Levelwise Mesh Sparsification for Shortest Path Queries

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

[1] National Institute of Informatics, Hitotsubashi 2-1-2, Chiyoda-ku, Tokyo 101-8430, Tokyo, Japan,
uno@nii.jp,
[2] Sophia University, Kioicho 7-1, Chiyoda-ku, Tokyo 102-8554, Tokyo, Japan,
y-miyamo@sophia.ac.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. Shortest path query is a problem to answer the shortest path from the origin to the destination in short time. The network is fixed, and the goal of the study is to develop efficient data structures to speed up the search algorithms which can be constructed in not so long time. In this paper, we propose levelwise mesh sparsification method for constructing a sparse network. The shortest path can be solved in the sparse network, thus the computation time for each query is reduced. We consider regions of several sizes, and construct the sparsified network for each region composed of edges which are parts of shortest paths of vertices far from the region. For each query, the sparse network is constructed by combining the sparsified networks for which the origin and the destination are distant. We show that the sparsified networks are actually very sparse compared to the original network by some computational experiments on real road networks.

1 Introduction

The shortest path problem is a most fundamental problem in optimization and graph algorithm. The shortest path problem has so many applications in both theory and practice. For example, dynamic programming is basically considered to solving the shortest path on the table, and the computation of the edit distance of two strings is reduced to the shortest path problem. In real world, car navigation systems utilize shortest path algorithms, and Internet packet routing needs a kind of shortest path to the destination. There are more and more applications, thus we can not list all them here. The shortest path problem can be solved by Dijkstra's algorithm [1] in $O(m + n \log n)$ time where n and m are the number of vertices and edges, respectively. Recently, quasi linear time sophisticated algorithms have been proposed [2].

The shortest path can be found in almost linear time by simple algorithms such as Dijkstra's algorithm. This is a big advantage for its applications. However, some recent applications have had some interesting requests to solve the problem in shorter time; linear time is too long. For example, a car navigation system solves the shortest path problem in a network having a huge number of edges, such as 50 million in US road network. Perhaps Dijkstra's algorithm often needs few minutes but we can not wait a long time before beginning driving. In on-line navigation services, the server system has to answer the shortest path in quite short time, say 0.01 second. In these applications, the network does not

change frequently, but we have to answer to so many problems with different pairs of a source vertex and a target vertex. In some sense, this is not an optimization problem but a database query problem, thus we here call the problem *shortest path query*.

In the recent researches on the shortest path query, preprocessing on the network to construct a kind of data structures have been considered to help the speeding up. Dijkstra's algorithm starts from the source vertex and searches the vertices in the increasing order of the distance from the source vertex. Thus, if the target vertex is closed to the source vertex, the algorithm terminates in short time. The central task is how to deal with distant source-target vertices.

In real world network data, we may observe that few edges are frequently used in the shortest paths between distant vertices. Such few edges are highways or arteries in road networks, and backbones in Web networks. So called layer method is based on this observation. It constructs a layer network composed only of highways and arteries as a preprocessing. When it solve the shortest path problem, it finds the nearest vertex to the source vertex, and that to the target vertex, then connects the vertices by the shortest path in the layer network. This method is perhaps the most popular method in the modern car navigation systems. However, the edges in the layer network are not chosen by mathematical way but according to their attributes, thereby we can not always obtain the optimal solution. Thus, one of the recent main topic is how to have data structures for example such abstract networks without losing the optimality. In this topic there are several studies [3–6].

One of them is a bit vector method [4]. Consider a partition of the network into regions $R_1, \ldots, R_q$. Then, for each vertex v and region $R_i, v \notin R_i$, find all the edges incident to v and included in the shortest path from v to a vertex in R_i. We memorize the edges as a data structure. When we search the shortest path from v to a vertex u in R_i, we can restrict the search to the edges. If v is far from R_i such edges are few, thus until the current visiting vertex is closed to the destination, the edges searched forms like a path. Therefore we can save the computation time much.

Bit vector can be considered that it finds a structure common to shortest paths from a vertex to the vertices in a region. On the other hand, when the source vertex is closed to the region including the target vertex, the edges to be visited become many, thus we need much computation time. By increasing the number of the regions q, we can increase the efficiency instead of the increase of memory usage. Moreover, the preprocessing essentially needs to solve the all pairs shortest paths problem, thereby takes long time.

