An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps

Sungwon Jung, *Member*, *IEEE Computer Society*, and Sakti Pramanik

Abstract—In this paper, we have developed a $HiTi$ (Hierarchical MulTi) graph model for structuring large topographical road maps to speed up the minimum cost route computation. The $HiTi$ graph model provides a novel approach to abstracting and structuring a topographical road map in a hierarchical fashion. We propose a new shortest path algorithm named $SPAH$, which utilizes $HiTi$ graph model of a topographical road map for its computation. We give the proof for the optimality of $SPAH$. Our performance analysis of $SPAH$ on grid graphs showed that it significantly reduces the search space over existing methods. We also present an in-depth experimental analysis of HiTi graph method by comparing it with other similar works on grid graphs. Within the $HiTi$ graph framework, we also propose a parallel shortest path algorithm named $ISPAH$. Experimental results show that *inter* query shortest path problem provides more opportunity for scalable parallelism than the *intra* query shortest path problem.

Index Terms—Shortest Path, digital road maps, grid graphs, parallel shortest path computation, HiTi graph model.

1 Introduction

1.1 Motivation

In navigation systems, a primary function is to find possible routes from the current location (e.g., where a driver is currently positioned) to the destination (e.g., where the driver wants to go) with a minimum expected cost. For this purpose, they use a topographical road map which is in the form of the following recursive relation:

$$\mathbf{topographical\_road\_map}(\mathbf{source}, \mathbf{destination}, \mathbf{cost}),$$

where the cost attribute indicates, for example, a minimum expected time of travel from point *source* to point *destination*. Another applicable cost can be the shortest distance between the two end points.

One of the major difficulties of navigation systems is the size of the topographical road map data. It requires about 2.4 Gbytes of storage to store a small 100 $mi \times 100$ mi map discretized at 100 feet intervals [4], [24]. Thus, the size of data involved is very large when larger maps are considered. Minimum cost route computation with this large amount of road map data requires a significant amount of computation time. Since navigation systems are real-time systems, it is critical to compute a minimum cost route satisfying a time constraint.

We have developed a $HiTi$ graph model of very large recursive relations (e.g., topographical road maps), for efficiently computing the optimal minimum cost path. The $HiTi$ graph model follows the recommendation of Shekhar et. al [39]. The basic idea of the $HiTi$ graph model is to partition a large graph into smaller subgraphs and pushing up the precomputed shortest paths between the boundary nodes of each subgraphs in a hierarchical manner.

Multiple levels of geographical boundaries (e.g., cities, counties, and states) can be easily mapped into the hierarchical structure of a $HiTi$ graph model. Various levels of hierarchical abstractions can also serve as the basis for efficient storage management for large road maps. Thus, it provides the basis for the development of a more controlled storage management suitable for navigation systems with limited available storage (e.g., navigation system inside an automobile). Based on the *HiTi* graph model, we propose a new single pair minimum cost path algorithm named *SPAH*. We show that the shortest path computed by $SPAH$ is optimal. We also experimentally show that *SPAH* dramatically reduces the explored search space. Further, we analyze *SPAH* by varying edge cost distribution and the number of hierarchical levels of *HiTi* graphs.

However, the performance of *SPAH* in a single processor environment is not good when multiple SPSP (Single Pair Shortest Path) computation requests are given simultaneously. This is defined as *inter* query SPSP problem. *Inter* query SPSP problem deals with parallelizing *multiple* SPSP computations. *Inter* query SPSP problem arises, say, in the domain of automobile navigation systems, where many vehicles send their shortest route computation requests to a central server. Then, the server must be able to handle these multiple SPSP computation requests satisfying a real-time constraint. In this paper, we have also developed and analyzed the performance of a parallel algorithm, named *ISPAH*, for *inter* query SPSP problem which is based on the *HiTi* graph model.

• S. Jung is with the Department of Computer Science, Sogang University, 1, Shinsoo-Dong, Mapo-Gu, Seoul, Korea, 121-742. E-mail: jungsung@ccs.sogang.ac.kr.
• S. Pramanik is with the Department of Computer Science, Michigan State University, East Lansing, MI 48824. E-mail: pramanik@cse.msu.edu.

Manuscript received 15 June 1998; revised 26 Sept. 2000; accepted 02 Apr. 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 106921.

Path View Connection for CROM 1:
{(A,H,3)}
Path View Connection for CROM 2:
{(B,C,1), (B,D,3), (C,D,1)}
Path View Connection for CROM 3:
{(E,G,2), (E,F,3), (F,G,4)}

Cut Connection for CROM 1 and 2:
{(A,B,3)}
Cut Connection for CROM 2 and 3:
{(C,E,4), (D,F,7)}
Cut Connection for CROM 1 and 3:
{(H,G,2)}

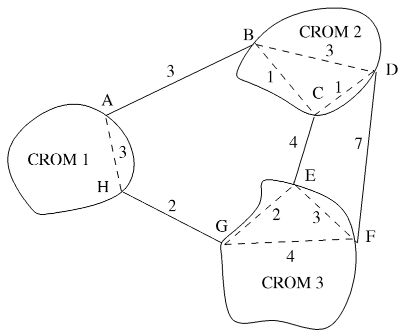

Fig. 1. Examples of *path view* and *cut* connection of CROMs.

1.2 Informal Description of HiTi Graphs

Consider a topographical road map viewed as a directed graph $G(V,E)$. Nodes in V correspond to discretized grid points representing map objects in a road map. Edges $(x, y, cost)$ in E correspond to the connections between the nodes x and y with $cost$ (e.g., distance) in V. Then, arbitrary shaped boundaries (e.g., political regional boundaries) partition a road map $G(V,E)$ into a set of Component ROad Maps (CROM). Each CROM can be viewed as a subgraph SG with its boundary nodes defining the boundary of the CROM. Connectivity between CROMs is represented by the connectivity of their boundary nodes. One CROM has direct connections with another, if boundary nodes of the former are directly connected to boundary nodes of the latter. We call this kind of connectivity *cut* connections of CROMs. For each CROM, we precompute the minimum cost path for each pair of connected boundary nodes of that CROM. These precomputed minimum cost paths are called *path view* connections of that CROM. Note that the *path view* connections do not capture the global minimum cost path in G but the local minimum cost path on that CROM only. By using the *cut* and *path view* connections, we can partition the entire road map. Examples of *cut* and *path view* connections of CROMs are shown in Fig. 1.

A *HiTi* graph is a graph whose nodes are the boundary nodes of the CROMs and edges are the *path view* and *cut* connections of CROMs. Note that a CROM can be defined to contain a set of CROMs, thus creating a multilevel hierarchy. Thus, we first need to determine a set of the lowest level CROMs which exclusively partition an entire road map. We call these CROMs level 1 CROMs. Then, we can recursively construct a set of level k CROMs by grouping a set of geographically adjacent level *k-1* CROMs where $k \geq 2$. These sets of level $1, 2, \ldots, k$ CROMs form a complete balanced tree structure where the root node of the tree is a whole road map. Note that in this tree structure, the number of hierarchical levels is limited by the memory requirement for *path view* connections.

The rest of the paper is organized as follows: Section 2 discusses related works. Section 3 discusses a formal framework and description of the *HiTi* graph. In Section 4, we propose a new SPSP algorithm *SPAH* that takes advantage of the *HiTi* graphs. Performance analysis of *SPAH* on grid graphs is given in Section 5. In Section 6, we present an efficient updating algorithm for the *HiTi* graph, essential for road navigation to reflect the dynamic traffic conditions. We also show the efficiency of the update algorithm for *HiTi* graph in this section. In Section 7, we compare our *HiTi* graph method with other similar works both theoretically and experimentally. Section 8 discusses a new parallel algorithm *ISPAH* for *inter* query SPSP problem. We give performance analysis of this algorithm in this section. Finally, Section 9 gives concluding remarks.

2 Related Works

There have been many research efforts reported in the literature that focus on the shortest path computation problem in database domain. Previously suggested transitive closure or graph traversal algorithms [2], [8], [13], [16], [17], [20], [33], [35], [42] are not directly applicable to **topographical_road_map (source, destination, cost)** for the computation of a minimum cost path due to the very large volume of data they have to search. Thus, we need an efficient database organization method for structuring the topographical road map to speed up the computation of a minimum cost path. In this regard, two different approaches have been studied in the past. One approach is to develop a database structure which gives a suboptimal minimum cost path quickly. The other approach is to develop the database structure which gives an optimal shortest path, primarily based on the precomputed shortest path information.

For the suboptimal shortest path generation, Ishikawa et al., Shapiro et al., Liu et al., and Huang et al. used road hierarchies (i.e., Freeways, Highways,..., Side roads) to speed up minimum cost routes [14], [18], [27], [37], [43]. They used multiple levels of hierarchical details of road maps to cut down the unnecessary search space. Huang et al. [14] proposed a hierarchical graph model which classifies edges according to road types. Ishikawa et al. [18] applied Dijkstra's algorithm to the hierarchically structured road maps. Shapiro et al. [37] proposed a new graph structure, named *LGS (Level Graph Structure),* which models the road hierarchies theoretically. Based on LGS, Shapiro et al. gave a new algorithm which generates approximate shortest paths rapidly. Their study showed that the length of the path produced by LGS converges rapidly to that of the actual minimum cost path as the distance between the source and destination nodes increases. Liu et al. [27] studied integrating Dijkstra's algorithm with a knowledge-based approach and case-based reasoning for computing a minimum cost path.

For the optimal shortest path generation, Agrawal and Jagadish recently studied a data organization technique which precomputes and stores some partial path information [4]. They use the precomputed partial path information to prune the search space when computing a minimum cost path. While their approach speeds up the computation of a minimum cost path, it does not optimize processing minimum cost path queries based on road map navigation. The reason is that their approach still requires searching all the intermediate nodes on a minimum cost path. In navigation systems, it may not be necessary for a navigator to know all the intermediate nodes on a minimum cost path.

The problem would be very serious if we need to compute a minimum cost path on a large topographical road map where its discretized interval is fine-grained and two end nodes on the path are far apart. To cope with this problem, some researches in database systems as well as AI and Pattern Recognition domains have suggested using the hierarchical abstraction of topographical road maps [7], [9], [11], [21], [22], [36], [34], [41]. Jing et al. proposed the *HEPV(Hierarchical Encoded Path View)* approach [21], [22]. Their basic idea is to divide a large graph into smaller subgraphs and organize them in a hierarchical fashion by pushing up border nodes. However, the $HEPV$ approach suffers from the excessive storage overhead for maintaining a large amount of precomputed path information. This is because the $HEPV$ approach precomputes the shortest paths between all the member nodes (including the boundary nodes) of each subgraph only within that subgraph.

Similar to the *HEPV* approach, Goldman et. al. proposed the *Hub Indexing* method [11]. The primary difference between the two methods is the amount of the precomputed paths information for each subgraph. In the Hub Indexing method, instead of precomputing all-pair shortest paths for each subgraph, as in *HEPV*, it precomputes the shortest paths between the boundary nodes of each subgraph and its interior nodes (i.e., nonboundary nodes of each subgraph). We believe that *HEPV* and *Hub Indexing* schemes will have a problem when large road maps have to be considered. In investigating this problem, Shekhar et. al. [39] have analyzed the materialization tradeoffs between the storage overhead with runtime computations of the stored precomputed information. Their analysis has shown the usefulness of the materialization tradeoffs in computing the shortest paths involving the boundary nodes of each subgraph. In Section 7, we compare our *HiTi* graph method with *HEPV* and *Hub Indexing* methods both theoretically and experimentally.

Little research has been done on parallel SPSP algorithms. Mohr and Pasche [31] studied the *intra* query SPSP problem, which deals with parallelizing a single transaction of SPSP computation. They developed a new parallel SPSP algorithm named *OTOpar*. *OTOpar* uses two processors where each processor uses A* algorithm with Manhattan distance estimation to build its corresponding tree, one rooted at the source node and the other rooted at the destination node. Their analysis shows that the performance improvement is limited by only two processors, which is not scalable at all.

3 Formal Framework and Description of the HiTi Graph

A topographical road map can be viewed as a directed graph $G(V,E)$, where each node in V represents map objects (i.e., the intersecting points of the roads) in a road map. Edges (x,y,c) in E correspond to the connections between the nodes x and y with the cost (e.g., distance) c in V. Suppose that $G(V,E)$ is partitioned into a set of subgraphs (i.e., $SG_1(V_1,E_1)$, $SG_2(V_2,E_2)$, ..., $SG_n(V_n,E_n)$) such that:

$$\begin{aligned}&V_1 \cup V_2 \cup \cdots \cup V_n \;=\; V, \quad E_1 \cup E_2 \cup \cdots \cup E_n \;\subset\; E\\ &V_i \cap V_j \;=\; \emptyset \;\; and \;\; E_i \cap E_j \;=\; \emptyset \;\; where\\ &1 \le i,j \le n \;\; and \;\; i \ne j.\end{aligned}$$

Definition 3.1. *For subgraphs $SG_i(V_i,E_i)$, $SG_j(V_j,E_j)$, where $i \neq j$, let $\chi(SG_i,SG_j)$ denote the set of cut edges between V_i and V_j, namely the set of edges in*

$$\{(x,y,c)|x \in V_i, y \in V_j \; or \; vice \; versa\}.$$

Let $\chi(SG_i) = \chi(V_i, V - V_i)$.

Definition 3.2. *Given a collection of subgraphs*

$$SG = \{SG_1, SG_2, \ldots, SG_m\},$$

define $\chi(SG) = \bigcup_{i,j,i\neq j} \chi(SG_i,SG_j)$. For a given subgraph $SG_i \in SG$, let

$$\chi^{SG}(SG_i) = \bigcup_{i\neq j} \chi(SG_i,SG_j).$$

Then, $\chi^{SG}(SG_i)$ is named the cut edge set for SG_i with respect to SG.

