GeoInformatica 1:2, 125–159 (1997)

A Hierarchical Path View Model for Path Finding in Intelligent Transportation Systems

YUN-WU HUANG, NING JING* AND ELKE A. RUNDENSTEINER†
University of Michigan, ywh@eecs.umich.edu
Chagsha Institute of Technology, ningjing@pdns.nudt.edu.cn
Worcester Polytechnic Insitute, rundenst@cs.wpi.edu

Received July 2, 1996; Revised January 31, 1997; Accepted March 21, 1997

Abstract

Effective path finding has been identified as an important requirement for dynamic route guidance in Intelligent Transportation Systems (ITS). Path finding is most efficient if the all-pair (shortest) paths are precomputed because path search requires only simple lookups of the precomputed path views. Such an approach however incurs path view maintenance (computation and update) and storage costs which can be unrealistically high for large ITS networks. To lower these costs, we propose a Hierarchical Path View Model (HPVM) that partitions an ITS road map, and then creates a hierarchical structure based on the road type classification. HPVM includes a map partition algorithm for creating the hierarchy, path view maintenance algorithms, and a heuristic hierarchical path finding algorithm that searches paths by traversing the hierarchy. HPVM captures the dynamicity of traffic change patterns better than the ITS path finding systems that use the hierarchical A^* approach because: (1) during path search, HPVM traverses the hierarchy by dynamically selecting the connection points between two levels based on up-to-date traffic, and (2) HPVM can reroute the high-speed road traffic through local streets if needed. In this paper, we also present experimental results used to benchmark HPVM and to compare HPVM with alternative ITS path finding approaches, using both synthetic and real ITS maps that include a large Detroit map ($>$ 28,000 nodes). The results show that the HPVM incurs much lower costs in path view maintenance and storage than the non-hierarchical path precomputation approach, and is more efficient in path search than the traditional ITS path finding using A^* or hierarchical A^* algorithms.

Keywords: path queries, path search, digital map databases, intelligent transportation systems

1. Introduction

1.1. Path finding issues for ITS

Centralized path finding has been recognized as one of the potential solutions to *dynamic* route guidance[1] for road networks in Intelligent Transportation Systems (ITS). In vehicle-based path finding [8], each vehicle conducts its own path finding using on-board computers and static road maps in CD-ROMs. In contrast, the centralized path finding relies on central stations such as the Traffic Management Centers (TMCs) to answer path queries submitted by the vehicles (see figure 1). Compared to the vehicle-based approach, the centralized path finding demands a lower per-vehicle cost for route guidance

* This work was performed while the author was a visitor at the University of Michigan.
† This work was performed while the author was a faculty member of the University of Michigan.

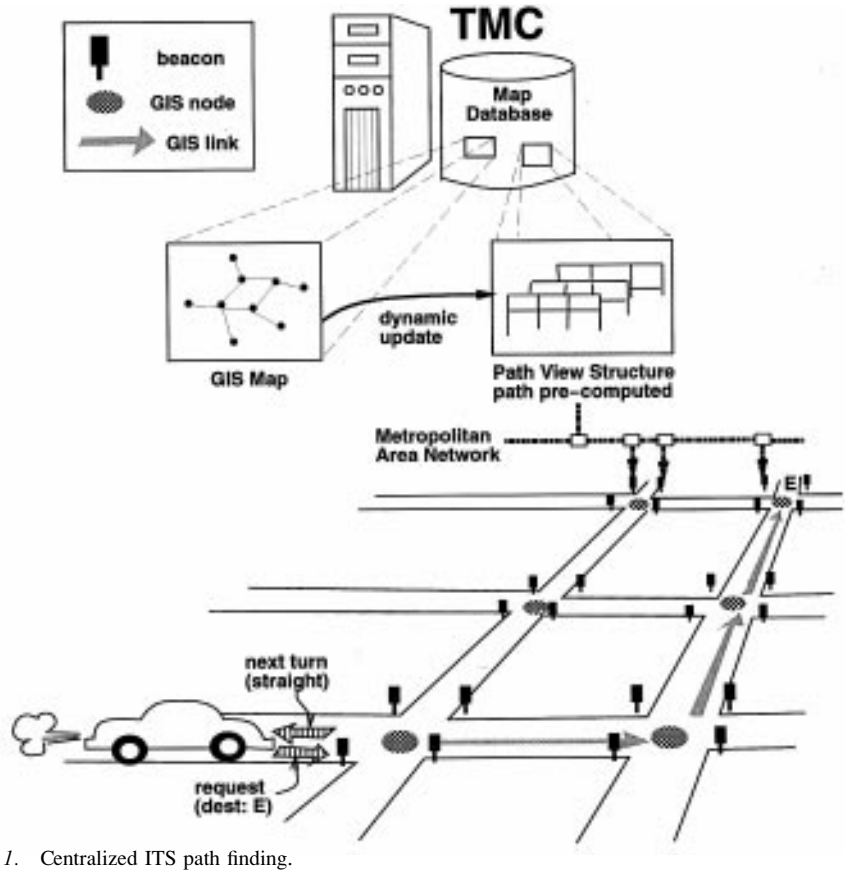


Figure 1. Centralized ITS path finding.

equipment in three ways. First, vehicles do not need computation devices because paths are computed by the TMCs. Second, vehicles do not need substantial storage in order to manage the map data which are typically large in size. Third, vehicles do not need to retrieve and analyze the up-to-date traffic information in order to monitor the changing traffic conditions across the network. Owing to the low cost of route guidance equipment for each vehicle, a greater market penetration of ITS route guidance is expected for centralized path finding.

In the centralized ITS architecture proposed in [20], the guided vehicles communicate with the TMCs through road-side beacons installed at every major intersection. A successful path query exchange means that a vehicle submits a path query to a beacon it encounters, and then receives the computed path information from the TMC through the same beacon where the query is submitted (see figure 1). The computed path information may consist only of the instruction that directs the guided vehicle where to turn next. A guided vehicle navigates a road network by continuously submitting path queries to the beacons it encounters, and following the next-turn instructions it receives, until the destination is reached.

Each path query exchange must be completed within a very short time. Otherwise, a vehicle may pass the beacon, to which the query was submitted, before the computed path information is returned. Furthermore, during one trip, a guided vehicle may issue many path queries—typically proportional to the number of beacons the vehicle encounters. If many guided vehicles are on the road, likely during the rush hours, the TMCs may receive altogether a large number of path queries within a short time. Therefore, a very stringent time constraint must be imposed on path query processing by the TMCs.

1.2. Problems of current approaches

Traditional ITS path finding solutions use variations of the heuristic A^* single-pair path search algorithm to compute paths [23], [25]. The A^* algorithm computes a shortest path by expanding links starting from the origin node until it reaches the destination node. During link expansion, it gives expansion priority to the link that leads to a node for which the actual expanded cost from the origin node to this node plus the estimated cost from this node to the destination node is the minimum among all unexpanded links. If the estimated cost between any given two nodes always under-estimates the real cost, the A^* algorithm guarantees finding an optimal shortest path.

Using the A^* algorithm to process path queries, a TMC needs to invoke one single-pair path search for each path query composed of a different Origin-Destination (O-D) pair. The potentially large number of path finding requests received by a TMC during rush hours may amount to a huge collection of computational tasks. As a result, the stringent constraint of the path query response time may not be satisfied. The ITS dynamic route guidance function may not operate properly.

Although incorporating a hierarchy into the A^* algorithm can improve path query response time [25], such an approach usually assumes a rigid hierarchical structure that may not be reorganized dynamically. Typically, each local node is associated with a fixed point which connects this local node to high-speed links. Such an approach does not allow for the selection of the best connecting points based on up-to-date traffic, neither for the rerouting of vehicles from high-speed links to local streets if the high-speed links are blocked by traffic.

An alternative solution to the centralized path finding is to precompute the best paths for all O-D pairs and store them on-line [13]. Upon receiving a path query request, the TMC needs only to look up the requested path from the precomputed path view structure. Therefore path queries can be processed very efficiently. However, to capture the dynamicity of changing traffic, each next-turn instruction received by the guided vehicles must be computed based on the most up-to-date traffic conditions. Therefore, very frequent updates of the precomputed path views is necessary for road networks where traffic conditions change continuously. Such a necessity dictates that updating a path view must be completed within a short time.[2]

Among the shortest path transitive closure algorithms in the literature [6], [24] the Dijkstra algorithm [6] is one of the preferred for ITS applications because it has a worst-case time complexity of $O(n^2 log(n))$ for ITS road networks.[3] Such a high computation

complexity means that computing the path view will take a long time for large networks such as the Detroit map ($>$ 28,000 nodes) and the Chicago map ($>$ 50,000 nodes). Consequently, the path view for large networks may not be up-to-date.

Furthermore, a path view modeled by the shortest path transitive closure has to capture at a minimum the aggregated path labels for all O-D pairs. This $n \times n$ adjacency matrix requires $O(n^2)$ storage where n is the number of nodes in the network. Therefore the storage cost of a path view may become unrealistically high for large networks.

1.3. The hierarchical path view solution

This paper presents a Hierarchical Path View Model (HPVM) that satisfies the query response time constraint, and, for large networks, requires realistic storage to maintain the precomputed path views which can be updated frequently. The structure of the hierarchy in HPVM is based on the following ITS road network characteristics: 1) high-speed roads usually interconnect different regions of a road network, and 2) road links are mostly short relative to the entire map, and (3) road links are strongly interconnected which means nearby nodes are mutually reachable in a few hops. The effectiveness of HPVM is based on the heuristic that high-speed links are preferred for inter-regional traveling. In other words, the longer the distance traveled the more coarse-grained the effective map should be, e.g., from country, to state, to city map, and so on. In HPVM, each granularity of the network represents a different level in the hierarchy, which is organized in a bottom-up fashion. First, the fine-grained original network (ground level) is divided into multiple regions. Next, links of higher speed are selected to form a graph of coarser granularity at the next higher level in the hierarchy. For large networks, the high-level graph is further divided into multiple regions to form an additional level, and so on.

After the creation of the hierarchy, the path view for each region at all levels in the hierarchy is then precomputed and stored. Because traffic conditions may change across the network continuously, path views should be frequently examined to see if they are affected by the recent changes. The affected path views are then updated to the latest traffic information. We use the term ''path view maintenance'' to represent both path view precomputation and update. To process a path query, the TMCs conduct path search by traversing the hierarchy based on path information materialized in the regional path views. In summary, HPVM achieves ITS dynamic route guidance by performing the following four tasks:

1. **Hierarchy Generation.** HPVM creates (or re-creates) the hierarchy only when the topology of the underlying network is altered. Such events are rare, therefore, hierarchy generation is a static process (Section 3).
2. **Path View Precomputation.** This paper presents a new algorithm, called Two Color Dijkstra (TCD), that computes a path view for an ITS road network (a full network or a fragmented region). The TCD algorithm takes advantage of the uniformly low out-degree of the ITS road network by incorporating a 2-color graph painting technique into the Dijkstra algorithm (Section 2).

3. **Path View Incremental Update.** To keep the path views up-to-date, HPVM updates each path view periodically to account for changes of traffic conditions of its underlying regional network. It is possible that within a short time interval, only a small fraction of road links are affected for a regional network. To exploit this situation, this paper presents two incremental update algorithms, called IUA_Decrease and IUA_Increase. The two incremental update algorithms can update a path view faster than recomputing it using the TCD algorithm if traffic condition for only a small number of road links has changed. Based on the number of links affected since the last update, HPVM can choose the appropriate algorithms, the TCD or the ICU algorithms, to perform the path view update (Section 2).
4. **Path Search.** In HPVM, a path query is processed by executing a single-pair path search algorithm, called Heuristic Hierarchical Path Search, or short, HHPS. The HHPS searches intra-regional paths by looking up the path view for this region. It searches the inter-regional paths by traversing the hierarchy in HPVM (Section 4).

This paper differs in many respects from its preliminary conference version [14]. First, we present a new and more effective graph fragmentation algorithm that divides a network into multiple regions in this paper (instead of assuming graph partitioning is done by hand). Second, we now propose incremental update algorithms that were not reported previously. Third, the HHPS algorithm presents in this paper is more efficient than the path search algorithm presented previously. This paper also presents formal definitions of the HPVM model, as well as extensive sets of experiments evaluating the HPVM that were not available in previous reports.

The paper is organized as follows. Section 2 gives the path view maintenance algorithms. Section 3 outlines the hierarchical path view model. Section 4 gives the hierarchical path search algorithm whereas the experimental results are presented in Section 5. We discuss the related work in Section 6, and conclude in Section 7.

