ISSN 1346-5597

National Institute of Informatics

NII Technical Report

Levelwise Mesh Sparsification for Shortest Path Queries

Yuichiro Miyamoto, Takeaki Uno, Mikio Kubo

NII-2008-004E
March 2008

URL: http://www.nii.ac.jp/ Tokyo 101-8430, JAPAN +81-3-4212-2000 FAX: +81-3-3556-1916

Levelwise Mesh Sparsification for Shortest Path Queries

Yuichiro Miyamoto[1], Takeaki Uno[2], and Mikio Kubo[3]

[1] Sophia University, Kioicho 7-1, Chiyoda-ku, Tokyo 102-8554, Tokyo, Japan,
y-miyamo@sophia.ac.jp,
[2] National Institute of Informatics, Hitotsubashi 2-1-2, Chiyoda-ku, Tokyo 101-8430, Tokyo, Japan,
uno@nii.jp,
[3] Tokyo University of Marine Science and Technology, Etchujima 2-1-6, Koto-ku, Tokyo 135-8633, Tokyo, Japan,
kubo@kaiyodai.ac.jp

Abstract. The shortest path problem is one of the most fundamental problems in computer science. Although several quasi linear time algorithms have been proposed, even an optimal algorithm does not terminate in sufficiently short time for large-scale networks, such as web networks and road networks. An approach to solving this problem is to construct data structures that enable us to find solutions in a short time. This approach is efficient for applications in which the networks are fixed and we have to answer possibly many questions in a short time, such as in car navigation systems. In this paper, we propose the levelwise mesh sparsification method for the problem. Several sparse networks are obtained by sparsifying the original network, and the shortest path problem is solved by finding the shortest path in these networks. The obtained networks are sparse and small, thus the computational time is short. Computational experiments on real world data show the efficiency of our method in terms of computational time and memory efficiency. Compared with existing approaches, the advantage of our method is that it can deal with negative costs and time-dependent costs, and uses only a small amount of memory.

1 Introduction

The shortest path is one of the most fundamental problems in computer science. The problem is studied well especially in the areas of optimization and algorithms, and it has many applications in theory and in practice. For example, dynamic programming is basically solving the shortest path on a table, and the computation of the edit distance of two strings is reduced to the shortest path problem. In the real world, car navigation systems utilize shortest path algorithms, and Internet packet routing needs a short path to the destination. Some problems in artificial intelligence and model checking need to solve shortest path problems to get the feasibility of the problem. In fact, there are more and more applications.

The shortest path problem can be solved with Dijkstra's algorithm [1] in $O(m + n \log n)$ time, where n and m are respectively the number of vertices and edges in the network to be solved. Recently, sophisticated quasi-linear time algorithms have been proposed [2]. These algorithms have a restriction that the distance of any edge has to be non-negative, but this restriction is quite natural in many applications, however, some recent applications require the problem to be solved in a very short time. For example, a car navigation system must solve the shortest path problem in a network having a huge number of edges; e.g., the US road network has 58 million edges. In such applications, linear time is too long; Dijkstra's algorithm may take several seconds to solve the problem, but users likely can not wait such a long time before starting driving. In on-line navigation services, the server system has to respond with the shortest path in quite a short time, say 0.01 second. We need much faster shortest path algorithms for these problems and for applications that need to compute shortest paths many times.

In such applications, the network to be solved does not change, or changes not so frequently or not so drastically. Thus, this is a natural motivation to construct a data structure that reduces the computational time. Such an approach can be considered as a database query; thus we call this problem the *shortest path query*.

In this paper, we propose a method called *Levelwise Mesh Sparsification* (LMS). LMS constructs sparsified networks based on a geometric partition $R_1, ..., R_q$ of the network. For each R_i, we define an outer region Q_i as a larger region including R_i and construct the sparsified network S_i of R_i composed of edges included in the shortest path connecting some pairs of vertices on the boundary of Q_i. To find the shortest path connecting two vertices outside of Q_i, we have

to look at only the edges in S_i. To get to distant destinations on real-world road networks, we can observe that narrow, bent or short roads are not used; only wide, straight, and long roads are used. Thus, we expect a sparsified network to have few edges and that the computational time can be shortened as a result. In LMS, we prepare many regions with different shapes and sizes, such as squares of sizes $c \cdot 2^k$ for some c, and combine them so that the sparsified networks would contain much fewer edges. The computational experiments show that such ideas dramatically reduce the edges; thus LMS should be quite efficient in practice.

