Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its Performance Evaluation

Ning Jing, Yun-Wu Huang, and Elke A. Rundensteiner, *Member*, *IEEE*

Abstract—Efficient path computation is essential for applications such as intelligent transportation systems (ITS) and network routing. In ITS navigation systems, many path requests can be submitted over the same, typically huge, transportation network within a small time window. While path precomputation (path view) would provide an efficient path query response, it raises three problems which must be addressed: 1) precomputed paths exceed the current computer main memory capacity for large networks; 2) disk-based solutions are too inefficient to meet the stringent requirements of these target applications; and 3) path views become too costly to update for large graphs (resulting in out-of-date query results). We propose a *hierarchical encoded path view* (*HEPV*) model that addresses all three problems. By hierarchically encoding partial paths, *HEPV* reduces the view encoding time, updating time and storage requirements beyond previously known path precomputation techniques, while significantly minimizing path retrieval time. We prove that paths retrieved over *HEPV* are optimal. We present complete solutions for all phases of the *HEPV* approach, including graph partitioning, hierarchy generation, path view encoding and updating, and path retrieval. In this paper, we also present an in-depth experimental evaluation of *HEPV* based on both synthetic and real GIS networks. Our results confirm that *HEPV* offers advantages over alternative path finding approaches in terms of performance and space efficiency.

Index Terms—Path queries, path view materialization, hierarchical path search, GIS databases, graph partitioning.

1 Introduction

THE capability of computing path queries is an essential feature in new database systems for many advanced applications such as navigation systems Geographical Information Systems (GIS), and computer networks [1], [2], [4], [5], [6], [11], [12], [14], [21], [22], [23], [24], [31]. For example, one of the primary functionalities of Intelligent Transportation Systems (ITS) [10], [27], [28], [32], [33] is to find routes from the current location of a vehicle to a desired destination with a minimum cost (where cost could represent travel time, shortest distance, minimal toll charges, etc.). This paper investigates solutions for path finding in general with a particular focus on addressing the problems inherent to navigation system applications [10], [32].

For these applications, we identify four critical issues. First, we focus on systems that must compute path queries submitted by a potentially large number of concurrent requests (e.g., during peak rush hours). Our solution therefore must be scalable in the number of path query requests. Second, our solution must handle the dynamic nature of the transportation network, i.e., it must provide up-to-date query results even when the underlying transportation network data changes frequently. Third, our solution must provide a response at a near real-time level of performance (i.e., within seconds). If a response cannot be given within a few seconds, then the driver will miss his next turn, and thus the system will not be effective. Fourth, based on the requirements of *instruction-based* navigation systems [29], we are interested in efficiently determining the next link for the desired path rather than the complete path. This is justified by the fact that a driver needs to know immediately *which next turn* to take whereas the knowing the *complete* path is less critical—especially as it may still be adjusted according to changing conditions while traveling.

Traditional path finding solutions use variations of the heuristic A^* search algorithm to compute paths [32], [33]. Using such algorithms to process path queries, the potentially large number of concurrent path requests specified over a large transportation network amounts to a huge collection of computational tasks. As a result, the stringent constraint of the path query response time may not be satisfied.

An alternative solution is to precompute all-pair shortest paths and store them on-line [15]. Path computation therefore is reduced to simple look-ups of the requested path from the precomputed path view structure. Although path queries can be processed very efficiently, the recomputation or update of the path view can be very inefficient for large networks (e.g., it took 4 minutes for a graph of 3,600 nodes on a Sun SPARC-20 based on our experiments.). This limitation prevents the path view from being updated frequently, undercutting the accuracy of the computed paths.

• N. Jing is with the Department of Electrical Engineering, Changsha Institute of Technology, Changsha, Peoples Republic of China. E-mail: ningjing@pdns.nudt.edu.cn.
• Y.-W. Huang is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109. E-mail: ywh@umich.edu.
• E.A. Rundensteiner is with the Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA 01609. E-mail: rundenst@cs.wpi.edu.

Manuscript received 24 July 1996.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 104462.

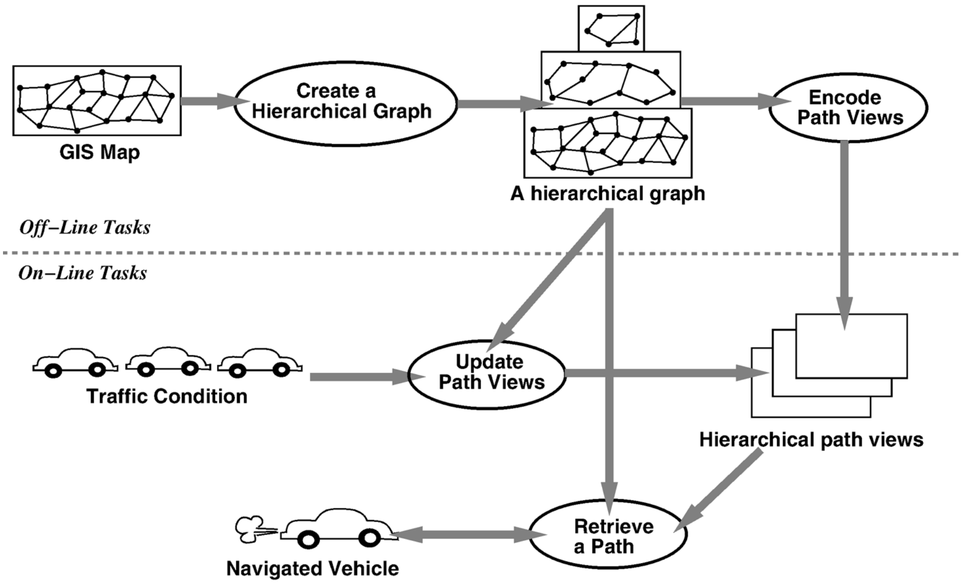


Fig. 1. *HEPV* framework.

To address the above issue, we have presented elsewhere a hierarchical graph model for ITS based on link classifications [16], [18]. In this system, the hierarchy is created by pushing up high-speed roads such as highways to the next higher level. Although path computation over this hierarchical structure is very efficient, each computed path is *not guaranteed to be optimal* in the sense that it may be different from the path computed for the same origin-destination (O-D) pair when no hierarchical structure is imposed.

To overcome this limitation, the goal of this paper is to achieve high efficiency in path view computation and update without losing the benefit of time-efficient retrieval of *optimal* paths. To this end, we propose a hierarchical graph model, called *Hierarchical Encoded Path View* (*HEPV*). *HEPV* accomplishes four major tasks (Fig. 1):

- The *hierarchy generation task* constructs a hierarchical graph by fragmenting the flat graph into partitions and by pushing up border nodes to generate the hierarchy.
- The *hierarchical path encoding task* precomputes and stores the shortest paths for each partition of the hierarchy.
- The *hierarchical path view update task* recomputes path views for partitions whose traffic condition has changed since last update.
- The *path retrieval task* retrieves an optimal path over the hierarchical graph based on user requests. Proofs of optimality are provided in Section 5.1 that can be used as basis for deriving optimal hierarchical path search algorithms.

