Appeared in the *Proceedings of the ACM SIGMOD Conference*, Vancouver, BC, Canada, June 2008, pp. 43–54
(Best paper award winner)

Scalable Network Distance Browsing in Spatial Databases

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

ABSTRACT

An algorithm is presented for finding the k nearest neighbors in a spatial network in a best-first manner using network distance. The algorithm is based on precomputing the shortest paths between all possible vertices in the network and then making use of an encoding that takes advantage of the fact that the shortest paths from vertex u to all of the remaining vertices can be decomposed into subsets based on the first edges on the shortest paths to them from u. Thus, in the worst case, the amount of work depends on the number of objects that are examined and the number of links on the shortest paths to them from q, rather than depending on the number of vertices in the network. The amount of storage required to keep track of the subsets is reduced by taking advantage of their spatial coherence which is captured by the aid of a shortest path quadtree. In particular, experiments on a number of large road networks as well as a theoretical analysis have shown that the storage has been reduced from $O(N^3)$ to $O(N^{1.5})$ (i.e., by an order of magnitude equal to the square root). The precomputation of the shortest paths along the network essentially decouples the process of computing shortest paths along the network from that of finding the neighbors, and thereby also decouples the domain S of the query objects and that of the objects from which the neighbors are drawn from the domain V of the vertices of the spatial network. This means that as long as the spatial network is unchanged, the algorithm and underlying representation of the shortest paths in the spatial network can be used with different sets of objects.

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

spatial networks, nearest neighbor, shortest path quadtree, decoupling, scalability

1. INTRODUCTION

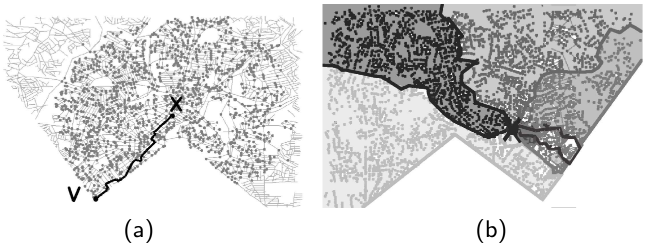
The growing popularity of online mapping services such as Google Maps and Microsoft MapPoint has led to an interest in responding in real time to queries such as finding shortest routes between locations along a spatial network as well as finding nearest objects from a set S (e.g., gas stations, markets, and restaurants) where the distance is measured in terms of paths along the network. Elements of S are usually constrained to lie on the network or at the minimum to be easily accessible from the network.

The online nature of these services means that responses must be generated in real time. For example, in Google Maps, once a shortest path from A to B has been obtained which passes through C, users can simply change the query to find the shortest path from A to B which is constrained to pass through D instead of C and the new shortest path is presented to the user instantly. Requiring that the result be obtained in real time (or almost real time) precludes the use of conventional algorithms that are graph-based (e.g., the INE and IER methods [21] and improvements on them [2]) which usually incorporate Dijkstra's algorithm [4] in at least some parts of the solution [25]. In particular, given a source vertex q (i.e., query vertex) and a connected graph G (i.e., the spatial network), Dijkstra's algorithm finds the shortest path (and hence the shortest distance along the network) to every vertex in the network where the paths are reported in order of increasing distance from q.

The problem with an approach that uses Dijkstra's algorithm is that it must visit every vertex that is closer to q via the shortest path from q than the vertices associated with the desired objects. Thus, the amount of work often depends on the number of vertices in the network whereas our goal is for the amount of work in the worst case to depend on the number of objects that are examined and on the number of links on the shortest paths to them from q. Thus, Dijkstra's algorithm may visit many vertices before reaching one which coincides with or is near one of the objects in which we are interested. In particular, it is not uncommon for Dijkstra's algorithm to visit a very large number of the vertices of the network in the process of finding the shortest path between vertices that are reasonably far from each other in terms of network hops. For example, Figure 1(a) shows the vertices that would be visited when finding the shortest path from the vertex marked by X to the vertex marked by V in a spatial network corresponding to Silver Spring, MD. Here we see that in the process of obtaining the shortest path from X to V of length 75 edges, 75.4% of the vertices in the network are visited (i.e., 3,191 out of a total of 4,233 vertices).

*Current Address: Google Inc., 4720 Forbes Avenue, Pittsburgh, PA 15213

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.
Copyright 2008 ACM 978-1-60558-102-6/08/06 ...$5.00.



(a) (b)

Figure 1: (a) A map of Silver Spring, MD, and the vertices, highlighted by circles, that are visited by Dijkstra's algorithm in determining the shortest path from X to V, and (b) its partition into regions r_i such that the shortest path from X to a vertex in r_i passes through the same vertex among the six vertices that are adjacent to X (i.e., the shortest-path map of X).

The algorithm that we describe satisfies our goals and is based on the observation that the spatial network is usually static (e.g., a road network) whereas the objects which are located on it are far more likely to change, or at least the domain from which the objects are drawn can change from query to query, while the underlying network does not. For example, the objects in S represent entities such as restaurants, hotels, gas stations, and so on. In fact, even if the domain from which the objects are drawn does not change, the values of the attributes of the objects may change (e.g., the type of food served in a restaurant or the price per gallon of gas at a gas station). Our algorithm is based on precomputing the shortest paths between all possible vertices in the network and then making use of an encoding that takes advantage of the fact that the shortest paths from vertex u to all remaining vertices can be decomposed into subsets based on the first edges on the shortest paths to them from u [27, 31], and represents the subsets using a shortest path quadtree which captures their spatial coherence. However, the algorithm does not use the actual distances and thus there is no need to store them. Experiments on a number of large road networks have shown that use of the shortest path quadtree leads to a significant reduction of the storage requirements from $O(N^3)$ to $O(N^{1.5})$ (i.e., by an order of magnitude equal to the square root).

The advantage of our algorithm is that it decouples the process of computing shortest paths along the network from that of finding the neighbors, and thereby also decouples the domain S of the query objects and that of the objects from which the neighbors are drawn from the domain V of the vertices of the spatial network. In other words, there is no need to recompute the shortest paths each time there are changes in q or S. This differentiates our approach from other approaches such as those proposed by Papadias et al. [21], as well as those of Cho and Chung [2], and Kolahdouzan and Shahabi [15], which must compute the shortest paths anew each time there are changes in q or S, which, unfortunately, may be quite frequent. Note though that Hu et al. [12] use a related approach to ours where for each vertex v of the spatial network T, they associate (1) rough distance estimates of the network distance from v to each object o in S and (2) the first link on the shortest path from v to o. However, the drawback of this approach is the lack of decoupling of the objects from the spatial network and the sheer volume of the data that must be stored for each pair (v,o), whereas our approach merely requires a spatial index for each different object set S, and one shortest path quadtree for each v.

The algorithm presented in this paper differs from the algorithm in [27] by being a k nearest neighbor algorithm rather than an incremental algorithm [27] which means that the k results need not be obtained in increasing order of network distance, and thus the storage requirements are also reduced especially for small values of k, which is the most common situation. It is also novel in being the first algorithm to make use of an estimate of the maximum of the network distance at which the kth nearest neighboring object can be found. Another contribution of this paper is the presentation of a detailed analysis and proofs of the storage requirements of this approach which involves more precise definitions of the underlying quadtree representations that enable it to achieve these results. Finally, we provide a detailed experimental evaluation as well as a comparison with related algorithms based on use of Dijkstra's algorithm. This experimental evaluation also demonstrates for the first time that use of the shortest path quadtree leads to a reduction of the storage requirements from $O(N^3)$ to $O(N^{1.5})$ (i.e., by an order of magnitude equal to the square root, which is quite substantial).

The rest of this paper is organized as follows. Section 2 presents our algorithm, while Section 3 analyzes its execution time and space requirements. Section 4 contains a detailed experimental evaluation of our algorithm and variants thereof as well as an experimental comparison with approaches based on Dijkstra's algorithm. Section 5 contains some concluding remarks and provides directions for future research.

2. BEST-FIRST K NEAREST NEIGHBOR ALGORITHM

Nearest neighbor finding is achieved by application of either a depth-first or a best-first algorithm. These algorithms are generally applicable to any index based on hierarchical clustering. The idea is that the data is partitioned into clusters which are aggregated to form other clusters, with the total aggregation being represented as a tree. The number k of neighbors that are sought is usually known in advance in which case the algorithms keep track of the set L of the k nearest neighbors found so far and update L as is appropriate. The most common strategy for nearest neighbor finding employs the depth-first *branch and bound* method (e.g., [6, 22]). The depth-first algorithm explores the elements of the search hierarchy in an order that is a result of performing a depth-first traversal of the hierarchy using the distance D_k from the query object q to the current k^{th}-nearest object to prune the search.

An alternative strategy is the best-first method (e.g., [9, 10]) which explores the nonobject elements of the search hierarchy in increasing order of their distance from q (hence the name "best-first"). This is achieved by storing the nonobject elements of the search hierarchy in a priority queue in this order. In addition, some of the best-first algorithms (e.g., [9, 10]) also store the objects in the same priority queue thereby enabling these algorithms to report the neighbors 1-by-1, and thus there is no need for k to be known in advance, as is the case in the depth-first approach, nor is there a need for L. This also enables the algorithms to halt once the desired number k of neighbors has been determined. On the other hand, variants can also be constructed that use L to keep track of the k nearest objects [10] as we do here.