Bit vector method finds the common first edges of the shortest paths. In contrast to that, highway hierarchy method [5] finds the edges common to the middle of the shortest paths. Suppose that an edge e is included in the shortest path from v to u, and an endpoint of e is k_vth closest vertex to v, and the other endpoint is k_uth closest vertex to u. If both k_v and k_u are smaller than a threshold value h, we call e highway. Highway hierarchy method constructs the highway network composed of highway edges. When we execute Dijkstra's algorithm on the original network, we go to the highway network after h steps. Highway network is usually sparse compared to the original network, thus the computation time is shorten. Constructing the highway network on the highway network recursively, the computation time become shorter. In contrast to the layer method, it does not lose the optimality. Short preprocessing time is also an advantage of this method.

The third method is transit node routing [6]. 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 memorize them as a data structure. When we execute the Dijkstra's algorithm, we can directly move from the transit vertices to the transit vertices near by the destination, thus we can save the computation time.

In this paper, we propose a new method LMS (*Levelwise Mesh Sparsification*) to the problem. The method is to construct a sparsified network based on the geometric partition of the network. LMS considers a partition of the network into the regions $R_1, ..., R_q$. For each region R_i, we define the outer region Q_i by the region including R_i, and construct the sparsified network of R_i composed of edges inside R_i and included in a shortest path connecting two vertices outside Q_i. For any two vertices outside Q_i, the edges of the shortest path placing in R_i are of the sparsified network of R_i. Thus, when we find the shortest path from v to u, we can restrict the search to sparsified network in regions whose outer regions include neither v nor u. Moreover, by considering regions with different sizes, for example the rectangles whose edge is of length $c \cdot 2^k$ for some c, we can use larger region with more sparse network for distant vertices.

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

To construct the semi sparsified network, we need to solve the all pairs shortest path problem on the vertices on the boundary of the outer region. This takes longer time than highway hierarchy method, but using the sparsified network inside the outer region recursively, we can reduce the computation time.

This organization of this paper is as follows. Section 2 is for the definitions and the notations. In Section 3, we explain the (semi-)sparsified network and state some lemmas to assure the optimality. In Section 4, we explain the method to construct the (semi-)sparsified network based on the geometric implementation. The result of the computational experiments is shown in Section 5, and we 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 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. In general the 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 s-t shortest path problem is to determine one

of the s-t shortest paths. In some context, the s-t shortest path problem requires only the length of a s-t shortest 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 Sparsified Network

Let V_{inner} and V_{outer} be subsets of V satisfying $V_{\text{inner}} \subseteq V_{\text{outer}} \subseteq V$. Here we define the *semi-sparsified network* of V_{inner} and V_{outer}.

Definition 1. *A semi-sparsified network of G derived by V_{inner} and V_{outer} is the subnetwork of $G[V_{inner} \cup N(V_{inner})]$ whose each edge is on the shortest path of G connecting two vertices not in $V \setminus V_{outer}$.*

We denote a semi-sparsified network of G derived by V_{inner} and V_{outer} by $S'_G(V_{\text{inner}}, V_{\text{outer}})$. We call the subset of vertices V_{inner} and V_{inner} *inner vertices* and *outer vertices* of $S'_G(V_{\text{inner}}, V_{\text{outer}})$, respectively.

Lemma 1. *All shortest paths whose both end vertices are in $V \setminus V_{outer}$ of G are included in $G[V \setminus V_{inner}] \cup S'_G(V_{inner}, V_{outer})$.*

We note that a semi-sparsified network $S'_G(V_{\text{inner}}, V_{\text{outer}})$ is obtained by specifying all shortest paths whose end vertices are in $N(V_{\text{outer}})$.

Next we define a *sparsified network* that is an edge contracted network of semi-sparsified network. An *edge contraction* of a network G is following procedures;

- if there exists $v \in V(G)$ whose incident edges are $(u, v), (v, w)$, then remove $(u, v), (v, w)$ and add (u, w) whose distance is $d(u, v) + d(v, w)$,
- if there exists $v \in V(G)$ whose incident edges are $(u, v), (v, u), (v, w), (w, v)$, then remove $(u, v), (v, u), (v, w), (w, v)$ and add $(u, w), (w, u)$ whose distances are $d(u, v) + d(v, w), d(w, v) + d(v, u)$ respectively.