In this paper, we present detailed analyses and experimental results for our *HEPV* approach using synthetic graphs and real transportation networks. Compared to other approaches, the *HEPV* approach is more efficient with respect to both performance and memory requirement. For a large graph of 3,600 nodes, the *HEPV* needs only 40 seconds to update the materialized path view in the worst case when all fragments are changed. The time to retrieve the next link of the desired path is too short to be measurable on our system. To retrieve a complete path, it takes less than 1 second on the average. The memory requirement is also kept to a small 10M bytes compared to 100M bytes required by the conventional path view approach. Our experimental results confirm that the *HEPV* approach represents an excellent compromise between the *compute-on-demand* and *precomputation* approaches for processing path queries.

Our proposed *HEPV* model successfully addresses the previously mentioned four path finding issues of the underlying navigation applications. First, by encoding path views, *HEPV* requires much less computation than the traditional A^* approach in processing path queries. Consequently, it is more suitable to handling a high volume of concurrent path queries. Second, the hierarchical path views encoded in *HEPV* can be updated in a shorter time than the nonhierarchical solutions; thus it can better capture the dynamic nature of the transportation network. Third, *HEPV*'s efficient optimal path retrieval guarantees that it can provide a near real-time path query response (<0.02 second on average for retrieving a long path in a network of 14,400 nodes). Lastly, *HEPV* adopts an encoded path view approach [3], which stores only the next hops of the shortest paths. Therefore, *HEPV* is most efficient in determining the next links of the optimal paths and fits very well with the query patterns of the underlying navigation applications.

This paper differs significantly from a preliminary report [25] in the following aspects;

1) we extend the two-level graph model to a general multilevel graph approach;
2) we present complete proofs of the optimality of the paths retrieved over the *HEPV*;
3) we propose a graph partitioning algorithm for the *HEPV* that now automates the partitioning step; and

4) we report comprehensive experimental results covering multilevel optimizations for real GIS as well as random graphs.

The remainder of this paper is organized as follows: In Section 2, we discuss background on the path view approach. Next, we show how a hierarchical graph is created in Section 3. In Section 4, we introduce both the task that encodes and the one that updates the hierarchical path views. The optimality theorems and the hierarchical path retrieval algorithms are presented in Section 5. Experimental results and analysis are given in Section 6 to compare the proposed *HEPV* approach to alternative path finding techniques. We provide related work in Section 7 and conclude with a summary of contributions as well as future work in Section 8.

2 Background on Encoded Path View

In this section, we review the basics of the *encoded path view* approach. For a more detailed treatment, see [3], [15].

2.1 Basic Graph Definitions

DEFINITION 1. *$G = (N, L, W)$ is a graph, where $N = \{N_i \mid 1 \le i \le n\}$ is the set of nodes with n the total number of nodes in the graph. $L = \{<N_i, N_j> \mid 1 \le i, j \le n\}$ is the set of links, where each link is a directed 2-tuple $<N_i, N_j>$, denoted simply as L_{ij}. $W = \{LW_{ij} \mid LW_{ij} = f_c(L_{ij})\}$, where LW_{ij} denotes the nonnegative link weight of L_{ij} and f_c is the cost function applied to the link L_{ij}.*

We call this graph a *flat graph* to distinguish it from the hierarchical graph, which we will define later.

DEFINITION 2. *A path $P_{ij} = <N_{k_1}, N_{k_2}, \cdots, N_{k_m}>$ in G is an ordered sequence of nodes, where $N_i = N_{k_1}$ and $N_j = N_{k_m}$, $N_{k_p} \in N, 1 \le p \le m$, and $L_{k_q k_{q+1}} \in L, 1 \le q \le m-1$. N_i, N_j are the source and destination nodes of the path, respectively. The path weight PW_{ij} is the sum of the link weights of all links on the path, i.e., $PW_{ij} = \sum_{p=1}^{m-1} LW_{k_q k_{q+1}}$. A shortest path SP_{ij} is the path from N_i to N_j in G that gives the minimum path weight PW_{ij} of all possible paths from N_i to N_j. The path weight of the shortest path SP_{ij} is denoted by SPW_{ij}. $SPW_{ij} = \infty$ if there is no path between N_i and N_j. $SPW_{ij} = 0$ if $i = j$.*

DEFINITION 3. *The transitive closure of a graph $G = (N, L, W)$ is defined by*

$$CLOSURE(G) = \{<N_i, N_j> \mid L_{ij} \in L \vee (\exists N_k \in N)\ (<N_i, N_k> \in CLOSURE(G) \wedge L_{kj} \in L)\}.$$

From the definition of transitive closure, if $<N_i, N_j> \in CLOSURE(G)$, then there exists a path P_{ij} in G. In other words, N_j is reachable from N_i.

2.2 Flat Path View (*FPV*)

A Flat Path View (*FPV*) stores the all-pair shortest paths for a given graph. Because storing all shortest paths in their entirety requires an unrealistically large amount of storage, *FPV* only stores the origin, destination, direct successor node (called next-hop), and the weight for a shortest path for this O-D pair [3], [15]. If there exist more than one shortest path between an O-D pair, each is stored for a different next-hop. This structure can well address the issue that only next-turn information about a shortest path is needed. An entire shortest path can be retrieved by iteratively looking up the path views using the next-hops as the keys. We implement *FPV* by associating with each source node a table such that each tuple stores the next-hop on a shortest path, together with the weight of this path, to a destination reachable from this source node. A tuple is uniquely identified by the combination of the destination and the next-hop.

DEFINITION 4. *Given a graph $G = (N, L, W)$, the encoded path table of node N_i is $EP_i = \{<j, k, SPW_{ij}> \mid \forall N_j, N_k \in N$ and N_k is the next-hop of a shortest path $SP_{ij}\}$, where $N_i \in N$.*

The flat path view corresponds to a set of tables of 3-tuples <*destination*, *nexthop*, *pathweight*> where each each table is associated with a source node. Below we give the formal definition of the *flat path view*.

DEFINITION 5. *The flat path view FPV of a flat graph $G = (N, L, W)$ is the set of encoded path tables of all the nodes in G, i.e., $FPV = \{EP_i \mid N_i \in N\}$.*

Fig. 2 shows the *FPV* of an example graph. The shortest path from N_a to N_e is $SP_{ae} = \{N_a, N_b, N_c, N_e\}$ and its path weight $SPW_{ae} = 9$. Thus a tuple <e, b, 9> is in the EP_a table associated with N_a.

In this paper, we use a variation of the well-known Dijkstra algorithm [9] to generate the path views. However, our solution is general, and any other shortest path algorithms [1], [3], [4], [15] could also be utilized.

2.3 Discussion

The computation of the *FPV*, equal to calculating all-pair shortest paths, is computationally expensive,[1] and the space requirement for the *FPV* is also high.[2] For large maps, computing or updating the *FPV* may take a long time, therefore can only be performed with longer, possibly unacceptable, delays. Our experiments (see Section 6.2) show that the *FPV* encoding time for a graph of 3,600 nodes is as long as 250 seconds on a Sun SPARC-20 workstation. Furthermore, the space requirement for the path view becomes so large that to efficiently cache data in memory becomes less cost effective, forcing more usage of secondary storage, which further slows down the encoding process. In this paper we explore alternative solutions for coping effectively with

1. The computational complexity of all-pair Dijkstra algorithm is $O(n^2 log(n) + en) = O(n^2 log(n) + dn^2)$, with n the number of nodes, e the number of links, and d the average out degree [8]. The ITS networks are sparse graphs with low out degree, hence the complexity is $O(n^2 log(n))$.