2. Path view maintenance algorithms

In this section, we introduce some basic notations and definitions. We then present the TCD algorithm followed by the incremental update algorithms.

2.1. Basic notations

An ITS road map can be modeled as a labeled directed graph $G = (N, L, W)$. A node in N corresponds to a map object (intersection) in the road map; a link in L corresponds to a one-directional connection between two neighboring nodes in N, and a label in W corresponds to the traversal cost (e.g., estimated link travel time, link distance, etc) for each link.

Definition 1: An ITS network is a graph $G = (N, L, W)$, where
$N = \{i | 0 \leq i < n\}$ is the set of nodes, where n is the number of nodes, and
$L = \{ < i, j, w > | i, j \in N, i \neq j, w \in W \}$ is the set of weighted links, and
W is the set of all possible link traversal costs.

We require $i \neq j$ because we disallow self-loops. We say i is the source node of link $< i, j, w >$ and j is the destination node. We say the link traversal cost w of $< i, j, w >$ is the link weight of the link from i to j, denoted by LW_{ij} in this paper.

A path in G is an ordered sequence of nodes. The weight of a path is the sum of the weight associated with each link along the path. We define PV^G to be the path view of G below.

Definition 2: PV^G is a $n \times n$ matrix where $n = |N|$, N in G. Each element PV_{ij}^G in PV^G is the weight of the shortest path from i to j, where $i, j \in N$ of G.

2.2. *The* TCD *algorithm—creating a path view*

The TCD algorithm integrates the Dijkstra shortest path transitive algorithm with a 2-color graph painting technique to reduce the path view computation time for an ITS network. For sparse graphs, such as the ITS networks, the Dijkstra algorithm [6] is preferable because it has a worst-case time complexity of $O(n^2 \times log(n))$, where n is the number of nodes in the network, as compared to the $O(n^3)$ complexity of other algorithms such as Warshall's [24], Depth-first search and Breadth-first search.

The TCD algorithm first assigns each node in the ITS graph one of the two colors, GREEN and RED, such that every child (direct descendent node) of a GREEN node is a RED node. We then run the single-source Dijkstra shortest path algorithm for RED nodes only. When this process is complete, the single-source transitive closure for a GREEN node can be easily derived because the single-source transitive closures for all its children are computed (Appendix A).

ITS graphs typically resemble grid patterns in that they are sparse with uniformly low out-degree for each node. For perfect grid patterns, the coloring process will result in equal numbers for GREEN ad RED nodes. For ITS graphs, the coloring process will likely result in a significant number of GREEN nodes.[4] For ITS graphs, the time complexity of computing the transitive closure is $O(n \times \log(n))$ for a RED node, and $O(n)$ for a GREEN node. Since the computation complexity for a GREEN node is lower than that for a RED node, and the number of GREEN nodes is significant, TCD can improve computation time over the Dijkstra algorithm.

We now introduce the definition of a colored ITS network (see the example in figure 2).

Definition 3: A colored ITS network is a graph $G^c = (N, L, W, GREEN, RED)$, such that
$N = \{i | 0 \leq i < n\}$ is the set of nodes, where n is the number of nodes, and
$L = < i, j, w > | i, j \in N, i \neq j, w, \in W\}$ is the set of weighted links, and W is the set of all possible link traversal costs, and

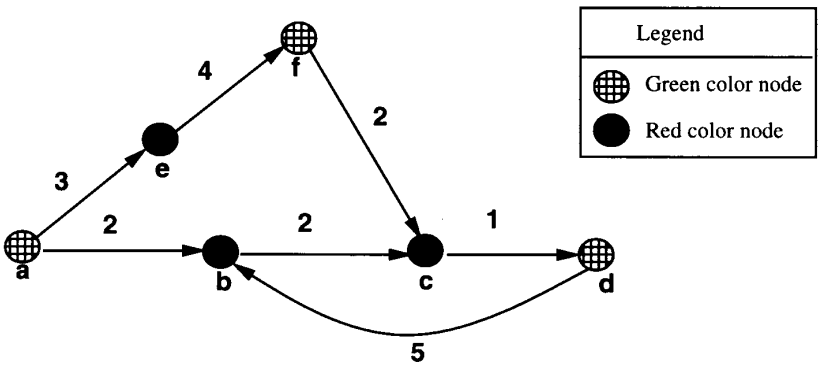


Figure 2. A sample two-color painted graph.

$GREEN, RED \subseteq N$, and $GREEN \cap RED = \phi$, and $GREEN \cup RED = N$, and all children of a node in *GREEN* are nodes in *RED*.

Given an ITS graph G, G^c is created by the 2ColorPaint algorithm (Appendix A). This algorithm paints each node with a color of either GREEN and RED while guaranteeing that each of the outgoing links of a GREEN node connects to a RED node.

The TCD algorithm (Appendix B) computes the shortest path transitive closure, called path view, for a colored ITS network G^c. The color sets GREEN and RED in G^c are first created by the 2ColorPaint algorithm. Then, for each RED set, the TCD algorithm calls the DijkstraSingleSourceAlgorithm which is the well-known Dijkstra algorithm [3] for computing its single-source shortest path transitive closure. For each GREEN node, the TCD algorithm derives its single-source path view by adding its outgoing link weights to the single-source path views of its children nodes and by choosing a shortest path for each destination.

Note that while the coloring may only be done once, the TCD algorithm can be repeated many times to recompute the path views whenever a link weight has changed. Next, we establish the following theorem.

Theorem 1: Let $G^c = (N, L, W, GREEN, RED)$, $n = |N|$, $e = |L|$, $r = |RED|$, and $g = |GREEN|$, the time complexity for the TCD algorithm is $max(O(r \times n \times log(n)), O(g \times n))$, where *max* is the maximum function.

Proof: See Appendix C.

The time complexity for the regular Dijkstra all-pair shortest path algorithm is $O(n^2 \times log(n))$ for ITS networks. If $g > r \times log(n)$, then the time complexity for the TCD algorithm is $O(g \times n)$. Otherwise, the time complexity of the TCD algorithm is $O(r \times n \times log(n))$. Since $r + g = n$, in either case the time complexity of the TCD does not exceed that of the Dijkstra algorithm. For typical ITS graphs, $n > r, g > 0$ and $g \leq r \times log(n)$ (more likely $r > g$ as in the Troy map where $n = 1590$, $r = 958$, and $g = 632$) will likely hold, giving the TCD has a time complexity of $O(r \times n \times log(n))$.

If $g \gg 0$ (such as in the Troy map where $g = 632$), then $r \ll n$ and a significant improvement by the TCD algorithm over the Dijkstra algorithm in computation time is expected.

2.3. The incremental update algorithms—updating the path views

In order for the path views to be up-to-date, their update interval must be small. It is possible that during such a short interval, only a small number of link weights have changed. To recompute the affected path views using the TCD algorithm requires fixed CPU time independent of the number of affected links. To exploit the small number of links that are affected, we develop update algorithms that update the path views incrementally. As a result, the fewer link weights have changed, the faster the update takes. However, link weight increase and link weight decrease have to be updated separately because the propagation pattern of their incremental path search is different. Therefore, we propose two different incremental update algorithms, the IUA–Decrease algorithm (Appendix D) for link weight decrease and the IUA–Increase algorithm (Appendix E) for link weight increase situations. In Section 5.3, we conduct experiments to show when the query processor should use incremental update algorithms instead of the TCD algorithm.

3. Hierarchical path view model

A flat path view for a large network demands unrealistically large storage and long computation time. To solve both the storage and computation time problems for large ITS networks, we propose a hierarchical path view model (HPVM) based on the road-type classification of the ITS networks and the assumption that roads of higher speed are preferred for long-distance traveling. We argue that typical long-distance traveling is hierarchical in a sense that the travelers first travel local streets to go on main roads in order to connect to highways that are some distance away. Once on a highway, travelers typically travel a relatively long distance while staying on highways until they are near their destinations. At this point, the travelers get off the highways, go down to the main roads, and then back on the local streets in order to reach their destinations. This travel pattern therefore is hierarchical by viewing all roads as the first level links, main roads and faster roads (e.g., highways) as the second level links, and highways as the highest level links.

To organize a hierarchy for precomputed path views, the HPVM first fragments a large network into smaller regions. After fragmentation, the TCD algorithm is used to compute the path view for each region. After the path views are created, either the TCD algorithm or the IUA algorithms are used to maintain the path views frequently. Next, some high-speed links that interconnect different regions are identified and their end nodes collectively form the network at the next higher level. If the network created at the next higher level remains too large to manage, fragmentation can be performed at this level. Links of even

higher speed that interconnect regions at this level are then identified and their end nodes constitute a network at the next level, and so on.

We next describe the fragmentation procedure, the creation of the HPVM for ITS networks, and a formal definition of the HPVM.

3.1. Graph fragmentation

Fragmentation creates smaller regional graphs for which the path views take less space and are much faster to compute. Previous work on hierarchical path search [25] did not employ fragmentation algorithms, thereby assuming graph partition can be done manually. For large ITS networks, manual fragmentation can be very inefficient and ineffective because a good graph fragmentation of a large number of partitions may be hard to find manually.

To address this issue, we first experimented with the optimal data clustering and decomposition algorithm [21] which minimizes the number of border nodes, but the exponential computational cost for this algorithm prevents its practical use. More importantly, our experimentation with this optimal algorithm as well as its sub-optimal heuristic companion proposed in the same paper [21] revealed that the algorithms are not adequate for our problem because they generate excessive number of interconnecting links. The center-based algorithm [11] raises the problem of selecting effective center nodes for each fragment, which is not intuitive for large ITS networks.

Based on this evaluation, we now propose a node-sorting partition algorithm which divides an ITS network into regions of about equal sizes[5] (see figure 4). The partition algorithm (figure 3) calls the following procedures:

- *NodeSortX*(G) sorts all nodes of graph G by their x coordinates.
- *NodeSortY*(G) sorts all nodes of graph G by their y coordinates.
- *Partition*(G,f) evenly partitions nodes of graph G into f regions.
- *IdentifyLink*(G) identifies two kinds of links: If the two end nodes of a link belong to the same region, then this link also belongs to this region, called local link. The border links are links whose two end nodes belong to two different regions.

ALGORITHM *Graph Partition* $(G,f_x \times f_y)$

// Partition $G = (N,L,W)$ into $f_x \times f_y$ regions, where f_x and f_y
// are the number of regions along the x and y dimensions, respectively.
1 *NodeSortX*(G);
2 *Partition*(G,f_x);
3 $\forall\ 1 \leq u \leq f_x$ do
4 *NodeSortY*(G_u);
5 *Partition*(G_u,f_y);
– od
6 *IdentifyLink*(G);

Figure 3. The algorithm to partition a graph.

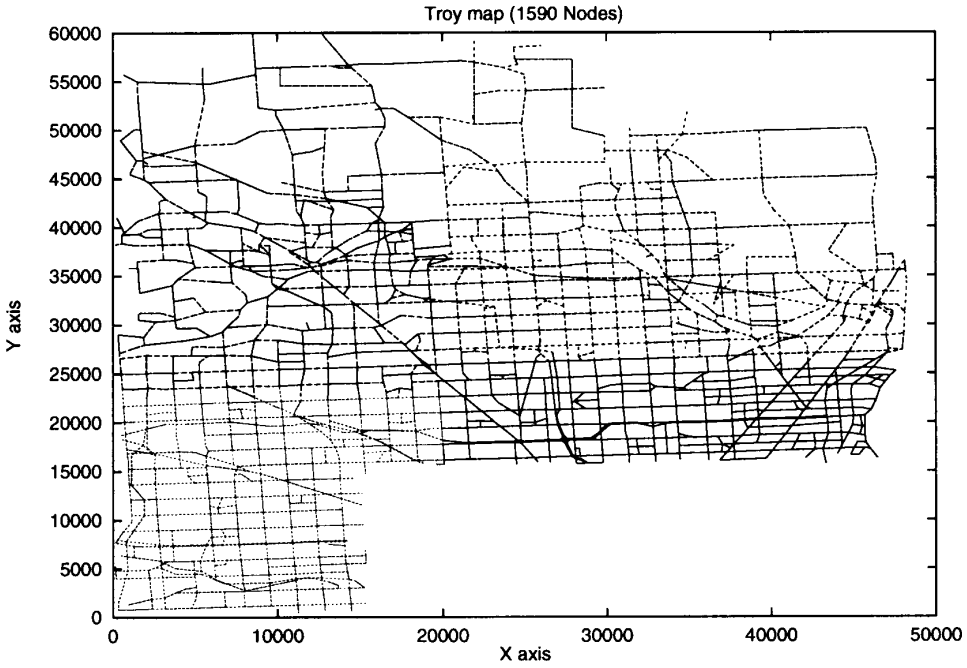


Figure 4. A fragmentation of the Troy City map by the node-sorting algorithm.