We define a *bridge* of a semi-sparsified network $S'_G(V_{\text{inner}}, V_{\text{outer}})$.

Definition 2. *A sparsified network of G derived by V_{inner} and V_{outer} is a minimal network obtained by edge contraction except for bridges of $S'_G(V_{inner}, V_{outer})$ recursively.*

We denote a sparsified network of G derived by V_{inner} and V_{outer} by $S_G(V_{\text{inner}}, V_{\text{outer}})$. As same as the semi-sparsified networks, V_{inner} and V_{outer} are inner vertices and outer vertices of $S_G(V_{\text{inner}}, V_{\text{outer}})$. The statement of Lemma 1 also holds for the sparsified network in the sense of path lengths. A bridge of (semi-) sparsified network is an edge (u, v) that $[u \in V_{\text{inner}},\ v \notin V_{\text{inner}}]$ or $[u \notin V_{\text{inner}},\ v \in V_{\text{inner}}]$.

In real world road networks, roads (edges) on highways are frequently used in the shortest paths between distant vertices, since the distance of such roads are short in some sense. In these case, the number of edges of the sparsified network would be much less than that of an original network, and the query time using the sparsified networks would be reduced significantly.

For a given constant integer I, let U_i, V_i, $i = 1, \ldots, I$ be subsets of V satisfying

$$U_i \subseteq V_i \subseteq V, \forall i = 1, \ldots, I,$$
$$U_i \cap U_j = \emptyset, \, i \neq j.$$

Lemma 2. *When $s, t \in V \setminus \{\bigcup_i V_i\}$, an s-t path on G is a shortest path if and only if the path is a shortest path on $G[V \setminus \{\bigcup_i U_i\}] \cup \{\bigcup_i S'_G(U_i, V_i)\}$, and the statement also holds for the sparsified network in the sense of path length.*

From Lemma 2, sparsified networks derived by a set partition of the original network is also useful for shortest path queries.

In the following section, we propose a simple implementation of constructing sparsified networks derived by a geometry-based partition of the original network.

4 Geometric Implementation and Levelwise Networks

In the following, we assume that all vertices are on a 2-dimensional plain and assume that the 2-dimensional coordinates of all vertices are given. This assumption is reasonable in the context of road networks or railroad networks. Let $x(v), y(v)$ be the x-coordinate and the y-coordinate of v, respectively. We call a rectangle region on the 2-dimensional plain *cell*. A cell is specified by its width, height and the smallest point. Let $C_H(i, j)$ be a cell defined by $C_H(i, j) = \{(x, y) \in \mathbb{R}^2 \mid iH \leq x < (i + 1)H, \; jH \leq y < (j + 1)H\}$, where H is a real constant. Clearly, the set of $C_H(i, j)$, $\forall i, j \in \mathbb{Z}$ is a partition of 2-dimensional plain.

4.1 Grid Based Sparsified Networks

Without loss of generality, we assume that $0 \leq x(v), y(v)$ and then $\exists L \in \mathbb{R}$, $x(v), y(v) < L$, $\forall v \in V$. We here suppose that $H = l$ for some constant $l \in \mathbb{R}^+$, Let $V_H(i, j)$ be a subset of V defined by $V_H(i, j) = \{v \in V \mid (x(v), y(v)) \in C_H(i, j)\}$. Let $G_H(i, j)$ be the subnetwork of G induced by $V_H(i, j) \cup N(V_H(i, j))$. We define the grid based sparsified network $S_{G,H}(i, j)$ whose inner vertex set is $V_H(i, j)$ and whose outer vertex set is the union of $V_H(i', j')$ where $i' \in \{i - 1, i, i + 1\}$, $j' \in \{j - 1, j, j + 1\}$. In precise,

$$S_{G,H}(i, j) = S_G(V_H(i, j), V_H(i - 1, j - 1) \cup \cdots \cup V_H(i + 1, j + 1)).$$