Related Work

Recent researches have dealt with mainly three techniques: highway hierarchy, bit vector, and transit node routing [3–7]. The highway hierarchy is similar to ours and the ordinary layer network approach. The layer network approach considers the set of edges which are frequently used when going to farther destinations and uses the network as a highway. However, it has no assurance of optimality; hence it definitely differs from more recent algorithms despite it likely being the most popular method in modern car navigation systems.

Suppose that an edge e is included in the shortest path from v to u, and an endpoint of e is the k_vth closest vertex to v, and the other endpoint is the k_uth closest vertex to u. If both k_v and k_u are larger than a threshold value h, we call e a highway. The highway hierarchy method [5, 6] constructs a highway network composed of highway edges. We start by executing Dijkstra's algorithm on the original network, then go to the highway network after h steps. The highway network is usually sparser than the original network; thus the computational time is shortened. Moreover, by constructing highway network upon highway network recursively, the computational time becomes much shorter. In contrast to the layer method, the highway hierarchy method does not lose optimality. Short preprocessing time is also an advantage of this method.

Bit vector [4] considers a partition of the network into regions $R_1, \ldots, R_q$. In the preprocessing phase, for each pair of vertex v and region $R_i, v \notin R_i$, it marks R_i on all the edges incident to v which are included in the shortest path from v to a vertex in R_i. The search for the shortest path from v to a vertex u in R_i is restricted to the edges marked with "R_i". If v is far from R_i, such edges are few; thus, until the current visited vertex is close to the destination, the edges searched likely form a path. Therefore, this method can save a lot of computational time.

The highway hierarchy method finds the edges common to the middle of the shortest paths. In contrast to this, bit vector finds a common structure to the shortest paths between a vertex and a region. When the source vertex is close to the region including the target vertex, the edges to be examined become many, and hence computational time increases. By increasing the number of the regions q, the bit vector can increase the efficiency instead of increasing memory usage. Moreover, the preprocessing of the bit vector needs to solve the all pairs shortest paths problem, and thereby it takes a long time.

The third method is transit node routing [7]. In the preprocessing phase, it selects the transit vertices such that for any pair of distant vertices, at least one transit vertex is included in their shortest path. Then, we compute all pairs shortest paths of the transit vertices, and these are stored as a data structure. When we execute the Dijkstra's algorithm, we can directly move from a transit vertex near the source to a transit vertex near the destination. In practice, every node has a transit node in its neighbors; thus we can find the shortest path in a short time (usually in $O(1)$ time).

Advantage and Contribution

Transit node routing is the fastest of these three algorithms. In terms of memory usage, highway hierarchy has an advantage, but it admits only a two way search; thus it can not solve problems with time-dependent costs. Moreover, it can not handle negative costs. On the other hand, the memory usage of LMS is quite small. In the experiments, the size of the additional data is less than 1/10th of the original network, and this is a big advantage for real-world uses. Moreover, it admits a one way search; thereby, it allows negative costs and time-dependent costs. In summary, the advantages of each algorithm are as follows.

	time	preprocess	update	memory	negative cost/ time dependent cost
bit vector	△	×	×	△	◯
highway hierarchy	△	◯	△	△	×
transit node routing	◯	×	×	×	◯
LMS (ours)	△	△	◯	◯	◯

The idea of LMS is similar to that of the highway hierarchy method in the point of constructing levelwise (hierarchy) sparse networks. A major difference is that LMS uses a geometric partition. The layered highway networks of the highway hierarchy method are densely connected everywhere to each other. In contrast, the sparsified networks of LMS are connected only at their borders; that is, they are not densely connected. Moreover, LMS is suited to geometry-based operations such as change of the network. When the network changes, we only have to update the regions whose outer regions include the change. In fact, LMS is not an algorithm but rather a way to construct a sparse network. Thus, it can adopt additional constraints and use the other modern shortest path algorithms such as the A^* algorithm and bit vector.