To create $f_x \times f_y$ different regions for a graph G, where f_x and f_y denote the number of regions along the x and y dimensions respectively, we first sort all nodes of the graph by their x coordinates (line 1). Next, we evenly partition the sorted nodes into f_x regions (line 2). For each fragment (line 3), we sort the nodes by their y coordinates (line 4) and evenly partition them into f_y regions (line 5). The above process geographically divides the original graph into $f_x \times f_y$ regions of approximately equal size. This algorithm is applicable for many applications as long as the nodes of the graph have a geographical location as found in GIS and ITS graphs. Figure 4 shows the fragmentation of an ITS graph, namely the Troy city map, by the proposed fragmentation algorithm.

To assure the effectiveness of our hierarchical path view model, we establish three desired properties of the fragmented graph. First, each region should be a strongly connected subgraph. This guarantees complete reachability within the regions. Second, each region must have at least one high-speed link that leads to other regions. This is important since it gives opportunity to exit and enter a local region. Third, the nodes at each higher level must also be strongly connected. For the real maps of Troy city and Detroit city we tested for this paper, all three properties are satisfied. We do not claim that all ITS graphs satisfy such constraints trivially. However, because the fragmentation process is an off-line one-time process that is independent of the traffic changes, manual adjustment could be applied in order to satisfy the constraints.

3.2. Hierarchical graph generation

3.2.1. Classification of links and nodes according to road types. Generally, the ITS roads can be classified by a series of road types such as residential streets, main roads, state highways, and interstate highways.[6] Let's refer to these classes of road types as $c_i, 0 \leq i < k$ (k is the number of different classes). To create a hierarchy of k levels, we associate each link in the network with a class c_i, $0 \leq i < k$. For example, we create two link type classes c_0 and c_1 for a 2-level hierarchy, and classify all local street links with the class c_0 and freeway links with the class c_1. This classification for nodes is done using the following policy: Each node adopts the highest among all classes associated with its incoming and outgoing links. For example, if a node has two outgoing links that are associated with classes c_0 and c_1 and one incoming link associated with class c_0, the class associated with this node is c_1.

Based on our previous example of hierarchical travel patterns, a 3-level hierarchical ITS network can be created by first setting all links and their end nodes to class c_0. Next, all links with maximum speed higher than that of the main roads, together with the two end nodes of these links, are set to class c_1. Lastly, all highway links and their end nodes are set to class c_2.

3.2.2. Creating the hierarchical path view structure. To create a k-level hierarchy, k classes from c_0 to c_{k-1} are created and all links and nodes are assigned to a class. The construction of the hierarchy on an ITS network corresponds to the induction process described below.

The **Base condition** corresponds to creating the ground level (level-0) of the hierarchical path view model using the flat graph $G = (N, L, W)$:

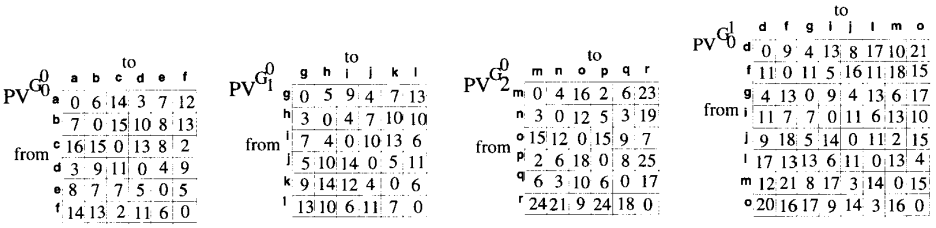
1. Given the level-0 graph $G^0 = G = (N, L, W)$. For example, figure 5(a) is a level-0 graph.
2. Determine the number of regions at level-0, called f_0. In figure 5(a), $f_0 = 3$.
3. G is fragmented into f_0 level-0 regions. In figure 5(b), this process creates regional maps G_0^0, G_1^0, G_2^0.
4. Color each regional subgraph of G by running the 2–Color–Paint algorithm.
5. Compute the path views for all regional subgraphs of G by running the TCD algorithm. In figure 5(b), $PV^{G_0^0}, PV^{G_1^0}, PV^{G_2^0}$ are created.

The **Induction process** creates the level-$k+1$ map from the level-k map:

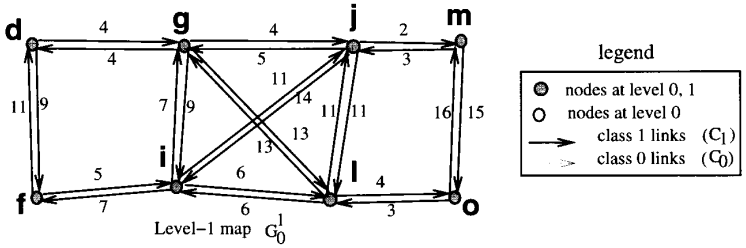
1. Create the level-$k+1$ map by the following steps:

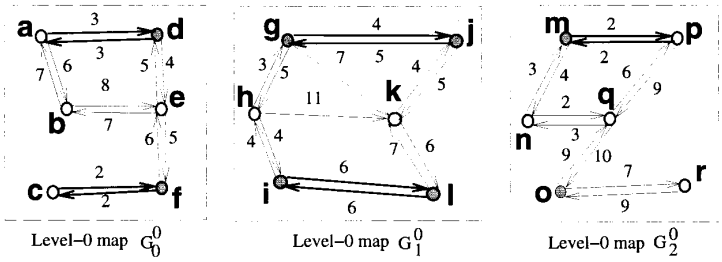
   a The level-$k+1$ nodes are the two end nodes i, j of level-k links such that i and j belong to two different level-k regions, and i and j are associated with classes c_x and c_y, respectively, such that c_x, c_y are higher than c_k. Intuitively, we choose, at level-k, two nodes from different level-k regions that are connected by a high-speed link. We only select links with class higher than c_k to inter-connect level-k

regions because we assume that high-speed roads are preferred for inter-regional traveling. Further, the selection of only high-speed inter-connection minimizes the network size at the next higher level, thereby making path view computation and path retrieval very efficient in HPVM (see Section 5). For example, in figure 5, nodes d and g are the two ends of a class c_1 link. Because they belong to different level-0 regions, they are also level-1 nodes. In contrast, nodes k and n are not the

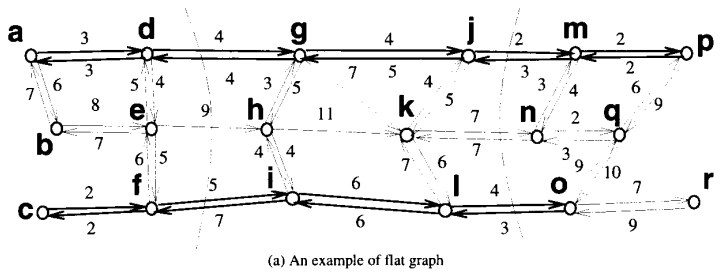


(c) Regional path views





(b) Two levels hierarchical graph



(a) An example of flat graph

Figure 5. An example of the hierarchical path view model.

end nodes of a high-speed (class c_1) link, they therefore are not selected to become higher-level nodes.

b. Any level-k link whose two end nodes, i and j, belong to different level-k regions such that i and j are also level-$k+1$ nodes is a level-$k+1$ link. The weight of this level-$k+1$ link is the same as that of the level-k link from i to j. These level-$k+1$ links are used to inter-connect level-k regions. For example, in figure 5, $< d, g >$ is a class c_1 link that connects nodes from two different level-0 regions, G_0^0 and G_1^0. Therefore, it is also a level-1 link.

c. Any pair-wise combination of two nodes, say i and j, in the same level-k region, which are also identified as level-$k+1$ nodes, form a level-$k+1$ link. The weight of this logic link from i to j is the shortest path weight from i to j encoded in the path view of the level-k region to which i and j belong. These level-$k+1$ links guarantee that within each level-k region, all level-$k+1$ nodes are inter-connected. In figure 5, nodes g, i, j, l are level-1 nodes and belong to the same level-0 region, G_1^0. Therefore, pair-wise combinations between any two of these four nodes form a level-1 link. For example, link $< g, k >$ does not exist in the level-0 map, but is created in the level-1 map with link weight equal to 13. This is the shortest path weight from g to k in the path view, $PV^{G_1^0}$.

2. If $k+1$ is the topmost level, the number of fragments at level-$k+1$ is 1, otherwise determine the proper number of regions at level-$k+1$, denoted by f_{k+1}. In figure 5, level-1 is the top level.
3. The level-$k+1$ graph is fragmented into f_{k+1} level-$k+1$ regions. In figure 5, no more fragmentation is necessary (one fragment).
4. Color each regional subgraph at level-$k+1$ by running the 2–Color–Paint algorithm.
5. Compute the path views for all regional subgraphs at level-$k+1$ by running TCD. In figure 5, this process corresponds to creating the path view, $PV^{G_0^1}$, for the level-1 map.
6. If $k+1$ is the topmost level, the construction of the hierarchical path view is complete.

3.3. Definition of the hierarchical path view model

To define the HPVM model in an unambiguous way, we now present formal definitions of our hierarchical model for an ITS graph $G = (N, L, W)$. We assume that the classification process (Section 3.2.1) is complete and each node and link is associated with a class $c_x, 0 \leq x < l$, with l the maximal level of the hierarchy. Let $class(i)$ be a function that returns the class associated with i, where $i \in N$ or $i \in L$.

Definition 4 (*Fragmentation*): A graph $G_x = (N_x, L_x, W_x), 0 \leq x < f$, is a regional subgraph of a graph
$G = (N, L, W)$ and f is the number of regions in G, where
$N_x \subseteq N$, and $\bigcup_{x=0}^{f-1} N_x = N$, and $N_x \cap N_y = \emptyset$, where $0 \leq x, y < f$ and $x \neq y$, and

$L_x = \{< i,j,w > | i,j \in N_x \text{ and } < i,j,w > \in L\}$, and
W_x is the set of all w in $< i,j,w > \in L_x$.

Definition 5 (*Hierarchy by induction*): An l-level ITS hierarchical graph of $G = (N,L,W)$ is $HG^l = \{G^0, G^1, \ldots, G^{l-1}\}$ defined by the following process:

Base condition:

1. The ground level graph is $G^0 = G = (N,L,W)$
2. Set the f_0 value, where f_0 is the number of level-0 regions.
3. G^0 is fragmented into level-0 regions, $G_x^0, 0 \le x < f_0$.
4. For $x := 0$ to $f_0 - 1$, construct $G_x^{0^c} := 2\text{–}Color\text{–}Paint(G_x^0)$.
5. For $x := 0$ to $f_0 - 1$, compute $PV^{G_x^0} := TCD(PV^{G_x^0}, G_x^{0^c})$.

Induction process (Assuming G^k, G_x^k and $G_x^{k^c}$ and $PV^{G_x^k}, 0 \le x < f_k$, have already been created):

1. The level-$k+1$ graph is $G^{k+1} = (N^{k+1}, L^{k+1}, W^{k+1}), 0 \le k < l-1$, where
   $N^{k+1} = \{i,j | i \in N_p^k, j \in N_q^k, 0 \le p,q < f_k, p \ne q, < i,j,w > \in L^k, class(i) > class(k), class(j) > class(k)\}$, and
   $L^{k+1} = \{< i,j,w > | i,j \in N_p^k, 0 \le p < f_k, i,j \in N^{k+1}, w := PV_{ij}^{G_p^k}\} \cup$
   $\{< i,j,w > | i \in N_p^k, j \in N_q^k, 0 \le p,q < f_k, p \ne q, i,j \in N^{k+1}, < i,j,w > \in L^k\}$, and
   W^{k+1} is the set of all w in $< i,j,w > \in L^{k+1}$.
2. If $k+1 = l$ then $f_{k+1} := 1$, else set the proper f_{k+1} value (f_{k+1} is the number of regions at level-$k+1$).
3. G^{k+1} is fragmented into f_{k+1} level-$k+1$ regions, $G_x^{k+1}, 0 \le x < f_{k+1}$.
4. For $x := 0$ to $f_{k+1} - 1$, $G_x^{k+1^c} := 2\text{–}Color\text{–}Paint(G_x^{k+1})$.
5. For $x := 0$ to $f_{k+1} - 1$, $PV^{G_x^{k+1}} := TCD(PV^{G_x^{k+1}}, G_x^{k+1^c})$.
6. If $k+1 = l$ then the construction of HG^l is complete.