Here we note that the set of vertices of $G_H(i, j)$ is the same as of $S_{G,H}(i, j)$. Let $G_H\{v\}$ be a union of networks $G_H(i, j)$ around $v \in V$ defined by $G_H\{v\} = \{G_H(i-1, j-1) \cup \cdots \cup G_H(i+1, j+1) \mid v \in V_H(i, j)\}$. Let $G_H\{s, t\}$ be a union of networks $G_H(i, j)$ around $s, t \in V$ defined by $G_H\{s, t\} = G_H\{s\} \cup G_H\{t\}$. We call a sparsified network $S_{G,H}(i, j)$ is *valid* for a pair of vertices $\{s, t\}$, if both s and t are out of outer vertices of $S_{G,H}(i, j)$. It means that the valid sparsified networks for $\{s, t\}$ are sufficiently far from s and t, and can be used in the search of the shortest s-t path. Let $S_{G,H}\{s, t\}$ be a union of sparsified networks $S_{G,H}(i, j)$ valid for {s,t}. From Lemma 2 the following lemma is immediate.

Lemma 3. *Let $s, t \in V$. The length of the shortest s-t path on G is the same as the length of the shortest s-t path on a network $G_H\{s, t\} \cup S_{G,H}\{s, t\}$. In particular, the shortest s-t path on G is obtained from the shortest s-t path on $G_H\{s, t\} \cup S_{G,H}\{s, t\}$ with edge decontraction.*

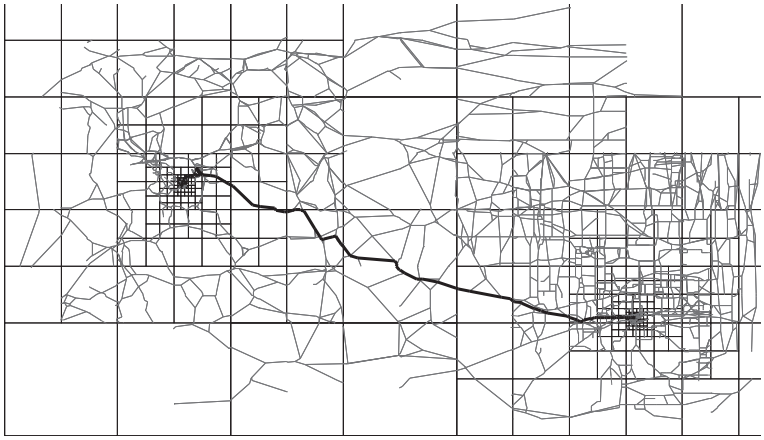

Fig. 1. An image of Theorem 1

If the size of cells H is smaller, the number of cells (that is the number of sparsified networks) are larger, and the number of edges of sparsified networks might be larger, and the region covered by sparsified networks are larger. Thus, there is a trade-off between the speed of shortest path search and the size of cells.

When a query (a source vertex and target vertex) is given, we want to use small cells near the vertices, and we want to use large cells far from the vertices, to use the sparsified network as much as possible. Thus, in the next section, we consider the combination of cells of various sizes. For the conciseness, we set the sizes of cells to $1, 2, 2^2, 2^3, \ldots$. Other sizes can be also considered, in future works.

4.2 Levelwise Sparsified Networks

Here we propose *levelwise sparsified network* that is more efficient than the sparsified networks based on fixed size cells. We denote $S_{G,2^k}(i,j)$ by $S_G^k(i,j)$, where k is nonnegative integer. We call sparsified networks $S_G^k(i,j)$ *level k sparsified networks*. We note that the bridges of level k sparsified networks are included in the bridges of level k' sparsified networks, if $k' < k$. Let K be the minimum integer satisfying $2^K > L$. Clearly, it is sufficient for shortest path queries to prepare $S_G^H(i,j)$ for $H = 1, ..., K$. In our construction, there are K level sparsified networks (from level 0 to level $K-1$) and the level K sparsified network is empty. For a pair of vertices $\{s,t\}$, a sparsified network $S_G^k(i,j)$ is *maximal valid*, if the network is valid for s and t, and no valid sparsified network $S_G^{k+1}(i',j')$ of one higher level satisfy $V_k(i,j) \subseteq V_{k+1}(\lfloor i/2 \rfloor, \lfloor j/2 \rfloor)$, that is, $i = \lfloor i/2 \rfloor$ and $j' = \lfloor j/2 \rfloor$. For $k = 0, 1, \ldots, K-1$, let $S_G^k\{s,t\}$ be a union of really valid sparsified networks $S_G^k(i,j)$ for a pair of vertices $\{s,t\}$.