To construct a sparsified network, we need to find shortest paths for all pairs of vertices on the boundary of the outer region. This takes a longer time than the highway hierarchy method takes. However, by recursively using the sparsified network inside the outer region, we can reduce the computational time.

Organization of the paper

The remainder of this paper is organized as follows. Section 2 describes the definitions and the notations. Section 3 explains our sparsification method and states lemmas that assure the optimality of the solution. A geometric implementation and a general framework are also proposed. Section 4 describes possible extensions and generalizations of our method. We show the results of the computational experiments in Section 5 and conclude the paper in Section 6.

2 Preliminaries

Let $\mathbb{R}^+$ be the set of positive real numbers. Let $G = (V, A, d)$ be a simple directed network: V is a vertex set, A is an edge set, $d : A \to \mathbb{R}^+$ is a positive distance function on the edges. An ordered sequence of edges $((v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k))$ is called a v_1-v_k path of G. The vertices v_1 and v_k are called end vertices of the path. The length of a path is the sum of distances of all edges of the path. A shortest s-t path is an s-t path whose length is shortest among all s-t paths. In general, a shortest s-t path is not unique. For a given network and a query specified by a source vertex s and a target vertex t, the shortest s-t path problem is to determine one of the shortest s-t paths. In some context, the shortest s-t path problem requires only the length of a shortest s-t path.

For a vertex v, let $N(v)$ be the set of neighbors of $v \in V$ defined by $N(v) = \{w \in V \mid (w, v) \in A \text{ or } (v, w) \in A\}$. Let $N(S)$ be the set of neighbors of $S \subseteq V$ defined by $N(S) = \{N(v) \mid v \in S\} \setminus S$. The subnetwork of G induced by $U \subseteq V$ is denoted by $G[U]$. The union of two graphs G_1 and G_2 is defined by the graph whose vertex set and edge set are the union of their vertex sets, and edge sets, respectively.

3 Levelwise Mesh Sparsification

Here, we describe levelwise mesh sparsification (LMS) when a network $G = (V, A, d)$ is given. The purpose of the method is to find edges that can be used in some shortest paths. We define sparsified networks and meshes on the base of a geometric partition in subsection 3.1, and show the way to answer a shortest path query in subsection 3.2. Subsection 3.3 describes a technique for speeding up the construction of sparsified networks.

3.1 Geometric Implementation

Hereafter we assume that all vertices are embedded on a 2-dimensional plane; thus each vertex has a 2-dimensional coordinates. This assumption is reasonable in the context of geometric networks such as road networks and railroad networks. Let $x(v)$ and $y(v)$ be the x- and y-coordinates of v, respectively. A square region is specified by its height (width) and the point with the smallest coordinates. A region $R(c, i, j)$ is a square region such that $R(c, i, j) = \{(x, y) \in \mathbb{R}^2 \mid i \cdot c \le x < (i + 1)c,\ j \cdot c \le y < (j + 1)c\}$, where c is a real constant. Clearly, the set of $R(c, i, j),\ \forall i, j \in \mathbb{Z}$ is a partition of a 2-dimensional plane. We define an outer region $R_{\text{outer}}(c, i, j)$ of $R(c, i, j)$ by $R_{\text{outer}}(c, i, j) = \{(x, y) \in \mathbb{R}^2 \mid (i - 1)c \le x < (i + 2)c,\ (j - 1)c \le y < (j + 2)c\}$. Let $V(c, i, j)$ and $V_{\text{outer}}(c, i, j)$ be a subset of V defined by $V(c, i, j) = \{v \in V \mid (x(v), y(v)) \in R(c, i, j)\}$ and $V_{\text{outer}}(c, i, j) = \{v \in V \mid (x(v), y(v)) \in R_{\text{outer}}(c, i, j)\}$, respectively. Clearly $V_{\text{outer}}(c, i, j)$ includes $V(c, i, j)$. Without loss of generality, we assume that $0 \le x(v), y(v)$ and that $\exists L \in$

$\mathbb{R},\ x(v), y(v) < L,\ \forall v \in V$. Under this assumption, it is sufficient to consider a finite number of subsets $V(c,i,j)$ and $V_{\text{outer}}(c,i,j)$ for a given constant c.