4. The heuristic hierarchical path search algorithm

The conventional approach of ITS path finding is accomplished by the invocation of A^* or hierarchical A^* algorithms [23], [25]. The flat (no hierarchy) path view approach retrieves paths by direct lookup of the path view [13]. In contrast to the above two approaches, the hierarchical path search finds a path by navigating the graph hierarchy of the HPVM, and by composing partial paths encoded in the path views of regional graphs at different levels into a whole path from origin to destination. We call the HPVM's path search algorithm the Heuristic Hierarchical Path Search (HHPS) algorithm (see Appendix F).

The HHPS algorithm captures the dynamicity of traffic in two ways. First, between any two different levels, the path search dynamically finds the best entry point from the low speed link to the high speed link and the best exit point in reverse. This improves the accuracy of the selected paths. Next, each expansion of the HHPS algorithm expands a

regional shortest path by looking up the regional path view. The algorithm does not force path search to follow certain high-speed links once it enters the higher level of the model. The actual path that can be retrieved from this system depends on the link traffic measurements in the path views that are up-to-date with respect to the last update. Therefore, the regional paths selected are always optimal until the last update. If there exists some slow-downs or blockages on high-speed links, HHPS automatically avoids these obstacles. In terms of ITS route guidance, this means that traffic can be automatically rerouted through local streets should there be incidents on the freeways.

The description of the HHPS algorithm and an example of the execution of this algorithm based on figure 5 are given in Appendix F.

5. Experimental Results

5.1. Experiments setup

We experimented with two kinds of maps, synthetic grid maps and real ITS street maps. Synthetic maps allow us to experiment with different parameters like map size. They are used because ITS road networks closely resemble a grid pattern, namely low-out-degree, strongly connection, high locality, etc. For each grid map, we randomly select several vertical and horizontal edges as high-speed links. We assign random weights to all links, but add a control to let the high-speed links have higher average travel speed than local links. Two real street maps, the Troy City with 1590 nodes and the Detroit City of 28,628 node are also used in our studies. For the real maps, we classify the links with a speed limit less than 25 miles per hour (mph) as class-1 links, over 25 mph as class-2 links, and over 45 mph as class-3 links. This fits well with the real-life road type classification as the speed limits are set to 25 mph for most residential streets, above 25 mph and below 45 mph for city main roads, and above 45 mph for highways and service roads.

We use matrices and arrays to implement the hierarchical path views. The procedures that implement all the algorithms are written in C/C++. All experiments are conducted on a dedicated Sun SPARC-20 workstation with a 128MB main memory.

5.2. Path view creation experiments

To test the performance of path view creation, we conducted two sets of experiments, one on medium-sized maps and the other on large maps. To test the medium-sized maps, we create path views using three approaches. They are the Dijkstra algorithm on flat graphs, the TCD algorithm on flat graphs, and the TCD algorithm on 2-level hierarchical graphs that have four regions at the ground level. For each approach, we experimented on a set of grid graphs, from 100 nodes to 3600 nodes, and the Troy map. The results are depicted in figure 6.

Figure 6 shows that, in path view creation, the TCD algorithm is more efficient than the Dijkstra algorithm on flat graphs, and, using the TCD algorithm, the 2-level hierarchical

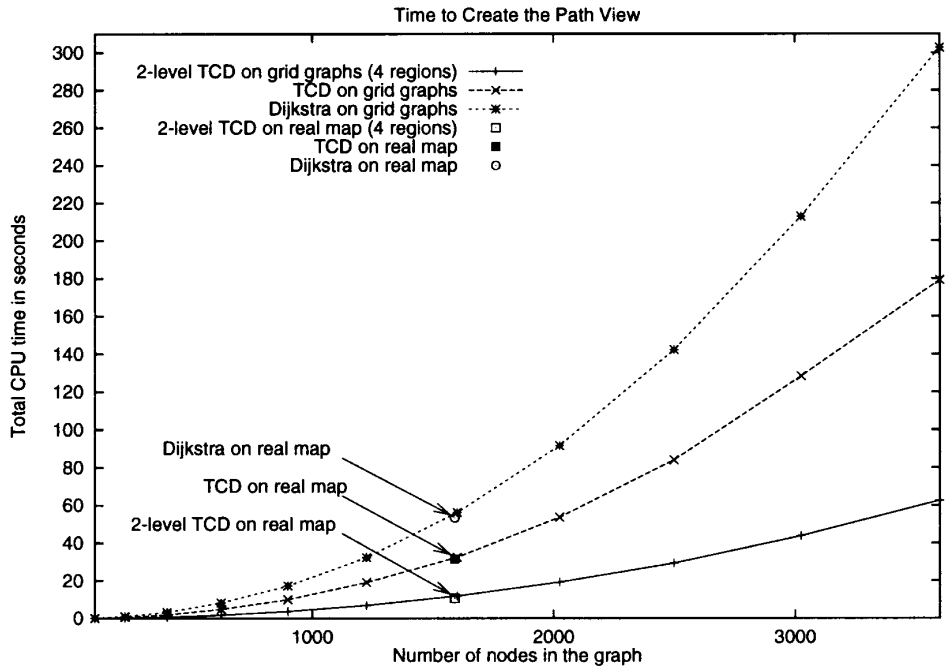


Figure 6. CPU time of creating the path view for medium maps.

graphs are more efficient than the flat graphs. The two approaches using the Dijkstra and TCD algorithms on flat graphs show a sharper upward curve as the number of nodes increases whereas the one using TCD on the 2-level hierarchical graphs shows a near-linear increase. This indicates that the hierarchical approach effectively reduces the time complexity in path view computation. The results for the real map in all approaches mirror those for the grid maps with a similar number of nodes, strengthening our assumption that grid maps are a good approximation of real ITS maps.

To test the large maps, we create path views using the TCD algorithm on flat graphs, 2-level hierarchical graphs, and 3-level hierarchical graphs. The hierarchy of the 2-level and 3-level hierarchical graphs is created by fragmenting the graph at the ground level into subgraphs of about 200 nodes. For the 3-level hierarchical graphs, we also fragment the graph at level 1 into subgraphs of about 200 nodes. The results are shown in figure 7.

We experimented with the flat graphs up to 4900 nodes because the main memory needed to store the path views exceeds the limit of our test machine for maps of greater sizes, and the time to create the path view already becomes unrealistic (> 400 seconds) for dynamic route guidance. In contrast, the time needed to create path views on the 2-level and 3-level hierarchies is much less. Furthermore, creating the path views on the 3-level hierarchy is significantly more efficient than that on the 2-level hierarchy for large maps. For maps with less than 10,000, there is no need for a 3-level structure. Note that creating path views for the 3-level Detroit map requires less than 2 minutes, well within the 3 minutes update interval required in order for path views to be considered up-to-date.

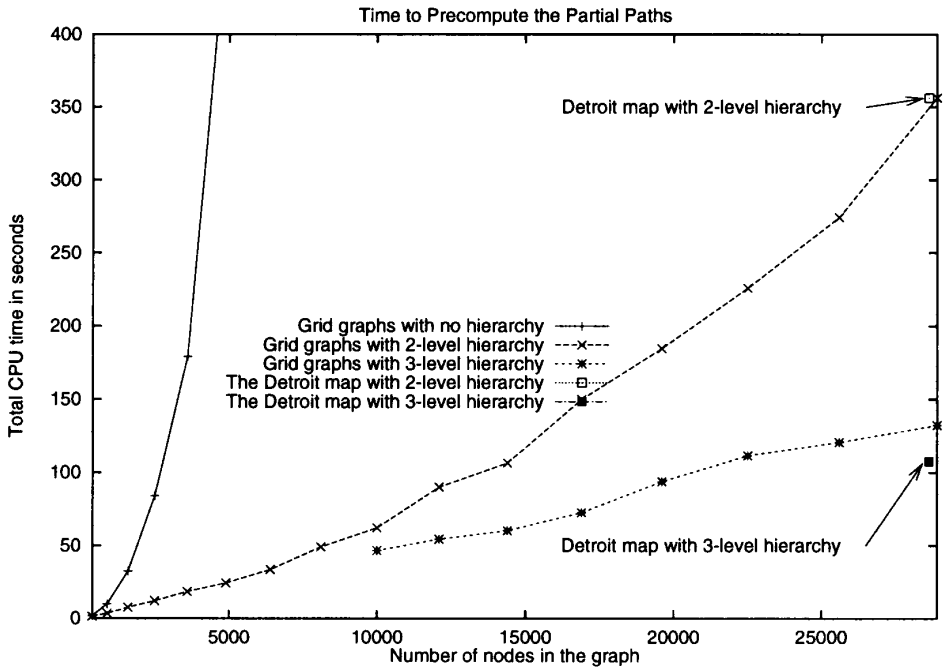


Figure 7. CPU time of creating the path view for large maps.

5.3. Path view maintenance: Recompute vs. incremental update

We conduct a set of experiments to compare two strategies in maintaining the path views. Namely, path views can be recomputed by the TCD algorithm periodically, or incrementally updated by the ICU–Decrease and ICU–Increase algorithms. Our goal is to find out when performing recomputation is advantageous over incremental updates, and vice versa. We first create a 100-node (10×10) grid graph that has 360 links. Next, we randomly select from 1 to 100 links and decrease their link weights to randomly selected non-negative values. For each set of different numbers of links selected, we run the incremental update algorithms to update the path view. We conduct the same experiments again, but this time we increase the weights of the randomly selected links.

The horizontal line in figure 8 depicts the cost of recomputing the path view using the TCD algorithm. The other two curves in figure 8 show that at about 70 links whose weights have decreased, or about 50 links whose weights have increased, the incremental update time is similar to the recomputation time. This means that, for link weight decrease, it is better to run the incremental update than recomputation if the number of links whose weights have decreased is fewer than 70. For increase, it is about 50. The results also show that the ICU–Decrease algorithm consistently outperforms the ICU–Increase algorithm. This can be attributed to the fact that the MIN function (line 5 and line 12 of figure 18), which requires $O(d)$ time where d is the maximum in-degree, is needed in the ICU–Increase algorithm but not in the ICU–Decrease algorithm. Although this set of

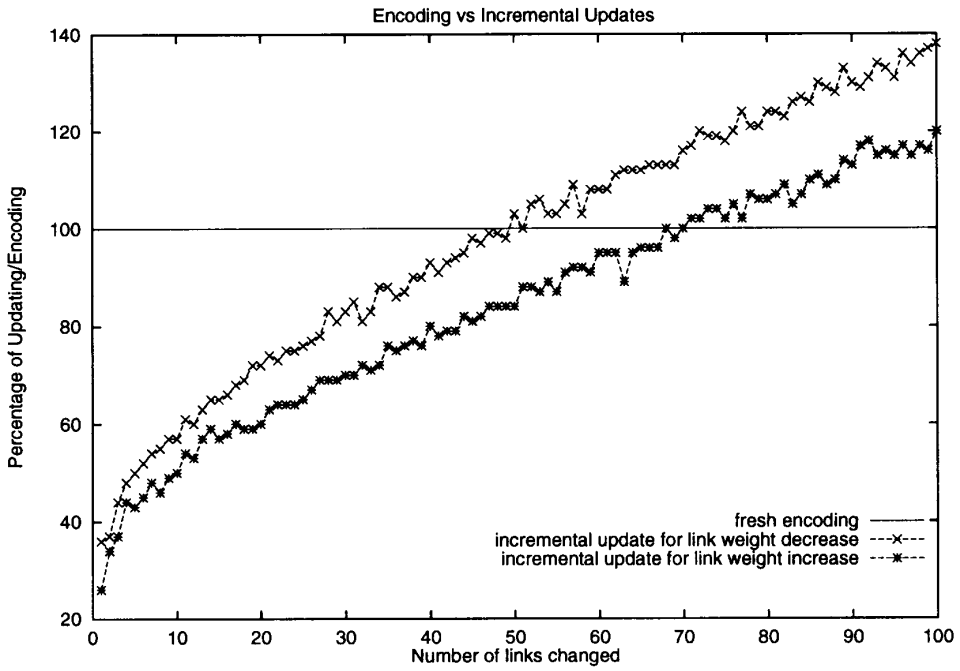


Figure 8. Path view maintenance: Recompute vs. incremental update.

experiments is conducted on a 100-node grid graph, the methodology can be applied to other ITS maps to derive a guideline in selecting the appropriate algorithm for path view maintenance.