Theorem 1. *Let $s, t \in V$. The length of the shortest s-t path on G is the same as the length of the shortest s-t path on a network $G_1\{s,t\} \cup S_G^0\{s,t\} \cup S_G^1\{s,t\} \cup \cdots \cup S_G^{K-1}\{s,t\}$.In particular, the shortest s-t path on G is obtained from the shortest s-t path on $G_1\{s,t\} \cup S_G^0\{s,t\} \cup S_G^1\{s,t\} \cup \cdots \cup S_G^{K-1}\{s,t\}$ with edge decontraction.*

Figure 1 shows an image of Theorem 1. In Figure 1, the broad line is the shortest path, and cells are corresponding to sparsified networks of various levels.

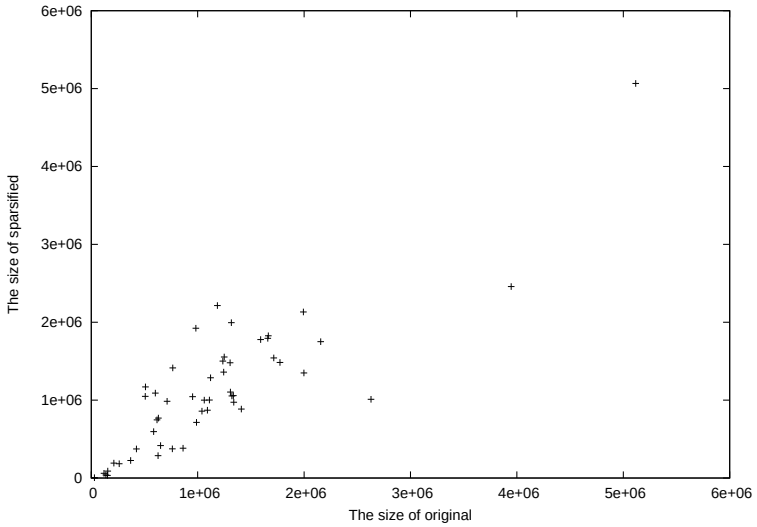

Fig. 2. The size of the levelwise sparsified networks

We also mention to the preprocessing time to construct the levelwise sparsified networks. As the level of the sparsified network grows, the size of the corresponding induced subnetwork of the original grows exponentially. A simple way to reduce the total preprocessing time is to use lower level sparsified networks in construction of a sparsified network. To implement this idea, the levelwise sparsified networks should be constructed in the bottom up manner. We use this technique in our experiments presented in the following section.

5 Computational Experiments

We tested our method on the road network of the United States. The network of the US has been obtained from the TIGER/Line Files [7]. We deal with networks of the 50 states and the District of Columbia. Since these networks are undirected, we interpreted the networks to bidirected networks. The number of vertices and edges of the smallest state are 9559 and 29818, respectively. Those of the largest are 2073870 and 5168318, respectively. In the instances, the longitude and the latitude of each vertex are also given.

We implemented our algorithms in Java 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 the response for the shortest path queries, we employed the Dijkstra's algorithm with binary heap implementation.

We fixed the minimum cell size H about 1/100 degree (that is nearly equal to 2km). When the size H is less than 1/100 degree, the level 0 sparsified networks are very close to original networks. Actually, a cell has many vertices up to several hundreds in downtown, but very few, say less than 10, in surburb. The maximum level of the sparsified network of the largest state is 8.

The size (the sum of the number of edges) of the levelwise sparsified networks are shown in Figure 2. The horizontal axis corresponds to the number of vertices of original networks. The vertical axis corresponds to the number of edges of levelwise networks.