2. The space requirement is $O(n^2)$ if every node is reachable from every other node.

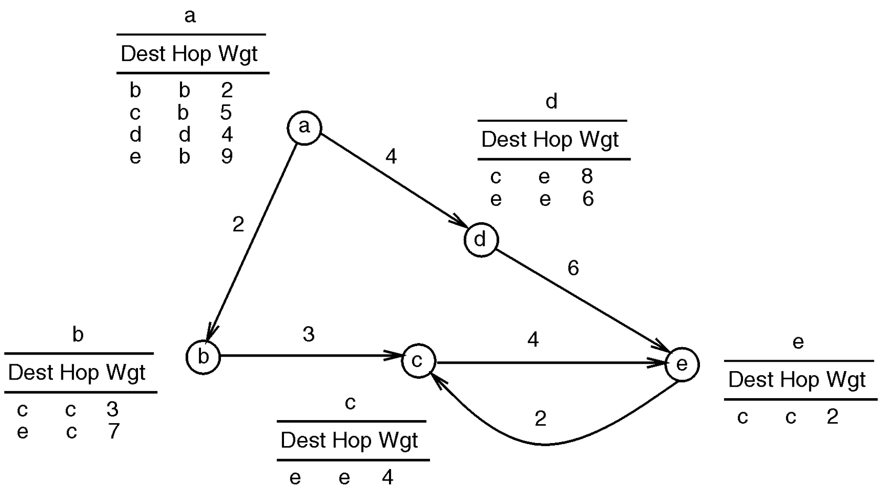


Fig. 2. Encoded path view structure.

large graph sizes in terms of both path encoding time and storage requirement.

3 CREATING A HIERARCHICAL GRAPH

We [16], [18] as well as other researchers [14], [26], [32] have proposed hierarchical graph models before. However, typically they are not designed to guarantee optimality of the retrieved path. In contrast, we introduce the *HEPV* model below and show that it is optimal.

3.1 Creating the Hierarchy through Fragmentation

The *HEPV* generates a hierarchical graph from a flat graph based on fragmentation. It then pushes up all border nodes, the nodes that belong to more than one fragment, to generate a map at the next higher level. For most maps of interest (such as GIS maps, computer network topology maps), the number of border nodes generated by an effective fragmentation is far smaller than that of regular nodes (explained in Section 3.2). A higher-level map consists of only border nodes. Therefore it is a much more compact graph which represents all cross-partition points on the map at the level below. Since all shortest paths spanning across more than one fragment must traverse some cross-fragment (border) nodes, the map at the next higher level can be used to capture the possible connections between border nodes at the current level. Because a higher-level map is much smaller than a lower-level one, path retrieval efficiency can be greatly improved by utilizing the path information associated with the higher-level map.

Fig. 3 is an illustration of this concept. The level-0 graph in Fig. 3 is the original flat map that consists of all nodes (grey dots). A fragmentation of 32 partitions is applied on level-0 graph, and all border nodes at this level are also classified as the regular nodes at the next higher level (level-1). Continuing the processing, we create four partitions at level-1 and generate the level-2 graph with the border nodes at level-1. No further fragmentation is necessary at level-2 because the number of nodes at this level is small.

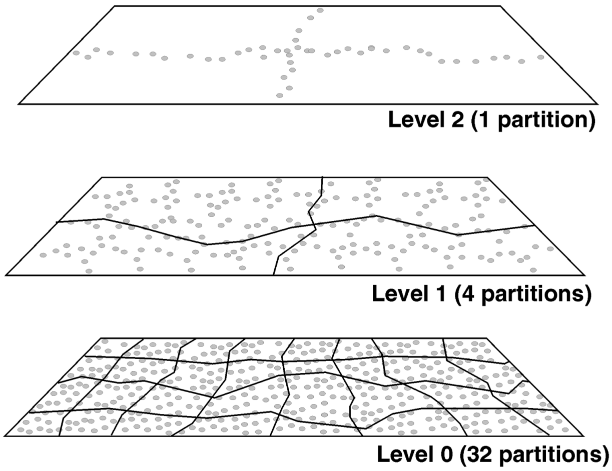


Fig. 3. Creating the hierarchy through fragmentation.

3.2 Effective Fragmentation

The effectiveness of a hierarchical graph is measured both by the overall efficiency in precomputation of *FPV*s for all fragments and by the compactness of the nonground-level maps. For the former, it is easy to prove by induction that the minimum total precomputation cost is achieved if all fragments at the same level have the same number of nodes.[3] To achieve the latter, we conducted a series of experimental evaluation.

We first experimented with the optimal decomposition algorithm (in minimizing the number of border nodes) proposed in [30] and found that its exponential time complexity makes it unrealistic for large maps. We next tested a more efficient suboptimal algorithm [30] which resulted in an excessive number of border nodes. We also experimented with a center-based greedy algorithm [13] and found that, in order to achieve approximately equally sized partitions, it relies on manually picking "good" center nodes. This manual intervention is unacceptable for large

3. The proof is trivial, so we omit it here.

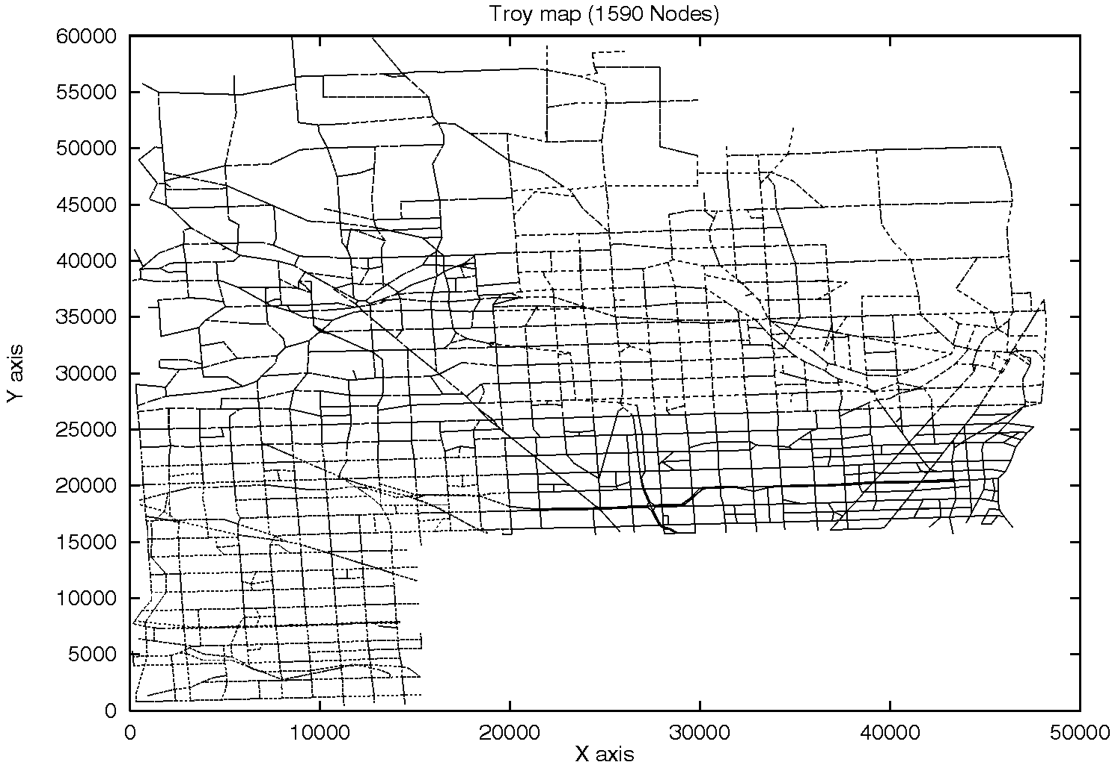


Fig. 4. A partition of the ITS graph by partition algorithm.

networks and for maps at the nonground level where no clues of picking good center nodes are available.