Definition 3.3. *Given a collection of subgraphs*

$$SG = \{SG_1, SG_2, \ldots, SG_m\},$$

let $\mu(SG_i)$ denote the set of all the outgoing edges in $\chi^{SG}(SG_i)$. Then, $\mu(SG_i)$ is named the semicut edge set for SG_i.

Definition 3.4. *Given a collection of subgraphs*

$$SG = \{SG_1, SG_2, \ldots, SG_m\},$$

let $\delta(SG_i)$ denote the set of vertices of V_i that have at least one incoming or outgoing edges in $\chi^{SG}(SG_i)$. Then, $\delta(SG_i)$ is called **boundary nodes** *set of SG_i.*

Set δ_i, boundary nodes of SG_i in Definition 3.4, consists of those nodes in V_i that are directly connected to/from the nodes outside of V_i. For example, consider Fig. 2 where a digraph G and its subgraphs SG_1, SG_2, and SG_3 are shown. Set $\delta(SG_2)$ of subgraph SG_2 is { G, H, M, O }. Each subgraph is described and identified by its boundary nodes since they exclusively belong to one subgraph. Based on boundary nodes of SG_i^1, Definition 3.3 gives the formal definition of *path view* edge set ϕ_i of SG_i.

Definition 3.5. *Given a collection of subgraphs*

$$SG = \{SG_1, SG_2, \ldots, SG_m\},$$

let

$$\begin{aligned}\phi(SG_i) = \{(x,y,f_c(x,y))|(x,y) \in (\delta(SG_i)\\ \times \; \delta(SG_i)) \wedge (x \overset{f_c(x,y)}{\longrightarrow} y \; in \; SG_i \wedge x \neq y\}.\end{aligned}$$

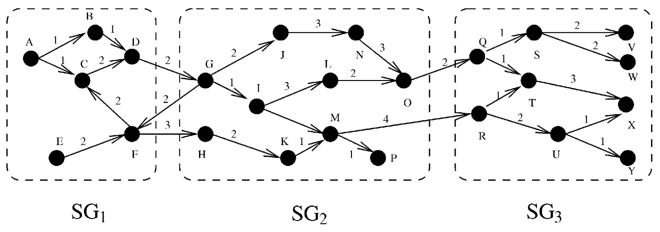

Fig. 2. A digarph G and its subgraph SG_1, SG_2, and SG_3.

Function $f_c(x, y)$ gives the shortest path cost (i.e., from node x to y) computed only within SG_i.

As an example of Definitions 3.3, 3.4, and 3.5, Table 1 shows the corresponding values of $\delta(SG_i)$, $\mu(SG_i)$, and $\phi(SG_i)$ of the three subgraphs shown in Fig. 2.

The above definitions can be generalized into multi-levels. Given a directed graph $G(V, E)$, we construct a subgraph tree T whose nodes are subgraphs of G. T is identified as follows:

- The root node is $G(V, E)$.
- A node v of T representing the subgraph SG of G has children $v_1, v_2, \ldots, v_m$ representing subgraphs $SG_1, SG_2, \ldots, SG_m$, where the subgraphs sets $\{SG_i\}$ form a partition of SG.
- We use the notation SG_i^k to denote the *i*th subgraph on the *k*th level of the tree (where the leaves of the tree are at the level 1).

By observing the structure of the subgraph tree T, we know that all subgraphs in the tree are related to each other in a complete balanced tree structure. The root node of the tree is considered as a level *k+1* subgraph and it has all level *k* subgraphs as child nodes. Recursively, each node symbolizing a level *k* subgraph has a set of level $k-1$ subgraphs as child nodes. All leaf nodes of T symbolize level 1 subgraphs. This subgraph tree T is named level $k+1$ if the root symbolizes level *k+1* subgraph (i.e., $G(V, E)$). The following Fig. 3 shows an example of level 3 subgraph tree. This tree structure shows a level 3 subgraph $SG_1^3 = G$ induced by the nodes in three level 2 subgraphs (i.e., SG_1^2, SG_2^2, and SG_3^2). It also shows three level 2 subgraph SG_1^2, SG_2^2, and SG_3^2 induced by the nodes in the six level 1 subgraphs SG_1^1, SG_2^1, SG_3^1, SG_4^1, SG_5^1, and SG_6^1.

For each level k subgraph SG_i^k of T, there exists corresponding level k *boundary* nodes set (i.e., δ_i^k) , level k *cut* edge set (i.e., χ_i^k), level k *semicut* edge set (i.e., μ_i^k), and level k *path view* edge sets (i.e., ϕ_i^k). They are formally defined in Definitions 3.6 and 3.7.

Definition 3.6. *Let $SG^k = \{S_i^k\}$. Further let $\chi^k = \chi(SG^k)$, $\chi_i^k = \chi^{SG^k}(SG_i^k)$, and let μ_i^k be the set of all the outgoing edges in χ_i^k. Similarly, let δ_i^k denote the set of vertices of V_i^k that have at least one incoming or outgoing edges in χ_i^k.*

Definition 3.7. *Given a collection of subgraphs*

$$SG^k = \{SG_1^k, SG_2^k, \ldots, SG_m^k\},$$

let

$$\phi_i^k = \{(x, y, f_c(x, y)) | (x, y) \in (\delta_i^k \times \delta_i^k) \wedge \ (x \overset{f_c(x,y)}{\longrightarrow} y \ in \ SG_i^k \wedge x \neq y\}.$$

Function $f_c(x, y)$ gives the shortest path cost (i.e., from node x to y) computed only within SG_i^k.

Based on *semicut* edge and the *path view* edge sets defined above, the formal definition of *HiTi* graph is given as below:

Definition 3.8. *A HiTi graph is a directed labeled graph defined in terms of semicut and path view edge sets associated with all nodes of a subgraph tree T. A level $k+1$ subgraph tree T defines a level k HiTi graph. It is represented by $H^k(V^k, E^k)$, where $V^k = \bigcup_{i=1}^k \bigcup_{j=1} \delta_j^i$ and $E^k = \bigcup_{i=1}^k \bigcup_{j=1} \{\mu_j^i \cup \phi_j^i\} \times \{i\}$.*

We have introduced basic concepts and formal definitions of a level *k HiTi* graph in this section. It is easy to see that the overhead of a level *k HiTi* graph comes from the size of *path view* edge sets, $|\bigcup_{i=1}^k \bigcup_{j=1} \phi_j^i|$. Thus, for the *HiTi* graph model to be efficient, $|\bigcup_{i=1}^k \bigcup_{j=1} \phi_j^i|$ should be minimized. Since the size of *path view* edge sets depend on the number of boundary nodes created, it is important to come up with an effective hierarchical fragmentation method for a graph $G(V, E)$ by minimizing the number of boundary nodes for each level subgraph.

There exists ways of fragmenting planar graphs such as road maps so that the boundary nodes for each subgraph form a small set. McCormick et al. proposed the optimal graph decomposition algorithm having an exponential time complexity [28]. The planar graph decomposition, first shown by Lipton and Tarjan [26], has been exploited to yield hierarchical decompositions of such graphs by Miller et al. [29], [30]. Houstma et al. proposed a center-based greedy graph decomposition algorithm which relies on the manual picking of good center nodes [12]. Huang et al. later proposed a good planar graph decomposition algorithm, called spatial partitioning, which clusters graph links into partitions based on spatial proximity [15]. Their experimental analysis showed that the spatial partitioning method works well for fragmenting road maps into subgraphs with the number of their boundary nodes minimized.

TABLE 1
Boundary Node Set, *Semicut Edge Set*, and *Path View Edge Set* of SG_i in Fig. 2

i	$\delta(SG_i)$	$\mu(SG_i)$	$\phi(SG_i)$
1	{D,F}	{(D,G,2),(F,H,3)}	{(F,D,4)}
2	{G,H,M,O}	{(O,Q,2),(M,R,4),(G,F,2)}	{(G,O,6),(G,M,3),(H,M,3)}
3	{Q,R}	$\emptyset$	$\emptyset$

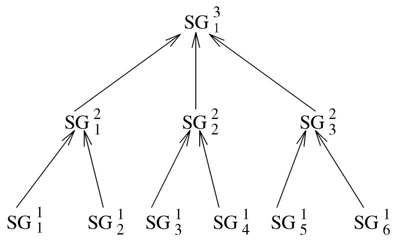

Fig. 3. An example of level 3 subgraph tree T.

4 Optimal Shortest Path Computation in a HiTi Graph

In this section, we discuss the use of a *HiTi* graph for computing a shortest path on $G(V, E)$. We first introduce a set of basic notations which will be used in the rest of this paper.

Definition 4.1. *Let SG represent a subgraph of graph $G(V, E)$. Then, $PC_{SG}(x, y)$ and $SPC_{SG}(x, y)$ represent the path cost and the shortest path cost from the nodes x to y only within SG, respectively.*

Definition 4.2. *Assume set X consist of a set of subgraphs (i.e., nodes) of a level $k+1$ subgraph tree T. Then,*

$A^j_T(X) =$
$\{\ y | y \text{ is a level } j \text{ ancestor subgraph of the subgraph in } X\},$

$A_T(X) = \bigcup_{j=1}^{k+1} A^j_T(X)$, *and*

$C_T(X) = \{Y | Y \text{ is a child subgraph of the subgraph in } X\}.$

Note that $C_T(\{SG^1_i\}) = \{SG^1_i\}$ for all the level 1 subgraphs SG^1_i (i.e., leaf nodes) of T.

Definition 4.3. *Assume set X consist of a set of subgraphs. Then, $\Phi(X)$ and $\Upsilon(X)$ consist of* semicut *and* path view *edge sets of all the subgraphs in X, respectively.*

$$\Omega(X) = \Phi(X) \cup \Upsilon(X),$$

and $\Delta(X)$ consists of boundary *node sets of all the subgraphs in X.*

Definition 4.4. *Let SG^l_i and SG^l_j be two distinct level l subgraphs defined in a level k subgraph tree T. Then, $lca(SG^l_i, SG^l_j)$ denote their least common ancestor subgraph.*

The examples of Definitions 4.2, 4.3, and 4.4 are illustrated through the level 4 subgraph tree T shown in Fig. 4. Assume that $X = \{SG^1_8, SG^1_{11}\}$. Then,

$$\begin{aligned}
&A^1_T(X) = \{SG^1_8, SG^1_{11}\},\\
&A^2_T(X) = \{SG^2_4, SG^2_5\},\\
&A_T(X) = \{SG^1_8, SG^2_4, SG^3_2, SG^4_1, SG^1_{11}, SG^2_5\},\\
&C_T(X) = \{SG^1_8, SG^1_{11}\},\\
&C_T(S^2_A(X)) = \{SG^1_8, SG^1_9, SG^1_{10}, SG^1_{11}, SG^1_{12}\},\\
&\Phi(X) = \{\mu^1_8, \mu^1_{11}\},\\
&\Upsilon(X) = \{\phi^1_8, \phi^1_{11}\},\\
&\Delta(X) = \{\delta^1_8, \delta^1_{11}\},
\end{aligned}$$

and $lca(SG^1_8, SG^1_{11})$ is SG^3_2.

4.1 Optimality of the Shortest Path Computed on the HiTi Graph

In order to prove the optimality of the shortest path cost computed on the *HiTi* graph, we present the following four theorems. The first three theorems will be used in proving the last theorem, which shows the optimality of the shortest path computation on the *HiTi* graph.

Theorem 4.1 shows that the shortest path cost between any pair of boundary nodes of a level l subgraph is the same as the one computed on the level $l-1$ *semicut* and *path view* edge sets of child subgraphs of the level l subgraph. Furthermore, this theorem gives us the basis for efficiently computing level l *path view* edge sets. That is, by using Theorem 4.1, we can recursively compute level l *path view* edge set ϕ^l_i only using level $l-1$ *path view* and *semicut* edge sets, $\Omega(C_T(SG^l_i))$.

Theorem 4.1. *Let SG^l_i be a level l subgraph node in a level $k+1$ subgraph tree T, where $k \geq 1$ and $1 \leq l \leq k+1$. Let $C_T(SG^l_i) = \{SG^{l-1}_1, SG^{l-1}_2, \ldots, SG^{l-1}_p\}$. For any pair of boundary nodes $x,\ y \in \Delta(C_T(SG^l_i))$, we have*

$$SPC_{SG^l_i}(x, y) = SPC_{\Omega(C_T(SG^l_i))}(x, y).$$

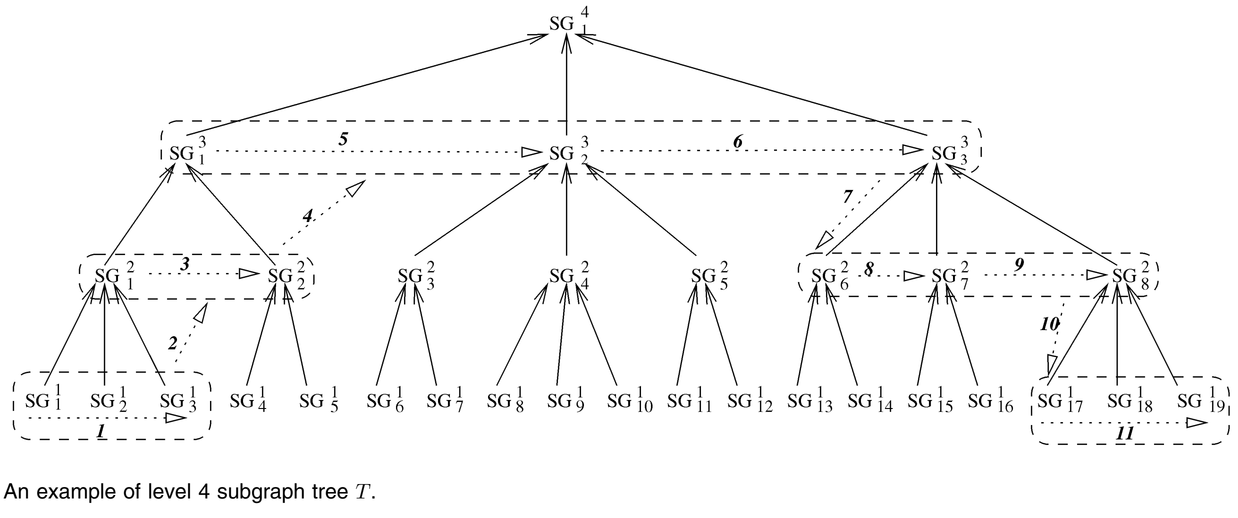

Fig. 4. An example of level 4 subgraph tree T.

