and I/O time (bottom) normalized by path length versus the size (i.e., number of vertices) of a randomly chosen rectangular sample of the data in the USA map of Figure 4a. c) Relationship between the deviation ratio of the shortest path length and the percentage of the path completed for three sample paths from the Silver Spring, MD map.

constituent vertices, and pairs. Also, retrieving a path of length k results in k disk accesses.

The second set of experiments tabulate the effect of the number of vertices n in the data set on the CPU and I/O costs (in milliseconds) of the shortest path algorithm. We used a set of randomly generated spatial networks obtained by extracting rectangular samples from the USA road data [35], which is a large road network. For each sample we extracted a number of vertex pairs at random and computed the CPU and I/O cost of the shortest path between them which was normalized by the length k of this path and is shown in Figure 6b. From the figure, we see that these normalized costs are relatively independent of n, which is in keeping with our earlier observation that that the size of the pilot-data is nearly independent of n.

The third set of experiments is designed to show the effectiveness of the RefineDist operator used in Algorithm 4. As we pointed out, every block keeps track of the maximum and minimum deviation of the network distance between the starting and ending vertices s and e for paths through points within the block (i.e., along the network) from the spatial distance between s and e. Use of the RefineDist operator tightens the interval as the path is computed by incurring one additional disk access and is important in processing the different spatial queries discussed in Section 4. Figure 6c shows the relation between the ratio of the deviation of the computed network distance interval to the actual network distance, and the percentage of the path completed for three sample paths from the Silver Spring, MD road dataset. Each marker in Figure 6c corresponds to one RefineDist operation. From the figure we observe that as we approach the destination, the error quickly reduces to a small value.

6. RELATED WORK

Shortest path computation on general graphs has been extensively investigated in the field of theoretical computer science. The best known algorithm to find single source shortest paths (SSSP) is the Dijkstra's algorithm [2]. A variation of the Dijkstra's algorithm that uses a Fibonacci heap structure [5] runs in $O(n \log n + m)$ time to find a single shortest path and takes $O(n^2 \log n + nm)$ time to find all pairs shortest paths (APSP). The Dijkstra's algorithm requires that all edges have non-negative weight. The Floyd-Warshall algorithm [4], on the other hand, can work on graphs with negative edge weights and takes $O(n^3)$ time to compute the all pairs shortest paths. A recent survey paper by Zhan and Noon [39] compares the relative performance of many of the classical shortest path algorithms when applied on a road-network dataset.

Of particular interest are techniques that deal with disk-based representations and bucketing [16, 22] strategies for storing large graphs. Few techniques strike a balance between preprocessing and real-time computation of the path and distance information. The *hierarchal graph* representation by Jing *et. al.* [17], and the more recent work by Filho and Samet [3], propose precomputing a hierarchal set of graphs from an input spatial network. Each level in the representation progressively simplifies the graph structure by replacing a set of vertices in the graph input by a smaller set, thereby reducing the size of the representation. Path and distance between pairs of vertices are identified at run time using the precomputed set of graphs. Mitchell *et. al.* [20] describe an algorithm for computing shortest paths on 3D meshes, and Surazhsky *et. al.* [32] demonstrate an effective implementation. Note that the SILC framework is also applicable to 3D meshes.

Wagner and Willhalm [36] present a *geometric approach* for speeding up shortest path computations in a spatial network. For each edge $e = (u, v) \in E$ in the network, consider the set $S(e)$ containing all vertices $t \in V$, such that the shortest path from u to t passes through e. For each edge $e \in E$ of the network, the method first computes $S(e)$, and then associates – and stores – $L(e)$, a *geometric container* with e. The geometric shape as defined by $L(e)$ contains all the elements $t \in S(e)$ and possibly few extra ones. Geometric containers can be of any simple geometric shape like circles, ellipses, or bounding boxes and require only $O(1)$ bits to store. Thus, the extra amount of space required for storing geometric containers is linear in the number of edges of the spatial network. Geometric containers are then used to speed up future shortest path queries on the graph representation. A shortest path query from s to r, only visits those edges e whose geometric container spatially contains r. This pruning may lead to significant speed-up. Although, shortest path queries can still be quite expensive. First of all, the geometric container stored along with each edge is an

approximation of the actual region spanned by $S(e)$. Consequently, $L(e)$ may contain many vertices, whose shortest path from u does not pass through e. Therefore, e may not be pruned from shortest path queries with such vertices as the destination. As a result the path and distance computations may be quite expensive as multiple paths need to be examined. The method does not explicitly store the distances between vertices, hence, distance computations are as expensive as the shortest path queries.

