ABSTRACT

Title of dissertation	SCALABLE QUERY PROCESSING ON SPATIAL NETWORKS
	Jagan Sankaranarayanan, Doctor of Philosophy, 2008
Directed by	Professor Hanan Samet Department of Computer Science

Spatial networks (e.g., road networks) are general graphs with spatial information (e.g., latitude/longitude) information associated with the vertices and/or the edges of the graph. Techniques are presented for query processing on spatial networks that are based on the observed coherence between the spatial positions of the vertices and the shortest paths between them. This facilitates aggregation of vertices into coherent regions that share vertices on the shortest paths between them. Using this observation, a framework, termed SILC, is introduced that precomputes and compactly encodes the n^2 shortest path and network distances between every pair of vertices on a spatial network containing n vertices. The compactness of the shortest paths from source vertex v is achieved by partitioning the destination vertices into subsets based on the identity of the first edge to them from v. The spatial coherence of these subsets is captured by using a quadtree representation whose dimension-reducing property enables the storage requirements of each subset to be reduced to be proportional to the perimeter of the spatially coherent regions, instead of to the number of vertices in the spatial network. In particular, experiments on a number of large road networks as well as a theoretical analysis have shown

that the total storage for the shortest paths has been reduced from $O(n^3)$ to $O(n^{1.5})$. In addition to SILC, another framework, termed PCP, is proposed that also takes advantage of the spatial coherence of the source vertices and makes use of the Well Separated Pair decomposition to further reduce the storage, under suitably defined conditions, to $O(n)$.

Using these frameworks, scalable algorithms are presented to implement a wide variety of operations such as nearest neighbor finding and distance joins on large datasets of locations residing on a spatial network. These frameworks essentially decouple the process of computing shortest paths from that of spatial query processing as well as also decouple the domain of the participating objects from the domain 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.

SCALABLE QUERY PROCESSING ON SPATIAL NETWORKS

by

Jagan Sankaranarayanan

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

This thesis is dedicated to
amma and *appa*
and to
Sriram and *Aswin*

Acknowledgments

This dissertation describes an algorithm for path finding in road networks. In the maze called life, there are no "algorithms" for finding one's way. But if one is fortunate, there is the wise counsel of good friends which can come close to a very good algorithm. Prof. Hanan Samet has been one such teacher, mentor, and friend to me during my graduate studies at College Park. It is needless to say that I have been inspired by his brilliance and his dedication to work. Most importantly, I am grateful for his generosity and hospitality that made me feel at home far away from home. I will cherish these good memories and my friendship with Hanan and his wife forever.

I am deeply indebted to Dr. Houman Alborzi who has been a good friend and a patient teacher. I have benefited immensely from my close collaboration with Houman. This thesis has turned out to be a lot better thanks to Houman's keen intellect and his good intuition. I wish him and his family all the very best.

I thank Prof. Varshney for all the stimulating conversations and guidance over the years. This thesis started out as a class project in his *Advanced Computer Graphics* course. I would like to thank other members of my dissertation and proposal committees consisting of Prof. Larry S. Davis, Prof. David M. Mount, and Prof. Galit Shmueli for their useful suggestions and their careful review of my dissertation. Many thanks to Prof. Leila de Floriani, Prof. Ramani Duraiswami, Prof. Samir Khuller, and Dr. Jon Sperling for their constant encouragements.

I have immensely benefited from my conversations and friendship of my fellow travelers through graduate school. I thank Dr. Houman Alborzi, Dr. Frantisek Brabec, Dr. Charles B. Cranston, Dr. Edwin H. Jacox, Dr. Michael Lee, Michael D. Lieberman, Frank L. Morgan III, Daniele Panozzo, Dr. David A. Tahmoush, Prof. Egemen Tanin, Benjamin E. Teitler, and Gregory M. Sanders. I wish them all personal and professional successes in life. Special thanks are due to Frank for the weekly squash games. More importantly I thank him for being a dear friend.

Many thanks to Dr. Charles B. Cranston, and also to Frank L. Morgan III, and Aswin C. Sankaranarayanan for their careful proof reading of my thesis. My deep appreciation to Mr. Fritz McCall and the UMIACS system staff for their excellent technical support. I thank Ms. Janice Perrone and Ms. Fatima Bangura for their help with my travel and other administrative work.

I thank all my friends from childhood until now for being constant companions and making this a fun adventure.

Last but not the least, my family and members of my extended family have been a constant source of inspiration, love, and support. My love and gratitude to my parents Smt. M.V. Shyamala and Shri. H. Sankaranarayanan, grandparents Smt. V. Valambal and Dr. M. Viswanathan, brothers Dr. Sriram Sankaranarayanan and Shri. Aswin C. Sankaranarayanan, *manni* Smt. Indira Sriram, uncles and aunts, and more than a dozen cousins.

This landmark in my life's journey would not have been possible if not for the love, encouragement, and sacrifices of my parents and brothers. To them, I dedicate this thesis.

Contents

List of Figures

Chapter 1

An Introduction to Query Processing on Spatial Networks

Spatial databases are being deployed in GIS applications with desirable outcomes. Some of the recent advances in spatial database techniques have resulted directly in the ability to handle larger volumes of GIS data and to perform queries of increasing complexity. Performing spatial queries on *transportation networks* is an application that is of immense interest to the GIS community [160, 166, 172]. Transportation networks form an integral part of GIS applications, such as location-based services [137] and locational analysis [27]. Location-based services deal with queries generated by a mobile host. Moving object databases [137, 166, 171, 191] and trip-planning [172] are closely related to location-based services. Locational analysis [27] 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 a city setting would issue a series of spatial queries to find a suitable location that is easily accessible to the general populace. This dissertation describes a framework that enables a wide variety of operations on transportation networks.

Spatial Networks

To make the discussion more general, we introduce the concept of a *spatial network*, which is an extension of a network model. In classical literature, 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 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 which 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 on an embedding space 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). We define the *network distance* between any two objects s and t on a spatial network to be the sum total of the weights of the edges constituting the *path of minimum possible weight* (termed *shortest path*) from s to t on the spatial network. In contrast, the *geodesic* or the *spatial distance* between the two objects s and t is a function of the spatial positions of s and t, that is — such a distance function does not take into account the constraints imposed by the spatial network.

Query Processing on Spatial Networks

Query processing on spatial networks refers to algorithms that can perform a variety of database queries on datasets of locations residing on spatial networks. Queries that we are interested in studying are predominantly *spatial* in nature. That is, we assume that one or more sets of objects in the embedding space are provided as inputs to the spatial query processing algorithms, such that the objects are free to coincide with a vertex, edge, or face of a spatial network. Moreover, the objects can be points, lines, or arbitrary regions in the embedding space. We first give examples of some of the types of queries that can be made on a spatial network.

1. **Shortest Path and Distance Queries:** [40,44,49,52,61,63,71,72,93–95,144,151, 152,154,165,166,170,186,188,195] Given a source and a destination location on a spatial network (say, in terms of postal addresses), we would like to obtain the shortest path and the network distance in miles, or in terms of trip time.

2. **Range Queries:** [127, 138, 155] Given a location s on a spatial network, find all restaurants that are within l miles of s, or alternatively, that can be reached within p minutes.

3. **Nearest Neighbor Queries:** [43, 71, 85, 86, 92, 101, 127, 144, 151, 152, 154, 166] An emergency worker queries a database for the closest k hospitals to the scene

of an accident arranged in increasing order based on trip time or trip distance.

4. **Incremental Nearest Neighbor Queries:** [144,154] Given a location q and a set S of restaurants on the spatial network, the first invocation of the incremental neighbor algorithm obtains the closest restaurant to q in S. Subsequent invocation of the algorithm produces the next (second) closest restaurant, and so on. The key idea here is that the user need not specify the number of neighbors in advance. Using this algorithm, more complicated queries like — find the *next* closest restaurant to q that serves, say "Sushi", and is open past 9:00pm — can be supported on spatial networks.

5. **Distance Join Queries:** [43,155] A car maintenance company has 3,000 locations in the US and services 30 million customers countrywide. Each customer receives a personalized reminder with the address of the nearest franchise location as well as the trip time, and distance.

6. **Closest Pair Query:** [127, 155] Given the locations of pizza shops and movie theaters on a spatial network, find the closest pairing between a pizza shop and a movie theater.

7. **Approximate Distance Queries:** [166] Given a source u and a destination v location on a spatial network, find the approximate network distance d' between u and v. d' must approximate the actual network distance between u and v with a

bounded error, and should be expediently computed.

8. **Network Voronoi Diagram:** [101] Given the locations of fire stations in Washington, DC, partition the road network of DC into regions such that each region is associated with its *closest* fire station.

9. **Trip and in-route Queries:** [106, 114, 172] Given the locations of gas stations, and gas prices, we can plan the following *trip* query. Given a source s, destination d, gas capacity g, initial capacity c, consumption rate of r miles/gallon, find the route that minimizes the cost of the trip.

10. **Traveling Salesman Problem (TSP):** [106, 114, 167] A pizza deliverer queries a database for a trip that visits all the k delivery locations in the shortest distance, or trip time.

11. **Continuous Queries:** [34, 100, 102, 130, 166] This family of queries deals with a mobile host c (e.g., a car traveling on a highway), and a server s that is continuously queried by c. The goal here is to minimize the server traffic by reusing some of the previous answers given by s based on a previous position of c, or by determining the next time the server should be queried in order to minimize the number of times s is probed.

12. **Network Skyline queries:** [168] Suppose that we are given two multidimensional objects x and y. If $x = x_1, x_2, \ldots, x_d$ and $y = y_1, y_2, \ldots, y_d$ such that $x_1 < y_1$, $x_2 < y_2$,

..., $x_d < y_d$, then y is said to *dominate* x. Now, given a set S, the *skyline* operator on S produces a subset S' of the objects in S, such that objects in S' are not dominated by any object in S. Given the location c of a conference, a set S of hotels around it, and the corresponding room rates, the network skyline operator finds a dominant set of hotels in S with respect to the network distance to c, and their room rates.

13. **Reverse Nearest Neighbor Queries on Spatial Networks:** [194] Suppose that we are given a set of fire stations F and a set of restaurants R on a spatial network. Given a particular fire station f in F, find the subset of restaurants in R that have f as their nearest neighbor.

14. **Aggregate Nearest Neighbor Queries on Spatial Networks:** [126, 193] Given a set S of meeting places, and the locations of l club members, find a meeting place $h \in S$ that minimizes the sum total of the trip time taken by the l members to reach h.

15. **Spatio-temporal queries:** [60, 64, 97] This family of queries accounts for a temporal dimension. For example, we can model traffic congestion by making the link weights of a spatial network into a time dependent function. The result of such a query would differ depending on the time of the day.

16. **Location Analysis Queries:** A bulk sales outlet plans on opening a new megastore in Maryland, and has several expectations from an optimal location l. The

optimal location should be within c miles of a highway, at least r miles from other stores owned by the same company, the average income of the *neighborhood* (say, 20 miles radius around l) should be less than m, and the average trip time should be no more than q minutes.

Even though the inputs to the algorithm are objects that have spatial positions and extents, the distance between two objects is not the shortest possible distance on the embedding space (such as "the way a crow flies"), but is dictated by the spatial network. Queries on spatial networks are complicated because the distances are constrained to lie along the shortest path through a network. Owing to the large size of spatial networks, graph-theoretical approaches such as shortest path algorithms (e.g., Dijkstra's shortest path algorithm [40]) cannot be used for real-time query processing. For example, given a source vertex X and destination vertex V on a spatial network $G(V,E)$, the shortest path and the network distance between X and V is obtained by invoking a shortest path algorithm (e.g., Dijkstra's algorithm) between X and V on G. However, as seen in Figure 1.1, the invocation of Dijkstra's algorithm between two vertices X and V on a road network of Silver Spring, MD resulted in 3191 of the 4233 vertices in the spatial network being visited by the algorithm. The shortest path algorithm visits many vertices that are unrelated to the shortest path between X and V in the spatial network resulting in wasteful work. Consequently, *scalable real-time* query processing on spatial networks is particularly challenging as the network distance between two *objects* on a spatial network is very expensive to compute in real-time.

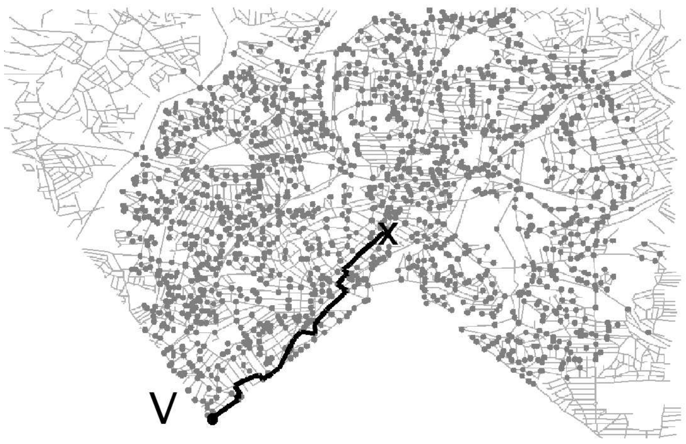


Figure 1.1: Dijkstra's algorithm between X and V on a road network of Silver Spring, MD visits 3191 of the 4233 vertices in the network. The nodes visited by Dijkstra's algorithm are marked with a darker shade

Network distance computations on a spatial network are expensive which makes spatial query processing on spatial networks different from traditional spatial query processing on a *normed space*. Moreover, traditional spatial query processing [82, 83, 135] on normed spaces rely heavily on the ability to compute *distance primitives* quickly and efficiently, in conjunction with a hierarchical spatial data structure on S to enable efficient query processing. In particular, the distance primitives provide the ability to compute the distance between two regions, between a point and a region, or between two points in the embedding space. Examples of such distance functions are MINDIST, MAXDIST, and MINMAXDIST(or MAXNEARESTDIST) which are defined in [82, 83, 135, 143–145]. However, in a spatial network, unlike a normed space, computing distance primitives is not trivial, unless special provisions are made.

Finally, we discuss in greater detail an example pertaining to a user-generated query on a real estate database. Using this example, we show why query processing on spatial networks is too expensive to perform in real time. Consider the following query scenario:

> Potential home buyers query a real estate database for houses that satisfy a set of requirements (e.g., type, price, neighborhood). They wish to rank each house in the result set based on the sum of the distances (or alternatively trip time) to a set of addresses (e.g., office addresses of the user and spouse, children's school addresses, etc.).

An existing traditional query processing system would process the query as follows. First, it would compute the result set based on the input specifications provided by the user. Next, it would compute the distance (and the trip time) from each entry in the result set to the set of preferred addresses. We know that computing the network distance between two locations on a spatial network *G* requires the invocation of a shortest path algorithm on *G*, which can be quite expensive. Moreover, as the result set is expected to be large, the number of network distance computations invoked as a result of the above mentioned query would also be large. To provide the reader with a modest estimate of the expected overhead, we assume that the result set contains 50 elements, each path and distance search on the spatial network takes one tenth of a second and that the user is interested in the trip time to five different locations. The expected execution time for such a query would be more than twenty seconds, which is unacceptable for most interactive applications. An alternative strategy is to use a geodesic distance measure as an approximation to the actual network distance on the spatial network. Even though using the geodesic measure makes distance computations faster, the result of the query is no longer useful as the errors associated in using a geodesic distance can be large.

Emergence of Web Services

The computing landscape is changing rapidly. Machines are getting more powerful and yet cheaper at the same time. Special purpose processors such as GPU and Cell processors that are capable of performing mathematical operations at a higher throughput than general purpose processors are now cheaper due to the dynamics of the scale. These advances are sparking a new revival of parallel and distributed computational techniques. This trend is evident in the rapid increase in the number of "cores" in machines, number of SPEs in cell processors, and the number of parallel pipelines in GPUs (some of which have up to 512 processors in them). There is a push towards "cloud computing" which enables massive distributed systems to be built by stringing together multiple cheap commodity machines. Google's MAP-REDUCE [39], and Microsoft's Dryad [91] frameworks are examples of cloud computing frameworks. In order to understand and make sense of this new trend, it requires the recognition of the fact that this trend is here to stay. Welcome to the new order where computation is cheap and plentiful, and storage is practically limitless.

Another trend in computing is that there is a premium for quick answers in real time. As more and more applications become web services, “cloud applications” in technology parlance, there is an expectation of getting answers in real time and at a scale that is able to service millions of people at the same time. Recently, we have witnessed the emergence of several web services, e.g., Mappoint, Mapquest, Yahoo! Yellow Pages, and Google Maps, that enable users to pose simple queries on spatial networks. Most of these applications employ a spatial database engine to ensure reliable and scalable delivery of services. In spite of a high level of sophistication of most spatial databases, queries on spatial networks are restricted to simple route finding operations and local searches on places such as restaurants, theaters and other places of interest. It is easy to see that queries on spatial networks can be computationally expensive and processing these queries in a realistic time frame remains a challenge.

We want to build a service that is capable of answering shortest path and nearest neighbor queries on spatial networks, in real time, and which is scalable enough to handle Internet traffic. Graph algorithms are well studied problems, and there are a number of classical approaches to problem solving on graphs; one of which is Dijkstra’s algorithm for computing shortest paths in a graph. However, if computation is cheap, and storage is limitless, it opens up new possibilities of using solutions that were considered unreasonable until a few years back. For example, it becomes feasible to precompute and compactly store the shortest path and the network distance between every pair of vertices, thereby providing the ability to answer shortest path and nearest neighbor queries on spatial networks in real-time.

In Defense of Precomputation

There is a vast body of graph theoretical approaches to computing shortest paths on general graphs. However, our work on spatial networks is different from previous approaches on shortest path algorithms for general graphs in several ways. First, we propose to precompute and store the shortest paths and the distances between every pair of vertices in a spatial network. We refer to this stored representation of the shortest paths and the distances between all pair of vertices as the *path encoding* and the *distance encoding* of a spatial network, respectively. Although, this can be quite expensive for large spatial networks, it can be achieved with a sufficient investment of time and hardware resources. We first argue that such a representation is feasible to compute. Given a graph $G(V,E), n = |V|, m = |E|$, Dijkstra's algorithm using a Fibonacci heap [53] 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 Dijkstra's algorithm would be $O(n^2 \log n)$. Empirical studies [195] have indicated that 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.* [78] present a linear time shortest path algorithm for planar graphs, while Thorup [178] provides a linear time shortest path algorithm for general graphs with integer edge weights. A host of other techniques like parallel processing and the use of sophisticated hardware such as Graphical Processing Units (GPU) [76,115] could further speed up the precomputation of all shortest paths of a graph.

Decoupling of Domain and Data

This brings us to the general idea of *decoupling*, which provides a cost justification to our precomputation strategy. Suppose that we precompute and store the shortest path and network distance between every pair of vertices in the spatial network of Manhattan, NY. Such a representation can potentially be used by several datasets (possibly, millions of user generated datasets) pertaining to postal addresses in Manhattan for query processing. Moreover, spatial networks are usually static structures, while datasets of objects may be updated frequently. When dealing with a set of mobile hosts on a road network, the current positions of the objects are frequently updated, while the road network in itself would largely remain static. So, precomputation is largely a one-time affair. In effect, what we have done is to *decouple* the *data* from the *underlying domain* which allows for the datasets and the network representation to be largely independent of each other. In this respect, our work is distinguished from the work of Papadias *et al.* [127] and Kolahdouzan *et al.* [101] who perform path and distance queries at run-time, while making no provisions to reuse such computations across queries and across datasets. For example, the network Voronoi diagram in [101] computed for a set of restaurants in Manhattan cannot be used for another dataset of post offices in Manhattan. In effect, what we have done is *decouple* the *data* from the *underlying domain* thereby making provisions to reuse computation across queries and across datasets.

Path Coherence and Path-Distance Encoding

We suggest precomputing and storing the shortest path and distance between every pair of a spatial network. However, the compact representation of the path and distance encoding of a spatial network is a very challenging problem as we will now examine. Consider a spatial network $G(V,E)$ containing n vertices. The path encoding of G usually requires $O(n^3)$ space, when we assume that each shortest path has $O(n)$ vertices. If instead of storing the entire shortest path between two vertices s and t, we can just store the next vertex w ("next hop") on the shortest path from s to t. We can then obtain the entire shortest path between s and t, by repeatedly finding the next vertex in the shortest path between w and t and so on. We have now reduced the total amount of space necessary to $O(n^2)$. Of course, this is at the cost of making the process of obtaining the shortest path be iterative, and which now takes $O(k)$ time, where k is the length of the shortest path.

In this dissertation, we will show how to reduce the space requirements even further. We make the key observation that the shortest paths between spatially proximate source vertices and spatially proximate destination vertices will often pass through a common vertex, which is termed *path coherence*. Using extensive theoretical and empirical analysis of our technique, we show the *scalability* and the applicability of our work to very large spatial network databases. In particular, by making suitable assumptions about the nature of spatial networks, we present the SILC and the PCP frameworks that are able to reduce storage requirements by appealing to the dimensionality reduction property [87, 88, 141, 142] of quadtrees [45, 124, 140–142, 144]. Our proposed framework takes advantage of the *coherence* between the spatial position of vertices and the shortest paths between them.

Vertices in a spatial network that are spatially close to one another share a number of common properties. In particular, often two vertices s and u that are spatially close to each other share large common segments of their shortest paths to two other vertices t and v that are also spatially close to each other, but far from s and u. We call the coherence between the shortest paths from nearby sources to nearby destinations as *path coherence*. 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. In fact, path coherence may explain the

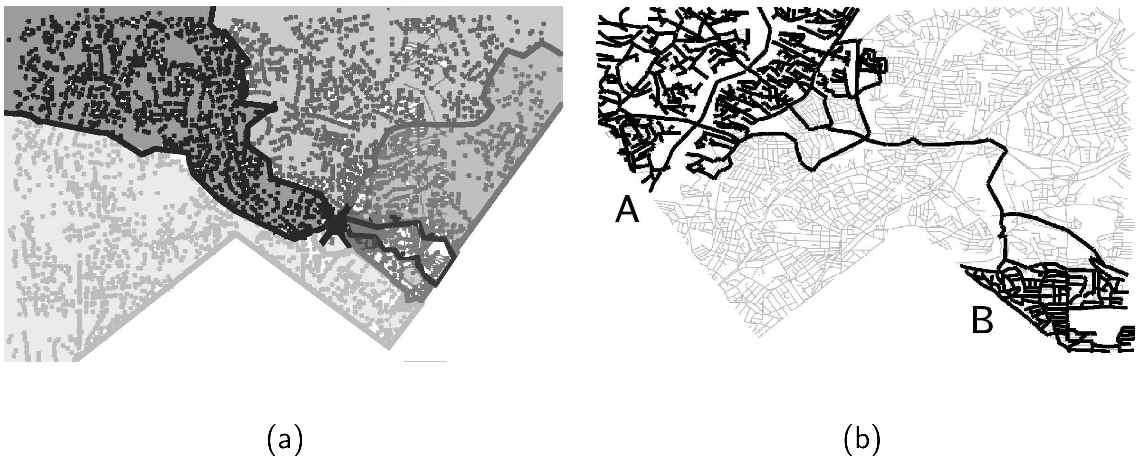


Figure 1.2: Example illustrating the path coherence in a road network of Silver Spring, MD. a) The vertices are assigned colors based on their shortest path from u through one of the six adjacent vertices of u. b) The 30,000 shortest paths (given in a darker shade) between every pair of vertices in A and B pass through a single vertex.

chronic traffic congestion on our roads. Moreover, experienced drivers can easily find the shortest route to an unknown location l simply based on its proximity and relative position to a set of known locations. All the above examples are the manifestations of path coherence in spatial networks.

Path coherence occurs naturally in most spatial networks that are of interest to the GIS community. Wagner and Willhalm [187] discuss the reverse problem of assigning spatial positions to vertices of a general graph so that path coherence is *induced* into the resulting spatial network. Figure 1.2a shows the shortest path from a vertex u in the spatial network G of Silver Spring, MD. Now, each vertex in G (other than u) is assigned a color based on which outgoing edge of u forms the first link in the shortest path from

u. For example, u has six outgoing edges and hence, every vertex in G is assigned one of the six possible colors based on which of the 6 outgoing edges of u form the first link in the shortest path from u. We see that the resulting spatial network has contiguous colored regions, which indicate that the two destination vertices v, w in G that are close to one another, but spatially far from u share common segments in the shortest path from u. Figure 1.2b shows the shortest paths between every pair of vertices between a set of source vertices A and a set of destination vertices B. In particular, all the $30,000$ shortest paths between A and B share a large segment of their shortest path. This is owing to *path coherence* in spatial networks. Our path and distance encoding strategies take advantage of the path coherence in spatial networks.

Development of a General Framework

We propose both the SILC framework [144, 152, 154–156], which uses a quadtree [45, 144], and another framework called Path Coherent Pairs (PCP) which uses the Well-Separated Pair decomposition [28, 30, 31] to encode the shortest path and network distance between every pair of vertices in a spatial network. The resulting representation has been shown to grow gracefully as the number of vertices in the spatial network increase and hence, is *scalable*. The focus of our work is not restricted to any particular query on spatial networks. In fact, using the framework proposed in this dissertation enables us to perform efficiently many spatial database queries such as distance, range queries, incremental nearest neighbors [80], and spatial joins [82] on spatial networks thereby potentially paving the way for other sophisticated queries to be applied on spatial networks. We use disk-efficient indices, such as a B-tree [37] to represent the resulting path and distance encoding of a spatial network. Thus, our technique can be used in conjunction with any existing commercial database system; the design of one such system is briefly mentioned in [156]. Moreover our framework enables current database processing techniques to be applied on spatial networks, thereby encouraging code reuse.

Query processing on spatial networks should not be restricted to a particular data structure on S. That is — the distance primitives for query processing (e.g., MINDIST, MAXDIST, MAXNEARESTDIST) should be available to a wide variety of spatial data structures on S. We achieve this by making the distance primitives oblivious to the spatial data structure on S. In other words, we have *decoupled* the spatial query processing algorithms and the data structure on S from the algorithms that compute the distance primitives on spatial networks.

Progressive Refinement of Distance

As mentioned before, query processing on spatial networks is complicated by the fact that obtaining the network distance between two objects on a spatial network requires the invocation of a shortest path algorithm, which can be expensive. In this dissertation, we propose a path encoding that explicitly stores the next link in the shortest path. The network distance between two objects u, v can be implicitly obtained by first retrieving the entire shortest path between u and v, and then obtaining the network distance by summing the weights of the edges that comprise the shortest path.

Computing the exact network distance may result in excessive and wasteful for most query processing algorithms. Now, consider the following example of a nearest neighbor query on a spatial network. Suppose that we wish to find which city among— "Washington" or "Princeton"— is closer to "College Park, MD"? Such an operation forms the basis of algorithms that provides an ordering of objects based on the network distance to a given query object, "College Park" in this case. We point out that such an operation does not require that the exact network distance between the objects and the query object be known. In other words, if we can provide an approximate estimate of the network distance, say in the form of an *interval*, in some cases, we can answer the query without ambiguity. Suppose, we estimate the network distance between College Park to Washington to be between 10 and 30 miles, and the network distance from College Park to Princeton to be between 50 and 100 miles. We can safely conclude that Washington is closer to College Park than Princeton, as the maximum estimated distance from College Park (30) to Washington is closer than the minimum estimated distance (50) to Princeton. Instead, suppose that the network distance estimate from College Park to Princeton is between $[25, 100]$ miles, in which case, it is not clear if Princeton, or Washington is closer to College Park as the distance intervals corresponding to Princeton and Washington from College Park are *intersecting*. In order to handle such cases, we can also provide an *refinement* operator that takes a network distance interval and *tightens* it. That is, when applied to the network distance interval $[25,100]$ the refinement operator tightens it; say makes it $[75,95]$; in which case, we are able to answer our original query without any ambiguity.

In our framework, we will provide mechanisms by which an initial network distance interval can be quickly computed, and can be *refined* as many times as needed until a query can be answered without any no ambiguity. We call such a model for network distance computation as *progressive refinement* of distances which makes for efficient algorithms that expend only as much work as needed.

Approximate Distance Oracles

Often two vertices s,u that are spatially close to each other share large common segments of their shortest paths to two other vertices t,v that are also spatially close to each other, but far from s,u. This also means that the network distances between the source and destination vertices are more or less the same, and that they can be approximated using a single network distance value. Using this idea, we develop what we term an ε-*approximate distance oracle* for spatial networks which is capable of answering ε-*approximate network distance* queries, of a specified approximation ε, between any two vertices in G, that is—given a source vertex s and a destination vertex w in G, the network distance $S_\varepsilon(s,w)$ produced by the oracle S_ε is guaranteed to be no more or less than an ε fraction of the actual network distance $d_G(s,w)$ between s and w in G.

Our construction of the oracle takes advantage of the coherence between the spatial positions of vertices in G and the network distance between them, thereby enabling us to find pairs of subsets of vertices in G, such that the network distance between all the vertices contained in a pair can be approximated by a single value. The utility of such an oracle is that it is capable of providing approximate answers very quickly as well as resulting in substantial savings in storage space.

Bringing “Spatial” Attribute in to Prominence

Spatial networks are general graphs whose vertices and edges are associated with additional spatial information. There are two distinct attributes to a spatial network — *connectivity* as represented by the general graph, and the *spatial* information in terms of the position of vertices and edges in an embedding plane. There is a rich tradition of performing graph operations on general graphs. In the past few years, there has been some interest in performing query processing on road networks. These approaches model road networks as general graphs, which in turn reduces query processing on road networks in to graph operations. That is, most existing approaches do not make use of the spatial information in road networks.

Our work and related work by Wagner et al. [186] are possibly the only two approaches that study the interplay between the spatial and the connectivity attributes of spatial networks. We show that there is a *coherence* between the spatial positions of vertices and the shortest paths between them. The observation of coherence leads us to the shortest-path quadtree representation which succinctly captures the shortest paths between a source u to all the remaining vertices in a spatial network. The ramification of our approach is that, we have shown that traditional graph-based operations on spatial networks can indeed be transformed into geometrical operations. In fact, one of the main contributions of our work is that we have raised the "spatial" attributes of spatial networks from an oft ignored position to a prominent status in query processing.

Optimal Neighborhoods for Point-Clouds

We discuss two geometric algorithms that have important applications to point based graphics. Given a dataset S of 3D points, the algorithm computes the k nearest neighbors of each point in S. In contrast to the previous methods, substantial cost savings is achieved by reusing the nearest neighbor of a point in the nearest neighbor computation of another proximate point. This results in an algorithm that is work efficient, and is scalable to large point-cloud datasets. Moreover, given a set s of points, we define the *neighborhood* r_s of s as a region containing the k nearest neighbors of all the points in s. We are able to show that the size of r_s constructed by our algorithm is *optimal*. That is, it is not possible to construct a smaller neighborhood than the one constructed by our algorithm. In addition to the above described algorithm, we also develop a neighborhood algorithm for point-clouds using the MAP-REDUCE framework [39] which is capable of processing a 3D point-cloud containing over two billion points in a little over 3.5 hours on a cluster containing 20 machines.

Indexing of Objects with Extents

Finally, we briefly study the properties of the cover fieldtree [50,51] and the more commonly known loose quadtree (octree) [150,182]. Both these data structures are designed to organize objects with extents (e.g., rectangles, or cells). The cover fieldtree and the loose quadtree overcome the drawback of spatial data structures that associate objects with their minimum enclosing quadtree (octree) cells which is that the size of these cells is independent of the size of the objects and may be as large as the entire space from which the objects are drawn. The loose quadtree (octree) achieves this by expanding the size of the space that is spanned by each quadtree (octree) cell c of width w by a cell expansion factor p ($p > 0$) so that the expanded cell is of width $(1+p) \cdot w$ and an object is associated with its minimum enclosing expanded quadtree (octree) cell. The maximum possible width w of c given an object o with minimum bounding box b of radius r is determined as just a function of r and p and is independent of the position of o. Moreover, the range of possible ratios of width $w/2r$ as a function of p is explored and for $p \geq 1$ is shown to take on at most two values, and usually just one value.

The cover fieldtree and the loose quadtrees are of interest to the work on point objects in the rest of the dissertation because these structures show how to effectively reduce objects with extents in to point objects. This is achieved by sacrificing the disjoint decomposition property of quadtrees in order to obtain data structures that are almost independent of the size of the objects, thus reducing objects with extents into point objects.

Organization of the Dissertation

The rest of the dissertation is organized as follows.

- **Chapter 2** formally defines spatial networks, and introduces the concepts of path coherence, shortest-path maps, and shortest-path quadtrees. It also discusses the SILC framework, and the distance operators for query processing on spatial networks.

- **Chapter 3** presents a variety of best-first nearest neighbor algorithms on spatial networks using the SILC framework.

- **Chapter 4** describes a distance join algorithm on spatial networks, and several of its variants.

- **Chapter 5** introduces another framework called Path Coherent Pair (PCP) decomposition which improves on the SILC framework by also exploiting the spatial coherence of the source vertices resulting in further reductions in the storage requirements.
- **Chapter 6** describes an approximate distance oracle for spatial networks that is capable of finding an approximate network distance between a vertex pair in $O(1)$ time.
- **Chapter 7** presents an all k nearest neighbor algorithm which has important applications to point-cloud graphics.
- **Chapter 8** analyzes properties of the loose quadtree data structure.
- **Chapter 9** describes future work on query processing on spatial networks, and other geometric algorithms on points.

Chapter 2

SILC Framework

The growing popularity of online mapping services such as Google Maps and Microsoft MapPoint has led to an interest in providing responses 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, restaurants, and campgrounds) where the distance is measured in terms of paths along the network. The 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 and be scalable to handle millions of users. 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 [127] and improvements on them [34]) which usually incorporate Dijkstra's algorithm [40] in at least some parts of the solution [144].

Graph-based algorithms for spatial networks end up visiting not just the set of objects participating in a query, but also a large subset of vertices in the spatial network. The run-time complexity of such algorithms is expressed usually in terms of the number of vertices in a spatial network. We expand on this idea using the example of nearest neighbor finding on a spatial network. Consider a query object q and a set of objects S on the spatial network from which the nearest neighbors of q are drawn. The drawback of graph-based algorithms for finding nearest neighbors is that they visit a large fraction of the vertices of the spatial network, while the neighbors in which we are interested are drawn from S which is usually considerably smaller than the number of vertices in the network. 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, rather 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 in cases where they are reasonably far from each other in terms of network hops. For example, Figure 1.1 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, which is 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).

Our strategy for query processing on spatial networks 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 [154, 186], 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). Using the above formulation, we show that most spatial query processing techniques that were originally developed for traditional spatial databases, can be now applied on spatial networks.

The rest of the chapter is organized as follows. First, we give a formal definition of a spatial network in Section 2.1. As mentioned before, we precompute and store the shortest path and distance between every pair of vertices on a spatial network. In this context, in Section 2.2, we define a path and distance mapping M of a spatial network that takes a pair of vertices in G, and produces the next link in the shortest path as well as the network distance between them. Section 2.3 introduces the concept of path coherence, and discusses strategies for taking advantage of path coherence to arrive at a path mapping that can be represented compactly. Section 2.4 introduces the shortest path quadtrees and the SILC framework. In Section 2.5, we discuss a strategy for encoding the network distance information in a spatial network. Section 2.6 derives a bound on the average size of the shortest path quadtrees of a given spatial network containing n vertices. Section 2.7 presents the experimental results, while Section 2.8 shows how traditional spatial techniques can be applied to the SILC framework. Related work is discussed in Section 2.9. Finally, concluding remarks are drawn in Section 2.10.

2.1 Preliminaries

Spatial networks are general graphs with spatial information augmented with the vertices and edges of the graph. 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 $e \in E$, $w(e) \geq 0$ denotes the distance along that edge. In addition, for every $v \in V$, $p(v)$ denotes the spatial position of v with respect to S, a *spatial domain* also referred to as an *embedding space* (*i.e.*, a reference coordinate system).

A *path* π of size j is a sequence of vertices $(\pi_1, \ldots, \pi_{j+1})$ such that $(\pi_i, \pi_{i+1}) \in E$ for $1 \leq i \leq j$. Notice that a path of size zero is a single vertex. We refer to π_1 as the *source* vertex of π and refer to π_{j+1} as the *destination* vertex of π. Moreover, $\pi(u,v)$ denotes a path with u as its source vertex and v as its destination vertex. The sequence of edges that make up the path π is denoted by the sequence $\varphi(\pi)$, where $\varphi_i(\pi) = (\pi_i, \pi_{i+1})$. Furthermore, the *length* of a path π of size n is $w(\pi) = \sum_{i=1}^{n} w(\varphi_i(\pi))$. Two paths $\pi_1(s,t)$ and $\pi_2(t,u)$ can be composed to form another path π denoted by $\pi_1 \rightsquigarrow \pi_2$. A *subpath* of a path π, is a subsequence of π. The set of vertices that make up the shortest path between a pair of vertices $u, v \in V$ is denoted by $\pi_G(u,v)$. Also, any subpath $\pi(s,t)$ of $\pi_G(u,v)$ is as well the shortest path between s and t. If there are multiple shortest paths of the same length between vertex pairs, extra care must be taken to ensure that the above property holds. In such cases, the first path in the lexicographic ordering on the set of possible shortest paths is chosen, such that the ordering is defined on triples $(w(\pi), n, \text{reverse}(\pi))$. Incidentally, this also assures that each vertex appears at most once in $\pi_G(u,v)$. Furthermore, two sequences π_1 and π_2 are *disjoint*, if and only if $\pi_1 \cap \pi_2 = \varnothing$. Notice that if two paths $\pi_1(s,t)$ and $\pi_2(t,u)$ are disjoint from a path π^* then the path $\pi_1 \rightsquigarrow \pi_2$ is also disjoint from π^*. Also, if π_1 is disjoint from π_2 then any subpath of π_1 is also disjoint from π_2.

For vertices $u,v \in V$, we define $d_G(u,v) = w(\pi_G(u,v))$ to be the *shortest network distance* from u to v with respect to $G(V,E)$. We denote the spatial distance (i.e., "as the crow flies") between vertices, $u,v \in V$ in a spatial network by $d_S(u,v)$, a function on $p(u)$ and $p(v)$. Given a set of vertices $A \subset V$, we define region R to be a *spatial container* of A, iff $\forall v \in A$, $p(v)$ is spatially contained in R. 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 edge on the shortest path from u to v is $(u, l_u(v))$. Furthermore, we define $l'_w(u)$ to be the *last vertex* visited before w on the shortest path from u to w. Note that the last edge on the shortest path from u to w is $(l'_w(u), w)$.

2.2 Path and Distance Mapping

We define a *path-distance* mapping to be a function of the form $M : V \times V \longrightarrow (\mathbb{R}^+, V)$ such that $M(u,v) = (d_G(u,v), l_u(v))$. In other words, given a pair of vertices $u,v \in V$, $M(u,v)$ provides the network distance from u to v and the next vertex in the shortest path from u to v. 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. Using the above representation, the shortest path from u to v can be obtained by the repeated invocation of M, till v is obtained.

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 $k = |\pi_G(u,v)|$ queries on S. So, in effect, the problem of performing scalable query processing on spatial networks depends on the availability of a data structure S that can efficiently perform path and distance queries.

A brute force implementation of S stores two values, $d_G(u,v)$ and $l_u(v)$, for each pair $u,v \in V$. This representation requires $O(n^2)$ storage, but can answer queries in $O(1)$ time. The expensive storage costs involved with such an implementation have led previous researchers [166] to reject the brute force method in favor of alternative methods that approximate the network distance measure.

Traditional approaches to query processing on spatial networks compute the shortest path and the network distance between a pair of vertices at run time, although, such approaches are not really suitable for real-time applications as they are not inherently *scalable*. We propose precomputing the shortest paths and the distances between all pairs of vertices in a spatial network. While this is perceived to be prohibitively expensive, in this chapter we provide firm evidence to the contrary and show that precomputing and storing the path and distance encoding of a spatial network is, in fact, feasible.

The first objection to precomputing the shortest paths between every pair of vertices is that it is perceived to be a time consuming process. We point out that there are a number of techniques that can be used to speed up this process. Dijkstra's algorithm using a Fibonacci heap [53] 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 Dijkstra's algorithm would be $O(n^2 \log n)$. Empirical studies [195] have indicated that 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.* [78] present a linear time shortest path algorithm for planar graphs, while Thorup [178] 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. A host of other techniques like parallel processing and the use of sophisticated hardware such as Graphical Processing Units (GPU) [115] could further speedup the precomputation of all shortest paths of a graph. In this chapter, we will not focus on making shortest path computations faster. Instead, we will assume 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., [44,52,61,62,93,195] as well as the comparative study by Zhang and Noon [195]).

The more difficult problem, that is of interest to us, is that of compactly storing the shortest paths between every pair of vertices in a spatial network and is addressed in this dissertation. Given a spatial network with n vertices, there are $O(n^2)$ possible paths and the cost of storing all of the 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 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} lists, 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 of the 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 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. Note that the sequential search on the list associated with vertex u_i can be avoided by storing a sorted list of vertices which can improve the search time to logarithm in the size of the list. The second is the space required for storing the lists as each list has $O(n)$ elements.

2.3 Path Coherence and Shortest-Path map

The space requirements of the adjacency list 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. 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 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 [187] discuss the reverse problem of assigning spatial positions to vertices of a general graph so that path coherence is *induced* into the the resulting spatial network.

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 $r_{u_i j}$ for each vertex $w_{u_i j}$ ($1 \leq j \leq M_{u_i}$) that is connected to u_i by an edge $e_{u_i j}$. Region $r_{u_i j}$ spans the space occupied by all vertices v such that the shortest path from u_i to v contains edge $e_{u_i j}$ (i.e., the shortest path makes a transition through vertex $w_{u_i j}$). Region $r_{u_i j}$ is bounded by a subset of the edges of the shortest paths from u_i to the vertices within it. Note that $r_{u_i j}$ does not include u_i nor does it include edge $e_{u_i j}$. 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 2.6). For example, Figure 2.1 is such a partition for the vertex marked by X in the road network of Figure 1.1, where we use different colors (i.e., shades of gray) to denote the different regions.

We illustrate the construction of a shortest-path map using a method of assigning *colors* to vertices in a spatial network. The goal of the coloring process is to arrive at a method for efficiently representing the shortest paths and the network distances from a vertex u_i to all other vertices in a spatial network. Suppose that u_i has M_{u_i}

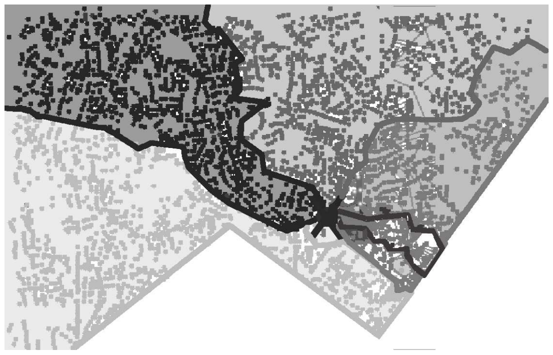


Figure 2.1: Partition of the underlying space spanned by the spatial network in Figure 1.1 into regions r_i such that the shortest path from the vertex marked by X to a vertex in r_i passes through the same vertex among the six vertices that are adjacent to X (i.e., the shortest-path map of X).

adjacent vertices. We use M_{u_i} distinct colors corresponding to each adjacent vertex of u_i. Assuming an ordering of the M_{u_i} adjacent vertices of u_i, we assign color j $(1 \leq j \leq M_{u_i})$ to vertex v, if and only if, $l_{u_i}(v)$ is the j-th adjacent vertex of u_i. 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 M_{u_i} colors. Although we assume that the input spatial network is fully connected, the shortest-path map can also handle spatial networks containing vertices that are not connected to u_i. We use a special color to represent such vertices. For a given vertex u_i, we first compute all the shortest paths from u_i to all other vertices. The coloring operation assumes that the shortest path from u_i to all other vertices in the spatial network is available to the coloring process, which is described in Algorithm 1. In line 1, the shortest path (SSSP) from u_i to all the other vertices in the spatial network is computed. In lines 2–3, all vertices $v \in V - \{u_i\}$ are assigned colors based on which adjacent vertex of u_i forms the first link in the shortest path from u_i to v.

Algorithm 1

```
Procedure ColorizeMap[$u_i$]
1.  Compute SSSP($u_i$)
2.  for each $v \in V$ and $v \neq u_i$ do
3.      assign $color[v] = color[l_{u_i}(v)]$
4.  end-for
5.  return
```

Figure 2.1 illustrates the coloring process. Once the coloring operation has been completed, the spatial network has large contiguous colored regions; the resulting representation is the shortest-path map of u_i. In particular, v and w belonging to the region denoted by color j means that the first link in the shortest path from u_i to v and w is the same. The shortest-path map of u_i is a decomposition of the space into M_{u_i} regions such that vertices contained in a region share the first link in the shortest path from u_i. The large contiguous colored regions are due to the *path coherence* between the vertices.

2.4 Shortest Path Quadtrees

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 spatial indexes: one based on an object hierarchy such as an R-tree [73] and one based on a disjoint decomposition of the underlying space such as one of a number of quadtrees variants (e.g., [45, 124, 140–142, 144]).

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 non-disjoint 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 $b_{u_i j}$, and hence the region $r_{u_i j}$ corresponding to the first vertex on the shortest path from u_i to v, is to associate the relevant vertices with $b_{u_i j}$ 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., [44, 61, 62, 93]) 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 [186] where the object hierarchy consists of bounding boxes. Figure 2.2(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 1.1. 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.

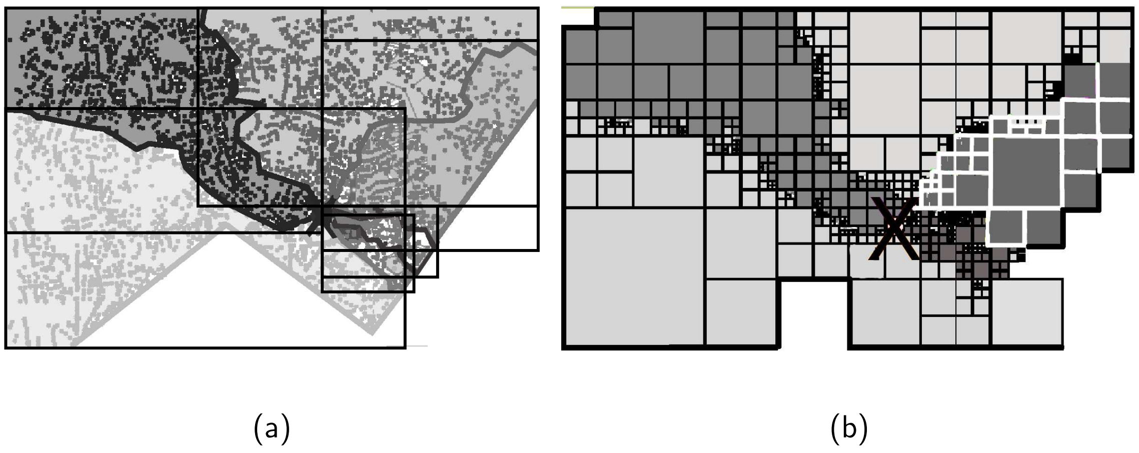


Figure 2.2: (a) Result of using minimum bounding boxes to approximate the regions in the partition for the vertex marked by X in the road network of Figure 2.1, and (b) the leaf blocks in the shortest-path quadtree for the regions of the same partition.

In contrast, we make use of a spatial index that is based on a disjoint decomposition of the underlying space. In particular, we represent the regions that make up the shortest-path map m_{u_i} using a variant of the region quadtree [87, 88, 142], termed a *shortest-path quadtree*, where there are M_{u_i} different disjoint regions r_{u_ij} all stored in the region quadtree s_{u_i}. Each region r_{u_ij} 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 2.2(b) is the block decomposition induced by the shortest-path quadtree on the shortest-path map given by Figure 2.1.

This leads us to our proposed framework, termed *SILC*[1] framework, which is an efficient representation of a path-distance mapping S. We now describe the organization of the SILC framework which stores the shortest-path quadtree for every vertex in the spatial network. We claim that such a representation efficiently captures the shortest path and network distance information between every pair of vertices in a given spatial network. The resulting representation is both computationally efficient (*i.e.*, provides efficient path and distance retrievals) and storage efficient (*i.e.*, less storage per vertex).

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 necessary 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

[1]SILC (pronounced *silk*) is a tribute to the ancient trading routes of antiquity known as the *silk roads*.

regions. For example, the boundaries of the regions could be represented by a variant of an MX quadtree [87, 88, 142] where the 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 the 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 of the 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 out-degree of the vertex with which the shortest-path quadtree is associated. However, as we discuss in Section 2.6, 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., [58]). 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* [118], 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 non-leaf blocks by using a variant of a path-compressed PR quadtree (e.g., [36, 74]) which ignores white blocks where all but one of the siblings are white.

Algorithm 2 describes an efficient method of constructing a shortest-path quadtree of a vertex v from an initial bucket PR quadtree [124, 142, 144] of bucket capacity $c > 0$ on the spatial position of vertices (line 2). Of course other representations could have been used as the input to the algorithm, but we leveraged our existing SAND spatial database system [146] that provided a robust disk-based bucket PR quadtree implementation. Blocks in the bucket PR 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 of the blocks may have to be split in order to ensure that all the vertices that they contain are of the same color. The resulting representation is a shortest-path quadtree s_{u_i} of u_i. Now, each leaf-block in s_{u_i} of u_i is represented as a single Morton block and is stored on disk. Next, a B$^+$-tree access structure is imposed on the sorted list of Morton blocks to speed-up point location operations on them. We omit explaining Lines 12–16 which deal with storing some network distance information along with each Morton block in s_{u_i}. They are explained in Section 2.5.

Algorithm 2

Procedure ConstructShortestPathQuadtree$[u_i, T]$

Input: $u_i \in V$

Input: T is a bucket PR quadtree on V

Output: $s_{u_i} \leftarrow$ shortest-path quadtree of u represented as a sorted list of Morton blocks

```
1.  for each leaf-block $b \in T$ visited in Morton order do
2.      if all vertices in $b$ are of same color (i.e., $b$ is single colored) then
3.          append $b$ to $s_{u_i}$
4.      else
5.          recursively split $b$ until all blocks in the resultant set $s_b$ are single colored
6.          merge $s_b$ with $s_{u_i}$
7.      end-if
8.  end-for
9.  while blocks in $s_{u_i}$ can be merged do
10.     merge sibling blocks in $s_{u_i}$ if of the same color
11. end-while
12. for each Morton block $b \in s_{u_i}$ do
13.     $\lambda^- = \min_{v \in b} \frac{d_G(u_i,v)}{d_S(u_i,v)}$
14.     $\lambda^+ = \max_{v \in b} \frac{d_G(u_i,v)}{d_S(u_i,v)}$
```

```
15.      associate $(\lambda^-, \lambda^+)$ with $b$
16.  end-for
17.  return $s_{u_i}$
```

Quadtree representations of regions have been shown to be good dimensionality reducing mechanisms [87, 88, 141, 142], 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 out-degree of the vertex, which is relatively small. Our experiments (see Section 2.7) show that the storage requirements for the SILC framework on road networks are achievable, that is, the storage per vertex is considerably smaller than $O(n)$ which represents a considerable improvement over the brute force encoding.

To improve the storage requirements even further, a number of alternate representations are suggested. We may choose not to store the colored region that takes the maximum number of Morton blocks to represent. This may result in some savings in storage, although path retrievals may become slightly more expensive. The shortest-path map, as seen in Figure 2.1, has radial structures. A simple transformation of the space to polar coordinates may help improve the storage costs. *Chain Code* techniques [54] or variations of *Medial Axis Transformation* (MAT) techniques like *Corner MAT* [144] and *Quadtree MAT* [139] could be used instead of representing regions as a set of Morton blocks. However, unlike the Morton blocks, they may not allow for efficient operations on them.

2.4.1 Retrieving Shortest Paths

As pointed out earlier, the advantage of the SILC framework which makes 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 associated with u, the 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. Also, the network distance between u and v can be obtained by summing up the weights of each individual edge comprising the shortest path. Thus, we see that the SILC framework explicitly encodes the path information, while the distance information is implicitly recorded.

Given a source vertex u and destination vertex v, Algorithm 3 describes an algorithm to obtain the next vertex in the shortest path from u to v. The point location query on s_u that retrieves a block b containing v is aided by the B$^+$-tree on the Morton blocks in s_u. Notice that retrieving the entire shortest path between u and v requires exactly

$k = |\pi(s,v)|$ invocations of the NEXTVERTEXSHORTESTPATH routine resulting in k disk accesses.

Algorithm 3

```
Procedure NEXTVERTEXSHORTESTPATH[$u$, $v$]
Input: $u$ is the source vertex, and $v$ is the destination
Output: $t$ is the next vertex in the shortest path from $u$ to $v$
Output: $b$ is the Morton block in the shortest-path quadtree $s_u$ of $u$ that contains $v$
Output: $w(u,t)$ is the edge weight of $(u,t)$
1.  retrieve the shortest-path quadtree $s_u$ of $u$
2.  Search on $s_u$ for a block $b$ containing $v$
3.  (* COLOR(.) returns vertex $w_j$ associated with the region $r_j$ in the shortest-path
    map $m_u$ of $u$ containing $b$ *)
4.  $t \leftarrow$ COLOR($b$)
5.  return $t$, $b$, $w(u,t)$
```

For example, consider the simple road network given in Figure 2.3(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 2.3(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 2.3(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.

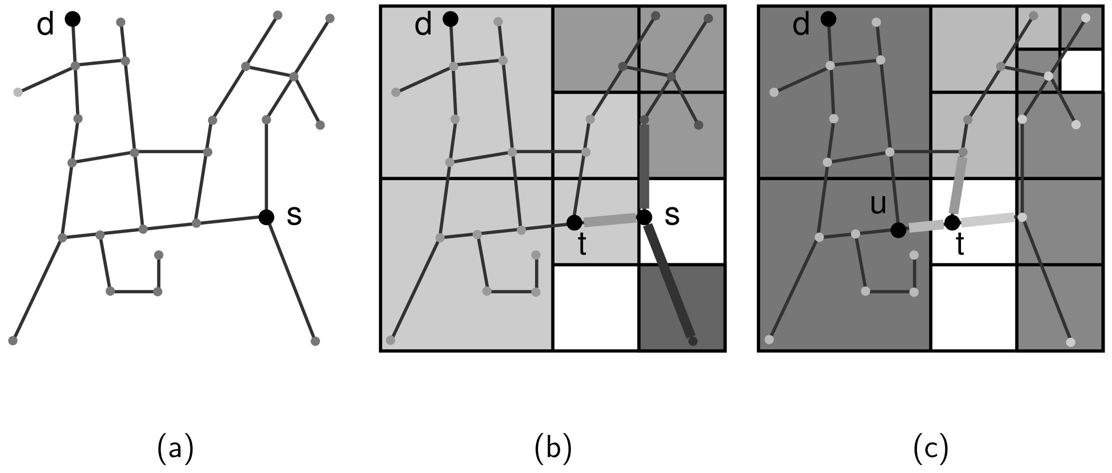


Figure 2.3: (a) Example road network, (b) the shortest-path quadtree of vertex s, and (c) the shortest-path quadtree of vertex t.

2.5 Distance Encoding

The SILC framework explicitly encodes the shortest path between every pair of vertices, while the network distance is implicitly recorded. That is, given a pair of vertices u, v in a spatial network, the shortest-path quadtree of u records the next link in the shortest path from u to v. The network distance between u and v is obtained once the complete shortest path from u to v is obtained using the framework. We point out that the output of many spatial operations is an ordering of a set of objects P on the spatial networks based on their network distance to a query object q, or a set S_q of query objects. For example, a nearest neighbor query is a total ordering of objects in P (restaurants) in an increasing order of their network distances from q (office), such that the closest object to q in P is the first object in the ordering of objects in P with respect to q, and so on. Similarly, a distance join operator is similar to a nearest neighbor operation except that we now have a set S_q of query objects as opposed to a single query object q.

Suppose the SILC framework is capable of providing an approximate network distance, in the form of an network distance interval $[\delta^-, \delta^+]$, such that the actual network distance between a pair of vertices is contained by the network distance interval. For example, given $u, v \in V$, the SILC framework is capable of obtaining a network distance interval $[\delta^-, \delta^+]$ such that $\delta^- \leq d_G(u,v) \leq \delta^+$, where δ^- is the *minimum network* distance of v from u and δ^+ is the *maximum network* distance of v from u. Moreover, let us suppose that the network distance interval can be obtained without expending too much work. We claim that the network distance interval is sufficient to answer queries that require the total ordering of objects based on their network distances to a query object, or a set of query objects. For example, if $p \in P$ is guaranteed to be the closest neighbor to q, if the maximum network distance of p from q is less than the minimum network distance of all other objects in P. 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. Notice that we are able to establish orderings of objects in P without even computing the exact network distance between q and the objects in P.

We illustrate the above idea with the following example. Suppose we would like to find which of the two cities—“Washington” or “Princeton”— is closer to “College Park”? Suppose, the network distance interval, provided by the SILC framework, from College Park to Washington is between 10 and 30 miles, while the network distance interval from College Park to Princeton is between 50 and 100 miles. We can safely conclude that Washington is closer to College Park than Princeton, as the maximum possible distance from College Park (30) to Washington is less than the minimum possible distance (50) to Princeton. Now, instead suppose that the network distance interval from College Park to Princeton is $[25,100]$ miles, in which case, it is not clear if Princeton or Washington is closer to College Park as the distance intervals corresponding to College Park and Washington are *intersecting*. In order to handle such cases, we define an *refinement* operator that takes a network distance interval and *tightens* it. That is, when applied to the network distance interval $[25,100]$ the refinement operator tightens it; say makes it $[75,95]$; in which case, we will be able to answer our original query without any ambiguity.

We above example illustrates the concept of *progressive refinement* of the network distance between objects on a spatial network which allows for *work efficient* algorithms on spatial networks. Our framework allows for the quick computation of an *initial network distance interval* between two objects on the spatial network. We then provide a mechanism, termed *refinement*, by which the network distance interval can be made *tighter* by expending more work. This allows for efficient algorithms that only perform as many *refinements* so as to be able to answer a query without ambiguity.

2.5.1 Network Distance Interval

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, we store 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^- and λ^+ that correspond respectively to the minimum and 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 as shown in lines 12–16 of Algorithm 2. 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 value of the ratio attains its minimum value need not necessarily 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, which is given below.

Algorithm 4

Procedure GETNETWORKDISTINTERVAL[v, s_q]

Input: $v \leftarrow$ destination vertex

Input: $s_q \leftarrow$ shortest-path quadtree of a vertex q

Output: network distance interval $[\delta^-, \delta^+]$ containing $d_G(q, v)$

```
1.  retrieve $\lambda^-$ and $\lambda^+$ from block $b$ in $R$ containing $v$
2.  $\delta^- \leftarrow \lambda^- \times d_S(q, v)$
3.  $\delta^+ \leftarrow \lambda^+ \times d_S(q, v)$
4.  return $[\delta^-, \delta^+]$
```

Definition 1. *Given a source vertex u and a destination vertex v, we define $\pi_G(u, v)$ to be the ordered set of vertices in the shortest path from u to v. Also, let $|\pi_G(u, v)|$ denote the number of vertices in the shortest path from u to v.*

Lemma 2.1. *Given that w is an intermediate vertex in the shortest path from u to v, we* *have that $d_G(u,v) = d_G(u,w) + d_G(w,v)$.*

Definition 2. *Given an interval $[a,b]$ and a value t, such that $a,b,t \in \mathbb{R}$, $[a,b]$ is said to* contain *t iff, $a \leq t \leq b$.*

Lemma 2.2. *Given a source vertex u and a destination vertex v and the shortest-path quadtree R of u, the network distance interval $[\delta^-, \delta^+]$ from u to v, obtained using* GETNETWORKDISTINTERVAL*(v, R), contains the actual distance distance between u and v.*

2.5.2 Refinement Operator

Whenever it is determined that the initial network distance interval $[\delta^-, \delta^+]$ is not sufficiently tight (i.e., in the context of nearest neighbor finding, we can say that a network distance interval is *tight* if it contains just one neighboring object—that is, 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 (given in Algorithm 3). 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 first summing the network distance from q to t (i.e., the weight of the edge from q to t) and $[\delta_t^-, \delta_t^+]$, and then taking the intersection of the network distance interval between q and

v before the invocation of the refinement operator as shown in lines 3–4 of Algorithm 5. Given a pair of vertices q and v and a length k for the shortest path between them, this process can be reinvoked at most another $k-2$ times until reaching v.

