

Scalable Network Distance Browsing in Spatial Databases

Seminar Report
Markus Cadonau
07-706-708

Department of Informatics
University of Zurich
Prof. Dr. M. Böhlen

May 31, 2010

As online map services have become popular, it is imperative for providers to deliver results to queries as fast as possible. Two common tasks performed on spatial networks, shortest path and k -nearest neighbor computation are looked at. For a static network, an existing best-first k -nearest neighbor algorithm is presented and discussed. While occupying minimal storage, the algorithm is trying to speed up finding results by using precomputed shortest path quadrees and estimating network distance ranges for possible nearest neighbors.

Contents

1	Trend Towards Location Based Web Services	4
2	Spatial Networks	5
2.1	Typical Properties of Road Networks and Their Representation	5
2.2	Reusing Road Network Graphs for Multiple Transportation Methods . . .	5
3	Common Tasks on Spatial Networks	6
3.1	Shortest Path Computation	7
3.2	Finding k Nearest Neighbors	7
4	Decoupling of Shortest Path Computation from Finding k Nearest Neighbors	8
4.1	Quadtree Representation for Shortest Paths	9
4.2	Using Pre-computed Quadtrees to Find k Nearest Neighbors	12
5	Potential and Limitations of Best-first k Nearest Neighbor Algorithm	14
6	Concluding Remarks and Outlook	16

1 Trend Towards Location Based Web Services

Nowadays many people own mobile devices, with which it is possible to do all kinds of communicational and other tasks even while being on the move. Such mobile devices call for specific services that can be used instantaneously and do not need any prior planning. Very typical mobile services that particularly take advantage of a user's mobility are location aware applications. While these applications seem to grow fast [5], they usually only make sense when the information about the current location can be used in a smart way. Besides games, especially navigation software for mobile phones has become popular. Although the phone is used to determine one's location, many navigation systems still operate with preinstalled maps, which occupy relatively large blocks of memory. With current and future reliable, high capacity mobile networks with faster and faster transfer speeds, however, it is also possible to access Web based services for obtaining directions. Maps and routes retrieved from servers have the advantage – if properly managed – of less likely being outdated and may also include additional information such as traffic situations.

In order to provide mobile users with desired directions, not only the mobile network has to be capable of doing so in a reasonable amount of time, but also the Web service itself needs to be efficient enough. Typical tasks users perform on map services are finding shortest routes between origin and destinations and looking for certain venues close by (e.g. the nearest few pizza places). In the near future, users might also expect map services to include live traffic data in their calculations and results. Following, after presenting the typical underlying model for road networks (see Section 2), two common tasks on such networks – namely, finding shortest paths and k nearest neighbors in terms of network distance – as well as existing algorithms will be looked at (see Section 3). In Section 4, an existing method [11] to efficiently perform these tasks on static networks will be presented. On one hand, it focuses on precomputing and efficiently storing and retrieving shortest path information (see Section 4.1). This is done by using regional similarities of the shortest paths. On the other hand, it uses estimates, again based on regional properties, to speed up finding the k nearest neighbors (see Section 4.2). The idea of spatial coherence in road networks is central to the approach. Finally, Section 5 discusses potential and limitations of introduced method, especially regarding non-static road networks.

2 Spatial Networks

Spatial networks are a subcategory of regular networks which specifically also represent and contain location elements. A classical network can be modeled as a graph $G(V, E)$, where V is the set of vertices and E the set of edges. Two vertices $a, b \in V$ are directly connected if there is an edge $e(a, b) \in E$. In spatial networks, for example, the geographical position is often associated with the vertices in respect to a coordinate system (i.e. latitude and longitude) [13]. In the context of this paper, as shortest path and nearest neighbor searches are looked at, a road network is taken as model theoretic view.