Our idea to find edges that can be used in some shortest paths is based on the observation that in real-world networks, only a few edges in an area are used in the middle of the shortest paths connecting distant vertex pairs. This motivates us to find all shortest paths connecting vertices outside of V_{outer} and to identify the edges that can be included in some of the shortest paths. To find a shortest path connecting distant vertices, we only have to refer to such edges in the middle of the search; thus we can dramatically reduce the number of edges that have to be looked at. The reason for using V_{outer} instead of directly using $V(c,i,j)$ is that possibly almost all edges near the boundary of R_{outer} will be used by some shortest path. Now let us define the *sparsified network* that is the network obtained by the above idea.

Definition 1. *A sparsified network of G derived by $V(c,i,j)$ is the subnetwork of $G[V(c,i,j) \cup N(V(c,i,j))]$ such that each edge is on the shortest path of G connecting two vertices outside of $V_{outer}(c,i,j)$.*

Figure 1 shows an illustration of finding the edges of a sparsified network. Small and large squares are $R(c,i,j)$ and $R_{\text{outer}}(c,i,j)$, respectively. The bold lines on the left are edges of a shortest path connecting two vertices outside of the outer region. The bold lines on the right are edges to be included in the sparsified network.

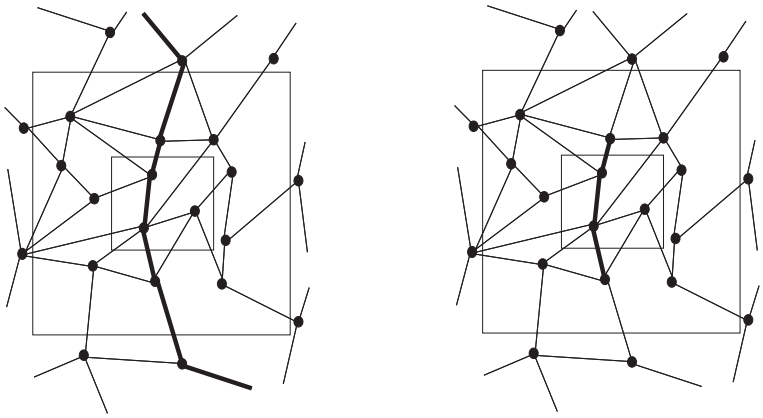

Fig. 1. Illustration of finding a sparsification network

We denote the sparsified network of G derived by $V(c,i,j)$ by $G'(c,i,j)$.

The operation "*replace of $G'(c,i,j)$*" is the operation to construct the network obtained from G by removing edges in $G[V(c,i,j)]$ and adding edges in $G'(c,i,j)$. More precisely, the constructed graph is the union of $G[V \setminus V(c,i,j)]$ and $G'(c,i,j)$. From the definition of the sparsified network, we have the following lemma.

Lemma 1. *All shortest paths connecting two vertices outside of $V_{outer}(c,i,j)$ are included in the network G replaced by $G'(c,i,j)$.*

Note that a sparsified network $G'(c,i,j)$ can be obtained by finding all shortest paths connecting two vertices in $N(V_{\text{outer}}(c,i,j))$.

We say an edge is *inside* of $G'(c,i,j)$ if both end vertices are in $V(c,i,j)$. Next, let us define a *sparsified mesh* that is an edge contracted network of the sparsified network. *Edge contractions* of a network G is following procedures;

- (One way): if only two inside edges (u,v) and (v,w) are incident to a vertex $v \in V(G)$, remove them and add (u,w) whose distance is $d(u,v) + d(v,w)$,
- (Bi-direction): if only four inside edges (u,v), (v,u), (v,w) and (w,v) are incident to a vertex $v \in V(G)$, remove them and add edges (u,w) and (w,u) whose distances are $d(u,v) + d(v,w)$ and $d(w,v) + d(v,u)$, respectively.

Definition 2. *A sparsified mesh of G derived by $V(c,i,j)$ is a minimal network obtained by edge contractions.*

We denote a sparsified mesh of G derived by $V(c,i,j)$ by $G(c,i,j)$. The replacement of a sparsified mesh is defined in the same way.