Algorithm 5

```
Procedure REFINENETWORKDISTINTERVAL[$s$, $u$, $v$, $d$, $[\delta^-, \delta^+]$]
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: network distance interval $[\delta^-, \delta^+]$ containing $d_G(s,v)$
Output: $t$ is the next vertex in the shortest path
Output: $d$ holds $d_G(s,t)$
Output: $[\delta^-, \delta^+]$ are the updated interval on $d_G(s,v)$
1.  $(t, b, d) \leftarrow$ NEXTVERTEXSHORTESTPATH($s$, $u$, $v$, $d$)
2.  Obtain $\lambda^-$ and $\lambda^+$ from $b$ (or using GETNETWORKDISTINTERVAL)
3.  $\delta^- \leftarrow \max(\delta^-, \lambda^- \times d_S(t,v) + d)$
4.  $\delta^+ \leftarrow \min(\delta^+, \lambda^+ \times d_S(t,v) + d)$
5.  return $t$, $d$, $[\delta^-, \delta^+]$
```

We now present a few interesting properties of the refinement operator.

Definition 3. *Given a source vertex u and a destination vertex v, let* $[\delta_i^-, \delta_i^+]$ *be the* *network distance interval between u and v after i refinements, such that* $[\delta_0^-, \delta_0^+]$ *is the initial network distance interval from u to v.*

Definition 4. *The ith invocation of the refinement operator on the network distance interval from a source vertex u to a destination vertex v takes an intermediate vertex w, the initial network distance interval* $[\mu_0^-, \mu_0^+]$ *from w to v, the network distance* $d_G(u,w)$ *from u to w, the current network distance interval* $[\delta_{i-1}^-, \delta_{i-1}^+]$ *and produces* $[\delta_i^-, \delta_i^+]$*, such that* $\delta_i^- = \max(\delta_{i-1}^-, d_G(u,w) + \mu_0^-)$*, and* $\delta_i^+ = \min(\delta_{i-1}^+, d_G(u,w) + \mu_0^+)$.

Lemma 2.3. *Given a source vertex u and a destination vertex v, and the initial network distance interval from u to v, the network distance interval from u to v converges to* $d_G(u,v)$ *after at most* $|\pi_G(u,v)| - 1$ *refinements.*

Lemma 2.4. *Given a source vertex u and destination vertex v, the network distance interval* $[\delta_i^-, \delta_i^+]$ *from u to v after i refinements,* $[\delta_i^-, \delta_i^+]$ *always contains* $d_G(u,v)$ *and* $\delta_i^- \leq \delta_i^+$*, such that* $1 \leq i \leq |\pi_G(u,v)| - 1$.

Proof. We show the following using mathematical induction. When $i = 0$, we know from Lemma 2.2 that the initial distance interval $[\delta_0^-, \delta_0^+]$ between u and v always contains $d_G(u,v)$. We now show that the Lemma also holds for $i > 0$.

Let w be an intermediate vertex in the shortest path from u to v. Furthermore, let us suppose that w is the intermediate vertex that is currently being used by the refinement operator. Let R be the shortest-path quadtree of w. Let $[\delta_{i-1}^-, \delta_{i-1}^+]$ be the network distance interval from u to v before the refinement operation. We assume that the network distance interval $[\delta_{i-1}^-, \delta_{i-1}^+]$ contains $d_G(u,v)$, that is –

$$\delta_{i-1}^- \leq d_G(u,v) \leq \delta_{i-1}^+. \tag{2.1}$$

We now show that after the refinement operation, the resulting network distance interval $[\delta_i^-, \delta_i^+]$ also contains $d_G(u,v)$.

First of all, we know from Lemma 2.2 that the network distance $[\mu_0^-, \mu_0^+]$ produced by the procedure GETNETWORKDISTINTERVAL(v, R) contains the network distance between w and v, that is –

$$\mu_0^- \leq d_G(w,v) \leq \mu_0^+. \tag{2.2}$$

Let $[\Delta^-, \Delta^+]$ be an interval, such that $\Delta^- = \mu_0^- + d_G(u,w)$ and $\Delta^+ = \mu_0^+ + d_G(u,w)$. By adding the term $d_G(u,w)$ to both sides of the Equation 2.2, we get:

$$\mu_0^- + d_G(u,w) \leq d_G(u,w) + d_G(w,v) \leq \mu_0^+ + d_G(u,w).$$

The above inequality reduces to:

$$\Delta^- \leq d_G(u,w) + d_G(w,v) \leq \Delta^+,$$

$$\Delta^- \leq d_G(u,v) \leq \Delta^+.$$

We know from the definition of the refinement operation that $\delta_i^- = \max(\Delta^-, \delta_{i-1}^-)$ and $\delta_i^+ = \min(\Delta^+, \delta_{i-1}^+)$. If we combine the above Equation and Equation 2.1, we get four inequalities, of which only one is true depending on the relative values of $\Delta^-, \delta_{i-1}^-, \Delta^+$, and δ_{i-1}^+.

$$\delta^-_{i-1} \leq \Delta^- \leq \quad d_G(u,v) \quad \leq \delta^+_{i-1} \leq \Delta^+$$

$$\Delta^- \leq \delta^-_{i-1} \leq \quad d_G(u,v) \quad \leq \delta^+_{i-1} \leq \Delta^+$$

$$\delta^-_{i-1} \leq \Delta^- \leq \quad d_G(u,v) \quad \leq \Delta^+ \leq \delta^+_{i-1}$$

$$\Delta^- \leq \delta^-_{i-1} \leq \quad d_G(u,v) \quad \leq \Delta^+ \leq \delta^+_{i-1}$$

Generalizing the above four inequalities, we get:

$$\max(\delta^-_{i-1}, \Delta^-) \leq d_G(q,p) \leq \min(\delta^+_{i-1}, \Delta^+), \tag{2.3}$$

which reduces to the required result:

$$\delta^-_i \leq d_G(q,p) \leq \delta^+_i. \tag{2.4}$$

Moreover, $\delta^-_i \leq \delta^+_i$ follows from Equation 2.4. Hence, proved. □

2.5.3 Network Distance Primitives for Blocks

We now show how to compute the network distance between a query vertex q and a spatial region b of the search hierarchy T. For the sake of simplicity, we will assume that b is an axis aligned rectangular search region. 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, given in Algorithm 6. 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 [73]), and even if they are both quadtree-based (e.g., T is a PR quadtree [124, 142]), s_q and T do not necessarily need to be in registration (i.e., they can have different origins, as can be seen in Figure 2.4).

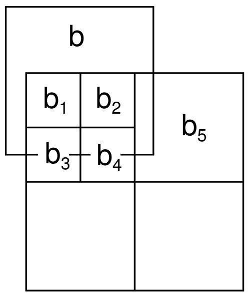


Figure 2.4: Example of the intersection of block b in a quadtree search hierarchy T with blocks b_1, b_2, b_3, b_4, b_5 in the shortest-path quadtree.

We now describe the workings of Algorithm MINNETWORKDISTBLOCK. We first define the UNION of two network distance intervals d_1, d_2 to be the tightest interval containing both d_1 and d_2. The result of applying the MINNETWORKDISTBLOCK operator on a vertex q and a region b is a network distance interval such that the interval contains the actual network distance from q to each vertex contained in b. Note that MINNETWORKDISTBLOCK operator is analogous to the MINDIST and MAXDIST operators in spatial databases. MINNETWORKDISTBLOCK operator returns a network distance interval $[\delta^-, \delta^+]$ such that for any vertex t contained in R, $\delta^- \leq d_G(q,t) \leq \delta^+$. We are able to compute suitable values for δ^- and δ^+ by using the MINDIST and MAXDIST distances between q and b, for each of the blocks in the shortest-path quadtree of q that is intersected by b as shown in Algorithm 6

Algorithm 6

```
Procedure MINNETWORKDISTBLOCK[$q$, $b$, $s_q$]
Input: $b$ is a region, $q$ is a vertex
Input: $s_q$ is the shortest-path quadtree of $v$
Output: $[\delta^-, \delta^+]$ forms the distance interval
1.  Compute the set $B_q$ of blocks in $s_q$ intersected by $b$
2.  for each $b_i \in B_q$ intersecting $b$ do
3.      retrieve $\lambda^-$ and $\lambda^+$ from $b_i$
4.      $r_i \leftarrow$ region formed by the intersection of $b_i$ and $b$
```

```
5.        $\mu_i^- \leftarrow \lambda^- \times \text{MinDist}(q, r_i)$
6.        $\mu_i^+ \leftarrow \lambda^+ \times \text{MaxDist}\ (q, r_i)$
7.    end-for
8.    $[\delta^-, \delta^+] \leftarrow$ Union of all $[\mu_i^-, \mu_i^+]$
9.    return $[\delta^-, \delta^+]$
```

To summarize, the SILC framework explicitly encodes the shortest path and a network distance interval (approximate network distance) between all pairs of vertices, while the actual network distances between vertices are implicitly recorded. A shortest-path quadtree s_u is associated with each vertex u in a spatial network, such that s_u is represented as an ordered set of Morton blocks. Furthermore, each Morton block in s_u also records a link (color) and a pair of ratios λ^+ and λ^-. The link color and the ratios are collectively referred to as the *pilot-data* of a vertex.

2.6 Space Requirements for the Shortest-Path Quadtree

In this section, we present bounds on the size of the shortest path quadtree by appealing to the *Quadtree Complexity Theorem* [87, 88, 142], which states that given a connected region R in a binary image, the number of blocks in a MX quadtree [144] on R requires $O(p)$ space, where p is the perimeter of R. Unfortunately, deriving the space requirements of a shortest path quadtree is much more involved than the direct application of the Quadtree Complexity Theorem to shortest-path quadtrees. This is because the shortest-

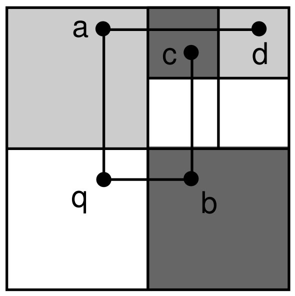


Figure 2.5: Example illustrating the presence of empty blocks in the shortest-path quadtree of the shortest-path map of a vertex q consisting of the non-contiguous quadtree blocks containing a and d, and one for vertex b consisting of the non-contiguous quadtree blocks containing vertices b and c.

path quadtree constructed on the regions of a shortest-path map need not be contiguous, as shown in the example described in Figure 2.5 and hence, the Quadtree Complexity Theorem may not be directly applicable to shortest-path quadtrees. However, we circumvent this problem by first showing that the shortest-path map corresponding to a vertex is always contiguous. We then derive a bound on the size of a MX quadtree on the colored regions in the shortest-path map of a vertex by the application of the Quadtree Complexity Theorem. Finally, we show that the size of the MX quadtree on the shortest-path map always upper bounds the size of the shortest-path quadtree.

At the onset, we assume that the underlying graphs that form the basis of the spatial networks are assumed to be connected planar graphs. This is not an unreasonable assumption as road networks are connected (at least within a landmass such as a continent), although it is not necessary for them to be planar as can be seen by the possibility of the presence of tunnels and bridges. This leads us to the following result on shortest-path maps for spatial networks that are planar.

Theorem 2.1. *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 of the vertices that make up each of the regions r_{u_ij} 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 2.2. *The shortest-path quadtree for vertex u_i requires $O(p_{u_i}+t)$ 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 $2^t \times 2^t$ 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_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}. From Theorem 2.1 we know that each of r_{u_ij} is connected. Now, for each region r_{u_ij} of u_i, apply an algorithm to determine its boundary which results in a polygon o_{u_ij} and build an MX quadtree t_{u_ij} for its edges. Assuming that t_{u_ij} is embedded in a $2^t \times 2^t$ space, we know from the Quadtree

Complexity Theorem (e.g., [87, 88, 142]) that $t_{u_i j}$ requires $O(p_{u_i j}+t)$ space, where $p_{u_i j}$ is the perimeter of $o_{u_i j}$. Next, construct X_{u_i}, the union of the MX quadtrees corresponding of the regions that make up m_{u_i} which will require $O(p_{u_i}+t)$ 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 2.4, we 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 of the 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 $O(p_{u_i}+t)$ space requirements of the MX quadtree X_{u_i} also hold for the shortest-path quadtree S_{u_i}. □

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 2.2, 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 1.1, which has 4333 vertices, found that, using S_{u_i}, the number of blocks in each of the shortest-path quadtrees for all of the 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 by converting the collection of polygons described in the proof of Theorem 2.2 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 [153] or even the PMR quadtree [122, 123]. Their space requirements have been analyzed in [109] 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 2.2.

We now prove the main result.

Theorem 2.3. *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 region 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. The embedding of 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 2.2 in Section 2.6 we have that the 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 [24] 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 $O(n^{1.5})$. □

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 2.5. This occurs when a 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 2.5 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 2.2 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 $r_{u_i j}$) 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.

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

Figure 2.6: Sample datasets.

2.7 Experiments

The SILC framework presented in this chapter provides a compact representation of the path and distance encoding of 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 [183] and the National Atlas [184] websites. Some of the datasets that we used are shown in Figure 2.6.

The framework presented in the chapter 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 average number of Morton blocks comprising the shortest-path quadtree stored with each vertex in the SILC framework depends on n and grows gracefully as n gets larger; (ii) The average number of Morton blocks comprising the shortest-path quadtrees stored with each vertex directly affects the time taken to perform the path and distance computations.

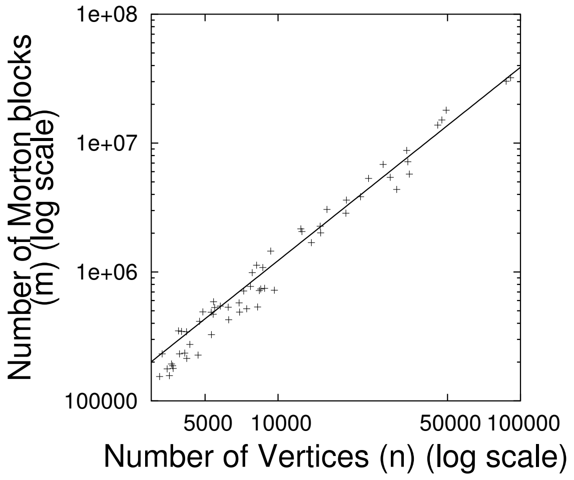


Figure 2.7: The total number of Morton blocks in the shortest path quadtree encoding of random subgraphs extracted from a larger dataset, as well as a line with slope 1.5.

The first set of experiments is important as they reinforce our claims (and proofs) that the shortest-path quadtrees provide a compact, storage-wise, alternative to storing the complete shortest path between every pair of vertices of the spatial network. In particular, our experiments demonstrate that the storage requirements of the shortest-path quadtrees are proportional to the number of vertices in the spatial network. Our experiments estimated the size of the shortest path quadtree for a variety of spatial networks by taking random samples from a large road-network dataset. In particular, we used a dataset containing all of the 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, we determined the shortest-path quadtree for each of its vertices and calculated the number of Morton blocks comprising it and then obtained their sum.

Figure 2.7 shows that the ratio of the total number M of Morton blocks in the shortest-path quadtrees for a spatial network s to the number of vertices n in s for a wide range of spatial networks of different sizes obeys $M = c \cdot n^{1.5}$ (where c is a constant). This has a very important ramification as it reduces the storage complexity of our approach from $O(n^3)$ to $O(n^{1.5})$ as given a spatial network with n vertices, we have $O(n^2)$ paths and in the worst case each shortest path can contain $O(n)$ vertices. In contrast, our representation requires $\sqrt{n}$ Morton Blocks on the average for each of the vertices for a total of $O(n^{1.5})$ space. This makes the shortest-path quadtree representation scalable as the total amount of space required for a spatial network has been significantly reduced (i.e., by an order of magnitude equal to the square root as $n^{1.5}$ is the square root of n^3).

Moreover, our shortest path quadtree experiments validate our results on the space complexity of the shortest path quadtrees in Theorem 2.3 in Section 2.6 which states that their space complexity depends on the sum of the perimeters of the polygons that make up the shortest path quadtrees. There are usually four polygons for each shortest path quadtree as this is the most likely degree of the vertices in the spatial network as they usually represent the intersection of two roads and given a fixed image resolution, the inherent digitization of the line segments that make up the boundaries of the polygons leads to the sum of the perimeters of the polygons being relatively constant. This observation agrees well with our earlier theorems on the dimension-reducing property of the shortest path quadtree representation.

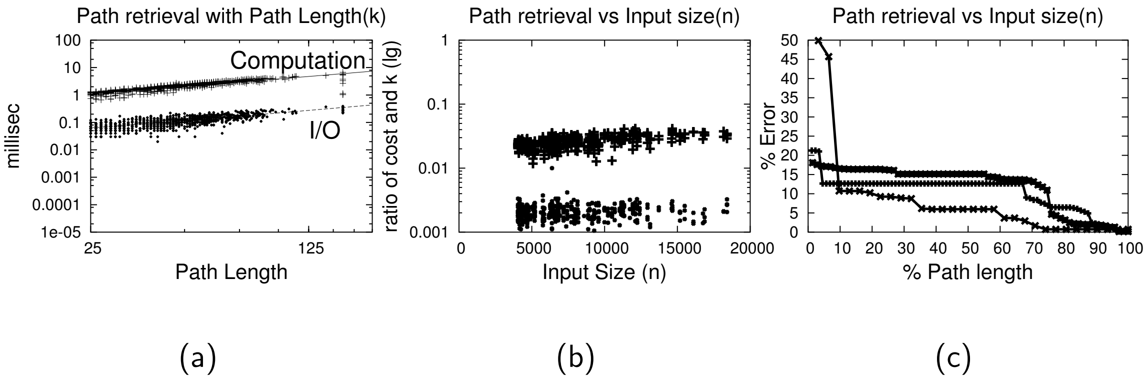


Figure 2.8: a) CPU time (top) and I/O time (bottom) to retrieve the shortest path between two arbitrary vertices versus the length of the path between them for the Silver Spring, MD map. b) CPU time (top) and I/O time (bottom) normalized by path length versus the size (i.e., number of vertices) of a randomly chosen rectangular sample of the data in a large USA map. 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.

In the first set of experiments shown in Figure 2.8a, 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 2.8a 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 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 [184], 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 2.8b. 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 REFINE-NETWORKDISTINTERVAL operator used in Algorithm 5. 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 REFINE-NETWORKDISTINTERVAL operator tightens the interval as the path is computed by incurring one additional disk access and is important to processing the different spatial queries on spatial networks. Figure 2.8c 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 2.8c corresponds to one REFINENETWORKDIST-INTERVAL operation. From the figure we observe that as we approach the destination, the error quickly reduces to a small value.

2.8 Applications

The SILC framework enables the use of many well known query processing techniques— that were developed for classic spatial databases —on spatial networks. Some of the examples include:

1. **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) as described in Section 2.4.1.

2. **Incremental nearest neighbors:** Retrieve the nearest neighbors to a query object in an incremental fashion (e.g., find the nearest hospitals to the accident scene in

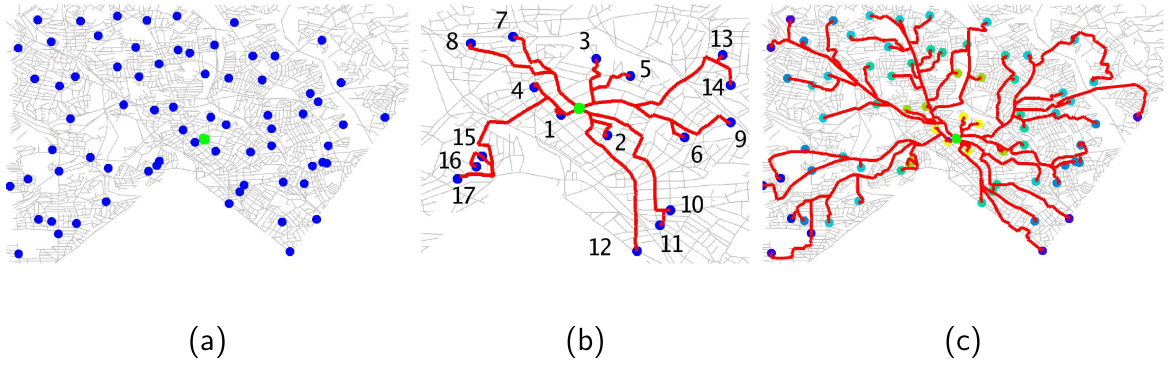
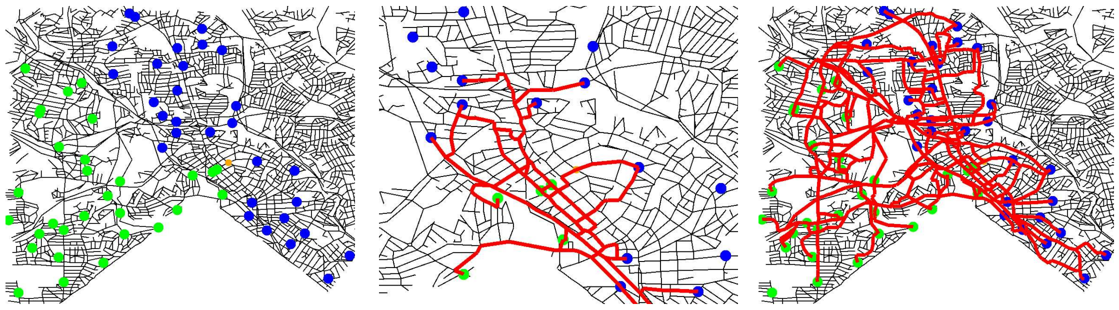


Figure 2.9: Mechanics of a nearest neighbor search [83] on a road network. a) Initial configuration: A query point (denoted by "X") and a set of locations filled circles. b) Query progression: Partial result of ranking the dataset of points based on the length of their shortest path from the query point. 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.

the increasing order of the trip time). One such algorithm is described in Chapter 3. The incremental nearest neighbor algorithm can also be modified to find all locations that lie within a distance of r from a specified query object (e.g., find all hospitals that are within a distance of one mile – or alternately, can be reached within five minutes of driving – from the accident scene).

3. **Distance join and distance semi-join:** Given two sets of spatial objects, S and R, the distance join retrieves all pairs of objects. Distance semi-join [82] requires that objects from S appear only once in the output (e.g., given a set of stores and another set of warehouses, the distance semi-join reduces the number of pairs by finding a unique (e.g., closest) element in S for each element in R). Such



(a) (b) (c)

Figure 2.10: Mechanics of an incremental distance join [82] 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.

algorithms are described in Chapter 4.

2.9 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 Dijkstra's algorithm [40]. A variation of Dijkstra's algorithm that uses a Fibonacci heap structure [53] 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). Dijkstra's algorithm requires that all edges have non-negative weight. The Floyd-Warshall algorithm [49], 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 [195] compares the relative performance of many of the classical shortest path algorithms when applied on a road-network dataset.

The *Incremental Euclidean Restriction* (IER) and *Incremental Network Expansion* (INE) techniques of Papadias *et al.* [127], and the subsequent improvements to the INE technique by Cho and Chung [34], use a technique that is based on Dijkstra's shortest path algorithm to find the k-nearest neighbors to a query object q on a spatial network. The difference between these approaches and Dijkstra's algorithm is that, in the case of the INE and IER techniques, the input objects are indexed using a R-tree [73] structure and hence, the objects are *decoupled* from the network representation.

The IER technique first obtains the k Euclidean nearest neighbors to a query object q using an incremental nearest neighbor algorithm, such as the one provided by [83]. Now, the network distances to each of the k Euclidean nearest neighbors from q is computed and the objects are sorted in an increasing order of their network distance from q. Let d_k^q be the network distance to the current kth nearest neighbor of q. The $(k+1)$th Euclidean nearest neighbor p of q is obtained by the algorithm and the network distance of p from q is also computed. If the network distance of p from q is less than d_k^q, then the ordered set of k nearest neighbors is updated and the value of d_k^q is updated as well. This process is repeated until the algorithm retrieves an Euclidean nearest neighbor of q whose Euclidean distance to q is farther than d_q^k, in which case the algorithm is terminated as the Euclidean distance between two objects is assumed to be a lower bound of the network distance between them.

The INE algorithm, on the other hand, closely resembles Dijkstra's algorithm. It uses a priority queue Q of vertices and visits the vertices of a spatial network G in a *best first* order. It also maintains an ordered list L (initially empty) of the k closest objects found thus far by the algorithm. The objects in L are sorted in an increasing order of their network distance from q. Whenever a vertex u is visited by the algorithm, all the objects that are associated with the outgoing edges of u are retrieved and inserted into the L, while taking care not to introduce duplicate instances of the same object in L. Let d_k^q be the network distance to the current kth nearest neighbor of q in L, such that d_k^q is set to ∞ if L contains fewer than k objects. When a vertex p is retrieved from the top of Q, such that the network distance of p from q is greater than d_k^q, the algorithm terminates. As both the INE and IER algorithms are based on Dijkstra's algorithm, they visit a large number of vertices in a spatial network. Moreover, in the case of INE, there is an added cost of performing region searches on the R-tree containing the input objects every time the algorithm visits a new vertex in G, which makes these algorithms expensive. The implementation of the INE algorithm in [127] produces incorrect result when the input objects lie on the edges of a spatial network. Samet *et al.* [151] provide an alternate implementation of the INE algorithm that corrects this problem, although experimental results show that the corrected version of the INE algorithm is at least two times slower than the original algorithm in [127].

Of particular interest are techniques that deal with disk-based representations and bucketing [89, 127] 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.* [93], and the more recent work by Filho and Samet [44], 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.* [116] describe an algorithm for computing shortest paths on 3D meshes, and Surazhsky *et al.* [176] demonstrate an effective implementation of the algorithm. Note that the SILC framework is also applicable to 3D meshes.

Wagner and Willhalm [186] 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 $S(e)$, as shown in Figure 2.2a. 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 [63] 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 [62] describe an implementation of the ALT algorithm on a hand held 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.* [166] is similar to the work of Goldberg and Harrelson [61]. 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* [14]. In a related note, the recent work by Gupta *et al.* [71] 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.* [186] and the ALT method [61] of Goldberg and Harrelson. Wagner *et al.* in [186] 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 *et al.* [61,62] and the RNE method by Shahabi [166] 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.

2.10 Summary

In this chapter, we presented the SILC framework which allows for the 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. We first introduced the concept of a *path-distance* mapping that given a pair of vertices u,v provides the the network distance from u to v and the next vertex in the shortest path from u to v. The SILC framework is an implementation of a path-distance mapping such that it provides the next vertex in the shortest path from u to v as well as a network distance interval $[\delta^-,\delta^+]$ such that $[\delta^-,\delta^+]$ contains the actual network distance between u and v.

The framework is based on an observation called *path coherence* which is the underlying coherence between the shortest paths and the spatial positions of vertices on a spatial network. We then discussed a coloring algorithm that takes a vertex u_i as input, and then assigns a unique color to all the other vertices in the spatial network based on the first link in the shortest path from u_i. The resulting representation, called a 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 $e_{u_i j}$. We also showed that for spatial networks that are *planar graphs*, the colored regions in the shortest-path map are contiguous. We represent the regions that make up the shortest-path map m_{u_i} using a variant of the region quadtree [142], termed a *shortest-path quadtree*, where all the M_{u_i} different disjoint regions $r_{u_i j}$ are 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. We also show that the shortest-path quadtree is a good dimensionality reducing mechanisms [144], 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. We finally introduced the SILC framework which precomputes and stores the shortest-path quadtree for every vertex in the spatial network.

Additionally, the SILC framework is capable of providing an approximate network distance, in the form of an network distance interval $[\delta^-, \delta^+]$, such that the actual network distance between a pair of vertices is contained by the network distance interval. We introduced the concept of *progressive refinement* of the network distance between objects on a spatial network which allows for *work efficient* algorithms on spatial networks. Our framework allows for the quick computation of an *initial network distance interval* between two objects on the spatial network. We provide a mechanism, termed *refinement*, by which the network distance interval can be made *tighter* by expending more work. This allows for efficient algorithms that only perform as many *refinements* as to be able to answer a query without ambiguity. Finally, we presented experimental results that showed that the SILC framework results in substantial savings in space when compared to a brute force implementation of the path-distance mapping which takes $O(n^2)$ storage, where n is the number of vertices in the spatial network. We also discussed few application scenarios, and introduced several related work from literature.

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 [93] 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.

Chapter 3

Nearest Neighbor Algorithms

Finding the nearest neighbors forms the heart of many search problems in a wide range of fields including pattern recognition, similarity searching, spatial databases, geographic information systems (GIS), computer vision, robotics, and computational geometry. In most of these applications, the problem reduces to one of being given a set of objects S and a particular query object or location q, and determining the nearest object or objects in S whose similarity to q is measured by their distance from q. Nearest neighbor queries are similar to range queries where the range, in terms of number of objects (nearest neighbors), specifies the extent of the region with respect to the query object or location q in which the search is conducted, with the difference that in the nearest neighbor query, the objects in S that are found within the range are ranked according to their distance from q.

There are many ways of measuring this distance. In most applications, it is assumed that the distance between two objects in the set of objects S is the shortest distance between them in the space in which they lie—that is, "as the crow flies." However, with the increasing interest in supporting location-based services, in many applications such a definition of distance is of limited value because both the set of objects S and the space in which their proximity is being measured are constrained to lie on a network such as a set of roads, air corridors, sea channels, and so on. In this case, the nearest objects determined using a spatial distance are not necessarily the closest objects when using a network distance. For example, consider the road network given in Figure 3.1a and 5 locations of a Kinko's store as well as a piano store. Figure 3.1b ranks the Kinko's stores in increasing order of their Euclidean distance from the piano store, while Figure 3.1c ranks the Kinko's stores in increasing order of their network distance from the piano store. We see a difference here as the Kinko's Monroeville store is the closest to the piano store in terms of the Euclidean distance separating them, whereas it is the fourth nearest in terms of the network distance.

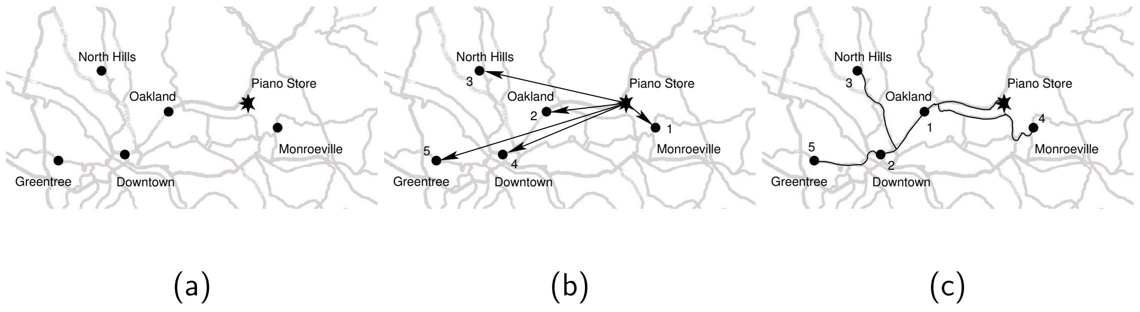


Figure 3.1: (a) Sample road network, and the result of ranking cities from the piano store according to their (b) Euclidean and (c) network distance from the piano store.

Ranking the objects in order of proximity, where proximity is defined in terms of reachability, is important in applications where we do not know in advance how many neighbors we will need as the termination condition stipulates a property that the desired neighbor must satisfy. For example, suppose that we want to find the nearest Kinko's store, in terms of driving distance, which has a capability to make large color posters. Of course, we could respond to the query by finding some number of stores and then selecting the closest one that has the capability to make large color posters, and if none have this capability, then we can find more stores which is usually done by restarting the search from scratch. What we really want to do is is browse the various Kinko's stores by their reachability from the piano store. An algorithm for browsing by distance "as the crow flies" has been devised independently by Henrich [77] and Hjaltason and Samet [80, 83]. In this chapter we show how to adapt this algorithm to an environment where the distance is a network distance.

Such naturally occurring networks can be modeled as spatial networks. In particular, spatial networks are extensions of general graphs $G = (V, E)$, where now the vertices in the graph have fixed spatial positions. As an example, road networks are modeled as directed weighted graphs where the vertices (i.e., road intersections) have spatial positions specified using geographical coordinates—that is, pairs of latitude and longitude values. The weights, while not mandatory, can be assigned to the edges in order to facilitate the modeling of properties of the path that lies between their constituent vertices such as the actual distance between them (necessary when the path between the vertices is not a straight line). Note that other non-spatial attributes, often reflecting constraints, could also be associated with the edges, such as road widths, weight limits, speed limits, trafficability, and so on, for the particular road network.

For all practical cases, the underlying graphs that form the basis of the spatial networks are assumed to be connected planar graphs. This is not an unreasonable assumption as road networks are connected (at least within a landmass such as a continent), although it is not necessary for them to be planar as can be seen by the possibility of the presence of tunnels and bridges. Similarly, it is also possible that the networks are not connected as is the case in islands, private estates, etc. Of course, in the case of water networks consisting of rivers, lakes, canals, and so on, the situation is somewhat different, and simpler due to the smaller network. We do not dwell on such situations here.

In our work, we restrict ourselves to a problem setting where we achieve the status of "nearest" by taking the shortest path through the network from the query object to the desired neighbors. This observation is important as it means that whatever approach we devise should have some advantage over the straightforward one of just applying Dijkstra's algorithm [40], which given a source vertex q (i.e., query vertex) and a connected graph G (i.e., the spatial network), finds the shortest path (and hence 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 such an approach 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 in the worst case for the amount of work to depend on the number of objects that are examined and the number of links on the shortest paths to them from q. The algorithm that we describe satisfies these goals and 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 [154, 186] as described in Section 2.3. However, the algorithm does not use the actual distances and thus there is no need to store them. In particular, Wagner and Willhalm [186] use this strategy for computing shortest paths in conjunction with an R-tree [73] representation of the subsets, while we outline the use of this strategy, calling it SILC, in conjunction with a quadtree-like representation of the subsets, called shortest-path quadtrees, for a a broader range of spatial queries.

In this chapter, we expand on how to adapt the SILC framework to find nearest neighbors in a spatial network in increasing order of network distance by giving a detailed algorithm to do so. As we will see, the advantage of basing the algorithm on the shortest-path quadtrees is that at each stage of the process of finding the shortest path from u to v, the next vertex on the path is unique. In contrast, this is not the case when the algorithm is based on the results in [186] thereby resulting in an approach that is more similar to approaches that make use of a subset of the shortest paths (e.g., [44, 52, 61, 93]). The algorithm that we present is an adaptation of the algorithm of Hjaltason and Samet [80, 83] to an environment that makes use of network distance, and in the process we differentiate it from other attempts to adapt it (e.g., [127]) which do not precompute the shortest paths.

It is important to note that the advantage of our approach 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 particular, the objects that make up the set S from which the neighbors are drawn, as well as the query object(s) need not be the same as the set V of vertices of the spatial network, and the size of S is usually considerably smaller than that of V. This differentiates our approach from other approaches such as those proposed by Papadias, Zhang, Mamoulis, and Tao [127], as well as those of Cho and Chung [34], Kolahdouzan and Shahabi [101], and Shahabi, Kolahdouzan, and Sharifzadeh [166], which must compute the shortest paths anew each time there are changes in q or S. Our approach 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, values of the attributes of the objects may change (e.g., recall the attribute of a Kinko's store that reflects the ability to make color posters or the price per gallon of gas at a gas station). Moreover, it is not unreasonable for the objects to lie on the edges of the network or even off the network, rather than being restricted to coincide with the vertices of the network. Nevertheless, for ease of the exposition of our algorithm, we assume that S is a subset of V although this restriction is easily overcome by associating objects with their "nearest" vertex in the network, assuming without loss of generality that "nearest" in this particular context is measured in terms of Euclidean distance.

The rest of this paper is organized as follows. Section 3.1 reviews related work and places it in the context of both Dijkstra's algorithm and the classical best-first nearest neighbor algorithm that uses distance "as the crow files." In this section we also describe the k nearest neighbor algorithms proposed by Papadias *et al.* [127] and point out that they do not obtain the k nearest neighbors in order of increasing distance nor do they report them in this order, although we also show how to transform them into incremental algorithms. In addition, we point out some errors in the incremental neighbor expansion (INE) algorithm of Papadias *et al.* as well as demonstrate that the improvement to INE proposed by Cho and Chung [34] only reduces its storage requirements while not leading to earlier termination. Section 3.2 describes the incremental variant of our best-first nearest neighbor algorithm, while Section 3.3 describes the k nearest variant of our algorithm. Sections 3.4 and 3.5 describe variants of our algorithm described in Section 3.3. Section 3.6 compares the performance of our algorithm with that of Papadias et al [127]. Section 3.7 contains some concluding remarks and directions for future research.

3.1 Background and Related Work

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. If the number k of neighbors that are sought is known in advance, then 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., [56, 79, 135]). 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., [11, 26, 35, 77, 80, 83]) which explores the non-object elements of the search hierarchy in increasing order of their distance from q (hence the name "best-first"). This is achieved by storing the non-object elements of the search hierarchy in a priority queue in this order. In addition, some of the best-first algorithms (e.g., [77, 80, 83]) 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, or a secondary condition has been satisfied (e.g., recall our earlier example query which sought the nearest Kinko's to the piano store that had the capability to make large color posters). On the other hand, variants can also be constructed that use L to keep track of the k nearest objects [83].

The best-first approach has been proved to be I/O optimal [16] in terms of the number of disk page accesses (i.e., non-leaf block accesses) when the query object and data objects are points and $k = 1$. This is a direct result of the best-first approach's advantage of avoiding having to visit non-object 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 of the non-objects (and thus all the objects) if all of their distances from q are approximately the same. This means that in the worst case the best-first approach requires as much storage as there are elements in S. Nevertheless, studies have shown the best-first approach to be better than the depth-first approach for k fixed [83], and the adaptation of the best-first approach to spatial networks is the subject of this chapter.

The best-first algorithm's use of a priority queue is analogous to Dijkstra's shortest path algorithm which works as follows. It uses a set C containing the vertices for whom the shortest path from q has already been determined and a set U containing those vertices for whom the shortest path from q has not yet been determined. Initially, U contains all the vertices in G (including q), and C is empty. In addition, the network distance of the vertex corresponding to the query object q is initialized to 0 and the network distances of all remaining vertices are initialized to ∞. The algorithm repeats the following sequence of steps until all the vertices of G are in C. Pick the node in U with the smallest cost (i.e., distance from q), say c (initially, this is q). Delete c from U, and add c to C. Next, for each node, say p, such that p is in U and adjacent to c (i.e., an edge exists between c and p that has not been visited before, which we know to be true by virtue of p being in U), update the new distance of p from q to be the minimum of the old distance of p from q and the sum of the distance of c from q and the distance along the edge between c and p. These three steps are repeated until U is empty at which time the algorithm has inserted all of the vertices in C in the order of their distances from q along the shortest paths to them from q.

Given n vertices and m edges, Dijkstra's algorithm has an execution time of $O(m \log n)$ when U is represented as a priority queue implemented using a binary heap, and the spatial network is represented using a vertex representation. It is important to observe that since the driving force behind the algorithm is proximity along the network (the actual length of the path), we cannot easily derive a complexity measure in terms of the steps k that make up the shortest path from q to an object o which we are seeking.

From the above, we see that the inherent structure of Dijkstra's algorithm is such that it proceeds in an incremental manner and returns the vertices of the graph in increasing order of their network distance from q. Dijkstra's algorithm is said to be optimal in the sense that it does not backtrack and revisit a vertex or an edge once it has been visited. An interesting property of the algorithm is that if we are looking for a particular vertex v, then we can halt the algorithm as soon as we encounter v. In the case of nearest neighbor finding, this property means that the ith nearest vertex to q is found after i steps of the algorithm. This is a very attractive property.

The *Incremental Network Expansion (INE)* nearest neighbor algorithm of Papadias, Zhang, Mamoulis, and Tao [127] is a direct adaptation of Dijkstra's algorithm to find the k nearest neighbors of q with some adjustments in order to facilitate implementing a variant that permits the query object q, as well as the objects that make up the set S from which the sought neighbors are drawn, to lie on the edges of the network rather than constraining them to be coincident with the vertices of the network. This variant means that unlike the conventional formulation of Dijkstra's algorithm we may need to visit an edge e more than once (i.e., possibly twice, once for each of its constituent vertices) as the shortest distance from q to the objects that lie on e depends on the path taken to reach them, although e's constituent vertices are still visited just once. In particular, if we do not visit edges containing objects twice, then we may report objects at wrong distances. For example, consider the simple scenario illustrated in Figure 3.2 where when edge $e = (\mathsf{b}, \mathsf{c})$ is visited just once object o is deemed to be at a distance of 25 on a path from q through q's nearest vertex b, while when e is visited twice, o is at a shorter distance of 23 on a path from q to o through q's second nearest neighbor a and c. Thus, from this example, we see why we may need to visit each edge twice. In the following, we describe our implementation of INE which enables us to overcome some of the shortcomings of the version presented by Papadias *et al.* [127] which, as we point out later, will lead to missing some of the neighboring objects, as well as a considerably larger priority queue.

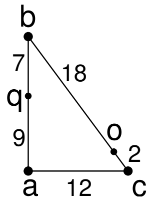


Figure 3.2: Example illustrating how the INE algorithm of Papadias et al [127] fails to find the shortest distance to object o from q by returning a distance of 25 on a path from q to o through vertex b instead of 23 on a path from q to o through vertices a and c.

U is implemented as a priority queue, say *Queue*, while the set of the k candidate nearest neighbors is maintained in L, and D_k denotes the network distance from q to the current kth-nearest object. The main difference from the conventional variant of Dijkstra's algorithm is that as each edge e is processed, the objects that are associated with e (i.e., lie on e) are inserted into L with their corresponding network distances from q. As objects are inserted into L or have their distances from q updated, the value of D_k will either stay the same or decrease and, of course, limiting the size of L to k objects means that objects in L whose distance from q is now greater than D_k must be removed from L. In order to enable q and the objects from which the neighbors are drawn to lie on the edges of the network, INE initializes L with the k nearest objects to q that lie on the edge $e = (v_1, v_2)$ that contains q, and the the distances of v_1 and v_2 in U (i.e., *Queue*) are initialized to their corresponding network distances from q instead of being initialized to ∞ as in Dijkstra's algorithm.

Another difference is that as the vertex c at the front of *Queue* is removed, each object o, if any, associated with an edge $e = (c, v)$ incident at c, regardless of whether v has been previously encountered, is inserted into L with its network distance from q via c, say d_o, provided d_o is less than D_k, the network distance of the kth nearest neighboring object in L, and that o is not already in L with a smaller associated network distance value. Each vertex v of $e = (c, v)$ is inserted into *Queue* with its associated network distance d_v from q unless v has already been visited (i.e., removed from *Queue* and kept track of with the aid of C) or if v is already in *Queue* in which case its associated distance from q is updated if it has decreased. INE halts upon encountering a vertex c such that the network distance between q and the kth nearest neighboring object in L is smaller than the network distance between q and c.

Our variant of the INE algorithm differs from the algorithm in [127] which only processes edges to vertices that have not yet been visited (see line 8). This may cause it to miss some objects when the network contains cycles as can be seen by tracing the execution of the following example illustrated in Figure 3.2. In particular, consider a network where q lies on the edge (a,b) of length 16 so that $d(q,a) = 9$ and $d(q,b) = 7$. In addition, there are edges (a,c) with $d(a,c) = 12$ and (b,c) with $d(b,c) = 20$, and let object o lie on edge (b,c) with $d(o,b) = 18$ and $d(o,c) = 2$. In this case, the variant of INE of Papadias *et al.* will first explore the edges incident at vertex b as b is closer to q than a. In this case, only edge (b,c) is visited as edge (a,b) contains q and has been visited during the initialization phase of the algorithm. This causes vertex c to be inserted on *Queue* with a distance of 27 as $d(q,c) = d(q,b) + d(b,c) = 7 + 20 = 27$. In addition, object o is inserted in L at a distance of $d(q,o) = d(q,b) + d(o,b) = 7 + 18 = 25$.

At this point, *Queue* contains a at a distance of 9 from q and c at a distance of 9 from q, and thus INE will now explore the edges incident at vertex a (i.e., only edge (a,c) as edge (a,b) was visited in the initialization phase). As no objects lie on (a,c), INE enqueues vertex c with a distance of 21 as $d(q,c) = d(q,a) + d(a,c) = 9 + 12 = 21$.

Notice that our variant of INE would not enqueue c again and instead would update its associated distance which is lower. By doing so, we ensure that *Queue* will not get too large as each vertex is present at most once. Of course, the duplicate entries do not impact the correctness of the algorithm as the implementation of *Queue* as a priority queue ensures that the instance with the lowest associated distance will be used.

Next, INE will attempt to explore the edges incident at vertex c as it is on the top of *Queue*. Unfortunately, the INE variant of Papadias *et al.* will not explore any of the edges as all of their vertices have already been explored. In particular, this will mean that edge (b,c) is not explored and we will miss the fact that object o on (b,c) now has a distance of 23 as $d(q,o) = d(q,c) + d(o,c) = 21 + 2 = 23$. In contrast, our variant of INE does not have this drawback as it explores all edges incident at each vertex that is removed from *Queue*. Of course, this does mean that each edge may be visited twice. The possibility of visiting an edge for the second time is reduced by Cho and Chung [34] who keep track of the edges that have been visited along with the maximum possible distance from q to an object on these edges, as described below. At a first glance, one would think that our example depended on the use of a network distance which means that we do not require that the triangle inequality be satisfied by the distance definition. Therefore, it is important to take note of the fact that the example that we presented can arise regardless of whether or not the network is a spatial network. In particular, our example is valid even if we used a definition of distance "as the crow flies" (i.e., a

Euclidean distance), which is the case here. The example given in Figure 3.2 shows that this may cause it to miss some objects when the network contains cycles. The problem with the solution of Papadias *et al.* is that it is only correct when the objects are restricted to coincide with the vertices.

INE's use of L to maintain the k nearest objects to q means that INE does not obtain them in incremental order. Thus, the qualifier "incremental" used to describe the algorithm is somewhat misleading as the algorithm is really a k-nearest neighbor algorithm. Nevertheless, we point out here that INE can be easily modified to be incremental by enqueueing the objects with their corresponding network distances from q as we process the edges adjacent to the vertex v that is removed from the priority queue, thereby also dispensing with the need for L. In addition, *Queue* must be initialized to contain both the vertices of the edge e that contains q and the objects that lie on e before making any attempts to remove elements from *Queue*. Note that the incremental variant of INE may still need to visit an edge e, and hence the objects that lie on e, more than once (i.e., possibly twice, once for each of its constituent vertices) as the shortest distance from q to the objects that lie on e depends on the path taken to reach them, although e's constituent vertices are still visited just once. Visiting the objects more than once is not a problem as prior to attempting to enqueue an object o we check that it has not been output already, and if not, then we update its distance from q if it is present in *Queue* with a higher distance from q, thereby ensuring that the objects do not appear more than once in *Queue*.

The correctness of this solution can be easily verified by observing that when processing vertex v_1 on account of v_1 being on top of the priority queue, the objects that lie on $e = (v_1, v_2)$ are closer to q than v_2 on a path through v_1. However, they are not necessarily closer to q on a path through v_2 that does not pass through v_1, unless, of course, they are encountered at the front of *Queue* before encountering v_2 at the front of *Queue*. It is interesting to observe that the way in which we created the incremental variant of the algorithm is analogous to the manner in which the "best-first" k-nearest neighbor algorithm of Arya *et al.* [11] was made incremental by Hjaltason and Samet [80,83]—that is, the edges of the spatial network play the same role as the non-object elements of the search hierarchy—and thus both algorithms are made incremental by simply eliminating L and ensuring that the priority queue also contains objects instead of just containing non-objects.

One of the drawbacks of INE is that it must explore many vertices when the distribution of the objects around the vertices of the spatial network is relatively sparse, which is often the case. For example, when the set S of objects consists of locations of gas stations, it is clear that S is much smaller than the set of vertices that make up the road network. Cho and Chung [34] try to improve on INE when the distribution of objects around q is sparse by preprocessing the spatial network to find vertices with a large outdegree (e.g., ≥ 3), termed *intersection vertices*, and computing the shortest paths from some small subset of them (elements of which are termed *condensing points*) to a predefined number m of their nearest neighboring objects. When the nearest neighbor algorithm encounters a condensing point c in the course of the search, the list L of k candidate nearest neighbors is updated with the m neighbors stored at c. Although the network distance from q to a neighboring object o of c, denoted by $d(q,o)$, is unknown, Cho and Chung make use of the triangle inequality to approximate it with the network distance $d(q,c)+d(c,o)$. The result is that their algorithm encounters neighboring objects of q earlier than the INE algorithm, thereby enabling other vertices and objects to be pruned, and consequently reducing the size of *Queue*. In other words, the effect is analogous to that achieved via the use of an estimate of the maximum distance at which the k^{th} nearest neighbor can be found in best-first k nearest neighbor finding (known as MAXNEARESTDIST [143–145] and also [135] where it is called MINMAXDIST). However, unfortunately, although the neighboring objects are encountered earlier, Cho and Chung do not provide a mechanism for the algorithm to terminate earlier as the network distances that are associated with the objects in L that are neighboring objects of the intersecting vertices that are condensing points are approximate network distances, and thus the algorithm only terminates when the search actually encounters these objects. Also, Cho and Chung's proposed improvement still does not make INE incremental.

It is worth pointing out that unlike the INE algorithm of Papadias et al. [127], Cho and Chung's algorithm [34] does not miss any objects by virtue of implementing INE with an extra data structure that keeps track of the edges that have been visited already and for each such edge e they record the maximum possible distance from q to an object on e. In particular, when processing vertex v_2 and encountering edge $e = (v_1, v_2)$ for a second time, they do not visit e if the recorded maximum possible distance from q to an object on e, from previous visits (i.e., via v_1) is less than the current minimum distance to an object on e from q via v_2. Note that when e is visited again, Cho and Chung [34] update the maximum possible distance from q to an object on e although this is not necessary as the fact that e was encountered while processing v_1 before being encountered while processing v_2 means that v_1 is closer to q than v_2 and thus the maximum possible distance to an object on e via v_1 will always be less than the maximum possible distance to an object on e via any other subsequently encountered vertex including v_2. Observe however, that like Papadias *et al.* [127], Cho and Chung also insert the same vertex on *Queue* a multiple number of times thereby causing *Queue* to become larger than necessary.

The main drawback of methods such as INE, as well as Cho and Chung's proposed improvement, which incorporate Dijkstra's algorithm, regardless of whether or not they are incremental, is that their steps involve vertices of the spatial network, while the neighbors in which we are interested are drawn from the set of objects S which is usually considerably smaller than the number of vertices in the network. Thus, Dijkstra's algorithm may visit many vertices before reaching one which coincides or is near one of the objects in which we are interested. Moreover, it is not uncommon for the 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 1.1 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, Maryland. Here we see that 75.4% of the vertices are visited in the network (i.e., 3,191 out of a total of 4,233 vertices) in the process of obtaining a shortest path of length 75 edges.

The k nearest neighbors can also be obtained by making use of the best-first method and noting that the Euclidean distance between two objects q and u in the spatial network serves as a lower bound on the distance between them through the network (i.e., the network distance). This observation/restriction forms the basis of the *Incremental Euclidean Restriction* (IER) nearest neighbor algorithm of Papadias, Zhang, Mamoulis, and Tao [127]. The IER algorithm uses a priority queue *Queue*, as well as an R-tree [73] to implement the search hierarchy T used to represent the set S of objects from which the neighbors are drawn. IER starts by applying the incremental nearest neighbor algorithm to obtain a set L of k candidate nearest neighbors that are the k nearest neighbors of q in terms of their Euclidean distance from q. Once the k nearest Euclidean distance neighbors have been determined, IER computes the network distance from q to all of the elements of L (using a shortest-path algorithm such as Dijkstra's algorithm), sorts them in increasing order of their network distance from q, and notes the farthest one o_k at network distance D_k from q. At this point, the incremental nearest neighbor algorithm is reinvoked to continue processing the remaining objects in order of increasing Euclidean distance from q until encountering an object o whose Euclidean distance from q is greater than D_k, at which time the algorithm terminates. Each time an object o is encountered whose Euclidean distance from q is less than D_k, o's network distance d_o from q is computed, and if it is closer to q than one of the elements i of L at network distance d_i from q (i.e., $d_i < D_k$), then o is inserted into L, thereby removing one of the objects at network distance D_k from q, and D_k is reset as is appropriate.

As in the case of INE, IER's use of L to maintain the k nearest objects to q means that IER does not obtain them in incremental order. Instead, it is only incremental in the sense that it operates in a "filter-and-refine" [125] mode where it obtains its candidate objects for the filter test in increasing order of their Euclidean distance from q. However, the actual objects that it finds are not reported in increasing order of network distance. Thus, as we characterized it initially, it is really a k-nearest neighbor algorithm that performs filtering of candidates in increasing order of their Euclidean distance from the query object. This means that once IER has found the k nearest neighbors of q in terms of their network distance, it cannot be resumed to find the $(k+1)$th nearest neighbor. Nevertheless, we point out here that IER can be easily modified to be incremental and during the process we can dispense with the need for L. The key idea is that each time an object o is encountered while processing a block of the search hierarchy T (i.e., the R-tree), we compute o's network distance d_o from q and insert o in the priority queue *Queue* with network distance d_o. Now, whenever an object is encountered at the front of *Queue*, we know that it is the nearest object and it can be output thereby simplifying the algorithm considerably while also making it incremental.

The main drawback of IER is its constant need to evaluate the network distance along the shortest path from q to each object that it encounters. This can be quite expensive for large graphs and entails much repeated work as the shortest paths often have common subpaths. Most importantly, due to IER's need to compute the shortest paths to the k nearest objects, with the farthest object being at network distance D_k from q, if IER makes use of Dijkstra's algorithm to obtain the shortest paths, then IER must visit all vertices whose network distance from q is less than D_k. Thus, in such a case, the execution time of IER is at worst as good as INE, and, most likely, much worse due to the need to repeatedly compute the shortest paths to the objects that are found at incrementally farther distances from q.

The problem with the methods that are based on Dijkstra's algorithm have led to an interest in precomputing the shortest paths thereby avoiding the constant reinvokation of Dijkstra's algorithm or avoiding the repeated visiting of all of the vertices in the network for each new query. In particular, our goal is to use the spatial position of the vertices and the objects to avoid visiting vertices that are not on the shortest paths to the nearest objects to the query object. We achieve this by precomputing and storing the shortest-path quadtrees of all the vertices of the network. The precomputation approach to finding shortest paths has been used by a number of researchers (e.g., [44,52,61,93]). The difference is that in our work we are using the results of the deployment of such algorithms to guide the process of finding neighbors in increasing order of distance in a spatial network. In particular, in some sense, we are using the shortest paths to constrain Dijkstra's algorithm in finding the shortest neighbors, and hence our work is complimentary to these solutions rather than a competitor.

We now briefly review the methods that precompute the paths. Filho and Samet [44] do so by imposing an auxiliary hierarchical structure on the spatial network which is an improvement on the method of Jing, Huang, and Rundensteiner [93]. Goldberg and Harrelson [61] propose an alternative method that embeds a set of pivots P at known positions (termed *landmarks* and very similar to the condensing points of Cho and Chung [34]) in the network and then make use of the known distances on the shortest paths between them. Kolahdouzan and Shahabi [100, 101] treat the objects in S as sites of a Voronoi diagram and precompute shortest paths between them and vertices (the landmarks) that the path reaches by crossing the boundary of the corresponding Voronoi region. As we have seen in Section 2.4, the method of Wagner and Willhalm [186] can also be viewed as making use of landmarks.

3.2 Incremental Nearest Neighbor Algorithm

Like the original incremental nearest neighbor algorithm of Hjaltason and Samet [80,83] described in Section 3.1, our incremental best-first nearest neighbor algorithm 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. Having defined the shortest-path quadtree in Section 2.4, we are now ready to describe the mechanics of procedure INCNEARESTSPATIALNETWORK shown in Algorithm 7. Figure 2.9 illustrates an application of our algorithm to a road network dataset. We use a best-first incremental algorithm that is similar to Hjaltason and Samet [80, 83], although as the network distances between objects in the SILC framework are network distance intervals, minor modifications are needed.

The algorithm takes three inputs— a pointer T to the root of a hierarchical spatial data structure containing the set S of objects (e.g., a set of hospitals) from which neighbors are drawn, a query object q (e.g., scene of an accident) and the shortest-path quadtree s_q of q. We now describe the organization of an hierarchical data structure H on S whose root node is T. Our algorithm is similar to the best-first algorithm of Hjaltason and Samet [80] in the sense that it imposes very little restriction on the nature of H. In particular, the data structure H can either belong to an object based family of data structures, such an an R-tree [73], or a space decomposition method, such as the quadtrees [142, 144]. Note that H defines a hierarchy consisting of two kinds of blocks, *non-leaf* blocks and *leaf* blocks. A non-leaf block contains children nodes that are in turn non-leaf or leaf blocks. A leaf block contains one or more objects.

The algorithm uses a priority queue *Queue* of *objects* and *blocks*, collectively referred to as *elements*. *Queue* also stores the network distance interval of the elements from q. Some additional information is stored when an element s in *Queue* is an object — such as, an intermediate vertex u in the shortest path from q to s, and the network distance $d_G(s,u)$ from s to u. *Queue* retrieves the stored elements in an increasing ordering of δ^-, the minimum network distance interval of elements from q.

Lines 1–4 initialize the priority queue *Queue* by inserting the root T along with its network distance interval $[\delta^-,\delta^+]$ from q. This is achieved using the MINNETWORK-DISTBLOCK Algorithm described in Section 2.5.2. At each iteration of the algorithm, the *top* element p in the queue is examined. If p is a *leaf* block, then it is replaced with all the objects contained within it after computing their initial network distance intervals from q using the GETNETWORKDISTINTERVAL algorithm. If p is a *non-leaf* block, then all of its children blocks are inserted into *Queue* after computing their network distance intervals to q. If p is an *object*, then the distance interval of p is checked against the current top element in *Queue* for possible *collisions*. A *collision* takes place when the distance interval of p intersects with the distance interval of the current top element in *Queue* in which case, the distance interval of p is *refined* by applying the REFINENETWORKDISTINTERVAL operator (described in Algorithm 5), after which p

is reinserted back into *Queue*. If the network distance interval of p is non-intersecting with the top element of *Queue*, p is reported (line 13) as the next nearest neighbor to q and the function returns to the control back to the caller. More neighbors of q can be retrieved by making subsequent invocations of the routine —subsequent invocations of the algorithm start execution at line 5 — resulting in an *incremental* retrieval of the nearest neighbors of q.