2.1 Typical Properties of Road Networks and Their Representation

A road network itself is one kind of spatial network. The graph $G(V, E)$ of such a network is normally weighted, since an edge $e(a, b) \in E$ with $a, b \in V$ represents a road segment from a to b and has either a distance, travel time, travel speed etc. associated [14]. $v \in V$ usually stands for an intersection, but can also be a point at the edge of the road network, a dead end. Therefore, the out degree of v is at least one. Often, for regular intersections, it is four. In general – except for few road networks where edges represent distances and no one-way streets exist – the graph G is directed. Even though most algorithms would also work on directed graphs, undirected graphs, for example representing pedestrian networks, shall be assumed for the further evaluations. Similarly, to be able to clearly demonstrate certain aspects of the algorithms, the road networks are assumed to be planar – basically meaning that tunnels and bridges, which under-/overpass other road segments, cannot always be represented adequately. Later, in Section 4.1, it will be seen what consequences non-planar networks could have on the presented method for storing shortest path information.

It should be clear that the weights of these road networks, based on actual streets and intersections, are non-negative.

2.2 Reusing Road Network Graphs for Multiple Transportation Methods

Instead of remodeling entire graphs, it can be efficient to introduce the multi-cost network model [9]. In a multi-cost network $G(V, E, W)$, W is a set of vectors. Each edge is

associated with a vector $w \in W$, where $w = (w_1, w_2, \dots, w_t)$. The index ($i = 1 \dots t$) represents the type of weight. This way, different information besides the actual distance can be represented; such as time needed on foot, by car, allowed and possible travel speeds, monetary costs (i.e. tolls) etc. Also, different weights could be stored for different travel times (as will be seen in Section 5).

3 Common Tasks on Spatial Networks

The two very typical tasks regarding spatial networks are finding the shortest path between two vertices a and b as well as finding the k nearest neighbors from a vertex q . At this point a clarification between network and spatial/euclidian distance (“as the crow flies”) is helpful: Network distance refers to the distance between two nodes/vertices on the network along the edges and their respective weights, whereas the spatial distance is the direct distance between two points in the spatial coordinate system (see Figure 1). Intuitively, latitude and longitude can be used as spatial coordinates; consistent arbitrary systems also work. Both, shortest path and nearest neighbor computation in an ordinary road network are based on network distance.

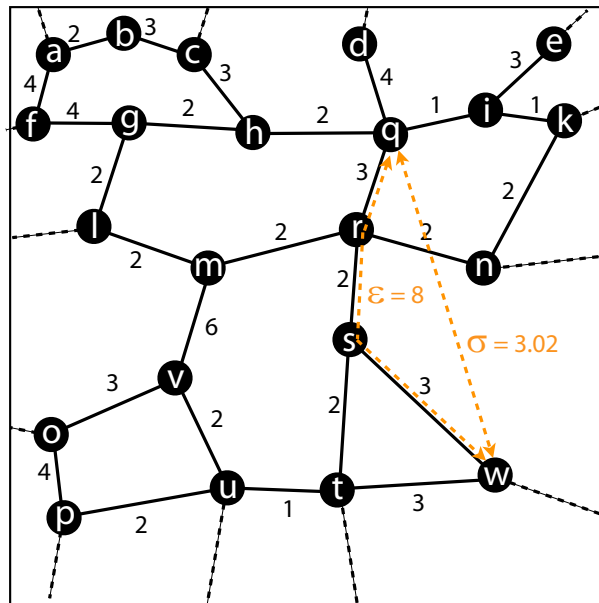


Figure 1: Network Distance ϵ and Spatial Distance σ between Nodes q and w

3.1 Shortest Path Computation

In order to compute the shortest path between a vertex a and a reasonably distant vertex b , one or another variant of Dijkstra’s algorithm [4] – or at least parts of it – is usually used [11]. In an average shortest path computational process, many vertices are visited and shortest paths to other vertices are found until eventually the desired node is reached. If a new query – or even repeatedly the same – has to be evaluated, Dijkstra’s algorithm does the computation all over again.

For static networks that do not frequently change (e.g. pedestrian networks) it might be worth to precompute the shortest paths and find efficient ways to store and retrieve them; especially if responses must be generated in real time for online services. It is obvious that for storage efficiency reasons not every vertex can store all shortest paths to all other vertices. In Section 4.1, one possible solution to partially store shortest paths, from which reconstruction is relatively simple, will be presented. Fast and efficient shortest path computation has been widely discussed ([6], [15], etc.) and will not be further addressed algorithmically within this context.