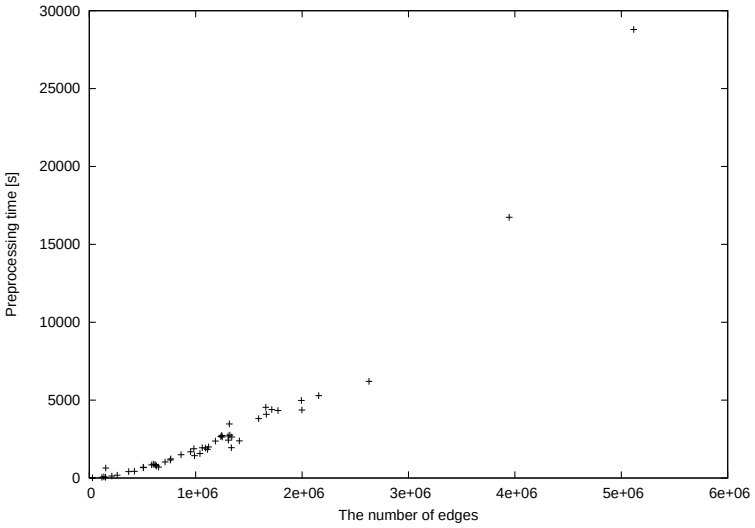

Fig. 3. The preprocessing time

From Figure 2, we can see that the whole size of the levelwise sparsified network is slightly larger than that of original network.

The preprocessing times are shown in Figure 3. The horizontal axis corresponds to the number of edges of original networks. The vertical axis corresponds to the computational time to make the levelwise sparsified networks.

We introduce the *Dijkstra rank* that is defined by the number of vertices Dijkstra's algorithm would have to settle for that query. The Dijkstra rank is shown in many previous works for a fairly natural measure of the difficulty of a query. The average over 1000 random queries with Dijkstra rank of the levelwise sparsified networks for all states are shown in Figure 4. The horizontal axis corresponds to the number of vertices of original networks. The vertical axis corresponds to the average Dijkstra rank.

Here the average Dijkstra rank of the levelwise sparsified network for the largest state is about 9412.993, though that of the original network is 1024810.323. In the case of the largest state, the average response time over 1000 queries using the levelwise sparsified network is 36.033ms. From Figure 4, the growth of the Dijkstra rank on levelwise sparsified networks is less than the growth of the size of the original networks.

6 Concluding Remarks

In this paper, we address the shortest path query problem. We considered a partition of the network based on the geometrical information and propose a new method to sparsify the network in each region of the partition. The edges in the sparsified network 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 not so many sparsified networks. The geometrical partition makes the boundary of the regions sparse, and may help dealing with additional constraints and updates of the network.

The implementation of our algorithm is currently straightforward. The preprocessing phase is one of the bottle neck of our algorithm. Some sophisticated

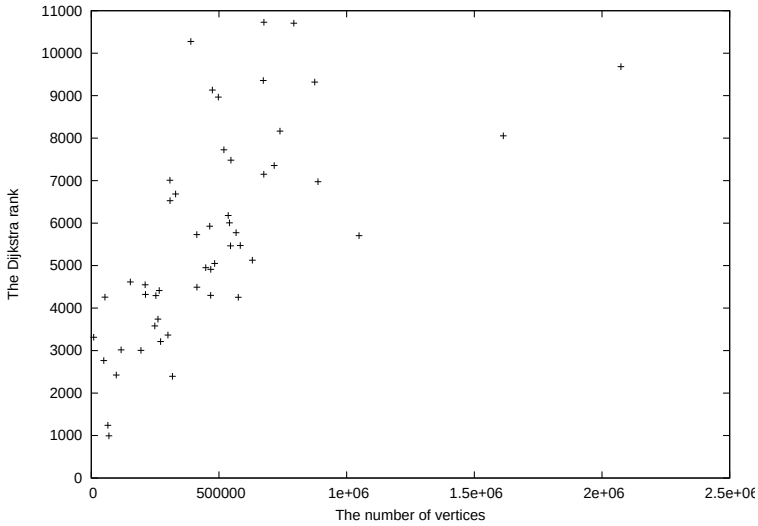

Fig. 4. Dijkstra rank of levelwise sparsified networks

shortest path algorithms such as A* algorithms can improve the performance. We can use non-square areas for the partition, such as circles. The ratio of the areas of the outer region and the inner region can also be optimized. Sparsification around the source vertex and target vertex, the consideration of directions, and skewed or shifted outer region can also be considered.

Acknowledgement

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. 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
7. U.S. Census Bureau, Washington, DC. UA Census 2000 TIGER/Line Files. http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html