Algorithm 7

Procedure INCNEARESTSPATIALNETWORK[T, q, s_q]

Input: $T \leftarrow$ root node of hierarchical data structure H on S

Input: q is the query object

Input: s_q is the shortest-path quadtree of q

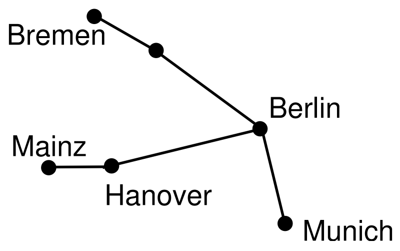
Output: p is the next nearest neighbor to $q \in S$

```
1.  INIT: [$\delta^-,\delta^+$] ← MINNETWORKDISTBLOCK($q$, $T$, $s_q$)
2.        Queue ← an empty priority queue of elements
3.        ENQUEUE(KEY=$\delta^-$, VALUE=$(T,[\delta^-,\delta^+],q,0)$, Queue)
4.  END-INIT
5.  while not ISEMPTY(Queue) do
6.    $(p,[\delta^-,\delta^+],u,d)$ ← VALUE(DEQUEUE(Queue)) (∗ Extract top element ∗)
7.    if ISOBJECT($p$) then
```

```
8.          $(\_, [\mu^-, \mu^+], \_, \_) \leftarrow$ VALUE(FRONTPRIORITYQUEUE(*Queue*))
9.          **if** INTERSECTS($[\mu^-, \mu^+], [\delta^-, \delta^+]$) **then**
10.            $(u, d, [\delta^-, \delta^+]) \leftarrow$ REFINENETWORKDISTINTERVAL($q, u, p, d, [\delta^-, \delta^+]$)
11.            ENQUEUE(KEY=$\delta^-$, VALUE=$(p, [\delta^-, \delta^+], u, d)$, *Queue*)
12.         **else**
13.            report $p$ (and **return**)
14.         **end-if**
15.      **else if** ISNONLEAFBLOCK($p$)   (* $p$ is a block *)
16.         **for** each children block $r$ in $p$ **do**
17.            $[\delta^-, \delta^+] \leftarrow$ MINNETWORKDISTBLOCK($q$, $r$, $s_q$)
18.            ENQUEUE(KEY=$\delta^-$, VALUE=$(r, [\delta^-, \delta^+], q, 0)$, *Queue*)
19.         **end-for**
20.      **else** (* $p$ is a leaf block *)
21.         **for** each children objects $o$ in $p$ **do**
22.            $[\delta^-, \delta^+] \leftarrow$ MINNETWORKDISTINTERVAL($q$, $o$, $s_q$)
23.            ENQUEUE(KEY=$\delta^-$, VALUE=$(o, [\delta^-, \delta^+], q, 0)$, *Queue*)
24.         **end-for**
25.      **end-if**
26.   **end-while**
```

Theorem 3.1. *An single invocation of the procedure* INCNEARESTSPATIALNETWORK *retrieves the next nearest neighbor of q, even though it is possible that its distance interval from q has not been fully refined.*

Proof. See Theorem 3.2. □



	Munich	Bremen
Mainz	[10,20]	[12,24]
Hanover	[13,18]	[17,20]
Berlin	[14,16]	

Figure 3.3: An example illustrating the working of an incremental best-first algorithm consisting of cities corresponding to objects.

We now illustrate the working of an incremental best-first algorithm using the example shown in Figure 3.3. Suppose that the objects and the vertices are drawn from the same domain—that is, cities in a road map of Germany. The question we are trying to answer is that which city among "Munich" or "Bremen" is closer to "Mainz". Suppose that the top of the priority queue contains objects corresponding to Munich and Bremen with distance intervals of [10,20] and [12,24] with respect to Mainz, respectively. Removing the top element of the priority queue finds us processing Munich as its minimum distance of 10 from Mainz is the smallest. However, Munich's maximum distance of 20 from Mainz is not smaller than the minimum distance from Mainz of Bremen, the new element at the top of the priority queue, which is 12. Thus, we refine the shortest path from Mainz to Munich by use of the shortest-path quadtree of Mainz to find that the first edge on the shortest path from Mainz to Munich passes through Hanover, and now the distance interval of Munich with respect to Mainz on a path through Hanover is [13,18], which causes Munich to be reinserted into the priority queue with distance interval [13,18].

At this point, the algorithm is resumed with Bremen at the top of the priority queue and it is the one that is removed. However, Bremen's maximum distance of 24 is not smaller than the minimum distance from Mainz of Munich, the new element at the top of the priority queue, which is 13. Thus, we refine the shortest path from Mainz to Bremen by use of the shortest-path quadtree of Mainz to find that the first edge on the shortest path from Mainz to Bremen also passes through Hanover and now the distance interval of Bremen with respect to Mainz on a path through Hanover is [17,20], which causes Bremen to be reinserted into the priority queue with distance interval [17,20].

We now resume the algorithm with Munich at the top of the priority queue and it is the one that is removed. However, Munich's maximum distance of 18 from Mainz is not smaller than the minimum distance from Mainz of Bremen, the new element at the top of the priority queue, which is 17. Thus, we refine the shortest path from Mainz to Munich by use of the shortest-path quadtree of Hanover (which is the most recently detected city on the shortest path from Mainz to Munich) to find that the first edge on the shortest path from Hanover to Munich passes through Berlin, and now the distance interval of Munich with respect to Mainz on a path through Hanover and Berlin is [14,16], which causes Munich to be reinserted into the priority queue with distance interval [14,16]. This causes the algorithm to be resumed with Munich at the top of the priority queue and it is the one that is removed. However, unlike previous occurrences, this time Munich's maximum distance of 16 from Mainz on a path through Hanover and Berlin is smaller than the minimum distance from Mainz of Bremen, the new element at the top of the priority queue, on a path through Hanover, which is 17. Thus, Munich is returned as the nearest neighbor of Mainz.

3.3 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 KNEAREST-SPATIALNETWORK which is probably the only algorithm of its kind to work 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.

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 in Section 3.2 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 [83]. The 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, we see that 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 [144] 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 non-leaf 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 of p continues as by Theorem 3.1 (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 30), 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 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 57).

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 3.2 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 non-incremental algorithm that returns in priority queue L the k near-
     est 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 non-leaf
     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. */
 **value object** $q$
 **value integer** $k$
 **value object_set** $S$
 **value pointer search_hierarchy** $T$
 **integer** $D_k$
 **priority_queue** $L$, $Queue$
 **object** $o$
 **vertex** $v$
 **interval** $i$
 **real** $s$
 **pointer search_hierarchy** $e, e_p$
 **priority_queue_entry** $t$
 $L \leftarrow$ NEWPRIORITYQUEUE()
 /* $L$ is the priority queue containing the $k$ nearest objects */
 $Queue \leftarrow$ NEWPRIORITYQUEUE()
 $e \leftarrow$root of the search hierarchy induced by $S$ and $T$
 ENQUEUE([E =]$e$, [D =]0, $Queue$)
 $D_k \leftarrow \infty$
 **while not** ISEMPTY($Queue$) **do**
   $t \leftarrow$ DEQUEUE($Queue$)
   $e \leftarrow$ E($t$)
     **if** ISOBJECT($e$) **then** /* $e$ is an object */
     **if** MINNETWORKDISTINTERVAL(D($t$)) $\geq$ $D_k$ **then**
        **return** $L$
     **elseif** MAXNETWORKDISTINTERVAL(D($t$))
           $\geq$ MINNETWORKDISTINTERVAL(
                D(FRONTPRIORITYQUEUE($Queue$))) **then**
       **if** MAXNETWORKDISTINTERVAL(D($t$)) $\leq$ $D_k$ **then**
         /* Ensure one entry/object in $L$ */
          REMOVEPRIORITYQUEUE($e$, $L$)
       **endif**
       $v \leftarrow$ NEXTVERTEXSHORTESTPATH(
              $e$, SHORTESTPATHQUADTREE(V($t$)))
       /* NEXTVERTEXSHORTESTPATH performs point location on $e$ in the
          SHORTESTPATHQUADTREE of V($t$) and returns the vertex $v$ associated
```

```
            with the block or region containing $\mathrm{V}(t)$. */
        $s \leftarrow \mathrm{I}(t) + $ EdgeWeight$(\mathrm{V}(t), v)$
        /* EdgeWeight$(\mathrm{V}(t), v)$ is the distance between $\mathrm{V}(t)$ and $v$ */
        $i \leftarrow s + $ GetNetworkDistInterval(
                $e$, ShortestPathQuadtree$(v)$)
        **if** MaxNetworkDistInterval$(i) \leq D_k$ **then**
        /* Update $L$ and $D_k$ as necessary */
          InsertL$(e,$ MaxNetworkDistInterval$(i), k, L, D_k)$
        **endif**
        **if** MinNetworkDistInterval$(i) < D_k$ **then**
          Enqueue$([\mathrm{E} =]e, [\mathrm{D} =]i, [\mathrm{V} =]v, [\mathrm{I} =]s, Queue)$
        **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 distance 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$([\mathrm{E} =]o, [\mathrm{D} =]i, [\mathrm{V} =]q, [\mathrm{I} =]0, Queue)$
          **if** MaxNetworkDistInterval$(i) < D_k$ **then**
            InsertL$(o,$ MaxNetworkDistInterval$(i), k, L, D_k)$
          **endif**
        **endif**
      **enddo**
  **else** /* $e$ is a non-leaf block */
      **foreach** child element $e_p$ of $e$ **do**
        **if** MinNetworkDistBlock$(q, e_p) < D_k$ **then**
          Enqueue$([\mathrm{E} =]e_p,$
                  $[\mathrm{D} =]$MinNetworkDistBlock$(q, e_p),$
                  $Queue)$
        **endif**
      **enddo**
  **endif**
```

```
72 enddo
```

We now prove a pair of theorems that are needed in the demonstration of the correctness of procedure KNEARESTSPATIALNETWORK.

Theorem 3.1. *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.*

Proof. We first show that 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 (i.e., $D_k \geq \delta_t^+$). Formally, given that $\delta_t^- \leq \delta_t^+ < \delta_p^- \leq \delta_p^+$, we first show that $D_k \geq \delta_t^+$.

Let r be an object in the priority queue L such that $D_k = \delta_r^+$. We first show that r is also present in *Queue*. We do this by contradiction by assuming that r is not in *Queue*. The only two cases when r could not be in *Queue* are:

- r has already been encountered which means that $\delta_t^- > D_k = \delta_r^+$, in which case the algorithm would have terminated when t was removed from *Queue* (line 25).

- r has been pruned by virtue of δ_r^- being greater than D_k which is impossible as $\delta_r^- > D_k = \delta_r^+$ as we assumed that the maximum of any distance interval is always greater than or equal to the minimum of the distance interval.

Hence, r is present in *Queue*. Recalling that t is the object that was just removed from *Queue* and p is the top object on *Queue*, we find that the only possible relative positions of δ_t^-, δ_p^-, and δ_r^- in *Queue* are:

- If $t = r$, then $\delta_r^+ = \delta_t^+ = D_k$.

- If $p = r$, then $\delta_t^- \leq \delta_t^+ < \delta_p^- = \delta_r^- \leq \delta_r^+ = \delta_p^+ = D_k$, which reduces to $D_k \geq \delta_t^+$.

- If r is present in *Queue* after p (i.e., at a lower priority), then from the hypothesis of the Theorem we have that $\delta_t^+ < \delta_p^-$. From the fact that r appears in *Queue* after p we have $\delta_p^- \leq \delta_r^-$. Combining these two inequalities with the fact that $\delta_r^- \leq \delta_r^+$, we have that $\delta_t^+ < \delta_p^- \leq \delta_r^- \leq \delta_r^+$, which reduces to $D_k \geq \delta_t^+$.

Hence, we have shown that in all relative positions of t, p, and r (i.e., $t = r$, or $p = r$, or r appears after p in *Queue*), we have that $D_k \geq \delta_t^+$. Therefore, t is a candidate object to serve as one of the k nearest neighbors of q.

Now, to show that t is indeed one of the k nearest neighbors of q, we point out that the distance interval of any object w in *Queue* is strictly greater than the distance interval of t (i.e., $\delta_t^- \leq \delta_t^+ < \delta_w^- \leq \delta_w^+$). Hence, *Queue* contains no objects that are closer to q than t. $\square$

Theorem 3.2. *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.*

Proof. We can show that the output of the KNEARESTSPATIALNETWORK is a total ordering of the k nearest neighbors of q by observing that if an object t is removed from *Queue* so that the maximum of the distance interval of t is less than the minimum of the distance interval of an element p which is currently at the top of *Queue*, then t is neither refined any further nor inserted into the priority queues *Queue* and L. In other words, t is not reinserted into *Queue* if $\delta_t^+ < \delta_p^-$. For any object t and its associated distance interval $[\delta_t^-, \delta_t^+]$, we know that $\delta_t^- \leq \delta_t^+$. Combining both of these inequalities yields $\delta_t^- \leq \delta_t^+ < \delta_p^- \leq \delta_p^+$, that is, the distance interval of t is strictly less than the distance interval of p. As the distance interval of t is strictly less than the distance interval of p, we can also show that the distance interval of t is also strictly less than the distance interval of any other element w present in *Queue*. In particular, suppose that w is present in *Queue* after p (i.e., at a lower priority). From our assumption that the distance interval of t is strictly less than the distance interval of p, we are given that $\delta_t^+ < \delta_p^-$. From the fact that w appears in *Queue* after p we have $\delta_p^- \leq \delta_w^-$ (recall that *Queue* is ordered by the minimum of the distance intervals of its constituent elements). Combining these two inequalities with the fact that $\delta_w^- \leq \delta_w^+$, we have that $\delta_t^+ < \delta_p^- \leq \delta_w^- \leq \delta_w^+$ implying that the distance interval of t is strictly less than the distance interval of any element w in

Queue. Hence, the distance interval of all subsequent nearest neighbors of q that are obtained from the elements in *Queue* are strictly greater than the distance interval of p. Hence, two successive neighbors o_i, o_{i+1} of q have the property that the network distance interval of o_i is strictly less than the network distance interval of o_{i+1}. Therefore, at the end of the algorithm, the total ordering of the k nearest neighbors of q is obtained by sorting the objects in L in an increasing order of either their minimum or maximum distance interval from q. Note that since L already contains the objects in decreasing order (recall that the object at the top of priority queue L whose distance interval's maximum is the largest) of the maximum of their distance interval thereby obviating the need to sort them. □

3.4 Best-First K Nearest Neighbor Algorithm With No Object Distance Updates

We now introduce a variant of the KNEARESTSPATIALNETWORK algorithm that uses the D_k estimator in order to obtain the k nearest neighbors of a query object on a spatial network. Algorithm 8 provides the pseudo code of the KNEARESTSPATIALNETWORK-INCR algorithm. Suppose, we are given a set S of objects and a query object q on a spatial network, such that the k nearest neighbors of q are drawn from S. The D_k estimator provides the upper bound on the maximum possible network distance to the kth nearest neighbor of q in S. The KNEARESTSPATIALNETWORKINCR algorithm is similar to KNEARESTSPATIALNETWORK algorithm in the sense that both of them use a priority queue L of objects in order to compute the value of D_k. However, the key difference between the two algorithms is that, in the case of KNEARESTSPATIALNETWORK algorithm, the value of D_k is updated as soon as the network distance interval of an object o contained in L is *refined*, while it is not updated in the case of the KNEARESTSPATIALNETWORKINCR algorithm. The trade-off here is between expending work to keep the objects in L up to date versus choosing to save work by not updating L. Consequently D_k in the latter case is not as small as it can possibly be resulting in fewer elements in *Queue* being pruned against D_k.

We now describe an efficient implementation of the KNEARESTSPATIALNETWORKINCR algorithm by making use of two priority queues similar to the KNEARESTSPATIALNETWORK algorithm. *Queue* retrieves objects in an increasing order of the minimum of their network distance intervals, while L, which has a capacity of k objects, organizes objects in a decreasing order of the maximum of their network distance intervals. When L contains k objects, the maximum of the distance interval of the top element of L corresponds to D_k. As the algorithm described in this section is similar to KNEARESTSPATIALNETWORK, we only describe the key implementational differences between the algorithms. The *Queue* is populated similar to KNEARESTSPATIALNETWORK. Let us now consider the case when a leaf block p is retrieved from the top of *Queue*. The children objects of p are inserted into *Queue* and L after their initial network distance interval distance to q have been estimated using s_q.

We now consider the case that p is an object. If the network distance interval is non-intersecting with the network distance interval of the current top element of *Queue*, we report p as the next nearest neighbor of q. If intersecting, we first refine the network distance interval of p, and insert it back into *Queue*. The algorithm terminates when k nearest neighbors to q have been reported by the algorithm.

The main difference between the algorithms is when p is an object and is refined. In the case of KNEARESTSPATIALNETWORK algorithm, we determine if p is also present in L, in which case, we update the distance interval of the p in L, potentially reducing D_k in the process. However, in case of the KNEARESTSPATIALNETWORKINCR algorithm, we will not update the network distance interval of p in L resulting in some savings in work at the expense of a larger value of D_k resulting in reduced pruning.

Algorithm 8

```
Procedure KNEARESTSPATIALNETWORKINCR[$T$, $q$, $s_q$, $k$]
Input: $T \leftarrow$ root node of a hierarchical data structure on $S$
Input: $q$ is the query object
Input: $s_q$ is the shortest-path quadtree of $q$
Output: $k$ nearest neighbors of $q$
1.  integer $f$
```

```
2.  **value** $D_k$
3.  **priority_queue** $Queue$, $L$
4.  $[\delta^-, \delta^+] \leftarrow$ MINNETWORKDISTBLOCK($q$, $T$, $M_q$)
5.  $Queue \leftarrow$ NEWPRIORITYQUEUE()
6.  $L \leftarrow$ NEWPRIORITYQUEUE()
7.  $D_k \leftarrow \infty$
8.  $f \leftarrow 0$
9.  ENQUEUE(KEY=$\delta^-$, VALUE=$(T, [\delta^-, \delta^+], q, 0)$, $Queue$)
10. **while** (**not** ISEMPTY ($Queue$) **and** $f \leq k$) **do**
11.     $(p, [\delta^-, \delta^+], u, d) \leftarrow$ VALUE(DEQUEUE($Queue$)) (* Extract top element *)
12.     **if** $\delta^- > D_k$ **then**
13.         **break**
14.     **end-if**
15.     **if** ISBLOCK($p$) **then**
16.         **for** each child block $e$ in $p$ **do**
17.             $[\delta^-, \delta^+] \leftarrow$ MINNETWORKDISTBLOCK($q$, $e$, $s_q$)
18.             **if** $\delta^- < D_k$ **then**
19.                 ENQUEUE(KEY=$\delta^-$, VALUE=$(e, [\delta^-, \delta^+], q, 0)$, $Queue$)
20.             **end-if**
21.         **end-for**
```

```
22.     else if IsLeafBlock($p$) then
23.         for each child object $e$ in $p$ do
24.             $[\delta^-, \delta^+] \leftarrow$ MinNetworkDistInterval($q$, $e$, $s_q$)
25.             if $\delta^- < D_k$ then
26.                 Enqueue(Key=$\delta^-$, Value=$(r, [\delta^-, \delta^+], q, 0)$, Queue)
27.                 UpdateD$_k$($L, p, \delta^+, D_k, k$)
28.             end-if
29.         end-for
30.     else (* $p$ is an object *)
31.         $(\_, [\mu^-, \mu^+], \_, \_) \leftarrow$ Value(FrontPriorityQueue(Queue))
32.         if Intersects($[\mu^-, \mu^+], [\delta^-, \delta^+]$) then
33.             $(u, d, [\delta^-, \delta^+]) \leftarrow$ RefineNetworkDistInterval($q, u, p, d, [\delta^-, \delta^+]$)
34.             Enqueue(Key=$\delta^-$, Value=$(p, [\delta^-, \delta^+], u, d)$, Queue)
35.         else
36.             report $p$
37.             $f \leftarrow f + 1$
38.         end-if
39.     end-if
40. end-while
```

3.5 Best-First K Nearest Neighbor Algorithm with kMinDist Estimator

We now introduce a variant of the KNearestSpatialNetwork algorithm that uses the kMinDist estimator, defined below, in addition to the D_k estimator to obtain the k nearest neighbors of a query object q on a spatial network. Algorithm 9 provides the pseudo-code of the KMinDistNearestNeighbor algorithm. Suppose, we are given a set S of objects and a query object q on a spatial network, such that the k nearest neighbors of q are drawn from S. The kMinDist estimator provides a lower bound on the network distance of the kth nearest neighbor of q, while D_k provides an upper bound on the network distance of the kth nearest neighbor of q. In other words, we have restricted the candidate objects that can potentially be the kth nearest neighbor of q to those in S that are farther than kMinDist, but closer than D_k. This leads us to the observation that any object o whose network distance interval from q is less than kMinDist is guaranteed to be a nearest neighbor of q. In other words, suppose $[\delta^-,\delta^+]$ is the network distance interval of an object t from q. t is a viable candidate if and only if kMinDist $< \delta^+ < D^k$. Note that, we can trivially obtain the kMinDist estimate by first sorting the objects of S in an increasing order of the minimum of their network distance intervals from q to obtain an ordered set S'. Then, the value of the kMinDist estimator is the maximum of the network distance interval associated with the kth object in S'.

We now describe the workings of the KMinDistNearestNeighbor algorithm which uses three priority queues instead of the two priority queues used in KNearestSpatialNetwork algorithm. To be precise, we use another priority queue $Queue_1$ in addition to *Queue* and L. Both *Queue* and $Queue_1$ retrieve objects in an increasing order of the minimum of their network distance intervals, while L arranges objects by the maximum of their network distance intervals. When L contains k objects, the maximum of the distance interval of the top element of L (kth largest) corresponds to D_k.

As the algorithm described in this section is similar to KNearestSpatialNetwork, we only describe the key implementational differences between them. The *Queue* is populated similar to KNearestSpatialNetwork. Let us now consider the case when an object is retrieved from the top of *Queue*, in which case, it is inserted into $Queue_1$ and L. Let p be the kth object that is retrieved from the top of *Queue*. The value of the kMinDist estimator is assigned to be the minimum of the network distance interval of p from q. Note that after inserting p into L, we are guaranteed that L has k objects and that D_k is set. We now continue dequeuing objects from *Queue* and enqueuing them into $Queue_1$ until the minimum of the network distance interval of an object retrieved from *Queue* is greater than D_k, in which case we are guaranteed that the remaining elements in *Queue* cannot contribute a nearest neighbor to q.

The rest of the algorithm operates only on $Queue_1$. Algorithm 9 proceeds similar to IncNearestSpatialNetwork. At each step of the algorithm, the top element p

of $Queue_1$ is retrieved. If the maximum of the network distance interval of p from q is less than KMINDIST, p is directly added to the result set and is one of the the nearest neighbor of q. An object is said to be *pruned against* the KMINDIST estimator if it is added to the result set by virtue of the maximum of its network distance interval from q being less than KMINDIST. One of the drawbacks of this algorithm is that the objects in the result set are not ordered. That is, compared to the KNEARESTSPATIALNETWORK algorithm which establishes a total ordering of the k nearest neighbors, this algorithm does not produce an ordered output. If the maximum of the network distance interval of p is greater than KMINDIST, we check if the network distance interval of p is non-intersecting with the network distance interval of the top element of $Queue_1$. If so, we report p as the next nearest neighbor of q, else we first refine the network distance interval of p, and insert it back into $Queue_1$. The algorithm terminates when k nearest neighbors have been reported by the algorithm.

Algorithm 9

Procedure KMINDISTNEARESTNEIGHBOR[T, q, s_q, k]

Input: $T \leftarrow$ root node of a hierarchical data structure on S

Input: q is the query object

Input: s_q is the shortest-path quadtree of q

Output: $k \leftarrow$ number of nearest neighbors of q

```
1.  **integer** $r, f$
2.  **value** KMinDist
3.  **value** $D_k$
4.  **priority_queue** $Queue$, $Queue_1$, $L$
5.  $[\delta^-, \delta^+] \leftarrow$ MinNetworkDistBlock($q$, $T$, $M_q$)
6.  $Queue \leftarrow$ NewPriorityQueue()
7.  $Queue_1 \leftarrow$ NewPriorityQueue()
8.  $L \leftarrow$ NewPriorityQueue()
9.  $D_k \leftarrow \infty$
10. KMinDist $\leftarrow 0.0$
11. $r \leftarrow 0$
12. $f \leftarrow 0$
13. Enqueue(Key=$\delta^-$, Value=$(T, [\delta^-, \delta^+], q, 0)$, $Queue$)
14. **while not** IsEmpty ($Queue$) **do**
15.     $(p, [\delta^-, \delta^+], u, d) \leftarrow$ Value(Dequeue($Queue$)) (* Extract top element *)
16.     **if** $\delta^- > D_k$ **then**
17.         **break**
18.     **end-if**
19.     **if** IsBlock($p$) **then**
20.         **for** each child block $e$ in $p$ **do**
```

```
21.             $[\delta^-, \delta^+] \leftarrow$ MinNetworkDistBlock($q$, $e$, $s_q$)
22.             **if** $\delta^- < D_k$ **then**
23.                 Enqueue(Key=$\delta^-$, Value=$(e, [\delta^-, \delta^+], q, 0)$, *Queue*)
24.             **end-if**
25.         **end-for**
26.     **else if** IsLeafBlock($p$) **then**
27.         **for** each child object $e$ in $p$ **do**
28.             $[\delta^-, \delta^+] \leftarrow$ MinNetworkDistInterval($q$, $e$, $s_q$)
29.             **if** $\delta^- < D_k$ **then**
30.                 Enqueue(Key=$\delta^-$, Value=$(r, [\delta^-, \delta^+], q, 0)$, *Queue*)
31.                 UpdateD$_k(L, p, \delta^+, D_k, k)$
32.             **end-if**
33.         **end-for**
34.     **else** ($*$ $p$ is an object $*$)
35.         Enqueue(Key=$\delta^-$, Value=$(p, [\delta^-, \delta^+], u, d)$, $Queue_1$)
36.         $r \leftarrow r + 1$
37.         **if** $r = k$ **then**
38.             kMinDist $\leftarrow \delta^-$
39.         **end-if**
40.     **end-if**
```

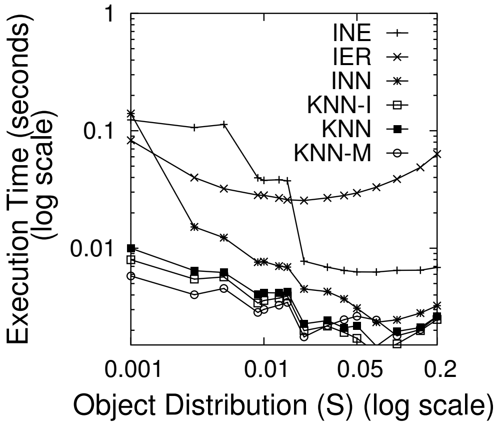
```
41. end-while
42. while (not IsEmpty ($Queue_1$)) and $f < k$ do
43.     $(p, [\delta^-, \delta^+], u, d)$ ← Value(Dequeue($Queue_1$)) (∗ Extract top element ∗)
44.     if $\delta^+ <$ kMinDist then
45.         report $p$
46.         $f \leftarrow f + 1$
47.     else
48.         $(\_, [\mu^-, \mu^+], \_, \_)$ ← Value(FrontPriorityQueue($Queue_1$))
49.         if Intersects($[\mu^-, \mu^+], [\delta^-, \delta^+]$) then
50.             $(u, d, [\delta^-, \delta^+])$ ← RefineNetworkDistInterval($q, u, p, d, [\delta^-, \delta^+]$)
51.             Enqueue(Key=$\delta^-$, Value=$(p, [\delta^-, \delta^+], u, d)$, $Queue_1$)
52.         else
53.             report $p$
54.             $f \leftarrow f + 1$
55.         end-if
56.     end-if
57. end-while
```

3.6 Experimental Evaluation

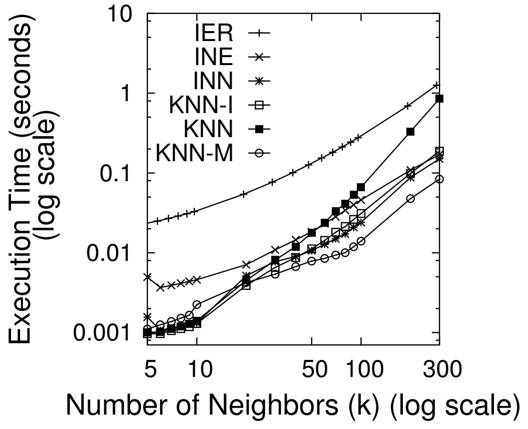
In this section, we evaluate the performance of our k-nearest neighbor algorithm and a number of its variants. We also compare them with two competing techniques—INE and IER of Papadias *et al.* [127] that are based on the use of Dijkstra's algorithm. They differ on the extent to which they make use 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 [144]. All of the experiments were carried out on a Linux (2.4.2 kernel) quad 2.4 GHz Xeon server with one gigabyte of RAM. We have implemented our algorithms using GNU C++. We tested our algorithms on a large road network dataset corresponding to the important roads in the eastern seaboard states of USA, consisting of $91,113$ vertices and $114,176$ edges. The shortest path quadtree of the vertices of this road network was precomputed and stored on disk. The average number of 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 [127]). 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. Another reason for this assumption is that the INE algorithm as described in [127] produces incorrect output when the objects are associated with the edges of a spatial network. Even though our alternate formulation of the INE algorithm remedies this problem, our experimental results show that our version of the INE algorithm, which works both when the objects are restricted to be coincident with the vertices and when they can also lie on the edges of the spatial network, is at least two times slower than the INE algorithm described in [127]. We represent the size of S as a fraction of the number of the vertices in the spatial network. We vary the size of S between $0.001n$ to $0.2n$ where n is the size of our input spatial network. 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.

The first and most important series of 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.* [127] 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 demon-



(a)



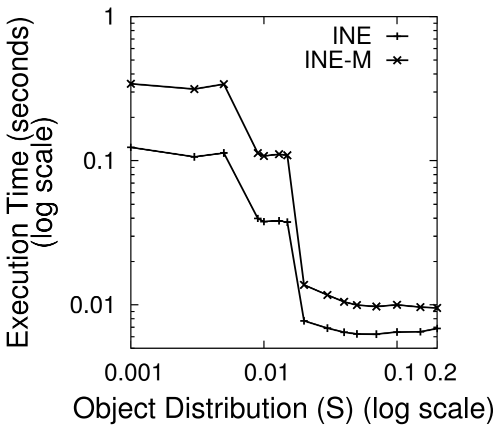
(b)

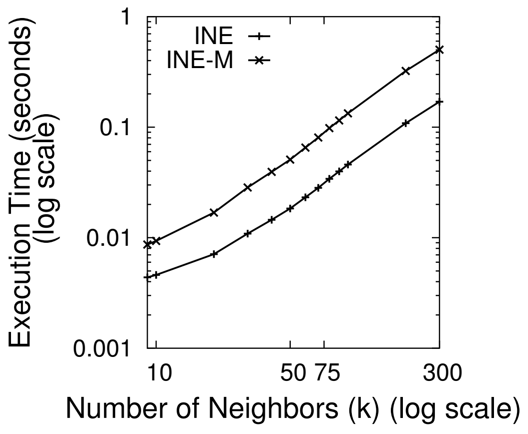
Figure 3.4: Comparison of KNN and its variants with INE and IER for (a) $k = 10$ and varying sizes of S, (b) $S = 0.07n$ and varying k.

strate 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 [127] 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 [144], which had the effect of doubling the execution time of INE (shown below). Figure 3.4 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 3.4a 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 $S >> .2n$, 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 3.4b 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 $S = .07n$. 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.





(a) (b)

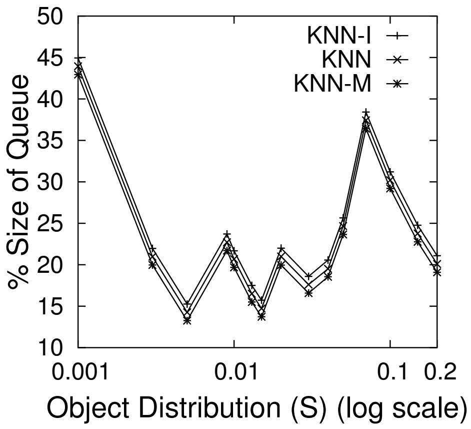
Figure 3.5: Comparison of the INE and INE-M algorithms a) $k = 10$,S varying between $0.001n$ and $0.2n$ b) $S = 0.07n$ and k varying between 5 and 300

As mentioned above, the INE algorithm described in [127] cannot correctly handle objects that lie on the edge of a spatial network. Our alternate algorithm ("INE-M" in Figure 3.5) corrects this problem. We compared the performance of both these algorithms, the results of which are shown in Figure 3.5. Figure 3.5a records the time taken to compute the $k = 10$ nearest neighbors as the value of S was varied between $0.001n$ and $0.2n$. Figure 3.5b records the time taken as k was varied between 5 and 300, while the size of S was kept constant at $0.07n$. In both the figures, we see that the time taken by INE-M to compute k neighbors was at least twice as much as the time taken by INE. One explanation for this observation is that, in most cases, INE-M ends up visiting an edge twice as compared to INE which visits an edge exactly once.

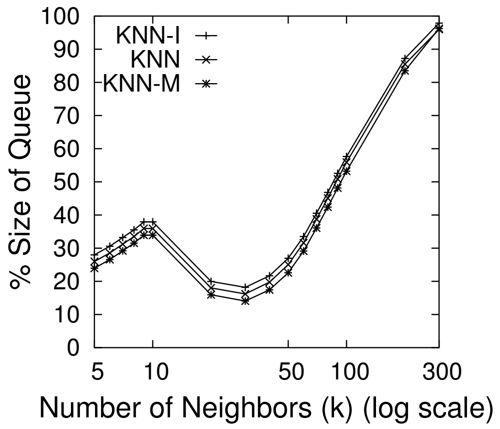
The next set of experiments evaluate some proposed modifications of KNN that are designed to overcome some of its shortcomings. Recall that KNN is a non-incremental best-first algorithm that uses an upper bound estimate D_k on the maximum possible distance to the kth nearest neighbor of a query object 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. The KNEARESTSPATIALNETWORKINCR algorithm (termed 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). Note that in the above discussion, we distinguish between the D_k

estimate used by the KNEARESTSPATIALNETWORK, and the more loose estimate of D^k computed in the KNEARESTSPATIALNETWORKINCR and KMINDISTNEAREST-NEIGHBOR algorithms which we refer to here as D^k_0.

The KMINDISTNEARESTNEIGHBOR algorithm (termed "KNN-M") uses KMINDIST, a lower bound on the minimum of the distance interval of the kth nearest neighbor, in addition to D^0_k, 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^0_k). 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^0_k, 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.



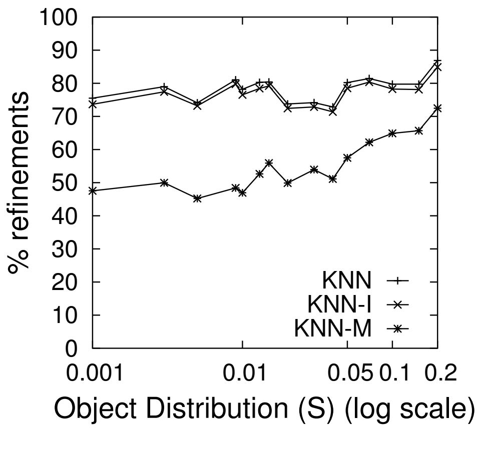
(a)



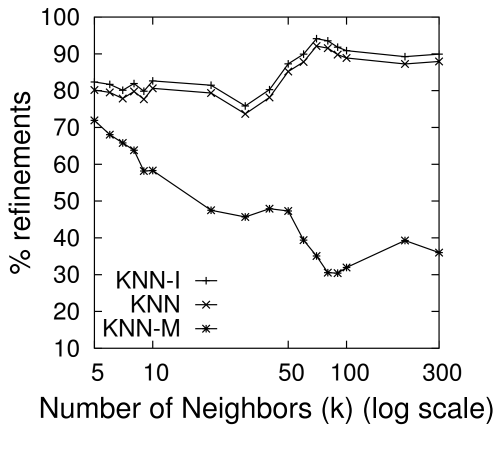
(b)

Figure 3.6: Percentage reduction in the size of the priority 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.

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 3.6 shows the reduction in the maximum size of the priority queue *Queue* for KNN, KNN-I, and KNN-M when compared with INN. For $k = 10$ and varying sizes of S, the maximum size of the priority queue for KNN, KNN-I, and KNN-M is, on the average, at most 35% of the size of the priority queue for INN as shown in Figure 3.6a. Figure 3.6b shows the effect of letting k vary between 5 and 300 on the maximum size of the priority queue, while keeping S fixed at $0.07n$. It is clear from the Figure that there is a large reduction in the size of the priority 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 the priority queue quickly reaches up to 100% of the maximum size of the priority 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.



(a)



(b)

Figure 3.7: 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 3.7a 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 3.7b 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.

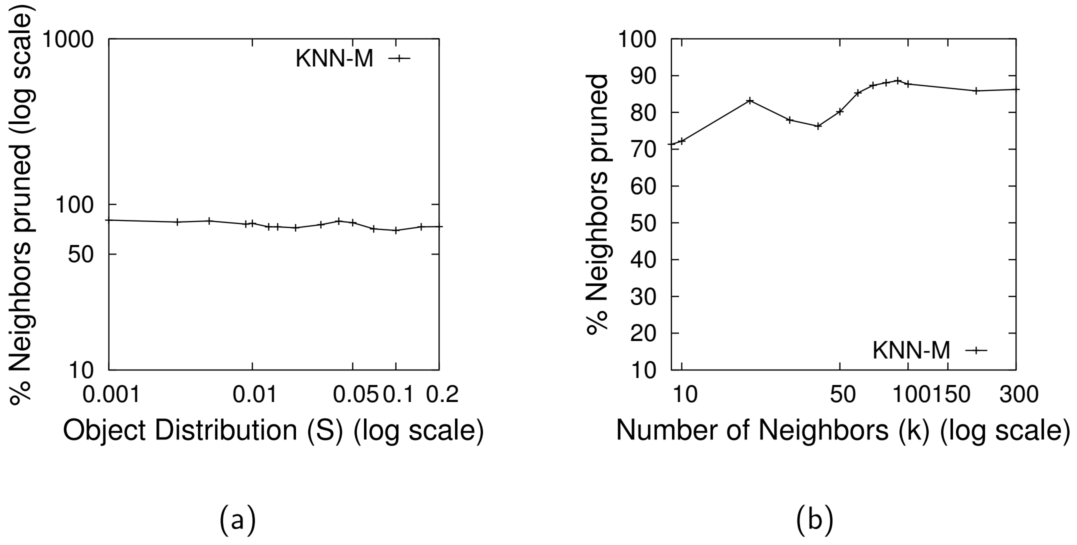


Figure 3.8: Percentage of the objects in the result set that were pruned against the KMINDIST estimate and hence, were added to the result set for (a) $k = 10$, and varying sizes of S, and (b) $S = 0.07n$ and varying values of k.

The observed large savings in the number of refinements performed by KNN-M in Figure 3.7b with increasing k is largely because more and more objects are pruned against the KMINDIST estimate. Figure 3.8 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.

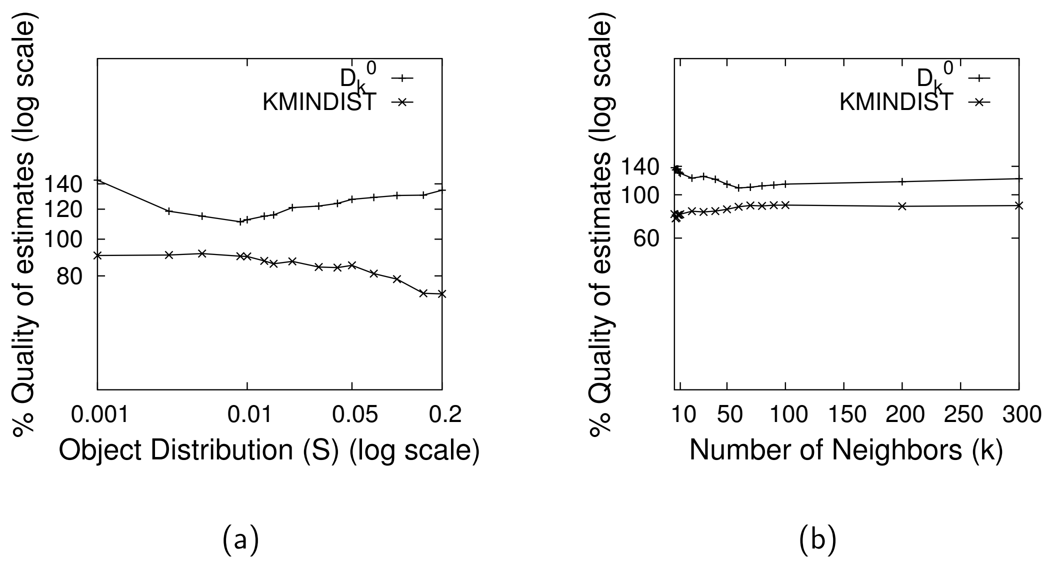


Figure 3.9: The values of D_k^0 and KMINDIST as a percentage of D_k for (a) $S = 0.07n$, varying values of k, and (b) $k = 10$, varying sizes of S.

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 3.9, 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 3.9a) and also varying k and keeping S constant at

0.07n (Figure 3.9b). 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 3.6 for KNN, KNN-I, and KNN-M are almost identical when compared to the maximum size for INN. Moreover, we can see from Figure 3.9 that the KMINDIST estimator 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 the relative performance of KNN and its variants. Figures 3.10a,c show the execution time of KNN and its variants, while Figures 3.10b,d show the corresponding I/O time. Figures 3.10a,b show the effect of varying k on the performance of KNN and its variants when S is fixed at 0.07n, while Figures 3.10c,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 3.10a,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.

- 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 3.10a–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 3.10c. 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.

3.7 Concluding Remarks

We have presented an incremental algorithm for finding nearest neighbors in a spatial network that is an adaptation of an existing incremental nearest neighbor algorithm that computes distance as the "crow flies". Our algorithm is based on the realization that finding the nearest neighbors along a spatial network requires knowing the shortest paths between them, and that computing these shortest paths dynamically as in the methods of Papadias *et al.* [127] that make use of variants of Dijkstra's algorithm is too slow as they must visit all of the vertices that are nearer the query object q than the desired neighbors. Unfortunately, precomputing the shortest paths has the drawback of requiring much space to store them, or even their distance values. Our algorithm overcomes this drawback by taking advantage of the spatial coherence of the shortest paths to the destination vertices from a given source vertex. The spatial coherence is captured with the aid of a quadtree representation which has the added benefit of being dimension-reducing thereby enabling the storage requirements to be reduced to be proportional to the sum of the perimeters of the spatially coherent regions instead of the number of vertices in the spatial network. 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. In addition, the algorithm does not make use of the actual distances between the various pairs of vertices and thus does not need to store them.

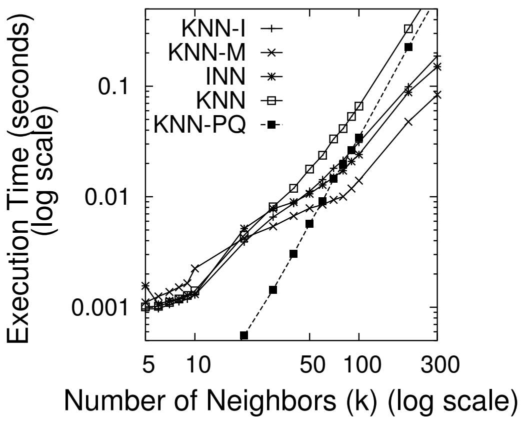
Our algorithm and analysis use a combination of quadtree variants that are based on the region quadtree and the MX quadtree, respectively. They assume a decomposition that is based on both the vertices and a rasterization of the boundaries of the spatially coherent regions. Future work involves attempting to further reduce the space requirements such as, for example, using an edge-based quadtree representation which is a member of the PM quadtree family or a PMR quadtree. Another key feature of our algorithm is the decoupling of the process of computing shortest paths along the network from that of finding the neighbors, and thereby also decoupling 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. Future work involves further generalization of this result to other queries in a spatial network.

In our algorithm, objects and blocks are inserted into the priority queue using the conventional incremental nearest neighbor algorithm in terms of their minimum spatial distance (i.e., "as the crow flies") from the query object q. However, they are removed from the priority queue in order of their increasing network distance from q, which is measured using knowledge of the shortest paths to them. This is in contrast to the methods of Papadias *et al.* [127] which obtain the neighbors either in increasing order of network distance via a direct application of Dijkstra's shortest path algorithm (i.e., the INE method), or by repeatedly invoking the conventional incremental nearest neighbor algorithm to find the neighbors in order of their increasing spatial distance from q while also computing their corresponding network distance from q via a direct application of Dijkstra's algorithm and halting the process when the spatial distance from q of the most recent nearest neighbor is greater than that of maximum network distances of k of the neighbors that have already been computed, where k is the ranking of the sought neighbor with respect to q (i.e., the IER method). The key difference is that in our algorithm the shortest paths between the various vertices in the spatial network are only computed once, whereas in the methods of Papadias *et al.* 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 method 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 of Papadias *et al.* may be preferable, especially if the desired neighbors are quite close to the query object, as the entire spatial network need not be explored.

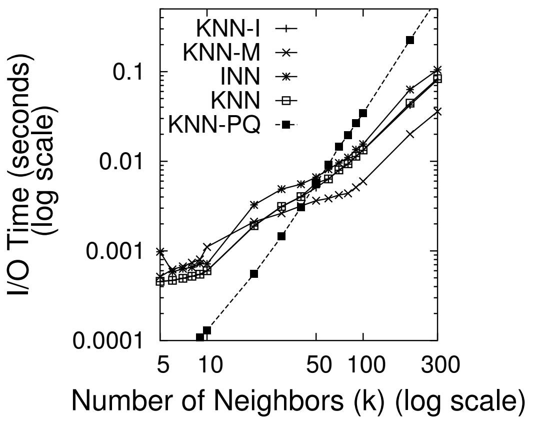
Another advantage of our algorithm is that since the set of objects S from which the neighbors are drawn is decoupled from the actual spatial network, the algorithm (and most importantly the shortest-path quadtrees for the spatial network) can be used with different sets of objects as long as the spatial network is unchanged. For example, we can have separate spatial indexes (i.e., search hierarchies) for gas stations, supermarkets, restaurants, etc. In this case, queries for the nearest gas stations, nearest supermarkets, nearest restaurants, etc. could be executed with no change and the algorithm would be more efficient than had we placed the gas stations, supermarkets, 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 of Papadias *et al.* the distinction between the vertices of the spatial network and the set of objects from which the neighbors 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 non-planar 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 non-planarity 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 non-planar spatial networks.

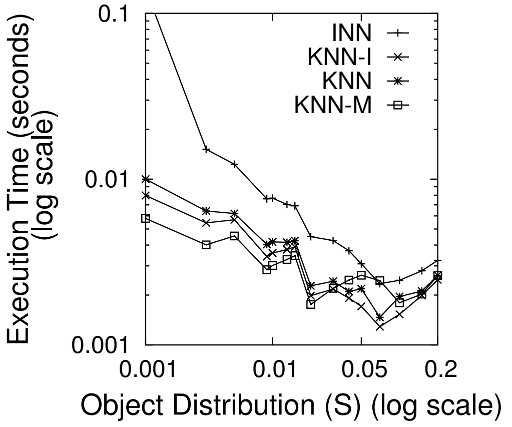
We have also provided a detailed overview of related methods and outlined some of their shortcomings. In particular, we pointed out how the INE method of Papadias et al. may possibly fail to provide the right answer and showed how to remedy it. We demonstrated that both the INE and IER methods of Papadias et al. as well as the improvement suggested by Cho and Chung must visit all of the vertices the length of whose shortest path from q is less than that of the shortest path to the kth nearest object. We pointed out that both INE and IER are really k-nearest neighbor algorithms rather than incremental algorithms (unlike our algorithm which can be executed in both manners), although we did indicate how to transform INE and IER into incremental algorithms. In addition, we showed that the improvement to INE proposed by Cho and Chung [34] does not reduce the number of vertices that are explored, and, instead, only possibly reduces the size of the priority queue. Finally, we pointed out that INE, IER, and the method of Chung and Cho make needless insertions of vertices into *Queue*.



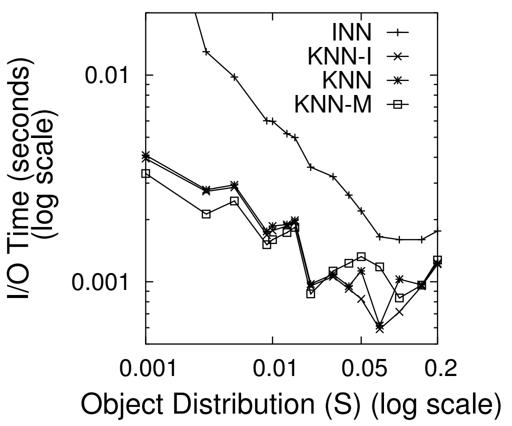
(a)



(b)



(c)



(d)

Figure 3.10: 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.

Chapter 4

Distance Join Queries on Spatial Networks

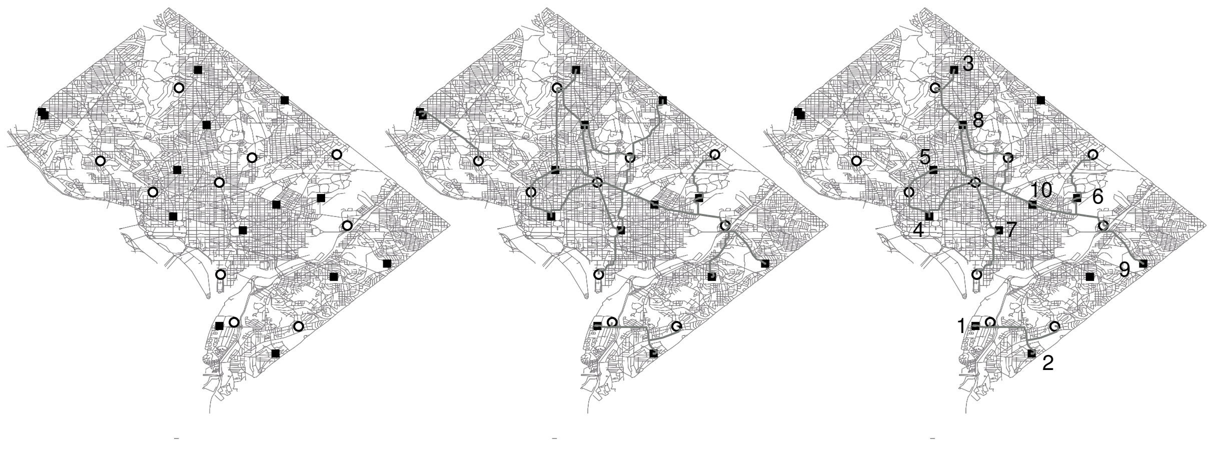


Figure 4.1: Example of a distance join operation on a road network of Washington, DC. a) Objects in R are shown using square icons, and objects in S are shown using circular icons. A subset of the result of a distance join operation, such that b) object pairs at a distance of less than 2.5 miles, and c) the top 10 object pairs in the result.

The distance join operation computes a subset of the Cartesian product $R \times S$ of two sets R and S of a specified order and is based on the distance [82]. The result of a distance join operation is a set P of ordered pairs of objects $<p,q>$, such that $p \in R$ and $q \in S$. In this chapter, we propose distance join operations on a spatial network G,

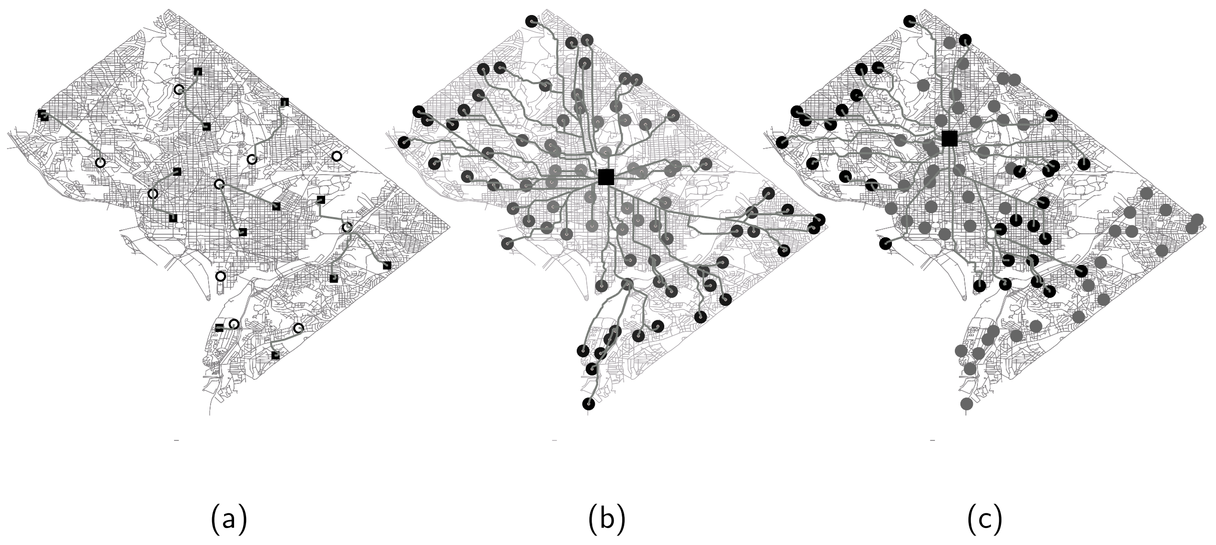


Figure 4.2: a) A subset of the result of a distance semi-join operation on the sets of objects R and S shown in Figure 4.1a. When R is an object, b) an ORDERED distance operation is an incremental nearest neighbor search on S. c) an UNORDERED distance join with a distance restriction is a range search on S.

where R and S are sets of objects in G and the distance of an object pair $<p,q>$ in P is the shortest distance $d_G(p,q)$ from p to q in G. Additionally, spatial and non-spatial constraints could be further imposed either on P, on the object pairs in P, or on both. This results in many different variants of the distance join operation, some of which are discussed below.

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

Distance join operations may generate a very large number of object pairs even for sets of objects R, S of a modest size. For example, R and S both containing $50,000$ objects may generate up to 2.5 billion object pairs. However, in practice, we may be only interested in a small number of pairs in the result set. Hence, computing all the possible pairs in $R \times S$ may result in wasted work. The TOP-K distance join operation computes the first k pairs of objects in P of a distance join operation. Note that the TOP-K constraint implicitly assumes that a distance ordering of the objects in P is specified. Hence, we slightly modify the effect of the TOP-K constraint on ORDERED and UNORDERED distance join operations. An UNORDERED TOP-K distance join operation computes the "TOP-K" object pairs of $R \times S$, although the object pairs in P are still unordered, *i.e.*, we do not establish a total ordering of the k object pairs in P. In contrast, the "TOP-K" constraint applied to an ORDERED distance join operation results in an ordered set of object pairs in P containing k object pairs. Such a join operation has interesting applications to GIS. For example, given a set of locations corresponding to exits R on a highway and a set of restaurants S, we may wish to determine the k closest pairs containing an exit on the highway and a restaurant. Incidentally, such a query is a variant of the "in-route" query proposed in [172]. Figure 4.1c illustrates the top-10 pairs of an ORDERED distance operation on a road network of Washington, DC.

Instead of limiting the cardinality of the result set P using a TOP-K constraint, the distance join operation may be constrained to report only those object pairs that are within a specified minimum d^- and maximum d^+ distance values *i.e.*, $\forall <p,q> \in P$, such that $d^- \leq d_G(p,q) \leq d^+$. For example, given a set of locations R on a road network corresponding to where employees of a chain store reside, and the set of stores S, we can find the set of stores that each employee can reach within ε distance, or alternatively within ε time when the time taken to travel an edges of the road network is provided. Figure 4.1b is an illustration of a distance join operation when the minimum and maximum distance between the object pairs is specified.

An interesting variant of the distance join is the distance SEMI-JOIN, which restricts the number of occurrences of any object $p \in R$ in P. The result of a distance SEMI-JOIN operation computes a set P of object pairs, such that for all $p \in R$, there exists exactly one pair $<p,q_i>, q_i \in S$ in P. The result of an ORDERED distance semi-join operation pairs is an ordered set of object pairs, such that each object in R is paired with its closest object in S. Incidentally, the result of an ORDERED distance semi-join is equivalent to the ANN join [36]. For example, given a set of stores R and a set of warehouses S on a spatial network, the ORDERED distance semi-join associates each store with its closest warehouse. Figure 4.2a is an illustration of a distance SEMI-JOIN operation on a road network.

Incidentally, an ORDERED distance join operation can also be INCREMENTAL – that is, each invocation of the join only computes the next object pair in the result set.

For simplicity sake, we assume that all ORDERED distance join operations are also INCREMENTAL, although this need not always be the case.

In this chapter, we introduce several distance join operations on a spatial network. Our algorithm is based on earlier work in [25, 82, 144, 173] which has now been applied to spatial networks. The "TOP-K" operator is based on the work of Carey and Kossmann in [32]. The distance semi-join operation of R and S, computes the first nearest neighbor to each object in R with neighbors drawn from S. Hence, it is related to the all nearest neighbor join (ANN join) query in [36, 185], although both [36, 185] deal with the problem in an Euclidean space. Moreover, as the first object pair in the result of an incremental distance join is the closest (or farthest) object pair drawn from $R \times S$, our work is also related to the "closest pair" queries introduced in [17]. We present an algorithm that can perform a variety of distance join operations on spatial networks. The inputs to the algorithm determine the nature of the distance join, *i.e.*, the user can choose between ORDERED versus UNORDERED, JOIN versus SEMI-JOIN, and distance pruning versus TOP-K. However, it should be clear that not all combinations of these operations yield meaningful results.

The only techniques that directly compete with our work are the Join Euclidean Restriction (JER) and Join Network Expansion (JNE) techniques of Papadias *et al.* [127], which are both based on Dijkstra's algorithm. Given two sets of objects R and S on a spatial network and a distance ε, the JER and JNE algorithms compute pairs of object pairs $<p,q>$, $p \in S$, $q \in R$, such that the network distance between p and q is less than or equal to ε.

The Join Euclidean Restriction (JER) and Join Network Expansion (JNE) algorithms of Papadias *et al.* [127] take two objects sets R and S on a spatial network as input such that both are indexed using R-tree [73] data structures. The algorithms also take a distance bound $\varepsilon \in \mathbb{R}$ as input. The result of the JER and the JNE algorithms is a set P of object pairs $<p,q>$, such that $p \in R$, $q \in S$, and $d_G(p,q) \leq \varepsilon$.

We now describe the workings of the JER algorithm. It first invokes an Euclidean distance join operation [82, 173] on R and S. This step results in a set P_e of object pairs $<p,q>$, such that $p \in R$ and $q \in S$, such that $d_S(p,q) \leq \varepsilon$. Note that the algorithm assumes that the spatial distance between two objects always lower bounds the network distance between them. So, the Euclidean distance join operation is used as a *filter* step before performing the more expensive network distance join operation. Now, all the unique objects in R that form an object pair in P_e are aggregated into a set R_e. Now, for every object p' in R_e, we aggregate all the objects in S that participates in object pairs of the form $<p',q_i>$ in P_e to form a set $S_{p'}$ consisting of objects in S. Now, the JER algorithm invokes a Dijkstra's algorithm with p' as the source vertex and objects in $S_{p'}$ as destinations, in order to compute the actual network distances of all the objects in $S_{p'}$ to p'. If the network distance of an object q' in $S_{p'}$ is less than ε, then the object pair $<p',q'>$ is added to the result set. Note that the JER algorithm invokes an instance of the Dijkstra's algorithm for every object in R_e. This means that the JER algorithm may have to invoke as many instances of the Dijkstra's algorithm as the number of objects in R, which can be large.

The key difference between the JNE and the JER algorithm of Papadias *et al.* [127] is that the JNE algorithm does not use the filter step of the JER algorithm. Instead, it first chooses the smaller to the two datasets among R and S, say R, to be the set of source vertices, while the larger set would form the set of destination vertices. For every object p in R, it first finds a set $S_p \subset S$ of p that are within a Euclidean distance of ε from p. It then invokes an variant of the Dijkstra's algorithm with p as the source vertex, S_p as the set of destination, such that the algorithm terminates when the network distance from p to the latest vertex visited by the algorithm is farther than ε.

Note, however, that both the JER and JNE algorithms are very limited in their capabilities. Neither JER nor JNE can compute distance joins incrementally. Moreover, neither JER nor JNE can perform SEMI-JOIN operations. It is not even possible to specify a network distance join operation with both a lower and an upper limit on distance between the object pairs that are reported cannot be handled efficiently by either the JER or JNE techniques. To perform such a query using JER or JNE would require obtaining all the object pairs that satisfy the upper bound on the distance between the object pairs and then pruning the result against the lower bound distance. Finally, it is not possible obtain the object pairs in P in an ordered fashion and hence, it is not possible to perform

TOP-K operations.

The rest of the chapter is organized as follows. Our join algorithm is presented in Section 4.1. Experimental results are discussed in Section 4.2, while concluding remarks are drawn in Section 4.3.

4.1 SILC Distance Join

Given a spatial network G, we assume that the SILC framework of G has been precomputed and stored on disk using an offline process. That is, for each vertex $v \in V$, we assume that the shortest-path quadtree has been precomputed and is available to our distance join algorithm. Recall, that the path information in the SILC framework is explicitly represented, while the network distance is implicitly recorded. Recall that each block b in the shortest-path quadtree m_s of s also records two distance values. In particular, for every vertex u in b, we first compute the ratio of the network distance $d_G(s,u)$ and the spatial distance $d_S(u,v)$ between s and u, and then store the minimum (λ^-) and the maximum (λ^+) values of the ratio among all the vertices in b. Using the two stored values λ^-, λ^+, we are able to quickly compute an approximate network distance between two objects as described in procedure GETNETWORKDISTINTERVAL, and between an object and a region on the spatial network as described in procedure MINNETWORKDISTBLOCK, although now the distances are obtained as intervals. Suppose that $d = [\delta^-, \delta^+]$ is a network distance interval. We refer to δ^- as the minimum of the network distance interval and to δ^+ as the maximum of the network distance interval.

Given two objects u and v on a spatial network, the SILC framework enables the computation of a network distance interval $[\delta^-, \delta^+]$, such that $\delta^- \leq d_G(u, v) \leq \delta^+$. Furthermore, we can tighten the network distance interval of u and v using the REFINE-NETWORKDISTINTERVAL operator, which improves the distance interval by expending work. Note that a network distance interval can be only be *refined* at most $|\pi_G(u, v)| - 1$ times, after which $\delta^- = \delta^+ = d_G(u, v)$.

Similarly, the network distance interval $[\delta^-, \delta^+]$ between an object v and a block r *contains* the actual network distance between v any object contained in v. The MINNETWORKDISTBLOCK operator in the SILC framework corresponds to the MINDIST(MAXDIST) operators commonly used in traditional spatial database query processing.

One of the drawbacks of the SILC framework is that the network distance interval between a pair of blocks $<b_1, b_2>$, or between a pair $<b, o>$ containing a block b and an object o, cannot be easily computed. We remedy the first problem by assuming that the *spatial* distance between two objects in a spatial network always lower bounds the network distance between them. For example, in the case of a road network, we can easily verify that the *geodesic*, or the *Euclidean* distance between the two vertices is always less or equal to the network distance between them. Thus, in our algorithm, we approximate the network distance the block pair $<b_1, b_2>$ with the spatial distance

$d_S(b_1,b_2)$ between them, which is easy to compute. We will avoid generating pairs of the second kind in our algorithm, thus circumventing the problem. In other words, we will modify our algorithm so that we will never generate pairs of the form $<b_1,o_1>$, where b_1 is a block and o_1 is an object.

In this section, we describe our distance join algorithm. We first assume that the SILC framework of a spatial network has been precomputed and stored on disk. Furthermore, we assume that we are provided with two sets of objects S and R on the spatial network. For the sake of simplicity, we assume that each object in R and S is associated with a vertex in the spatial network, although the algorithm can be easily modified to allow objects to be associated with the edges of the spatial network as in [127]. Moreover, we also assume that the objects in both R and S are indexed using a hierarchical disk-based data structure (e.g., PMR quadtree [144], or R-tree [73]).

Algorithm 10

Procedure DISTJOIN[T, U, $[d^-,d^+]$, k, OPTION]

Input: OPTION can be ORDERED or UNORDERED, JOIN or SEMI-JOIN, TOP-K or distance restriction

Input: $T,U \leftarrow$ root node of hierarchical structures on R and S

Input: $[d^-,d^+]$ are bounds on the network distance between the object pairs in the result, initially set to $-\infty$ and ∞, respectively.

Input: $S_o \leftarrow$ set of objects o_i whose pairs $<o_i, o_j>$ have already been reported. It is initially empty and used only if OPTION = SEMI-JOIN. If OPTION $\neq$ SEMI-JOIN, then the condition $p \notin S_0$ in lines 54, 61, and 65, is always *true*

Input: $k \leftarrow$ number of object pairs in result, default value 1

(* $d_k \leftarrow \infty$, distance estimate to the k^{th} object pair *)

Output: P: pairs of object $<p,q>$, $p \in R$ and $q \in S$