The best-first approach's advantage is avoiding having to visit nonobject elements that will eventually be determined to be too far from q due to poor initial estimates of D_k, which is possible in the depth-first approach, thereby not needing to traverse the entire search hierarchy. On the other hand, the advantage of the depth-first approach over the best-first approach is that the amount of storage is bounded by k plus the maximum depth of the search hierarchy in contrast to possibly having to keep track in the priority queue of all nonobjects (and thus all the objects) if all their distances from q are approximately the same. Nevertheless, studies have shown the best-first approach to be better than the depth-first approach for k fixed [10], and the adaptation of the best-first approach to spatial networks is the subject of this paper.

In the rest of this section we describe the KNearestSpatialNetwork algorithm. It assumes that the underlying graphs that form the basis of the spatial networks are connected planar graphs. This is not an unreasonable assumption as road networks are connected (at least within a landmass such as a continent), although they don't have to be planar as can be seen by the possibility of the presence of tunnels and bridges. We do not dwell on such situations here although we do revisit it briefly in Section 5. It also assumes that the shortest paths between all pairs of vertices u and v in V in the graph $G = (V,E)$ have been computed using either Dijkstra's algorithm or any of the other approaches that have been proposed to do so that involve precomputation to speed up the process of shortest path computation (e.g., [5, 8, 14, 32] as well as the comparative study by Zhang and Noon [32]). Unfortunately, given a spatial network with N vertices, there are $O(N^2)$ possible paths and the cost of storing all possible shortest paths takes $O(N^3)$ space, which is prohibitive. Instead, we store partial information about each shortest path. In particular, we only store the identity of the first edge along the shortest path from source vertex u to destination vertex v, which enables the shortest path between u and v to be constructed in time proportional to the length of the path by repeatedly following the edges that make up the shortest path as they are discovered. The shortest paths from v to all remaining vertices can be decomposed into subsets based on the identity of the first edges to them from v and the decomposition of the underlying space that is induced by these subsets is stored in a shortest path quadtree which is discussed in Section 2.1, and which experimental results discussed in Section 4 lead to a reduction in storage costs from $O(N^3)$ to $O(N^{1.5})$. The workings of KNearestSpatialNetwork are presented in Section 2.2.

2.1 Shortest Path Quadtrees

The simplest way of representing the shortest path information in the manner described above is to maintain an array A of size $N \times N$ so that element $A[u,v]$ contains the first vertex on the shortest path from u to v. In this case, finding the shortest path reduces to retrieving the elements $A[u_i,v]$, where $u_1 = A[u,v]$ and, in general, $u_{i+1} = A[u_i,v]$. An alternative representation makes use of N adjacency lists, one for each vertex u_i. In particular, the adjacency list for vertex u_i is a set of M_{u_i} elements, where M_{u_i} is the out degree of u_i and there is one element for each vertex w_{u_ij} $(1 \leq j \leq M_{u_i})$ such that there exists an edge e_{u_ij} from u_i to w_{u_ij}. The element of the adjacency list corresponding to w_{u_ij} contains all vertices v whose shortest path from u_i passes through vertex w_{u_ij}. Note that we assume that the spatial network is connected, and thus every vertex is in one of the elements of the adjacency list of u_i. Moreover, we also assume that the shortest path from u_i to each vertex is unique, thereby making the elements of the adjacency list of u_i disjoint.

There are several drawbacks to the use of adjacency lists. The first is the absence of an index which means that searches through the elements of the list associated with vertex u_i for the one that contains v must make use of sequential search, which can be costly. The second is the space required for storing the lists as each list has $O(N)$ elements. The space requirements can be reduced by taking advantage of the fact that the vertices that are members of a particular element of an adjacency list have some spatial coherence in the sense that they are likely to be in close spatial proximity. This results in conceptually viewing the elements of each adjacency list as regions, and leads to replacing the adjacency list by a map, termed the *shortest-path map*, so that we have one shortest-path map for each vertex in the spatial network. In particular, given vertex u_i, the shortest-path map m_{u_i} partitions the underlying space into M_{u_i} regions, where M_{u_i} is the out degree of u_i and there is one region r_{u_ij} for each vertex w_{u_ij} $(1 \leq j \leq M_{u_i})$ that is connected to u_i by an edge e_{u_ij}. Region r_{u_ij} spans the space occupied by all vertices v such that the shortest path from u_i to v contains edge e_{u_ij} (i.e., the shortest path makes a transition through vertex w_{u_ij}). Region r_{u_ij} is bounded by a subset of the edges of the shortest paths from u_i to the vertices within it. Note that r_{u_ij} does not include u_i nor does it include edge e_{u_ij}. We assume that the spatial network is planar which means that the regions that make up m_{u_i} are disjoint (they are also shown to be connected in Section 3). For example, Figure 1(b) is such a partition for the vertex marked by X in the road network of Figure 1(a) where we use different colors (i.e., shades of gray) to denote the different regions.

The advantage of grouping the vertices on the basis of the regions in which they lie and identifying each region by the first vertex on the shortest path into it from vertex u_i is that we can make use of a point location operation to find the region that contains the destination vertex. This also means that we can find the shortest path to a group of vertices that form a region, which is not possible or easy when using the array or adjacency list representations, respectively. Point location is sped up by imposing a spatial index on the regions. In essence, there are two types of a spatial index: one based on an object hierarchy such as an R-tree and one based on a disjoint decomposition of the underlying space such as one of a number of quadtree variants (e.g., [25, 29]).

An object hierarchy is usually accompanied by a hierarchy of bounding boxes to facilitate execution of a point location query by enabling the filtering of obviously wrong results. The bounding boxes result in a nondisjoint decomposition of the underlying space which means that the location occupied by a particular vertex may be contained in several bounding boxes. Thus, given a source vertex u_i and a destination vertex v, the only way to determine the actual bounding box b_{u_ij}, and hence the region r_{u_ij} corresponding to the first vertex on the shortest path from u_i to v, is to associate the relevant vertices with b_{u_ij} which defeats the rationale for not using the adjacency list method. The alternative is to have as many choices for the first vertex on the shortest path to v as there are bounding boxes that contain v. This has the effect of making the process of obtaining the actual shortest path from u_i to v considerably more expensive as it can no longer be determined in time proportional to the number of edges that make up the path. The result is that we are actually making use of a concept similar to the landmarks employed by several researchers (e.g., [8, 14, 30]) as an alternative to Dijkstra's algorithm to compute the shortest path between two vertices. In fact, this is indeed the motivation for the method of Wagner and Willhalm [31] where the object hierarchy consists of bounding boxes. Figure 2(a) shows the result of using minimum bounding boxes to approximate the regions in the partition for the vertex marked by X in the road network of Figure 1(a). Notice that the bounding boxes intersect, which means that vertices in the intersecting regions have more than one candidate next vertex for the shortest path to them from X.

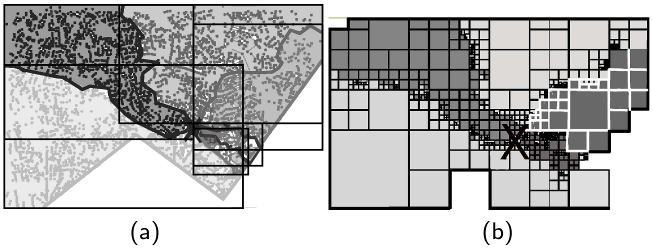


Figure 2: (a) Result of using minimum bounding boxes to approximate regions in the partition for vertex X in the road network of Figure 1(b), and (b) leaf blocks in the shortest-path quadtree for regions of the same partition.

In contrast, Sankaranarayanan et al. [27] propose the use of a spatial index based on a disjoint decomposition of the underlying space. In particular, they represent the regions that make up the shortest-path map m_{u_i} using a variant of the region quadtree [25], termed a *shortest-path quadtree*, where there are M_{u_i} different disjoint regions $r_{u_i j}$ all stored in the region quadtree s_{u_i}. Each region $r_{u_i j}$ consists of the disjoint quadtree blocks that make it up. Each of the quadtree blocks records the identity of the region of which it is a member. For example, Figure 2(b) is the block decomposition induced by the shortest-path quadtree on the shortest-path map given by Figure 1(b). As we pointed out earlier, the advantage of representations that make use of a disjoint decomposition of the underlying space, such as the region quadtree, is that once we locate the block containing the destination vertex, we know what region it is in and hence the edge emanating from the vertex whose shortest-path quadtree we are processing. In particular, given source vertex u, destination vertex v, the shortest-path map m_u and shortest-path quadtree s_u associated with u, the next vertex t in the shortest path from u to v is the vertex w_j associated with the quadtree block of s_u in region r_j of m_u that contains v. The complete path from u to v is obtained by repeating the process, successively replacing u with t and replacing u's shortest-path quadtree with that of t, until u equals v.

For example, consider the simple road network given in Figure 3(a) where we want to find the shortest path from vertex s to vertex d, and the shortest-path quadtree for s is given by Figure 3(b). Looking up vertex d in the shortest-path quadtree of s determines that d is in the region of the quadtree corresponding to the edge from vertex s to t. Therefore, the shortest-path from s to d passes through t. Next, we obtain the shortest-path quadtree of t which is given by Figure 3(c). Looking up vertex d in the shortest-path quadtree of t determines that d is in the region of the quadtree corresponding to the edge from vertex t to u. This process is continued until encountering an edge to vertex d.