3.2 Finding k Nearest Neighbors

For finding the k nearest neighbors – in terms of network distance (see Section 3) – in spatial networks, shortest path computation, of course, plays an important role. Normally, however, in a k -nearest neighbor query (such as finding nearby cafes) one is not primarily interested in the exact network distances of the k objects; but rather in their location and their relative order. If a few of those objects are then closer looked at, directions and exact distances can still be evaluated.

A k -nearest neighbor algorithm is especially interesting in road networks. It can be used to find the k closest elements of a set O from the source vertex q (i.e. query vertex), where O can represent pizza places, banks, movie theaters etc. Very simple models use different graphs for each set of O , which requires a lot of storage. Other models include all different sets of O in one graph. This requires that, when doing a k -nearest neighbor search, all encountered objects be checked whether they belong to the specific set O which is queried. A more sophisticated approach, based on precomputed shortest paths and independent object sets, is shown in Section 4. Unlike algorithms derived from Dijkstra’s, it does not explore various directions and only visits vertices which are on the shortest path to objects.

There are two main approaches to finding the k nearest neighbors. The incremental versions of the k -nearest neighbor search allow for k to be unknown at start time. They incrementally look for the next closest element in the search domain; hence the name. In the other approach, k needs to be specified with the query. This tends to be beneficial in computational time when k is known in advance, as elements that are way too far away from the query point can immediately be disregarded. For this, however, a set of current k -nearest neighbor candidates needs to be maintained.

4 Decoupling of Shortest Path Computation from Finding k Nearest Neighbors

As mentioned in Section 3.1, shortest path search is demanding and requires a quite big computational effort, especially in large networks. Nonetheless, given that many map services are of online nature, results have to be delivered in (or close to) real time.

The question is – for a vast amount of queries – if there are elements that are computed over and over again even though little has changed. These elements could then be decoupled, precomputed and efficiently stored, so that a query can quickly retrieve them again.

In map services, queries often vary in their set of objects O and their query vertex q , while the underlying network of roads rarely changes. This means that users often search for different kinds of venues in a relatively limited area; or for shortest routes to different locations from their point of origin.

In a static pedestrian network, where weights of edges remain the same over longer periods of time (such as lengths of road segments), shortest paths also do not change frequently. They can therefore be precomputed, (partly) stored and then only have to be retrieved upon request. There is, as seen in other k -nearest neighbor queries in road networks [1], a tradeoff between query processing performance and storage overhead. Depending on the application, it has to be decided if the additional storage needed can be justified by faster results.

K -nearest neighbor searches can profit from fast shortest path retrieval. Since the underlying road network in both, shortest path and nearest neighbor computation is the same, a way has to be found to separate the object sets from the network. If a mapping between an object set O and the spatial network can be made, the same network can be reused for queries involving different sets of O .

Each object $o \in O$ can be associated with an edge/road – even a specific position on the edge – or a vertex/intersection of the road network, meaning that every object lies on the network. Otherwise, an object could not be reached or would have to be associated with multiple vertices close by, which would require additional steps to figure out through which of the associated vertices the shortest path leads. To simplify following thoughts about querying, the assumption is made that all objects as well as q are a subset of V (i.e. $O \subset V, q \in V$), the vertices. Now, it is possible to simply ask which edges/roads lead to an object o on the shortest path from q to o . Additionally, in order to browse through the objects of O while trying to find the k nearest neighbors, O has an own spatial index T (such as an R-Tree), which is called “search hierarchy” [11].

As an example of the decoupling of shortest path computation and nearest neighbor search, as well as object sets from the underlying spatial network, Section 4.1 discusses an existing idea [11] to store shortest paths in so called quadrees and Section 4.2 addresses the issue of finding the k nearest neighbors using those quadrees.

4.1 Quadtree Representation for Shortest Paths

Aside from doing the actual shortest path calculations, which eventually have to be done anyway, mainly two properties are important. The first is the storage needed to save the computed paths; the second is how fast they can be retrieved again. The brute force method would be to store the entire path for each pair of vertices.