Recent work by Goldberg and Harrelson [8] introduces a strategy, termed ALT, utilizing the A^* search heuristic for speeding up the shortest path computations on a spatial network. To begin with, a set of points on the spatial network, called *landmarks*, are chosen. The shortest distance between all the vertices in the network and the landmarks are computed and explicitly stored. Given a shortest path query between two vertices, the method first identifies a subset of landmarks that can potentially aid the A^* search process. With the aid of the distances to the landmark points and using the triangle inequality, a large number of edges can be pruned away from the search. Goldberg and Werneck [9] describe an implementation of the ALT algorithm on a handheld device as a standalone application. We point out that our method is more suited for a client-server scenario, where a large number of shortest path and distance queries are handled simultaneously.

The *Road Network Embedding* (RNE) technique proposed by Shahabi *et. al.* [29] is similar to the work of Goldberg and Harrelson [8]. Instead of explicitly storing the distances from all vertices to the landmark points, the RNE technique embeds the vertices of the spatial network in a high-dimensional vector space using the distances from all vertices in the spatial network to a random set of landmark points. Once projected to this high-dimensional space, an L_∞ Minkowski metric (*i.e.*, the Chessboard metric) can be used to find the distances between points. In effect, the embedding method trades a complicated network distance function for a simpler distance function in a high-dimensional space. However, this embedding method does not preserve distances nor does it preserve the relative positions between the objects. The RNE approach has a number of other drawbacks. First of all, the method can only provide approximate distances between points with $O(\log n)$ distortion. Also, as the path information is not stored, an approximate path between vertices can be retrieved at a significantly higher cost than the SILC framework. Moreover, the RNE approach embeds the vertices in a high-dimensional vector space. Consequently, we suspect that this method may lead to poor performance owing to the *curse of dimensionality*. In a related note, the recent work by Gupta *et. al.* [10] propose a hypercube embedding of a planar graph with unit edge weight resulting in the representation of vertices in the planar graph as points in a high-dimensional space. A Hamming or Manhattan distance between two points in the projected space corresponds to the network distance between the vertices in the original planar graph. In contrast to the RNE approach, the hypercube embedding is able to preserve exact distances between vertices in a planar graph.

To place the SILC framework in proper perspective, we view it as an extension to both the geometric framework of Wagner *et. al.* [36] and the ALT method [8] of Goldberg *et. al.*. Wagner *et. al.* in [36] compute and store a simple shape (geometric container) for each edge. The containers of the edges incident at a vertex may overlap. In contrast, our SILC method uses a complex geometric container with no overlap between containers. The geometric containers are represented as a set of Morton blocks, thereby enabling efficient storage and handling of containment queries. In contrast, the ALT method by Goldberg [8] and the RNE method by Shahabi [29] compute the distances between all vertices to a few landmarks. This is similar to computing the distances to all vertices and then randomly choosing a representative set. In contrast, our method stores the aggregation of the distances over a certain region, which is determined by the path representation.

7. CONCLUDING REMARKS

We have presented the SILC framework and have shown the applicability of traditional spatial techniques to spatial networks. In a future study, we plan to undertake a more elaborate theoretical study of the storage requirements of the proposed framework. We also plan to provide a detailed comparative study with other competitive techniques, and to investigate disk organization strategies for efficient path retrievals. Although we do not make any provisions for updates on spatial networks, a method similar to the one proposed in [18] could be used. Note that SILC allows for updates on the datasets of locations which makes it desirable for applications involving moving objects.

8. REFERENCES

[1] P. A. Burrough and R. A. McDonnell. *Principles of Geographical Information Systems*. Oxford University Press, New York, NY, USA, Apr. 1998.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[3] G. G. Filho and H. Samet. A linear iterative approach for hierarchical shortest path finding. Computer Science Department CS-TR-4417, University of Maryland, College Park, MD, Nov. 2002.

[4] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, June 1962.

[5] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987.