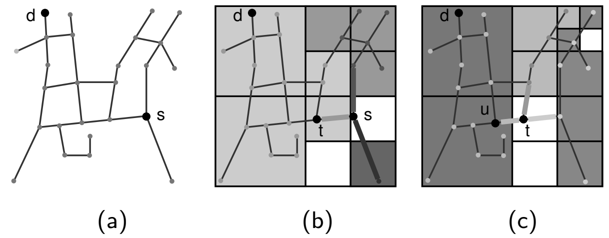


Figure 3: (a) Example road network, (b) the shortest-path quadtree of vertex s, and (c) the shortest-path quadtree of vertex t.

Although the idea of storing the shortest-path map as a shortest-path quadtree is conceptually simple, care must be taken in defining it. The most straightforward approach is to partition the underlying space into blocks so that each block is associated with just one region of the shortest-path map. The difficulty with this approach is that it presumes that we know the boundaries of the regions, which, as we will soon see, may not be worth the effort to compute. Of course, we can determine the boundaries but even if we do this, we still need to decide how to build an appropriate quadtree for the regions. For example, boundaries of the regions could be represented by a variant of an MX quadtree [13, 25] where boundary blocks would be treated no differently than the interior of the region that they bound. This is in contrast with the conventional MX quadtree where boundary blocks are viewed as being distinct from the interiors of the regions that they bound.

Therefore, instead, we adopt the following approach that assumes that all vertices have been assigned a color corresponding to the vertex $w_{u_i j}$ incident at the source vertex u_i through which the shortest path to them from u_i passes. We now recursively decompose the underlying space into blocks and halt whenever all vertices in the block have the same color. The fact that the shortest-path quadtree is built by decomposing on the basis of the presence and absence of vertices of the spatial network may result in some empty blocks, which are assigned an unused color (e.g., white). This has the side effect that it is possible for regions of a given color to be noncontiguous due to intervening white blocks, thereby resulting in more contiguous regions than the outdegree of the vertex with which the shortest-path quadtree is associated. However, as we discuss in Section 3, this is not really an issue for us as it does not affect the efficiency of the point location algorithm. In fact, there is really no need to keep track of the white blocks, and thus we use a pointerless quadtree representation that only keeps track of the nonempty leaf blocks (e.g., [7]). In this case, each of these nonempty blocks is represented by its locational code (i.e., a number formed by the concatenation of its size and the path to it from the root). Blocks that are represented in this way are known as *Morton blocks* [18], and access to a collection of such blocks is facilitated by making use of a B$^+$-tree access structure based on the values of their locational codes. Lesser space savings are achieved by not dispensing with all of the nonleaf blocks by using a variant of a path-compressed PR quadtree (e.g., [3]) which ignores white blocks where all but one of the siblings are white.

2.2 Best-first *k* Nearest Neighbor Algorithm

Given the shortest path quadtree representation of a spatial network, we can trivially obtain the shortest path between any source and destination pairs in real time. Similarly, other queries such as range and region searches can also be easily handled using the shortest path quadtree representation. We are interested in the k nearest neighbor algorithm on spatial networks as it has important applications to the provision of location-based services (e.g., "Google Local" and "Microsoft Live"). For example, suppose we want to "find the 10 closest restaurants to 5600 Broadway St., Manhattan". Note however that neither "Google Local" nor "Microsoft Live" are presently able (at least not yet) to calculate the actual network k neighbors to a query object in real time, and end up using Euclidean distance between two objects u, v as an approximation to the actual network distance between u and v. In the rest of this section, we describe KNEARESTSPATIALNETWORK which works in real time on a spatial network.

KNEARESTSPATIALNETWORK assumes the existence of a search hierarchy T (i.e., a spatial index) on a set of objects S (usually points in the case of spatial networks) that make up the set of objects from which the neighbors are drawn. For the sake of this discussion, we assume that S, as well as the set of query objects Q, is a subset of the vertices of the spatial network, although it is easy to modify it to handle the more general case by keeping track of two shortest paths to an object instead of just one.

In order to enable the computation of the range of network distances from query object q for the shortest paths that pass through Morton block b, KNEARESTSPATIALNETWORK stores some additional information with b. In particular, for a Morton block b in the shortest-path quadtree (i.e., s_q) for the shortest-path map m_q, it stores a pair of values, λ^- (λ^+), that correspond to the minimum (maximum) value of the ratio of the network distance (i.e., through the network) to the actual *spatial distance* (i.e., "as the crow flies") from q to all destination vertices in b. The ratios are computed on a vertex-by-vertex basis—that is, a ratio is computed for each destination vertex after which the minimums and maximums are computed. Thus the destination vertex for which the ratio attains its minimum value does not have to be the same as the destination vertex for which the ratio attains its maximum value.

At this point, let us elaborate on how the shortest-path quadtree is used to compute network distances. In particular, we first show how to compute the network distance between a query vertex q and a destination vertex v. We start by finding the block b in the shortest-path quadtree of q (i.e., s_q) that contains v (i.e., a point location operation). By multiplying the λ^- and λ^+ values associated with b by the spatial distance between q and v, we obtain an interval $[\delta^-,\delta^+]$, termed the *initial network distance interval*, which contains the range of the network distance between q and v. These two actions are achieved by procedure GetNetworkDistInterval (not given here). Whenever it is determined that the initial network distance interval $[\delta^-,\delta^+]$ is not sufficiently tight (i.e., where tightness means that the interval does not intersect an interval associated with another neighboring object), an operation, termed *refinement*, is applied that obtains the next vertex t in the shortest path between q and v using procedure NextVertexShortestPath (not given here). Having obtained t, we retrieve the shortest-path quadtree s_t for t and then calculate a new network distance interval $[\delta_t^-,\delta_t^+]$ by locating the Morton block b_t of s_t that contains v. The network distance interval of the shortest path between q and v is now obtained by summing the network distance from q to t (i.e., the weight of the edge from q to t) and $[\delta_t^-,\delta_t^+]$. Given a pair of vertices q and v and a length k in terms of the number of vertices on the shortest path between them, this process is reinvoked at most another $k-2$ times until reaching v.

We now show how to compute the network distance between a query vertex q and a block b of the search hierarchy T. First, we point out that in the case of a block, the concept of a network distance is complicated by the fact that there are usually many vertices of the spatial network in the area spanned by b, and thus we need to specify somehow the vertex (vertices) for which we are computing the network distance. Instead, we compute a minimum network distance for the block using procedure MinNetworkDistBlock (not given here). The minimum possible network distance δ^- of q from b is computed by intersecting b with s_q, the shortest-path quadtree of q, to obtain a set of intersecting blocks B_q of s_q. For each element b_i of B_q, the associated λ_i^- value is multiplied by the corresponding $\text{MinDist}(q,b_i\cap b)$ value to obtain the corresponding minimum shortest-path network distance μ_i^- from q to b_i. δ^- is set to the minimum value of μ_i^- for the set of individual regions specified by $b_i\cap b$. Note that the reason that block b can be intersected by a varying number of blocks b_i of B_q is that s_q and T need not be based on the same data structure (e.g., T can be an R-tree), and even if they are both quadtree-based (e.g., T is a PR quadtree [20, 25]), s_q and T do not have to be in registration (i.e., they can have different origins, as can be seen in Figure 4).

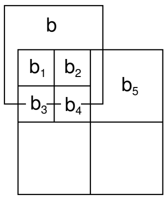


Figure 4: Example of the intersection of block b in a quadtree search hierarchy T with blocks b_1, b_2, b_3, b_4, b_5 in the shortest-path quadtree.

There are several ways of implementing a best-first k nearest neighbor algorithm. The simplest is to use the spatial network best-first incremental nearest neighbor algorithm [27] and terminate it once it has reported the first (i.e., nearest) k objects. This approach makes use of a priority queue *Queue* that is initialized to contain the root of the search hierarchy T and the root's network distance from the query object q. The principal difference between the spatial network adaptation of the incremental nearest neighbor algorithm and the conventional incremental nearest neighbor algorithm is that, in the case of a spatial network, objects are enqueued using their network distance interval (i.e., $[\delta^-,\delta^+]$) from the query object q, instead of just their minimum spatial distance from q. However, objects and blocks are ordered and removed from *Queue* in increasing order of their minimum network distance from q.

The drawback of this incremental approach is that the priority queue can be as large as the number of objects in the spatial network should they all be at approximately the same distance from q [10]. Our best-first k nearest neighbor algorithm given by procedure KNearestSpatialNetwork overcomes this by using the distance D_k from q of the kth candidate nearest neighbor o_k to reduce the number of needless priority queue insertions operations by enabling us to avoid enqueueing elements with a distance greater than or equal to D_k from q (lines 57 and 66) which would never be removed from *Queue* since the bound k on the number of neighbors means that the algorithm terminates by then. However, such a modification incurs the cost of additional complexity in the algorithm due to the need to check for it whenever insertions are made into *Queue*. In particular, knowing o_k means that we must keep track of the set L of k candidate nearest objects that have been encountered at any moment. Moreover, whenever it is determined that an insertion is to be made into L, we must be able to identify and remove the element in L with the largest distance. This is done most easily by implementing L as a priority queue that is distinct from *Queue*, which now contains the remaining types of elements. Thus, finding the k nearest neighbors makes use of two priority queues.