To achieve an effective fragmentation, we therefore developed a novel partitioning algorithm called spatial partitioning which clusters graph links into partitions based on spatial proximity [19]. Spatial partitioning takes advantage of ITS map characteristics such as the grid-like (near-planar) patterns, and the relatively short distance for the majority of links. Our experimental evaluation comparing this approach to alternative algorithms from the literature [19] shows that this partition works very well for GIS data sets (Fig. 4). A detailed discussion of the spatial partitioning algorithm can be found in [19].

3.3 Hierarchical Graph Model

We now present a formal model that shows how the hierarchy is created based on the above fragmentation method. We first extend the notations from Section 2.1 to hierarchical graphs:

$L_G(i, j)$—the link from N_i to N_j in graph G.
$LW_G(i, j)$—the link weight of $L_G(i, j)$.
$P_G(i, j)$—a path from N_i to N_j in graph G.
$SP_G(i, j)$—the shortest path from N_i to N_j in graph G.
$SPW_G(i, j)$—the shortest path weight of $SP_G(i, j)$.

DEFINITION 6. *A fragment* $G^f = (N^f, L^f, W^f)$ *of a graph* $G = (N, L, W)$ *is a graph, where* $N^f \subseteq N$, $L^f \subseteq L$, *and* $W^f = \{LW^f_{ij} \mid L^f_{ij} \in L^f \wedge L^f_{ij} = L_{ij} \wedge LW^f_{ij} = LW_{ij}\}$. *If* $L_{ij} \in L^f$, *then* $N_i \in N^f$ *and* $N_j \in N^f$.

A fragment of G is a subgraph of G that consists solely of a subset of nodes and links of G. Note that it is not required that all links between two nodes must be in one fragment, even if the two nodes are.

DEFINITION 7. *A partition* $P = \{G^f_1, G^f_2, \cdots, G^f_p\}$ *of* G, *denoted by* $PARTITION(G)$, *is a set of fragments of* G, *with the fragments satisfying the following three requirements*:

***Requirement 1*:** $N^f_1 \cup N^f_2 \cup \cdots \cup N^f_p = N$.

***Requirement 2*:** $L^f_1 \cup L^f_2 \cup \cdots \cup L^f_p = L$.

***Requirement 3*:** *For any two fragments* $G^f_u = (N^f_u, L^f_u, W^f_u)$ *and* $G^f_v = (N^f_v, L^f_v, W^f_v)$ *of* G, *where* $1 \le u, v \le p$ and $u \ne v$, *the following holds:* $L^f_u \cap L^f_v = \emptyset$.

In Definition 7, the first two requirements state that the fragments as a whole must contain all nodes and links of G. In other words, the fragments completely cover the graph. The second and third requirements state that each link of G belongs to exactly one and only one fragment, i.e., the partition is minimal with respect to the set of links of G. The nodes of different fragments may, however, overlap. If the nodes of a link belong to two fragments, the link can belong to only one of the two fragments. Fig. 5b depicts a partition of the flat graph from Fig. 5a.

DEFINITION 8. *Given a partition* $P = \{G^f_1, G^f_2, \cdots, G^f_p\}$ *of graph* G, *the node intersection of a fragment* G^f_u *with all other fragments of* P_G *together is called the border node set for* G^f_u. $BORDER(G^f_u) = N^f_u \cap \left(\bigcup_{v=1 \wedge v \ne u}^{p} N^f_v\right)$, *where* $1 \le u$

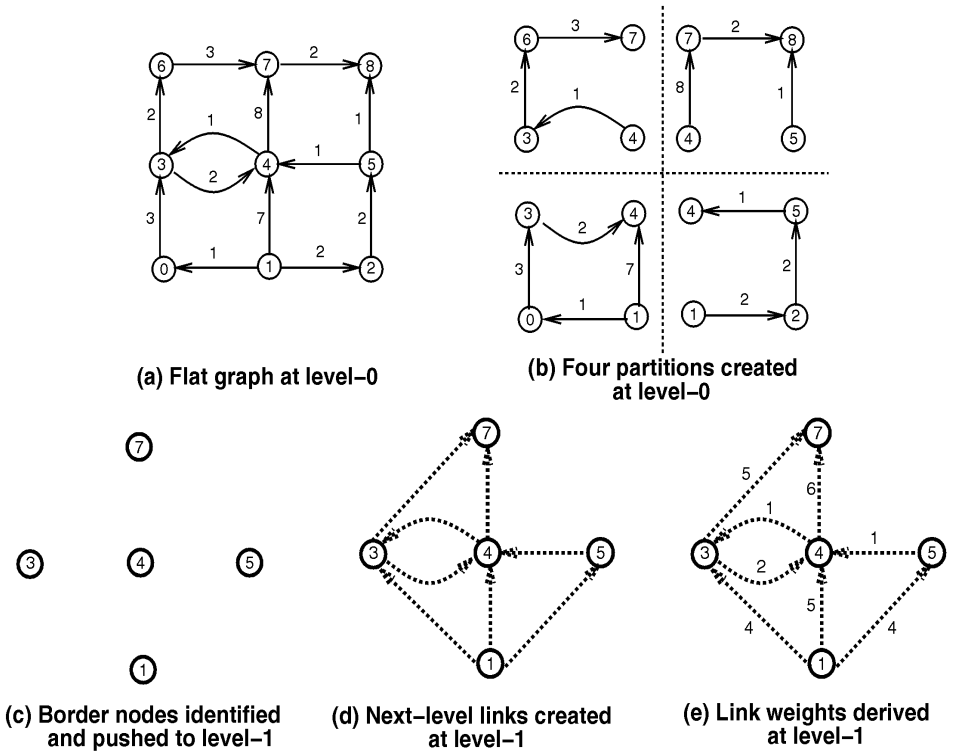


Fig. 5. An example of creating a two-level hierarchical graph from a flat graph.

$\le p$. If $BORDER(G_u^f) \cap BORDER(G_v^f) \ne \emptyset$ with $1 \le u, v \le p$ and $u \ne v$, then the fragments G_u^f and G_v^f are said to be adjacent. Conversely, the remaining node set of G_u^f is defined by $LOCAL(G_u^f) = N_u^f - BORDER(G_u^f)$.

A border node of a fragment appears in at least one other fragment of the same partition, while a local node appears in exactly one fragment only. Two fragments are said to be adjacent if they have at least one common border node. In Fig. 5b, $BORDER(G_1^f) = \{N_1, N_3, N_4\}$ because N_1 and N_4 also appear in G_2^f, N_3 also appears in G_3^f. $LOCAL(G_1^f) = \{N_0\}$ since N_0 is only in G_1^f.