The quadtree representation of the shortest paths [11] takes advantage of the fact that if x is known to be on the minimal path from a to b , the shortest path from a to x is also known [4]. In other words, if we know the first edge from a to b (i.e. a to x), we then only have to compute or find the shortest path from the next vertex (x) to b .

In a first improvement, this basically leads to adjacency lists. Each vertex has its adjacency list stored, which contains the next/first edge on the shortest paths to every other vertex (see Table 1, which corresponds to the network in Figure 1). Besides the fact that the space needed to store such lists is still bigger compared to other methods, the absence of an index leads to potentially costly sequential search when retrieving shortest paths [11].

In networks with a small out degree compared to the overall amount of vertices – in road networks typically four per intersection at most – many nodes share the same first path from a start point q . Instead of storing pairs of destination nodes and first edges for $N-1$ other vertices (if N is the number of all vertices and assuming that all are connected to the network), quadrees try to work with spatial coherence. It is likely that many vertices

Table 1: Adjacency List of Vertex q

Destination Vertex	First Vertex on Shortest Path
a	h
b	h
c	h
d	d
e	i
f	h
g	h
h	h
i	i
k	i
l	h
m	r
n	i
o	r
p	r
q	-
r	r
s	r
t	r
u	r
v	r
w	r

in an area have the same first edge on their shortest path from q . Existing quadtree solutions therefore group vertices into regions identified by their shared first edge [11]. For one vertex q , this works as follows.

For the vertex q there is a list of (or a reference to) all adjacent edges and their weights. Note that the weight would not be necessary only to retrieve the shortest path to a vertex, but will later help to determine whether a vertex belongs to the k nearest neighbors. Each edge is also associated with a region identifier (e.g. a simple integer or a color). Assuming that the first edges from origin q to each other vertex in the network have been computed and all vertices have a unique shortest path, it is possible to work with a region quadtree [10].

It is likely that the shortest paths in real-world based networks are unique. Should this not be the case, additional criteria such as number of intersections on the questionable paths – where a smaller amount would be seen as shorter – can be taken into consideration.

The mentioned quadtree first separates the whole spatial network, as the name implies,

into four equal squares, so that all vertices and edges are covered. From a central point of view, the four areas can be called Northwest (NW), Southwest (SW), Southeast (SE) and Northeast (NE). If all vertices within an area of this tree are also located in the same region of q – i.e., all vertices have the same first road segment on their shortest path from q – the identifier of the region is associated with this area in the quadtree, a so called Morton block [11]. In this quadtree representation, this block with only objects belonging to one region is a leaf-block/node. If not, refinement is needed for that particular area. The area then again gets separated into four subareas. This continues until there are no ambivalent areas left. It is also possible that one area does not contain any vertex and is empty. Access to a collection of Morton blocks, a quadtree, can be simplified by organizing them in a B⁺-tree structure [11].