5.4. Memory requirement

In this set of experiments, we compare the space requirement to store the path views for three different approaches. They are the flat graphs, 2-level hierarchical and 3-level hierarchical graphs. Figure 9 shows that the path views with hierarchy need much less memory storage than those with no hierarchy. The 3-level hierarchy only exhibits small improvement in space efficiency over the 2-level hierarchy because the dominant factor in storage is the space needed to store the level-1 path structures. In this set of experiments, we create the same number of subgraphs at the ground level for both 2-level and 3-level hierarchical approaches. Therefore, the difference in path view storage is that the 3-level hierarchy further fragments its level-1 graph whereas the 2-level approach does not.

5.5. Fragmentation for path view creation

5.5.1. Fragmentation for 2-level Hierarchical Path Views. In this set of experiments,

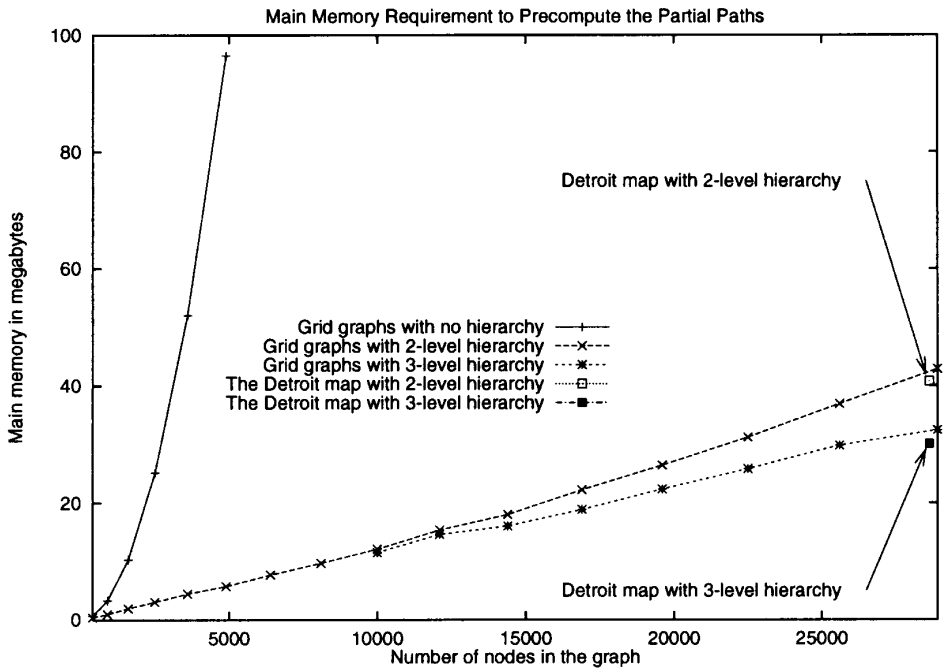


Figure 9. Memory requirement: Flat vs 2-level vs. 3-level.

we measure the impact of the number of subgraphs on the overall performance of HPVM. We first create various 2-level hierarchies for a 10,000-node grid graph, varying the number of subgraphs at level-1 and then running the TCD algorithm to create the path views. The results in figure 10 show that there is a lowest point such that to increase or decrease the number of subgraphs monotonically increases the computation time of the 2-level hierarchical graphs. This is because linearly increasing the number of subgraphs proportionally decreases the number of nodes in each region. The time complexity of computing each path view however is above linear. As a result, the CPU time in computing the level-0 path views decreases as the number of regions increases. However, increasing the number of subgraphs also increases the number of nodes at the next higher level (level 1 in this case), thereby increases the time needed to compute the level-1 path view.

For example, in the most extreme case, if a map of n nodes is fragmented into n subgraphs, each with one node, the total encoding time for all path views at level-0 decreases to $O(n)$, namely $O(1)$ for each graph. However, since the level-1 graphs now have possibly n nodes, the time to encode the level-1 structure needs $O(n^2 log(n))$, the same time complexity required to compute the path view without hierarchy. The results in figure 10 confirm such a U curve behavior with the lowest point being at 100 subgraphs for a 10000-node grid graph. The memory requirements (figure 11) also correspond to a U curve with the lowest point being at about 100 subgraphs.

If the number of subgraphs is large, causing each subgraph to consist of only a small number of nodes, the possibility of some of the subgraphs being disconnected from other

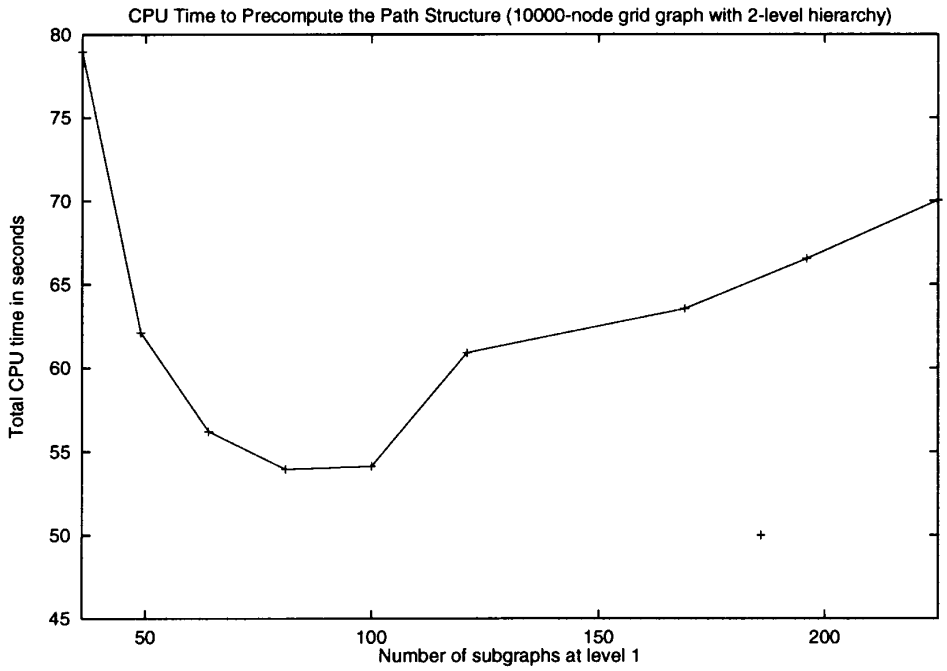


Figure 10. 2-level path structures computation cost by number of subgraphs.

subgraphs is high. For this reason, we did not study graphs with more than 220 fragments in these experiments. To set up an actual ITS road guidance system, we suggest that tools can be built to evaluate the options of fragmentation in validating or invalidating a selected fragmentation scheme. We do not discuss the validation process in this paper, and recognize that, based on its unique road type classification, one-way streets, and inter-connectivity, each different street map may have a different limit for the maximum number of valid subgraphs.

5.5.2. Fragmentation for 3-level Hierarchical Path Views. The fragmentation for the 3-level hierarchy is created by maintaining subgraphs of about equal size across all levels. We experimented on the real map of Detroit City with five fragmentation schemes with different numbers of subgraphs, namely 57, 95, 72, 143, 286. The numbers correspond to creating subgraphs of 500, 400, 300, 200, and 100 nodes respectively. The result in CPU time for computing the path views (figure 12) shows that the more subgraphs, the more efficient the computation is. However, theoretically, the computation time will pick up if the number of subgraphs is very large as in the case of the 2-level hierarchy described previously. The extreme case argument would still apply. Besides, there is a limit of the maximum number of valid subgraphs (or minimum size of each subgraph) in order for subgraphs to remain inter-connected. With the map data we experimented with, we noticed that irregular disconnectivity begins to emerge when the region size falls below 100 nodes. The results in figure 13 show a similar curve for memory requirement.

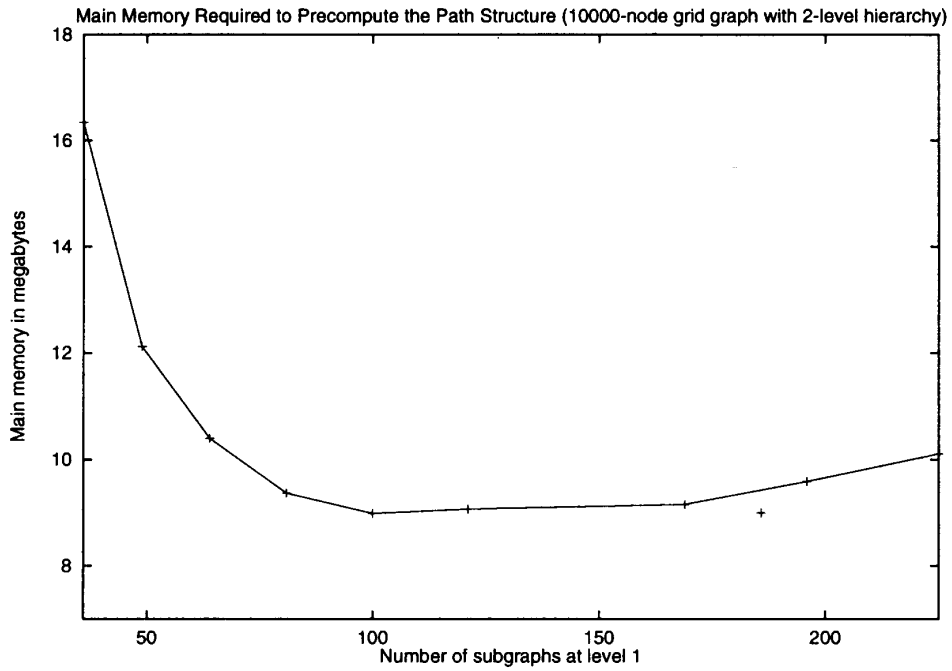


Figure 11. 2-level path structures storage cost by number of subgraphs.

5.6. *Path search time*

We run experiments to compare path search efficiency between different approaches, namely, the flat graphs, the 2-level hierarchy, the 3-level hierarchy, the A^* approach, and the hierarchical A^* approach. The path search time under our 2-level and 3-level hierarchies is the CPU time of running the HHPS algorithm. For A^* and hierarchical A^*, the path search time is the CPU time of running the A^* and hierarchical A^* algorithms [23], [25], respectively. We choose a popular estimation function based on the Euclidean distance between two nodes times the minimum link weight per distance unit for the HHPS, A^*, and hierarchical A^* algorithms: estimate $= \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ * minimum link weight per distance unit. This estimate will always underestimate the actual cost required to travel between any two points and thus represents an appropriate estimate function for ITS applications.

In the hierarchical A^* approach, each node in the graph has a fixed entry/exit node connecting to the high-speed links based on the shortest geographic distance [25]. Once the search reaches the entry node of the source node, it stays on the high-speed links until the exit node for the destination is reached. From the exit node, the search goes down to the local links until the destination is reached.

We randomly selected 1000 paths and conducted path search for all approaches. Because the next-turn information is most crucial in centralized ITS path finding, our path search for all implementations returns the shortest path weight between the source and

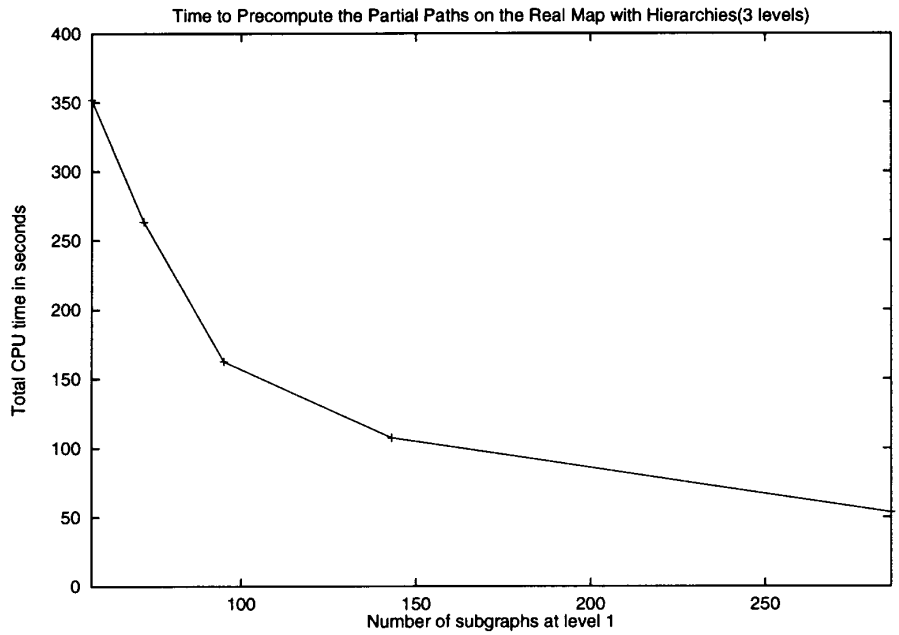


Figure 12. 3-level path structures computation cost by number of subgraphs.

destination. The next-turn information can be trivially incorporated into all approaches in constant time. and the computation for the next-turn information is the same as the computation of the shortest path weight.