DEFINITION 9. *Given $P = \{G_1^f, G_2^f, \cdots, G_p^f\}$ a partition of G, then a supergraph $G^s = (N^s, L^s, W^s)$ is defined by*:

1) $N^s = \bigcup_{u=1}^{p} BORDER(G_u^f)$.
2) $L^s = \{L_{ij} \mid (N_i, N_j \in N^s) \wedge (\exists u)((1 \le u \le p) \wedge (N_i, N_j \in N_u^f) \wedge (<N_i, N_j> \in CLOSURE(G_u^f))\}$.
3) *For any link $L_{ij} \in L^s$, its link weight $LW_{G^s}(i, j) = MIN(\{SPW_{G_u^f}(i, j) \mid (1 \le u \le p) \wedge (<N_i, N_j> \in CLOSURE(G_u^f))\})$, where MIN is the minimum function.*

Intuitively, the supergraph of a partition P consists of all the border nodes, i.e., cross-partition nodes. If there exists a pair of border nodes N_i and N_j in a fragment and N_j is reachable from N_i via a path entirely within that fragment, there is a link from N_i to N_j in the supergraph G^s. Its link weight is the minimum path weight of the shortest paths among all the fragments which have paths from N_i to N_j within each individual fragment. We use $SUPER(P)$ to denote the supergraph of a partition P.

For example, Fig. 5a is a flat graph which is divided into four fragments (Fig. 5b). In Fig. 5c, border nodes are identified and are used to form the supergraph (Fig. 5d). In Fig. 5e, $L_{G^s}(1, 3) \in L^s$ because there is a path $P_{G_1^f}(1, 3) = \{N_1, N_0, N_3\}$. $L_{G^s}(5, 7) \notin L^s$ because there is no path contained entirely in a single fragment. $LW_{G^s}(1, 4) = 5$ since there are two shortest paths in two fragments: $SP_{G_1^f}(1, 4) = \{N_1, N_0, N_3, N_4\}$ with $SPW_{G_1^f}(1, 4) = 6$, and $SP_{G_2^f}(1, 4) = \{N_1, N_2, N_5, N_4\}$ with $SPW_{G_2^f}(1, 4) = 5$. We choose the one with minimum shortest path weight.

DEFINITION 10. *A hierarchical graph of a (flat) graph G is defined by $G^{hier} = \{P^0, P^1, \cdots, P^n\}$, where*

$$P^0 = PARTITION(G),$$
$$P^1 = PARTITION(G^1) \text{ with } G^1 = SUPER(P^0),$$

and so on, until,

$$P^n = \{G^n\} \text{ with } G^n = SUPER(P^{n-1}).$$

This hierarchical graph G^{hier} is said to be an n-level hierarchical graph.

ALGORITHM *CreateAHierarchicalGraph*(G, n)
// **Create the hierarchical graph** $G^{hier} = \{P^0, P^1, ..., P^n\}$ **from the flat graph** G.
01 $P^0 := GraphPartition(G);$ // $P^0 = \{G_1^0, G_2^0, \cdots, G_{p_0}^0\}$.
02 **for all** $1 \leq l \leq n$
03 $N^l := \bigcup_{u=1}^{|p^{l-1}|} BORDER\left(G_u^{l-1}\right);$
04 $L^l := \emptyset;$
05 **for all** $1 \leq u \leq p_{l-1}$
06 **for all** $(N_i, N_j \in BORDER(G_u^{l-1})) \wedge N_i \neq N_j$
07 f $<N_i, N_j> \in CLOSURE(G_u^{l-1}) \wedge <N_i, N_j> \notin L^l$
08 $^l := L^l \cup \{<N_i, N_j>\};$
09 **endif endfor endfor**
10 $P^l := GraphPartition(G^l);$ // $P^l = \{G_1^l, G_2^l, \cdots, G_{p_l}^l\}$.
11 **endfor**

Fig. 6. The hierarchical graph creation algorithm.

Thus, the hierarchical graph is a multilevel set of graphs. The graphs at the bottom level are fragments of a partition P^0 of the flat graph. A supergraph at l level, say G^l, is created over a partition P^{l-1} at $l - 1$ level. By partitioning this supergraph G^l, the graphs at $l + 1$ level are next constructed. The graph at the top level is a trivial partition (one fragment) of the supergraph at the next lower level. The set of graph fragments at level l can be represented by $P^l = \{G_1^l, G_2^l, \cdots, G_{p_l}^l\}$, where $0 \leq l \leq n$ and $p_l \geq 1$. G_u^l is a fragment of P^l with $1 \leq u \leq p_l$. We call G_u^l the uth graph at lth hierarchical level. Fig. 3 can be viewed as an abstract representation of the hierarchical graph model with three levels. In Section 3.4, we present an algorithm that creates a hierarchical graph from a flat graph based on our hierarchical graph model.

3.4 An Algorithm to Create the Hierarchical Graph

Given a flat graph, we can create a hierarchical graph by the algorithm outlined in Fig. 6. In this algorithm, the hierarchical graph G^{hier} is built bottom-up (line 2). First, a partition P^0 of the flat graph G is created by the graph partition procedure *GraphPartition* (line 1). The supergraph G^l contains all the border nodes of graphs at level $l - 1$ (line 3). For any border node pair in each fragment (line 6), if there is a path between these two nodes (line 7), then there is a link between these two border nodes in the supergraph (line 8). After the supergraph G^l is generated, it is again partitioned to create fragments at level l (line 10). This process continues until a small enough supergraph is generated that required no further partitioning. This algorithm follows the definition of the hierarchical graph given in Definition 10.