Proof. The proof is given in the appendix. □

Theorem 4.2 gives an efficient method to compute the shortest path cost on a level l subgraph SG_i^l when both source and destination nodes, neither of them are level $l-1$ boundary nodes, are inside the same child subgraph SG_q^{l-1} of SG_i^l in a level $k+1$ subgraph tree. Instead of searching all the edges in SG_i^l for the computation, it shows that it only needs to search the edges in SG_q^{l-1} and the level $l-1$ *semicut* and *path view* edge sets of all the child subgraphs of SG_i^l.

Theorem 4.2. *Let SG_i^l be a level l subgraph node in a level $k+1$ subgraph tree T, where $k \geq 1$ and $1 \leq l \leq k+1$. Let $S_C(SG_i^l) = \{SG_1^{l-1}, SG_2^{l-1}, \ldots, SG_p^{l-1}\}$. For any node pair $x, y \in SG_q^{l-1}$, where $1 \leq q \leq p$, the following holds: $SPC_{SG_i^l}(x,y) = SPC_{SG_q^{l-1} \cup \Omega(C_T(SG_i^l))}(x,y)$.*

Proof. The proof is given in the appendix. □

Theorem 4.3 gives an efficient method to compute the shortest path cost on a level l subgraph SG_i^l when source and destination nodes are respectively in the two distinct children subgraphs SG_u^{l-1} and SG_v^{l-1} of SG_i^l in a level $k+1$ subgraph tree. Instead of searching all the edges in SG_i^l for the computation, it shows that it only needs to search the edges in SG_u^{l-1} and SG_v^{l-1}, and the level $l-1$ *semicut* and *path view* edge sets of all the child subgraphs of SG_i^l.

Theorem 4.3. *Let SG_i^l be a level l subgraph node in a level $k+1$ subgraph tree T, where $k \geq 1$ and $1 \leq l \leq k+1$. Let $C_T(SG_i^l) = \{SG_1^{l-1}, SG_2^{l-1}, \ldots, SG_p^{l-1}\}$. For any node pair $x \in SG_u^{l-1}$, $y \in SG_v^{l-1}$, where $1 \leq u, v \leq p$ and $u \neq v$, then $SPC_{SG_i^l}(x,y) = SPC_{SG_u^{l-1} \cup SG_v^{l-1} \cup \Omega(C_T(SG_i^l))}(x,y)$.*

Proof. The proof is given in the appendix. □

Based on Theorems 4.1, 4.2, and 4.3 above, Theorem 4.4 proves that, for any pair of nodes in $G(V,E)$, the shortest path cost computed on a selected part (i.e., subgraph) of a level k *HiTi* graph together with not necessarily distinct two level 1 subgraphs is the same as the one computed on $G(V,E)$.

Theorem 4.4. *Let a level $k+1$ subgraph tree T be constructed from $G(V,E)$, where $k \geq 1$ and T defines a level k HiTi graph $H^k(V^k, E^k)$. For any node pair $x \in SG_i^1$ and $y \in SG_j^1$, we have $SPC_G(x,y) = SPC_{SG_i^1 \cup SG_j^1 \cup D}(x,y)$ such that $D = \Omega(C_T(A_T(\{SG_i^1, SG_j^1\})))$.*

Proof. The proof is given in the appendix. □

From Theorem 4.4, it is easy to see how a *HiTi* graph can significantly reduce the search space necessary for a shortest path computation. That is, without using a *HiTi* graph, the search space would be $G(V,E)$. Furthermore, Theorem 4.4 allows more controlled storage management that is suitable for navigation systems (e.g., automobile navigation system) where available storage is limited. This is possible because we do not need to have an entire $G(V,E)$ to compute the shortest path. Instead, by Theorem 4.4, we only need a selected part of the level *k HiTi* graph and two level 1 edge sets (i.e., one for a source node and the other for a destination node).

We take a level 4 subgraph tree in Fig. 4 to exemplify Theorem 4.4. Assume that we want to find the shortest path from a source node $START$ in SG_1^1 to a destination node $DEST$ in SG_{19}^1. Then, the necessary search space is at most $E_1^1 \cup E_{19}^1$ together with the corresponding *semicut* and *path view* edge sets of the subgraphs enclosed by the dotted rectangles in Fig. 4.

4.2 Shortest Path Algorithm SPAH

We now describe the Shortest Path Algorithm named $SPAH$ based on Theorem 4.4. Algorithm $SPAH$ is shown in Fig. 5. $SPAH$ consists of two steps. The first step, by taking advantage of Theorem 4.4, selects the necessary search space by marking the subset of nodes in $\{V_1^1, V_2^1, \ldots, V_{n_1}^1\}$ with an edge level number. Each marked edge level number represents the lowest level number of the edges, which are incident to the marked node, $SPAH$ can traverse. In the second step, $SPAH$ applies a variation of A^* algorithm to the selected search space obtained in the first step. A^* uses the function $f(u, DEST)$ to estimate the cost of the shortest path from node u to $DEST$. In the domain of road maps, the function $f(u, DEST)$ computes the Euclidean distance between the node u and $DEST$. This is possible because the coordinates (i.e., longitude and latitude) of all nodes on a road map are assumed to be available. Assuming $(u.x, u.y)$ and $(DEST.x, DEST.y)$ are the corresponding coordinates of the nodes u and $DEST$, $f(u, DEST)$ computes

$$\sqrt{(DEST.x - u.x)^2 + (DEST.y - u.y)^2}.$$

Since Euclidean distance always underestimates the actual shortest path between node u and $DEST$, A^* finds the optimal shortest path. The detailed proof for the optimality of A* algorithm is found in [10].

$SPAH$ traverses the edges, incident to the marked nodes, from node $START$ in SG_i^1 to node $DEST$ in SG_j^1 to find the optimal shortest path cost. For this computation, $SPAH$ takes advantage of the precomputed shortest path costs stored in *path view* edge sets. Although the precomputed shortest path costs are not necessarily the global shortest path on $G(V,E)$, $SPAH$ guarantees giving the correct global shortest path by using a specially created subgraph tree, which is proven in Theorem 4.4. The subgraph tree provides information about which subgraphs can be skipped in the computation of the global shortest path.

Edge traversal of $SPAH$ consists of two phases, the *ascending* and *descending* phases. During the *ascending* and *descending* phases, $SPAH$ traverses edges in a nondecreasing and nonincreasing edge level order, respectively. Note that each of the considered paths has its own thread of processing. Some of the paths are still in the *ascending* phase, whereas the others are in the *descending* phase.

The preceding Fig. 4 also shows an example of the search progress in $SPAH$ when the source and destination nodes are in SG_1^1 and SG_{19}^1, respectively. The five dotted rectangles in the figure include the subgraphs whose *path view* and *semicut* edge sets are selected by step 1 of $SPAH$ as the necessary search space. At step 2, $SPAH$ first traverses the edges in subgraph $SG_1^1(V_1^1, E_1^1)$ until it reaches the boundary nodes of SG_1^1. It then enters into the ascending

```
begin
/* Assume we have a level $k$ $HiTi$ graph defined on a level $k+1$ $T$; */
/* $\{SG_1^1, SG_2^1, \ldots, SG_{n_1}^1\}$ and $H^k(V^k, E^k)$ are maintained as adjacency lists; */
/* All the edges in $SG_t^1$ for $1 \le t \le n_1$ are assumed to be level 0 edges; */
/* All the edges in $E^k$ have their corresponding edge level */

Step 1:
  For $1 \le i, j \le n_1$, find $SG_i^1$ and $SG_j^1$ where $START \in SG_i^1$ and $DEST \in SG_j^1$;
  Let $l$ be the level number of $lca(SG_i^1, SG_j^1)$;
  for ($p = k+1$; $p > l$; $p--$)      /* $A_T^p(SG_i^1) = A_T^p(SG_j^1)$ */
    Mark all nodes in $\Delta(C_T(A_T^p(SG_i^1)))$ with level $p-1$;
  if ($l = 1$)      /* $SG_i^1 = SG_j^1$ */
    Mark all nodes in $SG_i^1$ with 0;
  else {
    for ($p = l$; $p > 1$; $p--$) {
      Mark all nodes in $\Delta(C_T(A_T^p(SG_i^1)))$ with level $p-1$;
      Mark all nodes in $\Delta(C_T(A_T^p(SG_j^1)))$ with level $p-1$; }
    Mark all nodes in $SG_i^1$ and $SG_i^1$ with 0; }

Step 2:
  Let $\lambda(x) = \infty$ for all the marked nodes $x$ and $\lambda(START) = 0$;
  $FSet = \{ START \}$; $ESet = \emptyset$;
  while ($FSet \neq \emptyset$) {
    Select $u$ from $FSet$ with minimum $\lambda(u) + f(u, DEST)$;
    /* the function $f(u, DEST)$ estimates the Euclidean distance
       from node $u$ to $DEST$ */
    $FSet = FSet$ - $\{ u \}$; $ESet = ESet \cup \{ u \}$;
    if ($u = DEST$) stop;
    Let $\tau$ be the highest level number of the edges incident on node $u$;
    if ($u$ is marked) {
      Let $\beta$ be the marked level number on node $u$;
      for each edge $u \xrightarrow{z} v$ with the levels from $\beta$ to $\tau$ {
        if (($v \notin FSet$) and ($v \notin ESet$)) {
          $\lambda(v) = \lambda(u) + z$;
          $FSet = FSet \cup \{ v \}$; }
        else {
          if ($\lambda(v) > \lambda(u) + z$) $\lambda(v) = \lambda(u) + z$; } } } }
end
```

Fig. 5. $SPAH$ finding the shortest path cost from $START$ to $DEST$.

phase in which the *semicut* and *path view* edges with nondecreasing level number are traversed. The sequence of edge traversal in the ascending phase is illustrated by the dotted arrows with number 1, 2, 3, 4, 5, and 6. The number represents the order of the search progress. When $SPAH$ encounters the boundary nodes of SG_3^3, it enters into the descending phase where the *semicut* and *path view* edges with nonincreasing level numbers are traversed. The sequence of edge traversal in the descending phase is illustrated by the dotted arrows with number 7, 8, 9, 10, and 11. When $SPAH$ reaches the boundary nodes of SG_{19}^1, it traverses the edges in subgraph $SG_{19}^1(V_{19}^1, E_{19}^1)$ until it reaches the destination node.

After obtaining the shortest path cost from $SPAH$, a navigator (e.g., driver) may want to find a more fine-grained path connection on some edges (i.e., high-level edges) with a level number greater than 0. This can be accomplished by specializing a high-level edge. High-level *path view* edges are specialized by representing them in terms of lower level edges. For this purpose, we keep actual shortest path information for each corresponding *path view* edges. A high-level *semicut* edge cannot be specialized in terms of lower level edges. This is because all *semicut* edges in $\cup_{i=1}^{k} \cup_{j=1} \mu_j^i$ are the edges in $G(V, E)$.

5 Performance Analysis

For the analysis of Algorithm $SPAH$, we create two-dimensional grid graphs $G(V, E)$ with four adjacent nodes. Two-dimensional grid graphs are considered as typical examples of road maps [24], [38]. In grid graph G, $|V|$, and $|E|$ are equal to 800×800 nodes and $4 \times 800 \times 799$ directed edges. From $G(V, E)$, we create a level 4 subgraph tree T, where each level 1 subgraph $SG_i^1(V_i^1, E_i^1)$ has $|V_i^1| = 100 \times 100$, $|E_i^1| = 4 \times 100 \times 99$. Thus, the level 4 subgraph tree T consists of 64 level 1, 16 level 2, 4 level 3, and 1 level 4 subgraphs, which are shown in Fig. 6.

We use the above level 4 subgraph tree ST to generate a level 3 *HiTi* graph for the analysis of $SPAH$ in the following sections.

5.1 Comparison between $SPAH$ and A* Algorithm

To show the search space savings of $SPAH$ over the traditional A* algorithm (presented in [38]), we create five level 3 *HiTi* graphs where $10 \le |\delta_i^1| \le 20$ and where the edge cost is generated based on a uniform distribution [100,120] with five different seeds. We represent $|\delta_i^1|$ as the total number of boundary nodes defined on level 1 subgraph SG_i^1. Next, we create five plain grid graphs which are simply unions of χ^1 and level 1 subgraphs used in the above five level 3 *HiTi* graphs. For each level 3 *HiTi* graph and plain grid graph created above, we compute 20 different shortest paths randomly prefixed pairs of

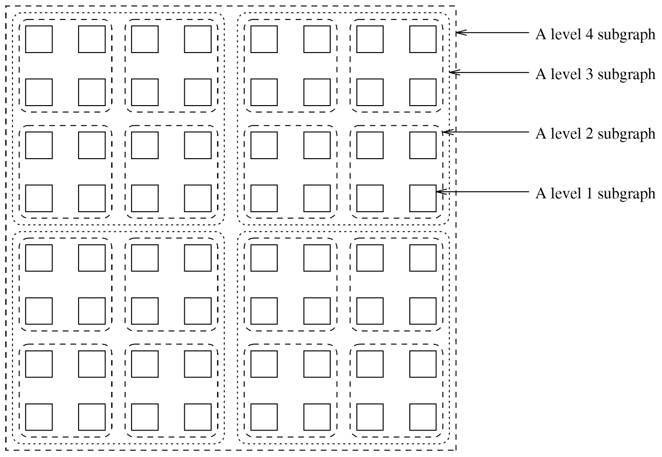

Fig. 6. A 800×800 grid graph partitioned according to the level 4 subgraph tree.

source and destination nodes. Let H_n and A_n be the total number of edges visited by $SPAH$ and A*, respectively. Note that throughout this paper, we multiply the estimation of $f(u, DEST)$ by 100 to normalize the estimation with respect to the edge cost. The number 100 is used for this normalization because the Euclidean distance between two adjacent nodes is 1 and the edge cost between two nodes is at least 100. We compare $SPAH$ and A* by observing the ratio A_n/H_n. These values are then averaged over the five different level 3 *HiTi* and plain grid graphs with the same source and destination nodes. They are shown in Fig. 7 where the numbers on x axis represent 20 ordered $< source, destination >$ pairs with path length increasing from 1 to 20.