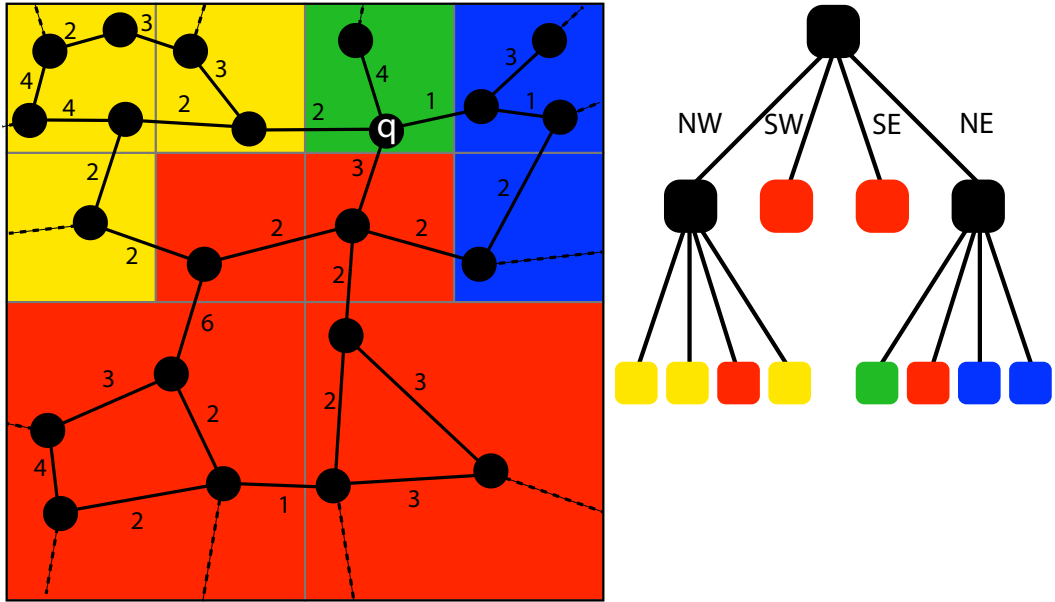


Figure 2: Shortest Path Quadtree Representation for Node q [11]

It is apparent that for regular road networks quadtrees on average require less storage space than adjacency lists. If a network was – contrary to the assumption made – non-planar, the quadtrees could look more scattered as the regions are not necessarily contiguous anymore. This would, of course, demand more storage space. Albeit, in real-world networks tunnels and bridges are too sparse to eradicate all spatial coherence. To retrieve the next edge on the shortest path, a simple point location operation is done on the quadtree representation. Given a destination's latitude and longitude, a query vertex q , its quadtree's center and initial square size, the quadtree of q (see Figure 2) can easily be traversed based on implicitly defined child areas. When a leaf-block is reached, one knows in which region relative to q the destination must be and can advance along the edge that defines the region to the next vertex t on the shortest path. t 's quadtree is then

processed next.

Using this quadtree method to store information about shortest paths, a shortest path query merely depends on the number of vertices on the shortest path from origin to destination. This is the reason why the approach is very scalable and works well for big networks.

However, while the retrieval of information from a quadtree is relatively inexpensive in computational cost, the calculations needed to build a large quadtree and the amount of storage that is necessary might not be worth the benefits. Thus, one idea would be to limit a quadtree to about the size of a city and define certain highways that connect the different “city-quadtrees”. This way, only shortest paths from entire sets of nodes (i.e. cities) to other sets would have to be stored additionally [14]. This probably works for countries like the United States, where relatively few major highways connect cities with each other. However, for very urban countries with a very connected road network topology it does not seem feasible.

4.2 Using Pre-computed Quadtrees to Find k Nearest Neighbors

With the introduced quadtree representation (see Section 4.1), it has already become easier to also find the k nearest neighbors. In order to use a developed best-first k -nearest neighbor algorithm [11], there are two more properties that have to be stored in each Morton block b of the quadtrees: λ^- and λ^+ . λ^- corresponds to the minimum value of the ratio of the network distance and the actual spatial distance (“as the crow flies”) from q to all destination vertices in b ; λ^+ corresponds to the maximum value, respectively. λ^- and λ^+ already allow to define a range in which a destination vertex w (located in b) has to be, based on the direct, spatial distance between q and w ; all without further processing/refining of the shortest path. By multiplying the spatial distance from q to w , easily calculated using their coordinates, with λ^- and λ^+ of the corresponding Morton block (in which w lies) of q ’s quadtree, an initial network distance range (δ^-, δ^+) from q to w can be estimated. An example for the calculation of λ as well as the use of it, is given for q and w (and its corresponding block) in Figure 3.