4 Encoding and Updating the Hierarchical Path Views (*HPV*)

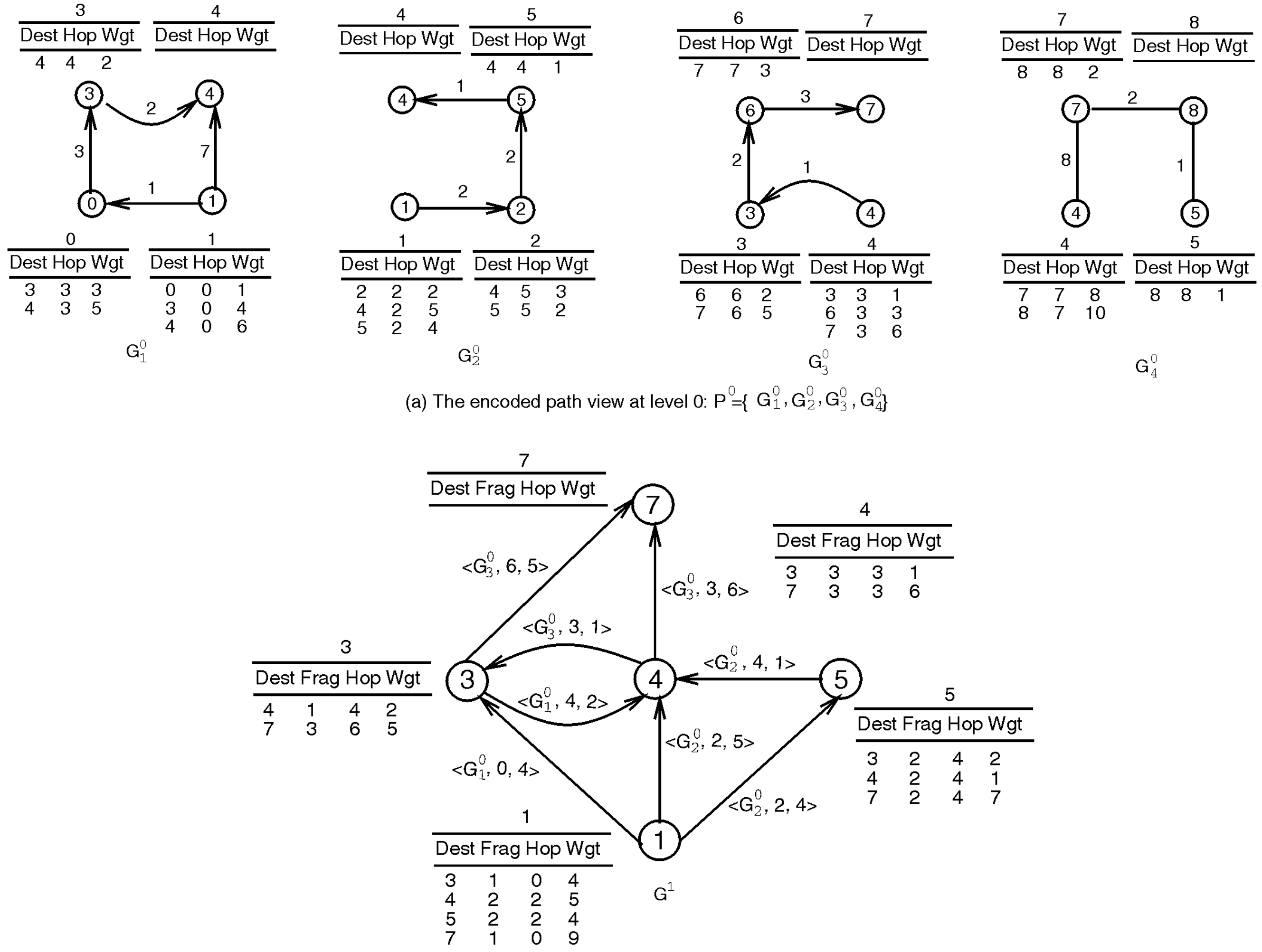
After the hierarchical graph is created (Section 3), *HEPV* generates a *FPV* for each fragment at all levels by precomputing all-pair shortest paths within this fragment. This collection of *FPVs* across all levels in the hierarchy is called the Hierarchical Path Views (*HPV*).[4] Note that the creation of the hierarchical graph and the initial precomputation of *HPV* are off-line tasks (see Fig. 1) for which the costs they incur can be considered as one-time only. In order to achieve a high accuracy of the information stored in *HPV*, they must be updated as frequently as possible if their underlying link weights (e.g., link travel time) have changed. The *HPV* update cost therefore is a on-line dynamic cost that should be minimized because the more efficiently a *HPV* can be updated, the more frequent the update task can be performed. Frequent updates therefore result in an overall more accurate *HPV*.

4.1 Encoding the Hierarchical Path Views

Assume a hierarchical graph $G^{hier} = \{P^0, P^1, \cdots, P^n\}$ is created based on its corresponding flat graph. For each fragment graph at the ground level, G_u^0, we create and maintain a *FPV* (Section 2.2). For the graphs at higher levels, e.g., G_u^l with $0 < l \leq n$ and $1 \leq u \leq p_l$, each link between nodes N_i and N_j corresponds to a shortest path between N_i and N_j in at least one fragment at level $l - 1$ (Definition 9).

Each link at nonground levels is represented by the from-node, to-node, link weight, and two other pieces of information:

4. We use *HEPV* to represent the entire hierarchical path view model and *HPV* to represent the stored path views.



(b) The encoded path view at level 1: $P^1 = \{G^1\}$

Fig. 7. The *HPV* of the hierarchical graph.

1) the fragment ID of the fragment at the ground level, through which the path modeled by this link first crosses, and
2) the next-hop of this path in that respective fragment.

We call these two additional link attributes *fragmentID* and *nexthop*. For example, in Fig. 7b, the link $L_{G^1}(1, 3)$ is associated with three attributes $<G_1^0, 0, 4>$, which means that the corresponding path of this link from N_1 to N_3 is in the fragment G_1^0 of the flat graph, the next-hop of the flat path is N_0, and the link weight is $LW_{G^1}(1, 3) = 4$.

As each link models a path in the underlying flat graph on which the hierarchical graph is constructed, the *fragmentID* and *nexthop* of the link jointly identify the direct successor of this path. The reason for this kind of link representation is to optimize the path retrieval over the hierarchical graph, i.e., rather than having to recursively search for the next-hop by iterating down the hierarchical graph, we now have direct access to the next-hop from any level of the hierarchy.

For each node above ground levels, its *encoded path view* structure therefore corresponds to a table of 4-tuples <*destination*, *fragmentID*, *nexthop*, *pathweight*>. While the attributes *destination*, *nexthop*, and *pathweight* are the same as those in *FPV* (Definition 4), the *fragmentID* is used to identify the ground-level fragment of the *nexthop* of a shortest path, which is crucial for optimal path retrieval over *HEPV*.

DEFINITION 11. *Given a hierarchical graph* $G^{hier} = \{P^0, P^1, \cdots, P^n\}$, *the encoded path table of node* N_i *in* G_u^l *is defined by*:

1) *If* $l = 0$, *then*

$$EP_{G_u^0}(i) = \{<j, k, SPW_{G_u^0}(i, j)> \mid N_j, N_k \in G_u^0, N_k \text{ is the nexthop of } SP_{G_u^0}(i, j)\},$$

where $1 \le u \le p_0$ *and* $SPW_{G_u^0}(i, j)$ *is the shortest path weight with respect to* G_u^0.

ALGORITHM $EncodeHPV(G^{hier})$
// **Create the** HPV **for a hierarchical graph** G^{hier}.
01 **for all** $1 \le l \le n$
02 **for all** $1 \le u \le p_l$
03 **for all** $<N_i, N_j> \in L_{G_u^l}$
04 $LW_{G_u^l}(i, j) := \infty$;
05 **endfor endfor endfor**
06 **for all** $1 \le l \le n$
07 **for all** $1 \le u \le p_{l-1}$
08 $EncodeFragment(G_u^{l-1})$;
09 **for all** $(N_i, N_j \in BORDER(G_u^{l-1})) \wedge N_i \ne N_j$
10 **if** $<N_i, N_j> \in CLOSURE(G_u^{l-1})$
11 $v := GetFragment(l, <N_i, N_j>)$;
12 **if** $SPW_{G_u^{l-1}}(i, j) < LW_{G_v^l}(i, j)$
13 $LW_{G_v^l}(i, j) = SPW_{G_u^{l-1}}(i, j)$;
14 **endif endif endfor endfor endfor**
15 $EncodeFragment(G_1^n)$;

Fig. 8. The *HPV* encoding algorithm.

2) *If* $0 < l \le n$, *then*

$$EP_{G_u^l}(i) = \{<j, v, k, SPW_{G_u^l}(i, j)> \mid N_j \in G_u^l, N_k \in N_v^0,$$
$$N_k \text{ is the nexthop of the shortest path } SP_{G_u^l}(i, j)\},$$

where $1 \le u \le p_l$. *Note that* N_k *is a node at ground-level (the flat graph) which may not be at level* l.

For example, in Fig. 7b, the *encoded path view* structure of node N_1 in G^1 denoted by the tuple <7, 1, 0, 9> tells us that the shortest path weight $SPW_{G^1}(1, 7) = 9$, the next-hop of the shortest path $SP_{G^1}(1, 7)$ is the node N_0 in the fragment G_1^0.