The results in figure 14 show that the HHPS path search is most efficient (close to 0 time in figure 14) for the flat graphs because path finding can be accomplished by looking up the precomputed path views. The storage requirement of flat encoded graphs prevents us from experimenting with maps with more than 5000 nodes. The 2-level hierarchy is only slightly better than the 3-level hierarchy, and both are significantly faster than the two A^* approaches. HPVM is faster than the traditional A^* approach in path finding because path search is no longer based on the traversal of individual links, rather, it follows the encoded regional shortest paths stored in the path views. Therefore, HPVM is a compromise between achieving efficient path search and the deployment of resources (both time and space) to maintain the path views.

5.7. *Effectiveness of path search in HPVM*

Table 1. Comparison of average path weight ratios.

Optimal	*2-level HPVM*	*3-level HPVM*
1.00	1.0051	1.0396

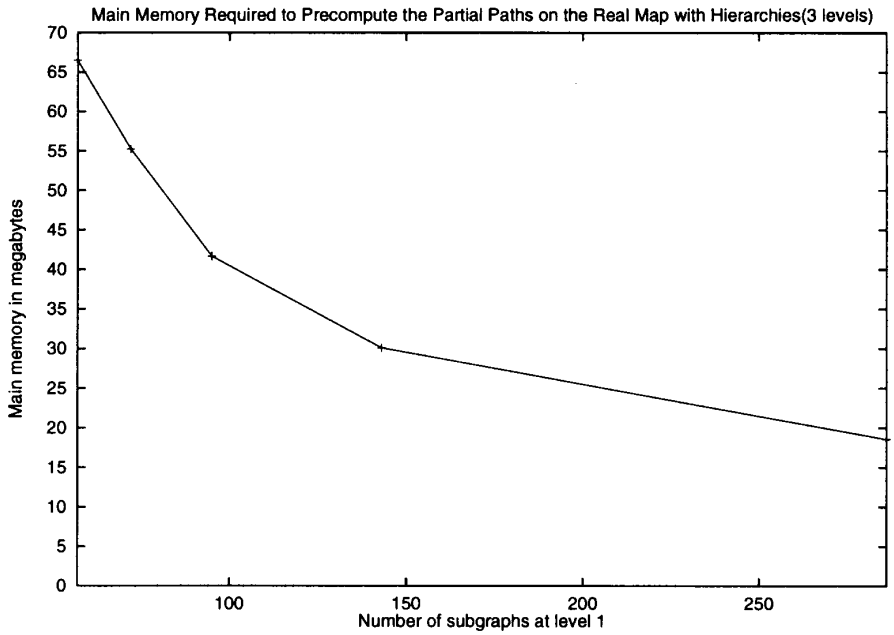


Figure 13. 3-level path structures storage cost by number of subgraphs.

To verify the effectiveness of paths retrieved in HPVM, we compare the average path weights of all paths retrieved in the experiments described in Section 5.6 with the average optimal (minimal) path weights of the same paths. Table 1 shows the path effectiveness of our HPVM using the ratios of average weight of the paths retrieved using a 2-level and 3-level HPVMs (the number of first-level regions for the 3-level HPVM is set to 100) against the optimal average weight of the same paths. The results show that paths retrieved by HPVM are (on average) more costly than the optimal shortest paths only by roughly ½% for a 2-level, and 4% for a 3-level hierarchy. Both increases are practically very small, demonstrating that HPVM gains significantly in reducing the path search time (figure 14) with only negligible loss of path effectiveness.

6. Related research

Transitive closure algorithms presented recently [1], [2], [4], [7], [16], [17], [18] focused on general path problems in disk-based systems. The performance results in [1], [18] in computing the shortest path transitive closure are unsatisfactory for cyclic graphs such as ITS maps because the cycles in the graphs increase the number of node expansions dramatically.

Yang et al, [25] describe an ITS path finding system that is based on a hierarchical A^* algorithm. Their system assumes fixed entry and exit between local streets and freeways. Consequently, it lacks flexibility in dynamic selection of connecting points between links

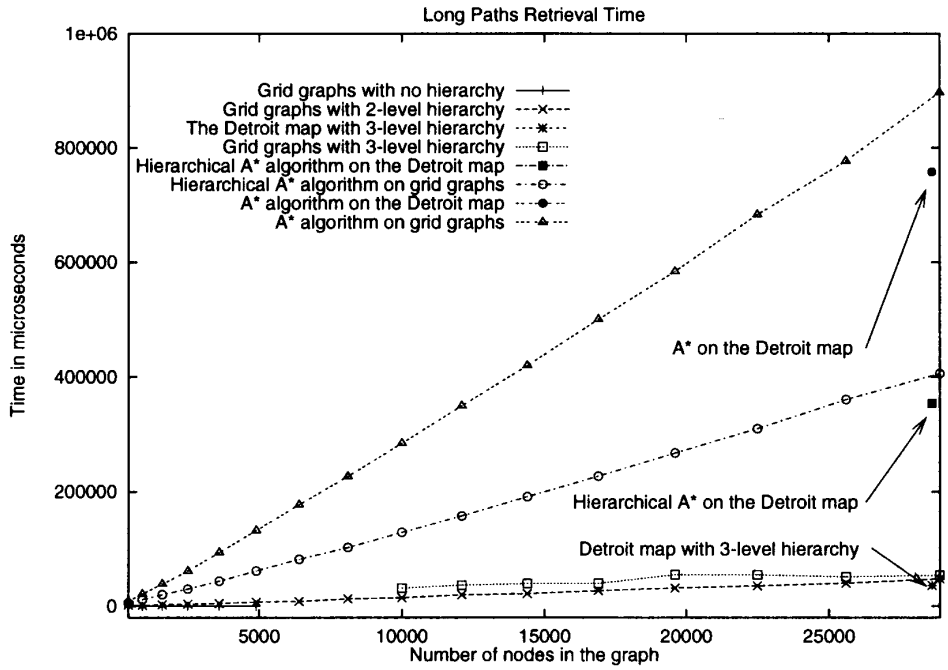


Figure 14. Path retrieval time.

of different levels in the hierarchy, such as provided by HPVM. Furthermore, their system elevates only the selected freeway links to the higher level of the hierarchy. This means that once a vehicle enters the freeways, it remains on the freeways until the fixed exit point associated with the destination is reached. Such a system does not consider rerouting the vehicle through local streets if a local path becomes better due to traffic problems on freeways. HPVM accomplishes such a rerouting because the links on the higher levels are the regional shortest paths stored in the path views. These materialized regional shortest paths are continuously updated when path views are maintained frequently. If freeway blockages exist, the most recent update of the path views will reflect the situation by encoding regional shortest paths that do not pass through the blockage areas. Whether the regional shortest paths are on freeways or local streets depends on their latest traffic conditions.

Several recent efforts have focused on using hierarchical structures for query optimization. Distributed and parallel transitive closure computation is proposed in [9], [10] that divide a relation into fragments. To answer a path query, their systems first select the relevant fragments, and then perform path computation over the selected fragments. Such approaches prefer the underlying graph of the target relation to be acyclic. Therefore, their systems may not be suitable for ITS graphs which are typically strongly connected. Houstma [12] continues upon his previous work [9], [10] with the introduction of the notion of a hierarchical fragment (i.e., the super-graph). Unfortunately, the formation of

the hierarchical fragment is very sensitive to the update of the underlying base relation, and therefore is recommended for stable graphs. Consequently, such a system may not be appropriated for the path finding problem in ITS networks where link weights are changing continuously.

In studying the graph fragmentation, we also experimented with alternative clustering and partitioning algorithms [11], [21]. The optimal data clustering and decomposition algorithm [21], owing to its exponential computational cost, proved to be too inefficient. The sub-optimal heuristic algorithm [11], [21] we tested is also unacceptable because it generates excessive border links. The center-based partition algorithm [11] does accomplish the fragmentation adequately for our purpose as shown in our previous work [14]. However, it raises the problem of proper selection of center nodes, which is not intuitive for large ITS networks. For example, in the 3-level hierarchy experiments we present in Section 5.2, the ground level of the Detroit map is divided into more than 140 regions. Applying the center-based partition algorithm, we need to first manually select more than 140 center points, and then to ensure the selection creates proper fragmentation. Such a task could be very laborious. The fragmentation algorithm presented in this paper (Section 3.1) requires no such laborious process, therefore is more suited for large ITS networks.

In [13], we studied path encoding for ITS applications in the context of flat graphs, and studied the effectiveness of various link clustering strategies for path search in GIS maps [15]. In another related research [19], we also used a fragmentation method to create hierarchical graphs, and to precompute a path view for each subgraph. The hierarchical graph model itself, however, is very different. First, it does not rely on type classification of the links for hierarchical graph decomposition. Second, it pushes up to the next higher level all border nodes—even if they are not classified to belong to the next higher class. This increases the number of nodes at the higher levels. Consequently, view maintenance is not as efficient as HPVM presented in this paper. Because it does not rely on the road type classification, this alternate model is guaranteed to find optimal paths [19]. It thus can be a potential solution to general database recursive query problems in which the road type classification is not applicable.

7. Conclusion and future research

In this paper, we present a hierarchical path view model (HPVM) as a solution to the centralized ITS route guidance. The creation of the hierarchy in HPVM is based on the classification of the road types, and the fragmentation of the network into smaller regions. After the hierarchy is created, the all-pair shortest paths (path views) are precomputed for each region at all levels. The road type classification provides an effective heuristic in that high-speed roads are preferred for inter-regional traffic. The fragmentation decreases the size of each path view, therefore reduces the path view maintenance (precomputation and incremental update) costs in terms of computation time and storage space.

This paper presents the complete HPVM system that includes a graph fragmentation algorithm, a path view computation algorithm (TCD), two path view incremental update

algorithms (ICU–Decrease and ICU–Increase), and a heuristic hierarchical path search algorithm (HHPS). The graph fragmentation algorithm divides an ITS network into smaller regions based on node sorting. The TCD algorithm improves the Dijkstra algorithm in path view computation for ITS networks by incorporating a two color graph-painting technique. The two path view incremental update algorithms can update the path views more efficiently if the traffic changes only affect a small number of links in the network. The HHPS algorithm performs path search more efficiently than the A^* and hierarchical A^* algorithms.

Because the efficiency of our path view maintenance algorithms, the path views in HPVM can be frequently updated to capture the dynamicity of road traffic conditions. With frequently updated path views, the HPVM accomplishes ITS dynamic route guidance by providing the following two features: 1) The entry (exit) point from the lower (higher) level to the higher (lower) level is determined dynamically based on the path views, and 2) the actual routing within each region is determined dynamically based on the path views.

We have conducted experiments that show the computational performance of our proposed algorithms and model. The experimental results show that HPVM requires much less path view maintenance costs in terms of both computation time and storage space. Furthermore, path finding in HPVM is more efficient than the A^* and the hierarchical A^* algorithms. In conclusion, HPVM can better satisfy the requirements for centralized ITS dynamic route guidance than the alternatives.

Our future work includes applying HPVM to a disk environment and integrating other types of ITS query (e.g., spatial path query) processing with path finding computations.

Acknowledgments

This work was supported in part by the University of Michigan ITS Centre of Excellence grant (DTFH61-93-X-0017-Sub) sponsored by the U.S. Department of Transportation and by the Michigan Department of Transportation. Ning Jing, on leave from the Changsha Institute of Technology, is currently visiting the University of Michigan and likes to thank the State Education Commission of P.R. China.