```
1.  INIT: $[\delta^-, \infty] \leftarrow d_S(T,U)$
2.          if (OPTION = ORDERED) then
3.              $Q \leftarrow$ NEWPRIORITYQUEUE()
4.          else
5.              $Q \leftarrow$ NEWLIST()
6.          end-if
7.          INSERT([KEY=$\delta^-$], VALUE=$T, U, [\delta^-, \infty]$, $Q$)
8.  END-INIT
9.  while not ISEMPTY($Q$) do
10.     $(p, q, [\delta^-, \delta^+]) \leftarrow$ VALUE(REMOVETOPELEMENT($Q$))
11.     if (OPTION = TOP-K) then
12.         if ($\delta^- > d_k$) then
13.             continue while-loop
14.         end-if
```

```
15.       Update $d_k$ using $[\delta^-, \delta^+]$ as in [82]
16.    end-if
17.    if $p$ is a NON-LEAF BLOCK and $q$ is a NON-LEAF BLOCK then
18.       for each child block $b_p$ in $p$ do
19.          for each child block $b_q$ in $q$ do
20.             $[\delta^-_{pq}, \infty] \leftarrow d_S(b_p, b_q)$
21.             if INTERSECTS($[\delta^-_{pq}, \infty]$, $[d^-, d^+]$) then
22.                INSERT(KEY=$\delta^-_{pq}$, VALUE=$(b_p, b_q, [\delta^-_{pq}, \infty])$)
23.             end-if
24.          end-for
25.       end-for
26.    else if $p$ is a LEAF BLOCK and $q$ is a NON-LEAF BLOCK then
27.       for each child object $o_p$ in $p$ do
28.          for each child block $b_q$ in $q$ do
29.             $[\delta^-_{pq}, \delta^+_{pq}] \leftarrow$ MINNETWORKDISTBLOCK$(o_p, b_q)$
30.             if INTERSECTS($[\delta^-_{pq}, \delta^+_{pq}]$, $[d^-, d^+]$) then
31.                INSERT(KEY=$\delta^-_{pq}$, $o_p, b_q, [\delta^-_{pq}, \delta^+_{pq}]$)
32.             end-if
33.          end-for
34.       end-for
```

```
35.   else if $p$ is a Non-leaf Block and $q$ is a Leaf Block then
36.     for each child block $b_p$ in $p$ do
37.       $[\delta^-_{pq}, \infty] \leftarrow d_S(b_p, q)$
38.       if Intersects($[\delta^-_{pq}, \infty]$, $[d^-, d^+]$) then
39.         Insert(Key=$\delta^-_{pq}$, $b_p, q, [\delta^-_{pq}, \infty]$, $Q$)
40.       end-if
41.     end-for
42.   else if $p$ is a Leaf Block and $q$ is a Leaf Block then
43.     for each child object $o_p$ in $p$ do
44.       for each child object $o_q$ in $q$ do
45.         $[\delta^-_{pq}, \delta^+_{pq}] \leftarrow$ GetNetworkDistInterval$(o_p, o_q)$
46.         if Intersects($[\delta^-_{pq}, \delta^+_{pq}]$, $[d^-, d^+]$) then
47.           Insert(Key=$\delta^-_{pq}$, Value=($o_p, o_q, [\delta^-_{pq}, \delta^+_{pq}]$)], $Q$)
48.         end-if
49.       end-for
50.     end-for
51.   else if $p$ is an Object and $q$ is a Non-leaf Block then
52.     for each child block $b_q$ in $q$ do
53.       $[\delta^-_{pq}, \delta^+_{pq}] \leftarrow$ MinNetworkDistBlock$(p, b_q)$
54.       if Intersects($[\delta^-_{pq}, \delta^+_{pq}]$, $[d^-, d^+]$) and $p \notin S_o$ then
```

```
55.             INSERT(KEY=$\delta^-_{pq}$, VALUE=($p, b_q, [\delta^-_{pq}, \delta^+_{pq}]$), $Q$)
56.           end-if
57.         end-for
58.     else if $p$ is an OBJECT and $q$ is a LEAF BLOCK then
59.         for each child object $o_q$ in $q$ do
60.           $[\delta^-_{pq}, \delta^+_{pq}] \leftarrow$ GETNETWORKDISTINTERVAL$(p, o_q)$
61.           if INTERSECTS($[\delta^-_{pq}, \delta^+_{pq}]$, $[d^-, d^+]$) and $p \notin S_o$ then
62.             INSERT(KEY=$\delta^-_{pq}$, VALUE=($p, o_q, [\delta^-_{pq}, \delta^+_{pq}]$), $Q$)
63.           end-if
64.         end-for
65.     else if $p \notin S_o$ then   (* $p$ and $q$ are objects *)
66.         if (OPTION = UNORDERED) then
67.           if ($[d^-, d^+]$) CONTAINS $[\delta^-, \delta^+]$ then
68.             if (OPTION = SEMI-JOIN) then
69.               add $p$ to $S_o$
70.             end-if
71.             report $<p, q>$
72.           elseif INTERSECTS($[\delta^-_t, \delta^+_t], [\delta^-, \delta^+]$) then
73.             REFINENETWORKDISTINTERVAL($[\delta^-, \delta^+]$)
74.             INSERT(KEY=$\delta^-$, VALUE=($p, q, [\delta^-, \delta^+]$), $Q$)
```

```
75.             end-if
76.         else (* OPTION = ORDERED *)
77.             (_, [$\delta_t^-, \delta_t^+$]) ← TOPELEMENT($Q$)
78.             if INTERSECTS([$\delta_t^-, \delta_t^+$], [$\delta^-, \delta^+$]) or not ([$d^-, d^+$] CONTAINS
                [$\delta^-, \delta^+$]) then
79.                 REFINENETWORKDISTINTERVAL([$\delta^-, \delta^+$])
80.                 INSERT(KEY=$\delta^-$, VALUE=($p, q$, [$\delta^-, \delta^+$]), $Q$)
81.             else
82.                 if (OPTION = SEMI-JOIN) then
83.                     add $p$ to $S_o$
84.                 end-if
85.                 report <$p, q$> (and return)
86.             end-if
87.         end-if
88.     end-if
89. end-while
```

Algorithm 10 describes our distance join algorithm on a spatial network G which is based on the algorithm by Hjaltason and Samet [82] and Shin *et al.* [173]. The algorithm takes two hierarchical data structures T and U, PMR quadtrees in our case, on the spatial positions of the objects in R and S, respectively. Even though, in the description of the algorithm, we assume T and U to be quadtrees, our discussion is equally applicable to both object hierarchies, such as R-tree [73], as well as space hierarchies such as quadtrees and its variants [144]. The output of our algorithm is a set P of object pairs $<a,b>$, such that $a \in R$ and $b \in S$. In addition to T and U, our algorithm also accepts a network distance interval $[d^-, d^+]$ corresponding to the minimum and maximum bounds on the network distances between the object pairs in P. That is, by specifying network distance interval $[d^-, d^+]$, we are restricting the output pairs in P to only contain those object pairs $<a,b>$ that are between d^- and d^+. The next parameter k provides an upper bound on the number of object pairs to be computed by the join algorithm. Finally, the OPTION parameter serves to differentiate ORDERED and UNORDERED output of object pairs, and between a JOIN and a SEMI-JOIN. If OPTION is set to ORDERED, then the first invocation of the algorithm returns the first (closest according to the distance measure) object pair in the result set P. Subsequent object pairs in the result set are obtained by repeated invocation of the algorithm, *i.e.*, each invocation computes and returns only the next object pair in the result set. If OPTION is set to UNORDERED, then the algorithm returns a set of object pairs corresponding to the result of the distance join operation.

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

In line 10, we obtain the pair $<p,q>$ from the *head* of Q. Let $[\delta^-,\delta^+]$ be the network distance interval between p and q. The algorithm's control structure is such that blocks p and q are split in an *asymmetric* manner in order to avoid obtaining pairs of the form $<p,q>$, where p is a block and q is an object. What we mean by that is that if $<p,q>$ is a block pair, we further test to see if q is a *leaf block*. If so, we only split p and keep q as is. What we have done here is that we have avoided generating block-object pairs which would have resulted has we split the pair $<p,q>$ symmetrically. As computing the network distance interval from a block to an object using the SILC framework is expensive, we have avoided generating such pairs.

If p is a block and q is a *non-leaf* block (lines 17–34), then they are split into their c_p and c_q children elements, respectively. The resulting $c_p \cdot c_q$ pairs of children of p and q are inserted into Q after computing their initial network distance intervals. Such a splitting strategy (termed "Simultaneous" in Section 4.2), may not be suitable when both p and q have a large outdegree as it may result in an explosion of pairs in Q. Another strategy (termed "Even" in Section 4.2) would be to split the block pairs more evenly *i.e.*, each time the larger one in p and q is split.

As we approximate the network distance between a a pair of blocks with its spatial distance, we may adopt different strategies to break ties between pairs of blocks at the same distance, each resulting in a different traversal of T and U. Choosing the pair with a block at the deepest level, results in a *depth first* traversal of T and U, while choosing the pair with a block at the shallowest level results in a *breadth first* traversal of the tree structures. These two competing strategies (termed "DFS" and "BFS") are further explored in Section 4.2.

In lines 35–50 of the algorithm, we handle pairs $<p,q>$, such that p is a block and q is a leaf block. Lines 51–64 handle the case when p is an object and q is a block. In the above two cases, the network distance interval of the c_p (c_q) children of p (q) to q (p) are computed and inserted into Q.

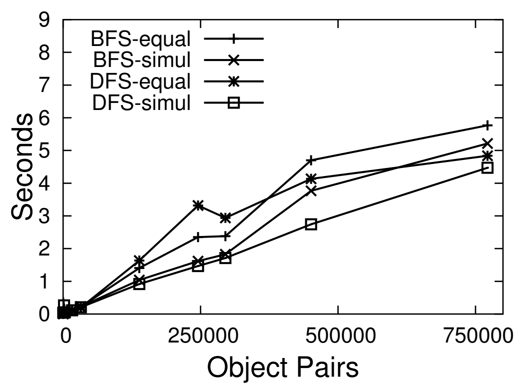
The final case when both p and q are objects is handled in line 65. If OPTION is set to UNORDERED, then the network distance interval $[\delta^-,\delta^+]$ between the object pairs is first checked against $[d^-,d^+]$ for *containment*. Next, if the two intervals do not intersect, then $<p,q>$ cannot belong to the result set and is pruned. If the network distance interval $[\delta^-,\delta^+]$ intersects, but is not contained in $[d^-,d^+]$, then the network distance of $<p,q>$ is *refined*, and the pair $<p,q>$ along with its refined network distance interval is inserted back into Q. If the distance interval $[\delta^-,\delta^+]$ is contained in $[d^-,d^+]$, then the object pair is added to the result set P.

If OPTION is set to ORDERED, then $<p,q>$ cannot be reported unless it is certain that it is not being reported “out of order”. In addition to checking against the input network distance constraint $[d^-, d^+]$, we also need to compare it against the next element in the priority queue Q. The network distance interval of $<p,q>$ is compared against the network distance interval of the current top element in line 78 for intersection. If $[\delta^-, \delta^+]$ is disjoint from the network distance interval $[\delta_t^-, \delta_t^+]$ of the current “top” pair in Q, and δ^+ is less than δ_t^-, then $<p,q>$ is reported as the “next” object pair in the result. The algorithm at this point (line 85) returns the control back to the user. Subsequent object pairs can be obtained incrementally by invoking the algorithm as many times as needed. If the two network distance intervals $[\delta^-, \delta^+]$ and $[\delta_t^-, \delta_t^+]$ intersect, then it is not clear if $<p,q>$ is the next object pair in the result set. Hence, the network distance interval of $<p,q>$ is refined as before and reinserted back into Q (as shown in lines 79–80).

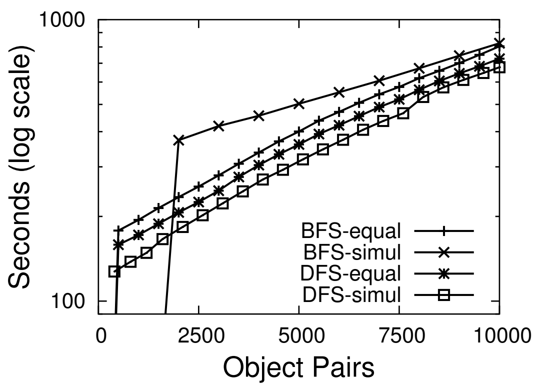
If OPTION is set to SEMI-JOIN, then the algorithm computes a distance semi-join operation on the two input datasets. The distance semi-join requires that the algorithm keep track of the pairs that have been already reported —that is, if an object pair $<o_1, o_i>$ has already been reported, subsequent object pairs of the form $<o_1, o_j>$ should be pruned. In other words, an object $o_i \in R$ can only participate in one of the object pairs in P, while no such restriction is made on an object from R. We achieve this by storing a list S_o of objects in S, initially empty, that have already been reported by the algorithm. In particular, object $p \in R$ is added to S_o in line 83 thereby no subsequent pair containing p would be reported by the algorithm. This facilities pruning of pairs in lines 54, 61, and 65, by comparing each pair with the objects in S_o. More aggressive pruning strategies, described in [82], can be employed here, which may result in further reduction in the number of elements in Q.

When OPTION is set to TOP-K, we would estimate, as in [82], the upper bound d_k to the network distance of the k^{th} object pair in the result set. This would enable us to prune object pairs (line 11–16) that cannot possibly be present in the top k result. We use a separate priority queue, as described in [82], in order to estimate the value of d_k. We could have also used the technique described in [173] and leave the investigation of its use to a future study. Now, using d_k as the upper bound, we are able to prune object pairs, while constantly improving the estimate as more pairs are examined by the algorithm.

One interesting observation is that when T is an object, and if OPTION is set to ORDERED, Algorithm 10 is identical to the Best First Search (BFS) algorithm [80]. If T is an object, OPTION is set to UNORDERED and a distance constraint is specified, our algorithm performs a *range search*. If T is an object and the TOP-K constraint is specified, the algorithm retrieves the k nearest neighbors to T on the spatial network.



(a)



(b)

Figure 4.3: Execution time for list ordering and different block pair splitting strategies for an a) UNORDERED, b) ORDERED distance join operation on a road network.

4.2 Experimental Results

In this section, we evaluate the performance of our algorithm, and compare it with two other competing techniques reported in the literature – the Join Euclidean Restriction (JER) and Join Network Expansion (JNE) techniques of Papadias *et al.* [127]. Both of these algorithms are based on Dijkstra algorithm. Given two sets of objects R and S on a spatial network and a distance ε, the JER and JNE algorithms compute pairs of object pairs $<p,q>$, $p \in S$, $q \in R$, such that the network distance between p and q is less than or equal to ε. Note however that both these algorithms are limited in their capabilities. Neither JER nor JNE can compute distance joins incrementally. Moreover, neither JER nor JNE can perform SEMI-JOIN operations. Finally, a distance join operation that specifies both a lower and an upper limit on distance between the object pairs that are reported cannot be handled efficiently by either the JER or JNE techniques. To perform such a query using JER or JNE would require obtaining all the object pairs that satisfy the upper bound on the distance between the object pairs and then pruning the result against the lower bound distance.

All of the experiments were carried out on a Linux (2.4.2 kernel) quad 2.4 GHz Xeon server with one gigabyte of RAM. We have implemented our algorithms using GNU C++. We tested our algorithms on a large road network dataset corresponding to the important roads in the eastern seaboard states of USA, consisting of $91,113$ vertices and $114,176$ edges. The SILC encoding of this road network was precomputed and stored on disk. The average number of the Morton blocks in the SILC map of a vertex in the dataset is 353. Objects belonging to the sets, R and S, were chosen at random from the vertices of the spatial network. The algorithm uses an LRU based cache that can hold 5% of the disk pages in the main memory.

In our experiments we examined the effect of choosing various block pair splitting strategies, as well as the various tree traversals on the performance of the algorithm. A block pair $<p,q>$ can be split simultaneously (termed "Simul") into children blocks *i.e.*, both p and q are split into children blocks, or only the bigger block among p and q is split (termed "Even"). Also, the algorithm can choose between a *depth first* (DFS) or a *breadth first* (BFS) traversal of T and U. The effect of choosing one of these four variants on an UnOrdered distance join is shown in Figure 4.3a. From the graph it is clear that depth first traversal ("DFS") with simultaneous ("Simul") splitting of block pairs was found to be slightly better than the other techniques. Figure 4.3b shows the effect of these variations on an INCREMENTAL distance join operation. Again, we can see that depth first traversal ("DFS") with simultaneous ("Simul") splitting of block pairs was found to be slightly better than the other techniques.

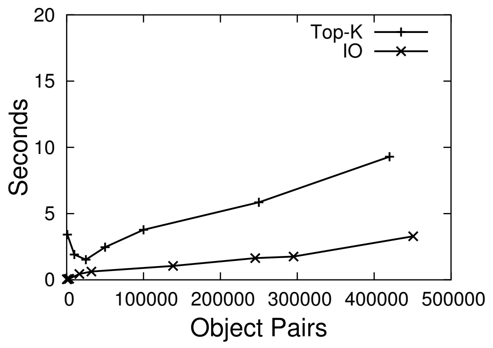


Figure 4.4: Execution time for a "TOP-K" distance join operation with different values of k.

Figures 4.4–4.5 shows the performance of the TOP-K distance join and semi-join operations on a road network dataset. Figure 4.4 shows the effect of varying the value of k on the performance of TOP-K UNORDERED distance joins between two sets of objects, R and S containing 1000 and 450 objects, respectively. Notice that the TOP-K UNORDERED distance join operation is slightly more expensive than an UNORDERED distance join operation shown in Figure 4.3a. Figure 4.5 shows the effect of varying the value of k on the performance of an UNORDERED distance semi-join operation with 18000 objects in R and 4500 objects in S. Notice that a larger proportion of the cost in Figure 4.5 is attributed to the CPU than in Figure 4.4, indicating that the algorithm expends a large number of clock cycles to ensure that an object pair is not present in S_o.

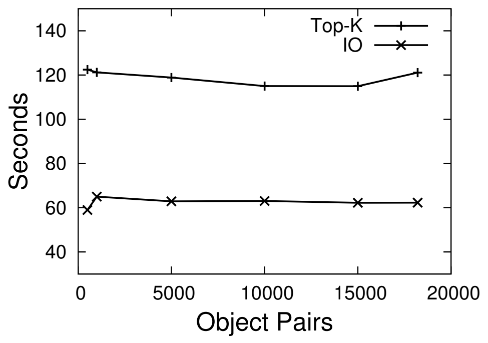
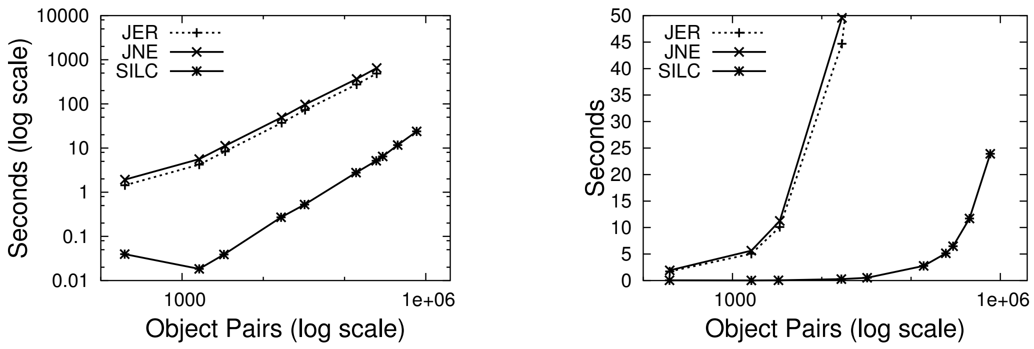


Figure 4.5: Execution time for a "TOP-K" distance semi-join operation with different values of k.



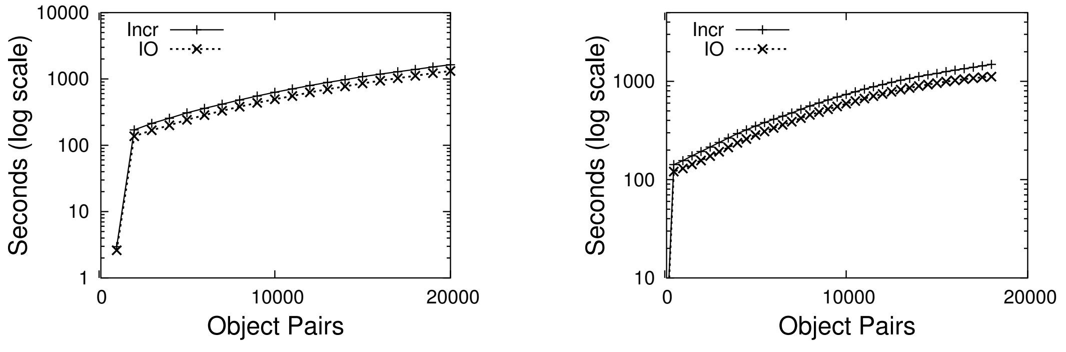
(a) (b)

Figure 4.6: a) Execution times for the JER, JNE and our method ("SILC") is shown for distance join operation with a limit on the maximum distance between the object pairs. b) Figure in (a) has been redrawn (y axis not in logscale) in order to contrast the relative performance of the methods.

Figure 4.6 is a comparison of the time taken to perform distance join operations with an upper bound restriction on the network distance between object pairs using our approach (termed "SILC") with the JER and the JNE techniques. Our implementation of the JER and the JNE techniques is slightly different from the original formulation of Papadias *et al.* in [127]. In particular, we associate each object is R and S with a vertex in the spatial network, whereas Papadias *et al.* associate each object with an edge in the spatial network. Consequently, the "find_entities" operation discussed in [127] is no longer performed in our implementation of the JER and the JNE algorithms. Figure 4.6 shows that our technique, when compared to the JER and the JNE techniques, is at least an order of magnitude faster.

Figure 4.7a shows the performance of an incremental distance join operation on a road network dataset. As we can see from Figure 4.7a, the bulk of the time taken by the join operation is spent on performing disk IO. Incidentally, a bulk of the IO time resulted from the repeated invocation of the REFINE operator in line 73 of Algorithm 10. In effect, as the result of this operation is ordered, it has to establish a total ordering of the pairs in the result set, thereby resulting in several invocations of the REFINE operator. One possible explanation is that choosing R and S uniformly has resulted in a large number of objects in S to be at the same distance interval with respect to objects in R thereby requiring many REFINE operations.

Figure 4.7b shows the performance of a distance semi-join algorithm on the road dataset. The performance of this algorithm is similar to Figure 4.7a.



(a) (b)

Figure 4.7: Execution time for incremental a) distance join and b) distance semi-join algorithms

4.3 Summary

In this chapter, an algorithm that can perform a variety of distance join operations on spatial networks was presented. The result of a distance join operation on two sets of objects R,S on a spatial network G is a set P of object pairs $<p,q>$, $p \in R$, $q \in S$ such that the distance of an object pair $<p,q>$ is the network distance from p to q in G. We also introduced several variations — such as UNORDERED, INCREMENTAL, TOP-K, and SEMI-JOIN— of the distance join algorithm that impose additional constraints on the distance between the object pairs in P, the ordering of object pairs in P, and on the cardinality of P. An ORDERED distance join reports the object pairs in P in an increasing (or decreasing) order of the network distance between the object pairs, while no such ordering is specified in the case of the UNORDERED variant of the algorithm. If each invocation of the distance join only computes the next object pair in the result set, the join is said to be INCREMENTAL. A distance SEMI-JOIN restricts the number of occurrences of any object $p \in R$ in P, while a TOP-K distance join only computes the k closest object pairs in P.

Our distance join algorithm assumes that the shortest-path quadtrees of all the vertices in the spatial network G have been precomputed using an offline process. Our algorithm is similar to the one proposed by Hjaltason and Samet [82], and Shin *et al.* [173]. Two key differences between our algorithm and the algorithms in [82, 173] are the following. First, all the distances are in terms of network distance intervals. Second, the SILC framework cannot easily compute the network distance interval between a block and an object, or between two blocks. So, we approximate the network distance between two blocks by their spatial distance, and avoid generating pairs containing a block and an object. We compared our algorithm with the JNE and the JER algorithms of Papadias *et al.* [127]. Both of these algorithms are based on Dijkstra's algorithm, and hence unsuitable for large spatial networks. Experimental results demonstrate up to an order of magnitude speed up when compared with the JER and JNE algorithms.

One possible way of obtaining the network distance interval between a block and an object using the SILC framework would be to compute an *inverse* shortest-path quadtree, in addition to the shortest-path quadtree, for each vertex in a spatial network G. An inverse shortest-path quadtree of a vertex w is the spatial aggregation of the vertices based on which source vertices share the same last link in their shortest paths to w.

Such an inverse shortest-path quadtree would enable the computation of the network distance interval between a block and an object, although at the expense of additional disk storage.

Chapter 5

Path Coherent Pair Decomposition

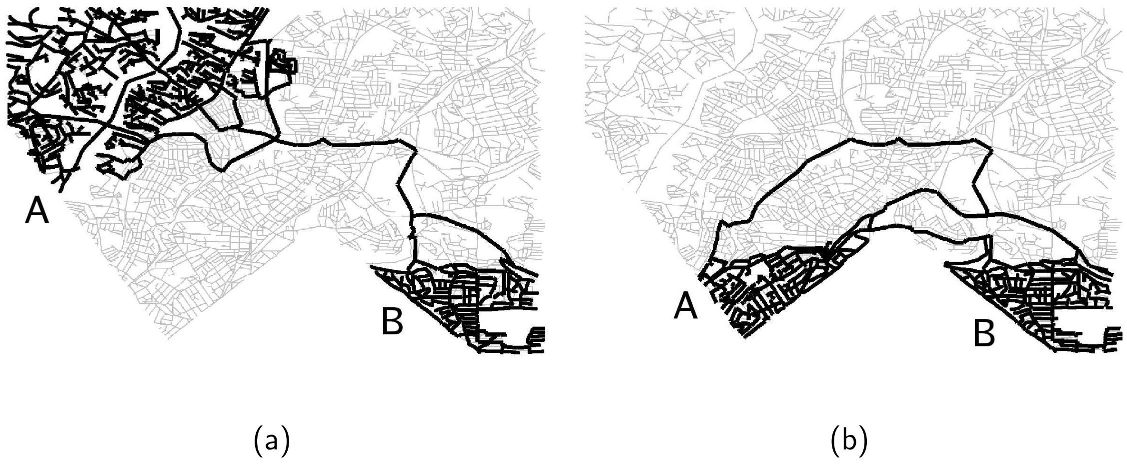


Figure 5.1: Two subsets A, B of vertices in a road network of Silver Spring, MD. The shortest paths between all pairs of vertices in A and B are marked in a darker shade. (a) The 30,000 shortest paths pass that through a single vertex, and (b) the 20,000 shortest paths that pass through one of the two vertices

The focus of this work is the compact encoding and retrieval of the shortest paths and distances on a spatial network. As we pointed out in Section 2.2 a brute-force encoding of the shortest path explicitly stores the shortest path $\pi_G(u,v)$ between every pair of vertices u and v in V. This representation requires $O(n^3)$ storage. In this case, given a source s and destination t, the set of objects (i.e., edges or vertices) that make up the shortest path $\pi_G(s,t)$ can be retrieved in $O(1)$ time, although in a technical sense retrieving the actual objects (i.e., the individual vertices or edges) that make up $\pi_G(s,t)$ takes $O(k)$ time, where $k = |\pi_G(s,t)|$ is the number of vertices or edges in $\pi_G(s,t)$. We also pointed out that the storage requirement can be easily reduced to $O(n^2)$, by recording $l_u(v)$, for every pair $u, v \in V$, while retrieving the shortest path still takes $O(k)$ time. The brute-force distance encoding also records $d_G(u,v)$ for every pair $u, v \in V$, requiring a total of $O(n^2)$ space, while enabling the retrieval of $d_G(u,v)$ in $O(1)$ time.

In this Chapter, we show how to reduce the space requirements even further to $O(n)$ by making use of the observation that the shortest paths between spatially proximate source vertices and spatially proximate destination vertices will often pass through a common vertex (termed *path coherence*). This common vertex is associated with the relevant groups of source and destination vertices. The partitioning of the vertices into appropriate subsets of source and destination vertices is achieved by appealing to the well separated pair decomposition [28,31], and conditions under which it is satisfied for a spatial network are specified here. Using this solution enables the iterative process of finding the shortest path to be achieved in $O(k \log n)$ time, where the next vertex in the shortest path from a particular vertex to the destination vertex can be found in $O(\log n)$ time.

We also show that the SILC framework requires as much as $O(n \log n)$ space, but can retrieve the next link in the shortest path in average $O(\log \log n)$ time. Our results can be compared to those of Gupta *et al.* [71] who propose a distance encoding of a planar graph restricted to integer edge weights that requires $O(n\sqrt{n})$ space. The network distance between two vertices, in their approach, can be retrieved in $O(\sqrt{n})$ time, while the shortest path can be obtained in $O(k\sqrt{n})$ time. In Section 5.4, we discuss the classes of spatial networks for whose elements our approach can be applied. We have used our approach in conjunction with a *progressive refinement* optimization strategy of distance computation [154] to implement a number of algorithms for basic spatial database queries on a spatial network with little or no modifications to the algorithms.

At the very onset, we contrast our work to the work of Frederickson [52] who presents a technique that compactly encodes the $O(n^2)$ shortest paths of a planar graph $G(V,E)$ in $O(np)$ space, where n is the number of vertices in G and p is the cardinality of a smallest subset of faces that covers all the vertices in G. The value of p ranges between 1 and $O(n)$. Frederickson's approach is not applicable to non-planar spatial networks. This is a major shortcoming as many spatial networks that are of practical significance (e.g., road networks) are non-planar. However, Frederickson's approach can be applied to planar spatial networks. The value of p for planar spatial networks whose faces have fewer than c vertices, where c is a constant, is $O(n)$, consequently resulting in $O(n^2)$ storage.

Our framework can be used to incrementally find the nearest neighbors to a query point q from a set of locations P on the spatial network. On the other hand, the IER and

INE techniques of Papadias *et al.* [127], and the subsequent improvements to the INE technique by Cho and Chung [34], use a technique that is based on Dijkstra's shortest path algorithm to find the k-nearest neighbors [162] to a query object q on a spatial network. However, their methods are not incremental. Kolahdouzan and Shahabi [101] speed-up the process of finding k-nearest neighbors by precomputing the Voronoi diagram for a set of locations P on a spatial network from which the k neighbors to q are drawn. Different variants of the *landmark* approach are also used by Jing *et al.* [93], Filho and Samet [44], and Goldberg and Harrelson [61] to speed up Dijkstra's algorithm.

There are three key differences between our approach and those just discussed. First, our technique requires precomputation of the shortest path between every pair of vertices in a spatial network. Although, this can be quite expensive for large spatial networks, it can be achieved with the sufficient investment of time and hardware resources. Our method enables compactly storing the shortest paths in $O(n)$ space and makes a provision for retrieving them efficiently. In this chapter, using extensive theoretical and empirical analysis of our technique, we will show the *scalability* and applicability of our work to very large spatial network databases.

The second difference is that our method has an advantage when multiple datasets share the path encoding to perform queries on spatial networks. For example, a path encoding of Manhattan, NY can potentially be shared by several datasets of landmarks pertaining to postal addresses in Manhattan for query processing. Moreover, spatial networks are usually static structures, while datasets of objects may be updated frequently. For example, when dealing with a set of mobile hosts on a road network, the current positions of the objects are frequently updated, while the road network itself would largely remain static. Moreover, the datasets and the network can be updated independently of each other. In effect, what we have done is *decouple* the *data* from the *underlying domain*. In this respect, our work is distinguished from the work of Papadias *et al.* [127] and Kolahdouzan *et al.* [101] who perform path and distance queries at run-time, while making no provisions to reuse such computations across queries and across datasets. For example, the network Voronoi diagram in [101] computed for a set of restaurants in Manhattan cannot be used for another dataset of post offices in Manhattan.

The third difference is that the focus of our work is not restricted to any particular query on spatial networks. In fact, using the framework proposed in this chapter enables us to perform efficiently many spatial database queries such as distance, range queries, incremental nearest neighbors [80], and spatial joins [82] on spatial networks thereby potentially paving the way for other sophisticated queries to be applied on spatial networks. We use disk efficient indices, such as a B-tree [37] to represent the resulting path and distance encoding of a spatial network. Thus, our technique can be used in conjunction with any existing commercial database system; the design of one such system is briefly mentioned in [156]. Moreover our framework enables current database processing techniques to be applied on spatial networks, thereby encouraging code reuse.

The rest of the chapter is organized as follows. Section 5.1 introduces the concept of a *Well Separated Pairs* (WSP) decomposition [31]. A path encoding using Path Coherent Pairs (PCP) is presented in Section 5.2, and analyzed in Section 5.4. An algorithm for computing shortest paths using the proposed framework is described in Section 5.3. The results of experiments are given in Section 5.5, and, finally, concluding remarks are drawn in Section 5.6.

5.1 Well Separated Pairs

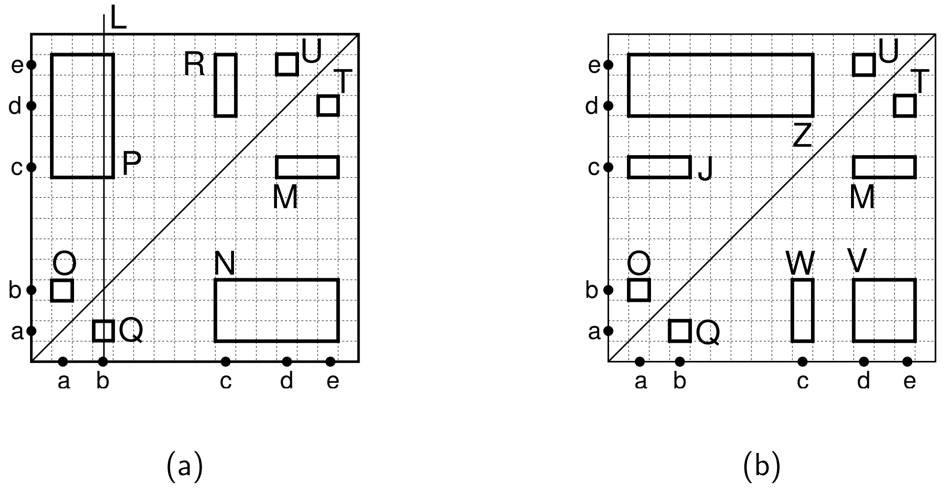


Figure 5.2: Example of a well separated pair (WSP) decomposition of a one-dimensional point set containing 5 points. The separation factors for the decompositions are (a) $s = 1$, and (b) $s = 0.25$

Given a set of points A, the *diameter* of A is the maximum possible distance between any two points belonging to A. Similarly, given two sets of points A and B, the *minimum distance* between A and B is the minimum possible distance between a point in A and a point in B. Two sets of points A and B are said to be *well separated* [1] [31], if the minimum distance between A and B is at least $s \cdot r$, where $s > 0$ is a *separation factor* and r is the larger diameter of the two sets. A well separated pair (WSP) *decomposition* of a point set S, decomposes S into pairs of subsets (A,B) of S, such that $\forall p,q \in S, p \neq q$, there exists exactly one WSP (A,B), such that $p \in A, q \in B$ and that each pair is well separated. The simplest WSP decomposition of a point set S of N points contains $N \cdot (N-1)$ pairs of singleton element subsets (p,q) $\forall p,q \in S, p \neq q$. The key motivation for the use of the WSP decomposition is that for data of dimension d, and a separation factor s, we can always construct a WSP decomposition containing $O(Ns^d)$ pairs in $O(N \log N + Ns^d)$ time. Thus, the number of pairs is reduced to $O(N)$ as s is a fairly small constant independent of N.

As an example, consider the set of 5 one-dimensional points a, b, c, d, e at positions 1.5, 3.5, 9.5, 12.5, and 14.5, respectively. There are a number of possible WSP decompositions for this dataset. Letting $s = 1$, one decomposition consists of $M = (\{d,e\},\{c\})$, $N = (\{c,d,e\},\{a,b\})$, $O = (\{a\},\{b\})$ $P = (\{a,b\},\{c,d,e\})$, $Q = (\{b\},\{a\})$, $R = (\{c\},\{d,e\})$, $T = (\{e\},\{d\})$, and $U = (\{d\},\{e\})$. This decomposition can be visualized by treating the individual pairs that make up the WSP decomposition as rectangles in a two-dimensional space where the axes correspond to the elements that make up the two sets involved in the WSP. For example, Figure 5.2a illustrates the WSP decomposition described above for $s = 1$, while Figure 5.2b illustrates

[1] This is slightly different from the original definition in [31].

another decomposition for the same points with $s = 0.25$. Notice that from the figure, we can see that any vertical (or horizontal) line through one of the points, say p, will cut the disjoint rectangles through which it passes so that the projection of their constituent points onto the y (or x) axis covers all of the points in S with the exception of p. Observe also that just because the WSP (A,B) is a member of a WSP decomposition does not necessarily mean that the symmetric pair (B,A) is also a member of the same WSP decomposition. For example, consider the WSP decomposition in Figure 5.2b where the symmetric pairs of $Z = (\{a,b,c\},\{d,e\})$, $M = (\{d,e\},\{c\})$, and $V = (\{d,e\},\{a,b\})$ are not present. Of course, we do see that together M and V make up the symmetric pair of Z.

5.2 Definition of Path Coherent Pairs

The arrangement of vertices and shortest paths in Figure 5.1a describes a *dumbbell*-like structure. A *path coherent pair* (PCP) (A,B,Ψ) in a spatial network $G(V,E)$ consists of a set of source vertices $A \subset V$, a set of destination vertices $B \subset V$, and a set Ψ which represents a vertex or an edge, such that all the shortest paths between source vertices in A to destination vertices in B contain Ψ. We refer to A and B as the *heads* of the PCP and Ψ as the STEM-CUT of the PCP. An algorithm for computing the STEM-CUT of a PCP is described in detail in Section 5.2.2. A PCP (A,B,Ψ) belongs to one of the two possible configurations given below.

- $(A, B, \Psi = \{w\})$, where $A \cap B = \varnothing$, $w \in V$, $w \notin A$ and $w \notin B$
- $(A, B, \Psi = \{u, v\})$, where $< u, v > \in E$, and $u \notin B$ and $v \notin A$

Note that any edge $< u, v > \in E$ in G is a PCP of the form $(\{u\}, \{v\}, \{< u, v >\})$.

5.2.1 Decomposition of G into Path Coherent Pairs

We now perform a decomposition of G, termed $G \otimes G$, into a set of PCPs, such that the resulting decomposition has the following properties.

1. $G \otimes G = \bigcup_{i=1}^{l} (A_i, B_i, \Psi_i)$, where (A_i, B_i, Ψ_i) is a PCP $\forall i = 1 \ldots, l$.
2. $(A_i \bigcap B_i) = \varnothing$, $\forall i = 1 \ldots, l$.
3. For any two PCPs (A_i, B_i, Ψ_i), (A_j, B_j, Ψ_j), $1 \leq i < j \leq l$ in $G \otimes G$, the resulting decomposition has the property that $(A_i \bigcap A_j) \times (B_i \bigcap B_j) = \varnothing$. In other words, for any pair of vertices (u, v), there exists an unique PCP (A_i, B_i, Ψ_i) in $G \otimes G$, *s.t.* $u \in A_i, v \in B_i$.

The first property ensures that the decomposition of G results in a set containing l PCPs, where the i^{th} PCP in the decomposition is denoted by (A_i, B_i, Ψ_i). Furthermore, the heads A_i and B_i of a PCP are disjoint. The final property ensures that any pair of vertices (u, v) in G is contained in exactly one of the PCPs in the decomposition.

Algorithm 11 decomposes G into a set of PCPs. The algorithm takes a spatial network $G(V, E)$ and the root block T of a PR quadtree [144] on the spatial positions of V

as inputs. Q is a list containing pairs of blocks and is initialized with the pair (T,T). At each stage of the algorithm, a pair A,B is retrieved from Q and examined (line 4). Note that A and B correspond to blocks in the PR quadtree on V. If A and B do not refer to the same block (line 5), then the STEM-CUT Ψ of the pair is determined (line 6). An efficient method for computing the STEM-CUT of two sets is described in the Section 5.2.2. If Ψ is a vertex, we consider the following possible cases. If $\Psi \in A$ and $A = \{\Psi\}$, then B needs to be further decomposed. If $\Psi \in A$, then A needs to be further decomposed (line 8–14). Similarly, the symmetric cases with A and B interchanged are handled in lines 15–20 of the algorithm. If $\Psi = (u,v)$ is an edge, the only feasible case is when $u \notin B$ and $v \notin A$, which results in a valid PCP. We point out that the other possible cases when $u \in B$, or $v \in A$, or $u \in B$ and $v \in A$ result are infeasible, in which case both A and B (line 31) are each split into c child blocks and the resulting c^2 pairs are inserted into Q. The algorithm terminates when Q is empty at which point the decomposition of G is complete.

Algorithm 11

Procedure NETWORKDECOMPOSE[G, T]

Input: $G(V,E)$ is the input spatial network

Input: $T \leftarrow$ root node of spatial structure on spatial positions of V

Output: Decomposition of V into a set of PCP (A,B,Ψ)

```
(* $Q \leftarrow$ list of pair of blocks *)
1.  INIT: $Q$.insert($A$=$T$,$B$=$T$)
2.  END-INIT
3.  while not ($Q$.empty()) do
4.      $(A,B) \leftarrow Q$.pop() (* Extract top element *)
5.      if $A \neq B$ then
6.          Compute $\Psi \leftarrow$ STEM-CUT of $(A,B)$
7.          if $\Psi$ is a single vertex then
8.              if $\Psi \in A$ then
9.                  if $\{\Psi\} = A$ then
10.                     decompose $B$ (or $A$ as well)
11.                 else
12.                     decompose $A$ (or $B$ as well)
13.                 end-if
14.             end-if
15.             elseif $\Psi \in B$ then
16.                 if $\{\Psi\} = B$ then
17.                     decompose $A$ (or $B$ as well)
18.                 else
19.                     decompose $B$ (or $A$ as well)
```

```
20.                 end-if
21.             else
22.                 $(A, B, \Psi)$ is a valid PCP
23.             end-if
24.         end-if
25.         elseif the $\Psi$ is an edge $(u, v)$ then
26.             if $u \notin B$ and $v \notin A$ then
27.                 $(A, B, (u, v))$ is a valid PCP
28.             end-if
29.         end-if
30.     else
31.         Split $A, B$ into $c$ children each
32.         $Q$.insert($A_i, B_j$) $\forall i, j$ between 1 and $c$
33.     end-if
34. end-while
```

We briefly show that Algorithm 11 decomposes G into a set of PCPs that satisfies properties 1–3. First of all, we distinguish between pairs in Q that point to the same block in the quadtree and those that point to different blocks in the quadtree. We refer to them as SINGLETONS and PAIRS, respectively. The list Q is initialized with the SINGLETON (T, T) in line 1 at the start of the algorithm. During the course of the algorithm, if a SINGLETON (A,A) is retrieved at the top of Q, then it is replaced with c SINGLETONS and (c^2-c) PAIRS formed by the children nodes of A (or B). If a PAIR (A,B) is retrieved at the top of Q, then it is replaced with c^2 PAIRS formed by the children nodes of A and B. By induction, we can show that every block in the quadtree is eventually retrieved from Q as a SINGLETON. To show that the heads of the reported PCPs are disjoint (Property 2), we point out that only SINGLETONS have overlapping heads (owing to the fact that quadtrees are disjoint space decompositions), but only PAIRS are reported as PCPs. We now show that any vertex pair u,v is contained in one and only one PCP in the decomposition (Property 3). To do this, we use contradiction. Assume that u and v are contained in the two PCPs, say (A_i,B_i) and (A_j,B_j) *s.t.*, $i \neq j$, $u \in A_i, A_j$ and $v \in B_i, B_j$. Let M be the nearest common ancestor block of u,v in the quadtree. Before M is retrieved from the top of Q as a SINGLETON, u and v are contained in the same head. When M is decomposed, u and v are no longer contained in any SINGLETON, but present in different blocks, which may not yet be a PCP. After each subsequent decompositions, only one PAIR contains both u and v. Thus, the nature of our decomposition makes it impossible that u and v are contained in both (A_i,B_i) and (A_j,B_j) *s.t.*, $i \neq j$. Similarly, we can also claim that given a pair of vertices u,v, exactly one PCP in the decomposition contains it. Finally, as every vertex pair is contained in exactly one of the PCPs in the decomposition, we have also shown that the PCP decomposition of G encodes all the $O(n^2)$ shortest paths in G.

5.2.2 Computing Stem-Cut

Given that A,B are nodes in the PR quadtree on G such that $A \cap B = \varnothing$, we now present an algorithm that can compute the *stem-cut* of A and B efficiently using the SILC [154] representation of G, which we assume has been precomputed. Given a source vertex $s \in V$, the SILC map M_s of s is a decomposition of the embedding space S into a set of regions such that all the vertices contained in a region share the same first link in the shortest path from s. In other words, M_s is a set of pairs of the form (R_s^j, l_s^i), where R_s^j is a spatial region, and $l_s^i \in V$, $< s, l_s^i > \in E$, $1 \leq i \leq$ out-degree(s), such that any vertex v contained in the spatial region R_s^j satisfies the condition $l_s(u) = l_s^i$. We refer to the vertex l_s^i associated with a region R_s^j in M_s as the *color* of the region. The color of a destination vertex w is the *color* of the region in M_s that contains it. Similarly, the *color set* of a set of destination vertices, denoted by a spatial region R are the colors of regions in M_s that intersect R, while ensuring that the overlapping region between R and a region in M_s contains at least one vertex in R. A region R is said to be *single colored* with respect to s, iff the color set of R with respect to M_s contains an unique single color. We also define an *inverted SILC map* M'_w of a vertex w as a set of pairs of the form (R_w^j, l_w^i), where R_w^j is a spatial region, and $l_w^i \in V$, $< l_w^i, w > \in E$, $1 \leq i \leq$ in-degree(w), such that any vertex v contained in the spatial region R_s^j satisfies the condition $l'_u(w) = l_w^i$.

Lemma 5.1. *Given a source vertex s, the shortest path quadtree M_s of s, and a spatial region R, the shortest path between s and all destination vertices in R are non-disjoint,* *iff R is single colored with respect to s.*

Proof. The shortest paths from s to all vertices in R share the same first link in the shortest path and hence the shortest paths are non-disjoint. □

We state the next lemma without providing a proof.

Lemma 5.2. *Suppose A, B are disjoint blocks in the quadtree on G containing a subset of vertices in G. The shortest paths between all the source vertices in A to all the destination vertices in B are mutually non-disjoint, iff B is* single colored *with respect to all the source vertices in A and A is* single colored *with all the destination vertices in B.*

The following lemma describes the main result of this section.

Lemma 5.3. *Suppose A, B are disjoint blocks in the quadtree on G, containing a subset of vertices in G, that satisfy Lemma 5.2. A subpath that is common to the shortest paths from $s \in A$ to all the vertices in B, and shortest paths from all the vertices in A to a destination vertex $w \in B$, where s and w are chosen arbitrarily, is common to all the shortest paths between A and B.*

Proof. From Lemma 5.11. □

5.2.3 Linear Quadtree Representation of the Decomposition

The heads A, B of a PCP (A, B, w) correspond to nodes in the PR quadtree on G. Consequently, A and B can be represented as Morton codes [144], which is a compact bit-interleaved path compressed representation of a block in a quadtree. A PCP, in turn, is represented by concatenating the Morton codes of A and B. The resulting Morton code representation of PCPs serve as keys to records that are stored in a B-tree or B$^+$-tree. Such a representation, also known as a *linear quadtree* [58, 142], is useful as it provides a disk-efficient access to the PCPs in time logarithmic to the number of PCPs stored in the B-tree or B$^+$-tree. The set of vertices that make up the STEM-CUT of the PCP form the data that is associated with each of the records in the linear quadtree. A PCP in the linear quadtree is represented as (A, B, USEEDGE, t, HEAD, TAIL), such that A, B are represented as Morton codes and are concatenated to form the *keys* to the access structure, USEEDGE is a boolean flag that indicates if the PCP stores an intermediate vertex or an edge. If USEEDGE is set to *false*, then t stores an intermediate vertex, else $<$HEAD, TAIL$>$ stores an intermediate edge.

Given a pair of vertices (u, v), we access the STEM-CUT (i.e., an intermediate vertex or edge on the shortest path from u to v) by concatenating the bit-interleaved representation (i.e., Morton code) of the two vertices, say M_{uv}, and identify the Morton code b in the B-tree with the longest *prefix match* to M_{uv}. The data associated with b contains either an intermediate vertex or edge in the shortest path from u to v. An algorithm to retrieve the shortest paths using the PCP decomposition of G is discussed in the following section.

5.3 Finding Shortest Path

In this section, we present an algorithm to retrieve the shortest path between a pair of vertices $u, v \in V$. Algorithm 12 takes a source vertex u and a destination vertex v as inputs and retrieves the shortest path $\pi_G(u,v)$ between u and v in G. Let J be a B-tree on the PCP decomposition of G. In lines 1–3, we check if u and v are the same, in which case, the algorithm returns $\{u\}$. This is the *base case* of the recursive algorithm. In line 4, using the Morton representation of u and v as the search key, the B-tree J is searched for a PCP containing both u and v. Recall that the nature of the PCP decomposition of G guarantees that every pair of vertices in G is contained in some PCP in J. Let P $(A, B, \text{USEEDGE}, t, \text{HEAD}, \text{TAIL})$ be a PCP, such that A contains u and B contains v. If A and B share an edge (in which case USEEDGE is set to *true* as shown in lines 5–6), $\pi_G(u,v)$ is represented as a composition of $\text{SPATH}(u, \text{HEAD}) \rightsquigarrow <\text{HEAD}, \text{TAIL}> \rightsquigarrow \text{SPATH}(\text{TAIL}, v)$, resulting in subsequent recursive calls to Algorithm 12. If A and B share an intermediate vertex t (in which case USEEDGE is set to *false*), we recursively call two instances of Algorithm SPATH with inputs $\{u,t\}$ and $\{t,v\}$ as shown in line 8.

Algorithm 12

Procedure SPATH[u, v]

Input: u is the source vertex, and v is the destination

(∗ J is a B-tree on the PCP decomposition of G ∗)

Output: Shortest path $\pi_G(u,v)$ between u and v