Fig. 7 clearly shows how effectively Algorithm $SPAH$ cuts down the search space over the traditional A* algorithm. It is interesting to observe that the ratio A_n/H_n increases rapidly as the path lengths from source to destination increase. This occurs because the search space A* algorithm needs to explore grows exponentially, whereas that of $SPAH$ grows very slowly due to the hierarchical structure of $HiTi$ graphs.

5.2 Effects of Edge Cost Distribution

In this section, we study the effects of edge cost distributions on the performance of $SPAH$. For this study, we generate 4×5 (i.e., four uniform distributions with five seeds) level 3 *HiTi* graphs where $10 \leq |\delta_i^1| \leq 20$. Note that the four uniform distributions [100,120], [100,200], [100,300], and [100,500] correspond to 20 percent, 100 percent, 200 percent, and 400 percent variations of edge costs, respectively. We apply $SPAH$ to each level 3 *HiTi* graph by randomly creating 50 different $< source, destination >$ pairs and then averaging the cost. Let MA and MD symbolize $SPAH$ when the estimator $f(u, DEST)$ gives Euclidean distance and zero (i.e., no estimation), respectively. Fig. 8 shows the effect of edge cost distributions on the performance of $SPAH$ in terms of H_n. Note that the values of H_n are averaged over five seeds.

When $f(u, DEST)$ gives Euclidean distance estimation, the performance of $SPAH$ deteriorates as the variation of edge cost is increased. The primary reason is that

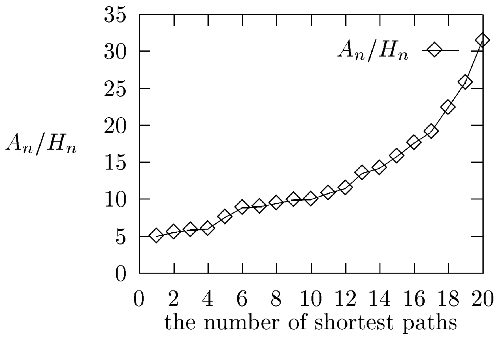

Fig. 7. Performance comparison between A* and Algorithm $SPAH$.

increasing the variation degrades the quality of Euclidean distance estimation to the shortest path. This degradation of Euclidean distance estimation seems to have a more severe impact on the performance of MA initially (i.e., between 20 percent and 200 percent) than for the rest (i.e., between 200 percent and 400 percent). Unlike MA, MD gives a very stable performance with varying edge cost distribution. Since Euclidean distance estimation constitutes the computational overhead, MD is better suited for topographical road maps with large edge cost variations than MA.

5.3 Effects of the Number of Hierarchical Levels

We examined the effects of different levels of *HiTi* graphs on the performance of $SPAH$ (i.e., *MA* and *MD*). We constructed different levels of *HiTi* graph out of the same set of level 1 subgraphs. For this analysis, we create two (i.e., $4 \leq |\delta_i^1| \leq 8$ and $10 \leq |\delta_i^1| \leq 20) \times 5$ (i.e., an edge cost distribution [100,200] with five seeds) level 3 *HiTi* graphs. Similarly, we create 2×5 level 1 and level 2 *HiTi* graphs. Then, we measure average H_n of MA and MD the same way as we did in Section 5.2. They are shown in Tables 2 and 3.

As we can see from Tables 2 and 3, higher level *HiTi* graphs do not necessarily guarantee the better performance when using $SPAH$ than the lower level *HiTi* graphs. It depends on the average search space for $SPAH$. The average search space for $SPAH$ on a level k $HiTi$ graph, represented by ξ^k, is formulated as follows:

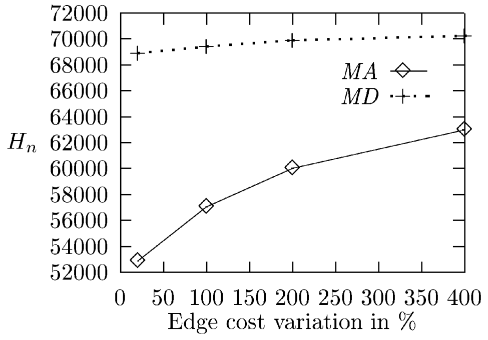

Fig. 8. Effect of edge cost on the performance of MA and MD.

TABLE 2
Effects of Level HiTi Graphs on H_n when $10 \leq |\delta_i^1| \leq 20$

	level 1 $HiTi$ graph	level 2 $HiTi$ graph	level 3 $HiTi$ graph
H_n for MA	57539	57421	57161
H_n for MD	72972	72756	70375

TABLE 3
Effects of Levels HiTi Graphs on H_n when $4 \leq |\delta_i^1| \leq 8$

	level 1 $HiTi$ graph	level 2 $HiTi$ graph	level 3 $HiTi$ graph
H_n for MA	51348	51014	51498
H_n for MD	60173	60132	60714

$$\xi^k = 2 \cdot |e| + \sum_{i=1}^{n_k} |\phi_i^k| + \frac{2}{n_k}\sum_{i=1}^{n_{k-1}} |\phi_i^{k-1}| + \frac{2}{n_{k-1}}\sum_{i=1}^{n_{k-2}} |\phi_i^{k-2}| + \cdots + \frac{2}{n_3}\sum_{i=1}^{n_2} |\phi_i^2| + \frac{2}{n_2}\sum_{i=1}^{n_1} |\phi_i^1|.$$

Note that $|e|$ represents the average total number of edges in a level 1 subgraph and n_l represents the total number of level l subgraphs where $1 \leq l \leq k$. If $\xi^p < \xi^q$, where $p > q$, then $SPAH$ is likely to perform better on the higher level p *HiTi* graph than on the lower level q *HiTi* graph. Otherwise, the higher level *HiTi* graph is not likely to provide a performance advantage over the lower level *HiTi* graph. Table 4 shows the values of ξ^1, ξ^2, and ξ^3 corresponding to levels 1,2, and 3 $HiTi$ graphs used in Tables 2 and 3.

Table 4 verifies our conjecture on the performance of $SPAH$ for different levels of $HiTi$ graphs. An interesting thing to note from Tables 2 and 3 is that the search space (i.e., H_n) does not vary significantly going beyond a level 1 *HiTi* graph. This is because our experiments were done with the grid graphs having the property that the difference between ξ^i and ξ^{i+1} does not vary significantly as i increases. We believe that, for most road maps, this is the case. As a result, creating higher level *HiTi* graphs does not contribute to the reduction in computation time. From this observation, we can conclude that a level i *HiTi* graph is good enough for road map applications where the difference between ξ^i and ξ^{i+1} is small.

5.4 Memory Requirement

A* shortest path algorithm has been shown to be more efficient than the breadth-first search single pair shortest path algorithm when the database can fit in main memory [24], [32]. This is likely to be the case for Algorithm $SPAH$. That is, since $SPAH$ processes the whole graph in terms of *path view* and *semicut* edge sets of the smaller subgraphs, it might be expected that each time the average main memory need will be small.

For a more detailed analysis of the above claim, we use a simplified analytical model suggested in [22]. In this model, a grid graph $G(V, E)$ is exclusively partitioned into a set of n_1 level 1 subgraphs where $|V| = v = m \times m$ and $n_1 = f \times f$. The total number of boundary nodes is $2m(f-1) \approx 2\sqrt{vn_1}$. On average, each level 1 subgraph has v/n_1 nodes. Since there are at most $\sqrt{v/n_1}$ boundary nodes on each side of a level 1 subgraph, a level 1 subgraph can have up to $4\sqrt{v/n_1}$ boundary nodes. Then, we use the following four parameters represented in terms of v and n_1.

- n_1: the number of level 1 subgraphs that exclusively partition a grid graph $G(V, E)$,
- $n_j = n_1/4^{j-1}$: the number of level j subgraphs,
- $b_j = 4 \times (2^{j-1}\sqrt{v/n_j})$: the average number of boundary nodes for each level j subgraph, and
- $w_j = b_j \times (b_j - 1)$: the average size of level j *path view* edge set.

Let MR^k be the average size of the part of the level k HiTi graph $SPAH$ needs to load into main memory in the worst case (i.e., when $lca(SG_i^1, SG_j^1)$ is $k+1$). Based on Theorem 4.4 and the above simplified analytical model, we can formulate MR^k as follows:

$$MR^k = n_k w_k / n_{k+1} + 2n_{k-1}w_{k-1}/n_k + 2n_{k-2}w_{k-2}/n_{k-1} + \cdots + 2n_1 w_1/n_2.$$

In Fig. 9, we compare the memory requirements of $SPAH$ when levels 1, 2, and 3 $HiTi$ graphs are used. We fixed the number of level 1 subgraphs (i.e., n_1) at 64 and measure MR^1, MR^2, and MR^3 by varying the number of nodes in G from 100×100 to 800×800.

Fig. 9 shows that the average memory requirements of a level 3 $HiTi$ graph increases very sharply with the increase of the number of nodes. However, for a level 1 $HiTi$ graph, its memory requirement is increasing very slowly. For example, its memory requirement at 640,000 nodes are

TABLE 4
Effects of Levels of HiTi Graphs on ξ^k

	ξ^1	ξ^2	ξ^3
$10 \leq |\delta_i^1| \leq 20$	99000	98737	98487
$4 \leq |\delta_i^1| \leq 8$	82272	83364	83332

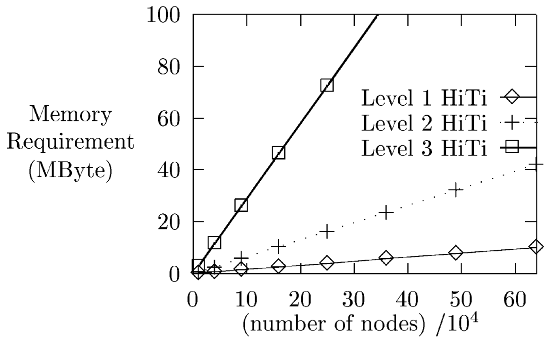

Fig. 9. Memory requirement of levels 1, 2, and 3 *HiTi* graphs.

```
/* Assume we have a level $k$ $HiTi$ graph defined on a level $k+1$ subgraph tree $T$; */

1. Let $U$ be a set of edges of $G(V,E)$ whose costs are updated.
2. $U$ can be represented as $U^0 \cup U^1 \cup U^2 \cup \cdots \cup U^k$ where
   $U^0$ contains the updated edges in $\cup_{t=1}^{n1} SG_t^1$,
   $U^1$ contains the updated edges in $\cup_{t=1}^{n1} \mu_t^1$,
   $U^2$ contains the updated edges in $\cup_{t=1}^{n2} \mu_t^2$, ... ,
   $U^k$ contains the updated edges in $\cup_{t=1}^{nk} \mu_t^k$.
3. Identify the level 1 subgraphs whose edges are in $U^0$ and let $l = 1$.
4. Recompute the level $l$ path view edge set of those identified level $l$ subgraphs.
5. Let $X$ contain the level $l$ subgraphs whose path view edge sets are updated by the
   recomputation.
6. Let $Y$ contain the level $l$ subgraphs whose semi-cut edge sets include the edges in $U^l$.
7. For each level $l+1$ subgraph $SG_i^{l+1}$ in $A_T^{l+1}(X \cup Y)$, recompute the level $l+1$ path
   view edge set $\Upsilon(SG_i^{l+1})$ by using $\Omega(C_T(SG_i^{l+1}))$.
8. Let $l = l+1$. If $l < k$ then goto step 4. Otherwise stop.
   /* When $l = k$, the recomputation of level $k+1$ path view edge sets is not necessary
   since the path view and semi-cut edge sets of a level $k+1$ subgraph (i.e., the root node
   of $T$) are always empty. */
```

Fig. 10. Updating algorithm for a level k $HiTi$ graph.

10 Mbytes, which is small enough to fit into main memory. Although the average memory size $HiTi$ graphs require is not much as it was shown in Fig. 9, it may not fit in main memory for some cases. In those cases, $SPAH$ should compute SPSP on a disk-based $HiTi$ graph by reducing I/O cost as much as possible. For this, we need efficient spatial access methods for path computation queries over network data such as road maps. Network data should be efficiently clustered based on connectivity of nodes rather than geographic proximity.

A few have done research in connectivity-based access methods [3], [6], [13], [17], [25]. However, these proposed methods are not suitable for aggregate queries, e.g., path evaluation, over general networks including road maps. Shekhar and Liu proposed an efficient access method, CCAM, to efficiently support aggregate queries over general networks such as road maps [40]. Their experiments with path computations on the Minneapolis road map show the efficiency of the CCAM method. We believe that the CCAM method can be applied to store a $HiTi$ graph on the disk to reduce I/O cost for SPSP computation.

6 Updating a HiTi Graph

In a $HiTi$ graph, it is necessary to modify the graph whenever there are updates on the costs of the edges in $G(V,E)$. For example, road maps need to be updated as traffic conditions change. Thus, an efficient updating algorithm for a $HiTi$ graph is essential for applications like road navigation. In Fig. 10, we give an efficient updating algorithm for a level k $HiTi$ graph.