DEFINITION 12. *Given a hierarchical graph* $G^{hier} = \{P^0, P^1, \cdots, P^n\}$, *for* $G_u^l \in P^l$ *with* $0 \le l \le n$ *and* $1 \le u \le p_l$, *its HPV is the set of encoded path tables for all the nodes in* G_u^l, *i.e.*, $PV_{G_u^l} = \{EP_{G_u^l}(i) \mid N_i \in G_u^l\}$. *The specific encoded path view at the* lth *level is denoted by* $PV^l = \{PV_{G_1^l}, PV_{G_2^l}, \cdots, PV_{G_{p_l}^l}\}$.

DEFINITION 13. *Given a hierarchical graph* $G^{hier} = \{P^0, P^1, \cdots, P^n\}$, *its hierarchical path view is* $HPV = \{PV^0, PV^1, \cdots, PV^n\}$.

In other words, the *HPV* of a hierarchical graph G^{hier} corresponds to a set of encoded path tables, each associated with a node, for all nodes at all levels. For example, the *HPV* in Fig. 7 corresponds to the set of all encoded path tables at level 0 and level 1.

The algorithm that encodes the *HPV* (called *EncodeHPV*) is presented in Fig. 8. The *EncodeHPV* algorithm calls two functions:

- $EncodeFragment(G)$ creates the *FPV* for fragment G by invoking an all-pair shortest path algorithm.
- $GetFragment(l, <N_i, N_j>)$ returns the fragment at level l to which the link $<N_i, N_j>$ belongs.

The *EncodeHPV* algorithm encodes the *HPV* for a hierarchical graph starting from the ground level upwards to the top-most level. At each level, it encodes all path views at this level by invoking an all-pair shortest path algorithm (line 8). While encoding a path view at this level, the *EncodeHPV* algorithm also initializes or updates the link weights at the next higher level. The lines 11-13 guarantee that the weight of a link $<N_i, N_j>$ at the next higher level equals to the minimum weight of all shortest paths from N_i to N_j that each is entirely contained in one fragment at this level (Definition 9). The values of the shortest path weights are obtained directly from the associated path views computed during *HPV* encoding.

4.2 Updating the Hierarchical Path Views

In the hierarchical graph, whenever a link weight of a fragment (i.e., of the flat graph) is changed, one or possibly several corresponding link weights of the graphs at a higher level of the hierarchical graph may also be affected. This would happen if the link, say $L_{G_u^l}(i, j)$, of graph G_u^l at level l no longer models the shortest path $SP_{G_u^l}(i, j)$ of G^l, where G^l is the supergraph of the partition P^{l-1} at level $l-1$. One important property of our *HEPV* structure is that

ALGORITHM *UpdateHPV*$(G^{hier},\ \{G^0_{v_1}, G^0_{v_2}, \cdots, G^0_{v_q}\})$
// Update the hierarchical graph $G^{hier} = \{P^0, P^1, \cdots, P^n\}$ **and its** *HPV*.
// $G^0_{v_1}, G^0_{v_1}, \cdots, G^0_{v_q}\}$ **are fragments which link weights have changed.** $\{G^0_{v_1}, G^0_{v_2}, \cdots, G^0_{v_q}\} \subseteq P^0$.

01 **for all** $G^0_v \in \{G^0_{v_1}, G^0_{v_2}, \cdots, G^0_{v_q}\}$
02 $\quad\quad$ *MarkFragment*$(G^0{}_v)$;
03 **endfor**
04 **for all** $0 \le l \le n$
05 $\quad$ **for all marked** $G^l_v \in P^l$
06 $\quad\quad$ *EncodeFragment*(G^l_v);
07 $\quad\quad$ *UnmarkFragment*(G^l_v);
08 $\quad\quad$ **for all** $(N_i, N_j \in BORDER(G^l_v) \wedge N_i \ne N_j$
09 $\quad\quad\quad$ **if** $<N_i, N_j> \in CLOSURE(G^l_v)$
10 $\quad\quad\quad\quad$ $u := GetFragment1(l + 1, <N_i, N_j>)$;
11 $\quad\quad\quad\quad$ $w := GetFragment2(L_{G^{1+1}_u}(i, j))$;
12 $\quad\quad\quad\quad$ **if** $v \ne w \wedge LW_{G^{l+1}_u}(i, j) \le SPW_{G^l_v}(i, j)$
13 $\quad\quad\quad\quad\quad$ **continue**;
14 $\quad\quad\quad\quad$ **endif**
15 $\quad\quad\quad\quad$ *MarkFragment*(G^{l+1}_u);
16 $\quad\quad\quad\quad$ $LW_{G^{l+1}_u}(i, j) := \infty$;
17 $\quad\quad\quad\quad$ **for all** $w \in \{w' \mid <N_i, N_j> \in CLOSURE(G^l_{w'}) \wedge 1 \le w' \le p^l\}$
18 $\quad\quad\quad\quad\quad$ if $SPW_{G^l_w}(i, j) < LW_{G^{l+1}_u}(i, j)$
19 $\quad\quad\quad\quad\quad\quad$ $LW_{G^{l+1}_u}(i, j) := SPW_{G^l_w}(i, j)$;
20 $\quad\quad\quad\quad\quad$ **endif endfor endif endfor endfor endfor**

Fig. 9. The *HPV* update algorithm.

we only need to update the *HPV* of the affected graphs at each hierarchical level, while other unaffected graphs are not touched. In the *FPV* approach, instead, the whole *path view* is typically affected for a strongly connected graph such as transportation data sets, even if only few links are changed [15]. We present the update algorithm in Fig. 9 which incrementally updates the hierarchical graph as well as the *HPV* when link weights of some fragments change.

The *UpdateHPV* algorithm calls the following functions:

- *MarkFragment*(G) marks a fragment G which needs to be updated.
- *UnmarkFragment*(G) unmarks a fragment G after it has been updated.
- *EncodeFragment*(G) creates the *encoded path view* of fragment G by some all-pair shortest path algorithms.
- *GetFragment1*$(l, <N_i, N_j>)$ returns the fragment at level l which contains the link $<N_i, N_j>$.
- *GetFragment2*$(L_{G^{1+1}_u}(i, j))$ returns the fragment, say v, at level l whose shortest path weight $SPW_{G^l_v}(i, j)$ corresponds to the link weight $LW_{G^{l+1}_u}(i, j)$, i.e., $SPW_{G^l_v}(i, j) = LW_{G^{l+1}_u}(i, j)$.

The *UpdateHPV* algorithm first marks all the ground-level fragments that contain links whose weights have changed (lines 1 to 3). It then updates the *HPV* starting from the ground-level upwards until the top-most level is reached. At level l, the *UPdateHPV* algorithm first reencodes the path views for all marked fragments (lines 5 and 6). For every connected border node pair (lines 8 and 9), if the shortest path weight that contributes to the weight of its associated link at the next higher level has changed, or another shortest path weight corresponding to the same next-level link has become smaller than the weight of this link (lines 10-12), then the weight of this link must be updated according to Definition 9 (lines 17-19). The fragment to which this link belong at the next level must also be marked for *HPV* update (line 15) in order for the path views at that level to be updated properly (lines 5 and 6) at the next iteration.