```
1. if ($u = v$) then (* base case of the recursion *)
2.     return $\{u\}$
3. end-if
4. Search for a PCP $P(A, B,$ USEEDGE, $t$, HEAD, TAIL) in $J$ such that $u \in A$ and $v \in B$
5. if (USEEDGE = true) then
6.     return SPATH($u$, HEAD) $\rightsquigarrow <$ HEAD, TAIL $> \rightsquigarrow$ SPATH(TAIL, $v$)
7. else (* USEEDGE = false *)
8.     return SPATH($u$, $t$) $\rightsquigarrow$ SPATH($t$,$v$)
9. end-if
```

5.4 Analysis of Path Coherence

In this section, we provide bounds on the size of decomposition of G into PCPs by appealing to the equivalence between the PCP decomposition of a spatial network and the WSP decomposition of a point set. Given a point set S in a d-dimensional space, we construct a well separated (WSP) decomposition on S by first constructing a PR quadtree [124, 144] on S. For the sake of simplicity, we assume that S is contained in a unit $[0,1]^d$ d-dimensional hypercube. This hypercube forms the root block T of the PR quadtree on S. The hierarchical structure of the quadtree is constructed by recursively decomposing the block into 2^d congruent children blocks. The process continues until each block contains a single point, in which case, further subdivision is not possible. Unfortunately, if two points are close to one another in S, it may lead to a long path of trivial blocks of which only one block would form an internal node. Callahan and Kosaraju's construction [29] did not incur this problem because they used a fair-split tree which is a a data-dependent decomposition. This problem is remedied by Fischer and Har-Peled [46] through the use of a variant of a *path-compressed* quadtree which is obtained from the PR quadtree by compressing such trivial paths into one single compressed link. The advantage of the path compressed quadtree over the PR quadtree is that its use results in a tree with a total of $O(n)$ nodes.

Our discussion does not need to resort to the path-compressed quadtree while still using regular decomposition because of certain assumptions that we make about the distribution of the vertices in the embedding space. In particular, letting Δ be the ratio of the diameter of the set of vertices V to the distance between the closest pair of vertices in V and letting T be a PR quadtree on V, the maximum depth of T is $O(\log\Delta)$. Consequently, given a vertex v in V, the Morton code representation of $p(v)$, the spatial position of v, would be $O(\log\Delta)$ bits long. To cast this quantity in terms of n, we note that even if the data is heavily skewed so that Δ is linear in n, the length of the Morton code representation of v would still be $O(\log n)$. We claim that this assumption fits closely with the actual nature of real road networks. From a practical standpoint with respect to our experience with real data such as that found in road networks, we observe that the minimum geodesic distance between any two vertices on a road network is at least 1 meter. A PR quadtree on a sphere corresponding to the Earth with radius 6378 km and depth 24, has a 1 meter resolution at the equator. For such data, the size of the Morton code for a vertex on the road network using geographical coordinates is at most 48 bits in length.

The decomposition of S into WSPs is a *realization* on T, *i.e.*, subsets A_i, B_i of S forming a WSP (A_i, B_i) in $S \otimes S$ are pairs of nodes in T. The algorithm that decomposes S into WSPs using T and s (i.e., the separation factor) as inputs, proceeds in a similar manner to Algorithm 11. The algorithm uses a list Q which is initialized by the pair (T, T) corresponding to the root of the quadtree on S. At each iteration of the algorithm, a pair (A, B) of blocks in T is retrieved from Q. If (A, B) is $s-$separated, it is reported as a WSP. Otherwise, new pairs are obtained by replacing A and B with their 2^d children blocks, which are inserted into Q. The algorithm terminates when Q is empty.

Suppose that a pair (u, v) is reported as a well separated pair by the algorithm. This would indicate that $(P(u), P(v))$ is not well separated, where $P(b)$ is the parent block of b. Suppose further that the side length of $P(u)$ (or $P(v)$) is x and hence also its maximum possible diameter. The total number of blocks that are not well separated from $P(u)$ is bounded by the number of blocks of diameter x that are contained within a hypersphere of diameter $(2s+1)x$ centered at $P(u)$, which contains a maximum of $O(s^d)$ blocks. Recalling that T has $O(n)$ nodes means that the algorithm creates a maximum of $O(s^d n)$

WSPs. This result and the proof sketch is due to Callahan and Kosaraju [29] and we restate it below as Lemma 5.4, which is referenced in the subsequent discussion.

Lemma 5.4. *Given a point set S containing n d-dimensional points, a fixed separation factor $s \geq 2$, the WSP decomposition of S, $S \otimes S$ has $O(s^d n)$ WSPs [31][2].*

We now introduce the concept of well separated pairs on spatial networks. We assume that the ratio between the network and spatial distances is bounded both from above and below.

Assumption 1. $\gamma_1 \leq \frac{d_G(s,t)}{d_S(s,t)} \leq \gamma_2$, *$s,t \in V, \gamma_1$ and $\gamma_2 > 0$.*

At this point, we show how to extend the notion of a well separated pair decomposition in terms of a spatial distance to one in terms of a network distance. This is captured by Lemma 5.5 below.

Lemma 5.5. *Given a s-WSP decomposition of the vertices V of a spatial network $G(V,E)$ based on a spatial distance also yields an s'-WSP decomposition of V using a network distance with $s' = s \cdot \frac{\gamma_1}{\gamma_2}$.*

Proof. Given a s-WSP, (A,B) in the decomposition of $V \otimes V$ using the spatial distance measure, the minimum spatial distance between A and B is at least $s \cdot r$, where r is the larger of the diameters of A and B.

[2]The Lemma also holds for values of s, $0 < s < 2$, although s^d needs to be replaced with $\max(2, s^d)$

Consider u,v two vertices in A (or B), we have $d_G(u,v) \leq \gamma_2 \cdot d_S(u,v) \leq \gamma_2 \cdot r$ (because $d_S(u,v) \leq r$ by virtue of r being the diameter of A or B). r', the maximum value of $d_G(u,v)$, is the diameter of A (and B) using a network distance measure and we have that $r' \leq \gamma_2 \cdot r$. Therefore, the spatial distance diameter of A (or B) is scaled by at most a factor of γ_2 to obtain the network distance diameter r'.

Considering a vertex pair (a,b), such that $a \in A, b \in B$, we have from the well separated pair condition and Assumption 1 that:

$$s \cdot r \leq d_S(a,b) \leq \frac{d_G(a,b)}{\gamma_1} \tag{5.1}$$

Replacing r with $\frac{r'}{\gamma_2}$ in (5.1), we obtain $s \cdot \frac{r'}{\gamma_2} \leq d_S(a,b) \leq \frac{d_G(a,b)}{\gamma_1}$. The above relationship between the lower and upper bounds on $d_S(a,b)$ can be rewritten as $r' \cdot s \cdot \frac{\gamma_1}{\gamma_2} \leq d_G(a,b)$. Now, letting, $s' = s \cdot \frac{\gamma_1}{\gamma_2}$, leading to the desired result $s' \cdot r' \leq d_G(a,b)$ which is equivalent to saying that A and B are well separated using the network distance measure with a separation factor of s'. $\square$

At this point, we introduce another assumption on the nature of our spatial networks which enables us to ensure that there cannot be many disjoint shortest paths between subsets of sources and destinations in a spatial network. Spatial networks for which the number of such paths is bounded are said to be *path coherent*. The motivation is that the greater the number of disjoint shortest paths, the less likelihood that the shortest paths share a common vertex.

Assumption 2. *Letting $s,t \in V$ be a part of spatial network $G(V,E)$, then $\frac{1}{\delta}$ serves an upper bound on the ratio of $d_G(s,t)$, the network distance along the shortest path $P_1 = \pi_G(s,t)$ between s and t, and $d_{\{N \setminus \pi_G(s,t)\}}(s,t)$, the network distance along a disjoint shortest path in the spatial network $G(V - \pi_G(s,t), E - \rho(s,t))$ between the same two vertices when the vertices along the shortest path between them have been removed (i.e., $\pi_G(s,t)$). Formally, $\frac{d_G(s,t)}{d_{\{N \setminus \pi_G(s,t)\}}(s,t)} \leq \frac{1}{\delta}, (s,t) \in V, \delta > 1$*

In other words, given any pair of vertices $s,t \in V$ in G, there cannot exist two disjoint paths, $P_1 = \pi_G(s,t)$ and P_2 in G, such that, the ratio of the length of P_2 to P_1 is less than δ.

Definition 5. *Two cycle free paths $\pi_1(u_1,v_1)$ and $\pi_2(u_2,v_2)$ are* core-disjoint*, if and only if they share no common vertices besides the source or destination vertices. Formally $\pi_1(u_1,v_1)$ and $\pi_2(u_2,v_2)$ are core-disjoint if and only if the $\pi_1 \cap \pi_2 = (\{u_1\} \cap \{v_1\}) \cup (\{u_2\} \cap \{v_2\})$.*

Definition 6. *A pair of vertices (u,v) is* shortest path δ-redundant *with respect to $G(V,E)$ if and only if for any path π from u to v that is core-disjoint from $\pi_G(u,v)$, we have $w(\pi) \geq \delta w(\pi_G(u,v))$.*

Definition 7. *The graph $G(V,E)$ is* shortest path δ-redundant *if and only if every pair of vertices is shortest path δ-redundant with respect to $G(V,E)$.*

Lemma 5.6. *If the path $\pi_1(u_1,v_1)$ is core-disjoint from the two paths $\pi_2(u_1,s)$ and $\pi_3(s,v_2)$, then π_1 is core-disjoint from $\pi_2 \rightsquigarrow \pi_3$.*

Proof. As π_1 and π_2 are core-disjoint, we have, $\pi_1 \cap \pi_2 = (\{u_1\} \cap \{u_1\}) \cup (\{v_1\} \cap \{s\}) = \{u_1\} \cup (\{v_1\} \cap \{s\})$. As π_1 and π_3 are core-disjoint, we have, $\pi_1 \cap \pi_3 = (\{u_1\} \cap \{s\}) \cup (\{v_1\} \cap \{v_2\})$. We derive $P = \pi_1 \cap (\pi_2 \rightsquigarrow \pi_3)$,

$$\begin{aligned} P &= \pi_1 \cap (\pi_2 \cup \pi_3) && (5.2) \\ P &= (\pi_1 \cap \pi_2) \cup (\pi_1 \cap \pi_3) && (5.3) \\ P &= \{u_1\} \cup (\{v_1\} \cap \{s\}) \cup (\{u_1\} \cap \{s\}) \cup (\{v_1\} \cap \{v_2\}) && (5.4) \\ P &= \{u_1\} \cup (\{v_1\} \cap \{s\}) \cup (\{v_1\} \cap \{v_2\}) && (5.5) \end{aligned}$$

To show π_1 is core-disjoint from $\pi_2 \rightsquigarrow \pi_3$, it suffices to show that $P = (\{u_1\} \cap \{u_1\}) \cup (\{v_1\} \cap \{v_2\}) = \{u_1\} \cup (\{v_1\} \cap \{v_2\})$. We consider two cases: (i) $s \neq v_1$. $P = \{u_1\} \cup \varnothing \cup (\{v_1\} \cap \{v_2\})$

$P = \{u_1\} \cup (\{v_1\} \cap \{v_2\})$. (ii) $s = v_1$. $P = \{u_1\} \cup (\{v_1\}) \cup (\{v_1\} \cap \{v_2\})$. $P = \{u_1\} \cup (\{v_1\})$.

However, as π_1 is core disjoint from π_3, we have $\pi_1 \cap \pi_3 = (\{u_1\} \cap \{s\}) \cup (\{v_1\} \cap \{v_2\})$, $\pi_1 \cap \pi_3 = (\{u_1\} \cap \{v_1\}) \cup (\{v_1\} \cap \{v_2\})$, $\pi_1 \cap \pi_3 = \{v_1\} \cap (\{u_1\} \cup \{v_2\})$.

However, as v_1 belongs to both π_1 and π_3, $v_1 \in \pi_1 \cap \pi_3$. Hence, $v_1 \in (\{u_1\} \cup \{v_2\}$. That is, $v_1 = v_2$, or $v_1 = u_1$. We need to show that $P = \{u_1\} \cup (\{v_1\} \cap \{v_2\})$. In case $v_1 = v_2$, we need to show that $P = \{u_1\} \cup \{v_1\}$, which is already shown. In case, $v_1 \neq v_2$ and $v_1 = u_1$, we need to show that $P = \{u_1\}$. But we already know that $P = \{u_1\} \cup (\{v_1\}) = \{u_1\}$. □

Lemma 5.7. *Consider a shortest path δ-redundant spatial network $G(V,E)$, $A \subset V$, and $B \subset V$, such that (A,B) is a well separated pair with separation factor $s > 2 + \frac{2}{\delta - 1}$. For $u_1, u_2 \in A$ and $v_1, v_2 \in B$, the shortest paths $\pi_1 = \pi_G(u_1, v_1)$ and $\pi_2 = \pi_G(u_2, v_2)$ are not disjoint.*

Proof. Assume to the contrary that the paths π_1 and π_2 are disjoint. Figure 5.3 shows an example of such a scenario. Let $d_1 = w(\pi_1)$ and $d_2 = w(\pi_2)$. Without loss of generality, we assume that $d_2 \leq d_1$.

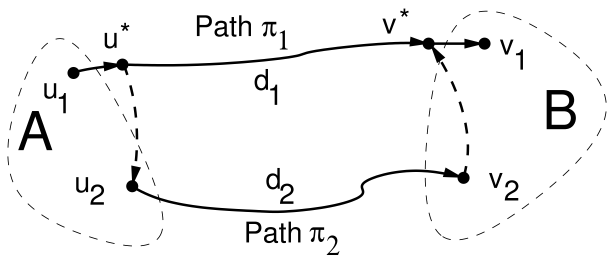


Figure 5.3: (A,B) is a s-WSP configuration containing two disjoint paths between them

Consider the path $\pi_G(u_1, u_2)$. Let u^* be the last vertex in $\pi_G(u_1, u_2)$ that is common with $\pi_G(u_1, v_1)$. Similarly, let v^* be the first vertex in $\pi_G(v_2, v_1)$ that is common with $\pi_G(u_1, v_1)$. Notice that $\pi_G(u^*, v^*)$ is a subpath of π_1 and is hence disjoint from π_2 because of our assumption. The definition of u^* and v^* ensure that $\pi_G(u^*, u_2)$ and $\pi_G(v_2, v^*)$ are both core-disjoint from $\pi_G(u^*, v^*)$. Let $d^* = d_G(u^*, v^*)$.

Consider the path $\pi_3 = \pi_G(u^*, u_2) \rightsquigarrow \pi_2$. $\pi_G(u^*, v^*)$ is core-disjoint from π_3 due to Lemma 5.6. Consider the path $\pi^* = \pi_3 \rightsquigarrow \pi_G(v_2, v^*)$ between u^* and v^*. By re-applying

Lemma 5.6, we claim that $\pi_G(u^*, v^*)$ is core-disjoint from π^*.

For r_G, the larger diameter of A and B, we have, $d_G(u_1, u_2) \leq r_G$, and $d_G(v_2, v_1) \leq r_G$. Therefore, $d_G(u_1, u_2) + d_G(v_2, v_1) \leq 2r_G$.

We have,

$$\begin{aligned} d_1 &= d_G(u_1, u^*) + d^* + d_G(v^*, v_1) \qquad (5.6) \\ w(\pi^*) &= d_G(u^*, u_2) + d_2 + d_G(v_2, v^*) \end{aligned}$$

Hence,

$$\begin{aligned} w(\pi^*) + d_1 &= d_G(u_1, u^*) + d_G(u^*, u_2) + d^* + \\ &\quad d_2 + d_G(v_2, v^*) + d_G(v^*, v_1) \\ w(\pi^*) + d_1 &= d_G(u_1, u_2) + d_G(v_2, v_1) + d^* + d_2 \\ w(\pi^*) + d_1 &\leq 2r_G + d^* + d_2 \qquad (5.7) \end{aligned}$$

By assumption of δ-redundancy, we have $w(\pi^*) \geq \delta d^*$, hence combining with Equation 5.7, we get, $\delta d^* + d_1 \leq 2r_G + d^* + d_2$, and as $d_2 \leq d_1$ we get $\delta d^* + d_1 \leq 2r_G + d^* + d_1$, or,

$$(\delta - 1)d^* \leq 2r_G \qquad (5.8)$$

As (A, B) is a s-WSP, the network distance between a pair of points in A and B is at least $s \cdot r_G$, hence $s \cdot r_G \leq d_1$, combined with Equation 5.6 we get, $s \cdot r_G \leq d_G(u_1, u^*) + d^* + d_G(v^*, v_1)$. But as $d_G(u_1, u^*) \leq d_G(u_1, u_2) \leq r_G$ and $d_G(v^*, v_1) \leq d_G(v_2, v_1) \leq r_G$, we get $s \cdot r_G \leq r_G + d^* + r_G$, or

$$(s-2)r_G \leq d^* \tag{5.9}$$

By combining Equations 5.8 and 5.9, we get $(\delta-1)(s-2)r_G \leq 2r_G$, but $s > \frac{2}{\delta-1} + 2$ and hence by substituting s, we get $2 < 2$, which is a false statement. This contradicts our initial assumption that π_1 and π_2 are disjoint. □

We state the following two properties of shortest paths without providing a proof.

Lemma 5.8. *If $t \in a \rightsquigarrow b$ and $r \in a \rightsquigarrow t$, then $t \in r \rightsquigarrow b$.*

Lemma 5.9. *If $r \in a \rightsquigarrow t$ and $t \in a \rightsquigarrow r$ then $t = r$.*

5.4.1 Four Connected Path (4-CP) Problem

Lemma 5.10. *Consider two source vertices u_1, u_2 and two destination vertices v_1, v_2 in a δ-redundant spatial network $G(V,E)$, such that there exist shortest paths between the source and the destination pairs, namely, $\pi_1 = u_1 \rightsquigarrow v_1$, $\pi_2 = u_1 \rightsquigarrow v_2$, $\pi_3 = u_2 \rightsquigarrow v_1$, and $\pi_4 = u_2 \rightsquigarrow v_2$. Given that no two shortest paths are disjoint, there exists a vertex t that is common to all the four shortest paths.*

Proof. Figure 5.4a shows the shortest paths $\pi_1 = u_1 \rightsquigarrow v_1$ and $\pi_2 = u_1 \rightsquigarrow v_2$, such that t is the outgoing vertex of π_1 (π_2) with respect to π_2 (π_1). The vertex t can be trivially shown to exist, as $t = u_1$ is at least one vertex that is always common to π_1 and π_2. We refer to $\pi_G(u_1, t)$ as the *stem* of the shortest paths.

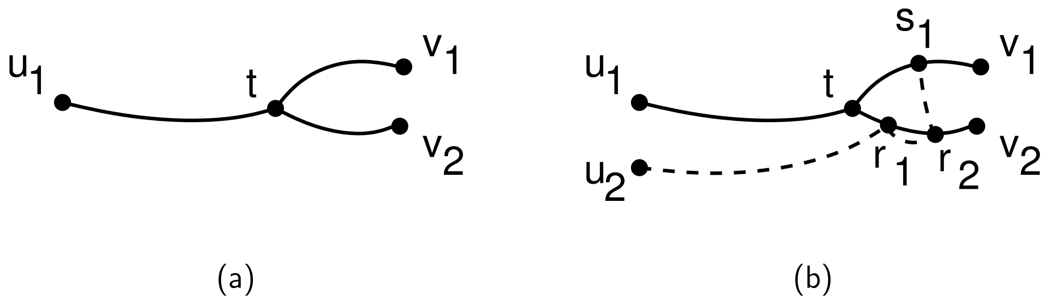


Figure 5.4: (a) Shortest paths π_1 and π_2, where t the outgoing vertex of π_1 (π_2) with respect to π_2 (π_1), (b) Shortest path π_3 has been added to the setup in (a), such that $r_1 \in t \rightsquigarrow v_2$, $r_2 \in t \rightsquigarrow v_2$, $s_1 \in t \rightsquigarrow v_1$, $v_1 \notin t$, $v_2 \notin t$, and $s_1 \notin t$

We now add the $\pi_3 = u_2 \rightsquigarrow v_1$ to the setup in Figure 5.4a, while ensuring that that π_3 is not disjoint from either π_1, or π_2. Let r_1 be the incoming vertex of π_3 with respect to π_2. Let r_2 be the outgoing vertex of π_3 with respect to π_2. Let s_1 be the incoming vertex of π_3 with respect to π_1. Note that π_3 cannot have an outgoing vertex with respect to π_1, as $s_1 \rightsquigarrow v_1$ is a common subpath to both $u_1 \rightsquigarrow v_1$ and $u_2 \rightsquigarrow v_1$.

We point out that there can only be three possible configurations of r_1, r_2 and s_1 with respect to t, u_1, v_1 and v_2, which are listed below.

1. If $r_1 \neq t, r_1 \in t \rightsquigarrow v_2$, $r_2 \in u_1 \rightsquigarrow v_2$, and $s_1 = t$, it can be verified that path π_4 cannot exist now without being disjoint from either π_1 or π_2. Hence, this case is infeasible.

2. If $r_1 \neq t$, $r_2 \neq t$, $r_1, r_2 \in t \rightsquigarrow v_2$, and $s_1 \in t \rightsquigarrow v_1$, $s_1 \neq t$, we show that π_4 can exist only iff $\delta = 1$, and hence is infeasible.

3. If $r_1, r_2, s_1 \in u_1 \rightsquigarrow t$, we show that π_4 can exist only iff $r_1 = r_2 = s_1$, thus proving the lemma.

Figure 5.4b illustrates case-2, which we show to be infeasible. We have added the shortest path $\pi_4 = u_2 \rightsquigarrow v_2$ to the configuration in Figure 5.4b, resulting in Figure 5.5.

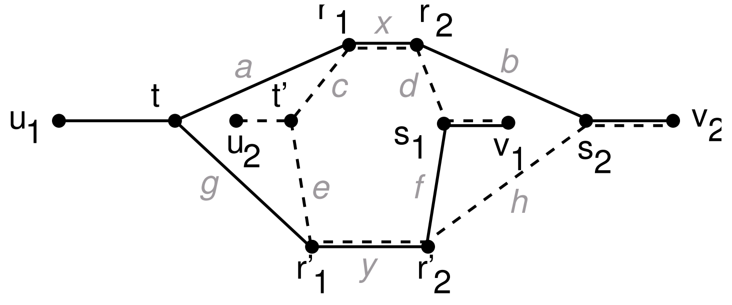


Figure 5.5: Figure 5.4b has been redrawn to include π_4, such that $r'_2 \in t \rightsquigarrow s_1$ and $s_2 \notin r_1 \rightsquigarrow r_2, s_2 \in r_1 \rightsquigarrow v_2$. Note that labels a–h, x, y on an edge indicates its weight. $r'_1 \rightsquigarrow r'_2$ is the subpath shared between $t \rightsquigarrow s_2$ and $t' \rightsquigarrow s_1$. Similarly, $r_1 \rightsquigarrow r_2$ is shared between $t' \rightsquigarrow s_2$ and $t \rightsquigarrow s_1$

Let r'_1 and r'_2 be incoming and outgoing vertices of π_4 with respect to π_1. While ensuring that π_4 is not disjoint with π_1, we observe that the only feasible condition is when $r'_1 \in t \rightsquigarrow s_1$ and $r'_2 \in t \rightsquigarrow s_1$ is satisfied. Let s_2 be the incoming vertex of π_4 with respect to π_2, and also $s_2 \notin r_1 \rightsquigarrow r_2, s_2 \in r_1 \rightsquigarrow v_2$. Figure 5.5 shows the shortest path configuration containing the four shortest paths. Also, the labels (a–h, x, y) assigned to edges in Figure 5.5 correspond to the weights of the edges. We will now show that this configuration is realizable, iff $\delta = 1$.

In Figure 5.5, we observe that $t \rightsquigarrow s_1$ is at a distance $g + y + f$, while an alternate core-disjoint path $t \rightsquigarrow r_1 \rightsquigarrow r_2 \rightsquigarrow s_1$ is at a distance $a+x+d$, leading to

$$a+x+d \geq \delta(g+y+f) \tag{5.10}$$

Similarly, $t \rightsquigarrow s_2$ is at a distance $a+x+b$, while an alternate core-disjoint path $t \rightsquigarrow r'_1 \rightsquigarrow r'_2 \rightsquigarrow s_2$ is at a distance $g+y+h$, leading to

$$g+y+h \geq \delta(a+x+b) \tag{5.11}$$

$t' \rightsquigarrow s_1$ is at a distance $c+x+d$, while an alternate core-disjoint path $t' \rightsquigarrow r'_1 \rightsquigarrow r'_2 \rightsquigarrow s_2$ is at a distance $e+y+f$, leading to

$$e+y+f \geq \delta(c+x+d) \tag{5.12}$$

$t' \rightsquigarrow s_2$ is at a distance $c+x+b$, while an alternate core-disjoint path $t' \rightsquigarrow r_1 \rightsquigarrow r_2 \rightsquigarrow s_2$ is at a distance $e+y+h$, leading to

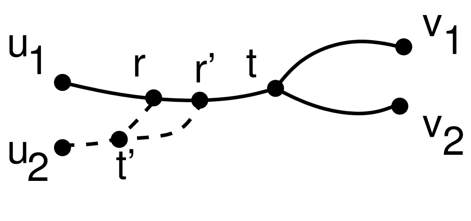
$$e+y+h \geq \delta(c+x+b) \tag{5.13}$$

Now adding the inequalities in Equations 5.10–5.13, we get

$$(a+b+c+d+e+f+2\cdot x+2\cdot y) \geq \delta(a+b+c+d+e+f+2\cdot x+2\cdot y)$$

As, $(a+b+c+d+e+f+2\cdot x+2\cdot y) > 0$, we get $\delta = 1$, which contradicts our assumption that $\delta > 1$. Hence, case-2 is infeasible.

We now examine case-3, when $r_1, r_2, s_1 \in u_1 \rightsquigarrow t$. First of all, we can trivially show that $r_2 = t$, and $s_1 = r_1$. Let r'_1 and r'_2 be incoming and outgoing vertices of π_4 with



(a) (b)

Figure 5.6: a) $r, r' \in u_1 \rightsquigarrow t$, such that $r = r_1 = s_1$, $r' = r'_1 = s_2$, and t' is the outgoing vertex of π_3 (π_4) with respect to π_4 (π_3), b) the only feasible configuration is when $r = r'$ and $t = t'$

respect to π_1. Let s_2 be the incoming vertex of π_4 with respect to π_2. Upon adding π_4, we can further show that that $r'_1 = s_2$ and $r'_2 = t$. Let $r' = r'_1 = s_2$, $r = r_1 = s_1$, and t' is the outgoing vertex of π_3 (π_4) with respect to π_4 (π_3). The resulting configuration is shown in Figure 5.6a.

We can now further claim that $r = r'$, failing which there would be two shortest paths from u_2 to r' or alternately, from u_2 to r, if $r' \in u_1 \rightsquigarrow r$. Hence, the only feasible configuration is shown in Figure 5.6b, where r is the incoming vertex of π_3 and π_4 with respect to π_1 and π_2, and $t = t'$ is the common vertex to the four shortest paths. □

5.4.2 All Connected Pair (all-CP) Problem

Lemma 5.11. *Suppose A is a set of source vertices and B is a set of destination vertices. If any pair of source vertices $u_1, u_2 \in A$ and destination vertices $v_1, v_2 \in B$ satisfy the* 4-*CP condition, then all the shortest paths from A to B, pass through one single common vertex.*

Proof. We will adopt the following strategy in proving this lemma. Our initial configuration consists of the shortest paths between two source vertices u_1 and u_2 in A, and destination vertices v_1 and v_2 in B. We would then add one additional destination vertex from B to this arrangement, in no particular order, until all the shortest paths from u_1 and u_2 to all vertices in B have been accounted for. We then add one additional source vertex from A to the arrangement, until all the possible shortest paths from A to B have been account for.

Suppose u_1, $u_2 \in A$ are source vertices and $v_1, v_2 \in B$ are destination vertices satisfying the 4-CP condition, as seen in Figure 5.4b. The four possible shortest paths are $\pi_1 = \pi_G(u_1, v_1)$, $\pi_2 = \pi_G(u_1, v_2)$, $\pi_3 = \pi_G(u_2, v_1)$, and $\pi_4 = \pi_G(u_2, v_2)$. Let t be the common vertex to the four shortest paths and corresponds to the outgoing vertex of π_1 with respect to π_2 (from 4-CP condition). Also, r is the incoming vertex of π_1 and π_3 with respect to π_2 and π_4.

We claim that the addition of an additional vertex $v_3 \in B$, may potentially replace t with another vertex t', such that $t' \in r \rightsquigarrow t$. Thus, after all vertices $v \in B$ have been added, $r \in u_1 \rightsquigarrow t$ would still be in 4-CP. With the addition of all the destination vertices in B, we would have accounted for the shortest paths from u_1 and u_2 to all destination vertices in B.

Let v_3 be a destination vertex in B. Let $\pi_5 = \pi_G(u_1, v_3)$, and $\pi_6 = \pi_G(u_2, v_3)$ be the shortest paths from u_1 and u_2 to v_3. Let t_1 be the outgoing vertex of π_5 with respect to

π_1 and π_3. Let t_2 be the outgoing vertex of π_6 with respect to π_2 and π_4.

In the three cases below, the addition of v_3 does not affect t.

- If $t_1 \in t \rightsquigarrow v_1$ and $t_1 \neq t$, then $t_2 = t$.
- If $t_2 \in t \rightsquigarrow v_2$ and $t_2 \neq t$, then $t_1 = t$.
- $t_1 = t_2 = t$.

If $t_1, t_2 \in r \rightsquigarrow t$, then t_1 must be equal to t_2. Let $t' = t_1 = t_2$. We replace t with t', v_1 (or v_2) with v_3. This resulting configuration would still resemble Figure 5.4b. We would then proceed with the insertion of another destination vertex in B to the new setup consisting of sources u_1 and u_2 in A and destination vertices v_1 and v_3 satisfying the 4-CP condition.

The final case is when t_1 is the outgoing vertex of π_5 with respect to π_1 and π_2, and let r_1, r_2 be the incoming and outgoing vertex of π_5 with respect to π_3 and π_4. Similarly, let r_1', r_2' be the incoming and outgoing vertex of π_6 with respect to π_1 and π_2, and let t_2 be the outgoing vertex of π_6 with respect to π_3 and π_4. The resulting configuration resembles Figure 5.5, and does not satisfy 4-CP. Hence, it is infeasible.

We have shown that an addition of a destination vertex v_3, either does not affect t, in which case, it can be ignored, or replaces t with $t' \in r \rightsquigarrow t$. After all the destination vertices have been added, r would still satisfy $r \in u_1 \rightsquigarrow t$ and $r \in u_2 \rightsquigarrow t$.

Adding the a source vertices u_3 in A to the setup in Figure 5.4b is symmetric to adding a destination vertex v_3, although the insertion of u_3 may affect r instead of t. In effect, after all the source vertices in A have been accounted for, all the shortest paths from A to B pass through t. $\square$

An immediate result of Lemma 5.7 and Lemma 5.11 is that for the separation factor $s > 2 + \frac{2}{\delta - 1}$, the WSP decomposition is, in fact, a PCP decomposition. That is – the shortest paths between all sources in A to all destinations in B in a WSP (A, B) pass through a single common vertex or an edge. We now show that given such a decomposition, the shortest path between any vertex pair can be retrieved in $O(k \log n)$ time, where k is the length of the shortest path.

Theorem 5.12. *Given a PCP decomposition of a spatial network $G(V,E)$ of size $O(s^d n)$, the shortest path between any vertex pair in V can be retrieved from the decomposition in $O(k \log n)$ time, where k is the length of the shortest path.*

Proof. Given a PCP (A,B,w), $A,B \subset V$, $w \in V$, or $w \in E$ in the decomposition of a spatial network of size l such that A,B are nodes in the PR quadtree on the spatial positions of V. The pair A,B is represented as a Morton code. which is then stored in a B-tree. The resulting structure L containing the l PCP pairs is termed a linear quadtree [144]. Given a source $s \in V$ and a destination $t \in V$, the PCP containing s and t is retrieved by invoking a binary search on the L, which takes $O(\log l)$ time. The entire shortest path can be obtained recursively in $O(k \log l)$ time, where $k = \pi_G(s,t)$. As $l = O(s^d n)$, the shortest path between any pair of vertices in G can be retrieved in

$O(k \log n)$ time. $\square$

Note that in the above analysis of the PCP decomposition of a spatial network G, the value of the separation factor s is dependent on the δ value of G. Spatial networks that have δ values close to 1 require s to be large. An example of such a spatial network is a regular grid (e.g., roads in Manhattan). We point out that this does not mean that our method cannot be used on spatial networks that have subgraphs resembling a regular grid. We now show that a large number of shortest paths in a spatial network can have δ values equal or close to 1, without affecting the linear bound on the storage. Furthermore, we experimentally validate the above claim by applying our technique on a road network dataset of Manhattan, NY which is discussed in Section 5.5.

Lemma 5.13. *Given a spatial network G which is a regular grid, the size of the PCP decomposition of G is $O(n\sqrt{n})$.*

Proof. Let G be a spatial network which is a regular grid containing n vertices as shown in Figure 5.7. Let A, B, C and D be the blocks resulting from the partition of the embedding space spanned by G into 4 congruent blocks. All the shortest paths between vertices in A and D, B and C pass through the common vertex w and is recorded using two Morton blocks. However, the shortest paths between the pairs (A,A), (A,B), (A,C), (B,B), (B,D), (C,C), (C,D), and (D,D) would still have to be recorded. Each of the above eight pairs is a smaller instance of the original problem (one-fourth the size), and

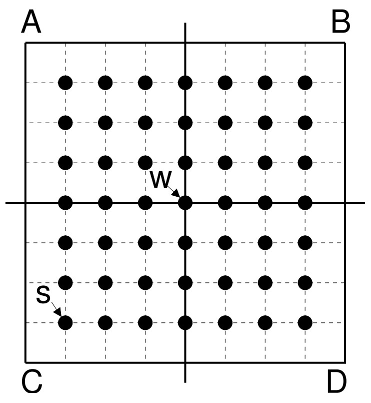


Figure 5.7: Figure shows a spatial network G which is a regular grid. Let s and w be vertices in G. Let A, B, C and D be the blocks resulting from the partition of the embedding space spanned by G into four congruent blocks. Notice that w is an intermediate vertex in at least one of the shortest paths from a source vertex in A, B, C, D to a destination vertex in D, C, B, A, respectively

hence the total storage of the PCP decomposition of G in terms of Morton blocks can be represented by the following recurrence relation:

$$T(n) = 8T(\frac{n}{4}) + 2 \tag{5.14}$$

Solving Equation 5.14, we can show that the shortest paths between all pair of vertices in G can be represented using $n\sqrt{n} + n$ Morton blocks. □

Lemma 5.14. *Given a PCP decomposition of size $O(s^d n)$ of a spatial network $G(V,E)$, the δ values of $O(\frac{ns^d}{hc^2})$ shortest paths in G can now be made equal to 1 without affecting the $O(s^d n)$ bound on the size of the decomposition.*

Proof. Given a PCP decomposition of size $O(s^d n)$ of a spatial network G, we first count the number of extra PCPs that are created as a result of adding a path that is disjoint to $\pi_G(u,v)$ of length $d_G(u,v)$ between two vertices $u, v \in V$, and thus the δ value of the vertex pair δ_{uv} is 1. For the sake of simplicity, we assume that the addition of an disjoint path only affects the δ value of the shortest paths in G that pass through both u and v. Let us suppose that the pair of vertices (u,v) is contained in the PCP (A,B,w), such that $u \in A$, $v \in B$ and $w \in V$, or $w \in E$. As δ_{uv} is $1, A, B$ may no longer be a PCP *i.e.*, A and B may not have a single common intermediate vertex. Consequently, we split A and B into c children each. Only one of the children of A, say A_i contains u. Similarly, only one of the children of B, say B_j contains v. All the c^2 pairs, with the exception of the pair (A_i,B_j), are PCPs as (A,B) was initially a PCP before δ_{uv} became 1. The pair (A_i,B_j) is still not a PCP as it still contains (u,v) and needs to be split further. Such an operation needs to be performed $O(h)$ times until u and v are both leaf nodes, where h is the depth of the PR quadtree on the spatial positions of V. The number of extra PCPs created as a result of making $\delta_{uv} = 1$ is $O(hc^2)$. Thus, potentially $O(\frac{ns^d}{hc^2})$ shortest paths can have $\delta = 1$, without affecting the linear bounds on the size of decomposition. $\square$

5.4.3 Bounds on the SILC Decomposition

We now show that there exists a $O(s^d nh)$ decomposition of V into WSPs of the form $(\{a\},B)$, where a is a vertex, B is a subset of V and h is the height of the PR quadtree on the spatial positions of V. We refer to a WSP of the form $(\{a\},B)$ as *one-to-many (OM)* WSP. We now describe a simple technique for *transforming* a WSP decomposition on V into a OM-WSP decomposition on V. Given a $O(s^d n)$ decomposition of V into WSPs of the form $(A,B), A,B \subset V$, we further decompose each WSP (A,B) into OM-WSPs of the form $(\{a\},B)$ *s.t.*,, each vertex $a \in A$ forms a OM-WSP of the form $(\{a\},B)$. We now show that the total number of OM-WSPs generated by the above transformation is $O(s^d nh)$.

Lemma 5.15. *A WSP decomposition of size $O(s^d n)$ of a set of points S of the form (X,Y), $X,Y \subset S$, can be further decomposed into $O(s^d nh)$ OM-WSPs of form $(\{x\},Y)$, where $x \in X$.*

Proof. Let T be a PR quadtree of height h on the spatial positions of S. Let i be a node in T containing s_i points which is paired with a_i other nodes in T during the WSP decomposition of S, as a result of which $s_i \cdot a_i$ OM-WSPs are created. The total number of OM-WSPs generated by $p = O(n)$ nodes in T is given by $\sum_{i=1}^{p} a_i s_i$, which is less than $s^d \sum_{i=1}^{p} s_i$ as $O(s^d)$ upper-bounds a_i. In a quadtree of height h, we know that $\sum_{i=1}^{p} s_i = O(nh)$. Substituting the above result in $s^d \sum_{i=1}^{p} s_i$, we obtain that the total number of OM-WSPs created by the algorithm is $O(s^d nh)$. □

We now discuss an alternate path encoding of a spatial network using PCPs of the form $(\{u\},B,w)$, such that u is a vertex, B is subset of vertices and w is a vertex or an edge that is common to the shortest paths from u to vertices in B. We know from Lemma 5.15, that a WSP decomposition of size $O(s^d n)$ can be further decomposed into a OM-WSP decomposition of size $O(s^d nh)$. Similarly, a PCP decomposition (which is a WSP decomposition of V of a suitable s) of size $O(s^d n)$ can be further decomposed into

$O(s^d nh)$ PCPs of the form $(\{u\}, B, w)$. We refer to PCPs of the form $(\{u\}, B, w)$ as *OM-PCPs*. Note that we assume that the height h of a PR quadtree on V is upper-bounded by $O(\log n)$. Hence, the size of the OM-PCP decomposition of V is $O(s^d n \log n)$. Furthermore, OM-PCPs of the form $(\{u\}, B, w)$ are replaced by $(\{u\}, B, l_u(w))$, that is, w is replaced with the next vertex after u in the shortest path from u to w. The resulting decomposition is termed SILC [154] and leads to our main result of this section.

Theorem 5.16. *Under certain conditions, the size of a OM-PCP decomposition (also known as the SILC decomposition of G) of a spatial network $G(V,E)$ is $O(s^d n \log n)$.*

We now show that SILC is able to retrieve the next link of a shortest path faster than the WSP technique, although at the expense of a slightly larger storage.

Theorem 5.17. *Given a OM-PCP decomposition of a spatial network $G(V,E)$ of size $O(s^d n \log n)$, the shortest path between any source and destination vertices in G can be retrieved in $O(k \log n)$ time, where k is the length of the shortest path. However, the average time to retrieve the next link in the shortest path is $O(\log \log n)$.*

Proof. We first briefly describe an efficient access structure on the OM-PCP decomposition of G. For each vertex $u \in V$, let E_u be the number of OM-PCPs of the form $(\{u\}, B_{u_i}, w_i)$, $1 \geq i \geq E_u$ in the decomposition. Note that B_{u_i} is represented as a Morton block. Each vertex u is associated with a sorted list (or B-tree) L_u of Morton blocks, which is populated with E_u Morton blocks from the decomposition.

Given a source vertex s and a destination vertex t, let k be the length of the shortest path between them. To retrieve the next link after s in the shortest path $\pi_G(s,t)$, we first obtain the sorted list of Morton blocks L_s associated with s. Let M_t be the Morton block representation of t. A simple binary search on L_s for a Morton block containing M_t is expected to take $O(\log E_u)$ time. The entire shortest path would require k such lookups. The total time T_k taken to obtain the entire shortest path is $\sum_{\forall v \in \pi_G(s,t)} \log E_v$. Rearranging the above formula we get, $T_k = \log \Pi_{\forall v \in \pi_G(s,t)} E_v$. To maximize the value of T_k, we set all the E_v to an equal value. From Lemma 5.15, we know that $\sum_{i=0}^{n} E_i = nh$, setting E_v to be $\frac{nh}{k}$, we get $T_k = \log(\frac{nh}{k})^k$, which simplifies to $k(\log n - \max(0, \log \frac{k}{h}))$. As h is $\log n$, this reduces to $k(\log n - \max(0, \log \frac{k}{\log n}))$.

Let T_a be the average time taken to retrieve the next link of a shortest path. We formulate T_a as follows; $T_a = \frac{1}{n} \sum_{i=1}^{n} \log E_i = \frac{1}{n} \log \Pi_{i=1}^{n} E_i$. The maximum value of T_a is obtained, if all the E_i are set to $\frac{nh}{n}$. Substituting this in the above equation, we obtain $T_a = \frac{1}{n} \log(\frac{nh}{n})^n$, which reduces to $\log h$. As h is $\log n$, we obtain $T_a = O(\log \log n)$. $\square$

Lemma 5.18. *Given a spatial network G which is a regular grid, the size of the SILC decomposition of G is $n \log n$.*

Proof. Let s be any vertex in the spatial network G which is a regular grid containing n vertices, as shown in Figure 5.7. Notice that the shortest path from s to all destination vertices in A, B and D are reached by taking a common first link from s. At this point, only the destination vertices in C need to be accounted as the shortest paths from s to

A, B and D can be recorded using 3 Morton blocks. Thus, the problem now reduces to a smaller instance of the original problem, that is, we now need to record the shortest path from s to all $\frac{n}{4}$ destination vertices in C. Hence, the total storage in terms of Morton blocks can be represented as a recurrence relation as follows:

$$T(n) = T(\frac{n}{4}) + 3 \tag{5.15}$$

Solving Equation 5.15, the shortest path from s to n vertices in G is recorded using $\log n$ Morton blocks. As, G has n vertices, the total size of the SILC encoding of G is $n \log n$. $\square$

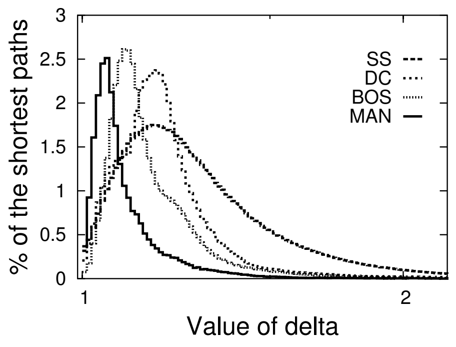
5.5 Experimental Results

In this section, we evaluate the performance of our technique that uses the PCP decomposition of a spatial network, and compare it with the SILC technique [154] in Chapter 2, which is the only other technique that directly competes with our work. We evaluate the relative merits of the WSP and the SILC techniques by comparing the size of the resulting path encoding of a spatial network and the time taken to retrieve the next link in the shortest paths. Furthermore, in this section we verify some of the results derived in Section 5.4 and, in particular, show that the experimental results closely follow the theoretical bounds on the size of both the path encoding of the PCP and the SILC techniques.

The experiments were carried out on a Linux (2.4.2 kernel) quad 2.4 GHz Xeon

Dataset	Vertices	height	Size		s_{eff}	
	n	h	WSP	SILC	WSP (S_w)	SILC (S_s)
Silver Spring (SS)	4233	11	277918	543191	8.1	3.4
Washington (DC)	12304	12	1695600	3414676	11.7	4.8
Boston (BOS)	17397	12	1605074	3298173	9.6	4.0
Manhattan (MAN)	39604	13	5932839	12391544	12.2	5.1

(a)



(b)

Figure 5.8: Different urban datasets used in evaluation of the path encoding are recorded in (a). The variations in the number of shortest paths for different values of $\delta \geq 1$ presented in (b).

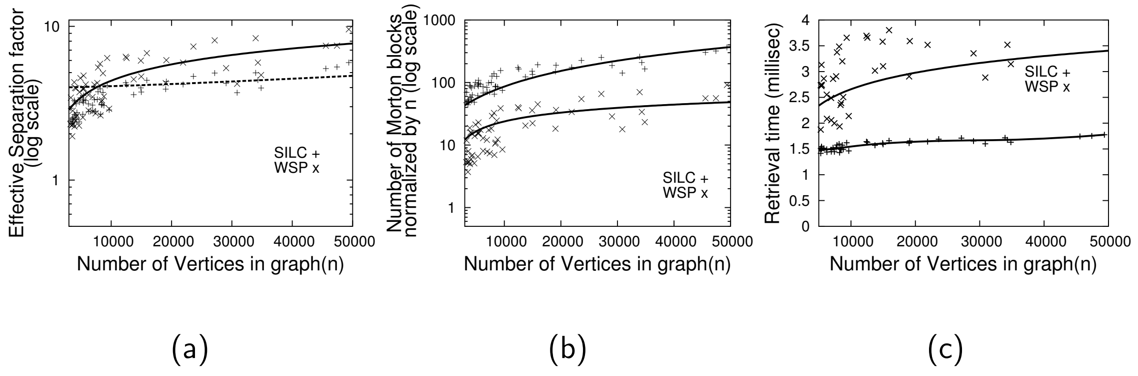


Figure 5.9: (a) The effective separation factor s_{eff} of the path encoding of the road network using SILC and WSP techniques. (b) Number of Morton blocks in the path encoding normalized by n, and (c) time in milli-seconds to retrieve a Morton block, in the path encoding of a road network containing n vertices using SILC and WSP techniques.

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 [183] and the National Atlas [184] websites. Some of the datasets that we used are described in Figure 5.8. 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. 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 variations in the number of blocks in the resulting PR quadtree on the spatial positions of the vertices.

We first examine the effect of δ on the size of the path encoding of road networks. A number of urban datasets were chosen in the evaluation and are shown in Figure 5.8. Some of the datasets, especially the road networks of Manhattan (MAN) and Washington, DC (DC) have large number of vertices that lie on a regular grid. We remind the reader that the δ value of a vertex pair that lie on a grid is close to 1. Figure 5.8b is the plot of the variation in the number of shortest paths (as a percentage of the total number of shortest paths) for different values of $\delta \geq 1$. The road network datasets of MON has a larger percentage of shortest paths whose δ values are closer to 1 than the Silver Spring (SS), Boston (BOS) and the Washington,DC (DC) datasets. Figure 5.8a records the number of vertices n, the height h of the PR quadtree on the spatial position of the n vertices, the resulting size of the WSP decomposition S_w, and the SILC S_s decomposition of the dataset in terms of the number of Morton blocks in the path encoding. Furthermore, we define the term, the *effective separation* factor, s_{eff} of a road network. The effective separation s_{eff} of a WSP decomposition of a dataset is the square-root of the ratio of the size of WSP to the number of vertices, *i.e.*, $s_{\text{eff}} = \sqrt{\frac{S_w}{n}}$. Similarly, the effective separation s_{eff} for SILC is the square root of the ratio of the size of the SILC path encoding to the product of n and h, *i.e.*, $s_{\text{eff}} = \sqrt{\frac{S_s}{n \cdot h}}$. Figure 5.8a records the s_{eff} values for both the SILC and WSP path encoding.

Figure 5.9a shows the variation of the effective separation factor s_{eff} for several road networks datasets of various sizes. It is seen that the value of s_{eff} for both the WSP and the SILC techniques, for the various datasets, are almost of a constant of about 8 and 4, respectively. When we compare this to the s_{eff} values of 12.2 and 5.1 for the MAN dataset (from Figure 5.8a), we notice that these values are at least 50% larger than the average of 8 and 4. Please note that the MAN dataset represents the *worst-case* scenario for our algorithm, even so, the s_{eff} values are within reasonable limits. In effect, what we have shown is that our algorithm performs well on most datasets as evidenced in Figure 5.8a and 5.9a. Moreover, this also validates an earlier result presented IN Lemma 5.14, which shows that the size of the path encoding for both the SILC and the WSP techniques is unaffected, even if a large fraction of shortest paths have δ values equal to 1.

We further investigate the effect of the size of the spatial network on the size of the path encoding and the time taken to retrieve a link in the shortest path by applying our technique to road networks of varying sizes. Figure 5.9b shows the number of Morton blocks normalized by n, in the path encoding of a road network for the WSP and SILC techniques. It is seen that the size of the path encoding using the WSP technique is almost of a constant value, indicating that the size of the decomposition is linear in the size of the network. This strongly suggests that the WSP technique is *scalable*. SILC, on the other hand, is not quite linear, but seems to be close to $\log n$, indicating that size of the SILC decomposition is $n \log n$. Furthermore, a comparison of the time taken to retrieve the next link in the shortest paths between SILC and WSP techniques is also performed. Figure 5.9c shows the time taken to retrieve a Morton block from the path encoding of a road network containing *n* vertices. It can be seen that both the WSP and the SILC techniques have almost constant access times. However, SILC outperforms WSP by at least a factor of 2.

5.6 Summary

The key idea behind the SILC and the Path Coherent Pair (PCP) frameworks is that they take advantage of the *path coherence* in spatial networks. We observed that there is coherence between the spatial position of vertices and the shortest paths between them. The SILC framework takes advantage of the coherence in the shortest paths from a source vertex *u* to destination vertice, which enables us to aggregate destination vertices into regions that share the first link in their shortest paths from *u*. The PCP framework, on the other hand, captures the coherence in the shortest paths between a set of source vertices *A* to one another to a set of destination vertices *B*. In some sense, in contrast to the SILC framework, the PCP framework is able to capture the coherence in the shortest paths between proximate sources to proximate destinations.

Next, we introduced the concept of a *Well Separated Pair* (WSP) decomposition of a point set *S* containing *n* points. Two point sets *A* and *B* are said to be well separated if the minimum separation between *A* and *B* is at least *s* times the diameter of of *A* (or *B*), where *s* is referred to as the *separation factor*. For a specified value of *s*, Callahan and Kosaraju showed [29, 31] that S can be decomposed into $O(s^d n)$ WSPs where d is the dimensionality of the input space. A well separated pair (WSP) *decomposition* of a point set S has the property that for any pair of points p, q in S there is exactly one WSP (A, B) in the WSP decomposition of S that satisfies the condition $p \in A, q \in B$.

We made two key assumptions about the nature of spatial networks. First, we assumed that for any given spatial network G, the ratio of the network distance to the spatial distance between any pair of vertices is bounded between two constants— γ_1 and γ_2. The second assumption is that for any pair p, q of vertices, the ratio of the network distance along the shortest path π to the network distance along a path that is disjoint to π is less than $\frac{1}{\delta}$.

This lead us to the main result of the chapter. We first showed that a WSP decomposition using a separation factor s of the set of vertices in G using a spatial distance function is also a WSP decomposition using a network distance function and a separation factor $s' = s \cdot \frac{\gamma_1}{\gamma_2}$. Given the set of vertices V of G, the WSP pairs, say (A, B), in the WSP decomposition of V with a separation factor $O(s^d n)$, $s > 2 + \frac{2}{\delta - 1}, \delta > 1$, have the property that the shortest paths between every pair of vertices from A to B pass through a single vertex.

We introduced the *Path Coherent Path* (PCP) framework which is a WSP decomposition of the vertices of the spatial network using a network distance function. The PCP is a path encoding of size $O(s^d n)$, $s > 2 + \frac{2}{\delta - 1}, \delta > 1$. Moreover, a shortest path can be retrieved in $O(k \log n)$ time, where $k = \pi_G(.,.)$. We also showed that the SILC framework takes $O(s^d n \log n)$ space, $s > 2 + \frac{2}{\delta - 1}, \delta > 1$ and can retrieve the shortest path in $O(k \log n)$ time in the worse case, but can retrieve the next link in the shortest path in average $O(\log \log n)$ time. In the most common case of two-dimensional spatial networks such as a road network, $d = 2$, and the fact that s is small reduces the space bounds to $O(n)$ and $O(n \log n)$, respectively. These results show an interesting interplay between the storage requirements and the retrieval time.

Chapter 6

Distance Oracles for Spatial Networks

Given a spatial network G (e.g., a road network), we address the problem of finding the *approximate network distance* between a start and a destination vertex in G. Such a query is termed an *approximate network distance query* and is the focus of this work. In particular, suppose that we are given two street addresses on the road network of Washington, DC. We are able to find the approximate shortest distance along the road network, either in terms of distance, or the time taken to travel between the start and the destination addresses so that the maximum possible error of the approximation is bounded.

Approximate network distance queries are important to querying on transportation networks. For example, given a query point q (e.g., location of an office) and a set T of restaurants, we can find the *approximate network k-nearest neighbors* [127, 154, 166] to q in T by obtaining the approximate network distance from q to objects in T and then establishing a total ordering of the objects in T with respect to q. Furthermore, an approximate *network range query* [127] can be performed by imposing an upper and/or a lower-bound on the network distance of the objects in T to q. All these queries rely on the ability to perform a large number of network distance queries in real-time on a spatial network. Furthermore, an approximation error is acceptable, in most cases, if it is bounded and small (e.g., say 10%).

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 $e \in E$, $w(e) \geq 0$ denotes the distance along that edge. In addition, for every $v \in V$, $p(v)$ denotes the spatial position of v with respect to S, a *spatial domain* also referred to as an *embedding space* (*i.e.*, a reference coordinate system). We define the *network distance* $d_G(u,v)$ to be the distance along the shortest path between u and v in the spatial network. Similarly, we define the spatial distance $d_S(u,v)$ to be a function of the position of the vertices u and v on the embedding plane. For example, in the case of a road network the network distance between two vertices is the shortest distance in miles, or the time taken to travel the road network, while the spatial distance (e.g., "crow flying" distance) is a function of latitude/longitude positions of the vertices.

Furthermore, we assume that for some spatial networks (e.g., the road networks), the network distance between any two vertices is bounded from above and below by two constants γ_1, γ_2 (presumably large), such that

$$\gamma_1 \leq \frac{d_G(u,v)}{d_S(u,v)} \leq \gamma_2;\ \gamma_1, \gamma_2 > 0.$$

The constants γ_1, γ_2 are termed the lower and upper-bound distortions of G. Narasimhan and Smid [121] provide a simple technique for estimating the values of γ_1, γ_2 for Euclidean networks which can be easily adapted to spatial networks.

In this chapter, we introduce a construct, termed an *approximate distance oracle* for spatial networks, that is capable of answering *network distance queries* between any two vertices in G with a specified approximation ε, that is — given a start vertex s and a destination vertex w in G, the network distance $S_\varepsilon(s,w)$ produced by the oracle S_ε is no more or less than an ε fraction of the actual network distance $d_G(s,w)$ between s and w in G. Our construction of the oracle takes advantage of the coherence between the spatial positions of vertices in G and the network distance between them. This enables us to find pairs of subsets of vertices in G, such that the network distance between all the vertices contained in a pair can be approximated by a single value. In effect, our oracle is a set of such pairs, such that it can answer approximate distance queries between any pair of vertices in G. The most important property of the oracle is that it only takes up linear space in the number of vertices in G, and can answer network distance queries in $O(\log n)$ time using a B-tree [12] structure. Another variant of the oracle of size $O(n \log n)$ that can retrieve the approximate network distance in $O(1)$ time using a hash table is also proposed.

Our technique is similar to the RNE technique of Shahabhi *et al.* [166] who apply a Lipschitz embedding [111] to spatial networks. The RNE technique *embeds* the vertices of the spatial network in a high-dimensional vector space, such that vertices of the spatial network are now points in a high-dimensional vector space. A simpler distance measure (e.g., L_∞ metric) between these high-dimensional vector space points approximates the network distance between the corresponding vertices in the spatial network. The RNE technique takes $O(n\sqrt{n})$ storage, has a distortion of $O(\log n)$ and an approximate network distance query takes $O(\sqrt{n})$ time. On the other hand, our technique can be viewed as an embedding technique that retains the vertices in their original embedding space (*i.e.*, two-dimensional for road networks), takes $O(\frac{n}{\varepsilon^d})$ storage, has a distortion between $(1-\varepsilon)$ and $(1+\varepsilon)$, and can answer approximate network distance queries in $O(\log n)$ time.

Approximate distance oracles have been proposed for a variety of graph networks. Thorup and Zwick [179] show that it is possible to construct an approximate oracle of size $O(kn^{1+\frac{1}{k}})$ for general graphs that can answer approximate distance queries in $O(1)$ time. The distortion of the approximate oracle of Thorup and Zwick [179] is between 1 and $(2k-1)$, where $k \geq 1$ is an integer. Gudmundsson *et al.* [68] construct an approximate oracles of size $O(n\log n)$ for *geometric t-spanner* graphs, such that the shortest path queries can be performed in $O(1)$ time with a distortion of $(1+\varepsilon)$. Gao and Zhang [57] propose an approximate oracle of size $O(n\log n)$ for unit-disk graphs, that can retrieve approximate network distance in $O(1)$ time, with a distortion of $(1+\varepsilon)$. Our work extends the above results to spatial networks, while taking advantage of the spatial positions of the vertices to provide efficient search structures such as B-trees and hash tables to the oracle.

Our algorithm and several other techniques in the literature [57, 68] make use of the *well separated decomposition* of a point set, which was proposed by Callahan and Kosaraju [31]. The concept of well separated decomposition of a point set is important to our work and was discussed in more detail in Section 5.1. The rest of the chapter is organized as follows. Section 6.1 describes an algorithm to construct an approximate distance oracle on spatial networks, which is analyzed in Section 6.2. Finally, concluding remarks are drawn in Section 6.3.

6.1 Oracle Construction

In this section, we describe an algorithm to construct the distance oracle S_ε of a spatial network $G(V,E)$. The algorithm takes G as input and produces pairs of sets $<u,v>$, such that $u, v \subset V, u \cap v = \emptyset$. The pair $<u,v>$ has the property that the maximum and the minimum network distance between any source vertex s in u and any destination vertex w in v can be approximated by the exact network distance $d_G(p_u, p_v)$ between a source p_u vertex in u and a destination vertex p_v in v, both p_u and p_v are chosen at random. The source p_u and destination p_v vertices are termed the *representative* points of u and v, respectively. For the sake of simplicity, the discussion below assumes that G is undirected. Note however that the results below are equally applicable to directed spatial networks as well.

The oracle S_ε of a spatial network is defined as follows: $S_\varepsilon = \{(< u,v >$

$, d_G(p_u, p_v))|u, v \subset V, p_u \in u, p_v \in v\}$. Now, given a source vertex s and a destination vertex w in V, $S_\varepsilon(s,w)$ is an approximation of $d_G(s,w)$ such that,

$$(1-\varepsilon) \cdot S_\varepsilon(s,w) \leq d_G(s,w) \leq (1+\varepsilon) \cdot S_\varepsilon(s,w).$$

The ε-approximate network distance of a vertex pair $<s,w>$ is obtained by searching for a triple $(u, v, d_G(p_u, p_v))$ in S_ε, such that u contains s and v contains w, in which case, $d_G(p_u, p_v)$ is the ε-approximate network distance of $d_G(s,w)$. We later show that the size of S_ε is linear in the number of vertices in G and can produce an ε-approximate network distance in $O(\log n)$ time using a B-tree. Alternatively, another representation of the oracle takes $O(\frac{n \log n}{\varepsilon^d})$ space, but can produce ε-approximate network distances in $O(1)$ time using a hash table.

The construction of the oracle proceeds as follows. Algorithm BUILDORACLE takes the spatial network G, a PR quadtree T [144] on the spatial positions of the vertices in G, and the desired approximation ε as inputs. The output of the algorithm is a list L of Morton codes [118], such that a Morton code m in L uniquely corresponds to a pair of blocks $<u,v>$ in T. Another equivalent interpretation of m is that it corresponds to a pair of subsets $<k,l>$ of vertices, such that $k(l)$ is the set of vertices contained in the subtree of T with $u(v)$ as the root block. Henceforth in this chapter, we assume that the three interpretations of m; namely — block pair, Morton code, and pair of subsets of V — are all equivalent.

The algorithm uses a list Q of block pairs. At the start of the algorithm, Q is initialized with a block pair formed by the root block of T as shown in line 1. The output list L is initially empty.

Each iteration of the algorithm retrieves the top block pair $<u,v>$ in Q. If u and v point to the same block (as is the case of the first iteration of the algorithm when the block pair $<\text{ROOTOF}(T),\text{ROOTOF}(T)>$ is retrieved), then u and v are both split into their C children blocks and the resulting C^2 block pairs are inserted into Q as shown in lines 4–7.

If u and v point to different blocks in T, then the algorithm examines if the block pair $<u,v>$ is well separated. We first choose two representative points $p_u \in u$, $p_v \in v$ at random (line 10). We then estimate the *network diameter* (or an over-approximation of the network diameter) of the blocks u and v, which is defined as the farthest vertex from p_u (or p_v) in u (or v) using a network distance measure (line 12). Estimating the diameter r can be done in a number of ways and is described in more details in Section 6.1.1. If the ratio of the network distance $d_G(p_u, p_v)$ to the diameter r is greater than or equal to $s = \frac{2}{\varepsilon}$, then the block pair is well separated and the block pair $<u,v>$ is added to the output list L (lines 13–14). We later show in Section 6.2 that if $<u,v>$ are well separated using a separation factor $s = \frac{2}{\varepsilon}$, then $d_G(p_u, p_v)$ is an ε-approximation of the network distance between any source vertex in u and destination vertex in v.

If the block pair $<u,v>$ is not well separated, then u (v) is split into its C children blocks if u (v) is not a leaf block, else it is not split. The resulting block pairs are inserted into L as shown in lines 16–26.

Algorithm 13

```
Procedure BUILDORACLE[$G$, $T$, $\varepsilon$]
Input: $G \leftarrow$ spatial network $G(V,E)$
Input: $T \leftarrow$ PR quadtree on the spatial positions of vertices in $V$
Input: $\varepsilon \leftarrow$ desired approximation; $\varepsilon > 0$
Output: $L \leftarrow$ set of Morton codes; initially empty
  (* $s \leftarrow \frac{2}{\varepsilon}$; separation factor *)
  (* $Q \leftarrow$ list containing pairs of blocks in $T$; initially empty *)
  (* $b_u, b_v \leftarrow$ list of blocks; initially empty *)
1.  INSERT($Q$, <ROOTOF($T$), ROOTOF($T$)>)
2.  while not (ISEMPTY($Q$)) do
3.    $<u,v> \leftarrow$ TOP($Q$)
4.    if ($u = v$) then
5.      (* reject the pair if $u$ ($v$) is a LEAF block *)
6.      split $u, v$ each into $C$ children blocks
7.      insert $C^2$ children block pairs of $<u,v>$ into $Q$
8.    else
9.      (* Choose representative points *)
```