Lemma 2. *The length of a shortest s-t path of G is equal to that of G replaced by $G(c,i,j)$, if both s and t are outside of $V_{outer}(c,i,j)$.*

We say that a sparsified mesh $G(c,i,j)$ is *valid* for a pair of vertices $\{s,t\}$, if both s and t are out of $V_{\text{outer}}(c,i,j)$.

Lemma 3. *The length of a shortest s-t path on G is equal to that on G replaced by a valid $G(c,i,j)$ for $\{s,t\}$.*

If the size of square regions c is small, the number of square regions (that is the number of sparsified networks) will be large; thus the network to be processed will not be sparse. On contrary, we have to access many edges to get valid sparsified networks if c is large. To offset these disadvantages, we introduce a *levelwise sparsified mesh*, in which we introduce a hierarchic structure of sparsified networks.

Definition 3. *For a given constant C, a* levelwise sparsified mesh *of G is a collection of sparsified meshes $G(2^kC,i,j)$ for a positive number k.*

A sparsified mesh in $G(2^kC,i,j)$ is called a *k-level mesh*. In particular, we call $G[V(C,i,j)]$ a 0-level meshe. All 0-level meshes are valid for all pairs of vertices. We call a square region $R(2^kC,i,j)$ corresponding to a mesh $G(2^kC,i,j)$ a region of the mesh. The *scale* of the k-level mesh is the size of regions of meshes. When the source vertex s and target vertex t are given, we use a *query network* that is constructed from a combination of valid meshes of many levels. Figure 2 illustrates a network used for finding the s-t shortest path. The details of the construction of the query network are described in the next subsection.

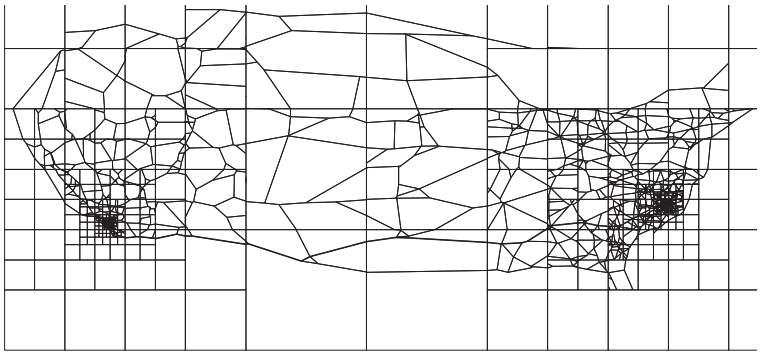

Fig. 2. Example of a query network and a shortest path

3.2 Shortest Path Query

To make the query network sparse, we should use as high level meshes as possible. Thus, we construct one by combining maximal valid meshes. A k-level valid mesh is called *maximal* if its region is not covered by a region of any mesh of level $k+1$. Since any region of k-level mesh is partitioned by disjoint regions of $(k-1)$-level meshes, any two maximal valid meshes are disjoint. The definition of a query network is as follows.

Definition 4. *For given vertices s and t,* a query network *of s and t in G is the union of maximal valid meshes.*

From Lemma 3, the shortest path length is the same in G and the query network. Thus, we have the following theorem.

Theorem 1. *For any embedded network G and constant number c, the shortest path distance from s to t is invariant in G and the query network of s and t in G.*

The highway hierarchy method uses a two-direction search starting from both s and t [6]. Consequently, this method does not admit time-dependent costs. Moreover, its sparsification strongly depends on Dijkstra's algorithm. Thus, this method can not use negative costs either. In contrast, our sparsification method is orthogonal to any shortest path algorithm. After constructing a query network, we can execute any shortest path algorithm, such as Dijkstra's algorithm, A* search, etc. Thus, we can use time-dependent and negative costs.