The update cost of our algorithm in Fig. 10 is linear with respect to the number of subgraphs in a level $k+1$ subgraph tree T. That is, the influence of changing an edge cost is limited to the *path view* edge sets of the ancestor subgraphs of the level 1 subgraph the link is in and *path view* edge sets of all other subgraphs are unaffected. This is possible because the shortest paths are precomputed between all the boundary nodes of each SG_i^l only within SG_i^l, where $1 \leq l \leq k$ and $1 \leq i \leq n_l$. Thus, when changing an edge cost, the update cost, UC, is the size of the *path view* edge sets that needs to be recomputed as shown in step 7 in Fig. 10. For a more detailed analysis of UC, we use a simplified analytical model mentioned in Section 5.4. For a level k HiTi graph, UC is formulated as $\sum_{l=1}^{k} w_j$. To see how UC behaves when varying the number of level 1 subgraphs updated, assume that 64 level 1, 16 level 2, and 4 level 3 subgraphs are defined on a grid graph G of 10,000 nodes. Fig. 11 shows the update costs of levels 1, 2, and 3 $HiTi$ graphs.

We observe in Fig. 11 that the update costs of levels 1 and 2 $HiTi$ graphs are increasing almost linearly with the increase of the number of level 1 subgraphs changed. However, the update cost of a level 3 $HiTi$ graph is increasing linearly from 1 to 16 subgraphs, and jump sharply to a higher cost at 17 subgraphs and so on. This is because lower level *path view* edge sets updated are likely to belong to the same next higher level *path view* edge set that needs to be recomputed. As a result, the jump of the update cost occurs when the updated lower level *path view* edge sets have different next higher level *path view* edge sets that need to be recomputed. The above tendency is also shown in a level 2 $HiTi$ graph in the figure, although the jump of the update cost is shown to be relatively small compared to that in a level 3 $HiTi$ graph. We conclude that a $HiTi$ graph provides an efficient structure which allows the edges in the higher level *path view* edge sets to be less sensitive to the updates at lower level edges.

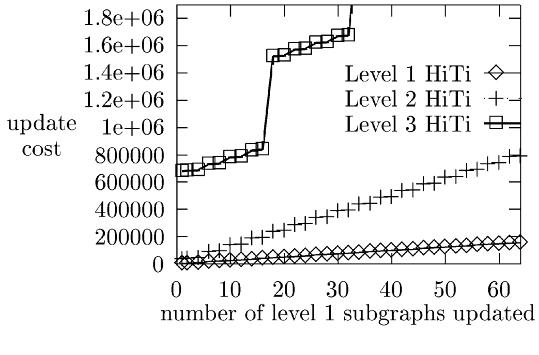

Fig. 11. Update costs for levels 1, 2, and 3 $HiTi$ graphs.

While our approach focuses on maintaining the different levels of path views (i.e., the precomputed shortest paths), a significant amount of research has been done on maintaining materialized views of the transitive closures including path information [1], [5], [19], [23]. In their approaches, incremental update techniques are used to avoid recomputing the entire transitive closures from scratch. However, their update techniques do not provide the upper bound where an update propagation is limited as is the case in our approach. This is because their materialized view of transitive closures are not based on the graph decomposition like a HiTi graph model. Thus, in the worst case, most of the materialized view of transitive closures may have to be updated.

7 Comparison with Other Similar Works

In this section, we analyze other similar works in comparison with our $HiTi$ graph model. Two research results are quite relevant to the $HiTi$ graph approach. They are $HEPV$ and *Hub Indexing* methods [21], [22], [11]. In the $HEPV$ method, it first generates an FPV(Flat Path View) for each subgraph by precomputing all-pair shortest paths only within the corresponding subgraph.[1] By utilizing the path information in all FPVs, it then constructs a *supergraph* for all subgraphs. The supergraph captures the path information between the boundary nodes of each subgraph. $HEPV$ constructs an FPV for the supergraph by precomputing globally optimal all-pair shortest paths between the boundary nodes of all subgraphs. We call this FPV as SFPV from hereon.

In the *Hub Indexing* method, it generates a *distance table* for all subgraphs by precomputing the shortest path costs[2] between the boundary nodes of each subgraph and its interior nodes (i.e., nonboundary nodes of each subgraph) only within it's corresponding subgraph. We denote the part of the distance table as PDT that contains the shortest path information corresponding to each subgraph. The Hub Indexing method also generates a *Hub Set* for all subgraphs by precomputing the globally optimal all-pair shortest paths between the boundary nodes of all subgraphs.

Like the $HiTi$ graph method, these two methods also limit the influence of changing an edge cost to the confines of the subgraph the link is in, and all other subgraphs remain unaffected. However, all the shortest paths stored in SFPV and the Hub Set are affected by an update due to their global optimality. By considering the characteristics of the above two methods, we believe that they will have more storage overhead and update maintenance problems when large road maps have to be considered.

Unlike $HEPV$ and *Hub Indexing* methods, the $HiTi$ graph model does not suffer from the excessive storage overhead for maintaining a large amount of precomputed path information. This is because in the $HiTi$ graph model, we only precompute the shortest paths between the boundary nodes of each subgraph only with respect to that subgraph. The reduction in the storage overhead of a $HiTi$ graph model over that of the $HEPV$ and *Hub Indexing* approaches, also gives much less update cost than the other two approaches, since the amount of the shortest path recomputation necessary for updating the $HiTi$ graph is a lot less than for the $HEPV$ and *Hub Indexing* approaches.

For a more detailed analysis, we reuse a simplified analytical model mentioned in Section 5.4. Since the Hub Indexing method deals with only level 1 subgraphs, for a fair comparison between these three methods, we only consider level 1 subgraphs for this analysis.[3] Thus, from hereon, mentioned otherwise, we denote level 1 subgraphs as just subgraphs. We use the following five parameters represented in terms of v and n_1:

- v: the total number of nodes in a grid graph $G(V,E)$,
- n_1: the number of level 1 subgraphs that exclusively partition a graph $G(V,E)$,
- $t = 2\sqrt{vn_1}$: the total number of boundary nodes for G,
- $b_1 = 4\sqrt{v/n_1}$: the average number of boundary nodes for each level 1 subgraph, and
- $i_1 = v/n_1 - b_1$: the average number of interior nodes of each level 1 subgraph.

By utilizing the above five parameters, we analyze each method in terms of *update cost*, *storage overhead*, and *SPSP computation time*. We represent the update cost in terms of the number of affected precomputed shortest paths that need to be recomputed. The storage overhead is represented by the total number of the precomputed shortest paths. SPSP computation time gives the computational complexity of each method.

We first discuss the cost of updates when changing an edge cost. For the $HEPV$ approach, the cost consists of the cost of updating an FPV, i.e., $(b_1 + i_1)(b_1 + i_1 - 1)$ and that of a supergraph, i.e., $t(t-1)$. In the *Hub Indexing* method, the cost consists of updating the *Hub Set*, i.e., $t(t-1)$, and PDT, i.e., $b_1 i_1$. In the $HiTi$ graph method, the cost is updating a *path view* edge set, i.e., $b_1(b_1 - 1)$.

Next, we discuss a storage overhead for each method. For the $HEPV$ method, the storage overhead comes from the size of all FPVs, i.e., $(b_1 + i_1)(b_1 + i_1 - 1)n_1$, and the size of SFPV, i.e., $t(t-1)$. For the *Hub Indexing* method, the overhead consists of the size of the distance table, i.e., $b_1 i_1 k_1$, and the size of the *Hub Set*, i.e., $t(t-1)$. For the $HiTi$ graph method, the storage overhead is the size of all the *path view* edge sets, i.e., $b_1(b_1 - 1)n_1$.

Finally, we discuss SPSP computation time for each method. The detail SPSP computation algorithms for the $HEPV$ and *Hub Indexing* methods are given in [11], [21], [22]. For the $HEPV$ method, the time complexity is dominated by searching a necessary portion, i.e., b_1^2, of SFPV. For the Hub Indexing method, the time complexity mainly comes from searching PDT (i.e., searching a subgraph with cost i_1^2) and a part of the Hub Set, i.e., b_1^2. For the $HiTi$ graph method, the time complexity consists of searching two level 1 subgraphs, i.e., $2i_1^2$, and a level 1 $HiTi$ graph, i.e., t^2. We summarize the performance comparison of all three methods in Table 5.

We now analyze the performance of the $HEPV$, *Hub Indexing*, and $HiTi$ graph methods experimentally by using the results given in Table 5. In Fig. 12, we first measure

1. The shortest paths computed here are not necessarily globally optimal.

2. Like the $HEPV$ approach, the shortest paths are not necessarily globally optimal.

3. In $HEPV$ and $HiTi$ graph methods, general multilevel subgraphs are also handled.

TABLE 5
Cost Comparisons of the $HEPV$, *Hub Indexing*, and $HiTi$ Graph Methods

Methods	Update cost	Storage overhead	SPSP Computation time
HEPV	$(b_1+i_1)(b_1+i_1-1)+t(t-1)$ $=O(v^2/n_1^2+vn_1)$	$(b_1+i_1)(b_1+i_1-1)n_1+t(t-1)$ $=O(v^2/n_1+vn_1)$	b_1^2 $=O(v/n_1)$
Hub Indexing	$b_ii_1+t(t-1)=O(vn_1)$	$b_1i_1n_1+t(t-1)=O(vn_1)$	$i_1^2+b_1^2=O(v^2/n_1^2)$
HiTi Graph	$b_1(b_1-1)=O(v/n_1)$	$b_1(b_1-1)n_1=O(v)$	$2(b_1+i_1)^2+t^2=O(v^2/n_1^2+vn_1)$

update cost, *storage overhead*, and *SPSP computation time* for each method by varying the number of subgraphs, i.e., n_1 defined on a grid graph of 10,000 nodes. Fig. 12a shows that the update cost of a $HiTi$ graph is much better than that of the $HEPV$ and *Hub Indexing* methods. This is because the influence of changing an edge cost in the $HiTi$ graph method is only limited to the *path view* edge set of the corresponding subgraph the link is in. However, in the $HEPV$ and *Hub Indexing* methods, the influence of the update is propagating not only to FPV and PDT for the corresponding subgraph but also to SFPV and the Hub Set. Note that the update cost in the *Hub Indexing* method are smaller than the one in $HEPV$ since PDT, i.e., b_1i_1 is much smaller than the size of the FPV, i.e, $(b_1+i_1)(b_1+i_1-1)$.

We also note that the update cost in the $HiTi$ graph method is decreasing with the increase in the number of subgraphs. However, the update costs of the $HEPV$ and *Hub Indexing* methods are decreasing up to 25 and 16 subgraphs, respectively, and then start to increase narrowing down their gaps. This happens because of the following two reasons:

- Increasing the number of subgraphs in the $HEPV$ and *Hub Indexing* methods has a positive effect on reducing their update costs up to 25 and 16 subgraphs, respectively.
- After that, the FPV size, i.e., $t(t-1)$, of the supergraph and the *Hub Set* size, i.e., $t(t-1)$ are becoming the dominant factor of their update costs.

From this, we think that the update cost for $HEPV$ and *Hub Indexing* methods will be almost the same for a grid graph having a large number of subgraphs. Fig. 12b shows that the storage overhead of the $HiTi$ graph method is not sensitive to the varying number of subgraphs as opposed to the $HEPV$ and *Hub Indexing* methods.

The storage overhead of the $HEPV$ and *Hub Indexing* methods has similar characteristics as those of their update costs. The justification for this is given by the same two reasons given above. For a SPSP computation cost, Fig. 12c shows that the $HEPV$ and *Hub Indexing* methods[4] give the comutation time that is much less than that for the HiTi graph model. This is expected from Table 5. Fig. 12c also shows the optimal number of subgraphs (i.e., 36 subgraphs) for the $HiTi$ graph method given the grid graph size fixed at 10,000 nodes. In general, by considering *update cost*, *storage overhead*, and *SPSP computation time*, we can decide the optimal number of subgraphs for each of the three methods when the size of a graph is fixed. For example, the optimal number of subgraphs for the Hub Indexing method is 16, 25, or 36 depending on which cost factor is given the highest priority.

4. Note that the computation time in $HEPV$ and *Hub Indexing* methods is decreasing with the increase of the number of subgraphs.

In Fig. 13, we measure *update cost*, *storage overhead*, and *SPSP computation time* for each method by varying the number of nodes in grid graphs from 100×100 to 150×150, 200×200, 250×250, $\ldots$, and 400×400. Note that we fixed the number of subgrapahs at 36. Figs. 13a and 13b shows that the HiTi graph method outperforms the $HEPV$ and *Hub Indexing* methods in terms of update cost and storage overhead. However, in terms of SPSP computation cost, the $HEPV$ method outperforms the rest of two methods as shown in Fig. 13c. Note that the performance of the *Hub Indexing* method is always between the $HiTi$ graph and $HEPV$ methods. The above performance tendency can be explained by the amount of precomputed shortest paths stored in each method. That is, the amount of precomputed shortest paths are increased from the $HiTi$ graph to the *Hub Indexing* and $HEPV$ methods.

8 HiTi Graph-Based Inter Query Parallel Shortest Paths Computation

In this section, we study the parallel processing for computing *inter* query shortest path based on the $HiTi$ graph. Efficient and fast computation of multiple single pair shortest paths are very critical for those automobile navigation systems where all route computations are performed on a single server. For this purpose, we propose a new *inter* query parallel shortest path algorithm named $ISPAH$ in the following section.

8.1 Description of ISPAH Algorithm

Before describing $ISPAH$, we first compare $SPAH$ with *OTOpar* [31]. *OTOpar* uses two processors where each processor uses A* algorithm with Manhattan distance estimation to build its corresponding tree, one rooted at the source node and the other rooted at the destination node. Note that $SPAH$ is an improved A* algorithm which takes advantage of the $HiTi$ graph. Thus, for fair comparison, we modified *OTOpar* so that it also utilizes the $HiTi$ graph. In other words, in the modified *OTOpar* named *MOTOpar*, we use $SPAH$ for building two trees from both source and destination nodes. Note that we use Euclidean distance for the lower bound estimation in *MOTOpar*. This is because Manhattan distance estimation does not guarantee the optimal shortest path generation.