Since the process of finding the nearest k neighbors relies on estimating the network distance of the objects from q, objects cannot be inserted into L until their exact distances are known (i.e., they have been fully refined). However, this means that the convergence of D_k from its initial value of ∞ to its final value cannot begin to take place until k of the objects have been fully refined. This can take quite a bit of time. In order to speed up the convergence of D_k, and hence reduce the potential size of the priority queue *Queue*, we modify the definition of L so that L also stores partially refined objects (as does *Queue*). In this case L also keeps track of the maximum of their associated network distance intervals (see [25] where such an approach is used in a conventional non-network k nearest neighbor algorithm to keep track of the maximum possible distance at which a nearest neighbor can be found). In particular, given object p with distance interval $[\delta_p^-,\delta_p^+]$, L stores the pair (p,δ_p^+)) when the network distance value of p from q is less than or equal to D_k. Note that *Queue* also stores partially refined objects with the difference that they are stored in *Queue* with their corresponding network distance interval, while they are only stored in L with the maximum of their corresponding network distance interval.

The actual mechanics of the algorithm are similar to the general conventional best-first algorithm with the difference that objects are associated with distance intervals instead of distances. When a nonleaf block b is removed from *Queue*, the minimum network distance is computed from q to each of the children of b, and they are inserted into *Queue* with their corresponding minimum network distances. When a leaf block b is removed from *Queue*, the objects (i.e., points) in b are enqueued with their corresponding initial network distance intervals, which are computed with the aid of the λ^- and λ^+ values associated with b.

On the other hand, when the algorithm processes an object t (i.e., when the most recently removed element from *Queue* corresponds to an object), it determines if the minimum network distance δ_t^- of t is greater than or equal to that of D_k (the current distance of the kth nearest neighbor of q), in which case it exits and returns L as the set of k nearest neighbors because t and all other objects in *Queue* or in blocks in *Queue* cannot be found at a distance from q which is less than D_k. Otherwise, it checks to see if the maximum network distance δ_t^+ of t is less than the minimum network distance δ_p^- of the element p that is currently at the top of *Queue*. In this case, further processing of t is halted and processing continues of p as by Theorem 1 (given the end of this section) we can guarantee that $D_k \geq \delta_t^+$ which means that t is one of the k nearest neighbors of of q (otherwise we would need to refine t and enqueue it with the refined distance interval). If $\delta_t^+ \geq \delta_p^-$, then the algorithm attempts to tighten the network distance interval for t by applying one step of the refinement operation described earlier, and then enqueues t with the updated network distance interval. Note that when the network distance intervals associated with an object p in *Queue* have been obtained via refinement, *Queue* must also keep track of the most recently determined intermediate vertex v on the shortest path from q to t and the network distance d from q to v along this path. Observe also that no such information need be recorded for blocks, and, in fact, each time we process a block, its associated intermediate vertex and minimum network distance are the query object q and 0, respectively.

In order to avoid having duplicate entries in L for a particular partially refined object, each time a partially refined object is removed from *Queue* for processing, we also attempt to remove it from L (line 32), if it is there (i.e., the value of the maximum of its corresponding distance interval is less than or equal to D_k), using procedure RemovePriorityQueue (not given here). Similarly, once its network distance interval has been refined, we attempt to insert it into L with its associated maximum network distance provided that this value is less than or equal to D_k (line 43) using procedure InsertL (not given here) which also updates D_k if necessary (i.e., if L contains k elements). However, we do not enqueue it in *Queue* (line 46) if the value of its associated minimum network distance is greater than or equal to D_k as this means that its further processing will not result in a closer neighbor. Note that when the minimum and maximum network distance values are equal to D_k, such an action results in the object o being in L while no longer being in *Queue* (lines 41–46) which is allowed as this means that there is no longer a need to refine o further. Of course, if subsequently closer objects to q are found than o at network distances less than D_k, then o will be removed implicitly from L.

Procedure InsertL makes use of procedure MaxPriorityQueue (not given here) to determine the element of a priority queue with the maximum distance. MaxPriorityQueue is equivalent to FrontPriorityQueue when priority is given to elements at a maximum distance. Note that InsertL is also invoked when we first encounter an object as part of a leaf block (line 59).

It is important to note that procedure KNearestSpatialNetwork takes advantage of the fact that for a given object o, there is no need to refine its distance further once it is known that the maximum network distance associated with o is less than the minimum network distance associated with other objects. This means that when the algorithm terminates, the set L does not necessarily contain the actual network distance from q of all of its constituent objects. In other words, the identity and relative ranking (see Theorem 2 at the end of this section) of the k nearest neighbors of q is known, but their distance from q is not known. All that is known are upper bounds on their distance from q. This is the price that we pay for not refining the distances but it does result in a faster convergence to the desired goal of finding the k nearest neighbors. Of course, if the actual distances are desired for some of the k nearest neighbors, then the algorithm can be modified to store in L the identity of the intermediate vertex t on the path from q to neighbor p (and the distance s from q to t) at the time at which the refinement process for p was halted and then simply perform repeated lookup operations on the shortest path quadtree to obtain the remaining shortest path to p and the distance to it. Note also that if there are several objects at the maximum distance from q, then we only report as many as necessary rather than all of them, which could possibly result in reporting more than k objects.

```
 1 procedure KNearestSpatialNetwork($q,k,S,T$)
 2 /* A best-first nonincremental algorithm that returns in priority queue
      $L$ the $k$ nearest neighbors of $q$ from a set of objects $S$ on a spatial
      network. $S$ is organized using the search hierarchy $T$. It assumes
      that each element in the priority queue Queue has four data fields E,
      D, V, and I, corresponding to the nature of the entity $x$ that Queue
      contains (which can be an object, leaf block, or nonleaf block), the
      network distance interval of $x$ (just one value if $x$ is not an object),
      the most recently determined vertex $v$ via refinement when $x$ is an
      object, and the network distance from $q$ to $v$ along the shortest path
      from $q$ to $x$ when $x$ is an object. Note that Enqueue takes four
      arguments when the enqueued entity is an object instead of the usual
      two. In both cases, the field names are specified in its invocation. */
 3 value object $q$
 4 value integer $k$
 5 value object_set $S$
 6 value pointer search_hierarchy $T$
 7 integer $D_k$
 8 priority_queue $L$, Queue
 9 object $o$
10 vertex $v$
11 interval $i$
12 real $s$
13 pointer search_hierarchy $e, e_p$
14 priority_queue_entry $t$
15 $L \leftarrow$ NewPriorityQueue()
16 /* $L$ is the priority queue containing the $k$ nearest objects */
17 Queue $\leftarrow$ NewPriorityQueue()
18 $e \leftarrow$root of the search hierarchy induced by $S$ and $T$
19 Enqueue([E =]$e$, [D =]0, Queue)
20 $D_k \leftarrow \infty$
21 while not IsEmpty(Queue) do
22   $t \leftarrow$ Dequeue(Queue)
23   $e \leftarrow$ E($t$)
24   if IsObject($e$) then /* $e$ is an object */
25     if MinNetworkDistInterval(D($t$)) $\geq$ $D_k$ then
26       return $L$
27     elseif MaxNetworkDistInterval(D($t$))
28          $\geq$ MinNetworkDistInterval(
29               D(FrontPriorityQueue(Queue))) then
30       if MaxNetworkDistInterval(D($t$)) $\leq$ $D_k$ then
31         /* Ensure one entry/object in $L$ */
32         RemovePriorityQueue($e, L$)
33       endif
34       $v \leftarrow$ NextVertexShortestPath(
35            $e$, ShortestPathQuadtree(V($t$)))
36       /* NextVertexShortestPath does point location on $e$
            in the ShortestPathQuadtree of V($t$) and returns the
            vertex $v$ associated with the block or region containing V($t$)
            */
37       $s \leftarrow$ I($t$) + EdgeWeight(V($t$), $v$)
38       /* EdgeWeight(V($t$), $v$): distance between V($t$) and $v$ */
39       $i \leftarrow s +$ GetNetworkDistInterval(
40               $e$, ShortestPathQuadtree($v$))
41       if MaxNetworkDistInterval($i$) $\leq$ $D_k$ then
42       /* Update $L$ and $D_k$ as necessary */
43         InsertL($e$, MaxNetworkDistInterval($i$), $k, L, D_k$)
44       endif
45       if MinNetworkDistInterval($i$) $< D_k$ then
46         Enqueue([E =]$e$, [D =]$i$, [V =]$v$, [I =]$s$, Queue)
```

```
      endif
    endif
  elseif $D(t) \geq D_k$ then /* $e$ is a non-object */
    return $L$
  elseif IsLeaf($e$) then /* $e$ is a leaf block */
    foreach object child element $o$ of $e$ do
      /* Insert each object $o$ in $e$ in Queue along with the network dis-
      tance interval of $o$, which is obtained by performing a point
      location operation for the block containing $o$ in the shortest-
      path quadtree of $q$. In addition, insert each object $o$ in $L$ for
      which the maximum distance from $q$ is less than $D_k$. */
      $i \leftarrow$ GetNetworkDistInterval(
          $o$, ShortestPathQuadtree($q$))
      if MinNetworkDistInterval($i$) $< D_k$ then
        Enqueue([E =]$o$, [D =]$i$, [V =]$q$, [I =]0, $Queue$)
        if MaxNetworkDistInterval($i$) $< D_k$ then
          InsertL($o$, MaxNetworkDistInterval($i$), $k$, $L$, $D_k$)
        endif
      endif
    enddo
  else /* $e$ is a nonleaf block */
    foreach child element $e_p$ of $e$ do
      if MinNetworkDistBlock($q$, $e_p$) $< D_k$ then
        Enqueue([E =]$e_p$,
                 [D =]MinNetworkDistBlock($q$, $e_p$),
                 $Queue$)
      endif
    enddo
  endif
enddo
```

We now state a pair of theorems, whose proofs are omitted for lack of space, that are needed in the demonstration of the correctness of procedure KNearestSpatialNetwork.