Trying to find the k nearest neighbors from q , the search hierarchy T – the set of objects in question, e.g. pizza places in one borough – first needs to be mapped to corresponding Morton blocks, as T can be an individual spatial index such as an R-tree. This is done [12] by first finding all Morton blocks of q ’s quadtree which are intersecting with the top

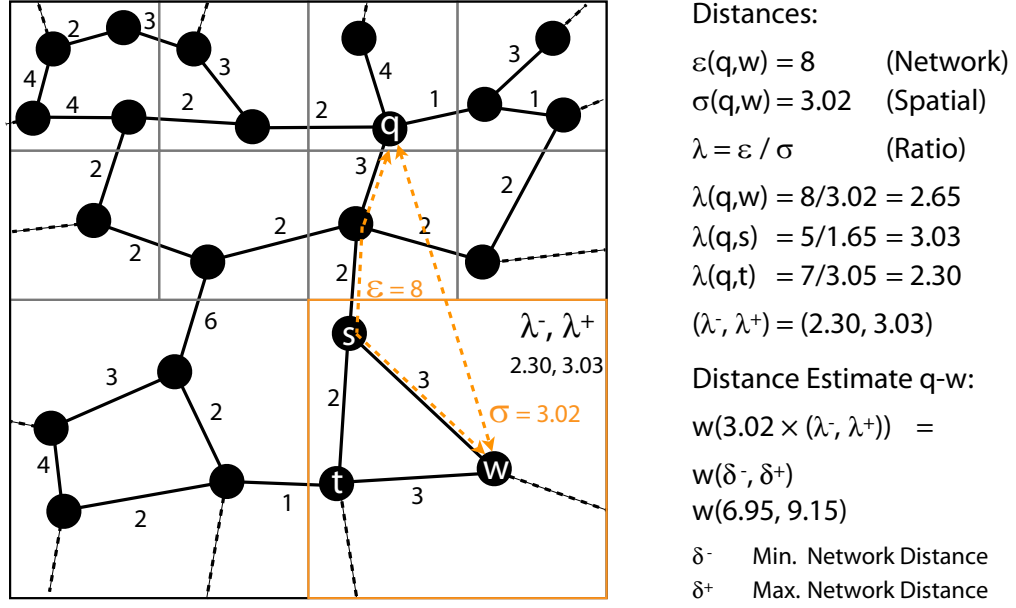


Figure 3: Storing and Calculating Network Distance Estimates Using Quadrees [11]

region of search hierarchy T . Then, minimum and maximum spatial distances from q to that region are multiplied with all λ^- and λ^+ of found blocks to find the initial minimum and maximum network distance estimates, δ^- and δ^+ , of all vertices in the region.

The aforementioned best-first k -nearest neighbor algorithm [11] operates with two priority queues, *Queue* and L . Briefly described, it works as follows: Let D_k be the (maximum network) distance of the k th nearest neighbor candidate. It is set to ∞ at the beginning. Initially, the top region/block of the search hierarchy T is placed in *Queue* along with its δ^- and δ^+ . *Queue* is ordered by the minimum network distances δ^- of its elements. *Queue* can contain following elements of the search hierarchy: non-leaf blocks, leaf blocks and objects. L contains the candidate set of the k nearest objects according to their maximum network distance. When retrieving an element from *Queue*, obviously the one with the shortest minimum network distance from q is handled first. L is also ordered with the smallest values first, but based on the maximum network distances. Elements in *Queue* are stored along with their minimum and maximum network distance; objects in L only with their maximum network distance. For objects o in *Queue*, two additional properties are stored: the vertex $V(o)$ where the refinement currently halted – i.e., a vertex on the shortest path from the query vertex q to the object, to which the exact network distance is already known – and said distance $I(o)$ from q to the refinement position.

The algorithm works on *Queue* – retrieving one element at a time – until the k nearest neighbors are found or as long as *Queue* is not empty. There are three types of elements *Queue* can return:

1. When a non-leaf block is encountered, each child element's minimum network distance is compared to the current D_k . If it is smaller than D_k , the element is again placed in *Queue* at the appropriate position according to the sorting conditions previously mentioned.
2. When a leaf block is retrieved from *Queue*, for each child object o in it, the network minimum and maximum distances, δ^- and δ^+ , are obtained by performing a point location operation in the shortest-path quadtree of q for the block containing the object. If the minimum network distance from q to o is smaller than D_k , o is put in the *Queue* again. If the maximum network distance is also smaller than D_k , o is inserted in L as well.
3. When an object o is removed from *Queue*, it is immediately disregarded and L is returned if its minimum network distance is bigger than D_k , since the remaining elements in *Queue* – as they are ordered – can also not be closer to q than D_k . If o 's maximum network distance is smaller than the *Queue*'s next element's minimum network distance, it implies that o 's maximum network distance is smaller than D_k and hence o is already in L – being a k nearest neighbor (not only a candidate). This is known because of the fact that if the next element's minimum network distance was not smaller than D_k , it would not be in *Queue*. If o 's maximum network distance is not smaller than the *Queue*'s next element's minimum network distance, a refinement operation is performed. Using the quadtree of o 's previously halted refinement position $V(o)$, the next edge is determined. Following the edge to the next vertex v , v 's quadtree can be used to look up the minimum and maximum network distance from v to o . The weight of the followed edge is added to the total weight of the previously followed edges, $I(o)$. The new minimum/maximum network distances from query vertex q to object o can now be generated by also adding the minimum/maximum network distances from v to o to the exact network distance from q to v . If the refined maximum network distance is now smaller than D_k , o will be inserted in L with the new maximum property and D_k be adjusted as necessary. If o 's new minimum network distance is still smaller than D_k , it will be placed in *Queue* again with its refined properties.

5 Potential and Limitations of Best-first k Nearest Neighbor Algorithm

Shortest path and k -nearest neighbor queries can be relatively efficiently computed in static spatial networks. Presented best-first k -nearest neighbor algorithm (as seen in Section 4) uses pre-computation of shortest path quadtrees. It is safe to say that it makes

sense to apply it on pedestrian networks. Assuming the weights of the edges of the network correspond to actual distances, results can even be converted to time durations given an average speed of a pedestrian.

It is theoretically demonstrated that the quadtree representation significantly reduces storage costs for shortest paths and it is experimentally shown that this best-first k -nearest neighbor algorithm is well-suited for smaller values of k , i.e. $k < 20$ [11]. A typical user is probably not interested in more than this amount of nearest neighbors of a certain domain (such as restaurants of one kind). Therefore, an incremental k -nearest neighbor algorithm is also mostly not necessary and presented best-first algorithm with a reasonable default k can perform faster. A big k as well as similar distances of many objects in a set to q can slow down the best-first k -nearest neighbor algorithm compared to others, because the priority queues grow bigger and operations on them are computationally more demanding. Both is not very likely to occur in real-world network applications; hence the algorithm can be useful for many tasks.

However, pre-computation relies on fixed weights of the edges: a static network. It is well known that traffic plays a major role in road networks when one is trying to quickly get from a to b . For this reason, for modern real-world map applications traffic should also be considered in shortest path suggestions and related k -nearest neighbor searches.

So, for road networks, the picture looks different than for pedestrian networks. While only different speed limits and average speeds for different road segments alone do not impose a big problem on pre-computing the shortest paths yet, constantly changing speeds in traffic flow for many road segments do. There are numerous approaches trying to deal with this issue and model the time-dependent component of road networks, meaning that weights of edges can vary over time. One study introduces probabilistic edges for this purpose [7]. As seen in Section 2.2, it is possible to store multiple weights for different time instances or intervals in a network's weight vectors. Another study tries to find functions based on historic traffic data that report the expected traffic for specific areas during certain times [3]. Patterns for locations and times of a day are tried to be found.

A simple, arguably naive approach would be to discretize over time and have separate graphs and shortest path computations for each time instant (e.g. for a duration of a day or week), also based on experienced traffic data. This would require a huge computational effort, not to mention lots of memory. Such different snapshots could be combined to a time-expanded network, but it still does not seem to be very efficient [2]. For a k -nearest neighbor search, another variant of Dijkstra's algorithm [4] has been introduced where the weight of the adjacent edges can only be determined knowing the time needed to reach

the current vertex [2]. In general, a spatial network is usually transformed into a traffic time network [8].

Albeit, true (near-) live traffic data has hardly been considered so far. As of now, Google Maps for example shows live traffic information and suggests different routes, of which both regular travel time and the time needed when hold up in traffic can be seen. However, it does not seem as if live traffic data had an influence on returned search results for time needed to travel.