We implemented $SPAH$ and *MOTOpar* on a BBN GP1000 shared memory multiprocessor system, which has a nonuniform memory architecture. The BBN GP1000 multiprocessor currently consists of 85 nodes, each one with 4MBytes of local memory, linked together by a high-speed butterfly switch. In this system, the globally shared memory is the sum of the memories local to all processors. Thus, the size of available main memory increases with an increasing number of nodes in the system. The BBN GP1000 system can have up to 250 processing nodes.

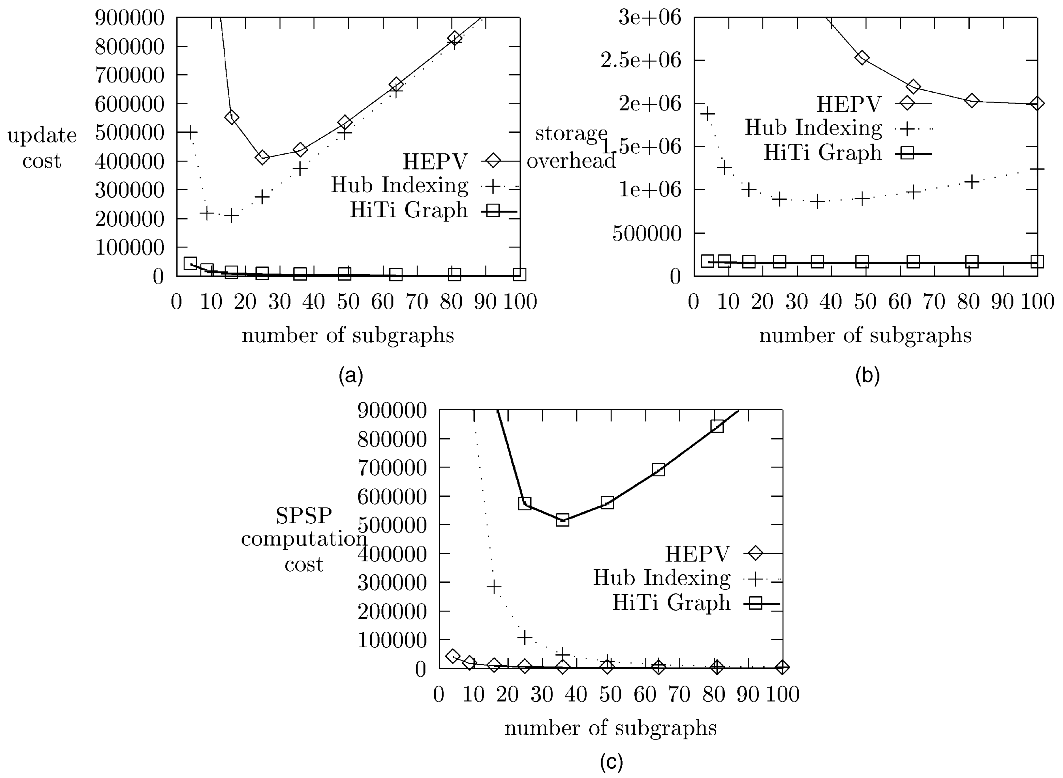

Fig. 12. Cost comparison on a grid graph of 10,000 nodes with varying number of subgraphs. (a) Update cost, (b) storage cost, and (c) SPSP computation cost.

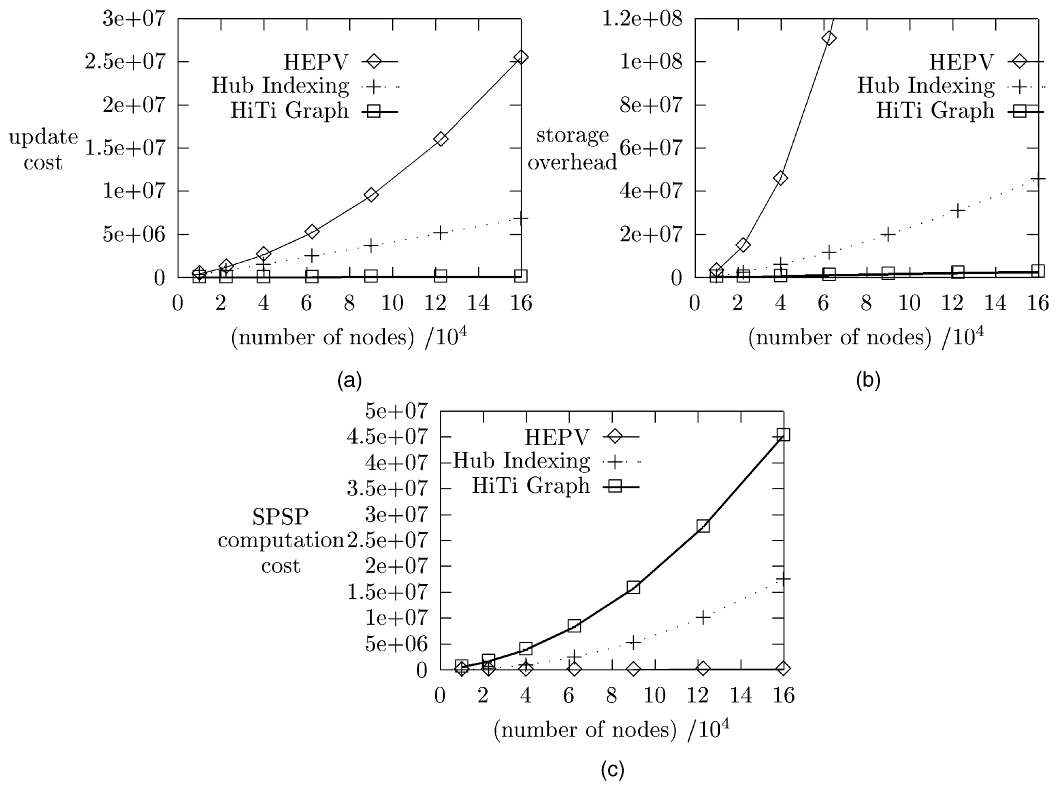

Fig. 13. Cost comparison on the varying sizes of grid graphs having 36 subgraphs. (a) Update cost, (b) storage cost, and (c) SPSP computation cost.

For our analysis, we create two-dimensional grid graphs $G(V,E)$ with four adjacent nodes as we did in Section 5. In grid graph G, $|V|$, and $|E|$ are equal to 400×400 nodes and $4 \times 400 \times 399$ directed edges. From $G(V,E)$, we create a level 4 subgraph tree ST where each level 1 subgraph $SG_i^1(V_i^1, E_i^1)$ has $|V_i^1| = 50 \times 50$, $|E_i^1| = 4 \times 50 \times 49$. The

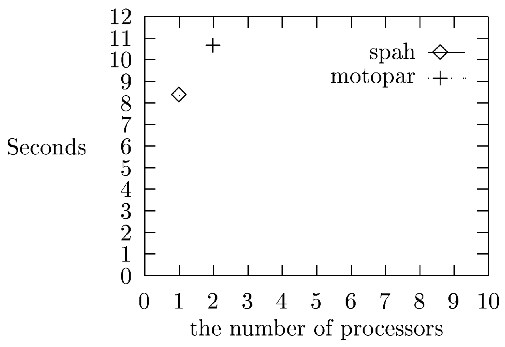

Fig. 14. Performance of *SPAH* and *MOTOpar*.

level 2 subgraph tree ST consists of 64 level 1, 16 level 2, four level 3, and one level 4 subgraphs. Based on the above level 4 subgraph tree, we create a level 1 HiTi graph which will be used throughout this section. To show performance comparison between *MOTOpar* and *SPAH*, we create five level 3 HiTi graphs where $3 \leq |\delta_i^1| \leq 8$ and where the edge cost is generated based on a uniform distribution [100,120] with five different seeds. Note that $|\delta_i^1|$ is the total number of boundary nodes defined on level 1 subgraph SG_i^1.

Fig. 14 shows that *SPAH* performs better than *MOTOpar*, which is a different result from the one given in [31]. The results can be explained by the following two reasons. The first reason is that the *HiTi* graph structure significantly reduces the advantage of using the two tree expansion approach given in [31]. In other words, the most of the nodes in the two level 1 subgraphs (i.e., where source and destination nodes are in) are already explored when two trees start to include the nodes in a *HiTi* graph. The second reason is that the lower bound estimation of *MOTOpar* is Euclidean distance which provides a lot tighter bound than Manhattan distance of *OTOpar* in [31]. As a result, compared with *OTOpar*, *MOTOpar* takes much longer time to stop building the two trees before it finds the shortest path.

Since *SPAH* outperforms *MOTOpar*, we use *SPAH* as the unit operation for parallelizing *inter* query shortest paths computation. In other words, *SPAH* performs the computation of each shortest path in parallel. The detail description of the *inter* query parallel shortest path algorithm, named Inter SPAH (*ISPAH*), is shown in Fig. 15.

Algorithm *ISPAH* executes *SPAH* in parallel on a BBN GP1000 shared memory multiprocessor system. For *SPAH* running on each processor, it accesses both local memory and globally shared memory for the computation. Algorithm *SPAH* accesses the globally shared memory only when it needs to access $G(V,E)$ or level k *HiTi* graph $H^k(V^k, E^k)$. Other than that, *SPAH* accesses the local memory. Note that there is no memory access contention for reading or writing a local memory. In the following section, we analyze the performance of *ISPAH*.

8.2 Performance Evaluation of ISPAH

We implemented *ISPAH* on a BBN GP1000 shared memory multiprocessor system. In our experiment, we used the same grid graph as described in Section 8.1. Based on these data sets, we computed 43 randomly prefixed shortest paths. The performance is then measured in terms of the average *speedup* T_n, where T_n represents the total time taken by n processors to compute M shortest paths. We express the *speedup* S_n of n processors as T_1/T_n.

Fig. 16 shows S_n as n is increased from 1 to 43. As we can see from Fig. 16, the speed up of *ISPAH* increases almost linearly up to 10 processors, after 10 it is beginning to level off, and after 25 processors, the performance begins to deteriorate. This occurs because of the globally shared memory access contentions. In *ISPAH*, all processors have to access a single *HiTi* graph in a globally shared memory, which causes a severe memory access contention when more than 25 processors are used.

In order to verify our conjecture of this memory access contention, we modified ISPAH so that an entire level k *HiTi* graph $H^k(V^k, E^k)$ is replicated on the local memory of each processor. This revised *ISPAH* is named a Modified ISPAH (MISPAH).

We analyzed *MISPAH* the same way as we did for *ISPAH*. Fig. 17 shows the *speedup* of *MISPAH* up to 43 processors showing the advantage of replicating a level k *HiTi* graph on each processor. Although *MISPAH* performs better than *ISPAH*, it is scalable up to 41 processors. From this analysis, we conclude that the parallel processing for *inter* query SPSP (Single Pair Shortest Path) problems is much more promising than *intra* query SPSP problems.

9 Conclusion

In this paper, we developed a new graph model, called a *HiTi* graph model, for very large topographical road maps. *HiTi* graphs provide a powerful formal framework for structuring topographical road map data in a hierarchical fashion. We first showed that our proposed shortest path algorithm *SPAH*, based on *HiTi* graph model, significantly reduces the search space for computing the minimum cost

```
begin
/* Assume we have a level $k$ $HiTi$ graph defined on a level $k+1$ $T$; */
/* $\{SG_1^1, SG_2^1, \ldots, SG_{n_1}^1\}$ and $H^k(V^k, E^k)$ are stored as adjacency lists in the globally shared memory; */
/* All the edges in $SG_t^1$ for $1 \leq t \leq n_1$ are assumed to be level 0 edges; */

  Let $R$ contain a set of source and destination nodes pairs (START,DEST);
  for each $(START, DEST) \in R$ do in parallel {
    Perform $SPAH$ for $(START, DEST)$; }
    /* $SPAH$ is given in Figure 5 in section 4.2 */
    /* We maintain separate marking places for each $(START, DEST)$ computation in Step 1 of $SPAH$ in $ISPAH$ */
end
```

Fig. 15. ISPAH algorithm.

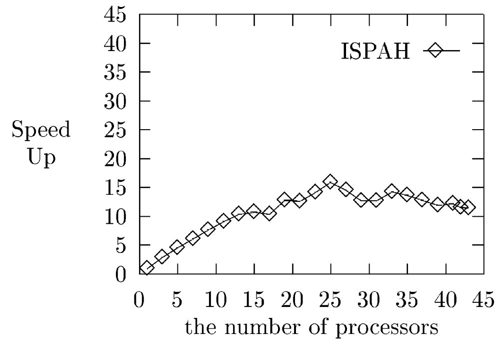

Fig. 16. Performance of ISPAH.

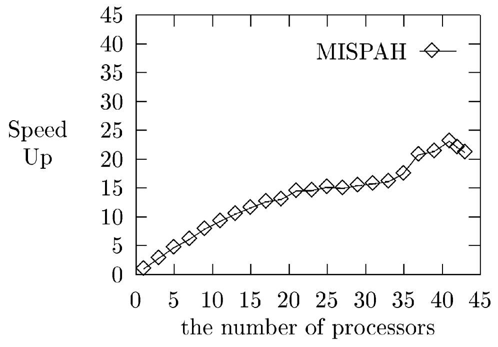

Fig. 17. Performance of MISPAH.

path over a very large topographical road map. We formally prove that the shortest path computed by $SPAH$ is optimal. The memory requirement of the $HiTi$ graph, and the efficient $HiTi$ graph updating algorithm, essential for many navigation systems to reflect dynamic traffic condition, are also investigated. Last, the $HiTi$ graph method is compared with other similar works such as the $HEPV$ and *Hub Indexing* methods, both theoretically and experimentally in terms of *update cost*, *storage overhead*, and *SPSP computation cost*.