```
10.         $p_u \leftarrow R(u)$; $p_v \leftarrow R(v)$
11.         (* Estimate diameter of $u$ and $v$ *)
12.         $r \leftarrow$ MAX(DIAMETER($u$), DIAMETER($v$))
13.         if ($\frac{d_G(p_u,p_v)}{r} \geq s$) then
14.             INSERT($L$, MORTONCODE($u,v$), $d_G(p_u,p_v)$)
15.         else
16.             if ISNOTLEAF($u$) then
17.                 $b_u \leftarrow C$ children blocks of $u$
18.             else
19.                 $b_u \leftarrow u$
20.             end-if
21.             if ISNOTLEAF($v$) then
22.                 $b_v \leftarrow C$ children blocks of $v$
23.             else
24.                 $b_v \leftarrow v$
25.             end-if
26.             INSERT all possible pairs in $b_u \times b_v$ into $Q$
27.         end-if
28.     end-if
29. end-while
```

It is not difficult to see that Algorithm 13 is a WSP decomposition of the set of vertices in G into a set of block pairs. Hence, given a vertex pair $<s, w>$, the properties of a WSP decomposition guarantees that there exists exactly one pair $(u, v, d_S(p_v, p_u))$ in L, such that u contains s and v contains w.

6.1.1 Estimating Network Diameter

Given a vertex p_u and a subset of vertices u, the network diameter r of u, is the farthest vertex from p_u in u using a network distance measure. One simple strategy to computing the network diameter of u is to compute the network distance from p_u to every vertex in u and then to find the maximum value. However, this can be expensive to compute. Our strategy is to compute an over-approximation r of the network diameter of u, which may be easier to compute than the exact network diameter of u. However, this strategy has the unfortunate consequence that as r is an over-approximation of the network diameter of u, Algorithm 13 would have to split the block pairs even further in order to make them well separated. Consequently, there is a potential trade-off between the time spent on computing the network diameter of a block and the total storage space expended by the algorithm. Below, we discuss several strategies to compute the network diameter of a set of vertices.

1. Given a block pair $<u, v>$, we first obtain the network distance $d_G(p_u, p_v)$ between the representative points $p_u \in u$ and $p_v \in v$. We then apply an early terminating

variant of Dijkstra's algorithm from p_u (p_v) that takes advantage of the *incremental* nature of Dijkstra's algorithm, that is – Dijkstra's algorithm with p_u (p_v) as a starting vertex visits vertices in G in an increasing order of their network distance from p_u (p_v). Dijkstra's algorithm terminates when it encounters a vertex that is farther than $\frac{d_G(p_u,p_v)}{s}$ from p_u (p_v). We now check to see if all the vertices in u (v) was visited by Dijkstra's algorithm. If yes, then u and v are well separated.

2. If r' is the diameter of the geometric bounding box of u, the network diameter of u can be over-approximated by $\gamma_2 r'$.

3. We can construct the oracle of Thorup and Zwick [179] for a large value of k (that is, large distortion) and use it for computing the network diameter.

4. The *landmark*-based approach of Goldberg and Harrelson [61] first selects a set of vertices, termed landmarks, at random. The network distance from each of the landmark vertices to all the vertices in G is precomputed. Once precomputed, the diameter of $u(v)$ can be upper bounded using the triangle inequality and the network distance to the nearest landmark.

There is one potential problem in using an over-approximation of the network diameter. If the approximation error of r cannot be easily estimated, the size of oracle in Algorithm BUILDORACLE cannot be bounded as well. We remedy this situation by assuming that the lower γ_1 and upper bound distortions of G, which can be computed using the algorithm of Narasimhan and Smid [121], is known. Given the value of γ_2, r cannot exceed $r'\gamma_2$, where r' is the diameter of the *geometric bounding box* of u. Hence, if r is larger than $r'\gamma_2$, it is set to $r'\gamma_2$.

6.1.2 Querying the Oracle

The output of Algorithm 13 is a list L of Morton codes. For each of the Morton codes in L, we associate the exact network distance between the representative points. Moreover, given a source and a destination vertex, an access structure e.g., a B-tree or a hash table, aids efficient searching on L for a block pair containing the source and the destination vertices.

Given a source vertex s and a destination vertex w, the oracle obtains the ε-approximate network distance by first computing the Morton code corresponding to the spatial position of s and w. With the aid of the access structure on L, we are able to obtain the ε-approximate network distance of s and w, which is the network distance between the representative points of the block pair $<(\!> u,v)$ in L, such that u contains s and v contains w.

6.2 Analysis of the Oracle

In this section, we provide bounds on the size of the distance oracle of G by appealing to the equivalence between the decomposition of a spatial network in Algorithm 13 and the WSP decomposition of a point set. Given a point set R in a d-dimensional space, we construct a WSP decomposition on S by first constructing a PR quadtree [124, 144] on R. For the sake of simplicity, we assume that R is contained in a unit $[0,1]^d$ d-dimensional hypercube. This hypercube forms the root block T of the PR quadtree on R. The hierarchical structure of the quadtree is constructed by recursively decomposing the block into 2^d congruent children blocks. The process continues until each block contains a single point, in which case, further subdivision is not possible. Unfortunately, if two points are close to one another in R, it may lead to a long path of trivial blocks of which only one block would form an internal node. Callahan and Kosaraju's construction [29] did not incur this problem because they used a fair-split tree which is a a data-dependent decomposition. This problem is remedied by Fischer and Har-Peled [46] through the use of a variant of a *path-compressed* quadtree which is obtained from the PR quadtree by compressing such trivial paths into one single compressed link. The advantage of the path-compressed quadtree over the PR quadtree is that its use results in a tree with a total of $O(n)$ blocks.

Our discussion does not need to resort to the path-compressed quadtree while still using regular decomposition because of certain assumptions that we make about the distribution of the vertices in the embedding space. In particular, letting Δ be the ratio of the diameter of the set of vertices V to the distance between the closest pair of vertices in V and letting T be a PR quadtree on V, the maximum depth of T is $O(\log\Delta)$.

Consequently, given a vertex v in V, the Morton code representation of $p(v)$, the spatial position of v, would be $O(\log \Delta)$ bits long. To cast this quantity in terms of n, we note that even if the data is heavily skewed so that Δ is linear in n, the length of the Morton code representation of v would still be $O(\log n)$. We claim that this assumption fits closely with the actual nature of real road networks. From a practical standpoint with respect to our experience with real data such as that found in road networks, we observe that the minimum geodesic distance between any two vertices on a road network is at least 1 meter. A PR quadtree on a sphere corresponding to the Earth with radius 6378 km and depth 24, has a 1 meter resolution at the equator. For such data, the size of the Morton code for a vertex on the road network using geographical coordinates is at most 48 bits in length.

The decomposition of R into WSPs is a *realization* on T, *i.e.*, subsets A_i, B_i of R forming a WSP (A_i, B_i) in $R \otimes R$ are pairs of blocks in R. The algorithm decomposes R into WSPs using T and s (*i.e.*, the separation factor) as inputs. The algorithm uses a list Q which is initialized by the pair $<T,T>$ corresponding to the root of the quadtree on R. At each iteration of the algorithm, a pair $<A,B>$ of blocks in T is retrieved from Q. If $<A,B>$ is $s-$separated, it is reported as a WSP. Otherwise, new pairs are obtained by replacing A and B with their 2^d children blocks, which are inserted into Q. The algorithm terminates when Q is empty.

Suppose that a pair $<u,v>$ is reported as a WSP by the algorithm. This would indicate that $(P(u), P(v))$ is not well separated, where $P(b)$ is the parent block of b. Suppose further that the side length of $P(u)$ (or $P(v)$) is x and hence also its maximum possible diameter. The total number of blocks that are not well separated from $P(u)$ is bounded by the number of blocks of diameter x that are contained within a hypersphere of diameter $(2s+1)x$ centered at $P(u)$, which contains a maximum of $O(s^d)$ blocks. Recalling that T has $O(n)$ nodes means that the algorithm creates a maximum of $O(s^d n)$ WSPs. This result and proof sketch is due to Callahan and Kosaraju [29] and we restate it below as Lemma 6.1, which is referenced in the subsequent discussion.

Lemma 6.1. *Given a point set S containing n d-dimensional points, a fixed separation factor $s \geq 2$, the WSP decomposition of S, $S \otimes S$ has $O(s^d n)$ WSPs [29][1].*

We now introduce the concept of WSPs on spatial networks. We assume that the ratio between the network and spatial distances is bounded both from above and below.

Assumption 3. $\gamma_1 \leq \frac{d_G(s,t)}{d_S(s,t)} \leq \gamma_2$, $s,t \in V, \gamma_1$ *and* $\gamma_2 > 0$.

At this point, we show how to extend the notion of a WSP decomposition in terms of a spatial distance to one in terms of a network distance. This is captured by Lemma 6.2 below.

Lemma 6.2. *Given a s-WSP decomposition of the vertices V of a spatial network $G(V,E)$ based on a spatial distance also yields a s'-WSP decomposition of V using a network distance with $s' = s \cdot \frac{\gamma_1}{\gamma_2}$.*

[1]The Lemma also holds for values of s, $0 < s < 2$, although s^d needs to be replaced with $\max(2, s^d)$

Proof. Given a s-WSP, $<A,B>$ in the decomposition of $V \otimes V$ using the spatial distance measure, the minimum spatial distance between A and B is at least $s \cdot r$, where r is the larger of the diameters of A and B.

Consider u,v two vertices in A (or B), we have $d_G(u,v) \leq \gamma_2 \cdot d_S(u,v) \leq \gamma_2 \cdot r$ as $d_S(u,v) \leq r$ by virtue of r being the diameter of A or B. r', the maximum value of $d_G(u,v)$, is the diameter of A (and B) using a network distance measure and we have that $r' \leq \gamma_2 \cdot r$. Therefore, the spatial distance diameter of A (or B) is scaled by at most a factor of γ_2 to obtain the network distance diameter r'.

Considering a vertex pair $<a,b>$, such that $a \in A, b \in B$, we have from the WSP condition and Assumption 1 that:

$$s \cdot r \leq d_S(a,b) \leq \frac{d_G(a,b)}{\gamma_1} \tag{6.1}$$

Replacing r with $\frac{r'}{\gamma_2}$ in (6.1), we obtain $s \cdot \frac{r'}{\gamma_2} \leq d_S(a,b) \leq \frac{d_G(a,b)}{\gamma_1}$. The above relationship between the lower and upper bounds on $d_S(a,b)$ can be rewritten as $r' \cdot s \cdot \frac{\gamma_1}{\gamma_2} \leq d_G(a,b)$. Now, letting, $s' = s \cdot \frac{\gamma_1}{\gamma_2}$, leading to the desired result $s' \cdot r' \leq d_G(a,b)$ which is equivalent to saying that A and B are well separated using the network distance measure with a separation factor of s'. □

We now show that a WSP decomposition of the vertices of a spatial network is a realization of an approximate distance oracle.

Lemma 6.3. *Let $<A,B>$ be a WSP in the s-WSP decomposition of G using a network distance measure, such that u^*, v^* are the representative points of A and B, respectively.*

The network distance $d_G(u^*, v^*)$ between the representative points is an $\varepsilon = \frac{2}{s}$ approximation of the network distance $d_G(u, v)$ between any pair of vertices <u, v>, such that, $u \in A$ and $v \in B$.

Proof. Given a pair of vertices <u, v>, such that $u \in A, v \in B$, from the triangle inequality, we know that

$$d_G(u^*, v^*) - d_G(u, u^*) - d_G(v^*, v) \leq d_G(u, v)$$

$$d_G(u, u^*) + d_G(u^*, v^*) + d_G(v^*, v) \geq d_G(u, v)$$

Without loss of generality, we assume that $d_G(v^*, v) \geq d_G(u, u^*)$. Substituting above, we get

$$d_G(u^*, v^*) - 2d_G(v^*, v) \leq d_G(u, v)$$

$$d_G(u^*, v^*) + 2d_G(v^*, v) \geq d_G(u, v)$$

$$d_G(u^*, v^*)(1 - \frac{2d_G(v^*, v)}{d_G(u^*, v^*)}) \leq d_G(u, v)$$
$$d_G(u^*, v^*)(1 + \frac{2d_G(v^*, v)}{d_G(u^*, v^*)}) \geq d_G(u, v)$$

In line 13 of Algorithm 13, we ensure that the condition $\frac{d_G(u^*, v^*)}{d_G(v^*, v)} \geq s$ is satisfied for all vertices in B. Substituting it above,

$$(1 - \frac{2}{s})d_G(u^*, v^*) \leq d_G(u, v) \geq (1 + \frac{2}{s})d_G(u^*, v^*)$$

Substituting, $\varepsilon = \frac{2}{s}$, we get

$$(1 - \varepsilon)d_G(u^*, v^*) \leq d_G(u, v) \geq (1 + \varepsilon)d_G(u^*, v^*)$$

□

Lemma 6.4. *For a given value of* $\varepsilon = \frac{2}{s}$*, the size of the oracle produced by Algorithm 13 is no worse than* $O((\frac{\gamma_2}{\varepsilon\gamma_1})^d n)$.

Proof. Let $<A,B>$ be a WSP, such that u^*, v^* are the representative points of A and B, respectively. We assume that A (B) is contained in a bounding hypersphere of diameter r. The network diameter of A and B is bounded by

$$\gamma_1 r \geq \textsc{Diameter}(A) \leq \gamma_2 r$$

$$\gamma_1 r \geq \textsc{Diameter}(B) \leq \gamma_2 r$$

As $<A,B>$ is a WSP, $d_G(u^*, v^*)$ can be similarly bounded by

$$d_G(u^*, v^*) \geq \gamma_2 rs.$$

The effective separation factor s' of the WSP decomposition is $\frac{s\gamma_2}{\gamma_1}$. Hence, the worse case storage requirement of the oracle is $O((\frac{\gamma_2}{\varepsilon\gamma_1})^d n)$. □

Theorem 6.5. *Given a spatial network* $G(V,E)$*, we can construct an oracle of size* $O(\frac{n}{\varepsilon^d})$ *that can retrieve the* ε*-approximate network distance between any vertex pair in* $O(\log n)$ *time.*

$O(1)$ Distance Oracle

We now introduce an alternative WSP decomposition of a point set R into pairs of the form $<p,B>$, where p is a point in R and B is a subset of R. Such a pair is termed a *one-to-many WSP* (OM-WSP) and the resultant decomposition of R into OM-WSPs is termed an *one-to-many WSP decomposition* (OM-WSPD) of R.

Lemma 6.6. *Given a point set R containing n points and a separation factor $s > 2$, a WSP decomposition of R can be decomposed into $O(s^d nh)$ OM-WSPs of form $(\{a\},B)$, where $a \in R$, $B \subset R$, and h is the height of the PR quadtree on R.*

Proof. Let T be a PR quadtree of height h on the spatial positions of R. Suppose that a block i in T containing s_i points is paired up with a_i other blocks in T during the WSP decomposition of R. As a result, $s_i a_i$ OM-WSPs are created. The total number of OM-WSPs generated by $p = O(n)$ nodes in T is given by $\sum_{i=1}^{p} a_i s_i$, which is less than $s^d \sum_{i=1}^{p} s_i$ as $O(s^d)$ upper-bounds a_i. In a PR quadtree of height h, we know that $\sum_{i=1}^{n} s_i = O(nh)$. Substituting the above result in $s^d \sum_{i=1}^{p} s_i$, we obtain that the total number of OM-WSPs created by the algorithm is $O(s^d nh)$. □

From our assumption on the properties of spatial networks, we know that for a PR quadtree T on the position of vertices on a spatial network, the height h of T is $h = O(\log n)$. Hence, the WSP decomposition of the vertices of a spatial network results in $O(s^d n \log n)$ OM-WSPs.

We now show that given a source vertex s and a destination vertex w, the OM-WSP containing the pair can be found in $O(1)$ using the properties of a WSP decomposition. Given a OM-WSPD of a spatial network, for each vertex $u \in V$, let P_u be set of pairs of the form $<u,B>$ in the decomposition. Furthermore, we construct *distance classes D_j*

using the OM-WSPs in P_u, such that D_j contains all the pairs $<u,B>$ in P_u satisfying the condition $(1-\rho)j \leq d_S(u,R(B)) \leq (1+\rho)j$, where $R(B)$ is the representative point of B and $\rho > 0$. The following lemma shows that the use of a hash table will enable finding the OM-WSP containing the input source and the destination vertex pair in $O(1)$ time.

Lemma 6.7. *Given a vertex pair $s,w \in V$, the number of OM-WSPs of the form $<s,B>$ in the canonical realization, such that $d_S(s,B) \leq d_S(s,w)$ and $d_S(s,R(B)) \in DC_j$ is a constant depending only on ρ, d, γ_1, and γ_2.*

Proof. From [28]. □

This leads us to our final result.

Theorem 6.8. *Given a spatial network $G(V,E)$, we can construct an oracle of size $O(\frac{n\log n}{\varepsilon^d})$ that can retrieve the ε-approximate network distance between any vertex pair in $O(1)$ time.*

6.3 Summary

Given a spatial network G, we addressed the problem of finding the approximate network distance between a given pair of vertices. That is, given a pair of vertices $<u,v>$ in V, we propose an approximate distance oracle S_ε for spatial networks that can compute an approximate network distance $S_\varepsilon(u,v)$ between u and v such that $S_\varepsilon(u,v)$ is no more or less than a ε-fraction than the actual actual network distance between u and v. We take advantage of the coherence between the spatial position of vertices and the network distance between them. Using this observation, we are able to construct a distance oracle that is able to obtain the ε-approximate network distance between two vertices of the spatial network. This work is important to the real-time processing of a variety of spatial queries on transportation networks. With the aid of the well separated pair technique, which has been applied to spatial networks, the network distance between every pair of vertices in the spatial network is efficiently represented. We first introduce an ε-approximate distance oracle of size $O(\frac{n}{\varepsilon^d})$ that is capable of retrieving the approximate network distance in $O(\log n)$ time using a B-tree structure. We also propose another ε-approximate distance oracle of size $O(\frac{n \log n}{\varepsilon^d})$ that can retrieve the approximate network distance in $O(1)$ time using a hash table.

Chapter 7

All *k* Nearest Neighbor Algorithm for Point-clouds

In recent years there has been a marked shift from the use of triangles to the use of points as object modeling primitives in computer graphics applications (e.g., [6, 67, 84, 96, 104, 105, 129]). A point model (often referred to as a *point-cloud*) usually contains millions of points. Improved scanning technologies [104] have resulted in enabling even larger objects to be scanned into point-clouds. Note that a point-cloud is nothing more than a collection of scanned points and may not even contain any topological information. However, most of the topological information can be deduced by applying suitable algorithms on the point-clouds. Some of the fundamental operations performed on a freshly scanned point-cloud include the computation of *surface normals* in order to be able to illuminate the scanned object, applications of *noise-filters* to remove any residual noise from the scanning process, and tools that change the *sampling rate* of the point model to the desired level. What is common to all three of these operations is that they work by computing the *k* nearest neighbors for each point in the point-cloud. There are two important distinctions from other applications where the computation of neighbors is required. First of all, neighbors need to be computed for all points in the dataset, potentially this task can be optimized. Second, no assumption can be made about the size of the dataset. In this Chapter, we focus on a solution to the *k-nearest-neighbor*(*kNN*) problem, also known as the all-points k-nearest-neighbor problem, which takes a point-cloud dataset R as an input and computes the k nearest neighbors for each point in R.

We start by comparing and contrasting our work with the related work of Clarkson [36] and Vaidya [185]. Clarkson proposed an $O(n \log \delta)$ algorithm for computing the nearest neighbor to each of n points in a dataset S, where δ is the ratio of the diameter of S and the distance between the closest pair of points in S. Clarkson uses a PR quadtree (e.g., see [144]) Q on the points in S. The running time of his algorithm depends on the depth $d = \delta$ of Q. This dependence on the depth is removed by Vaidya who proposed using a hierarchy of boxes, termed a Box tree, to compute the k nearest neighbors to each of the n points in S in $O(kn \log n)$ time. There are two key differences between our algorithm and those of Clarkson and Vaidya. First of all, our algorithm can work on most disk-based (out of core) data structures regardless of whether they are based on a regular decomposition of the underlying space such as a quadtree [144] or on object hierarchies such as an R-tree [73]. In contrast to our algorithm, the methods of Clarkson and Vaidya have only been applied to memory-based (i.e., incore) data structures such as the PR quadtree and Box tree, respectively. Consequently, their approaches are limited by the amount of physical memory present in the computer on which they are executed. The second difference is that it is easy to modify our algorithm to produce nearest neighbors incrementally, *i.e.*, we are able to provide a variable number of nearest neighbors to each point in *S* depending on a condition, which is specified at run-time. The incremental behavior has important applications in computer graphics. For example, the number of neighbors used in computing the normal to a point in a point-cloud can be made to depend on the curvature of a point.

The development of efficient algorithms for finding the nearest neighbors for a single point or a small collection of points has been an active area of research [81, 135]. The most prominent neighbor finding algorithms are variants of Depth-First Search (DFS) [135] or Best-First Search (BFS) [81] methods to compute neighbors. Both algorithms make use of a search hierarchy which is a spatial data-structure such as an R-tree [73] or a variant of a quadtree or octree (e.g., [144]). The DFS algorithm, also known as branch-and-bound, traverses the elements of the search hierarchy in a predefined order and keeps track of the closest objects to the query point that have been encountered. On the other hand, the BFS algorithm traverses the elements of the search hierarchy in an order defined by their distance from the query point. The BFS algorithm that we use [81], stores both points and *blocks* in a priority queue. It retrieves points in an increasing order of their distance from the query point. This algorithm is *incremental* as the number of nearest neighbors *k* need not be known in advance. Successive neighbors are obtained as points are removed from the priority queue. A brute force method to perform the *kNN* algorithm would be to compute the distance between every pair of points in the dataset and then to choose the top k results for each point. Alternatively, we also observe that repeated application of a neighbor finding technique [119] on each point in the dataset also amounts to performing a *kNN* algorithm. However, like the brute-force method, such an algorithm performs wasteful repeated work as points in proximity share neighbors and ideally it is desirable to avoid recomputing these neighbors.

Some of the work entailed in computing the k nearest neighbors can be reduced by making use of the approximate nearest neighbors [119]. In this case, the approximation is achieved by making use of an error-bound ε which restricts the ratio of the distance from the query point q to an approximate neighbor and the distance to the actual neighbor to be within $1+\varepsilon$. When used in the context of a point-cloud algorithm, this method may lead to inaccuracies in the final result. In particular, point-cloud algorithms that determine local surface properties by analyzing the points in the neighborhood may be sensitive to such inaccuracies. For example, such problems can arise in algorithms for computing normals, estimating local curvature, as well as sampling rate and local point-cloud operators such as *noise-filtering* [47, 96], *mollification* and removal of *outliers* [190]. In general, the correct computation of neighbors is important in two main classes of point-cloud algorithms: algorithms that identify or compute properties that are common to all of the points in the neighborhood, and algorithms that study variations of some of these properties.

An important consideration when dealing with point models that is often ignored is the size of the point-cloud datasets. The models are scanned at a *high fidelity* in order to create an illusion of a smooth surface. The resultant point models can be on the order of several millions of points in size. Existing algorithms such as normal computation [117] which make use of the suite of algorithms and data structures in the Approximate Nearest Neighbor (ANN) library [119] are limited by the amount of physical memory present in the computer on which they are executed. This is because the ANN library makes use of in-core data structures such as the k-d tree [15] and the BBD-tree [10]. As larger objects are being converted to point models, there is a need to examine neighborhood finding techniques that work with data that is out of core and and thus out-of-core data structures should be used. Of course, although the drawback of out-of-core methods is the incurrence of I/O costs, thereby reducing their attractiveness for real-time processing, the fact that most of the techniques that involve point-clouds are performed *offline* mitigates this drawback.

There has been a considerable amount of work on efficient disk-based nearest neighbor finding methods [81, 135, 192]. Recently, there has also been some work on the *kNN* algorithm [20, 192]. In particular, the algorithm by Böhm [20], termed *MuX* uses the DFS algorithm to compute the neighborhoods of one block, say b, at a time (i.e., it computes the k nearest neighbors of all points in b before proceeding to compute the k nearest neighbors in other blocks) by maintaining and updating a best set of neighbors for each point in the block as the algorithm progresses. The rationale is that this will minimize disk I/O as the *k* nearest neighbors of points in the same block are likely to be in the same set of blocks. The GORDER method [192] takes a slightly different approach in that although it was originally designed for high-dimensional data-points (e.g., similarity retrieval in image processing applications), it can also be applied to low-dimensional datasets. In particular, this algorithm first performs a Principal Component Analysis (PCA) to determine the first few dominant directions in the data space and then all of the objects are projected to this dimensionally-reduced space, thereby resulting in drastic reduction in the dimensionality of the point dataset. The resulting blocks are organized using a regular grid, and, at this point, a *kNN* algorithm is performed which is really a sequential search of the blocks

Even though both the GORDER [192] and the MuX [20] methods compute the neighborhood of all points in a block before proceeding to process points in another block, each point in the block keeps track of its *k*-nearest neighbors encountered thus far. Thus this work is performed independently and in isolation by each point with no reuse of neighbors of one point as neighbors of a point in spatial proximity. Instead, in our approach we identify a region in *space* that contains all of the *k* nearest neighbors of a *collection* of points (the space is termed *locality*). Once the best possible *locality* is built, each point searches only the locality for the correct set of *k* nearest neighbors. This results in large savings. Also, our method makes no assumption about the size of the dataset or the sampling-rate of the data. Experiments (Section 7.5) show that our algorithm is faster than both the GORDER and the MuX methods and performs substantially fewer distance computations.

The rest of the Chapter is organized as follows. Section 7.1 defines the concepts that we use and provides a high level description of our algorithm. Section 7.2 describes the *locality* building process for *blocks*. Section 7.3 describes an incremental variant of our *kNN* algorithm, while Section 7.4 describes a non-incremental variant of our *kNN* algorithm. Section 7.5 presents the results of our experiments, while Section 7.6 discusses related applications that can benefit from the use of our algorithm. Section 7.7 presents a neighborhood algorithm that can process massive datasets containing up to two billion points on a cluster containing 20 machines running the MAP-REDUCE framework [39]. Finally, concluding remarks are drawn in Section 7.8.

7.1 Preliminaries

In this Chapter we assume that the data consists of points in a multi-dimensional space and that they are represented by a hierarchical spatial data structure. Our algorithm makes use of a disk-based quadtree variant that recursively decomposes the underlying space into blocks until the number of points in a block is less than some *bucket capacity* *B* [144]. In fact, any other hierarchical spatial data structure could be used including some that are based on object hierarchies such as the R-tree [73]. The blocks are represented as nodes in a tree access structure which enables point query searching in time proportional to the logarithm of the width of the underlying space. The tree contains two types of nodes: leaf and non-leaf. Each non-leaf node has at most 2^d nonempty *children*, where d corresponds to the *dimension* of the underlying space. A *child* node occupies a region in space that is fully contained in its parent node. Each leaf node contains a pointer to a *bucket* that stores at most B points. The *root* of the tree is a special block that corresponds to the entire underlying space which contains the dataset. While the blocks of the access structure are stored in main-memory, the buckets that contain the points are stored on disk. In our implementation, a count is maintained of the number of points that are contained within the *subtree* of which the corresponding block b is the root and a *minimum bounding box* of the space occupied by the points that b contains.

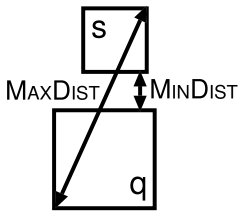


Figure 7.1: Example illustrating the values of the MinDist and MaxDist distance estimates for blocks q and b.

We use the Euclidean metric (L_2) for computing distances. It is easy to modify our *kNN* algorithm to accommodate other distance metrics. Our implementation makes extensive use of the two distance estimates MinDist and MaxDist (Figure 7.1). Given two blocks q and s, the procedure $\text{MinDist}(q,s)$ computes the minimum possible distance between a point in q to a point in s. When a list of blocks is ordered by their

MINDIST value with respect to a reference block or a point, the ordering is called a MINDIST ordering. Given two blocks q and s, the procedure MAXDIST(q,s) computes the maximum possible distance between a point in q to a point in s. When a list of blocks is ordered by their An ordering based on MAXDIST is called a MAXDIST ordering. The *kNN* algorithm identifies the k nearest neighbors for each point in the dataset. We refer to the set of k nearest neighbors of a point p as the *neighborhood* of p. While the neighborhood is used in the context of points, *locality* defines a neighborhood of blocks. Intuitively, the locality of a block b is the region in space that contains all the k nearest neighbors of all points in b. We make one other distinction between the concepts of neighborhood and locality. In particular, while neighborhoods contain no other points other than the k nearest neighbors, locality is more of an approximation and thus the locality of a block b may contain points that do not belong to the neighborhood of any of the points contained within b.

Our algorithm has the following high-level structure. It first builds the locality for a block and later searches the locality to construct a neighborhood for each point contained within the block. The pseudo-code presented in Algorithm 14 explains the high level workings of the *kNN* algorithm. Lines 1–2 compute the *locality* of the blocks in the search hierarchy Q on the input point-cloud. Lines 3–4 build a neighborhood for each point in b using the locality of b.

Algorithm 14

Procedure $kNN[Q,k]$

Input: Q is the search hierarchy on the input point-cloud

```
   (∗ high-level description of kNN algorithm ∗)
1.  for each block b in Q do
2.      Build locality S for b in Q
3.      for each point p in b do
4.          Build neighborhood of p using S and k
5.      end-for
6.  end-for
7.  return
```

7.2 Building the Locality of a Block

As the locality defines a region in space, we need a measure that defines the extent of the locality. Given a *query block*, such a measure would implicitly determine if a point or a block belongs to the locality. We specify the extent of a locality by a distance-based measure that we call PRUNEDIST. All points and blocks whose distance from the query block is less than PRUNEDIST belong to the locality. The challenge in building localities is to find a good estimate for PRUNEDIST. Finding the smallest possible value of PRUNEDIST requires that we examine every point which defeats the purpose of our algorithm which is why we resort to estimating it.

We proceed as follows. Assume that the query block q is in the vicinity of other blocks of various sizes. We want to find a set of blocks so that the total number of points that they contain is at least k, while keeping PRUNEDIST as small as possible. We do this by processing the blocks in increasing order of their MAXDIST order from q and adding them to the locality. In particular, we sum the counts of the number of points in the blocks until the total number of points in the blocks that have been encountered exceeds k and record the current value of MAXDIST as the value of PRUNEDIST. At this point, we process the remaining blocks according to their MINDIST order from q and add them to the locality until encountering a block b whose MINDIST value exceeds PRUNEDIST. All remaining blocks need not be examined further and are inserted into list PRUNEDLIST. Note that an alternative approach would be to initially process the blocks in MINDIST order, adding them to the locality, and set PRUNEDIST be the maximum MAXDIST value encountered so far and halting once the sum of the counts is greater than k to prune every block whose MINDIST value is greater than PRUNEDIST. This approach does not yield as tight an estimate for PRUNEDIST as can be seen in the example in Figure 7.2.

The pseudo-code for obtaining the locality of a block is given in Algorithm 15. The inputs to the BUILDLOCALITY algorithm are the query block q, a set of blocks Q corresponding to the partition of the underlying space into a set of blocks, and the value

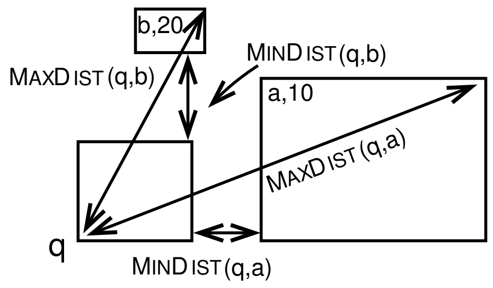


Figure 7.2: Query block q in the vicinity of two other blocks a and b containing 10 and 20 points respectively. When k is 10, choosing a with a smaller MINDIST value does not provide the lowest possible PRUNEDIST bound.

of k. Using these inputs, the algorithm computes the locality S of q. The while-loop in lines 1-7 visits blocks in Q in an increasing MAXDIST ordering from q and adds them to S. The loop terminates when k or more points have been added to S, at which point the value of PRUNEDIST is known. Lines 8–14 of the algorithm now add blocks in Q to S, whose MINDIST to q is lesser than the PRUNEDIST value. Line 17 returns the locality S of q, a set PRUNEDLIST of blocks in Q that does not belong to S, and the value of PRUNEDIST.

The mechanics of the algorithm are illustrated in Figure 7.3. The figure shows q in the vicinity of several other blocks. Each block is labeled with a letter and the number of points that it contains. For example, suppose that $k = 3$, and let $Q = \{$a, b, c, d, e, f, i, j, k, l, m, o, p, q, x, y$\}$ be a decomposition of the underlying space into a set of blocks. The algorithm first visits blocks in a MAXDIST ordering from q, until 3 points are found. That is, the algorithm adds blocks x and y to S and PRUNEDIST is set to MAXDIST(q, y). We now choose all blocks whose MINDIST from q is less than

PRUNEDIST resulting in blocks b, e, f, i, d, p, q, k, m, and o being added to S.

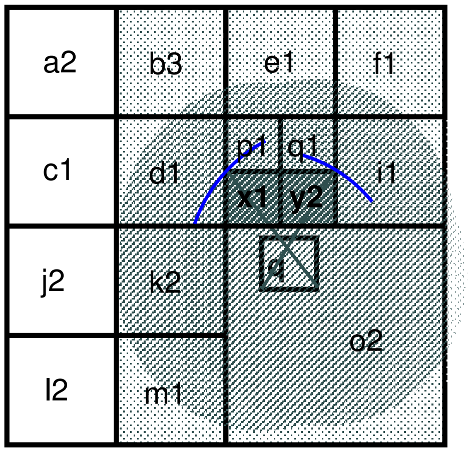


Figure 7.3: Illustration of the workings of the BUILDLOCALITY algorithm. The labeling scheme assigns each block a label concatenated with the number of points that it contains. q is the query block. Blocks x and y are selected based on the value of MAXDIST, while blocks b, e, f, i, d, p, q, k, m, and o are also selected as their MINDIST value from $q \leq$ PRUNEDIST.

Algorithm 15

Procedure BUILDLOCALITY[q, Q, k]

Input: q is the *query block*

Input: Q is a set of blocks; decomposition of underlying space

Output: $S \leftarrow$ set of blocks, initially empty; locality of q

Output: PRUNEDLIST $\leftarrow$ set of blocks, initially empty; $\forall\, b \in Q$ *s.t.,* $b \notin S$

Output: PRUNEDIST $\leftarrow$ size of the locality; initially 0

(∗ COUNT(b) is the number of points contained in the block b ∗)

```
    (* integer total ← k *)
    (* block b ← NULL *)
1.  while (total ≥ 0) do
2.      b ← NextInMaxDistOrder(Q, q)
3.      (* Remove b from Q *)
4.      PruneDist ← MaxDist(q, b)
5.      total ← total − count(b)
6.      Insert(S, b)
7.  end-while
8.  while not (IsEmpty(Q)) do
9.      b ← NextInMinDistOrder(Q, q)
10.     (* Remove b from Q *)
11.     if (MinDist(q,b) ≤ PruneDist) then
12.         Insert(S, b)
13.     else
14.         Insert(PrunedList, b)
15.     end-if
16. end-while
17. return (S, PrunedList, PruneDist)
```

7.2.1 Optimality of the BUILDLOCALITY algorithm

In this Section, we present few interesting properties of the BUILDLOCALITY algorithm. The discussion below is based on [16].

Definition 7.1 (locality)**.** *Let Q be a decomposition of the underlying space into a set of blocks. The locality S of a point q is defined to be a subset of Q, such that all of the k nearest neighbors of q are contained in S. The locality S of a block b is defined to be a subset of Q, such that all the k nearest neighbors of all the points in b are contained in S.*

Definition 7.2. *Given a point q, let n_i^q be the i^{th} nearest neighbor of q at a distance of d_i^q. Let b_i^q be a block in Q containing n_i^q.*

Definition 7.3 (*kNN*-hyper-sphere)**.** *Given a point q, the kNN-*hyper-sphere *$H(q)$ of q is a hyper-sphere of radius r_q centered at q, such that $H(q)$ completely contains all the blocks in the set $L = \{b_i^q | i = 1...k\}$.*

Corollary 7.4. *The number of points contained in the* kNN*-hyper-sphere $H(q)$ of a point q is $\geq k$.*

Definition 7.5 (Optimality)**.** *The locality S of a point q is said to be* optimal*, if S contains only those blocks that intersect with $H(q)$.*

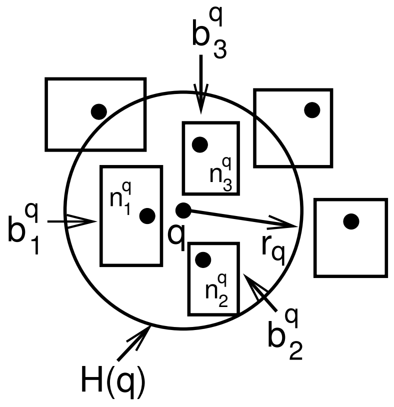


Figure 7.4: Figure shows the *kNN*-hyper-sphere $H(q)$ of a point q when $k = 3$. Note that $H(q)$ completely contains the blocks b_1^q, b_2^q and b_3^q.

The rationale behind the definition of optimality is explained below. Let us assume that an optimal algorithm to compute the locality of q consults an *oracle*, which reveals the identify of the set of blocks $L = \{b_1^q, b_2^q..b_q^k\}$ in Q containing the k nearest neighbors of a point q (as shown in Figure 7.4). Given such a set L by the oracle, the optimal algorithm would still need to examine the blocks in the hyper-region $H(q)$ in order to verify that the points in L are indeed the k closest neighbors of q. We now show that our algorithm is optimal – that is, in spite of not using an oracle, the locality of q computed by our algorithm is always optimal.

Lemma 7.6. *Given a space decomposition Q into set of blocks, the locality of a point q produced by Algorithm 15 is optimal.*

Proof. Algorithm 15 computes the locality S of a point q by adding blocks from Q to S in an increasing MAXDIST ordering from q, until S contains at least k points. At this point, let PRUNEDIST be the maximum value of MAXDIST encountered so far (*i.e.*, to the last block in the MAXDIST ordering that was added to S). Next, the algorithm adds all blocks whose MINDIST value is less than the PRUNEDIST. We now demonstrate that the locality S is optimal by showing that a block that does not intersect with the *kNN*-hyper-sphere $H(q)$ of q cannot belong in S. Suppose that $b \in S$ is a block that does not intersect $H(q)$ of radius r_q, – that is, by definition

$$r_q < \textsc{MinDist}(q,b) \leq \textsc{PruneDist}. \tag{7.1}$$

From Corollary 7.4, we know that $H(q)$ contains at least k points.

Hence,

$$\textsc{PruneDist} \leq r_q. \tag{7.2}$$

Combining Equations 7.1 and Equation 7.2, we have

$$\textsc{PruneDist} \leq r_q < \textsc{MinDist}(q,b) \leq \textsc{PruneDist},$$

which is a contradiction. □

Note however that not all the blocks that intersect with $H(q)$ must be in S, as shown in Figure 7.5, where D and E intersect $H(q)$ while not being in S.

We now show that the locality of a block b that is computed by Algorithm 15 is also optimal.

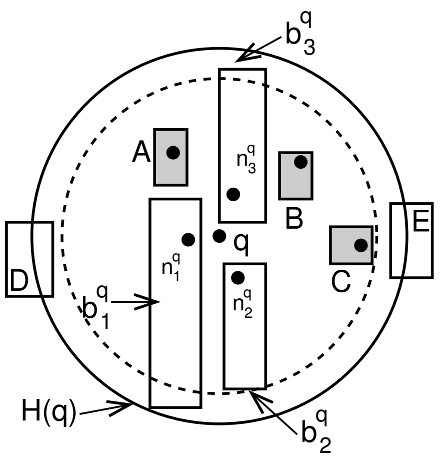


Figure 7.5: The locality S of a point q computed by Algorithm 15 ($k = 3$) initially adds A, B, C to the locality of q, thus satisfying the initial condition that the number of points in S be equal to 3. Now PRUNEDIST is set to MAXDIST(q, C). Next, we add blocks whose MINDIST to q is less than the PRUNEDIST, thus adding the blocks b_1^q, b_2^q, and b_3^q to S. Notice that the locality of q computed by Algorithm 15 may not contain all the blocks that intersect with $H(q)$ *i.e.*, blocks D and E intersect with $H(q)$, but are not in S.

Definition 7.7 (*kNN*-hyper-region)**.** *Given a block b, let L be the subset of blocks in Q such that any block in L contains at least one of the k nearest neighbors of a point in b. The kNN-*hyper-region $H(b)$ *of b is a hyper-region R, such that any point contained in R is closer to b than the block r containing the farthest possible point from b in L – that is, r is the farthest block in L, if* $\forall b_i \in L$, MAXDIST$(r,b) \geq$ MAXDIST(b_i,b). *Now,* $H(b)$ *is a hyper-region R, such that the minimum distance of a point in R to b is less than or equal to* MAXDIST(r,b).

Definition 7.8 (Optimality)**.** *The locality S of a block b is said to be* optimal, *if S contains only those blocks that intersect with the* kNN-*hyper-region of b.*

The rationale behind the definition of the optimality of a block is the same as that for a point. Even if our algorithm is provided with an *oracle*, which identifies the subset of blocks in Q containing at least one of the k nearest neighbors of a point in b, the blocks that intersect with $H(b)$ must be examined in order to prove correctness of the result.

Corollary 7.9. *The number of objects contained in the* kNN-*hyper-sphere* $H(b)$ *of a block* b *is* $\geq k$.

Lemma 7.10. *Given a space decomposition* Q *into set of blocks, the locality of a block* b *produced by Algorithm 15 is optimal.*

Proof. Follows from Lemma 7.6. □

Note however, that the algorithm is optimal with respect to the given space decomposition Q. That is, the BUILDLOCALITY algorithm will never add a block b to the locality that cannot contain a nearest neighbor to any point contained in b, although, depending on the nature of the decomposition, the size of the locality may be large.

7.3 Incremental *kNN* Algorithm

We briefly describe the working of a incremental variant of our *kNN* algorithm. This algorithm is useful when variable number of neighbors are required for each point in the dataset. For example, when dealing with certain point-cloud operations, where the number of neighbors required for a point p is a function of its local characteristics (e.g., curvature), the value of k cannot be pre-determined for all the points in the dataset, *i.e.*, a few points may require more than k neighbors. The incremental *kNN* algorithm given in Algorithm 16 can produce as many neighbors as required by the point-cloud operation. This is contrast to the standard implementation of the ANN algorithm [119], where retrieving the $k+1^{th}$ neighbor of p entails recomputing all of the first $k+1$ neighbors to p.

Algorithm INC*kNN* computes the nearest neighbors of a point p incrementally. The inputs to the algorithm are the point p whose nearest neighbors are being computed, the leaf block b containing p and the locality S of b. A priority queue Q in line 1 retrieves elements in increasing MINDIST ordering from p. Initially, the locality S of b is *enqueued* in Q in line 3. At each step of the algorithm the top element e in Q is retrieved. If e is a BLOCK, then e is replaced with its children blocks (line 16–17). If e is a point, it is reported (line 19) and the control of the program returns back to the user. Additional neighbors of p are retrieved by making subsequent invocations to the algorithm. Note that S is guaranteed to only contain the first k nearest neighbors of p, after which the PRUNEDLIST of the parent block of b (subsequently, an ancestor) in the search hierarchy is enqueued into Q, as shown in lines 7–13.

Algorithm 16

```
Procedure IncKNN[p, b, S]
Input: b is a leaf block
Input: p is a point in b
Input: S is a set of blocks; locality of b
    (∗ FindPruneDist(b) returns the PruneDist of the block b ∗)
    (∗ FindPrunedList(b) returns the PrunedList of the block b ∗)
    (∗ Parent(b) returns the parent block of b in the search hierarchy ∗)
    (∗ element e ∗)
    (∗ priority_queue Q ← empty; priority queue of elements ∗)
    (∗ float d ← FindPruneDist(b) ∗)
1.  Init:  InitQueue(Q)
2.         (∗ MinDist ordering of elements in Q from p ∗)
3.         EnQueue(Q, S)
4.  End-Init
5.  while not (IsEmpty(Q)) do
6.      e ← DeQueue(Q)
7.      if (MinDist(e,b) $\geq$ d) then
8.          if (b = root) then
9.              d ← ∞
10.         else
```

```
11.             b ← PARENT(b)
12.             ENQUEUE(Q, FINDPRUNEDLIST(b)
13.             d ← FINDPRUNEDIST(b)
14.         end-if
15.     end-if
16.     if (e is a BLOCK) then
17.         ENQUEUE(Q, CHILDREN(e))
18.     else (∗ e is a POINT ∗)
19.         report e as the next neighbor (and return)
20.     end-if
21. end-while
```

7.4 Non-Incremental *kNN* Algorithm

In this Section, we describe our *kNN* algorithm that computes the k nearest neighbors of each point in the dataset. A point x whose k neighbors are being computed is termed the *query point*. An ordered set containing the k nearest neighbors of x is termed the *neighborhood* $n(x)$ of x. Although the examples in this Section assume a two-dimensional space, the concepts hold true for arbitrary dimensions. Let $n(x) = \{q_1^x, q_2^x, q_3^x ... q_k^x\}$ be the neighborhood of point x, such that q_i^x is the i^{th} nearest neighbor of x, $1 \leq i \leq k$ with q_1^x being the closest point in $n(x)$. We represent the L_2 distance of a point $q_i^x \in n(x)$ to x as

$L_2^x(q_i) = \|q_i - x\|$ or d_i^x. Note that all points in the neighborhood of x are drawn from the *locality* of the leaf block containing x. The L_∞ distance between any two points u and v is denoted by $L_\infty^u(v)$.

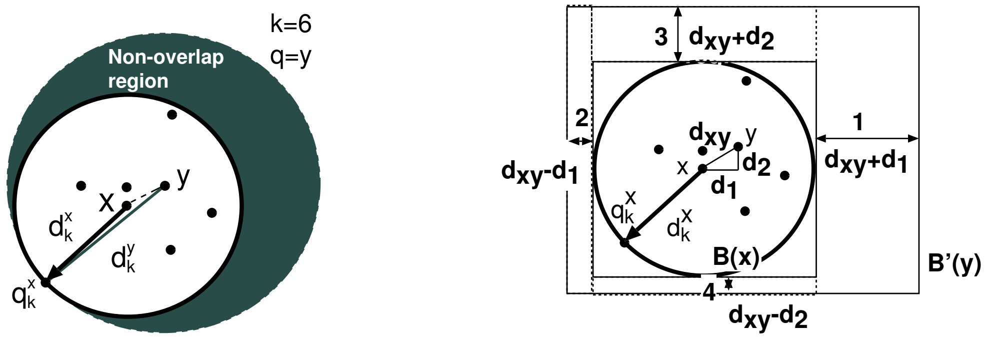
The neighborhood of a succession of query points is obtained as follows. Suppose that the neighborhood $n(x)$ of the query point x has been determined by a search process. Let q_k^x be the farthest point in $n(x)$, such that the k nearest neighbors of x are contained in a circle (a hyper-sphere in higher dimensions) of radius d_k^x centered at x. Let y be the next query point under consideration. As mentioned earlier, the algorithm benefits from choosing y to be close to x. Without loss of generality, assume that y is to the *east* and *north* of x as shown in Figure 7.6a. As both x and y are spatially close to each other, they may share many common neighbors and thus we let y use the neighborhood of x as an initial estimate of y's neighborhood, termed the *approximate neighborhood* of y and denoted by $n'(y)$, and then try to improve upon it. At this point, let d_k^y record the distance from y to the farthest point in the approximate neighborhood $n'(y)$ of y.

Of course, some of the points in $n'(y)$ may not be in $n(y)$. The fact that we use the L_2 distance metric means that the region spanned by $n(x)$ is of a circular shape. Therefore, as shown in Figure 7.6a, we see that some of the k nearest neighbors of y may lie in the shaded crescent-shaped region formed by taking the set difference of the points contained in the circle of radius d_k^y centered at y and the points contained in the circle of radius d_k^x centered at x. Thus, in order to ensure that we obtain the k

nearest neighbors of y, we must also search this crescent-shaped region whose points may displace some of the points in $n'(y)$. However, it is not easy to search such a region due to its shape, and thus the *kNN* algorithm would benefit if the shape of the region containing the neighborhood could be altered to enable efficient search, while still ensuring that it contains the k nearest neighbors of y; although it could contain a limited number of additional points.

Let $B(x)$ be the *bounding box* of $n(x)$, such that any point p contained in $B(x)$ satisfies the condition $L^x_\infty(p) \leq d^x_k$, *i.e.*, $B(x)$ is a square region centered at x of width $2 \cdot d^x_k$, such that it contains all the points in $n(x)$. Note that $B(x)$ contains all the k nearest neighbors of x and additional points in the region that does not overlap $n(x)$. While estimating a bound on number of points in $B(x)$ is difficult, at least in two-dimensional space we know that the ratio of the non-overlap space occupied by $B(x)$ to $n(x)$ is $\frac{4-\pi}{\pi}$. Consequently, the expected number of points in $B(x)$ is proportionately larger than $n(x)$.

Once we have $B(x)$ of a point x, we obtain a rectangular region $B'(y)$, termed *approximate bounding box* of $n(y)$, such that $B'(y)$ is guaranteed to contain all the points in $n(y)$. This is achieved by adding four simple rectangular regions to $B(x)$ as shown in Figure 7.6b. In general for a d-dimensional space, 2^d such regions are formed. Although this process is simple, it may have the unfortunate consequence that its successive application to query points will result in larger and larger bounding boxes – that is, $B'(y)$ computed using such a method is larger than $B(y)$. We avoid this repeated growth by following the determination of d_k^y using $B'(y)$ with a computation of a smaller $B(y)$ with a width of $2 \cdot d_k^y$.



(a) (b)

Figure 7.6: a) Searching the shaded region for points closer to y than q_k^y is sufficient. b) To compute $B(y)$ from $B(x)$ requires four simple region searches. Compared to searching the crescent shaped region, these region searches are easy to perform.

Algorithm 17 takes a leaf block b and the locality S of b as input and computes the neighborhood for all points in b. First of all, the points in b are visited in some pre-determined sequence (line 1), usually the ordering of points is established using a space-filling curve [144]. The neighborhood $n(u)$ of the first point u in b (lines 5-8) is computed by choosing the k closest points to u in S. This is done by making use of an incremental nearest neighbor finding algorithm such as BFS [81]. Note that at this stage, we could also make use of an approximate version of BFS as pointed out earlier in this Section. Once the k closest points have been identified, the value of d_k^u is known (line 9). At this point we add the remaining points in $B(u)$ as they are needed for the computation of the neighborhood of the next query point in b. In particular, $B(u)$ is constructed by adding points $o \in S$ to $n(u)$ such that they satisfy the condition $L^u_\infty(o) \leq d^u_k$ (lines 10-15). Subsequent points in b are handled in lines 16–21. The points in the bounding box $B'(u)$ of u is computed by using the points in the bounding box $B(u)$ of the previous point p and then making 2^d region searches on S as shown in Figure 7.6b (line 18). Finally, $B(u)$ is computed by making an additional region search on $B'(u)$ as shown in line 21.

Algorithm 17

```
Procedure BUILDNEIGHBORHOOD[$b$, $S$]
Input: $b \leftarrow$ a leaf block
Input: $S \leftarrow$ set of blocks; locality of $b$
   (* point $p, u \leftarrow$ empty *)
   (* ordered_set $B_p, B_u, B'_u \leftarrow$ empty *)
   (* If $B$ is an ordered set, $B[i]$ is the $i^{th}$ element in $B$ *)
   (* integer count $\leftarrow 0$ *)
1.   for each point $u \in b$ do
2.      if ($p$ = empty) then
3.         (* compute the neighborhood of the first point in $b$ *)
4.         count $\leftarrow 0$
5.         while (count $< k$) do
```

```
6.          INSERT($B_u$, NEXTNN($S$))
7.          $count \leftarrow count + 1$
8.       end-while
9.       $d_k^u \leftarrow L_2^u(B_u[k\,])$
10.      $o \leftarrow$ NEXTNN($S$)
11.      (* add all points that satisfy the $L_\infty$ criterion *)
12.      while ($L_\infty^u(o) \leq d_k^u$) do
13.         INSERT($B_u$, $o$)
14.         $o \leftarrow$ NEXTNN($S$)
15.      end-while
16.   else (* $p \neq$ empty *)
17.      (* $2^d$ region searches as shown in Figure 7.6b for a two dimensional case *)
18.      $B'_u \leftarrow B_p \bigcup$ REGIONSEARCH($S$,$L_2^p(u)$,$d_k^p$)
19.      $d_k^u \leftarrow L_2^u(B'_u[k\,])$
20.      (* Search a box of width $d_k^u$ around $u$ *)
21.      $B_u \leftarrow$ REGIONSEARCH($B'_u$, $d_k^u$)
22.   end-if
23.   $d_k^u \leftarrow L_2^u(B'_u[k\,])$
24.   $p \leftarrow u$
25.   $B_p \leftarrow B_u$
```