The other advantage of our method is small additional memory. The existing methods need a lot of memory; the highway hierarchy needs an additional edge on each vertex to connect two layers, bit vector needs additional bits on each edge, and transit node routing needs complete shortest path distances between all pairs of transit nodes. The sparsified networks/meshes are defined on the same vertex set; thus they need no extra information for connecting networks/meshes. What we have to do is simply to take the union of the networks/meshes. As we show later, the total space needed to store the sparsified meshes is quite small; less than 1/10th of the original network. This is a great advantage because it enables advanced algorithms to be used on small devices, such as personal navigation systems.

3.3 Construction of Levelwise Sparsified Mesh

LMS requires preprocessing to construct a levelwise sparsified mesh. A straightforward approach is to solve the all pairs shortest path problem on the boundary of each outer region. Since this involves heavy computation, we use a method to reduce the computational time.

The construction is done in a bottom-up way. We first compute all the 1-level meshes. Although this is done in a straightforward way, it does not take long since the outer regions are quite small. Next, we compute the 2-level meshes by finding shortest paths connecting all pairs of vertices on the boundary of each outer region. Note that for any pair of vertices on the boundary, the sparsified meshes of some internal areas of the outer region are valid. Thus, in this process, we can replace induced subgraphs of G with 1-level meshes. Similarly, to compute k-level meshes, we replace induced subgraphs of G with maximal meshes that are valid for any pair of boundary vertices. Figure 3 shows a network used for finding $G(c, i, j)$. Each square corresponds to a maximal mesh. This recursive use of sparsified meshes reduces the computational time. The construction of the levelwise sparsified mesh admits simple parallel computation, since $G(c, i, j)$ and $G(c, i', j')$ can be computed independently.

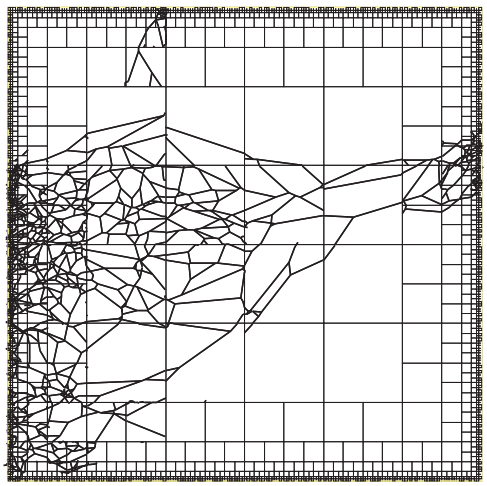

Fig. 3. An example of a network for finding a sparsified network

4 General Framework

The idea of LMS has many possibilities for extensions and generalizations. This section discusses some of these possibilities.

We defined the outer region $R_{\text{outer}}(c,i,j)$ around $R(c,i,j)$ as $R_{\text{outer}}(c,i,j) = \{(x,y) \in \mathbb{R}^2 \mid (i-1)c \leq x < (i+2)c,\ (j-1)c \leq y < (j+2)c\}$. In general, we could define $R_{\text{outer}}(r,c,i,j)$ as $R_{\text{outer}}(r,c,i,j) = \{(x,y) \in \mathbb{R}^2 \mid (i-r)c \leq x < (i+r+1)c,\ (j-r)c \leq y < (j+r+1)c\}$. As the radius r increases, the sparsified network/mesh becomes sparser. However, the cost of construction increases, and we can not use sparse networks near s and t. Thus, there would be an optimal choice on r. This involves another optimization problem.

Another extension is the use of variable shapes for regions and outer regions. We can use non-square areas for the partition, such as non-square rectangles, hexagons and circles. Another idea is using four outer regions shifted in vertical and horizontal directions for each region. This enables us to use much larger sparsified networks in areas close to s or t. When t is north of s, we could use outer regions shifted in the north. Accordingly, the southern parts of the outer regions will be small; thus, we could use sparsified networks of much larger outer regions even if they are close to s. In contrast, we could use outer regions shifted to the south near t. In some sense, LMS considers only stable geometrical partitions, but this introduces "directions" to the geometrical partition.