6 Concluding Remarks and Outlook

A best-first k -nearest neighbor algorithm, which makes use of precomputed shortest paths, has been shown. Shortest path information is efficiently stored in quadrees using spatial coherence and can be retrieved with an effort depending on intermediate vertices. The best-first k -nearest neighbor algorithm benefits from precomputed shortest paths as well as from network distance estimates for entire blocks of the stored quadrees due to the fact that object sets are decoupled from the underlying network, of which the properties do not change as frequently as other query elements. For static networks, such as pedestrian networks, and a fairly small k , the algorithm outperforms most others. However, it does not take into account that travel times on roads can change due to traffic. In future studies, it will be interesting to see if precomputed quadrees and live traffic data can somehow be efficiently merged.

References

- [1] J. Bao, C.-Y. Chow, M. F. Mokbel, W.-S. Ku: *Efficient Evaluation of k -Range Nearest Neighbor Queries in Road Networks*; MDM 2010.
- [2] U. Demiryurek, F. Banaei-Kashani, C. Shahabi: *Towards K -Nearest Neighbor Search in Time-Dependent Spatial Network Databases*; DNIS 2010, pp. 269-310.
- [3] U. Demiryurek, B. Pan, F. Banaei-Kashani, C. Shahabi: *Towards Modeling the Traffic Data on Road Networks*; IWCTS '09: Proceedings of the Second International Workshop on Computational Transportation Science, Seattle, WA, USA, 2009, pp. 13-18.
- [4] E. W. Dijkstra: *A Note on Two Problems in Connexion with Graphs*; Numerische Mathematik 1, 1959, pp. 269-271.

- [5] D. Fletcher: *10 Tech Trends for 2010*; TIME, March 22, 2010, (Available:) http://www.time.com/time/specials/packages/article/0,28804,1973759_1973760,00.html.
- [6] A. V. Goldberg, C. Harrelson: *Computing the Shortest Path: A* Search Meets Graph Theory*; SODA '05: Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms, Vancouver, Canada, 2005, pp. 156-165.
- [7] M. Hua, J. Pei: *Probabilistic Path Queries in Road Networks: Traffic Uncertainty Aware Path Selection*; EDBT '10: Proceedings of the 13th International Conference on Extending Database Technology, Lausanne, Switzerland, 2010, pp. 347-358.
- [8] W.-S. Ku, R. Zimmermann, H. Wang, C.-N. Wan: *Adaptive Nearest Neighbor Queries in Travel Time Networks*; GIS '05: Proceedings of the 13th annual ACM international workshop on Geographic information systems, Bremen, Germany, 2005, pp. 210-219.
- [9] K. Mouratidis, Y. Lin, M. L. Yiu: *Preference Queries in Large Multi-Cost Transportation Networks*; ICDE '10: Proceedings of the IEEE International Conference on Data Engineering, Long Beach, CA, USA, 2010.
- [10] H. Samet: *The Quadtree and Related Hierarchical Data Structures*; ACM Computing Surveys, Vol. 16(2), New York, NY, USA, 1984, pp.187-260.
- [11] H. Samet, J. Sankaranarayanan, H. Alborzi: *Scalable Network Distance Browsing in Spatial Databases*; SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, Vancouver, Canada, 2008, pp. 45-54.
- [12] J. Sankaranarayanan: *Scalable Query Processing on Spatial Networks*; Dissertation, University of Maryland, Department of Computer Science, USA, 2008, pp. 1-360.
- [13] J. Sankaranarayanan, H. Alborzi, H. Samet: *Efficient Query Processing on Spatial Networks*; GIS '05: Proceedings of the 13th annual ACM international workshop on Geographic information systems, Bremen, Germany, 2005, pp. 200-209.
- [14] J. Sankaranarayanan, H. Samet: *Distance Oracles for Spatial Networks*; ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering, Washington, DC, USA, 2009, pp. 652-663.
- [15] F. B. Zhan, C. E. Noon: *Shortest Path Algorithms: An Evaluation Using Real Road Networks*; Transportation Science, Vol. 32(1), 1998, pp. 65-73.