Theorem 1. *If the maximum of the distance interval associated with the most recently removed element t from Queue is less than the minimum of the distance interval associated with the element p currently on the top of the Queue (i.e., $\delta_t^+ < \delta_p^-$), then D_k is always greater than or equal to the maximum of the distance interval associated with t or formally $D_k \geq \delta_t^+$, which implies that t is one of the k nearest neighbors of q.*

Theorem 2. *The output of* KNearestSpatialNetwork *is a total ordering of the set of k nearest neighbors of q, even though it is possible that their distance intervals were not fully refined.*

3. EXECUTION TIME AND SPACE REQUIREMENTS

In this section we analyze the execution time and space requirements of the IncNearestSpatialNetwork algorithm. The execution time requirements of the algorithm are quite simple and are captured by the following theorem.

Theorem 3. *The worst case execution time of the* IncNearestSpatialNetwork *algorithm is proportional to the number of objects examined and the number of links on the shortest paths to them from the query object q.*

Proof. This is proved easily by noting that the algorithm performs a sequence of point location operations to locate the vertices of the network that coincide with the positions of the objects. The number of blocks in the search hierarchy T is proportional to the number of objects in the search hierarchy. In the worst case, the algorithm retrieves all of the blocks, and, in the worst case, all of the shortest paths to the objects within them are explored. However, only these paths are explored. The worst case of the algorithm arises when all nonleaf blocks of the search hierarchy are at approximately the same distance from q, which is the worst case of the conventional best-first incremental nearest neighbor algorithm [10]. Therefore, in the worst case, the number of point location operations is equal to the sum of the number of links in the shortest paths from q to al of the objects in the spatial network. Of course, such a worst case scenario (i.e., the retrieval of all objects) will rarely exist as it depends on a particular positioning of q and the objects being equidistant from it. Note that the complexity of the point location operation itself is just the depth of the search hierarchy which can be treated as a constant (i.e., the resolution of the underlying decomposition space). □

As we pointed out, the bulk of the storage is needed to store the shortest-path quadtrees. Before obtaining the actual bound, we first prove that the regions of the shortest-path map are connected.

Theorem 4. *The regions that make up the shortest-path map m_{u_i} of vertex u_i are connected.*

Proof. This is proved easily by noting that from the point of view of a graph, ignoring the spatial embedding of its vertices, all vertices that make up each of the regions $r_{u_i j}$ are connected. Therefore, the only way that the space spanned by one of these regions associated with vertex w_1 incident at u_1 can be disconnected, say consisting of two regions g_1 and g_2, is if the shortest path from u_1 to some vertex v_2 in g_2 would "jump" from some vertex v_1 in g_1 over some region that is associated with a vertex w_2 incident at u_1 which is impossible as the spatial network is planar. □

Theorem 5. *The shortest-path quadtree for vertex u_i requires $O(p_{u_i}+n)$ space, where p_{u_i} is the sum of the perimeters of the polygons corresponding to the regions that make up the shortest-path map of u_i and the map is embedded in a $2^n \times 2^n$ space.*

Proof. The shortest-path map m_{u_i} partitions the underlying space into M_{u_i} regions, where M_{u_i} is the out degree of u_i and there is one region $r_{u_i j}$ for each vertex $w_{u_i j}$ ($1 \leq j \leq M_{u_i}$) that is connected to u_i by an edge $e_{u_i j}$. From Theorem 4 we know that each of $r_{u_i j}$ is connected. Now, for each region $r_{u_i j}$ of u_i, apply an algorithm to determine its boundary which results in a polygon $o_{u_i j}$ and build an MX quadtree $t_{u_i j}$ for its edges. Assuming that $t_{u_i j}$ is embedded in a $2^n \times 2^n$ space, we know from the Quadtree Complexity Theorem (e.g., [13, 25]) that $t_{u_i j}$ requires $O(p_{u_i j}+n)$ space, where $p_{u_i j}$ is the perimeter of $o_{u_i j}$ (also known as the dimension reducing property). Next, construct X_{u_i}, the union of the MX quadtrees corresponding of the regions that make up m_{u_i} which will require $O(p_{u_i}+n)$ space, where p_{u_i} is the sum of the perimeters of the polygons corresponding to the regions that make up m_{u_i}.

As we saw in Section 2, Sankaranarayanan et al. [27] make use of another representation of the shortest-path quadtree which we call S_{u_i}. S_{u_i} is built by processing the shortest-path map m_{u_i} directly and recursively decomposing the underlying space that it spans into blocks and halting the decomposition process whenever all vertices in the block have the same color (i.e., a variant of the region quadtree). It is easy to see that this decomposition rule results in no more blocks than the MX quadtree X_{u_i} as all vertices that are in the interior of one of the regions of m_{u_i} remain in interior blocks of both the quadtree blocks of the appropriate $t_{u_i j}$ and the corresponding blocks of X_{u_i} and S_{u_i}. However, for blocks that are on the boundaries of regions, in the case of the shortest-path quadtree S_{u_i}, there is no need to decompose the underlying space to the pixel level. Therefore, we only need to ensure that the vertices lie in separate blocks rather than also to ensure that the edges that connect them lie in separate blocks. In other words, region boundaries are represented implicitly in S_{u_i} in contrast to being represented explicitly in the MX quadtree X_{u_i}. Thus, the shortest-path quadtree

S_{u_i} requires no more space than the MX quadtree,X_{u_i}, and therefore the $O(p_{u_i}+n)$ space requirements of the MX quadtree X_{u_i} also hold for the shortest-path quadtree S_{u_i}. □

We now prove the main result.

THEOREM 6. *Assuming a spatial network embedded in a square grid so that each vertex occupies a random position within a grid cell and that the boundaries forming the regions in the shortest path quadtrees are monotonic, the total number of quadtree leaf blocks in the shortest path quadtrees for a spatial network with N vertices is* $O(N^{1.5})$.

PROOF. Embedding the N vertices in a square grid implies that the grid width is $\sqrt{N}$ grid cells. Assuming an outdegree of c per vertex (c is usually much smaller than N for a spatial network corresponding to a road network for which c is usually 4 as the vertices usually represent the intersection of two roads), the shortest path map has just c polygonal regions. From Theorem 5 in Section 3 we have that the space complexity of the shortest path quadtree corresponding to the shortest path map is proportional to the sum of the perimeters of the polygons that make up the shortest path map. We now observe that the digitization using Bresenham's algorithm [1] of the line segments that make up the monotonic boundaries of the polygons of the shortest path map means that the sum of their lengths (i.e., perimeters) are no more than c times the length of the width $\sqrt{N}$ of the embedding space. Therefore, the space required by the N shortest path quadtree for the spatial network of N vertices is $O(N^{1.5})$. □

It should be clear that there are many possible quadtree variants that could have been used to represent the shortest path map m_{u_i}. In the proof of Theorem 5, we used the MX quadtree X_{u_i} because of the way in which its space requirements can be obtained. The actual implementation of the shortest-path quadtree using S_{u_i} has a lower number of blocks, but a formal derivation of a more precise estimate is more complex. In any case, experiments with some actual map data such as the Silver Spring map given in Figure 1(a), which has 4333 vertices, found that, using S_{u_i}, the number of blocks in each of the shortest-path quadtrees for all vertices in the map ranged between 1 and 538 with an average of 128.3. This number is significantly smaller than $N = 4333$ which is what we would need had we we used adjacency lists.

An alternative quadtree representation can be obtained [23, 24, 28] after converting the collection of polygons described in the proof of Theorem 5 to a polygonal map where the edges of the individual polygons o_{u_ij} that border adjacent polygons are merged into one edge. The result can be represented using an MX quadtree, which of course, will require less space than X_i as there are fewer edges to decompose. However, the order of the space complexity will still be the same. An alternative which will require even less space is to use one of the members of the PM quadtree family [11, 26] or even the PMR quadtree [19]. Their space requirements have been analyzed in [17] where the space requirements of the PMR quadtree has been shown to be on the order of the number of edges making up the polygonal subdivision and independent of the depth of the quadtree (i.e., the resolution of the underlying space). Note that in order to use these structures, we would have to determine the actual polygons that correspond to the regions of the shortest-path map as outlined in the proof of Theorem 5.