Direct application of LMS is infeasible when the given network is not embedded on a 2-dimensional plane. In this case, we can use a generalization of LMS. Essentially, our sparsification method depends on the definitions of the (inner) region and outer region. The key to the sparseness is that any vertex outside of the outer region is far from the vertices in the (inner) region. Thus, if we can define (inner) regions and outer regions by using vertex sets satisfying such a property, we can obtain the sparsified networks in the same way. We can partition the vertex set into many groups so that each group consists of many edges and try to minimize the number of edges between two groups. We can define region R as the group of vertices and the outer region of R as vertices near some vertices in R. Another way is to recursively bi-partition the vertex set by a minimal cut to get the regions, and define the outer region as a larger region including the (inner) region. In such a way, we can partition non-geometric networks and use an LMS-like method on them.

5 Computational Experiments

We evaluated our method on the road network of the United States. The network of the US was obtained from the TIGER/Line Files [8]. Since the network is undirected, we transformed it into a bi-directed network. The numbers of vertices and edges of the network are 23,947,347 and 58,333,344, respectively. The network data contains the longitude and the latitude of each vertex, so we used longitude and latitude as the x-coordinate and y-coordinate in LMS. The network data were composed of edges and vertices. An edge has its ID of end vertices and its distance; a vertex has its longitude and latitude. We use four bytes to store each of these values. Hence, the size of the network data was about 900M bytes; about 37 bytes/vertex. The length of each edge represents the travel time between end vertices in the network data.

We implemented our algorithms in C and ran all our experiments on a Mac Pro with 3GHz Intel Xeon CUP and 2GByte RAM, running Mac OS 10.4.10. In the preprocessing and shortest path queries, we employed Dijkstra's algorithm of a simple binary heap implementation. Table 2 shows the preprocessing costs and query speeds of LMS. LMS was over a thousand times faster than Dijkstra's algorithm.

Preprocessing CPU time is the amount of the computational time for constructing levelwise sparsified meshes. Preprocessing overhead accounts for the additional memory that is needed by LMS: the ID of endpoints and the length of each edge in the sparsified meshes. The table shows the size of the additional memory divided by the number of vertices. We employed the number of settled vertices in Dijkstra's algorithm as the measure of the speed of the shortest path search (response time). This measure has been used in previous studies as a fairly natural measure of query difficulty, since it does not depend on computational environments. The table lists the average/worst number of settled vertices in LMS over random 1,000 queries. Query speedup is the average of the number of settled vertices on the original network divided by the number of settled vertices in LMS over 1,000 random queries. The scale of 1-level mesh is an important parameter in LMS. The preprocessing time, overhead and query speed depend on this scale. We tested several scale sizes as shown in the table. These scales are practical in the sense of preprocessing time. When

Table 1. Preprocessing costs and query speed of LMS in US

The scale of 1-level meshes [degree]		1/2	1/4	1/8	1/16
The highest level of meshes		5	6	7	8
Preproc. CPU time [minute]		1228	805	640	1036
	overhead/vertex [byte]	0.07	0.25	0.87	2.70
Query	#settled vertices (ave.)	188,701	62,910	22,704	9,638
	#settled vertices (worst)	585,463	289,419	104,744	33,655
	speed up (ave.)	81	274	762	1,492

the scale of 1-level mesh is 1/8 degree, the highest level of sparsified meshes is 7. This means that the outer regions of 8-level meshes are sufficiently large so that there is no pair of vertices whose shortest path crosses the meshes.

Compareed with the highway hieracy [6], LMS settles about 10 times more vertices, while it uses about 1/20th the overhead in the instance. We implemented only sparsification; the use of a sophisticated search method in conjunction with it would reduce the number of settled vertices even further.

From Table 2, we can see the tendencies of the trade-off between overhead and query speed. The overhead is large and the query speed is fast when the scale is small. Ignoring preprocessing time, we can select a suitable scale for the situation. Compared with the highway hierarchy, LMS needs far less additional memory. In the case of the US road network, the highway hierarchy needs at least 25 bytes/vertex additional memory for any parameter setting [6].