```
26. end-for
27. return
```

7.5 Experimental Comparison with Other Algorithms

A number of experiments were conducted to evaluate the performance of the *kNN* algorithm. The experiments were performed on a Quad Intel Xeon server running Linux(2.4.2) operating system with one gigabyte of RAM and SCSI hard disks. The datasets used in the evaluation consists of 3D scanned models that are frequently used in computer graphics applications. The three-dimensional point models range from 2k to 50 million points, including two *synthetic* point models of size 37.5 million and 50 million respectively. We developed a toolkit in C++ using STL that implements the *kNN* algorithm. The performance of our algorithm was evaluated by varying a number of parameters that are known to influence its performance. We collected a number of statistics such as the time taken to perform the algorithm, the number of distance computations, the average locality size, page size, cache size, and the resultant number of page faults. The average size of the locality is the average number of blocks in the locality of all points in the dataset.

A good benchmark for evaluating our algorithm is to compare it with a *sorting* algorithm. We make this unintuitive analogy with a sorting algorithm by observing that the work performed by the *kNN* algorithm in a one-dimensional space is similar to sorting a set of real numbers. Consider a dataset S containing n points in a one-dimensional space as an input to a *kNN* algorithm. An efficient *kNN* algorithm would first sort the points in S with respect to their distance to some origin, thereby incurring $O(n \log n)$ distance computations. It would then choose the k closest neighbors to each point in the sorted list, thus, incurring an additional $O(kn)$ distance computations. We point out that it is difficult for any *kNN* algorithm in a higher dimensional space to asymptotically do better than $O(n \log n)$ as the construction of any spatial data structure is, in fact, an implicit sort in a high-dimensional space. We use the *distance sensitivity* [192], defined below,

$$\text{distance sensitivity} = \frac{\text{Total number of distance calculations}}{n \log n}$$

to evaluate the performance of our algorithm. Notice that the denominator of the above equation corresponds to the cost of a sorting algorithm in a one-dimensional space. A reasonable algorithm should have a low, and more importantly, a constant distance sensitivity value.

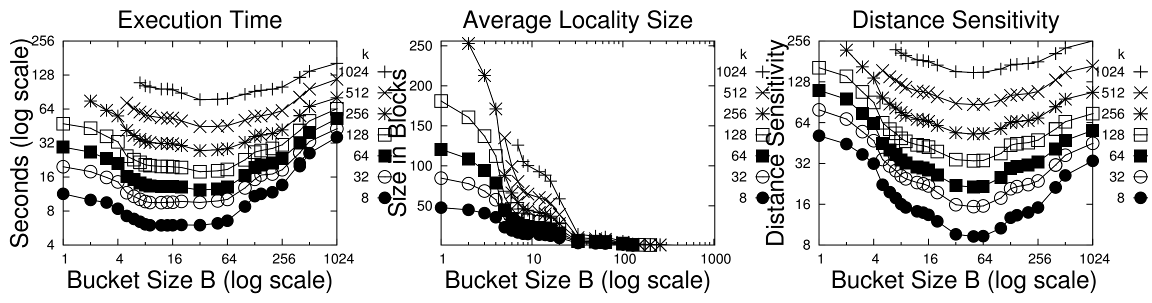
We evaluated our algorithm by comparing the execution time and the distance sensitivity of our algorithm with that of the GORDER method [192] and the MuX method [19]. We also compared our algorithm with traditional methods like the *nested join* [181] and a variant of the BFS algorithm [81]. We use both a bucket PR quadtree [144] and an R-tree [73] variant of the *kNN* algorithm in our study. Our evaluation was in terms of three-dimensional point models as we are primarily interested in databases for computer graphics applications. The applicability of our algorithm to data of even higher dimensionality is a subject for future research. We first discuss the effect of each of the following three variables on the performance of the algorithms.

1. The size of the disk pages which is related to the value of the bucket capacity in the construction of the bucket PR quadtree (Section 7.5.1).
2. The memory cache size (Section 7.5.2).
3. The effect of the size of the data set (Section 7.5.3).

Once we have determined the effect of these variables on the algorithm, we choose appropriate values to compare our algorithm with the other methods in Section 7.5.4.

7.5.1 Effect of Bucket Capacity (*B*)

In this Section, we study the effect of the bucket capacity *B* on the performance of our *kNN* algorithm. The bucket capacity *B* also corresponds to the size of the *disk page*. For a given value of *k* between 8 and 1024, the value of *B* was varied between 1 and 1024. The performance of our algorithm using a bucket PR quadtree was evaluated by measuring the execution time of the algorithm, the average number of blocks in the locality of the leaf blocks, and the resulting *distance sensitivity* of the algorithm. In this set of experiments, we made use of the Stanford Bunny model containing 35,947 points.



(a) (b) (c)

Figure 7.7: Effect of Bucket capacity *B* on the (a) execution time, (b) average size of the locality in blocks, and (c) distance sensitivity for different values of *k* for our *kNN* algorithm.

Figure 7.7a shows the effect of *B* on the execution time of the *kNN* algorithm. Notice that for smaller values of *B* (≤ 16), the *kNN* algorithm has a large execution time. However, it quickly decreases for slightly larger values of *B*. For values of *B* between 32 and 128, our *kNN* algorithm has some of the lowest execution times. Figure 7.7b shows the average number of blocks in the *locality* of the leaf blocks of the bucket PR quadtree. When *B* is small, the size of the locality is large. As a result, for small values of *B* the algorithm has a higher execution time. However, as the value of *B* increases the size of the locality quickly reduces to a small constant value. For larger values of *B*, the increase in execution time can be attributed to a larger number of points stored in the blocks in the locality, even though the number of blocks in the locality remains almost the same. The sensitivity analysis shown in Figure 7.7c is similar to Figure 7.7a. To summarize, the *kNN* algorithm performs well for moderately small values of *B*, and

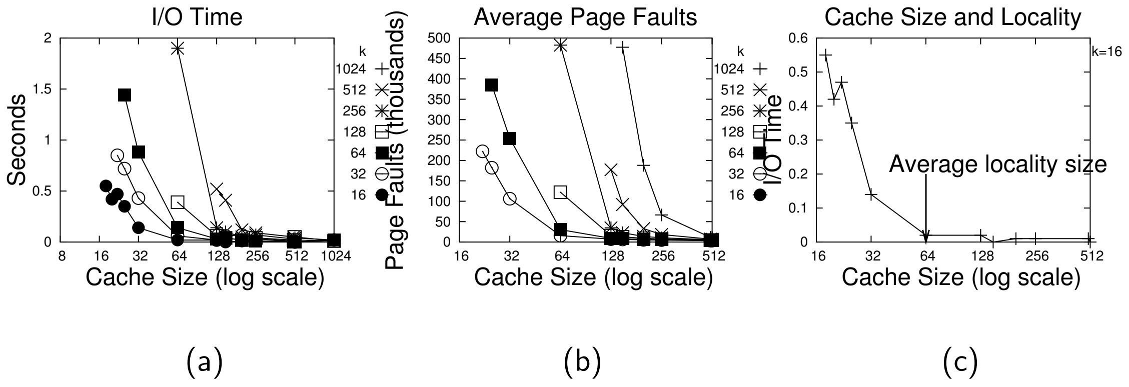


Figure 7.8: Effect of cache size on (a) the time spent on I/O; (b) the number of page faults; for varying values of k and B =32. (c) A comparison between the cache size and the average size of the locality for k =16.

in particular for the range of B between 32 and 128.

7.5.2 Effect of Cache Size

The next set of experiments examines the effect of the *cache size* on the performance of our *kNN* algorithm. The cache size is defined in terms of the number of leaf blocks that can be stored in the main memory. We use a *least recently used* (`LRU`) replacement policy on the disk pages stored in the cache. The size of each memory page is determined by the value of B. We record the effect of the size of the cache on the resulting number of page faults, and the time spent on I/O operations. Figures 7.8a–b shows the result of the experiments for $B = 32$ and for varying values of k ranging between 8 and 1024. We observed high values for the I/O time and the number of page-faults for small ($\leq$ 32) cache sizes, but these values quickly decreased when the cache size was increased

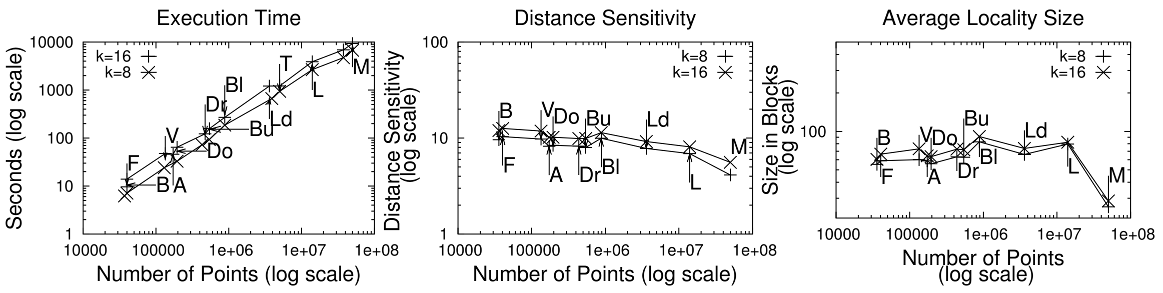
Model Name	Size (millions)	Model Name	Size (millions)
Bunny (B)	0.037	Femme (F)	0.04
Igea (I)	0.13	Dog (Do)	0.195
Dragon (Dr)	0.43	Buddha (Bu)	0.54
Blade (Bl)	0.88	Dragon (Ld)	3.9
Thai (T)	5.0	Lucy (L)	14.0
Syn-38 (S)	37.5	Syn-50 (M)	50.0

Figure 7.9: Pseudo names of the point models and the corresponding number of points (in millions) used in the evaluation.

beyond a certain value. This value, incidentally, corresponds to the average size of the locality, as seen in Figure 7.8c. Moreover, this also explains the occurrence of large number of page faults when the size of the cache is smaller than the size of the locality. The rule of thumb is that the cache size should be at least as large as the average size of the locality.

7.5.3 Effect of Dataset Size

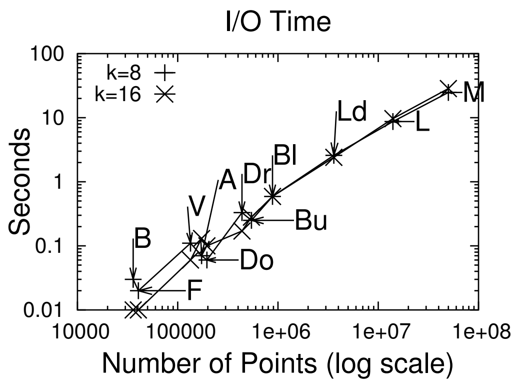
Experiments were also conducted to evaluate the *s*calability of the algorithm as the size of the input dataset is increased. We experimented with several three-dimensional point models ranging in size from 2k to 50 million points as shown in Figure 7.9. The bucket



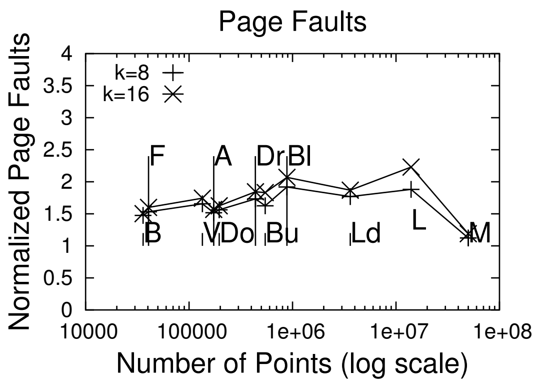
(a) (b) (c)

Figure 7.10: Effect of the *size* of the dataset on (a) execution time, (b) distance sensitivity, and (c) average locality size for various point models with $B = 32$ and 500 blocks in the memory cache.

size B and the cache-size were set to 32 points and 500 blocks, respectively. The results of the experiments are given in Figures 7.10–7.11. Figure 7.10a shows the effect of size on the time taken to perform the *kNN* algorithm. Figure 7.10b records the distance sensitivity of the algorithm. As the distance sensitivity of our approach is almost linear, our algorithm exhibits $O(n \log n)$ behavior. Figure 7.10c records the average size of the locality. We also notice that the average locality size is almost constant for datasets of all sizes used in the evaluation. Also, the size of the locality showed only a slight increase even as the value of k is increased from 8 to 16. The I/O time and the resultant number of page faults are given in Figure 7.11. Figure 7.11a shows the effect of the *size* of the dataset on the time spent by the algorithm on I/O operations. Figure 7.11b shows the number of page faults normalized by size for datasets of various sizes. Both the time spent on I/O and the number of page faults exhibit linear dependence on the size of the



(a)



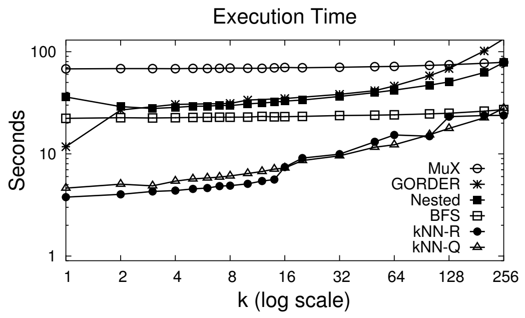
(b)

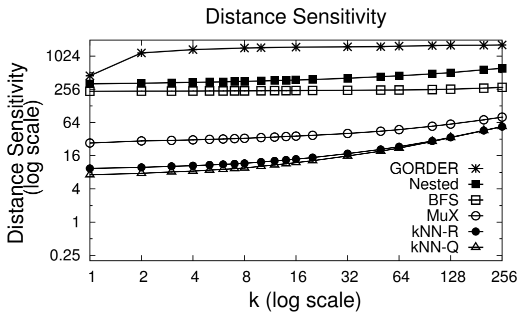
Figure 7.11: Effect of size of the dataset on (a) time spent on I/O, (b) the number of page faults normalized by size for datasets of various sizes.

dataset.

7.5.4 Comparison

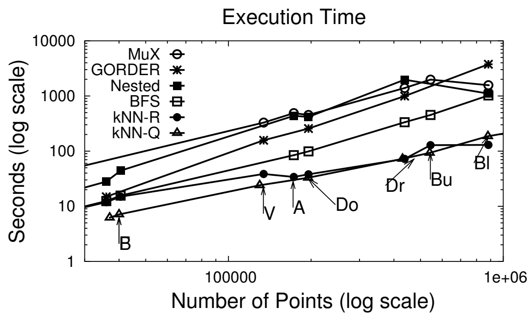
We evaluated our algorithm by comparing its execution time and distance sensitivity with that of the GORDER method [192] and the MuX method of Böhm *et al.* [20]. Our comparison also includes traditional methods like the *nested join* [181] and a variant of the BFS algorithm that invoked a BFS algorithm for each point in the dataset. We used both a bucket PR quadtree and an R-tree variant of the algorithm in the comparative study. The R-tree implementation of our algorithm used a *packed* R-tree [136] with a bucket-capacity of 32 and a *branching factor* of 8. Note however, that the values of *B* and *k* are chosen independent of each other. We retained 10% of the disk pages in the main memory using a LRU based page replacement policy. For the GORDER

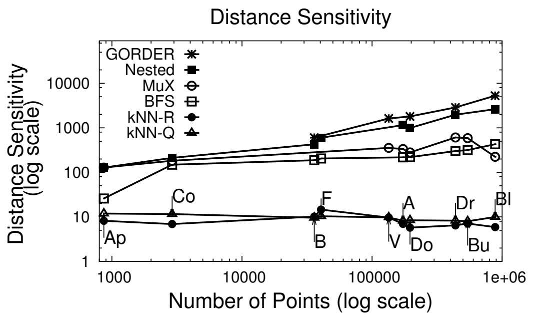




(a) (b)

Figure 7.12: Performance comparison of our *kNN* algorithm with the BFS, GORDER, MuX and the Nested join algorithms. 'kNN-Q' and 'kNN-R' refers to the quadtree and R-tree implementations of our algorithm respectively. Plots a–b show the performance of the techniques on the *Stanford Bunny* model containing 35,947 points for values of *k* ranging between 1 and 256; (a) execution time, and (b) distance sensitivity.





(a) (b)

Figure 7.13: Performance comparison of our *kNN* algorithm with the BFS, GORDER, MuX and the Nested join algorithms. 'kNN-Q' and 'kNN-R' refers to the quadtree and R-tree implementations of our algorithm, respectively. Plots a–b record the performance of all the techniques on datasets of various sizes for $k = 8$; (a) execution time, and (b) distance sensitivity.

algorithm, we used the parameter values that led to its best performance, according to its developers [192]. In particular, the size of a sub-segment was chosen to be 1000, the number of grids were set to 100, and the size of the data set buffers was chosen to be more than 10% of the data set size. For the MuX-based method, a page capacity of 100 buckets and a bucket capacity of 1024 points was adopted. There are a few differences between the MuX method as described in [20] and our implementation. In particular, we adapted our implementation into a three level structure structure with a set of *hosting pages* where each page contains several buckets with pointers to a disk-based store. Also, we did not use a *fractionated* priority-queue as described in [20] but replaced it with a heap-based priority queue. However, we did not take into the account the time taken to manipulate the heap structure, thereby ensuring that these differences in the implementation do not affect the comparison results. Also, we only count the point-point distance computations in determining distance-sensitivity and disregard all other distance computations even though they form a substantial fraction of the execution time. We used a bucket capacity of 1024 for the BFS and nested join [181] methods. The results of our experiments were as follows.

1. Our algorithm clearly out-performs all the other methods for all values of k on the Stanford Bunny model as shown in Figure 7.12a–b. Our algorithm leads to at least an order of magnitude improvement in the distance sensitivity over the MuX, the GORDER, the BFS and the nested join techniques for smaller values of k (≤ 32)

and at least 50% improvement for larger k (< 256) as seen in Figure 7.12b. We observed an improvement of at least 50% in the execution time (Figure 7.12a) over the competing methods.

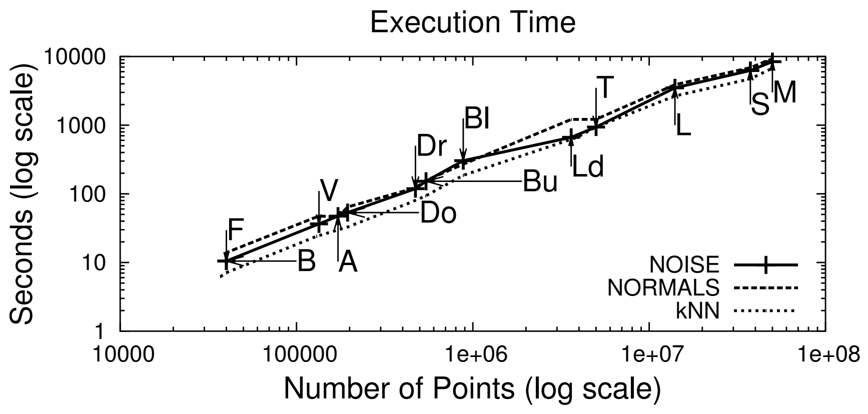
2. However, as size of the input dataset is increased the performance of the MuX algorithm was comparable to the nested, BFS and the GORDER based methods (Figure 7.13a). Moreover, our method has an almost constant distance sensitivity even for large datasets. The distance sensitivity of the comparative algorithms are at least an order of magnitude higher for smaller datasets and up to several orders of magnitude higher for the larger datasets in comparison to our method (Figure 7.13b). We observed similar execution time speedups as seen in Figure 7.13a.
3. Figure 7.13 shows similar performance for the R-tree and the quadtree variants of our algorithm.

7.6 Applications

Having established that our algorithm performed better than the GORDER and MuX methods, we next evaluated the use of our algorithm in a number of applications for different data sets that included both publicly available and synthetically generated point-cloud models. The size of the models ranged from 35,947 points (*Stanford Bunny* model) to 50 million points (*Syn-50* model). These applications include computing the

Model Name	Size (millions)	*kNN*	Surface Normals	Noise Removal
Bunny (Bu)	0.037	6.22	9.0	9.64
Femme (F)	0.04	7.13	10.5	13.9
Igea (I)	0.13	24.05	36.6	47.52
Dog (Do)	0.195	32.9	53.4	64.45
Dragon (Dr)	0.43	72.62	118.9	122.2
Buddha (Bu)	0.54	93.04	152.3	157.25
Blade (Bl)	0.88	185.92	304.2	270.0
Dragon (Ld)	3.9	663.84	900.0	1209.8
Thai (T)	5.0	940.04	1240.0	1215.7
Lucy (L)	14.0	2657.9	3504.0	3877.78
Syn-38 (S)	37.5	4741.79	-	-
Syn-50 (M)	50.0	6427.5	-	-

(a)



(b)

Figure 7.14: (a) Tabular and (b) graphical views of the execution time of the *kNN* algorithm for different point models, and the time to execute a number of operations (i.e., normal computation and noise removal) using it. All results

surface normals to each point in the point-cloud using a variant of the algorithm by Hoppe *et al.* [84] and removing noise from the point surface using a variant of the *bilateral filtering* method [47,96]. Figure 7.14 shows the time needed for these applications when incorporating an algorithm with a neighborhood of size $k = 8$ for each point in the point-cloud model. Figure 7.14b shows that our algorithm results in *scalable* performance even as the size of the dataset is increased so that it exceeds the amount of available physical memory in the computer by several orders of magnitude. The scalable nature of our approach is readily apparent from the almost uniform rate of finding the neighborhoods, i.e., 5900 neighborhoods/second for the Stanford Bunny model and 7779 neighborhoods/second for the Syn-50 point-cloud models.

In the rest of this Section, we describe in greater detail how our algorithm can be used in these computer graphics applications, and give a qualitative evaluation of its use. In particular, we discuss its use in computing surface normals (Section 7.6.1), noise removal through mollification of surface normals and bilateral mesh filtering (Section 7.6.2), as well as briefly mentioning additional related applications (Section 7.6.3).

7.6.1 Computing Surface Normals

Point-cloud models are distinguished from other models by not containing any topological information. Thus, one of the initial preprocessing steps required before the point-cloud model can be successfully used is to compute the surface normal for each

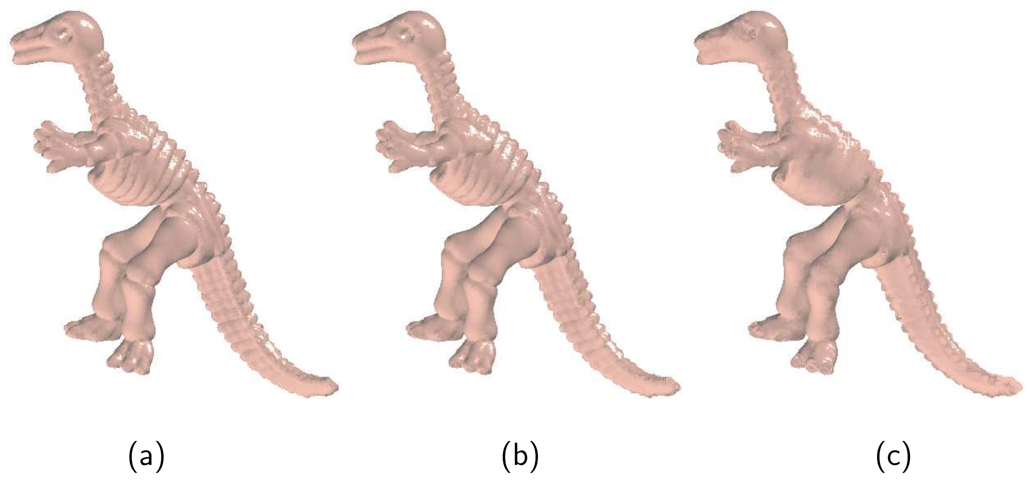


Figure 7.15: Dinosaur point-cloud models displayed using surface normals computed with neighborhoods of (a) 16, (b) 64, and (c) 128 neighbors.

point in the model. Computing the surface normal is important for the proper display and rendering of point-cloud data. Using the surface normal information, other topological features of a point surface can be estimated. For example, we can estimate the presence of sharp corners on the point-cloud models with reasonable certainty. A sudden large deviation in the orientation of the surface normals within a small spatial distance may indicate the presence of a sharp corner. Many such local surface properties can be estimated by examining the surface normals and the neighborhood information.

One of the most prominent methods for computing surface normals for unorganized points is due to Hoppe *et al.* [84]. This method relies on computing the *k* nearest neighbors to each point in the dataset. The neighborhood is *fit* with a *hypothetical surface* which minimizes the sum of the squared distances from each point in the neighborhood to the hypothetical surface. A *covariance analysis* of the resulting neighborhood leads to the estimation of the normals to the surface and the query-point.

A more recent contribution is by Mitra *et al.* [117] which deals with the computation of the surface normals to a point-cloud in the presence of noise. This algorithm computes the neighborhood of points in the dataset after taking into consideration the sampling density and the curvature of the neighborhood. There is also the alternative approach of Floater and Reimers [48] that *triangulates* the neighborhood and computes the surface normals from the resulting mesh surface.

The neighborhood finding algorithms used in these methods are as diverse as the methods themselves. The algorithm by Hoppe [84] assumes a uniform sampling of points in the point-cloud. This makes the computation of neighborhood almost trivial, although not realistic. Also, many algorithms use either an approximate brute-force method or compute the neighborhood by repeated computation of the neighborhood for one point at a time (e.g., see [117]).

We computed the surface normal information of several datasets using a method similar to that of Hoppe *et al.* [84]. We tabulated the time taken for datasets of different sizes and also recorded the effect of varying the size of the neighborhood on the resulting neighborhood calculation. The effect of varying the size of the dataset when computing the surface normals is given by the appropriately-labeled column in Figure 7.14a. The main results of using our algorithm to compute surface normals are as follows.

1. The *quality* of the surface normals depends on the size of the neighborhood as

can be seen in Figure 7.15. Using the surface normals for $8 \leq k \leq 64$ retains the finer details on the surface (Figures 7.15a–b). Using a larger neighborhood such as $k \geq 64$ leads to a loss of many of the finer surface details (Figure 7.15c). This effect can be attributed to the *averaging* property of the neighborhood.

2. When dealing with noisy meshes, the surface normals computed using the topological information of the mesh are often erroneous as can be seen in the dragon model in Figure 7.16a. In such cases, we can use our *kNN* algorithm to compute the surface normals by just using the neighborhood of the points and the result is relatively error-free as seen in Figure 7.16b when using 8 neighbors. This leads us to observe that correct surface normals are important for the proper display of the point model, and that the normals computed by analyzing the neighborhood are resilient to noise, but result in a loss in surface details if an unsuitable value of k is used as seen in Figure 7.15c.

7.6.2 Noise Removal

With advances in scanning technologies, many objects are being scan converted into point-clouds. The objects are scanned at a high resolution in order to capture the surface details and to provide an illusion of a smooth compact surface by the close placement of the points comprising the point-cloud model. However, in reality, points in a freshly scanned point-cloud model are noisy due to environmental interference, material prop-

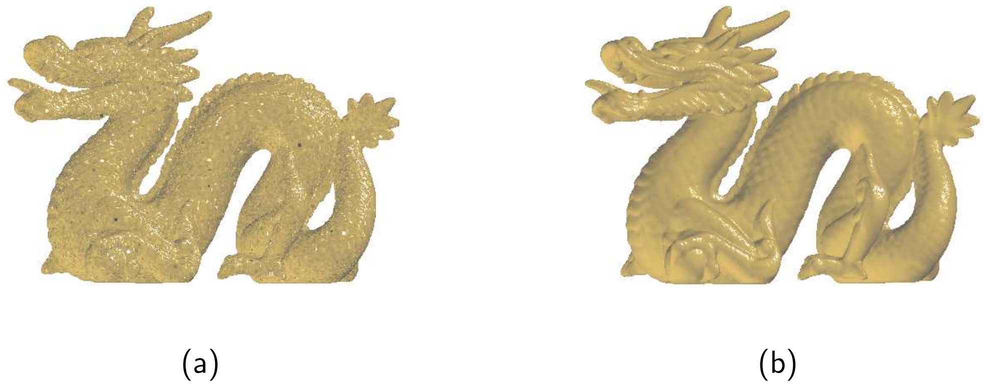


Figure 7.16: (a) A noisy mesh-model of a dragon, and (b) the corresponding model whose surface normals were recomputed using our *kNN* algorithm. The algorithm took about 118 seconds and used 8 neighbors.

erties of the scanned object, and calibration issues with the scanning device. Often, an additional corrective procedure needs to be performed in order to account for the residual noise before the model can be successfully employed. In fact, such an unprocessed point-cloud model would have a *scarred appearance* as illustrated in Figure 7.16a which has been obtained by adding a noise element to each of the points in the original model.

Noise is removed by applying a filtering algorithm to the points in the point-cloud model. *Bilateral mesh filtering* [47, 96] and *mollification* [120] are two prominent techniques for removing noise from a mesh. While the bilateral mesh filtering algorithm attempts to correct the position of erroneous points, the mollification approach, instead, tries to correct the surface normals at the point. Bilateral mesh filtering is analogous to *displacement mapping* [38] and mollification is analogous to *bump mapping* [18], both of which are prominent texturing techniques that can be used to achieve the same result. In particular, displacement mapping relies on shifting the points themselves to bring about texturing of the surface, while bump-mapping modifies the surface normals at each vertex of the mesh surface. Bilateral filtering differs from another class of techniques, that include MLS noise removal [5], which correct the points by reconstructing a smooth local surface and re-sampling points from the surface. In the rest of this Section, we discuss the results of our application of both bilateral mesh filtering and mollification to remove noise in large point-cloud models.

We applied the bilateral mesh filtering algorithm in [47,96] to the point-cloud model as follows. We initially computed a neighborhood for each point in the model. Our adaptation of the bilateral filtering method assigns weights (an influence measure analogous to the Gaussian weights in the bilateral filtering method) to each point in the neighborhood in such a way that the computation becomes less *sensitive* to outlier points. Note that mollification corrects the normals instead of the point, but is similar in approach. Figure 7.17 shows the results of applying our point-cloud model adaptation of the conventional bilateral mesh filtering algorithm to the bunny model (35,947 points) for different pairs of values of the Gaussian kernel. Notice that the quality of the results when using our adaptation does not depend on the values of the Gaussian kernel.

As pointed out earlier, mollification is similar to bilateral mesh filtering with the difference being that instead of performing the filtering operation on the points, the filtering operation is applied to the original surface normals of the points. In order to

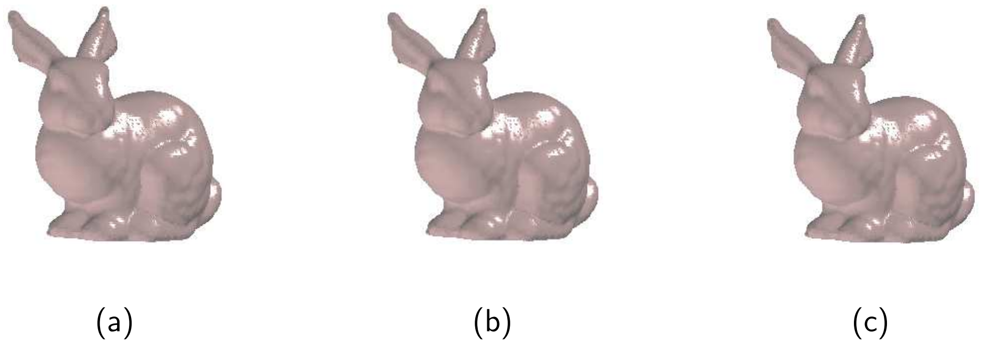
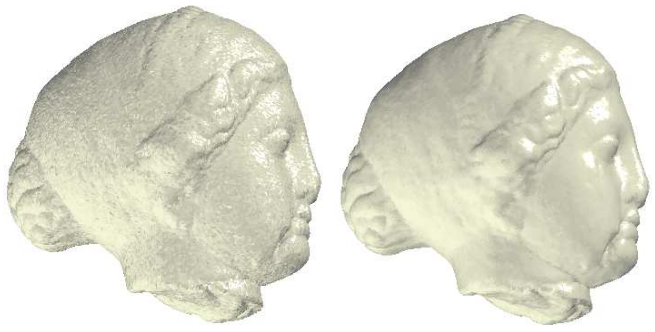
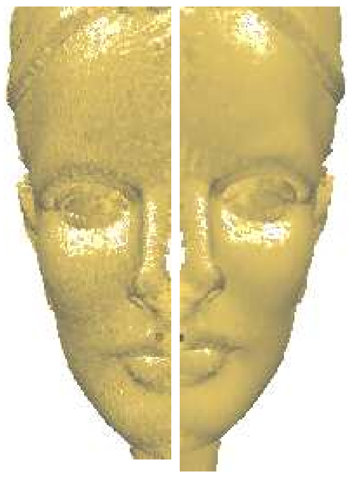


Figure 7.17: Results of applying the neighborhood-based adaptation of the bilateral mesh filtering algorithm to the bunny model for Gaussian kernel pairs (a) $\sigma_f = 2, \sigma_g = 0.2$, (b) $\sigma_f = 4, \sigma_g = 4$, and (c) $\sigma_f = 10, \sigma_g = 10$ for a neighborhood of size 8. The results are independent of the size of the Gaussian kernel that was chosen.

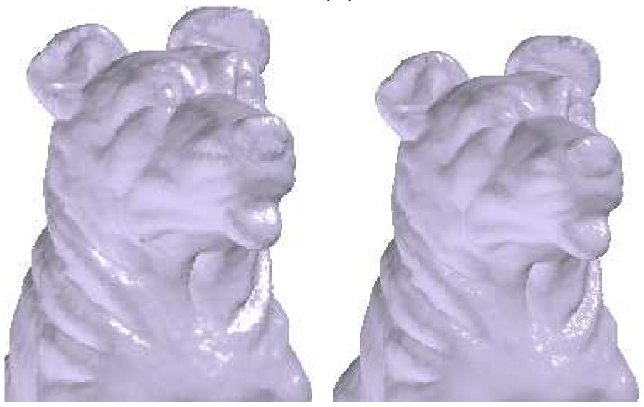
evaluate the sensitivity of our filtering and surface computation methods to noise, we added Gaussian noise using the Box-Muller method [189] to a bunny mesh-model. We computed the surface normals at each vertex in the noisy mesh using the connectivity information contained in the mesh. The resultant mesh, disregarding the connectivity information, is a point-cloud (as shown in Figure 7.19a) with noisy point positions and *noise-corrupted* normals. We use this approach to create the noisy point-clouds used in Figures 7.18 and 7.19. Figures 7.19b–d compare the result of using the mollification method (Figure 7.19d) with the computation of surface normals as in Section 5.1 (Figure 7.19b) and our adaptation of the bilateral mesh filtering method (Figure 7.19c). All three methods were applied for 8 neighbors. From the figure, we see that when using our *k* nearest neighbor method to compute the neighborhoods to be used in computing the surface normals, there is no perceptible difference between the three methods even



(a)



(c)



(b)

Figure 7.18: Three noisy models which were de-noised using filtering and mollification techniques. In the pairs of figures shown for each of the models, the figure on the left is the noisy model, while the figure on the right is the corrected point model. The (a) *Igea* and (b) *dog* models were denoised with the filtering method, while the (c) *femme* model was denoised using the mollification technique.

in the case of noisy data.

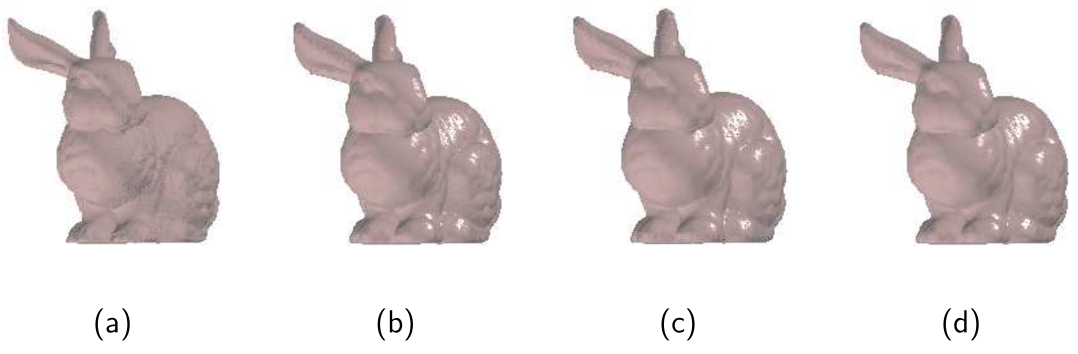


Figure 7.19: (a) A bunny point-cloud model to which Gaussian noise was added, and the result of applying (b) the surface normal computation method in Section 5.1, (c) our adaptation of bilateral mesh filtering, and (d) mollification.

7.6.3 Related Applications

The most obvious application of the *kNN* algorithm is in the construction of *kNN* graphs [161]. *kNN* graphs are useful when repeated nearest neighbor queries need to be performed on a dataset. The *kNN* algorithm may also be used in *point reaction-diffusion* [180] algorithms. Such algorithms mimic a physical phenomenon to uniformly distribute points on a given surface or space. Many of natural texture patterns encountered in nature can be recreated using this technique. The algorithm works as follows. Each point is assigned a unit positive charge. The resultant repulsion force acting on the point is computed using the *k* nearest neighbors at each point. Next, the point is moved along the direction of the force, and the *kNN* algorithm is repeatedly reinvoked at each iteration until an equilibrium condition is reached.

A recent contribution in the construction of approximate surfaces from point sets is the moving least squares (MLS) [5] method. *et al.* [190] have identified useful point-cloud operations that use the MLS method. Of these operations, we believe that MLS *point-relaxation*, MLS *smoothing*, MLS based *upscaling* [5], and *downscaling* can all benefit when used in conjunction with the *kNN* algorithm.

Tools that perform *upscaling* [59, 128] and downscaling [5] of point-clouds all use the *kNN* algorithm to generate varied *levels of detail(LOD)* [113] of point models. The *quadratic error simplification* method [59, 128] simplifies a point-cloud by removing the points that make the least significant contribution to the surface details.

Figure 7.20: (a) Initial apple model (867 points) and (b) the result of applying an upscaling algorithm to it using the *kNN* algorithm (27,547 points).

A similar method to increase the point sampling uses a variant of MLS [5] to insert additional points in the neighborhood (termed *upscaling*). The algorithm computes the *k* nearest neighbors to each point using the *kNN* algorithm. Points are then evenly distributed [112] on the hypothetical surface that is fit through the points in the neighborhood. We built a variant of the algorithm which when applied to the *apple* model (Figure 7.20a) containing 867 points resulted in a new point model containing 27,547 points (Figure 7.20b) which took about 1.2 seconds to construct.

7.7 MAP-REDUCE Neighborhood Algorithm for Point-Clouds

Google uses a computing technology, dubbed *Cloud Computing* in layman parlance, that makes use of hundreds of thousands of machines in order to handle the mammoth computational task of building an index of all the documents in the Internet. Now, consider an emerging computational problem in the computer graphics domain. Laser scanning devices are making it possible to scan larger and larger artifacts into 3D point-clouds [104]. Soon it may even be possible to scan entire cities, and large terrains into point-clouds, in which case, we have a computational problem that is as daunting as processing all the documents in the Internet. Some of the existing 3D point-clouds are large enough that performing operations on them using a single machine, or an adhoc cluster of machines takes an unrealistic amount of time. For example, we estimate (see Section 7.7.4) that a sequential point operation on a point-cloud containing a billion points would take more than 36 hours. In this chapter, we propose the use of the MAP-REDUCE framework [39] which is a programming framework developed by Google in order to perform large parallel tasks on massive documents datasets. The MAP-REDUCE framework represents a scalable, robust, fault tolerant, computing platform to handle large computationally demanding operations such as geometric algorithms on massive point-clouds.

From the time a point-cloud is scanned and before it can be rendered, a number of computationally expensive operations [67, 196] need to be performed on it. This includes the computation of *surface normals*, *noise filtering*, alterations to the *sampling rate*, removing *outliers*, *hole-filling*, and *surface* reconstruction of the point-cloud. What is common to all these operations is that they all involve operations on the *neighborhood* of points. The neighborhood of a point p is the set of points in the dataset that are close to p. The neighborhood algorithm is important to point-cloud processing and forms the basis for several other operations on 3D point-clouds. Therefore, the main contribution of this chapter is opening up the possibility of making MAP-REDUCE framework a standard part of the pipeline in performing offline point-cloud operations, such as in PointShop 3D [196], as well as visualization of scientific data.

Problem Statement: Given a set S of unorganized points in $[0,1)^d$, $d = 3$ in our case, we wish to find for every point $p \in S$ all other points in S that are within a distance of $\varepsilon \in [0,1)$ from p, where ε is a small value. Given two points p, q in S, let $D_E(p,q)$ be the Euclidean distance between p and q. Note that for point-clouds where the sampling density of the dataset is known, the value of ε is chosen such that the neighborhood contains at least $k > 0$ points. So, specifying neighborhood in terms of ε, or k is similar for such points sets.

The rest of the section is organized as follows. In Section 7.7.1, we review the workings of the MAP-REDUCE framework. Section 7.7.2 presents a broad outline of the neighborhood algorithm on point-clouds, while the actual algorithm is described in Section 7.7.3. Finally, Experimental results are given in Section 7.7.4.

7.7.1 Map Reduce Framework

MAP-REDUCE [39] is a scalable, parallel programming framework that can process large datasets on large clusters containing hundreds, possibly even thousands of machines. MAP-REDUCE's framework provides little or no synchronization, locking, or inter process communication mechanisms, which makes for a restrictive parallel environment. Programmers have to write programs that do not require any synchronization mechanisms, which can be a challenge. On the other hand, owing to these restrictions, the resulting programs that run on this framework are simple, intuitive, and are able to fully exploit the available parallelism without incurring the overhead of supporting synchronization.

A program that works on the framework spawns two kind of functions – MAP and REDUCE, which are then scheduled in run in parallel on the machines in the cluster. Let us assume that the input to the framework is a set of *records*. A program first spawns *m* MAP and *r* REDUCE processes. A MAP process takes a record as input, and outputs one or more *key-value* pairs. After all the *m* MAP processes have executed in parallel, the framework sorts the output *key-value* pairs from all the m MAP processes, and *aggregates* them based on equality of the keys. That is, if k_1 is a key in the output, then all values in the output such that *key* = k_1 are added to a set S_{k_1}. A REDUCE process receives a *key*, say k_1, and the corresponding set of *values* S_{k_1} as inputs. The output of the REDUCE processes is written to disk.

7.7.2 Outline of an Algorithm

A naive approach to computing the neighborhood of a point-cloud is to compute the neighborhood of each point, one point at a time. An alternative, and more sophisticated, approach [157, 158] computes the neighborhood of a point p, say $N(p)$, and then reuses $N(p)$ to compute the neighborhood of other spatially proximate points q of p in the dataset. Although, this alternative results in better *work efficiency* compared to the naive algorithm, such an approach is inherently sequential and thus may not be suitable for a parallel framework. Our proposed algorithm uses the MAP-REDUCE framework to yield a two step solution that is based on imposing a grid G^l on the dataset. First, for each point in the dataset, it computes in parallel the set of grid cells in G^l that can potentially contain points belonging to $N(p)$, the neighborhood of p. This is the MAP step of the framework. Next it computes the *inverse* relation so that for any grid cell g we obtain the set of points S_g for which g serves as a neighborhood, whether completely or partially (i.e., the neighborhood of elements of S_g has a nonempty intersection with

g). The neighborhood of the points contained in g can now be computed by examining the points in S_g. This is the REDUCE step of the framework. The key to this process is a hidden implicit intermediate step that occurs between the MAP and REDUCE steps which aggregates all of the elements resulting from the MAP step that share the same grid cell, which is achieved by a distributed sorting algorithm.

The above results in a simple, yet elegant, algorithm for computing the neighborhood of all points in a dataset that makes use of the MAP-REDUCE framework. It is interesting to observe that, in essence, what the MAP-REDUCE framework accomplishes is provide us the ability to construct an associative memory where we associate grid cells with points and then retrieve all points associated with a grid cell to speed up the neighborhood calculation. It is important to note that the ability to parallelize the sorting step via the distribution process is what gives the MAP-REDUCE framework its power and generality as this is what enables it to be applied to many different problem domains as long as they involve some sorting, which is often the case.

7.7.3 Neighborhood Algorithm

We define a multidimensional regular grid G^l on $[0,1)^d$ such that G^l partitions the space into $c = 2^{ld}$ *cells*. Given a 3D point-cloud S on $[0,1)^d$ and the value of ε, we can compute $l_0 \in \mathbb{Z}$ such that $2^{-l_0-1} \leq \varepsilon < 2^{-l_0}$. Given an ε and G^l, stipulating that the side length 2^{-l} of the grid cells in G^l lies in the range $2^{-l_0} \leq 2^{-l} \leq 2^0$ means that the search region defined by ε intersects (overlaps) at most 2 grid cells along each of the d dimensions. We refer to l as the *level* of the grid G^l. Additionally, we impose a one-dimensional ordering on the cells such that each cell has a uniquely identifiable address (or code). Such a one-dimensional ordering on the cells is imposed by the use of a *space filling curve* on the centroids of the cells in G^l. In our case, we use a Morton or Z-ordering on the centroids of the cells. Let G_i be the ith, $0 \leq i < c$, cell in the ordering of the cells in G^l.

Let $R^{\varepsilon}(p)$ be a search region of radius $\varepsilon/2$ around $p \in S$. Given $R^{\varepsilon}(p)$, let $G^l(R^{\varepsilon}(p)) = \{G_{p_1}, G_{p_2}, ..., G_{p_i}, ..., G_{p_t}\}$, be a set of t grid cells in G^l such that they completely *cover* $R^{\varepsilon}(p)$. Another property of $G^l(R^{\varepsilon}(p))$ is that it is minimal in the sense that the cells are unique, and only those cells that intersect $R^{\varepsilon}(p)$ are in $G^l(R^{\varepsilon}(p))$. Note that t depends the position of p as well as the value of l. Notwithstanding the position of p, t can be as large as $O(2^d)$ for $l = l_0$, and as small as one for $l = 0$.

Our neighborhood algorithm consists of a MAP process, given in Algorithm 18, and a REDUCE process, given in Algorithm 19. At the start of the algorithm, the user specifies the size n of the dataset, the value of ε and the value of l. The framework first invokes m instances of the MAP process in parallel, which is followed by r instances of the REDUCE process again in parallel. Both m and r are dependent on n and ε. A MAP process takes a point p from the dataset as input. It then computes the set $G^l(R^{\varepsilon}(p))$ of cells in G^l that intersect $R^{\varepsilon}(p)$ as shown in line 1. For each cell G_{p_i} in $G^l(R^{\varepsilon}(p))$, the algorithm outputs a *key-value* pair, where G_{p_i} is the *key* and p is the *value*.

Algorithm 18

```
Procedure MAP[$p$]
Input: $p$ is an input point
Output: set of key-value pairs
1.  Compute $G^l(R^\varepsilon(p))$
2.  for each $G_{p_i} \in G^l(R^\varepsilon(p))$ do
3.      Output $<key = G_{p_i}, value = p>$
4.  end-for
```

When all the MAP processes have finished execution, the *key-value* pairs are sorted and then the values associated with each distinct key are aggregated to serve as the key's value. In other words, all the values (points) corresponding to a key (cell G_u) in the output are aggregated and form the input to the REDUCE processes.

The REDUCE algorithm is invoked for every unique key in the output of the MAP process. The input to a REDUCE algorithm is a key (cell), say G_u and the set of values (points) S_{G_u} associated with G_u. We examine every pair of points p_i and p_j in S_{G_u}, and check if either p_i or p_j is contained in G_u. If so, we check if $D_E(p_i, p_j)$ is less than or equal to ε. In this case, we output the pair $<p_i, p_j>$ if p_i is contained in G_u. In addition, we also output the pair $<p_j, p_i>$ if p_j is contained in G_u. Note that our examination of the containment of both p_i and p_j in G_u ensures that every pair of points $<p_i, p_j>$, within ε of each other are reported only twice in the output; once as $<p_i, p_j>$ and again as $<p_j, p_i>$. In other words, our algorithm does not report duplicate pairs in the output. Note that when both p_i and p_j are not contained in G_u, there is no need to further examine the pairs as the REDUCE process associated with G_u only computes the neighborhood of the points contained in G_u.

Algorithm 19

Procedure REDUCE[key = G_u, value = S_{G_u}]

Input: $S_{G_u} \leftarrow$ set of points in G_u

```
1.   for each pair of points $(p_i, p_j) \in S_{G_u}, i < j$, do
2.       if ($p_i$ or $p_j$ is contained in $G_u$) then
3.           if $D_E(p_i, p_j) \leq \varepsilon$ then
4.               if ($p_i$ is contained in $G_u$) then
5.                   Output $<p_i, p_j>$
6.               end-if
7.               if ($p_j$ is contained in $G_u$) then
8.                   Output $<p_j, p_i>$
9.               end-if
10.          end-if
```