We then studied parallel processing for *inter* query SPSP problem on topographical road maps. We show that the speedup of the parallel version of $SPAH$ increases almost linearly for up to 10 processors on a shared memory multiprocessor system. However, this algorithm is scalable for up to 25 processors. We then presented an improved version of this algorithm which reduces the memory access conflicts through partial data replication. This algorithm is scalable for up to 41 processors.

Appendix

Theorem 4.1. *Let SG_i^l be a level l subgraph node in a level $k+1$ subgraph tree T, where $k \geq 1$ and $1 \leq l \leq k+1$. Let $C_T(SG_i^l) = \{SG_1^{l-1}, SG_2^{l-1}, \dots, SG_p^{l-1}\}$. For any pair of boundary nodes x, $y \in \Delta(C_T(SG_i^l))$, we have*

$$SPC_{SG_i^l}(x,y) = SPC_{\Omega(C_T(SG_i^l))}(x,y).$$

Proof. Let $P_{SG_i^l}(x,y)$ represent a path from the boundary nodes x to y. Then, for any path $P_{SG_i^l}(x,y)$, all the boundary nodes on it as well as the two end nodes can be represented sequentially by the node sequence, $x, z_1, z_2, \dots, z_m, y$. Hence, its path cost can be represented by

$$\begin{aligned} PC_{SG_i^l}(x,y) =& PC_{SG_i^l}(x,z_1) + PC_{SG_i^l}(z_1,z_2) \\ &+ \cdots + PC_{SG_i^l}(z_m,y). \end{aligned}$$

By the principle of optimality, the shortest path cost $SPC_{SG_i^l}(x,y)$ can be denoted by:

$$\begin{aligned} SPC_{SG_i^l}(x,y) =& SPC_{SG_i^l}(x,z_1) + SPC_{SG_i^l}(z_1,z_2) \\ &+ \cdots + SPC_{SG_i^l}(z_m,y). \end{aligned}$$

In the boundary node sequence, every two successive boundary nodes correspond to either the boundary node entering a subgraph in $C_T(SG_i^l)$ and the boundary node leaving that subgraph or vice versa. Therefore, every successive pair of nodes in the boundary node sequence $x, z_1, z_2, \dots, z_m, y$ belong to the same set of level $l-1$ *semicut* and *path view* edge sets, say $(x, z_1) \in \Omega(SG_{j_1}^{l-1})$;

$$(z_1, z_2) \in \Omega(SG_{j_2}^{l-1}); \cdots ; (z_m, y) \in \Omega(SG_{j_m}^{l-1})$$

with $1 \leq j_1, j_2, \cdots, j_m \leq p$. As the shortest path cost $SP_{SG_i^l}(x,z_1)$ can be obtained from $\Omega(SG_{j_1}^{l-1})$, we would have $SPC_{SG_i^l}(x,z_1) = SPC_{\Omega(SG_{j_1}^{l-1})}(x,z_1)$. For a similar reason, we have $SPC_{SG_i^l}(z_1,z_2) = SPC_{\Omega(SG_{j_2}^{l-1})}(z_1,z_2)$, and so on, until $SPC_{SG_i^l}(z_m,y) = SPC_{\Omega(SG_{j_m}^{l-1})}(z_m,y)$. Thus, the shortest path cost $SPC_{SG_i^l}(x,y)$ can be represented by:

$$\begin{aligned} SPC_{SG_i^l}(x,y) =& SPC_{\Omega(SG_{j_1}^{l-1})}(x,z_1) + SPC_{\Omega(SG_{j_2}^{l-1})}(z_1,z_2) \\ &+ \cdots + SPC_{\Omega(SG_{j_m}^{l-1})}(z_m,y). \end{aligned}$$

From this, we know that the shortest path cost $SPC_{SG_i^l}(x,y)$ can be computed by using only the edges in $\Omega(C_T(SG_i^l))$. Thus, we have proven the theorem that $SPC_{SG_i^l}(x,y) = SPC_{\Omega(C_T(SG_i^l))}(x,y)$. □

Theorem 4.2. *Let SG_i^l be a level l subgraph node in a level $k+1$ subgraph tree T, where $k \geq 1$ and $1 \leq l \leq k+1$. Let $S_C(SG_i^l) = \{SG_1^{l-1}, SG_2^{l-1}, \dots, SG_p^{l-1}\}$. For any node pair x, $y \in SG_q^{l-1}$, where $1 \leq q \leq p$, the following holds: $SPC_{SG_i^l}(x,y) = SPC_{SG_q^{l-1} \cup \Omega(C_T(SG_i^l))}(x,y)$.*

Proof. All paths from x to y can be classified into two categories:

Case 1. The path consists solely of edges from SG_q^{l-1}.

Case 2. The path consists of edges not only from SG_q^{l-1} but from other level $l-1$ subgraphs in $C_T(SG_i^l)$.

For paths of Case 1, the path cost from x to y is represented by $PC_{SG_q^{l-1}}(x,y)$. If all paths from x to y are of this kind, then we know that the shortest path from x to y is also defined in SG_q^{l-1}. That is,

$$SPC_{SG_i^l}(x,y) = SPC_{SG_q^{l-1}}(x,y).$$

Any path of Case 2 consists of edges of SG_i^l which can be represented by the node sequence $x, z_1, z_2, \dots, z_m, y$. As this path contains edges of other level $l-1$ subgraphs $C_T(SG_i^l)$, it would leave SG_q^{l-1} and eventually come back. Therefore, the path contains at least two boundary nodes a and b, such that $a, b \in \Delta(SG_q^{l-1})$ and $a \neq b$. Choose a to be the first such node and b to be the last such node on this path. The path cost of this particular path can be denoted by:

$$PC_{SG_i^l}(x,y) = PC_{SG_i^l}(x,a) + PC_{SG_i^l}(a,b) + PC_{SG_i^l}(b,y).$$

Assume the node sequence of the shortest path is $x, \cdots, a, \cdots, b, \cdots, y$. By the principle of optimality, we have

$$SPC_{SG_i^l}(x,y) = SPC_{SG_i^l}(x,a) + SPC_{SG_i^l}(a,b) + SPC_{SG_i^l}(b,y).$$

As the shortest path $SP_{SG_i^l}(x,a)$ consists only of edges of SG_q^{l-1}, it falls into Case 1. Thus, we have $SPC_{SG_i^l}(x,a) = SPC_{SG_q^{l-1}}(x,a)$. Similarily, we have

$$SPC_{SG_i^l}(b,y) = SPC_{SG_q^{l-1}}(b,y).$$

As for $SPC_{SG_i^l}(a,b)$, by Theorem 4.1, we have $SPC_{SG_i^l}(a,b) = SPC_{\Omega(S_C(SG_i^l))}(a,b)$. Therefore, it follows that the shortest path cost from x to y in the context of SG_i^l is given by:

$$\begin{aligned} SPC_{SG_i^l}(x,y) = SPC_{SG_q^{l-1}}(x,a) + SPC_{\Omega(C_T(SG_i^l))}(a,b) \\ + SPC_{SG_q^{l-1}}(b,y). \end{aligned}$$

From this equation, we know that the shortest path $SP_{SG_i^l}(x,y)$ is in the following path set:

$$\begin{aligned} \{SP_{SG_q^{l-1}}(x,a) + SP_{\Omega(C_T(SG_i^l))}(a,b) + \\ SP_{SG_q^{l-1}}(b,y) \mid a,b \in \Delta(SG_q^{l-1}) \wedge a \neq b\}. \end{aligned}$$

Thus, the shortest path cost $SPC_{SG_i^l}(x,y)$ can be denoted by:

$$\begin{aligned} SPC_{SG_i^l}(x,y) = \min(\{SPC_{SG_q^{l-1}}(x,a) + SPC_{\Omega(C_T(SG_i^l))}(a,b) + \\ SPC_{SG_q^{l-1}}(b,y) \mid a,b \in \Delta(SG_q^{l-1}) \wedge a \neq b\}). \end{aligned}$$

By combining Cases 1 and 2, we have

$$\begin{aligned} SPC_{SG_i^l}(x,y) = &\min(SPC_{SG_q^{l-1}}(x,y), \min(\{SPC_{SG_q^{l-1}}(x,a) \\ &+ SPC_{\Omega(C_T(SG_i^l))}(a,b) \\ &+ SPC_{SG_q^{l-1}}(b,y) \mid a,b \in \Delta(SG_q^{l-1}) \wedge a \neq b\})), \end{aligned}$$

which, in turn, proves that

$$SPC_{SG_i^l}(x,y) = SPC_{SG_q^{l-1} \cup \Omega(C_T(SG_i^l))}(x,y).$$

□

Theorem 4.3. *Let SG_i^l be a level l subgraph node in a level $k+1$ subgraph tree T, where $k \geq 1$ and $1 \leq l \leq k+1$. Let $C_T(SG_i^l) = \{SG_1^{l-1}, SG_2^{l-1}, \ldots, SG_p^{l-1}\}$. For any node pair $x \in SG_u^{l-1}$, $y \in SG_v^{l-1}$, where $1 \leq u, v \leq p$ and $u \neq v$, then $SPC_{SG_i^l}(x,y) = SPC_{SG_u^{l-1} \cup SG_v^{l-1} \cup \Omega(C_T(SG_i^l))}(x,y)$.*

Proof. Any path from x to y in SG_i^l can be represented by the node sequence $x, z_1, z_2, \ldots, z_m, y$. It is obvious that two nodes a and b must exist in this path where $a \in \Delta(SG_u^{l-1})$ and $b \in \Delta(SG_v^{l-1})$ since the path must ultimately leave SG_u^{l-1} and enter SG_v^{l-1}. Choose a to be the first such node and b to be the last such node on this path. The path cost of SG_i^l can be denoted by:

$$PC_{SG_i^l}(x,y) = PC_{SG_i^l}(x,a) + PC_{SG_i^l}(a,b) + PC_{SG_i^l}(b,y).$$

Assume the node sequence of the shortest path from x to y in SG_i^l is $x, \cdots, a, \cdots, b, \cdots, y$. By the principle of optimality, we have

$$SPC_{SG_i^l}(x,y) = SPC_{SG_i^l}(x,a) + SPC_{SG_i^l}(a,b) + SPC_{SG_i^l}(b,y).$$

As the shortest path $SP_{SG_i^l}(x,a)$ consists only of edges from SG_u^{l-1}, and the shortest path $SP_{SG_i^l}(b,y)$ consists only of edges from SG_v^{l-1}, we have $SPC_{SG_i^l}(x,a) = SPC_{SG_u^{l-1}}(x,a)$ and $SPC_{SG_i^l}(b,y) = SPC_{SG_v^{l-1}}(b,y)$. As for $SPC_{SG_i^l}(a,b)$, by Theorem 4.1, we have

$$SPC_{SG_i^l}(a,b) = SPC_{\Omega(C_T(SG_i^l))}(a,b).$$

Thus, the shortest path cost $SPC_{SG_i^l}(x,y)$ can be represented by

$$\begin{aligned} SPC_{SG_i^l}(x,y) = SPC_{SG_u^{l-1}}(x,a) + SPC_{\Omega(C_T(SG_i^l))}(a,b) \\ + SPC_{SG_v^{l-1}}(b,y) \end{aligned}$$

From the above equation, we know that the shortest path $SP_{SG_i^l}(x,y)$ is in the following path set:

$$\begin{aligned} \{SP_{SG_u^{l-1}}(x,a) + SP_{\Delta(C_T(SG_i^l))}(a,b) \\ + SP_{SG_v^{l-1}}(b,y) \mid a \in \Delta(SG_u^{l-1}) \\ \wedge b \in \Delta(SG_v^{l-1})\}. \end{aligned}$$

Accordingly, the shortest path cost $SPC_{SG_i^l}(x,y)$ can be denoted by:

$$\begin{aligned} SPC_{SG_i^l}(x,y) = \min(\{SPC_{SG_u^{l-1}}(x,a) + SPC_{\Omega(C_T(SG_i^l))}(a,b) \\ + SPC_{SG_v^{l-1}}(b,y) \mid a \in \Delta(SG_u^{l-1}) \\ \wedge b \in \Delta(SG_v^{l-1})\}) \end{aligned}$$

which, in turn, proves

$$SPC_{SG_i^l}(x,y) = SPC_{SG_u^{l-1} \cup SG_v^{l-1} \cup \Omega(C_T(SG_i^l))}(x,y).$$

□

Theorem 4.4. *Let a level $k+1$ subgraph tree T be constructed from $G(V,E)$, where $k \geq 1$ and T defines a level k HiTi graph $H^k(V^k, E^k)$. For any node pair $x \in SG_i^1$ and $y \in SG_j^1$, we have $SPC_G(x,y) = SPC_{SG_i^1 \cup SG_j^1 \cup D}(x,y)$ such that $D = \Omega(C_T(A_T(\{SG_i^1, SG_j^1\})))$.*

Proof. Since a level $k+1$ T has SG_1^{k+1} as the root node, we have $G = SG_1^{k+1}$ which gives

$$SPC_G(x,y) = SPC_{SG_1^{k+1}}(x,y).$$

Let $l = lca(SG_i^1, SG_j^1)$. Then, for $l < t \leq k+1$, $x, y \in A_T^t(SG_i^1)$ since $A_T^t(SG_i^1)$ is the same as $A_T^t(SG_j^1)$. By the Theorem 4.2, we have

$$SPC_G(x,y) = SPC_{A_T^k(SG_i^1) \cup \Omega(C_T(A_T^{k+1}(SG_i^1)))}(x,y). \qquad (1)$$

From (1), we can reduce the search space $A_T^k(SG_i^1)$ by recursively applying Theorem 4.2, while $x, y \in A_T^t(SG_i^1)$ such that $l < t \leq k+1$. As a result, we have

$$SPC_G(x,y) = SPC_{\cup_{t=l+1}^{t=k+1}\Omega(C_T(A_T^t(SG_i^1)))\cup A_T^l(SG_i^1)}(x,y). \quad (2)$$