To understand the trade-offs between preprocessing time, overhead, and query speed precisely, we tested our method on a smaller network, the road network of Colorado, USA. The vertices and edges of this network amounted to 448,253 and 1,078,590, respectively. The results in Table 2 seem to indicate an optimal scale of 1-level mesh in

Table 2. Preprocessing costs and query speed of LMS in Colorado

The scale of 1-level meshes [degree]		1/2	1/4	1/8	1/16	1/32	1/64	1/128
The highest level of meshes		2	3	4	5	6	7	8
Preproc. CPU time [second]		287	252	197	175	222	600	1,418
	overhead/vertex [byte]	0.24	0.82	2.13	4.75	9.52	17.56	29.74
Query	#settled vertices (ave.)	83,164	38,124	17,308	7,843	4,538	3,530	3,251
	#settled vertices (worst)	215,724	124,999	71,832	31,103	12,833	9,698	9,174
	speeding up ratio (ave.)	3.5	10.1	23.8	41.3	56.0	65.0	68.9

regard to preprocessing time. As the scale becomes smaller, the number of meshes to be found becomes larger, but these meshes are valid (and hence useful) for finding meshes of a higher level. Moreover, the overhead becomes larger and the settled vertices become fewer. But there seems to be a lower bound to the number of settled vertices.

Our method LMS finds "important" edges that are frequently used in shortest paths connecting distant vertices. When the length of each edge is the geometric length instead of travel times, the importance of edges should be less distinct since the difference between fast and slow road fades. The geometric length of each edge is also presented in the US road network data. We also tested LMS on this length. When the scale of the 1-level mesh is 1/8, the number of meshes is 7, the preprocessing time is about 670 minutes, the overhead is 2.94 bytes per vertex, the average number of settled vertices is 39,300, the worst number of settled vertices is 125,093, and the average speed up ratio is about 335 over 1,000 random queries. Although the preprocessing costs increased and query speed decreased in this case, the results show that our sparsification method is still useful.

6 Concluding Remarks

In this paper, we addressed the shortest path query problem. We considered a partition of a network based on geometrical information and proposed a new method to sparsify the network in each region of the partition. The edges

in a sparsified network are characterized by the edges included in the shortest path of some vertices far from the region. Using regions of several sizes, the shortest path can be obtained by combining a reasonably small number of sparsified networks. Unlike other methods, our sparsification method admits any shortest path algorithm; thus, it handle both negative and time-dependent costs. Moreover, when the network changes, we only have to replace the regions with respect to the edges. This is an advantage in the real-world systems such as car navigation systems. Since our method requires far less additional memory compared with other methods, it is advantageous for mobile devices with limited fast memory.

The computational experiments used a simple implementation of Dijkstra's algorithm. Using more sophisticated algorithms would give further performance improvements; hence, thir implementation would be an interesting future work. So would developing another algorithm for the preprocessing phase. The computational experiments did not examine any of the generalizations and extensions mentioned in this paper. Hence we could conduct more computational experiments with real-world constraints incorporating such generalizations and extensions.

Acknowledgment

This research was supported in part by Funai Electric Co., Ltd.

References

1. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1** (1959) 269–271
2. Thorup, M.: Undirected single-source shortest paths with positive integer weights in linear time. Journal of the Association for Computing Machinery **46**(3) (1999) 362–394
3. Goldberg, A.V., Kaplan, H., Werneck, R.: Reach for a^*: efficient point-to-point shortest path algorithms. In: Proceedings of the Workshop on Algorithm Engineering and Experiments (ALNEX). (2006) 129–143
4. Köhler, E., Möhring, R.H., Schilling, H.: Acceleration of shortest path and constrained shortest path computation. In: Experimental and Efficient Algorithms. Volume 3503 of Lecture Notes in Computer Science., Springer (2005) 126–138
5. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: Proceedings of the 13th European Symposium on Algorithms. Volume 3669 of Lecture Notes in Computer Science., Springer (2005) 568–579
6. Sanders, P., Schultes, D.: Engineering Highway hierarchies. In: Proceedings of the 14th European Symposium on Algorithms. Volume 4168 of Lecture Notes in Computer Science., Springer (2006) 804–816
7. Bast, H., Funke, S., Matijevic, D., Sanders, P., Schultes, D.: In transit to constant shortest-path queries in road networks. In: Proceedings of the Workshop on Algorithm Engineering and Experiments (ALNEX). (2007) 46–59
8. U.S. Census Bureau, Washington, DC. UA Census 2000 TIGER/Line Files. `http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html`