```
11.         end-if
12.     end-for
```

An advantage of our method is that for any given point p, all the points in the neighborhood of p are found when executing the REDUCE process associated with the cell containing p. This means that the REDUCE algorithm can be easily modified to perform a variety of point operations on p and the neighborhood of p.

Cost Model for Choosing l: Given a point p, the number of cells intersected by $R^{\varepsilon}(p)$ in G^l depends on the value of l as well as on the position of p. Note that while the correctness of the algorithm does not depend on the value of l, the performance of the algorithm does depend on l. As mentioned earlier, the value of l can lie between 0 and l_0. If $l = l_0$, the average number of cells in G^l intersected by $R^{\varepsilon}(p)$ can be as large as 2^d because the side length 2^{-l_0} is not much larger than ε. For values of $l < l_0$, we show below that the average number of cells intersected by $R^{\varepsilon}(p)$ decreases significantly. This can be explained by observing that the cells in G^l, when $l < l_0$, are larger than the cells in G^{l_0}. However, this also means that each REDUCE process, on the average, performs more work since the expected number of points per grid cell increases as the grid cell gets larger.

In the following analysis, we assume that the points in the dataset are drawn from a uniform distribution. We first consider G^l in a one-dimensional space. We determine the probability that a one-dimensional range of length ε ($R^{\varepsilon}(.)$ in one dimension is a range of length ε) intersects one of the grid lines which are at positions $a2^{-l}$, where a is an integer. The probability that $R^{\varepsilon}(.)$ intersects a grid line is given by $p = \left(\frac{\varepsilon}{2^{-l}}\right)$. We now consider the general d-dimensional case. The probability that $R^{\varepsilon}(.)$ intersects exactly k of the d grid lines that meet at a *corner* of a grid cell is given by $\begin{pmatrix} d \\ \\ k \end{pmatrix} p^k(1-p)^{d-k}$. Note that if $R^{\varepsilon}(p)$ interests k grid lines at level l, then $R^{\varepsilon}(p)$ intersects exactly 2^k cells in G^l. For any given level l, we can now compute the expected number of grid lines E that $R^{\varepsilon}(.)$ intersects. This is given by $E = \sum_{i=0}^{d} 2^i \begin{pmatrix} d \\ \\ i \end{pmatrix} p^i(1-p)^{d-i}$. In three-dimensions, we have:

$$E = (1-p)^3 + 6p(1-p)^2 + 12p^2(1-p) + 8p^3. \tag{7.3}$$

Multiplying E by n, the number of points in the dataset yields the expected total number of key-value pairs produced by all the MAP processes. E, also called the *replication factor*, is the average number of key-value pairs produced per point in the dataset. Our goal is to reduce the *replication factor*.

Given that the total number of cells is 2^{dl}, the expected number of points per cell in the REDUCE process is $P = 2^{-dl}En$. The expected number of distance computations performed per cell is P^2 and thus the expected total number of distance computations performed by the algorithm is given by $D = 2^{dl}P^2 = 2^{-dl}(En)^2$. The MAP-REDUCE framework uses a distributed merge sort in order to aggregate all the output keys from the MAP processes. The cost of this sort for the En key-value pairs is expected to be

$L = En\log(En)$. Therefore, the expected total cost C of the algorithm is the sum of the work to sort the key-value pairs and the expected number of distance computations. That is —

$$C = c_1 En\log(En) + c_2 2^{-dl}(En)^2, \tag{7.4}$$

where c_1 and c_2 are constants of proportionality for a cluster configuration that can be estimated empirically. Thus given values for ε and n, we use Equation 7.4 to compute the expected total cost C of the algorithm for different values of l between 0 and l_0 and choose the value of l that minimizes C.

7.7.4 Experiments

Since not everyone has access to Google's half a million plus servers, we report the results of experiments on a cluster size of 20 as this corresponds to what can be easily assembled by two to three people assuming individual workstations with 4 to 8 cores. In order to enable replication of our results, we used an open source implementation of the MAP-REDUCE Framework called *Hadoop* (`http://lucene.apache.org/hadoop`). Our cluster was configured to run two MAP and one REDUCE processes on each of the 20 hosts, thereby providing a parallelism factor of roughly 40 to the problem. The largest point-cloud in our experimental setup contained two billion 3D points, while the smallest contained 10 million points. For the point-clouds used in the evaluation, we had a good estimate of their sampling density. This means that the neighborhood algorithm specified in terms of the number of neighbors k per point of the dataset is equivalent to a neighborhood specified in terms of ε. As the former case is more intuitive, in the rest of the section, we specify the neighborhood size in terms of k, while the implementation of the algorithm uses an equivalent value of ε.

We first examine the effect of the size of the dataset n on the performance of our algorithm. We use a variety of datasets, the smallest contained 10 million points while the largest contained over two billion points. Figure 7.21a–c shows the effect of n on the performance of our algorithm, for a neighborhood of size $k = 8$. From Figure 7.21a we see that the time gracefully increases with n. In order to get a sense of the speedup here, we estimate that the serial kNN algorithm [157] would take more than 72 hours to process the point-cloud containing two billion points; a task our algorithm could accomplish in little more than 3.5 hours. Another comparison measure is the average number of the neighborhood computations per hour. The result of the comparison is shown in Figure 7.21b. Our algorithm has a peak throughput rate of 900 million neighborhoods/per hour, compared to 21.3 million neighborhoods/per hour of the kNN algorithm. The average throughput of our algorithm is around 700 million neighborhoods/hour.

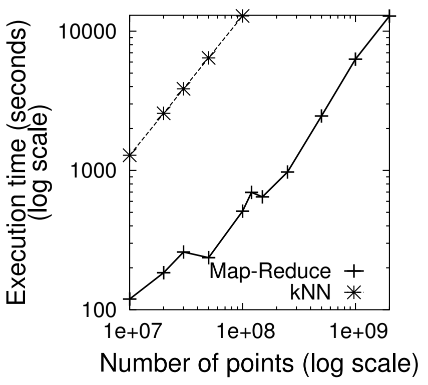
Our algorithm's replication factor, shown in Figure 7.21c, is almost independent of n. Using the cost model of Equation 7.4 to choose an appropriate value of l enables keeping the replication factor low, between 1 and 2 for any arbitrary point-cloud size, thereby reducing the cost contribution of the distributed sorting component and the average number of distance computations per cell.

Finally, Figure 7.21d shows the effect of varying the size of the neighborhood k between 4 and 300 on the performance of our algorithm. For this experiment, we used a point-cloud containing 50 million points. From the Figure we see that the time taken by our algorithm increases gracefully as the size of the neighborhood is increased. Thus, our algorithm is also suitable for point-cloud operations where large neighborhood sizes are needed.

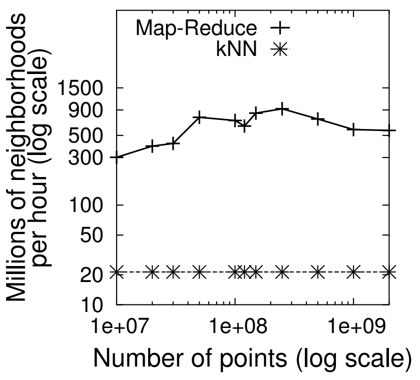
7.8 Summary

Points are becoming the modeling primitive of choice for graphics applications. Current and future applications will create point-clouds that are too large to be processed on a single machine. In this chapter, we presented a neighborhood algorithm for point-clouds that computes the k nearest neighbors of each point in a point-cloud. Substantial cost savings was achieved by reusing the nearest neighbor of a point in the nearest neighbor computation of another proximate point. This resulted in an algorithm that is work efficient, and is scalable to large point-cloud datasets. Moreover, given a set s of points, we define the *neighborhood* r_s of s as a region containing the k nearest neighbors of all the points in s. We are able to show that the size of r_s constructed by our algorithm is *optimal*. That it, is not possible to construct a smaller neighborhood than the one constructed by our algorithm. We presented another variation of the algorithm can obtain neighbors of points in a given point-cloud S in an incremental fashion. That is, the algorithm computes k nearest neighbors of each point in S, but can produce as many as additional neighbors as necessary by expending more work. We showed that our algorithm is scalable and can be applied to large point-clouds.

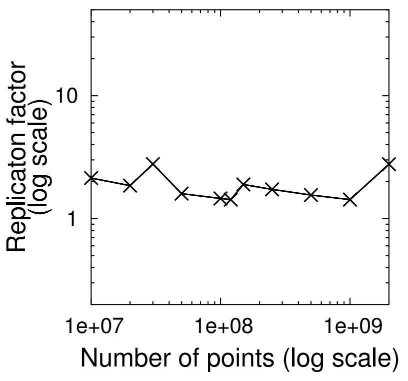
The research described in Section 7.7 opens up the possibility of using the Map-Reduce framework of Google for processing large point-clouds. This framework has been successfully used by Google for processing document datasets. Our work is the first and probably the only one to show applicability of such work to computer graphics. In this chapter, we described a simple yet elegant neighborhood algorithm on the MAP-REDUCE framework. We described a neighborhood algorithm for 3D point-clouds that takes a point-cloud S and a value ε as inputs and computes for each point p in S all other points that are within a distance of ε from it. Such an algorithm has important applications to point rendering and other geometric operations on point-cloud. We applied our algorithm to massive point-clouds, and were able to process a point-cloud containing two billion points in a little more than three and half hours on a cluster containing 20 hosts.



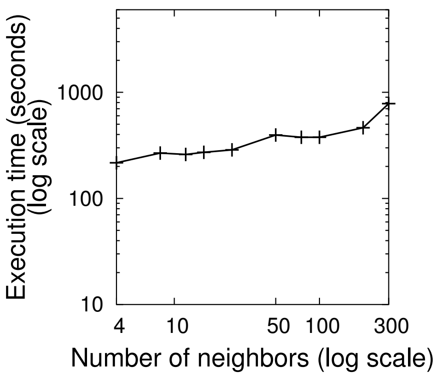
(a)



(b)



(c)



(d)

Figure 7.21: The a) time taken, b) neighborhoods computed per hour in millions, and the c) replication factor of our algorithm for values of n between 10 million and two billion points, and k=8. (d) Time to compute a neighborhood for values of k between 4 and 300 on a point-cloud with 50 million points. The comparison baseline kNN algorithm values for a) and b) are also shown.

Chapter 8

A Quadtree for Objects with Extents

The representation of collections of spatial objects is an important issue in many fields including spatial databases, computer graphics, solid modeling, computer vision, computational geometry, geographic information systems (GIS), game programming, and VLSI design (e.g., [132, 141, 142, 144, 169]). There are two principal methods of representing collections of spatial objects. The first is to use an object hierarchy that initially aggregates objects into groups, preferably based on their spatial proximity, and then uses proximity to further aggregate the groups thereby forming a hierarchy. Queries are facilitated by also associating a minimum bounding box with each object and group of objects as this enables a quick way to test if a point can possibly lie within the area spanned by the object or group of objects. Examples of this method include data structures that make use of axis-aligned bounding boxes (AABB) such as the R-tree [73, 134] and the R*-tree [13], as well as the more general oriented bounding box (OBB) where the sides are orthogonal, while no longer having to be parallel to the coordinate axes (e.g., [65, 131]).

The drawback of the object hierarchy approach is that from the perspective of a space decomposition method, the resulting hierarchy of bounding boxes often leads to a non-disjoint decomposition of the underlying space. This means that if a search fails to find an object in one path starting at the root, then it is not necessarily the case that the object will not be found in another path starting at the root.

The second method is based on a decomposition (usually recursive) of the underlying space into disjoint cells so that a subset of the objects is associated with each cell. There are several ways to proceed. The first is to simply redefine the decomposition and aggregation associated with the object hierarchy method so that the minimum bounding boxes are decomposed into disjoint boxes, thereby also implicitly partitioning the underlying objects that they bound. Representations such as the k-d-B-tree [133], R^+-tree [163], and the cell tree [69] are examples of such an approach.

The second way is to partition the underlying space at fixed positions so that all resulting cells (i.e., blocks) are of uniform size, which is the case when using the uniform grid (e.g., [99]). One drawback of the uniform grid is the possibility of a large number of empty or sparsely-filled cells when the objects are not uniformly distributed, as well as the possibility that most of the objects will lie in a small subset of the cells. This is resolved by making use of a variable resolution representation such as one of the quadtree variants (e.g., [144]) where the underlying space, as well as the objects that are associated with it (i.e., that it contains), is recursively decomposed into congruent sibling cells until each cell has fewer than some predetermined number of objects associated with it. Depending on the underlying representation, the result can be viewed as a hierarchy of congruent cells.

The principal drawback of the disjoint decomposition method is that when the objects have extent (e.g., line segments, rectangles, and any other non-point objects), then an object is associated with more than one cell when it has been decomposed. This means that queries such as those that seek the length of all objects in a particular spatial region will have to remove duplicate objects before reporting the total length (but see [7, 8, 41] which avoid this drawback by making use of the geometry of the type of the data that is being represented).

At times, we want to use a space decomposition method that utilizes a hierarchy of congruent cells while still not decomposing the objects. In this case, we relax the disjointness requirement by stipulating that only the cells at a given level (i.e., depth) of the hierarchy must be disjoint. In particular, we recursively decompose the cells that comprise the underlying space into congruent sibling cells so that each object is associated with just one cell, and this is the smallest possible congruent cell that contains the object in its entirety. Assuming a top-down subdivision process that decomposes each cell into four square cells (i.e., a quadtree) at each level of decomposition, the result is that each object is associated with its minimum enclosing quadtree cell. Subdivision ceases whenever a cell contains no objects. The MX-CIF quadtree (octree) [1, 98], multilayer grid file [174], R-file [90], filter tree [164] (used for spatial join algorithms), and SQhistogram [3] (used for selectivity estimation in processing spatial queries) are examples of this method whose primary difference lies in the nature of the access structure that is used.

In order to simplify our presentation, we assume that the objects stored in the MX-CIF quadtree are rectangles although the MX-CIF quadtree is applicable to arbitrary objects in which case it keeps track of their minimum bounding boxes. For example, Figure 8.1b is the tree representation of the MX-CIF quadtree for a collection of rectangle objects given in Figure 8.1a. Note that objects can be associated with both terminal and non-terminal nodes of the tree.

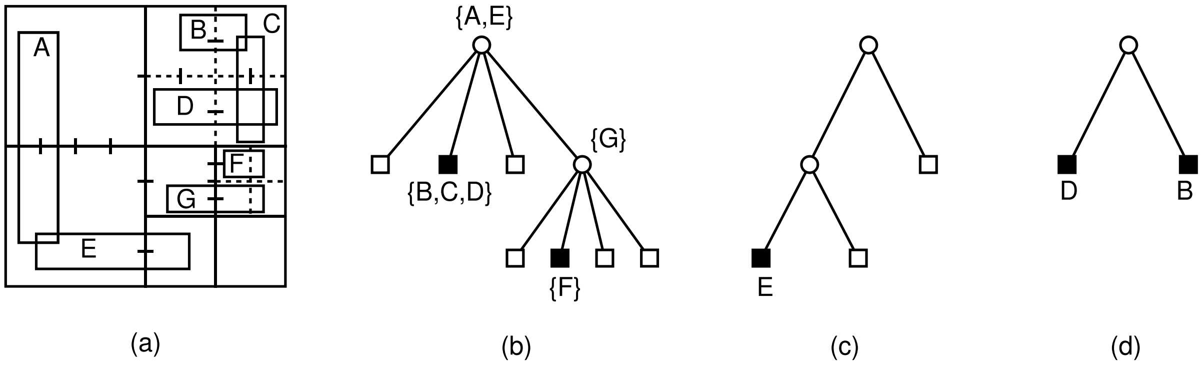


Figure 8.1: (a) Collection of rectangles and the cell decomposition induced by the MX-CIF quadtree; (b) the tree representation of (a); the binary trees for the y axes passing through the root of the tree in (b), and (d) the NE son of the root of the tree in (b).

Since there is no limit on the number of objects that are associated with a particular cell, an additional decomposition rule is sometimes provided to distinguish between these objects. For example, in the case of the MX-CIF quadtree, a one-dimensional analog of the two-dimensional decomposition rule is used. In particular, all objects that are associated with a given cell *b* are partitioned into two sets: those that intersect (or whose sides are collinear) with the vertical axis passing through the center of *b*, and those that intersect (or whose sides are collinear) with the horizontal. Objects that intersect with the center of *b* are associated with the horizontal axis. Associated with each axis is a one-dimensional MX-CIF quadtree (i.e., a binary tree), where each object *o* is associated with the node that corresponds to *o*'s minimum enclosing interval. For example, Figure 8.1c and Figure 8.1d illustrate the binary trees associated with the *y* axes passing through the root and the `NE` son of the root, respectively, of the MX-CIF quadtree of Figure 8.1b. Thus we see that the two-dimensional MX-CIF quadtree acts like a hashing function with the one-dimensional MX-CIF quadtree playing the role of a collision resolution technique.

The MX-CIF quadtree can be interpreted as an object hierarchy where the objects appear at different levels of the hierarchy and the congruent cells play the same role as the bounding boxes. The difference is that the set of possible bounding boxes is constrained to the set of possible congruent cells. Thus, we can view the MX-CIF quadtree as a variable resolution R-tree. An alternative interpretation is that the MX-CIF quadtree provides a variable number of grids, each one being at half the resolution of its immediate successor, where an object is associated with the grid whose cells have the tightest fit. In fact, this interpretation forms the basis of the *filter tree* [164] where the only difference from the MX-CIF quadtree is the nature of the access structure for the cells (i.e., a hierarchy of grids for the filter tree and a tree structure for the MX-CIF quadtree).

One of the main drawbacks of the MX-CIF quadtree is that the size of the cell c corresponding to the minimum enclosing quadtree cell of object o's minimum enclosing bounding box b is not a function of the size of b or o. Instead, it is dependent on the position of o. In fact, c is often considerably larger than b thereby causing inefficiency in search operations due to a reduction in the ability to prune objects from further consideration. This situation arises whenever b overlaps the axes lines that pass through the center of c, and thus w can be as large as the width of the entire underlying space.

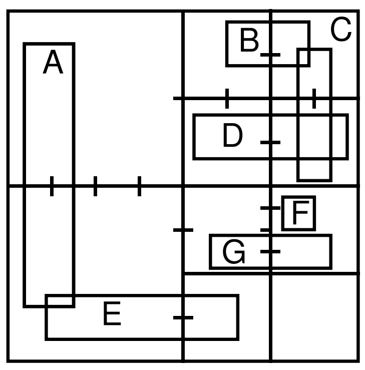
There are several ways of overcoming this drawback. One easy way is to introduce redundancy (i.e., representing the object several times thereby replicating the number of references to it) by decomposing the quadtree cell c into smaller quadtree cells, each of which minimally encloses some portion of o (or, alternatively, some portion of o's minimum enclosing bounding box b) and contains a reference to o. The expanded MX-CIF quadtree [2] is a simple example of such an approach where c is decomposed once into four subblocks c_i, which are then decomposed further until obtaining the minimum enclosing quadtree cell s_i for the portion of o, if any, that is covered by c_i. A more general approach, used in spatial join algorithms, sets a bound on the number of replications, (termed a *size bound* [125] and used in the GESS method [42]) or on the size of the covering quadtree cells resulting from the decomposition of c that contain the replicated references (termed an *error bound* [125]).

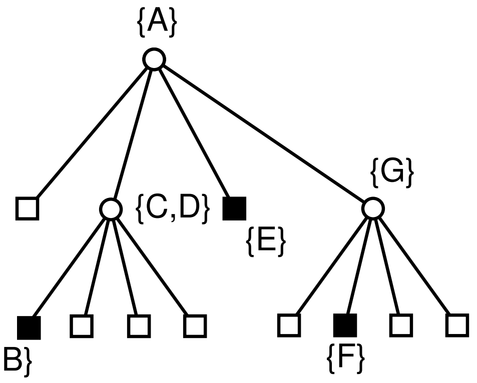
Replicating the number of references to the objects is reminiscent of the manner in which the non-disjointness of the decomposition of the underlying space resulting from the use of an object hierarchy was overcome, and thus has the same shortcoming of possibly requiring the application of a duplicate object removal step prior to reporting the answer to some queries. The *cover fieldtree* [50, 51], and the equivalent *loose quadtree* (*loose octree* in three dimensions) [182] as used in the rest of this paper and motivated by game programming applications, adopt a different approach at overcoming the independence of the sizes of c and b drawback. In particular, they do not replicate the objects. Instead, they expand the size of the space that is spanned by each quadtree cell c of width w by a cell expansion factor p ($p > 0$) so that the expanded cell is of width $(1+p)\cdot w$. In this case, an object is associated with its minimum enclosing expanded quadtree cell. For example, letting $p = 1$, Figure 8.2 is the loose quadtree corresponding to the collection of objects in Figure 8.1(a) and its MX-CIF quadtree in Figure 8.1(b). In this example, there are only two differences between the loose and MX-CIF quadtrees:

1. Rectangle object E is associated with the SW child of the root of the loose quadtree instead of with the root of the MX-CIF quadtree.

2. Rectangle object B is associated with the NW child of the NE child of the root of the loose quadtree instead of with the NE child of the root of the MX-CIF

quadtree.





(a) (b)

Figure 8.2: (a) Cell decomposition induced by the loose quadtree for a collection of rectangle objects identical to those in Figure 8.1, and (b) its tree representation.

Ulrich [182] has shown that given a quadtree cell *c* of width *w* and cell expansion factor *p*, the radius *r* of the minimum bounding box *b* of the smallest object *o* that could possibly be associated with *c* must be greater than $pw/4$. In particular, the utility of the loose quadtree is best evaluated in terms of the inverse of this relation. In particular, we are interested in the maximum possible width *w* of *c* given an object *o* with minimum bounding box *b* of radius *r*. This is because reducing *w* is the real motivation for the development of the loose quadtree as an alternative to the MX-CIF quadtree for which *w* can be as large as the width of the underlying space. We achieve our result in Section 8.1 by examining the range of the relative widths of *c* and *b* as this provides a way of taking into account the constraints imposed by the fact that the range of values of *w* is limited to powers of 2. Concluding remarks are drawn in Section 8.2.

8.1 Calculation of the Maximum Loose Quadtree Cell Width

A key principle to observe is that in the loose quadtree, the smallest expanded quadtree cell *c* of width *w* that contains the object *o* has the property that the centroid of *o* (actually of *o*'s minimum bounding box *b* of radius *r*) is contained in the non-expanded portion of *c*. Thus insertion proceeds by finding the smallest quadtree cell *c* that contains the centroid of *b*, and whose expanded cell also contains *o*. The traditional way of finding *c* is to recursively search the quadtree starting at the root and descend to the appropriate child based on the value of the centroid. In fact, it turns out that there is even an easier way of determining *c*, which involves little search (i.e., descents in the quadtree). In particular, we show below that the width *w* of *c* must lie within a relatively small range of values, thereby greatly restricting the number of possible cells that must be tested for the inclusion of *o*.

Recall that one of the key drawbacks of data structures such as the MX-CIF quadtree that associate an object *o* with the minimum sized quadtree cell *c* of width *w* that encloses the minimum bounding box *b* of radius *r* of *o* is that *w* is primarily a function of the position of *o*, and to a lesser extent, a function of *r*. In contrast, in the loose quadtree, as we show in the rest of this section, the dependence of *w* on the position of *o* is reduced significantly. In particular, we demonstrate that *w* lies within a range of values that only depend on the radius *r* of *o*'s minimum bounding box *b* and the value of the cell expansion factor *p*. In fact, we show that the ratio of the range of the widths of c and b (i.e., $w/2r$) is only dependent on p.

We first derive a lower bound on the range of the ratios. From the definition of the cell expansion factor p, we know that given an object o with minimum bounding box b of radius r, the smallest quadtree cell c of width w with which o can be associated so that o's centroid lies in the non-expanded portion of c arises when the centroids of b and c coincide, and moreover the cell c' resulting from the expansion of c (i.e., having width $(1+p)w$) is just large enough to contain b of width $2r$ (see Figure 8.3(a)). This leads to the inequality which is given below as:

$$(1+p)w \geq 2r \tag{8.1}$$

which can be rewritten as

$$\frac{w}{2r} \geq \frac{1}{1+p}. \tag{8.2}$$

We can use similar reasoning to obtain an upper bound on the range of the ratios, and in the process use a similar construction to that of Ulrich [182] except that for a given cell expansion factor p, Ulrich assumed the existence of a quadtree cell c of width w and was seeking the radius r of the minimum bounding box b of the smallest object o that could possibly be associated with the expanded cell c, while we are assuming that for a given cell expansion factor p, we are given an object o with minimum bounding box b of radius r and are seeking the width w of the largest cell c with whose expanded cell b would be associated. We make use of our observation that the centroid of the object o with minimum bounding box b of radius r is always required to be contained in the

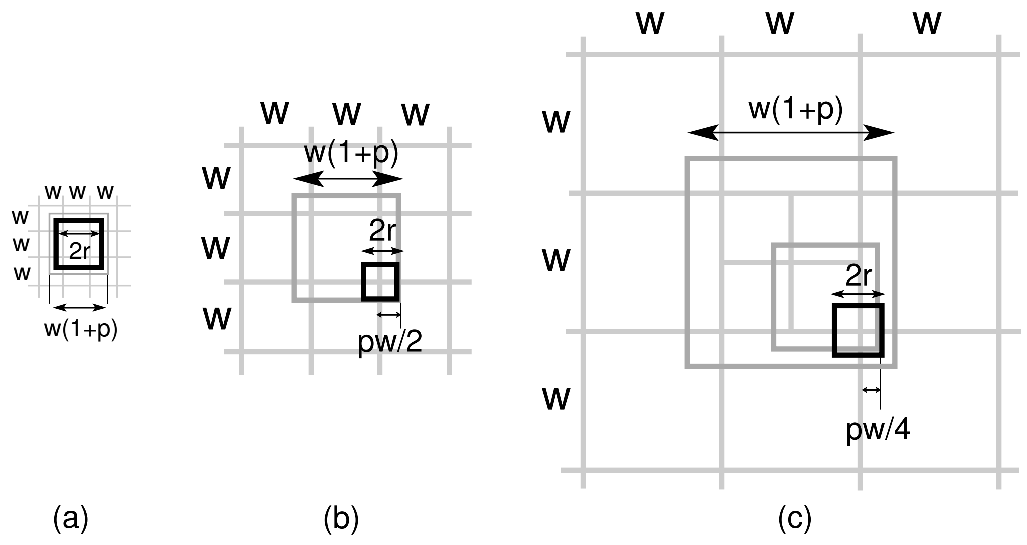


Figure 8.3: Assuming cell expansion factor p and an object o with minimum bounding box b of radius r, examples showing the (a) smallest ratio of the width w of the quadtree cell c associated with b and the width of b which is attained when the centroids of o and c coincide, and the (b) lower and (c) upper bounds on the largest ratio attained when the centroid of o coincides with one of the corners of c.

non-expanded portion of the associated quadtree cell.

Given this observation, we note that the largest quadtree cell c of width w that can satisfy this requirement on the placement of the centroid has the property that one of c's corners is coincident with the centroid of o, and that the radius r of b is not too large so that b is too large for the expanded region of c (see Figure 8.3(b)), and just large enough so that b does not fit in the expanded region of one of the subcells of c of width $w/2$ (see Figure 8.3(c)). Equivalently, for this particular configuration, we say that $pw/4 = 2^{k-1} = r - \delta' < r \leq 2^k = pw/2$ for some value of k and $\delta' > 0$. Alternatively, letting $\delta' = \delta w/4$, we have $pw/4 = 2^{k-1} = r - \delta w/4 < r \leq 2^k = pw/2$ for some $\delta > 0$.

Since the width w of c is the same for all values of r in this range, we point out that c's width relative to that of b is maximized when r takes on the value:

$$r = pw/4 + \delta w/4, \delta > 0. \tag{8.3}$$

which can be rewritten as:

$$\begin{aligned} w/2r &= \frac{w}{\frac{pw}{2} + \delta\frac{w}{2}}, \delta > 0, & (8.4)\\ w/2r &= \frac{2}{p+\delta} < \frac{2}{p}, & (8.5)\\ w/2r &< \frac{2}{p}. & (8.6) \end{aligned}$$

Combining relations 8.2 and 8.6 yields the range:

$$\frac{1}{1+p} \leq \frac{w}{2r} < \frac{2}{p}. \tag{8.7}$$

Without loss of generality, let us assume that the quadtree cell corresponding to the root of the loose quadtree is length of 2^g, where g is an integer. In this case, all cells c in the loose quadtree have width $w = 2^k$, such that $k \leq g$ is an integer. Now, for any given value x, let us define a function $M(x)$ which determines a k such that $2^{k-1} < x \leq 2^k$, and returns the value 2^k. In other words,

$$M(x) = 2^k, 2^{k-1} < x \leq 2^k. \tag{8.8}$$

Moreover, we also have that

$$1 \leq \frac{M(x)}{x} < 2. \tag{8.9}$$

The rationale behind the function $M(x)$ is that it *quantizes* x to the next higher power of 2 unless it is already a power of 2. To explain the utility of $M(x)$ from a geometric point of view, consider an input object R with a minimum bounding box of radius r. We have that $M(r)$ is the radius of the smallest quadtree cell that can potentially contain R. We now derive the minimum and maximum possible ratios of $w/2r$ in terms of $M(.)$. Our motivation is to be able to identify a set of quadtree cells (typically a few) in the loose quadtree that can potentially contain R. From relation 8.7, we are given that $w/2r$ is greater than or equal to $1/(p+1)$, but is less than $2/p$. Consider an input object R with a minimum bounding box of radius r. How many levels of the loose quadtree does the range $[1/(p+1), 2/p)$ span? This is upper-bounded by the number of numbers of the form 2^k, where k is an integer, that is contained in the range $[1/(p+1), 2/p)$. That is, the number of levels spanned by the range in relation 8.7 cannot exceed V, which is given by:

$$V = \log(M(2/p)) - \log(M(1/(p+1))). \tag{8.10}$$

8.2 Discussion

Now, let us make some observations on the possible ranges of relative cell widths on the basis of relations 8.7 and 8.10. First, for the degenerate case of the MX-CIF quadtree, in which case no expansion takes place ($p = 0$), we have an unbounded upper bound on the range of values and a lower bound of 1. As p increases towards 1, the range of values decreases. For example, for $p = 1/4$, we have a range of relative cell widths ranging $[4/5, 8)$. The set of quadtree cells containing a given input rectangle R with a minimum bounding box of radius r are between $[M(4/5) = 1, M(8) = 8) = \{1, 2, 4\}$. In other words, the quadtree cells containing R in the loose quadtree can be of radius $M(r)$, $2M(r)$, and $4M(r)$.

For $p = 1$, there are just two possible relative cell widths corresponding to $[M(1/2) = 1/2, M(2) = 2) = \{1/2, 1\}$. In other words, the associated quadtree cells of R can be either the a quadtree cell of radius $M(r)$, or can be half of $M(r)$. As p increases beyond 1, the number of possible ratios of relative cell widths oscillate between one and two. In particular, for $p = 1, 3, 7, 15, 31, 63 \ldots$ ($p = 2^k - 1$, where $k \geq 0$ is an integer), the ratio $w/2r$ takes on two values $[M(1/2^k) = 2^{-k}, M(2/(2^k - 1)) = 2^{2-k})$, while for all other values of p ($2^k \leq p < 2^{k+1} - 1$, where $k \geq 0$ is an integer), $w/2r$ takes on just one value $M(1/2^k) = 2^{-k}$.

We now briefly describe a simple $O(1)$ object insertion algorithm for the loose quadtree using the example of $p = 1/4$. From relation 8.7, we have that the quadtree cells containing a given input rectangle R with a minimum bounding box of radius r can be associated with one of three possible cells of radius $M(r)$, $2M(r)$, and $4M(r)$. The insertion algorithm proceeds as follows. We first find in $O(1)$ a cell b of radius $M(r)$, such that it contains the centroid of R. We then have that either b, parent of b (say b') of radius $2M(r)$, or the parent of b' (say b") of radius $4M(r)$ contains R and we insert R in the smallest one whose expanded region contains R.

This leads to the observation that as p takes larger values (even for p as small as $1/4$), the loose quadtree treats the input objects as if they are points and it is their centroid that determines their associated quadtree cell, while their size and the value of the cell expansion factor determine the size of their associated quadtree cell. Actually, the above statement must be tempered a bit. In particular, although it implies that the position of object o is not a factor in the determination of the width w of the expanded quadtree cell c with which o's minimum bounding box b is associated, this is not quite true as the existence of a range of values for the ratio $w/2r$ of the widths of c and b is a direct result of the variation in the position of o along with that of the value of p. However, as we showed above, for most realistic values of p (i.e., $p \geq 1$), the values of the ratio of the widths of c and b take on at most two values which differ by one where the only reason for the two possible ratio values is the fact that at times p takes on a value which is one less than a power of 2.

Of course, as we pointed out, when $p < 1$ and approaches 0, the ratio starts to take on an increasing number of values. Finally, we also observe that all of the results that we have described hold for loose quadtrees of arbitrary dimension (e.g., three dimensions such as the loose octree) as they are all formulated in terms of the widths of the quadtree cells.

Algorithms that make use of the loose quadtree are simplified by our observation that the centroid of object *o* (actually of *o*'s minimum bounding box *b* of radius *r*) is always contained in the non-expanded portion of the quadtree cell *c* with which *o* is associated. However, there are scenarios where users may wish to violate this property. For example, for certain values of *r* and *p*, *r* may be sufficiently large so that both the centroid of *o* lies in the expanded portion of *c* and *o* still fits in the expanded cell *c*. This situation is desirable when users want to move *o* as much as possible without having to associate it with another quadtree cell just because *o*'s centroid is no longer in the non-expanded region of *c*. Interestingly, this modification does not change the ranges of relative cell widths as the example in Figure 8.3(c) still corresponds to the largest value of the ratio. The difference is that now the motion of the object so that the centroid of *o* is also in the expanded portion of *c* does not result in the association of *o* with another cell as long as *o* lies entirely in the expanded portion of *c*. Of course, this complicates subsequent searches (as well as delete operations), as now instead of just looking for a cell whose non-expanded portion contains the centroid of *o*, we must examine all possible cells whose expanded cells can contain *o*. Notice that in essence, we have transformed the search problem from one involving points (i.e., centroids of the objects) to one involving regions (i.e., the minimum bounding boxes of the objects).

It is interesting to note that the loose quadtree (cover fieldtree) is not the only approach at overcoming the drawback of the MX-CIF quadtree. In particular, the partition fieldtree [50, 51] is an alternative method at achieving the same result by shifting the positions of the centroids of cells at successive levels of subdivision by one-half the width of the cell that is being subdivided. Figure 8.4 shows an example of such a subdivision. This subdivision rule guarantees that the width w of the minimum enclosing quadtree cell c for the minimum bounding box b for object o is bounded by eight times the maximum extent r of b [51, 144]. The same ratio is obtained for the cover fieldtree when $p = 1/4$, and thus the partition fieldtree is superior to the cover fieldtree when $p < 1/4$ [144].

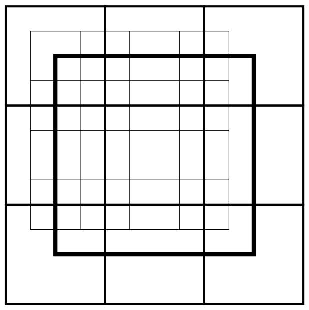


Figure 8.4: Example of the subdivision induced by a partition fieldtree.

8.3 Concluding Remarks

We have shown how to determine the maximum possible width w of the minimum enclosing quadtree cell c corresponding to an object o with minimum bounding box b of radius r and cell expansion factor p. We have also shown that w is independent of the position of o. This property enables determining the block with which o is associated and can be used, for example, in an algorithm to build a loose quadtree. In particular, this independence means that the algorithm requires little or no search and could be used, for example, to populate a spatial database with the latest wave of multiprocessors such as those that make use of GPUs (e.g., [66, 175]).

The actual properties of the cover fieldtree such as the reduced sensitivity of w to the size of the object o and its position can be seen by using VASCO [21–23], a system for Visualizing and Animating Spatial Constructs and Operations. VASCO consists of a set of spatial index JAVA$^{\text{TM}}$ (e.g., [9]) applets that enable users on the worldwide web to experiment with a number of hierarchical representations (e.g., [141, 142, 144]) for different spatial data types, and see animations of how they support a number of search queries (e.g., nearest neighbor and range queries). The VASCO system for the cover fieldtree can be found at `http://cs.umd.edu/~hjs/quadtree/rectangles/loosequad.html`. It can also be used to compare the cover fieldtree with numerous other spatial indexing methods. A couple of operations are particularly worth noting. The *Move* operation enables users to see the dependence of the associated quadtree cell on the position of the centroid of the object's minimum bounding box where both the associated quadtree cell c and the expanded quadtree cell are displayed. The *Motion Insensitivity* operation enables viewing the situation where users want to be able to move an object o as much as possible without having to associate it with another minimum enclosing quadtree cell c just because o's centroid is no longer in the non-expanded region of c.

Chapter 9

Open Problems

In this chapter, we discuss a few interesting related problems to our work on scalable query processing on spatial networks.

Open Problem in Query Processing on Spatial Networks

1. **Doubling Dimension Based Methods on Spatial Network**

   We first note that the vertices of a general graph and the network distance measure represent a metric space. This means that any spatial network G can be embedded in a high-dimensional vector space using an embedding technique (e.g., Lipschitz embedding [110, 144]). For example, the *Road Network Embedding* (RNE) technique of Shahabi *et al.* [166] uses the Lipschitz embedding to embed the vertices of a spatial network in a high-dimensional vector space. The obvious drawback of such an approach is that the resultant high-dimensional representation is cumbersome to work with, not to mention that the *distortion* of the embedding can be very large.

   Another related concept is that of *doubling dimensions* [70, 103] used in the context of finite metric spaces. The doubling dimension of a finite metric space $M(S,d)$, where S is the finite set of objects and d is a distance function, is defined as the logarithm of the minimum number of balls of radius $r/2$ in M required to cover a ball of radius r in M. Har-Peled and Mendel [75] characterize the doubling dimension as a generalization of a Euclidean dimension as $\mathbb{R}^d$ has a doubling dimension of $O(d)$. It can be easily verified [4] that the doubling dimension of road networks is small and bounded. This opens up the possibility of applying a whole body of research dealing with finite metric spaces with bounded doubling dimensions to spatial networks. In particular, results on compact routing tables [4, 177], well separated pair decomposition [75, 177], and low distortion embedding [70] of finite metric spaces with bounded doubling dimensions are applicable to our work on spatial networks. An interesting area for future research would be to determine if a better bound on the size and the access time of the PCP decomposition can be obtained using the concept of doubling dimensions.

2. **Distance Encoding on spatial networks using Adaptive Distance Fields (ADF)**

   We propose to represent the network distance between each pair of vertices in a spatial network G using an *adaptive distance field* (ADF) [55]. Recall that the spatial position of a vertex s in G is denoted by $p(s)$. A scalar field F is constructed as follows. Without loss of generality, we assume that the vertices in G are embedded in a two-dimensional vector space. The spatial position of a source vertex

s, say $p(s) = (x, y)$ and a destination vertex v, say $p(v) = (a, b)$ is augmented resulting in a four-dimensional point (x, y, a, b). The resulting point is associated with a scalar value, $f_{sv} = \frac{d_G(s,v)}{d_S(s,v)}$ corresponding to the ratio of the network and the geodesic distance between s and v. An ADF on a spatial networks consists of the following — a quadtree Q on the $O(n^2)$ points in F, a suitably provided *interpolation* function, and an *error threshold* function. Moreover, we also impose an additional restriction on the size of the resulting ADF representation. In particular, the size of the ADF is bounded by a value α which is specified during the construction of the ADF. We can now ask the following?

(a) How can the adaptive distance field F be used to compute the network distance interval between two objects, an object and a region, or between two regions on a spatial network?

(b) What is the relationship between α and the "quality" of the network distance intervals obtained by F?

3. **Progressive Refinement of distance on other domains**

In this dissertation, we introduced the concept of *progressive refinement* of distances, which enabled us to quickly compute the network distance between two objects s, t on a spatial network as a distance interval $[\delta^-, \delta^+]$, such that the actual network distance $d_G(s, t)$ between s, t is contained in $[\delta^-, \delta^+]$. Furthermore, we also introduced a *refinement* operator that can *improve* the distance interval by expending some additional work. An efficient query processing algorithm only performs as many *refinements* as necessary to be able to answer a query without ambiguity. We will investigate if such an approach to query processing is applicable to other *domains*, where the distance computations are similarly expensive.

4. **Dynamic updates on SILC and PCP**

   Can the SILC and PCP frameworks be modified to efficiently handle updates to the underlying spatial network? In particular, we are interested in the following kinds of updates to the underlying network.

   - The weight of an edge increases (e.g., traffic congestion)
   - A set of vertices (along with all incident and outgoing vertices) are removed from the spatial network (e.g., road closures)
   - A *bi-directional* edge becomes a *directed* edge (e.g., one-way restrictions based on the hour of the day)

5. **SILC on Mesh Models**

   Mitchell, Mount and Papadimitriou [116] describe the MMP algorithm for computing shortest paths and geodesic distances on 3D meshes, and Surazhsky *et al.* [176] demonstrate an effective implementation of the algorithm. Can the SILC framework be used to speed operations that make extensive use of geodesic distances on a 3D mesh models?

6. **Minimum and Maximum Distortion of Spatial Networks**

   Using the model of a spatial network described in this dissertation, can we come up with an algorithm to compute bounds on the minimum and maximum distortions [121] of a spatial network.

7. **Shortest-Path Quadtree Construction on Parallel and Distributed Frameworks**

   Both the SILC and the PCP frameworks require that the shortest path between every pair of vertices in the spatial networks be precomputed using an offline process. Such a process can be speeded up by using a host of parallel and distributed computational techniques, such as Graphical Processing Units (GPU) [76, 115] and cloud computing frameworks, such as MAP-REDUCE [39] and Dryad [91]. We are interested in the construction of shortest-path quadtrees on large road networks. An interesting topic of research would be to verify the scalability of the proposed approaches to massive spatial network datasets such as the road network of continental US containing up to 24 million vertices.

8. **Approximate Shortest-Path quadtrees**

   We 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. This is not unreasonable as most people do not want to drive more than 50 miles to get to a restaurant. If we limit the set of vertices in the spatial network to those within a radius of 50 miles around a vertex, then the resulting 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 [93] 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.

9. **Inverse Shortest-Path quadtree**

   A drawback of the shortest-path quadtree is that it is not capable of computing the network distance interval between a block and an object. One possible way of obtaining such an interval would be to compute an *inverse* shortest-path quadtree, in addition to the shortest-path quadtree, for each vertex in a spatial network G. In particular, an inverse shortest-path quadtree of a vertex w is the spatial aggregation of the vertices based on which source vertices share the same last link in their shortest paths to w. A problem of interest is that given the shortest-path quadtrees of all the vertices in a spatial network, is it possible to devise an algorithm that can compute the inverse shortest-path quadtrees from the given input?

10. Investigate the applicability of Reach measure of Gutman [72] to shortest-path quadtrees.

Future work for the *kNN* algorithm

1. **ε-approximate *kNN* algorithm**

   Although the focus of our work is the computation of the exact k nearest neighbors, it can also be used to compute the approximate nearest neighbors to an object. An interesting problem is the conceptualization of an ε-approximate locality L_ε of a block b, such that L_ε contains the ε-approximate k nearest neighbors of all the objects contained in b.

2. **A data structure with small locality guarantees**

   We have shown that for a given subdivision of space, the BUILDLOCALITY algorithm is *optimal*, although, the size of the locality depends solely on our choice of the data structure. It is not difficult to see that certain datasets and data structure configurations may result in large localities of objects. An interesting direction of research is the design of a data structure that can guarantee that the average size of the locality will be small, thereby providing good performance guarantees.

3. ***kNN* algorithm using other estimators**

Our *kNN* algorithm only requires the ability to compute MINDIST, MAXDIST and an estimate of the number of objects contained in a block. An interesting study would be to examine if smaller localities can be obtained by making use of additional statistics on the distribution of the objects contained in a block. One such statistics is the MAXNEARESTDIST estimator [138, 145]. In particular, we plan to compare the performance of our *kNN* algorithm with the recent algorithm of Chen and Patel [33] who use the MAXNEARESTDIST estimator.

4. **Radially well distributed neighbors**

   How can the locality L of a block b be redefined, so that neighbors of a query object q are also radially well distributed around q?

5. ***kNN* algorithm on higher dimensional datasets**

   Explore the applicability of some of the concepts discussed here to high-dimensional datasets.

Bibliography

[1] D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics, and Image Processing*, 24(1):1–13, Oct. 1983.

[2] D. J. Abel and J. L. Smith. A data structure and query algorithm for a database of areal entities. *Australian Computer Journal*, 16(4):147–154, Nov. 1984.

[3] A. Aboulnaga and J. F. Naughton. Accurate estimation of the cost of spatial selections. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 123–134, San Diego, CA, Feb. 2000.

[4] I. Abraham, C. Gavoille, A. V. Goldberg, and D. Malkhi. Routing in networks with low doubling dimension. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*, Lisbon, Portugal, July 2006. online proceedings.

[5] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin, and C. T. Silva. Point set surfaces. In *Proceedings of the conference on Visualization 2001*, pages 21–28, San Diego, CA, Oct. 2001. IEEE Computer Society.

[6] M. Andersson, J. Giesen, M. Pauly, and B. Speckmann. Bounds on the k-neighborhood for locally uniformly sampled surfaces. In *Proceedings of the Eurographics Symposium on Point-Based Graphics*, pages 167–171, Zurich, Switzerland, June 2004.

[7] W. G. Aref and H. Samet. Uniquely reporting spatial objects: yet another operation for comparing spatial data structures. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 178–189, Charleston, SC, Aug. 1992.

[8] W. G. Aref and H. Samet. Hashing by proximity to process duplicates in spatial databases. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM)*, pages 347–354, Gaithersburg, MD, Dec. 1994.

[9] K. Arnold and J. Gosling. *The JAVA$^{\text{TM}}$ Programming Language*. Addison-Wesley, Reading, MA, 1996.

[10] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, Arlington, VA, Jan. 1994. (journal version: [11]).

[11] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *Journal of the ACM*, 45(6):891–923, Nov. 1998. Also see *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 573–582, Arlington, VA, January 1994.

[12] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[13] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference*, pages 322–331, Atlantic City, NJ, June 1990.

[14] R. E. Bellman. *Adaptive Control Processes*. Princeton University Press, Princeton, NJ, 1961.

[15] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, Sept. 1975.

[16] S. Berchtold, C. Böhm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 78–86, Tucson, AZ, May 1997.

[17] S. N. Bespamyatnikh. An optimal algorithm for closest pair maintenance. In *Proceedings of the 11th Annual Symposium on Computational Geometry*, pages 152–161, Vancouver, Canada, June 1995.

[18] J. F. Blinn. Simulation of wrinkled surfaces. In *Proceedings of the SIGGRAPH'78 Conference*, pages 286–292, Atlanta, GA, Aug. 1978. ACM Press.

[19] C. Böhm and F. Krebs. Supporting KDD applications by the k-nearest neighbor join. In V. Marík, W. Retschitzegger, and O. Stepánková, editors, *Proceedings of 14th International Workshop on Database and Expert Systems Applications (DEXA'99)*, vol. 2352 of Springer-Verlag Lecture Notes in Computer Science, pages 504–516, Prague, Czech Republic, Sept. 2003.

[20] C. Böhm and F. Krebs. The *k*-nearest neighbor join: turbo charging the KDD process. In *Knowledge and Information Systems*, volume 6, pages 728–749, London, UK, Nov. 2004. Springer-Verlag.

[21] F. Brabec and H. Samet. The VASCO R-tree JAVA$^{\text{TM}}$ applet. In Y. Ioannidis and W. Klas, editors, *Visual Database Systems (VDB4). Proceedings of the IFIP TC2//WG2.6 Fourth Working Conference on Visual Database Systems*, pages 147–153, L'Aquila, Italy, May 1998. Chapman and Hall.

[22] F. Brabec and H. Samet. Visualizing and animating R-trees and spatial operations in spatial databases on the worldwide web. In Y. Ioannidis and W. Klas, editors, *Visual Database Systems (VDB4). Proceedings of the IFIP TC2//WG2.6 Fourth Working Conference on Visual Database Systems*, pages 123–140, L'Aquila, Italy, May 1998. Chapman and Hall.

[23] F. Brabec and H. Samet. Visualizing and animating search operations on quadtrees on the worldwide web. In K. Kedem and M. Katz, editors, *Proceedings of the 16th European Workshop on Computational Geometry*, pages 70–76, Eilat, Israel, Mar. 2000.

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

[25] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proceedings of the ACM SIGMOD Conference*, pages 237–246, Washington, DC, May 1993.

[26] A. J. Broder. Strategies for efficient incremental nearest neighbor search. *Pattern Recognition*, 23(1–2):171–178, Jan. 1990.

[27] P. A. Burrough and R. A. McDonnell. *Principles of geographical information systems*. Oxford University Press, New York, NY, Apr. 1998.

[28] P. Callahan. *Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications*. PhD thesis, The Johns Hopkins University, Baltimore, MD, Sept. 1995.

[29] P. B. Callahan and S. R. Kosaraju. Faster algorithms for some geometric graph problems in higher dimensions. In *Proceedings of the 4th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 291–300, Austin, Texas,, Jan. 1993.

[30] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 263–272, San Francisco, 1995.

[31] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42(1):67–90, Jan. 1995.

[32] M. J. Carey and D. Kossmann. On saying “enough already!” in SQL. In J. Peckham, editor, *Proceedings of the ACM SIGMOD Conference*, pages 219–230, Tucson, AZ, May 1997.

[33] Y. Chen and J. M. Patel. Efficient evaluation of all-nearest-neighbors queries. In *Proceedings of the 23rd IEEE International Conference on Data Engineering*, pages 1056–1065, Istanbul, Turkey, Apr. 2007.

[34] H.-J. Cho and C.-W. Chung. An efficient and scalable approach to CNN queries in a road network. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-A. Larson, and B. C. Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 865–876, Trondheim, Norway, Sept. 2005.

[35] P. Ciaccia and M. Patella. PAC nearest neighbor queries: approximate and controlled search in high-dimensional and metric spaces. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 244–255, San Diego, CA, Feb. 2000.

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

[37] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[38] R. L. Cook. Shade trees. In *Proceedings of the SIGGRAPH’84 Conference*, pages 223–231, New York, NY, July 1984. ACM Press.

[39] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI’04: 6th Symposium on Operating Systems Design and Implementation*, pages 137–150, San Franciso, CA, Dec. 2004.

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

[41] J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 535–546, San Diego, CA, Feb. 2000.

[42] J.-P. Dittrich and B. Seeger. GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *Proceedings of the 7th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 47–56, San Francisco, Aug. 2001.

[43] J. Feng, N. Mukai, and T. Watanabe. Incremental maintenance of all-nearest neighbors based on road network. In R. Orchard, C. Yang, and M. Ali, editors, *Proceedings 17th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE 2004*, vol. 3029 of Springer-Verlag Lecture Notes in Computer Science, pages 164–169, Ottawa, Canada, May 2004.

[44] G. G. Filho and H. Samet. A hybrid shortest path algorithm for intra-regional queries in hierarchical shortest path finding. Computer Science Technical Report TR–4417, University of Maryland, College Park, MD, Nov. 2002.

[45] R. A. Finkel and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.

[46] J. Fischer and S. Har-Peled. Dynamic well-separated pair decomposition made easy. In *CCCG'05: Proceedings of the 17th Canadian Conference on Computational Geometry*, pages 235–238, Windsor, Canada, Aug. 2005.

[47] S. Fleishman, I. Drori, and D. Cohen-Or. Bilateral mesh denoising. In *Proceedings of the SIGGRAPH'03 Conference*, volume 22(3), pages 950–953, San Diego, CA, July 2003. ACM Press.

[48] M. S. Floater and M. Reimers. Meshless parameterization and surface reconstruction. *Computer Aided Geometric Design*, 18(2):77–92, Mar. 2001.

[49] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, June 1962.

[50] A. Frank. Problems of realizing LIS: storage methods for space related data: the fieldtree. Technical Report 71, Institute for Geodesy and Photogrammetry, ETH, Zurich, Switzerland, June 1983.

[51] A. U. Frank and R. Barrera. The Fieldtree: a data structure for geographic information systems. In A. Buchmann, O. Günther, T. R. Smith, and Y.-F. Wang, editors, *Design and Implementation of Large Spatial Databases—1st Symposium, SSD'89*, vol. 409 of Springer-Verlag Lecture Notes in Computer Science, pages 29–44, Santa Barbara, CA, July 1989.

[52] G. N. Frederickson. Planar graph decomposition and all pairs shortest paths. *Journal of the ACM*, 38(1):162–204, Jan. 1991.

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

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

[55] S. F. Frisken, R. N. Perry, A. P. Rockwood, and T. R. Jones. Adaptively sampled distance fields: a general representation of shape for computer graphics. In *Proceedings of the SIGGRAPH'00 Conference*, pages 249–254, New Orleans, LA, July 2000.

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

[57] J. Gao and L. Zhang. Well-separated pair decomposition for the unit-disk graph metric and its applications. In *Proceedings of the 35th Annual ACM Symposium on the Theory of Computing*, pages 483–492, San Diego, CA, 2003.

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

[59] M. Garland and P. S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of the SIGGRAPH'97 Conference*, pages 209–216, Los Angeles, Aug. 1997.

[60] B. George, S. Kim, and S. Shekhar. Spatio-temporal network databases and routing algorithms: a summary of results. In D. Papadias, D. Zhang, and G. Kollios, editors, *Advances in Spatial and Temporal Databases— 10th International Symposium, SSTD'07*, vol. 4605 of Springer-Verlag Lecture Notes in Computer Science, pages 460–477, Boston, MA, July 2007. Springer.

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

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

[63] J. Goldstein, J. C. Platt, and C. J. C. Burges. Redundant bit vectors for quickly searching high-dimensional regions. In ??, editor, *The Sheffield Machine Learning Workshop*, vol. 3635 of Springer-Verlag Lecture Notes in Computer Science, Sheffield?, ?? 2005?

[64] H. Gonzalez, J. Han, X. Li, M. Myslinska, and J. P. Sondag. Adaptive fastest path computation on a road network: a traffic mining approach. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 794–805, Vienna, Austria, Sept. 2007.

[65] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: a hierarchical structure for rapid interference detection. In *Proceedings of the SIGGRAPH'96 Conference*, pages 171–180, New Orleans, LA, Aug. 1996.

[66] N. K. Govindaraju, B. Lloyd, W. Wang, M. C. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the ACM SIGMOD Conference*, pages 215–226, Paris, France, June 2004.

[67] M. Gross and H. Pfister, editors. *Point-Based Graphics*. Morgan Kaufmann, San Fransico, CA, 2007.

[68] J. Gudmundsson, C. Levcopoulos, G. Narasimhan, and M. Smid. Approximate distance oracles for geometric graphs. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 828–837, San Francisco, CA, Jan. 2002.

[69] O. Günther and J. Bilmes. Tree-based access methods for spatial databases: implementation and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):342–356, Sept. 1991. Also University of California at Santa Barbara Computer Science Technical Report TRCS88–23, October 1988.

[70] A. Gupta, R. Krauthgamer, and J. R. Lee. Bounded geometries, fractals, and low-distortion embeddings. In *Proceedings of the 44th IEEE Annual Symposium on Foundations of Computer Science*, pages 534–534, Cambridge, MA, Oct. 2003.

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

[72] R. Gutman. Reach-based routing: a new approach to shortest path algorithms optimized for road networks. In *Proceedings 6th Workshop on Algorithm Engineering and Experiments*, pages 100–111, New Orleans, LA, Jan. 2004. SIAM.

[73] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference*, pages 47–57, Boston, June 1984.

[74] S. Har-Peled. A practical approach for computing the diameter of a point set. In *Proceedings of the 17th Annual Symposium on Computational Geometry*, pages 177–186, Medford, MA, June 2001.

[75] S. Har-Peled and M. Mendel. Fast construction of nets in low dimensional metrics, and their applications. In *SCG '05: Proceedings of the 21st Annual Symposium on Computational geometry*, pages 150–158, Pisa, Italy, June 2005.

[76] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. In *HiPC'07: 14th International Conference on High Performance Computing*, vol. 4873 of Springer-Verlag Lecture Notes in Computer Science, pages 197–208, Goa, India, Dec. 2007. Springer.

[77] A. Henrich. A distance-scan algorithm for spatial access structures. In N. Pissinou and K. Makki, editors, *Proceedings of the 2nd ACM Workshop on Geographic Information Systems*, pages 136–143, Gaithersburg, MD, Dec. 1994.

[78] 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.

[79] F. S. Hillier and G. J. Lieberman. *Introduction to Operations Research*. Holden-Day, San Francisco, 1967.

[80] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In M. J. Egenhofer and J. R. Herring, editors, *Advances in Spatial Databases—4th International Symposium, SSD'95*, vol. 951 of Springer-Verlag Lecture Notes in Computer Science, pages 83–95, Portland, ME, Aug. 1995.

[81] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. Computer Science Technical Report TR–3919, University of Maryland, College Park, MD, July 1998.

[82] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In L. Hass and A. Tiwary, editors, *Proceedings of the ACM SIGMOD Conference*, pages 237–248, Seattle, WA, June 1998.

[83] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM Transactions on Database Systems*, 24(2):265–318, June 1999. Also University of Maryland Computer Science Technical Report TR–3919, July 1998.

[84] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. *Computer Graphics*, 26(2):71–78, July 1992. Also in *Proceedings of the SIGGRAPH'92 Conference*, Chicago, July 1992.

[85] H. Hu, D. L. Lee, and V. C. S. Lee. Distance indexing on road networks. In U. Dayal, K.-Y. Whan, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 894–905, Seoul, Korea, Sept. 2006.

[86] H. Hu, D. L. Lee, and J. Xu. Fast nearest neighbor search on road networks. In *Advances in Database Technology—EDBT 2006, Proceedings of the 10th International Conference on Extending Database Technology*, vol. 3896 of Springer-Verlag Lecture Notes in Computer Science, pages 186–203, Munich, Germany, Mar. 2006. Springer.

[87] G. M. Hunter. *Efficient computation and data structures for graphics*. PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1978.

[88] G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):145–153, Apr. 1979.

[89] 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.

[90] A. Hutflesz, H.-W. Six, and P. Widmayer. The R-file: an efficient access structure for proximity queries. In *Proceedings of the 6th IEEE International Conference on Data Engineering*, pages 372–379, Los Angeles, Feb. 1990.

[91] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Reviews*, 41(3):59–72, 2007.

[92] C. S. Jensen, J. Kolář, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In E. Hoel and P. Rigaux, editors, *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems*, pages 1–8, New Orleans, LA, Nov. 2003.

[93] 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.

[94] D. B. Johnson. A note on dijkstra's shortest path algorithm. *Journal of the ACM*, 20(3):385–388, July 1973.

[95] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, Jan. 1977.

[96] T. R. Jones, F. Durand, and M. Desbrun. Non-iterative, feature-preserving mesh smoothing. *ACM Transactions on Graphics*, 22(3):943–949, July 2003.

[97] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *Proceedings of the 22nd IEEE International Conference on Data Engineering*, Atlanta, GA, Apr. 2006. online proceedings.

[98] G. Kedem. The quad-CIF tree: a data structure for hierarchical on-line algorithms. In *Proceedings of the 19th Design Automation Conference*, pages 352–357, Las Vegas, NV, June 1982. Also University of Rochester Computer Science Technical Report TR–91, September 1981.

[99] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, second edition, 1998.

[100] M. R. Kolahdouzan and C. Shahabi. Continuous k-nearest neighbor queries in spatial network databases. In J. Sander and M. A. Nascimento, editors, *Proceedings of the 2nd International Workshop on Spatio-Temporal Database Management STDBM'04*, pages 33–40, Toronto, Canada, Aug. 2004.

[101] M. R. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakely, and K. B. Schiefer, editors, *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 840–851, Toronto, Canada, Sept. 2004.

[102] M. R. Kolahdouzan and C. Shahabi. Alternative solutions for continuous k nearest neighbor queries in spatial network databases. *GeoInformatica*, 9(4):321–341, Dec. 2005. Also *Proceedings of the 2nd International Workshop on Spatio-Temporal Database Management STDBM'04*, J. Sander and M. A. Nascimento, eds., pages 33–40, Toronto, Canada, August 2004.

[103] R. Krauthgamer and J. R. Lee. Navigating nets: simple algorithms for proximity search. In *SODA '04: Proceedings of the 15th annual ACM-SIAM symposium on Discrete algorithms*, pages 798–807, New Orleans, LA, Jan. 2004.

[104] M. Levoy, K. Pulli, B. Curless, S. Rusinkiewicz, D. Koller, L. Pereira, M. Ginzton, S. Anderson, J. Davis, J. Ginsberg, J. Shade, and D. Fulk. The digital Michelangelo project: 3D scanning of large statues. In *Proceedings of the SIGGRAPH'00 Conference*, pages 131–144, New Orleans, LA, July 2000.

[105] M. Levoy and T. Whitted. The use of points as display primitive. Technical Report 85–022, Univeristy of North Carolina at Chapel Hill, Chapel Hill, NC, Jan. 1985.

[106] F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. On trip planning queries in spatial databases. In C. B. Medeiros, M. J. Egenhofer, and E. Bertino, editors, *Advances in Spatial Databases— 9th International Symposium, SSD'05*, volume 3633 of *Lecture Notes in Computer Science*, pages 273–290, Angra dos Reis, Brazil, Aug. 2005. Springer.

[107] M. D. Lieberman, H. Samet, **J. Sankaranarayanan**, and J. Sperling. STEWARD: architecture of a spatio-textual search engine. In H.Samet, M. Schneider, and C. Shahabi, editors, *Proceedings of the 15th ACM International Symposium on Advances in Geographic Information Systems*, pages 186–193, Seattle, WA, Nov. 2007.

[108] M. D. Lieberman, **J. Sankaranarayanan**, H. Samet, and J. Sperling. Augmenting spatio-textual search with an infectious disease ontology. In *Proceedings of the Workshop on Information Integration Methods, Architectures, and Systems (IIMAS08)*, Cancun, Mexico, Apr. 2008. To appear.

[109] 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 Journal on Computing*, 35(1):22–58, Sept. 2005. Also see *Proceedings of the 10th International Conference on Pattern Recognition*, vol. II, pages 91–96, Atlantic City, NJ, June 1990 and University of Maryland Computer Science Technical Report TR–3455.1, February 2000.

[110] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. In *Proceedings of the 35th IEEE Annual Symposium on Foundations of Computer Science*, pages 577–591, Santa Fe, NM, Nov. 1994.

[111] N. Linial, E. London, and Y. Rabinovich. The geometry of graphs and some of its algorithmic applications. *Combinatorica*, 15:215–245, 1995. Also see *Proceedings of the 35th IEEE Annual Symposium on Foundations of Computer Science*, pages 577–591, Santa Fe, NM, November 1994.

[112] S. P. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):127–135, Mar. 1982.

[113] D. Luebke, M. Reddy, J. D. Cohen, A. Varshney, B. Watson, and R. Huebner. *Level of Detail for 3D Graphics*. Morgan-Kaufmann, San Francisco, 2003.

[114] X. Ma, S. Shekhar, H. Xiong, and P. Zhang. Exploiting a page-level upper bound for multi-type nearest neighbor queries. In *Proceedings of the 14th ACM International Symposium on Advances in Geographic Information Systems*, pages 179–186, Arlington, VA, Nov. 2006.

[115] 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*, volume 3, pages 1359–1365, Las Vegas, NV, June 2004. CSREA Press.

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

[117] N. J. Mitra and A. Nguyen. Estimating surface normals in noisy point cloud data. In *Proceedings of the 19th ACM Symposium on Computational Geometry*, pages 322–328, San Diego, CA, June 2003. ACM.

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

[119] D. M. Mount and S. Arya. ANN: a library for approximate nearest neighbor searching. In *Proceedings of the 2nd Annual Center for Geometric Computing Workshop on Computational Geometry*. electronic edition, Durham, NC, Oct. 1997.

[120] D. A. Murio. *The mollification method and the numerical solutions of Ill-posed problems*. Wiley, New York, NY, 1993.

[121] G. Narasimhan and M. Smid. Approximating the stretch factor of euclidean graphs. *SIAM J. Comput.*, 30(3):978–989, 2000.

[122] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, Aug. 1986. Also in *Proceedings of the SIGGRAPH'86 Conference*, Dallas, TX, August 1986.

[123] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *Proceedings of the ACM SIGMOD Conference*, pages 270–277, San Francisco, May 1987.

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

[125] J. A. Orenstein. Redundancy in spatial databases. In *Proceedings of the ACM SIGMOD Conference*, pages 294–305, Portland, OR, June 1989.

[126] D. Papadias, Y. Tao, K. Mouratidis, and C. K. Hui. Aggregate nearest neighbor queries in spatial databases. *ACM Transactions on Database Systems*, 30(2):529–576, June 2005.

[127] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors, *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, pages 802–813, Berlin, Germany, Sept. 2003.

[128] M. Pauly, M. Gross, and L. P. Kobbelt. Efficient simplification of point-sampled surfaces. In *Proceedings of the conference on Visualization 2002*, pages 163–170, Boston, MA, Oct. 2002. IEEE Computer Society.

[129] M. Pauly, R. Keiser, L. P. Kobbelt, and M. Gross. Shape modeling with point-sampled geometry. *ACM Transactions on Graphics*, 22(3):641–650, July 2003.

[130] D. Pfoser and C. S. Jensen. Indexing of network constrained moving objects. In E. Hoel and P. Rigaux, editors, *Proceedings of the 11th ACM International Symposium on Advances in Geographic Information Systems*, pages 25–32, New Orleans, LA, Nov. 2003.

[131] D. R. Reddy and S. Rubin. Representation of three-dimensional objects. Computer Science Technical Report CMU–CS–78–113, Carnegie-Mellon University, Pittsburgh, PA, Apr. 1978.

[132] P. Rigaux, M. Scholl, and A. Voisard. *Spatial Database Management Systems: Applications to GIS*. Morgan-Kaufmann, San Francisco, 2001.

[133] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD Conference*, pages 10–18, Ann Arbor, MI, Apr. 1981.

[134] K. A. Ross, I. Sitzmann, and P. J. Stuckey. Cost-based unbalanced R-trees. In *Proceedings of the 13th International Conference on Scientific and Statistical Database Management*, pages 203–212, Fairfax, VA, July 2001.

[135] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference*, pages 71–79, San Jose, CA, May 1995.

[136] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of the ACM SIGMOD Conference*, pages 17–31, Austin, TX, May 1985.

[137] S. Saltenis and C. S. Jensen. Indexing of moving objects for location-based services. In *Proceedings of the 18th IEEE International Conference on Data Engineering*, pages 463–472, San Jose, CA, Feb. 2002.

[138] H. Samet, , E. Tanin, and L. Golubchik. Scalable data collection infrastructure for digital government applications. In *Proceedings of the 5th National Conference on Digital Government Research*, pages 305–306, Atlanta, GA, May 2005.

[139] H. Samet. A quadtree medial axis transform. *Communications of the ACM*, 26(9):680–693, Sept. 1983. Also see CORRIGENDUM, *Communications of the ACM*, 27(2):151, February 1984 and University of Maryland Computer Science Technical Report TR–803, August 1979.

[140] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984. Also University of Maryland Computer Science Technical Report TR–1329, November 1983.

[141] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[142] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[143] H. Samet. Depth-first *k*-nearest neighbor finding using the MaxNearestDist estimator. In *Proceedings of the 12th International Conference on Image Analysis and Processing*, pages 486–491, Mantova, Italy, Sept. 2003.

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

[145] H. Samet. K-nearest neighbor finding using MaxNearestDist. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(2):243–252, Feb. 2008.

[146] 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. *Communications of the ACM*, 46(1):63–66, Jan. 2003.

[147] H. Samet, F. Brabec, and **J. Sankaranarayanan**. Importing abstract spatial data into the SAND database system. In *Proceedings of the 4th National Conference on Digital Government Research*, pages 285–286, Seattle, WA, May 2004.

[148] H. Samet, M. D. Lieberman, **J. Sankaranarayanan**, and J. Sperling. STEWARD: Demo of spatio-textual extraction on the web aiding the retrieval of documents. In *Proceedings of the 7th National Conference on Digital Government Research*, pages 300–301, Philadelphia, PA, May 2007.

[149] H. Samet, A. Phillippy, and **J. Sankaranarayanan**. Knowledge discovery using the SAND spatial browser. In *Proceedings of the 7th National Conference on Digital Government Research*, pages 284–285, Philadelphia, PA, May 2007.

[150] H. Samet and **J. Sankaranarayanan**. Maximum containing cell sizes in cover fieldtrees and loose quadtrees and octrees. Computer Science Technical Report TR–4900, University of Maryland, College Park, MD, Oct. 2007.

[151] H. Samet, **J. Sankaranarayanan**, and H. Alborzi. Scalable network distance browsing in spatial databases. Computer Science Technical Report TR–4865, University of Maryland, College Park, MD, Apr. 2007. Also in *Proceedings of the SIGMOD'08 Conference*, Vancouver, Canada, June 2008, To appear.

[152] H. Samet, **J. Sankaranarayanan**, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proceedings of the ACM SIGMOD Conference*, Vancouver, Canada, June 2008. Also University of Maryland Computer Science Technical Report TR–4865, April 2007. To appear.

[153] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics*, 4(3):182–222, July 1985. Also see *Proceedings of Computer Vision and Pattern Recognition'83*, pages 127–132, Washington, DC, June 1983 and University of Maryland Computer Science Technical Report TR–1372, February 1984.

[154] **J. Sankaranarayanan**, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *Proceedings of the 13th ACM International Symposium on Advances in Geographic Information Systems*, pages 200–209, Bremen, Germany, Nov. 2005.

[155] **J. Sankaranarayanan**, H. Alborzi, and H. Samet. Distance join queries on spatial networks. In *Proceedings of the 14th ACM International Symposium on Advances in Geographic Information Systems*, pages 211–218, Arlington, VA, Nov. 2006.

[156] **J. Sankaranarayanan**, H. Alborzi, and H. Samet. Enabling query processing on spatial networks. In *Proceedings of the 22nd IEEE International Conference on Data Engineering*, page 163, Atlanta, GA, Apr. 2006.

[157] **J. Sankaranarayanan**, H. Samet, and A. Varshney. Fast *k*-neighborhood algorithm for large point-clouds. In M. Botsch, B. Chen, M. Pauly, and M. Zwicker, editors, *Proceedings of the 3rd IEEE/Eurographics Symposium on Point-Based Graphics*, pages 75–84, Boston, July 2006.

[158] **J. Sankaranarayanan**, H. Samet, and A. Varshney. A fast all nearest neighbor algorithm for applications involving large point-clouds. *Computers & Graphics*, 31(2):157–174, Apr. 2007.

[159] **J. Sankaranarayanan**, E. Tanin, H. Samet, and F. Brabec. Accessing diverse geo-referenced data sources with the SAND spatial DBMS. In *Proceedings of the 3rd National Conference on Digital Government Research*, pages 331–334, 297, Boston, MA, May 2003.

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

[161] T. B. Sebastian and B. B. Kimia. Metric-based shape retrieval in large databases. In R. Kasturi, D. Laurendau, and C. Suen, editors, *Proceedings of the 16th International Conference on Pattern Recognition*, volume 3, pages 291–296, Quebec City, Canada, Aug. 2002.

[162] T. Seidl and H.-P. Kriegel. Optimal multi-step *k*-nearest neighbor search. In L. Hass and A. Tiwary, editors, *Proceedings of the ACM SIGMOD Conference*, pages 154–165, Seattle, WA, June 1998.

[163] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: a dynamic index for multi-dimensional objects. In P. M. Stocker and W. Kent, editors, *Proceedings of the 13th International Conference on Very Large Databases (VLDB)*, pages 71–79, Brighton, United Kingdom, Sept. 1987. Also University of Maryland Computer Science Technical Report TR–1795, 1987.

[164] K. Sevcik and N. Koudas. Filter trees for managing spatial data over a range of size granularities. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, pages 16–27, Mumbai (Bombay), India, Sept. 1996.

[165] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. In *Proceedings of the 10th ACM International Symposium on Advances in Geographic Information Systems*, pages 94–100, McLean, VA, 2002.

[166] 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. Also see *Proceedings of the 10th International Symposium on Advances in Geographic Information Systems*, A. Voisard and S.-C. Chen, eds., pages 94–100, McLean, VA, November 2002.

[167] M. Sharifzadeh, M. R. Kolahdouzan, and C. Shahabi. The optimal sequenced route query. *VLDB Journal*, 2008. To appear.

[168] M. Sharifzadeh and C. Shahabi. Spatial skyline queries. In U. Dayal, K.-Y. Whan, D. B. Lomet, G. Alonso, G. M. Lohman, M. L. Kersten, S. K. Cha, and Y.-K. Kim, editors, *Proceedings of the 32nd International Conference on Very Large Data Bases (VLDB)*, pages 751–762, Seoul, Korea, Sept. 2006.

[169] S. Shekhar and S. Chawla. *Spatial Databases: A Tour*. Prentice-Hall, Englewood-Cliffs, NJ, 2003.

[170] S. Shekhar, A. Kohli, and M. Coyle. Path computation algorithms for advanced traveller information system (atis). In *Proceedings of the 9th IEEE International Conference on Data Engineering*, pages 31–39, Vienna, Austria, Apr. 1993.

[171] S. Shekhar and T. A. Yang. Motion in a geographical database system. In O. Günther and H.-J. Schek, editors, *Advances in Spatial Databases—2nd Symposium, SSD'91*, vol. 525 of Springer-Verlag Lecture Notes in Computer Science, pages 339–358, Zurich, Switzerland, Aug. 1991.

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

[173] H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *Proceedings of the ACM SIGMOD Conference*, pages 343–354, Dallas, TX, May 2000.

[174] H.-W. Six and P. Widmayer. Spatial searching in geometric databases. In *Proceedings of the 4th IEEE International Conference on Data Engineering*, pages 496–503, Los Angeles, Feb. 1988.

[175] C. Sun, D. Agrawal, and A. E. Abbadi. Hardware acceleration for spatial selections and joins. In *Proceedings of the ACM SIGMOD Conference*, pages 455–466, San Diego, CA, June 2003.

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

[177] K. Talwar. Bypassing the embedding: algorithms for low dimensional metrics. In *Proceedings of the 36th Annual ACM Symposium on the Theory of Computing*, pages 281–290, Chicago, IL, USA, June 2004.

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

[179] M. Thorup and U. Zwick. Approximate distance oracles. In *Proceedings of the 33rd Annual ACM Symposium on the Theory of Computing*, pages 183–192, Hersonissos, Greece, 2001.

[180] G. Turk. Generating textures on arbitrary surfaces using reaction-diffusion. In *Proceedings of the SIGGRAPH'91 Conference*, pages 289–298, Las Vegas, NV, July 1991. ACM Press.

[181] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book.* Prentice Hall PTR, Upper Saddle River, NJ, 2001.

[182] T. Ulrich. Loose octrees. In M. A. DeLoura, editor, *Game Programming Gems*, pages 444–453. Charles River Media, Rockland, MA, 2000.

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

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

[185] P. M. Vaidya. An $O(n \log n)$ algorithm for the all-nearest-neighbor problem. *Discrete & Computational Geometry*, 4(1):101–115, 1989. Also see *Proceedings of the 27th IEEE Annual Symposium on Foundations of Computer Science*, pages 117–122, Toronto, Canada, October 1986.

[186] D. Wagner and T. Willhalm. Geometric speed-up techniques for finding shortest paths in large sparse graphs. In G. Di Battista and U. Zwick, editors, *Proceedings of the 11th Annual European Symposium on Algorithms (ESA 2003)*, vol. 2832 of

Springer-Verlag Lecture Notes in Computer Science, pages 776–787, Budapest, Hungary, Sept. 2003.

[187] 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, Canada, Jan. 2005. online proceedings.

[188] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, Jan. 1962.

[189] E. W. Weisstein. *Box-muller transformation.* MathWorld–A Wolfram Web Resource, Champaign, IL, 1999. `http://mathworld.wolfram.com/Box-MullerTransformation.html`.

[190] T. Weyrich, M. Pauly, S. Heinzle, R. Keiser, and S. Scandella. Post-processing of scanned 3D surface data. In *Proceedings of Eurographics Symposium on Point-Based Graphics*, pages 85–94, Zurich, Switzerland, June 2004. Eurographics Association.

[191] O. Wolfson, P. Sistla, B. Xu, J. Zhou, and S. Chamberlain. DOMINO: databases for moving objects tracking. In *Proceedings of the ACM SIGMOD Conference*, pages 547–549, Philadelphia, PA, June 1999.

[192] C. Xia, J. Lu, B. C. Ooi, and J. Hu. Gorder: an efficient method for KNN join processing. In M. A. Nascimento, M. T. Özsu, D. Kossmann, R. J. Miller, J. A. Blakely, and K. B. Schiefer, editors, *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB)*, pages 756–767, Toronto, Canada, Sept. 2004.

[193] M. L. Yiu, N. Mamoulis, and D. Papadias. Aggregate nearest neighbor queries in road networks. *IEEE Transactions on Knowledge and Data Engineering*, 17(6):820–833, June 2005.

[194] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse nearest neighbors in large graphs. *IEEE Transaction on Knowledge and Data Engineering*, 18(4):540–553, 2006.

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

[196] M. Zwicker, M. Pauly, O. Knoll, and M. Gross. Pointshop 3d: an interactive system for point-based surface editing. *ACM Transactions on Graphics*, 21(3):322–329, 2002.