Now, the reduced search space for (2) is defined as

$$\cup_{t=l+1}^{t=k+1} \Omega(C_T(A_T^t(SG_i^1))) \cup A_T^l(SG_i^1) \quad (3)$$

Then, the shortest path computed on (3) is the same as the one computed on G. From $A_T^l(SG_i^1)$ in (3), we know that $C_T(A_T^l(SG_i^1))$ includes $A_T^{l-1}(SG_i^1)$ and $A_T^{l-1}(SG_j^1)$, where $A_T^{l-1}(SG_i^1) \neq A_T^{l-1}(SG_j^1)$. Since $x \in A_T^{l-1}(SG_i^1)$ and $y \in A_T^{l-1}(SG_j^1)$, we rewrite (3) by applying Theorem 4.3 as

$$\begin{aligned}&\cup_{t=l+1}^{t=k+1} \Omega(C_T(A_T^t(SG_i^1))) \cup A_T^{l-1}(SG_i^1)\\ &\cup A_T^{l-1}(SG_j^1) \cup \Omega(C_T(A_T^l(SG_i^1))).\end{aligned} \quad (4)$$

For $A_T^{l-1}(SG_i^1)$ in (4), we need to compute the shortest paths from node x to the boundary nodes in $\Delta(A_T^{l-2}(SG_i^1))$, and the shortest paths from the boundary nodes in $\Delta(A_T^{l-2}(SG_i^1))$ to the boundary nodes in $\Delta(C_T(A_T^{l-l}(SG_i^1)))$. For $A_T^{l-1}(SG_j^1)$ in (4), we need to compute the shortest paths from node y to the boundary nodes in $\Delta(A_T^{l-2}(SG_j^1))$, and the shortest paths from the boundary nodes in $\Delta(A_T^{l-2}(SG_j^1))$ to the boundary nodes in $\Delta(C_T(A_T^{l-l}(SG_j^1)))$. Then, by using Theorem 4.1, we can rewrite (4) as

$$\begin{aligned}&\cup_{t=l+1}^{t=k+1} \Omega(C_T(A_T^t(SG_i^1))) \cup A_T^{l-2}(SG_i^1) \cup A_T^{l-2}(SG_j^1) \cup\\ &\cup_{t=l-1}^{t=l}; \Omega(C_T(A_T^t(SG_i^1, SG_j^1))).\end{aligned} \quad (5)$$

Note that $A_T^t(SG_i^1, SG_j^1)$ is the same as $A_T^t(SG_i^1)$ or $A_T^t(SG_j^1)$ for $l < t \leq k+1$. Similarly, we recursively apply Theorem 4.1 to (5). Then, we have

$$\begin{aligned}&\cup_{t=l+1}^{t=k+1} \Omega(C_T(A_T^t(SG_i^1))) \cup A_T^1(SG_i^1) \cup A_T^1(SG_j^1) \cup\\ &\cup_{t=1}^{t=l} \Omega(C_T(A_T^t(SG_i^1, SG_j^1))).\end{aligned} \quad (6)$$

We simplify (6) and get the reduced search space for computing the shortest path cost from the nodes x to y on G as follows:

$$\cup_{t=1}^{t=k+1} \Omega(C_T(A_T^t(SG_i^1, SG_j^1))) \cup SG_i^1 \cup SG_j^1 \quad (7)$$

From (6) and $\cup_{t=1}^{t=k+1} A_T^t(SG_i^1, SG_j^1) = A_T(SG_i^1, SG_j^1)$, we prove that $SPC_G(x,y) = SPC_{SG_i^1 \cup SG_j^1 \cup D}(x,y)$ such that $D = \Omega(C_T(A_T(\{SG_i^1, SG_j^1\})))$. □

Acknowledgments

The authors would like to thank the reviewers for their many insightful and helpful comments.

References

[1] R. Agrawal and H. Jagadish, "Materialization and Incremental Update of Path Information," *Proc. IEEE Fifth Int'l Conf. Data Eng.,* pp. 374-383, 1989.

[2] R. Agrawal, S. Dar, and H. Jagadish, "Direct Transitive Closure Algorithms: Design and Performance Evaluation," *ACM Trans. Database Systems,* vol. 15, no. 3, pp. 427-458, Sept. 1990.

[3] R. Agrawal and J. Kiernan, "An Access Structure for Generalized Transitive Closure Queries," *Proc. IEEE Ninth Int'l Conf. Data Eng.,* pp. 429-438, Apr. 1993.

[4] R. Agrawal and H. Jagadish, "Algorithms for Searching Massive Graphs," *IEEE Trans. Knowledge and Data Eng.,* vol. 6, no. 2, pp. 225-238, Apr. 1994.

[5] G. Ausiello, G. Italiano, A. Marchetti, and U. Nanni, "Incremental Algorithms for Minimal Length Paths," *Proc. First Ann. ACM-SIAM Symp. Discrete Algorithms,* pp. 12-20, 1990.

[6] J. Banerjee, S. Kim, W. Kim, and J. Garza, "Clustering a DAG for CAD Databases," *IEEE Trans. Software Eng.,* vol. 14, no. 11, pp. 1684-1699, 1998.

[7] E. Charniak and D. McDermott, *Introduction to Artificial Intelligence.* Addison-Wesley, 1985.

[8] S. Dar and R. Ramakirishnan, "A Performance Study of Transitive Closure Algorithms," *Proc. ACM-SIGMOD 1994 Int'l Conf. Management of Data,* pp. 454-465, 1994.

[9] T. Dean, J. Firby, and D. Miller, "Hierarchical Planning Involving Deadlines, Travel Time, and Resources," *Computational Intelligence,* vol. 4, no. 4, pp. 381-398, 1988.

[10] D. Galperin, "On the Optimality of A*," *Artificial Intelligence,* vol. 8, no. 1, pp. 69-76, 1977.

[11] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Gracia-Molina, "Proximity Search in Databases," *Proc. 24th VLDB Conf.,* pp. 26-37, 1998.

[12] M. Houstma, P. Apers, and G. Schipper, "Data Fragmentation for Parallel Transitive Closure Strategies," *Proc. IEEE Ninth Int'l Conf. Data Eng.,* pp. 447-456, 1993.

[13] K. Hua, J. Su, and C. Hua, "Efficient Evaluation of Traversal Recursive Queries Using Connectivity Index," *Proc. IEEE Ninth Int'l Conf. Data Eng.,* pp. 549-558, 1993.

[14] Y. Huang, N. Jing, and E. Rundensteiner, "Hierarchical Path Views: A Model Based on Fragmentation and Transportation Road Types," *Proc. Third ACM Workshop Geographic Information Systems,* pp. 93-100, 1995.

[15] Y. Huang, N. Jing, and E. Rundensteiner, "Effective Graph Clustering for Path Queries in Digital Map Databases," *Proc. Fifth Int'l Conf. Information and Knowledge Management,* pp. 215-222, 1996.

[16] Y. Ioannidis and R. Ramakirishnan, "Efficient Transitive Closure Algorithms," *Proc. 14th VLDB Conf.,* pp. 382-394, 1988.

[17] Y. Ioannidis, R. Ramakirishnan, and L. Winger, "Transitive Closure Algorithms Based on Graph Traversal," *ACM Trans. Database Systems,* vol. 18, no. 3, pp. 513-576, Sept. 1993.

[18] K. Ishikawa, M. Ogawa, S. Azume, and T. Ito, "Map Navigation Software of the Electro Multivision of the '91 Toyota Soarer," *Proc. Int'l Conf. Vehicle Navigation and Information Systems (VNIS IVHS),* pp. 463-473, 1991.

[19] H. Jagadish, "A Compression Technique to Materialize Transitive Closure," *ACM Trans. Database Systems,* vol. 15, no. 4, pp. 558-598, Dec. 1990.

[20] B. Jiang, "A Suitable Algorithm for Computing Partial Transitive Closures in Databases" *Proc. IEEE Sixth Int'l Conf. Data Eng.,* pp. 264-271, 1990.

[21] N. Jing, Y. Huang, and E. Rundensteiner, "Hierarchical Optimization of Optimal Path Finding for Transportation Applications," *Proc. Fifth Int'l Conf. Information and Knowledge Management,* pp. 261-268, 1996.

[22] N. Jing, Y. Huang, and E. Rundensteiner, "Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its Performance Evaluation," *IEEE Trans. Knowledge and Data Eng.,* vol. 10, no. 3, pp. 409-432, May/June 1998.

[23] V. King and G. Sagert, "A Fully Dynamic Algorithm for Maintaining the Transitive Closure," *Proc. 31st Ann. ACM Symp. Theory of Computing,* pp. 492-498, 1999.

[24] R. Kung, E. Hanson, Y. Ioannnidis, T. Sellis, L. Shapiro, and M. Stonebraker, "Heuristic Search in Data Base System," *Proc. First Int'l Workshop Expert Database Systems,* pp. 96-107, Oct. 1984.

[25] P. Larson and V. Deshpande, "A File Structure Supporting Traversal Recursion," *Proc. ACM-SIGMOD 1989 Int'l Conf. Management of Data,* pp. 243-252, 1989.

[26] R. Lipton and R. Tarjan, "Application of a Planar Seperator Theorem," *SIAM J. Computing,* vol. 9, no. 3, pp. 615-627, 1980.

[27] B. Liu, S. Choo, S. Lok, S. Leong, S. Lee, F. Poon, and H. Tan, "Integrating Case-Based Reasoning, Knowledge-Based Approach and Dijkstra Algorithm for Route Finding" *Proc. 10th Conf. Artificial Intelligence for Applications (CAIA '94),* pp. 149-155, 1994.

[28] W. McCormick Jr., P. Schweitzer, and T. White, "Problem Decomposition and Data Reorganization by a Clustering Technique," *Operation Research,* vol. 20, no. 5, pp. 993-1009, 1972.

[29] G. Miller, S. Teng, and S. Vavasis, "A Unified Geometric Approach to Graph Separators," *Proc. 31st Ann. Symp. Foundations of Computer Science,* pp. 538-547, 1991.

[30] G. Miller, S. Teng, W. Thurston, and S. Vavasis, "Automatic Mesh Partitioning," *Sparse Matrix Computations: Graph Theory Issues and Algorithms,* A. George, J. Gilbert, and J. Liu, eds. (An IMA Workshop Volume), New York: Springer-Verlag, 1993.

[31] T. Mohr and C. Pasche, "A Parallel Shortest Path Algorithm," *Computing,* vol. 40, pp. 281-292, 1988.

[32] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving,* Reading, Mass.: Addison Wesley, 1984.

[33] G. Qadah, L. Henschen, and J. Kim, "Efficient Algorithms for the Instantiated Transitive Closure Queries," *IEEE Trans. Software Eng.,* vol. 17, no. 3, pp. 296-309, Mar. 1991.

[34] S. Rizzi, "Genetic Operators for Hierarchical Graph Clustering," *Pattern Recognition Letters,* vol. 19, no. 14, pp. 1293-1300, 1998.

[35] A. Rosenthal, S. Heiler, U. Dayal, and F. Manola, "Traversal Recursion: A Practical Approach to Supporting Recursive Applications," *Proc. IEEE Third Int'l Conf. Data Eng.,* pp. 580-590, 1987.

[36] E. Sacerdoti, "Planning in a Hierarchy of Abstraction Spaces," *Artificial Intelligence,* vol. 5, no. 2, pp. 115-135, 1974.

[37] J. Shapiro, J. Waxman, and D. Nir, "Level Graphs and Approximate Shortest Path Algorithms," *Networks,* vol. 22, pp. 691-717, 1992.

[38] S. Shekhar, A. Kohli, and M. Coyle, "Path Computation Algorithms for Advanced Traveler Information System (ATIS)," *Proc. IEEE Ninth Int'l Conf. Data Eng.,* pp. 31-39, 1993.

[39] S. Shekhar, A. Fetterer, and B. Goyal, "Materialization Trade-Offs in Hierarchical Shortest Path Algorithms," *Proc. 1997 Symp. Spatial Databases,* 1997.

[40] S. Shekhar and D. Liu, "CCAM: A Connectivity-Clustered Access Method for Networks and Network Computations," *IEEE Trans. Knowledge and Data Eng.,* vol. 9, no. 1, pp. 102-119, 1997.

[41] S. Timpf, G. Volta, D. Pollock, and M. Egenhofer, "A Conceptual Model of Wayfinding Using Multiple Levels of Abstraction," *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space,* A.U. Frank, I. Campari, and U. Formentini, eds., Springer-Verlag, 1992.

[42] I. Toroslu and G. Qadah, "The Efficient Computation of Strong Partial Transitive-Closures," *Proc. IEEE Ninth Int'l Conf. Data Eng.,* pp. 530-537, 1993.

[43] T. Yang, S. Shekhar, B. Hamidzadeh, and P. Hancock, "Path Planning and Evaluation in IVHS Databases" *IEEE Int'l Conf. Vehicle Navigation and Information Systems (VNIS IVHS),* pp. 283-290, 1991.

Sungwon Jung received the BS degree in computer science from Sogang Univeristy, Seoul, Korea, in 1988. He received the MS and PhD degrees in computer science from Michigan State University, East Lansing, Michigan, in 1990 and 1995, respectively. He is currently an assistant professor in the Computer Science Department at Sogang University. His research interests include mobile databases, parallel processing in database systems, spatial databases, GIS, and ITS. He is a member of the IEEE Computer Society and the ACM.

Sakti Pramanik received the BE degree from Calcutta University and the MS degree from the University of Alberta, Edmonton, both in electrical engineering, and the PhD degree in computer science from Yale University. He is currently a professor in the Department of Computer Science and Engineering at Michigan State University. His research interests include parallel, distributed, and multimedia databases. He has also done work in bioinformatics.