[6] H. Freeman. Computer processing of line-drawing images. *ACM Computing Surveys*, 6(1):57–97, Mar. 1974.

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

[8] A. V. Goldberg and C. Harrelson. Computing the shortest path: A^* search meets graph theory. In *SODA '05: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 156–165, Vancouver, BC, Canada, Jan. 2005.

[9] 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, BC, Canada, Jan. 2005.

[10] S. Gupta, S. Kopparty, and C. Ravishankar. Roads, codes, and spatiotemporal queries. In *PODS '04: Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 115–124, Paris, France, June 2004.

[11] 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, USA, June 1984.

[12] M. R. Henzinger, P. Klein, S. Rao, and S. Subramanian. Faster shortest-path algorithms for planar graphs. *J. Comput. Syst. Sci.*, 55(1):3–23, Aug. 1997.

[13] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *SSD '91: Proceedings of the 4th International Symposium on Large Spatial Databases*, pages 83–95, Portland, ME, USA, Aug. 1991.

[14] 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, USA, June 1998.

[15] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Trans. Database Syst.*, 24(2):265–318, June 1999.

[16] D. A. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. *Discrete Applied Mathematics*, 126(1):55–82, 2003.

[17] 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.

[18] 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.

[19] P. Micikevicius. General parallel computation on commodity graphics hardware: case study with the all-pairs shortest paths problem. In *PDPTA '04: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1359–1365, Las Vegas, NV, USA, June 2004.

[20] J. Mitchell, D. Mount, and C. Papadimitriou. The discrete geodesic problem. *SIAM J. Comput.*, 16(4):647–668, Aug. 1987.

[21] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *SIGGRAPH Computer Graphics*, 20(4):197–206, Aug. 1986.

[22] 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.

[23] S. Saltenis and C. S. Jensen. Indexing of moving objects for location-based services. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, pages 462–472, San Jose, CA, USA, Feb. 2002.

[24] H. Samet. A quadtree medial axis transform. *Commun. ACM*, 26(9):680–693, Sept. 1983.

[25] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, San Francisco, CA, USA, 2005.

[26] H. Samet, H. Alborzi, F. Brabec, C. Esperança, G. R. Hjaltason, F. Morgan, and E. Tanin. Use of the SAND spatial browser for digital government applications. *Commun. ACM*, 46(1):63–66, Jan. 2003.

[27] R. E. Schofer and F. F. Goodyear. Electronic computer applications in urban transportation planning. In *Proceedings of the 1967 22nd ACM National Conference*, pages 247–253, Washington, DC, USA, 1967.

[28] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: a dynamic index for multi-dimensional objects. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Databases*, pages 507–518, Brighton, England, Sept. 1987.

[29] 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.

[30] S. Shekhar and T. A. Yang. Motion in a geographical database system. In *SSD '91: Proceedings of the 2nd Symposium on Large Spatial Databases*, pages 339–358, Zürich, Switzerland, Aug. 1991.

[31] 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, USA, 2003.

[32] V. Surazhsky, T. Surazhsky, D. Kirsanov, S. J. Gortler, and H. Hoppe. Fast exact and approximate geodesics on meshes. *ACM Trans. Gr.*, 23(3):553–560, Aug. 2005.

[33] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, May 1999.

[34] U.S. Census Bureau. TIGER/Line Files, Census 2000. U.S. Census Bureau, Washington, DC, USA, Oct. 2001. `http://www.census.gov/geo/www/tiger/tiger2k/tiger2000.html`.

[35] U.S. Geological Survey. Major Roads of the United States. U.S. Geological Survey, Reston, VA, USA, 199911. `http://nationalatlas.gov/atlasftp.html`.

[36] D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *ESA '03: Proceedings of the 11th Annual European Symposium on Algorithms*, pages 776–787. 2003.

[37] D. Wagner and T. Willhalm. Drawing graphs to speed up shortest-path computations. In *ALENEX '05: Proceedings of the 7th Workshop on Algorithm Engineering and Experiments*, Vancouver, BC, Canada, Jan. 2005.

[38] 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, USA, June 1999.

[39] F. B. Zhan and C. E. Noon. Shortest path algorithms: An evaluation using real road networks. *Transportation Science*, 32(1):65–73, Feb. 1998.