One of the interesting aspects of implementing the shortest-path quadtree using S_{u_i} is that the resulting quadtree may have some white (i.e., empty) blocks as can be seen in Figure 5. This occurs when a nonleaf quadtree block contains vertices from different regions of the shortest-path map. In this case, it could be said that the number of regions has increased if we also count the white (i.e., white disconnected regions). Furthermore, it is possible that the quadtree blocks that make up the M_{u_i} regions in the shortest-path map m_{u_i} are not contiguous, at least if contiguity is based on 4-adjacency. The example in Figure 5 shows the shortest-path quadtree for query object q which consists of two regions, one for vertex a consisting of the noncontiguous quadtree blocks containing vertices a and d, and one for vertex b consisting of the noncontiguous quadtree blocks containing vertices b and c. It is important to observe that the complexity bound obtained in Theorem 5 in terms of the perimeters of the regions comprising the shortest-path map is not formulated in terms of the regions formed by the quadtree blocks that make up S_{u_i}. Moreover note that these additional regions (i.e., those comprised of the white blocks and the noncontiguous 4-adjacent regions corresponding to the various r_{u_ij}) have no effect on the efficiency of the algorithm that determines the shortest paths in the incremental nearest neighbor process as these white regions contain no vertices and thus they are never accessed during the point location process which is the key to finding the segments that form the shortest paths.

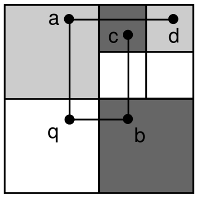


Figure 5: Example illustrating the presence of empty blocks in the shortest-path quadtree of the shortest-path map of query object q consisting of two regions: one for vertex a consisting of the noncontiguous quadtree blocks containing a and d, and one for vertex b consisting of the noncontiguous quadtree blocks containing b and c.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our k-nearest neighbor algorithm and a number of its variants. We also compare them with two competing techniques—INE and IER of Papadias et al. [21] that are based on the use of Dijkstra's algorithm. They differ on the extent to which they make use of Dijkstra's algorithm where INE uses it to find the neighbors as the graph is explored while IER first finds the neighbors using Euclidean distance and then uses Dijkstra's algorithm to find the shortest paths to them and hence the true network distance and then possibly seeks additional neighbors [25]. All 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 shortest path quadtree of the vertices of this road network was precomputed and stored on disk. The average number of the Morton blocks in the shortest path quadtree associated with each vertex in the dataset is 353. The algorithm uses an LRU based cache that can hold 5% of the disk pages in the main memory.

We now briefly describe our experimental setup. We randomly generated a set of objects S, which is indexed by a disk-based PMR quadtree in all of the algorithms that we tested (it was also used by the *find_entities* function in the INE method [21]). Even though, our algorithm can handle objects in S that lie on an edge or a face of a spatial network with equal ease, for the sake of simplicity, we assume that each of the objects in S is associated with a vertex on the road network. We represent the size of S as a fraction of N, the number of the vertices in the spatial network. We vary the size of S between $0.001N$ to $0.2N$. Moreover, in order to reduce some of the mathematical instabilities involved in using statistics derived from a random input dataset, we used the averages recorded by running the queries on at least 50 random input datasets of the same size.

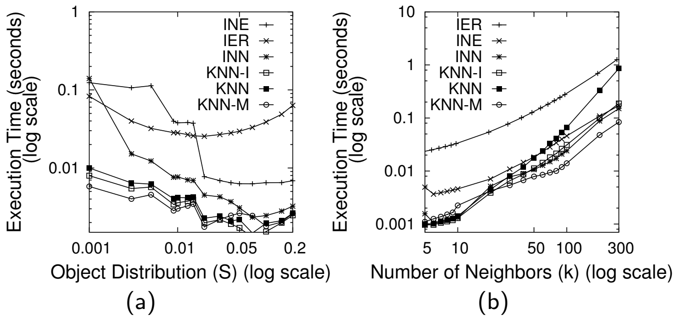


Figure 6: Comparison of KNN and its variants with INE and IER for (a) $k = 10$ and varying sizes of S, (b) $S = 0.07N$ and varying k.

The first, and most important, experiments were designed to compare KNearestSpatialNetwork (termed "KNN") and a number of its variants (KNN-M and KNN-I, as well as INN which simply invokes KNN k times and hence has no need for priority queue L and D_k is irrelevant as it is set to ∞) that are described and evaluated in greater detail in the rest of this section, with the IER and INE techniques of Papadias et al. [21] which are based on the use of Dijkstra's algorithm. These experiments are important as they shed light on the fundamental goal of this paper which is to demonstrate the efficacy of precomputing the shortest paths between the various nodes in the spatial network so that the complexity of the nearest neighbor process does not depend on the size of the underlying spatial network (i.e., the decoupling principle). We used the INE algorithm presented in [21] as in the interest of simplicity we assumed that each of the objects in S from which the neighbors are drawn is associated with a vertex. Without this assumption, in order to obtain the right result, this variant would need the modifications described in [25], which had the effect of doubling the execution time of INE (although not shown here). Figure 6 shows the execution time taken by KNN and its variants as well as INE and IER for varying values of k and S. We speak of the behavior of KNN and its variants collectively as they all outperform INE and IER for small values of k, which is the most common case in which these algorithms are used.

Figure 6a shows that KNN and its variants are at least one order of magnitude and up to two orders of magnitudes faster than INE and IER when using different object distributions for $k = 10$ which is not atypical. As the size of S is increased, the execution time of KNN and its variants, as well as that of INE and IER decrease (although at some point the execution time of IER does start to increase). KNN and its variants perform better than both INE and IER even for large values of $S = 0.2N$, although for extremely large values of $S >> 0.2N$, INE does start to perform better than KNN and its variants. This is because for very large values of S, INE is able to find k neighbors by just visiting a few edges around q in the road network, as there are so many of them. However, as we know well, most object datasets on road networks are sparse. For example, even $S = 0.2N$ is unrealistically large for a dataset of post-offices, pizza shops or restaurants.

Figure 6b shows that KNN and its variants are several magnitudes faster than INE and IER for small values of $k < 20$ as k is varied for a fixed object distribution $S = 0.07N$. In particular, we see that that the various alternative variants of KNN (i.e., KNN-I, INN, and KNN-M) provide a 3–8 times speed up over INE for values of k ranging between 20 and 300, although KNN itself is slower than INE for $k > 50$. As discussed earlier, typical nearest neighbor queries tend to use smaller values of k for which KNN is very well-suited, while the other variants of KNN are more suited for larger values of k. So, depending on the nature of k and S, a suitably designed query optimizer would be easily able to use the appropriate variant of KNN. However, when $k > 300$, only KNN-M is still faster than INE. Note that in these experiments IER was always slower than the remaining algorithms.

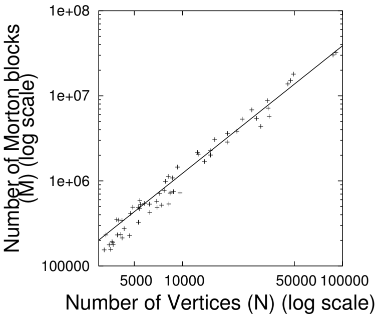


Figure 7: Total number of Morton blocks in the shortest path quadtree encoding of random subgraphs extracted from a large dataset, as well as a line with slope 1.5.

The second set of experiments tabulate the size of the shortest path quadtree for a variety of spatial networks. We used a dataset containing all major roads in the USA (i.e., more than 380,000 vertices and 400,000 edges). By extracting random connected subgraphs from the road network, we were able to account for variations in the various roads such as rural versus urban, and spatial network configurations that would lead to different storage requirements for the underlying shortest path quadtree. Given a spatial network G, we determined the shortest path quadtree for each of its N vertices and calculated the total number of Morton blocks comprising it and then obtained their sum M which is plotted in Figure 7 as a function of N. From Figure 7 we see that the ratio of the total number M of Morton blocks in the shortest path quadtrees for a spatial network G to the number of vertices N in G for a wide range of spatial networks of different sizes obeys $M = K \cdot N^{1.5}$ (where K is a constant) which validates Theorem 6. Recall from Section 3 that this has a very important ramification as it reduces the storage complexity of of our approach of precomputing the $O(N^2)$ shortest paths for the N vertices to $O(N^{1.5})$ from from $O(N^3)$ as in the worst case each of the $O(N^2)$ shortest paths can contain $O(N)$ vertices. This makes the shortest-path quadtree representation scalable as the total amount of space required for a spatial network has been drastically reduced (i.e., by an order of magnitude equal to the square root as $N^{1.5}$ is the square root of N^3).

The third set of experiments evaluate some proposed modifications of KNN that are designed to overcome some of its shortcomings. Recall that KNN is a non-incremental best-first algorithm that uses an upper bound estimate D_k on the maximum possible distance to the kth nearest neighbor of a query object q. An equivalent method of obtaining the k nearest neighbors of a query object is to invoke an incremental best-first variant of KNN (termed "INN") k times—that is, INN is a variant of KNN that does not make use of the priority queue L and where D_k is set to ∞, thereby making it irrelevant. The drawback of INN is that the priority queue *Queue* may get as large as the number of objects. KNN-I is a variant of INN that makes use of a variant of D_k and L to limit the size of the priority queue *Queue*. KNN-I proceeds like INN except that whenever KNN-I encounters a leaf block at front of *Queue* that contains objects, it inserts them into L which is ordered using the maximums of the distance intervals of the objects, although these associated maximum distance values are never updated even though they may be subsequently refined. KNN-I differs from KNN in that KNN also tries to insert objects into L when it encounters them at the front of *Queue*. Once k different objects have been inserted into L, KNN-I uses D_k^0, the maximum distance value associated with the objects in L, to avoid enqueueing any new object o for which the minimum of its distance interval is $\geq D_k^0$ (line 56).