Appendix A. The 2ColorPaint algorithm

The loop in line 1 of the 2ColorPaint algorithm (figure 15) guarantees that all nodes will be painted. An unpainted node j is initialized to be GREEN in line 7. If any of j's children is a GREEN node (lines 8 and 9), node j is changed from GREEN to RED (lines 10 and 11). If no child of j's is a GREEN node (line 12), then all of j's children are painted RED (lines 13 and 14). This guarantees the property that all children of a GREEN node are RED nodes, as specified in Definition. After j is painted, the 2ColorPaint algorithm traverses the network by expanding j's parent nodes (lines 15–17).

The algorithm in figure 15 is analogous to a greedy expansion because it always paints an unpainted node GREEN first, and then changes the color to RED only if it determines this node has at least one GREEN child already. Based on different starting points of expansion, the final paint of a node may be either color. But this is not a problem because we like to have as many nodes painted GREEN as possible and which node is painted with which color is of no concern. Based on our experience, the numbers of GREEN nodes generated by the algorithm when different starting points are chosen differ insignificantly. Furthermore, 2-color painting is a one-time process which has a low time complexity of $O(n \times d)$ where n is the number of nodes and d is the out-degree that is a small constant. For further optimization, one could run the algorithm with every possible starting point in order to find the starting point that yields the best result (maximum number of GREEN nodes). Such an interactive process has a complexity of $O(n^2)$ (the constant d is negligible), which is a very acceptable static cost.

The correctness of the algorithm lies in that if an unpainted node is painted GREEN, all its directly connected children nodes are painted RED. Since a node is painted GREEN only when it is unpainted and RED nodes will not be repainted, this process guarantees that the result of the 2ColorPaint algorithm does not violate the definition of G^c. Figure 2 shows a painted graph generated by the 2ColorPaint algorithm.

ALGORITHM 2*ColorPaint*$(G := (N, L, W))$

DATA STRUCTURES: queue $Q := \emptyset$, set $GREEN := RED := \emptyset$;

```
01  ∀ {i ∈ N | i ∉ GREEN, i ∉ RED} do
02        insert i into Q;
03        while |Q| > 0 do
04              remove j from Q;
05              mark j;
06              if j ∉ GREEN ∧ j ∉ RED then
07                    GREEN := GREEN ∪ {j};
08                    ∀ {k ∈ N | <j, k, w> ∈ L} do
09                          if k ∈ GREEN then
10                                GREEN := GREEN − {j};
11                                RED := RED ∪ {j};
—                           fi od
12                        if j ∈ GREEN then
13                                ∀{k ∈ N | <j, k, w> ∈ L} do
14                                      RED := RED ∪ {k};
—                           od fi fi
15              ∀{i ∈ N | <i, j, w'> ∈ L} do
16                    if i unmarked and i not in Q then
17                          insert i into Q;
—                     fi od od od
18  return G^c := (N, L, W, GREEN, RED);
```

Figure 15. The 2ColorPaint algorithm.

Appendix B. The TCD algorithm

The TCD algorithm is illustrated in figure 16. The AGG (aggregation) and CON (concatenation) in line 11 are path operators in the path algebra developed by [5], and refined by [22], [1], and [18]. For shortest path transitive closure computation, AGG is the minimum function and CON is the *add* function. For each GREEN node i, the TCD algorithm adds the weight of link $< i, k >$ the single-source transitive closure of node k (CON). Then we choose, for each destination, the smallest among all children (AGG) as the shortest path weight. The result is a new single-source path view for the GREEN node. The correctness of the algorithm is based on the fact that all children of a GREEN node are RED nodes, whose single-source transitive closures are already computed. For example, in figure 2, the shortest path from node a to f is represented in the path view as PV^G_{af}). The function $CON(LW_{fc}, PV^G_{af})$ adds the weight of the shortest path from a to f with the link weight from f to c in resulting a path of cost 9. The function $AGG(PV^G_{ac}, CON(LW_{fc}, PV^G_{af}))$ updates the shortest path weight from a to c computed so far (PV^G_{ac}) with the result of the CON operation if the latter yields a shorter path. In figure 2, the $PV^G_{ac} = 4$ which is smaller, therefore the result of $CON(LW_{fc}, PV^G_{af})$ is ignored.

Appendix C. Proof of Theorem 1

Let d be the maximum out-degree of the nodes in N. $O(e \times log(n))$ is the time complexity for the function Dijkstra Single Source Algorithm. The time complexity for processing the RED nodes thus is $O(r \times e \times log(n))$. To compute the path views for all GREEN nodes (lines 8–11 in figure 16) requires a total time complexity of $O(g \times d \times n)$. The total time complexity is: $O(r \times e \times log(n)) + O(g \times d \times n)$. The out-degree d in ITS graphs is a small constant, thus $O(d) = O(1)$, $O(e) = O(d \times n) = O(n)$. As a result, the time complexity becomes: $O(d) \times [O(r \times n \times log(n)) + O(g \times n)] = O(r \times n \times log(n)) + O(g \times n)$. Therefore the time complexity for the *TCD* algorithm is: $max(O(r \times n \times log(n)), O(g \times n))$

Appendix D. The incremental update algorithm for link weight decreases}

The IUA–Decrease algorithm (figure 17) differs from the TCD algorithm in that, for each RED node, the TCD algorithm runs the single-source Dijkstra algorithm for RED nodes, whereas the IUA–Decrease algorithm processes a Dijkstra-style search in reverse to compute the shortest paths from all nodes in the network to this RED node (lines 1–12 in figure 17). Initially, those links whose weights have decreased are stored in the set S. Then, for each RED node k, the shortest paths from all other nodes to this RED are computed by the following process. First, for each link $< i, j, w >$ in S, a new path from i to k is computed based on the decreased LW_{ij} (line 4). If the new path is better than the old path from i to k, i is inserted into the heap H for further expansion. Lines 2–6 correspond to initializing the heap, whereas lines 7–12 correspond to the incremental reverse traversal

```
ALGORITHM TCD (PV^G, G^c)
//  PV^G is the shortest path adjacency matrix for G.
//  G^c = (N, L, W, GREEN, RED).
01  ∀{i, j ∈ N} do
02      if i = j then
03          PV^G_ij := 0;
04      else
05          PV^G_ij := ∞;
—       fi od
06  ∀{i ∈ RED} do
07      DijkstraSingleSourceAlgorithm(i);
—   od
08  ∀{i | i ∈ GREEN} do
09      ∀{k | <i, k, w> ∈ L} do
10          ∀{j | j ∈ N} do
11              PV^G_ij := AGG(PV^G_ij, CON(LW_ik, PV^G_kj));
—           od od od
12  return PV^G;
```

Figure 16. The TCD algorithm.

starting from the nodes in the heap. During each expansion in reverse traversal, we only continue the expanding thread if a better path is found (lines 10–11), otherwise the thread is terminated. After the shortest paths from all nodes reaching the RED nodes are computed, we compute the shortest paths from all nodes reaching the GREEN nodes. The process for GREEN nodes in the IUA–Decrease algorithm (lines 13–16) is similar to that in the TCD algorithm, but in reverse direction.

The worst-case time complexity for the IUA–Decrease algorithm is the same as that of the TCD algorithm. But, the IUA–Decrease algorithm propagates search only to affected nodes (lines 5, 11 in figure 17) where the TCD propagates expansion through all nodes. If the number of affected links is small (i.e., $|S|$ is small), the actual process time for the IUA–Decrease algorithm can be shorter than for the TCD algorithm.

Appendix E. The incremental update algorithm for link weight increases

The IUA–Increase algorithm (figure 18) is designed to update the path view for an ITS network for the link weight increase situation (see figure 18) It is slightly different from the IUA–Decrease algorithm in that an additional MIN function in the inner loop (lines 5 and 12) is used to determine the new best paths. Only if the new path weight has increased (lines 6, 13 of figure 18), does the expansion thread continue. The time complexity of the IUA–Increase algorithm is the same as that of the IUA–Decrease algorithm because the MIN function only linearly increases the time complexity for ITS networks where the out-degree is a small constant.

```
PROCEDURE IUA_Decrease (S, PV^G, G^c)
//  S is a set of links whose link weights have decreased.
//  PV^G is the path view matrix of G, G = (N, L, W).
//  G^c = (N, L, W, GREEN, RED).

DATA STRUCTURES: heap H := ∅;
//  the heap H is an array of tuples <n, v> where the tuple with the
//  smallest v being at the top of the heap

01  ∀{k ∈ RED} do
02      ∀{<i, j, w> ∈ S} do
03          if i ≠ j then
04              PV^G_ik := AGG(PV^G_ik, CON(LW_ij, PV^G_jk));
05              if PV^G_ik changed then
06                  insert <i, PV^G_ik> into H; // replace if <i, −> already in H
—               fi fi od
07      while |H| > 0 do
08          remove <j, v> from top of H;
09          ∀{i ∈ N | <i, j, w> ∈ L} do
10              PV^G_ik := AGG(PV^G_ik, CON(LW_ij, PV^G_jk));
11              if PV^G_ik changed then
12                  insert <i, PV^G_ik> into H; // replace if <i, −> already in H
—               fi od od od
13  ∀{k | k ∈ GREEN} do
14      ∀{j | <K, j, w'> ∈ L} do
15          ∀{i | i ∈ N} do
16              PV^G_ij := AGG(PV^G_ij, CON(LW_kj, PV^G_ik));
17  return PV^G;
```

Figure 17. The *IUA_Decrease* algorithm.

Appendix F. The heuristic hierarchical path search algorithm

The Heuristic Hierarchical Path Search (HHPS) algorithm in figure 19 calls the following functions:

- $MarkNode(n_i, G_p^l)$ marks node n_i of region G_p^l in the hierarchical graph.
- $Border(G_p^l)$ denotes the nodes in G_p^l which also appear in regions at a higher level.
- $GetEstimate(n_i, dest)$ returns the estimated shortest path weight between n_i and *dest*. We use the Euclidean distance times the minimum weight per distance unit to estimate the shortest path weight. The estimate function always underestimates the actual shortest path weight, therefore it guarantees the correctness of the algorithm.

Starting from the destination node *dest*, the HHPS algorithm marks nodes upward in the hierarchy recursively if the nodes can lead to the designated destination (lines 1–7). The

PROCEDURE *IUA_INCREASE* (S, PV^G, G^c
// S is a set of links whose link weights have increased.
// PV^G is the path view matrix of G, $G = (N, L, W)$.
// $G^c = (N, L, W, GREEN, RED)$.
// *MIN* is the minimum function.

DATA STRUCTURES: heap $H := \emptyset$;
// a heap is an array of tuples $<n, v>$ where the tuple with the
// smallest v at the top of the heap

```
01  ∀{k ∈ RED} do
02      ∀{ <i,j,w> ∈ S} do
03          if i ≠ j then
04              if PV^G_ik = PV^G_ij + PV^G_jk then
05              PV^G_ik := MIN{LW_ix + PV^G_xk|(<i,x,w'> ∈ L};
06              if PV^G_ik changed then
07                  insert <i, PV^G_ik> into H; // replace if <i,−> already in H
—               fi fi od
08      while |H|>0 do
09          remove <j,v> from top of H;
10          ∀{i ∈ N|<i,j,w> ∈ L} do
11              if PV^G_ik = PV^G_ij + PV^G_jk then
12              PV^G_ik := min{LW_ix + PV^G_xk|(<i,x,w'> ∈ L};
13              if PV^G_ik changed then
14                  insert <i, PV^G_ik> into H; // replace if <i,−> already in H
—               fi od od od
15  ∀{i,k|i ∈ N, k ∈ RED} do
16      PV^G_ik := ∞;
—   od
17  ∀{k|k ∈ GREEN} do
18      ∀{j|<k,j,w''> ∈ L} do
19          ∀{i|i ∈ N} do
20              PV^G_ik := AGG(PV^G_ik, CON(LW_kj, PV^G_ik));
—           od od od
21  return PV^G;
```

Figure 18. The IUA_Increase algorithm.

HHPS algorithm traverses the hierarchy beginning from source *src* (line 8) until it reaches the destination *dest* (lines 11 and 12). The upward expansions at level-l traverse all the regions which contain the current expansion node n_i (lines 13 and 14), and for each region, traverse from the current expansion node n_i to all the border nodes of that region (lines 15 and 16). The downward expansions (lines 17–19) expand all the marked nodes, and terminate when the expanded node is the destination (line 11).