We also introduce another variant of KNN (termed "KNN-M") that uses KMINDIST, a lower bound on the minimum of the distance interval of the kth nearest neighbor, in addition to D_k^0, to obtain the k nearest neighbors of q with the same motivation of reducing the size of the priority queue *Queue*. It proceeds in the same manner as KNN-I with the modification that each time it encounters an object at the front of *Queue*, it enqueues it in an additional priority queue $Queue_1$. Once it has removed the kth object p from *Queue* and inserted it into $Queue_1$, it records the minimum (maximum) of p's distance interval in KMINDIST (D_k^0). Now, it keeps on processing the elements in *Queue* and inserts the objects that it finds in $Queue_1$ until the minimum of the retrieved object is greater than D_k^0, at which time, processing of elements in *Queue* halts as they can no longer be part of the set of k nearest neighbors. At this point, $Queue_1$ is guaranteed to contain all of the k nearest objects as well as other objects. Now, process the element e of $Queue_1$ the minimum of whose distance interval is the smallest. If the maximum of e's distance interval is less than KMINDIST, then report e as one of the k nearest neighbors (in which case e is said to be pruned against KMINDIST). If it is greater than KMINDIST, then check if e's distance interval overlaps that of the current element at the front of $Queue_1$, in which case, refine e and reinsert e into $Queue_1$. This process is continued until k neighbors have been reported. Note that a drawback of using KNN-M is that the objects in the result set are not ordered with respect to q. In other words, in comparison to KNN which establishes a total ordering of its k nearest neighbors, KNN-M does not produce an ordered output.

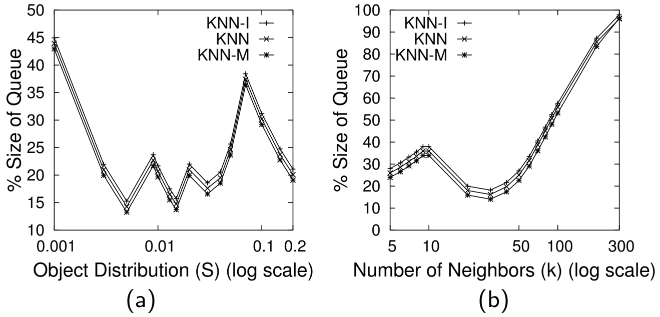


Figure 8: Percentage reduction in the size of the priority queue *Queue* for KNN, KNN-I, and KNN-M, when compared with INN for (a) $k = 10$, and varying sizes of S, and (b) $S = 0.07N$ and varying values of k.

Recall that one of the advantages of using KNN and its variants over INN is that there is a reduction in the size of the priority queue *Queue*, thereby leading to a reduction in the space needed to store it which means that all priority queue operations are faster. Figure 8 shows the reduction in the maximum size of *Queue* for KNN, KNN-I, and KNN-M when compared with INN. For $k = 10$ and varying sizes of S, the maximum size of *Queue* for KNN, KNN-I, and KNN-M is, on the average, at most 35% of the size of *Queue* for INN as shown in Figure 8a. Figure 8b shows the effect of letting k vary between 5 and 300 on the maximum size of *Queue*, while keeping S fixed at $0.07N$. It is clear from the Figure that there is a large reduction in the size of *Queue* for smaller values of $k \leq 100$. However, for larger values of k (e.g., $k > 100$), we observe that the maximum size of *Queue* quickly reaches up to 100% of the maximum size of *Queue* for INN. A possible explanation for this observation is that as k increases, so does the region that is being searched by the nearest neighbor algorithm. As S is obtained by uniformly sampling the set of vertices, the larger the distance that one moves away from q, the greater is the number of objects that have overlapping distance intervals from q. Hence, pruning of the objects using D_k becomes increasingly less effective.

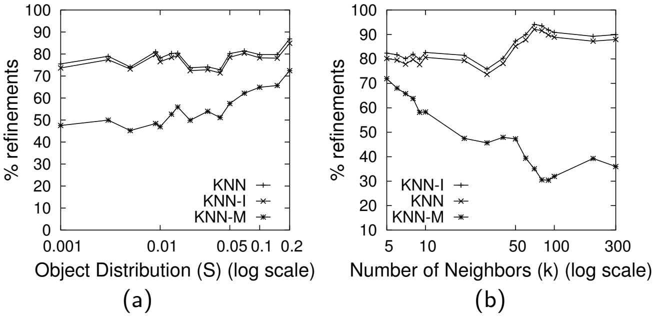


Figure 9: Percentage reduction in number of *refinement* operations for KNN, KNN-I, and KNN-M, when compared with INN for (a) $k = 10$, and varying sizes of S, and (b) $S = 0.07N$ and varying values of k.

Next, we examined the reduction in the number of refinement operations when using the KNN algorithm and its variants in comparison to INN. Figure 9a is the result of letting $k = 10$ and varying values of S. It shows that both KNN and KNN-I resulted in 10% fewer refinements when compared with INN, while KNN-M resulted in 40% fewer refinements. This means that up to 30% of the refinements performed in KNN are devoted to establishing a total ordering of the objects in the result set. Figure 9b is the result of letting k vary between 5 and 300 and fixing S at $0.07N$. It shows that as k increases, the number of refinements performed by KNN-M sharply decreases, while both KNN and KNN-I still perform up to 90% of the refinements performed by INN.

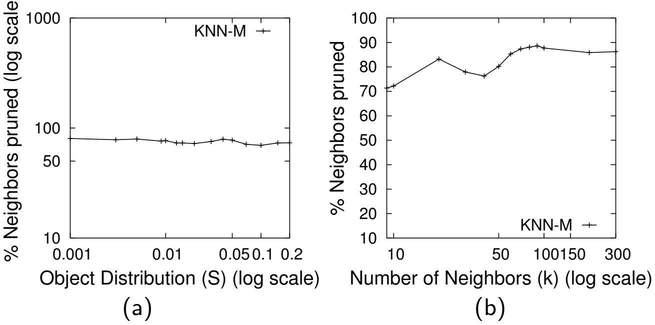


Figure 10: Percentage of the objects in the result set that were pruned against the KMINDIST estimate and hence, were added to the result set for (a) $k = 10$, and varying sizes of S, and (b) $S = 0.07N$ and varying values of k.

The observed large savings in the number of refinements performed by KNN-M in Figure 9b with increasing k is largely due to pruning more and more objects against the KMINDIST estimate. Figure 10 shows that up to 90% of the nearest neighbors in the result set were pruned against the KMINDIST estimate. However, this does not directly translate into an equivalent savings in the number of refinements performed by KNN-M because a nearest neighbor of q whose initial distance interval from q partially overlaps the KMINDIST estimate would still have to perform several refinements before it can be pruned against the KMINDIST estimate.

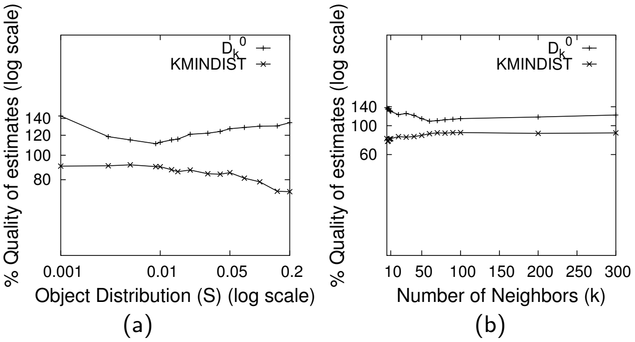


Figure 11: The values of D_k^0 and KMINDIST as a percentage of D_k for (a) $S = 0.07N$, varying values of k, and (b) $k = 10$, varying sizes of S.

Both KNN-I and KNN-M use the D_k^0 estimate which is obtained from the objects inserted into L in lines 58–60. Figure 11, shows both D_k^0 and KMINDIST as a percentage of D_k, which was obtained by running KNN on the same dataset while keeping k constant at 10 and varying S (Figure 11a) and also varying k and keeping S constant at $0.07N$ (Figure 11b). From the Figure we see that D_k^0 is up to 20% larger than D_k which is a possible explanation of why the maximum sizes of the priority queues in Figure 8 for KNN, KNN-I, and KNN-M are almost identical when compared to the maximum size for INN. Moreover, we can see from Figure 11 that the KMINDIST estimate is almost 90% of D_k which implies that many objects in the result set would be pruned against the KMINDIST estimate.

Finally, we compare the relative performance of KNN and its variants. Figures 12a,c show the execution time of KNN and its variants, while Figures 12b,d show the corresponding I/O time. Figures 12a,b show the effect of varying k on the performance of KNN and its variants when S is fixed at $0.07N$, while Figures 12c,d show the effect of varying the size of S on the performance of KNN and its variants when k is fixed at 10. Figures 12a,b also show (labeled "KNN-PQ") the time spent by the KNN in updating D_k (i.e., deleting and inserting elements in L). We make the following observations on the nature of KNN and its variants.

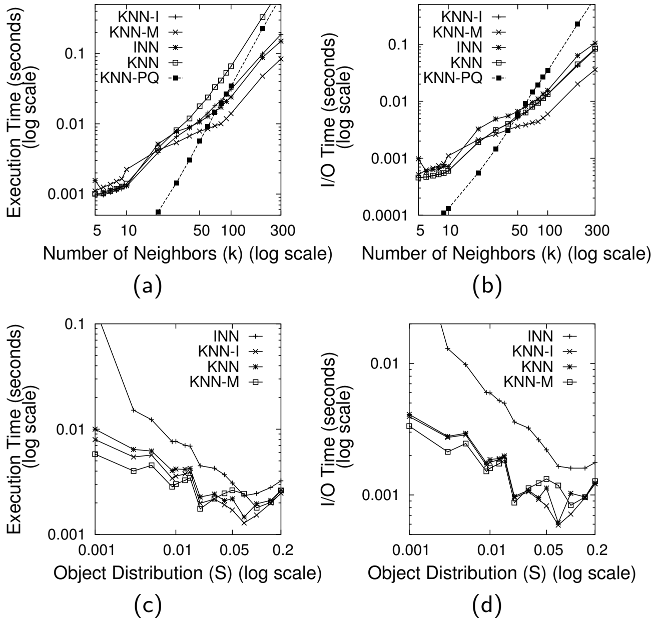


Figure 12: The execution (a,c) and the IO (b,d) time of KNN and its variants for (a,b) $S = 0.07N$, varying values of k, and (c,d) $k = 10$, varying sizes of S.

- For small values of $k \leq 20$, KNN has the fastest execution time among all its variants. For larger values of k ($k > 20$), the cost of updating (i.e., deleting and inserting objects into L) D_k starts dominating KNN's execution time and KNN becomes slower than all of its variants. From Figures 12a,b it can be seen that for $k = 50$, the cost of updating D_k in KNN uses up more than 50% of the execution time and is more than the time for I/O operations.
- For large values of k ($k > 20$), KNN-I and INN can be used instead of KNN.
- If the objects in the result set do not have to be sorted, then KNN-M can be used. However, as KNN-M incurs extra CPU time in computing the KMINDIST estimate, it may not be well-suited for small values of k. In such cases, it may be preferable to use KNN.
- The size of S affects KNN and all its variants in a similar manner, as seen in Figure 12c. The execution time of KNN and its variants decreases as the size of S increases.
- The I/O time dominates the execution time of KNN and its variants as each refinement operation may lead to a disk access. KNN-M is able to reduce the number of refinements by making use of the KMINDIST estimate, which results in a lower I/O cost and hence, lower execution time as well.

5. CONCLUDING REMARKS

A key difference between our algorithm and those that are based on Dijkstra's algorithm (e.g., INE and IER of Papadias et al. [21]) is that in our algorithm the shortest paths between the various vertices in the spatial network are only computed once, whereas in the methods that are based on Dijkstra's algorithm the shortest paths between some vertices are computed repeatedly as the query object and the number of sought neighbors change thereby causing the reapplication of the algorithm. Thus, our algorithm is preferable when many queries are made on a particular spatial network. On the other hand, if only few queries will be made on a given spatial network, then the methods based on Dijkstra's algorithm may be preferable especially if the desired neighbors are quite close to the query object as the entire spatial network need not be explored.

Another advantage of our algorithm is that since the set of objects S from which the neighbors are drawn is decoupled from the actual spatial network, the algorithm (and most importantly the shortest-path quadtrees for the spatial network) can be used with different sets of objects as long as the spatial network is unchanged. For example, we can have separate search hierarchies for gas stations, markets, restaurants, etc. In this case, queries for the nearest gas stations, markets, restaurants, etc. could be executed with no change and the algorithm would be more efficient than had we placed the gas stations, markets, and restaurants in one search hierarchy as each time we found a neighbor we would need to check its type and proceed to the next one if it was not the desired type. In contrast, in the methods based on Dijkstra's algorithm the distinction between the vertices of the spatial network and the set of objects from which the neighbors are drawn is not so clearcut.

It is important to note that although we restricted our spatial networks to be planar, this was only for the purpose of deriving the order of its space requirements which depended on the regions of the shortest-path map and corresponding shortest-path quadtree being disjoint and contiguous. However, the actual algorithms that we presented work with both planar and nonplanar spatial networks. In other words, the presence of tunnels and bridges will not affect the correctness of the algorithms. In fact, the definition of the shortest-path quadtree in terms of the vertices of the spatial network minimizes the effect of the nonplanarity as we saw that the resulting regions may be noncontiguous regardless of planarity or lack of it, although we did show that the order of the space requirements did not change for this formulation in the planar case. An interesting direction for future work is a derivation of the space requirements for nonplanar spatial networks.

One can take advantage of the fact that our framework will most commonly be deployed in an end user application that is mostly concerned with nearby destinations. It is not unreasonable as most people do not want to drive more than 50 miles to get to a restaurant. In this case, the shortest path quadtree will be much smaller, and far less expensive to compute. Another strategy is to assume that the shortest path between sources and destinations that are more than X miles of each other must use a highway. Such a situation is a marriage between multiresolution techniques of [14] and the shortest path quadtree techniques and could lead to substantial speedups in computing shortest paths, although this may possibly be at the expense of suboptimal shortest paths for distances spatially farther than X miles.

Repeatability Assessment Result. Figures 6 and 8–12 have been verified by the SIGMOD repeatability committee.

Acknowledgments. This work was supported in part by the U.S. National Science Foundation under Grants EIA-00-91474, CCF-05-15241, and IIS-0713501, as well as NVIDIA Corporation and the Virtual Earth and Bay Area Research Center groups of Microsoft Research.

6. REFERENCES

[1] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[2] H.-J. Cho and C.-W. Chung. An efficient and scalable approach to CNN queries in a road network. In *VLDB'05*, pp. 865–876, Trondheim, Norway, Sep. 2005.

[3] K. L. Clarkson. Fast algorithm for the all nearest neighbors problem. In *FOCS'83*, pp. 226–232, Tucson, AZ, Nov. 1983.

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

[5] G. N. Frederickson. Planar graph decomposition and all pairs shortest paths. *JACM*, 38(1):162–204, Jan. 1991.

[6] K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Trans. on Comp.*, 24(7):750–753, July 1975.

[7] I. Gargantini. An effective way to represent quadtrees. *CACM*, 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*, pp. 156–165, Vancouver, Canada, Jan. 2005.

[9] A. Henrich. A distance-scan algorithm for spatial access structures. In *ACM GIS'94*, pp. 136–143, Gaithersburg, MD, Dec. 1994.

[10] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, June 1999.

[11] E. G. Hoel and H. Samet. Efficient processing of spatial queries in line segment databases. In *SSD'91*, LNCS 525, pp. 237–256, Zurich, Switzerland, Aug. 1991.

[12] H. Hu, D. L. Lee, and V. C. S. Lee. Distance indexing on road networks. In *VLDB'06*, pp. 894–905, Seoul, Korea, Sep. 2006.

[13] G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *PAMI*, 1(2):145–153, Apr. 1979.

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

[15] M. R. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB'04*, pp. 840–851, Toronto, Canada, Sep. 2004.

[16] M. R. Kolahdouzan and C. Shahabi. Continuous k-nearest neighbor queries in spatial network databases. In *STDBM'04*, pp. 33–40, Toronto, Canada, Aug. 2004.

[17] M. Lindenbaum, H. Samet, and G. R. Hjaltason. A probabilistic analysis of trie-based sorting of large collections of line segments in spatial databases. *SIAM J. on Computing*, 35(1):22–58, Sep. 2005.

[18] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Tech. report, IBM Ltd., Ottawa, Canada, 1966.

[19] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *SIGMOD'87*, pp. 270–277, San Francisco, May 1987.

[20] J. A. Orenstein. Multidimensional tries used for associative searching. *Inf. Proc. Letters*, 14(4):150–157, June 1982.

[21] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB'03*, pp. 802–813, Berlin, Germany, Sep. 2003.

[22] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD'95*, pp. 71–79, San Jose, CA, May 1995.

[23] H. Samet. Region representation: quadtrees from binary arrays. *CGIP'80*, 13(1):88–93, May 1980.

[24] H. Samet. An algorithm for converting rasters to quadtrees. *PAMI*, 3(1):93–95, Jan. 1981.

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

[26] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *TOGS*, 4(3):182–222, July 1985.

[27] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *ACM GIS'05*, pp. 200–209, Bremen, Germany, Nov. 2005.

[28] C. A. Shaffer and H. Samet. Optimal quadtree construction algorithms. *CVGIP'87* , 37(3):402–419, Mar. 1987.

[29] C. A. Shaffer, H. Samet, and R. C. Nelson. QUILT: a geographic information system based on quadtrees. *Int. Jour. of Geog. Inf. Sys.*, 4(2):103–131, Apr.–Jun. 1990.

[30] 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, Sep. 2003.

[31] D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In *ESA'03*, LNCS 2832, pp. 776–787, Budapest, Hungary, Sep. 2003.

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