To prune the expansion tree, the HHPS algorithm uses a prediction that estimates

ALGORITHM $HHPS(HG, src, dest)$
// HG is the hierarchical path view of G
// src is the source node; $dest$ is the destination node of the search

DATA STRUCTURES: heap $H := \emptyset$, is an array of tuples $<n, w, e, l, d>$ where
// the tuple with the smallest $w + e$ being at the top of the heap, and n is
// the current expansion node, and w is the actual accrued weight from src to n,
// and e is the estimated weight from n to $dest$, and l is the current level of
// hierarchy, and d is the next expansion direction $(UP/DOWN)$ from n.
// $S := \emptyset$, is a queue of tuples $<n, G_p^l>$, where $n \in G_p^l$.

01 insert $<dest, G_p^0>$ into S; // $dest \in G_p^0$
02 while $|S| > 0$ do
03 remove $<n_i, G_p^l>$ from S;
04 $MarkNode(n_i, G_p^l)$;
05 $\forall\, n_j \in Border(G_p^l)$ do
06 $\forall\, q \in \{q' | n_j \in G_{q'}^{l+1}\}$ do
07 insert $<n_j, G_q^{l+1}>$ into S;
— od od od
08 insert $<src, 0, GetEstimate(src, dest), 0, U, P>$ into H;
09 while $|H| > 0$ do
10 remove $<n_i, w, e, l, d>$ from H;
11 if $n_i = dest$ then
12 return w;
— fi
13 if $d = UP$ then
14 $\forall\, p \in \{p' | n_i \in G_{p'}^l\}$ do
15 $\forall\, n_j \in Border(G_p^l)$ do
16 insert $<n_j, w + PV_{n_i n_j}^{G_p^l}, GetEstimate(n_j, dest), l + 1, UP>$ into H;
— od od fi
17 $\forall\, p \in \{u' | n_i \in G_{p'}^l\}$ do
18 $\forall$ marked $n_j \in G_p^l$ do
19 insert $<n_j, w + PV_{n_i n_j}^{G_p^l}, GetEstimate(n_j, dest), l - 1, DOWN>$ into H;
— od od od

Figure 19. The heuristic hierarchical path search algorithm HHPS.

the weight of the untraversed portion of the path from the current expanded node to the destination. Like the A^* algorithm, the HHPS algorithm is priority-based. The priority of the next expansion node n_i is given to the active expansion thread that has the smallest $w + e$ value. Unlike the A^* algorithm which traverses only one link at a time, the HHPS algorithm traverses a complete path segment encoded in the path view. The number of expansions needed for the HHPS is therefore smaller than that for the A^*. As a result, path retrieval of HHPS is expected to be more efficient than that of A^*—as our experiments confirm (see Section 5).

For example, if we use the HHPS algorithm to find the path from node b to node q in

figure 5, lines 1–7 of figure 19 mark nodes o and m in G_2^0. Line 8 puts $< b, 0, 0, 0, UP >$ into the heap H, assuming the $GetEstimate(n, dest)$ function always returns 0—worst-case prediction. Within the while loop (line 9), the algorithm executes statements in lines 13 to 16 after $< b, 0, 0, 0, UP >$ is removed from H in line 10. In figure 5(b), $Border(G_0^0) = \{d,f\}$. At this point, two tuples, $< d, 10, 0, 1, UP >$ and $< f, 13, 0, 1, UP >$ are inserted into H. Since there is no marked nodes in G_0^0, the statements in lines 18 and 19 are skipped.

In the next iteration of the while loop (line 9), the tuple $< d, 10, 0, 1, UP >$ is removed from H (line 10). Because G_0^2 has no border nodes, the statements in lines 15 and 16 are skipped. Because nodes m and o in G_0^1 are marked, the statements in lines 17–19 insert tuples $< m, 20, 0, 0, DOWN >$ and $< o, 31, 0, 0, DOWN >$ into H. In the next iteration of the while loop in line 9, the tuple $< f, 13, 0, 1, UP >$ is removed from H and tuples $< m, 31, 0, 0, DOWN >$ and $< o, 28, 0, 0, DOWN >$ are put into H. After $< m, 20, 0, 0, DOWN >$ is removed from H, tuple $< q, 26, 0, 0, DOWN >$ is inserted into H. Finally, the tuple $< q, 26, 0, 0, DOWN >$ is removed from H, q is the designated destination and 26 is returned by the HHPS algorithm. The weight of the path retrieved by the HHPS algorithm from node b to node q is 26.

Although the HHPS algorithm presented in this paper returns only the path weight, the actual path, $b - a - d - g - j - m - n - q$ in our example, can be retrieved by a modified HHPS algorithm which stores the interim paths, and by a modified path view structure that also encodes the next-hop node for each shortest path weight [13]. In this paper, without loss of generality, we only present the simplified model to explain the basic idea—while our implemented system incorporates the actual retrieval of the next hop itself.

Notes

1. Dynamic route guidance here means guided vehicles travel paths selected based on the most up-to-date traffic conditions.
2. We assume a path view is up-to-date if it is updated at an interval of < 3 minutes. Such a requirement is safe because preliminary testing in the Loral system showed that the communication delay is between 6 to 8 minutes.
3. In fact, the complexity is $O(e \times n \times log(n))$, where e is the number of links and n is the number of nodes in the network. But the outdegree in ITS road networks is usually a small constant c, i.e. $e = c \times n$, therefore the complexity becomes $O(n^2 log(n))$.
4. We processed this coloring technique on the 1590-node real map of the Troy city, and derived 632 GREEN nodes and 958 RED nodes.
5. This is desirable because the time required to compute the path views of all regions that divide the network is minimal if the regions are of the same size.
6. A difference between these road types is typically the average travel speed. Because no road classification is available in our experimental maps, we base the classification on the maximum speed limit.

References

1. R. Agrawal, S. Dar and H.V. Jagadish. Direct Transitive ''Closure Algorithms: Design and Performance Evaluation,'' *ACM Transactions on Database Systems*, Vol. 15, No. 3, pp. 427–458, 1990.

2. R. Agrawal and H.V. Jagadish, 1990, ''Hybrid Transitive Closure Algorithms,'' *Proc. of the 16th VLDB*, pp. 326–334.
3. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, pp. 207–209, 1974.
4. F. Bancilhon. ''Naive Evaluation of Recursively Defined Relations,'' in *On Knowledge Base Management Systems—Integrating Database and AI systems*, Springer-Verlag: New York, 1985.
5. B. Carr. Graphs and Networks, Clarendon Press, Oxford, England, 1979.
6. E.W. Dijkstra. ''A Note on Two Problems in Connection with Graphs,'' *Numerische Mathematik*, pp. 269–271, 1959.
7. J. Ebert. ''A Sensitive Transitive Closure Algorithm,'' *Information Processing Letters*, 12., pp. 255–258, 1959.
8. M.J. Egenhofer. ''What's Special about Spatial?'' Database Requirements for Vehicle Navigation in Geographic Space, *Proc. of the 1993 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 398–402, 1993.
9. M.A.W. Houstma, P.M.G. Apers and S. Ceri. ''Complex Transitive Closure Queries on a Fragmented Graph,'' *Proc. of the 3rd Int'l Conf. on Data Theory, Lecture Notes in Computer Science*, Springer-Verlag, pp. 470–484, 1990.
10. M.A.W. Houstma, P.M.G. Apers, and S. Ceri. ''Distributed Transitive Closure Computations: The Disconnection Set Approach,'' *Proc. of the 16th VLDB*., pp. 335–346, 1990.
11. M.A.W. Houstma, P.M.G. Apers, and G.L.V. Schipper. Data Fragmentation for Parallel Transitive Closure Strategies, *Proc. of the 9th Int'l Conf. on Data Engineering*, pp. 447–456, 1993.
12. M.A.W. Houstma, F. Cacace, and S. Ceri. ''Parallel Hierarchical Evaluation of Transitive Closure Queries,'' *1st Int'l Conf. on Parallel and Distributed Inf. Sys*., pp. 130–137, 1990.
13. Yun-Wu Huang, Ning Jing, and A. Elke Rundensteiner. ''A Semi-Materialized View Approach for Route Maintenance in IVHS,'' *Proc. of the 2nd ACM Workshop on Geographic Information Systems*, pp. 144–151, 1994.
14. Yun-Wu Huang, Ning Jing, and A. Elke Rundensteiner. ''Hierarchical Path Views: A Model Based on Fragmentation and Transportation Road Types,'' *Proc. of the 3nd ACM Workshop on Geographic Information Systems*, 1995.
15. Yun-Wu Huang, Ning Jing, and A. Elke Rundensteiner. ''Effective Graph Clustering for Path Queries in Digital Map Databases,'' *Proc. of 5th Int'l Conf. on Information and Knowledge Management*, pp. 215–222, 1996
16. Y.E. Ioannidis. ''On the Computation of the Transitive Closure of Relational Operators,'' *Proc. 12th Int'l Conf. on VLDB*, pp. 403–411, 1986.
17. Y.E. Ioannidis and R. Ramakrishnan. ''An Efficient Transitive Closure Algorithm,'' *Proc. 14th Int'l Conf. on VLDB*, pp. 382–394, 1988.
18. Y. Ioannidis, R. Ramakrishnan, and L. Winger. ''Transitive Closure Algorithms Based on Graph Traversal,'' *ACM Trans. on Database Systems*, Vol. 18, No. 3, pp. 512–576.
19. Ning Jing, Yun-Wu Huang, and A. Elke Rundensteiner. ''Hierarchical Optimization of Optimal Path Finding for Transportation Applications,'' *Proc. of 5th Int'l Conf. on Information and Knowledge Management*, pp. 261–268, 1996.
20. Laurel. IVHS architecture requirement document, 1994.
21. W.T.Jr. McCormick, P.J. Schweitzer, and T.W. White. ''Problem Decomposition and Data Reorganization by a Clustering Technique,'' *Operations Research*, Vol. 20, No. 5, pp. 993–1009, 1972.
22. A. Rosenthal, S. Heiler, U. Dayal, and F. Manola. ''Traversal Recursion: A Practical Approach to Supporting Recursive Applications,'' *Proc. ACM-SIGMOD*, pp. 166–176, 1986.
23. S. Shekhar, A. Kohli, and M. Coyle. ''Path Computation Algorithms for Advanced Traveller Information Systems,'' *IEEE 9th Int'l Conf. on Data Engineering*, pp. 31–39, 1993.
24. S. Warshall. ''A Theorem on Boolean Matrices,'' *JACM*, 9, 1, pp. 11–12, 1962.
25. T.A. Yang, S. Shekhar, B. Hamidzadeh, and P.A. Hancock. ''Path Planning and Evaluation in IVHS Databases,'' *VNIS*, pp. 283–290, 1991.

Yun-Wu Huang received the B.S. degree in Management Science from National Chiao-Tung University in 1982, and the M.S. degree in Computer Science from Indiana University in 1989. He had worked as a computer professional in the areas of database and computer network between 1989 and 1995. Currently, he is a Ph.D. candidate in the Department of Electrical Engineering and Computer Science at the University of Michigan. His current research interests include spatial databases, multi-media databases, and geographic information systems.

Ning Jing received the B.S. and M.S. degrees in Electrical Engineering, and the Ph.D. degree in Computer Science from the Changsha Institute of Technology, Changsha, China. Dr. Jing has received a High Education Award from the State Department in 1992 and an Outstanding Visiting Scholar Grant from the State Education Commission in 1994.

He is currently a faculty member of the Department of Electrical Engineering at the Changsha Institute of technology. From 1994 to 1996, he had been a visiting scholar in the Department of Electrical Engineering and Computer Science at the University of Michigan. His current research interests include object-relational databases, multi-media databases, Databases for internet information services, and geographic information systems.

Elke Angelika Rundensteiner received a BS degree (Vordiplom) from the Johann Wolfgang Goethe University, Frankfurt, West Germany, and a Ph.D. degree from the University of California, Irvine. Dr. Rundensteiner has received numerous honors and awards, including a Fulbright Scholarship, an NSF Young Investigator Award in databases in 1994, and an Intel Young Investigator Engineering Award, and an IBM Partnership Award. Dr. Rundensteiner is a member of IEEE and ACM.

She is currently a faculty member of the Department of Computer Science at the Worcester Polytechnic Institute, after having been an Assistant Professor in the Department of Electrical Engineering and Computer Science at the University of Michigan. Dr. Rundensteiner's goal is to develop database technology to address modeling and querying requirements of advanced applications. Her current research efforts include object-oriented databases, view techniques for data warehousing and database evolution, multi-media databases, and geographic information systems.