5 Optimal Path Retrieval in *HEPV*

In this section, we first present optimality theorems which assures that a path query issued in the *HEPV* system with more than one level gives the same optimal path as computed on the original graph (one-level *HEPV*). Based on the

HEPV structure and the optimality theorems, we then develop the hierarchical path retrieval algorithms.

5.1 Optimality of the Retrieved Hierarchical Paths

The first optimality theorem shows that the shortest path weight computed on a supergraph G^s corresponds to the actual shortest path weight of the graph G on which the partition of the supergraph is based on.

THEOREM 1. *Let $P = \{G_1^f, G_2^f, \cdots, G_p^f\}$ be a partition of graph G and G^s be the supergraph of partition P. For any node pair $N_a, N_b \in G^s$, we have $SPW_{G^s}(a, b) = SPW_G(a, b)$.*

PROOF. Prove by contradiction.

Suppose $SPW_{G^s}(a, b) \neq SPW_G(a, b)$. By the definition of the supergraph (Definition 9), each link of G^s corresponds to the shortest path from N_a to N_b in the context of a fragment, and every fragment is a subgraph of the flat graph G (Definition 6), thus it must be that $SPW_{G^s}(a, b) > SPW_G(a, b)$.

For any path $P_G(a, b)$, all the border nodes on it as well as the two end nodes can be represented sequentially by the node sequence N_a, $N_{i_1}, \cdots, N_{i_m}, N_b$. Hence, its path weight can be represented by $PW_G(a, b) = PW_G(a, i_1) + PW_G(i_1, i_2) + \cdots + PW_G(i_m, b)$. By simple reasoning (called the principle of optimality in [7]), the shortest path weight $SPW_G(a, b)$ can be denoted by:

$$SPW_G(a, b) = SPW_G(a, i_1) + SPW_G(i_1, i_2) + \cdots + SPW_G(i_m, b). \tag{1}$$

In the border node sequence, every two successive border nodes correspond to the border node entering a fragment and the border node leaving that fragment. Therefore every successive pair of nodes in the border node sequence $N_a, N_{i_1}, \cdots, N_{i_m}, N_b$ belong to the same fragment, say $N_a, N_{i_1} \in N_{j_1}^f$; $N_{i_1}, N_{i_1} \in N_{j_2}^f$; $\cdots$; $N_{i_m}, N_b \in N_{j_m}^f$ with $1 \le j_1, j_2, \cdots, j_m \le p$.

As the shortest path $SP_G(a, i_1)$ consists of only nodes from fragment $G_{j_1}^f$, we would have $SPW_G(a, i_1) = SPW_{G_{j_1}^f}(a, i_1)$. For a similar reason, we have $SPW_G(i_1, i_2) = SPW_{G_{j_2}^f}(i_1, i_2)$, and so on, until $SPW_G(i_m, b) = SPW_{G_{j_m}^f}(i_m, b)$.

Thus we can rewrite $SPW_G(a, b)$ as $SPW_G(a, b) = SPW_{G_{j_1}^f}(a, i_1) + SPW_{G_{j_2}^f}(i_1, i_2) + \cdots + SPW_{G_{j_m}^f}(i_m, b)$.

By the definition of the supergraph (Definition 9), the link weight $LW_{G^s}(a, i_1)$ is equal to the minimum of the shortest path weights of the shortest paths among all the fragments which have a path from N_a to N_{i_1}, thus: $SPW_{G_{j_1}^f}(a, i_1) \ge LW_{G^s}(a, i_1)$. Similarly, we have $SPW_{G_{j_2}^f}(i_1, i_2) \ge LW_{G^s} i_1, i_2)$, and so on, until $SPW_{G_{j_m}^f}(i_m, b) \ge LW_G^s(i_m, b)$.

But we know $SPW_G(a, b) = SPW_G(a, i_1) + SPW_G(i_1, i_2) + \cdots + SPW_G(i_m, b)$ (Equation 1), therefore $SPW_G(a, b) \ge LW_{G^s}(a, i_1) + LW_{G^s}(i_1, i_2) + \cdots + LW_{G^s}(i_m, b)$.

From the assumption $SPW_{G^s}(a, b) > SPW_G(a, b)$, it results $SPW_{G^s}(a, b) > LW_{G^s}(a, i_1) + LW_{G^s}(i_1, i_2) + \cdots + LW_{G^s}(i_m, b)$.

The above formula says that there exists a path from N_a to N_b in the supergraph G^s that results in a path which has a smaller path weight than the shortest path weight from N_a to N_b in the supergraph G^s. This is a contradiction. Thus we have proved the theorem. □

As the hierarchical graph is defined recursively by partitioning and building supergraphs, Theorem 1 can be applied to the hierarchical graph which results in the following corollaries.

COROLLARY 1. *Let $G^{hier} = \{P^0, P^1, \cdots, P^n\}$ be a hierarchical graph. For any node pair $N_a, N_b \in G^1$ with G^1 the supergraph of P^0, we have $SPW_{G^1}(a, b) = SPW_G(a, b)$.*

COROLLARY 2. *Let $G^{hier} = \{P^0, P^1, \cdots, P^n\}$ be a hierarchical graph. For any node pair $N_a, N_b \in G^l$ with $0 < l \le n$ and G^l the supergraph of P^{l-1}, we have $SPW_{G^l}(a, b) = SPW_{G^{l-1}}(a, b)$.*

Corollary 1 is the direct extension of Theorem 1 to the hierarchical situation. It is trivial to prove Corollary 2 if we treat G^{l-1} as a flat graph, P^{l-1} as its partition, and G^l as the supergraph of P^{l-1}.

COROLLARY 3. *Let $G^{hier} = \{P^0, P^1, \cdots, P^n\}$ be a hierarchical graph. For any node pair $N_a, N_b \in G^l$ with $l \ge 1$ and G^l the supergraph of P^{l-1}, we have $SPW_{G^l}(a, b) = SPW_G(a, b)$.*

It is straightforward to prove Corollary 3 by induction from Corollary 1 and Corollary 2. We thus have established that the shortest path computed at the supergraph at any level of the hierarchy corresponds to the actual shortest path of the flat graph, i.e., it is globally minimal.

COROLLARY 4. *Let $G^{hier} = \{P^0, P^1, \cdots, P^n\}$ be a hierarchical graph. For any node pair $N_a, N_b \in G^n$ with G^n the supergraph of P^{n-1}, we have $SPW_{G^n}(a, b) = SPW_G(a, b)$.*

Corollary 4 is the special case of Corollary 3 with $l = n$. It states that the shortest path weight computed on the graph G^n at the top level corresponds to the actual shortest path weight of the flat graph G on which the hierarchical graph G^{hier} is constructed.

Theorem 2 presents a method to compute the shortest path from the partition and supergraph if both source and destination nodes N_a and N_b are in the same fragment (N_a and N_b may not necessarily be nodes in the supergraph). The shortest path weight is the minimum of the two:

1) the local shortest path weight of the fragment, and
2) the minimum of all possible concatenated shortest paths composed of three segments: from N_a to a border node N_i, from N_i to another border node N_j, and from N_j to N_b.

THEOREM 2. *Let $P = \{G_1^f, G_2^f, \cdots, G_p^f\}$ be a partition of graph G and G^s be the supergraph of P. For $N_a, N_b \in LOCAL(G_u^f)$, where $1 \le u \le p$, the following holds*:

$$SPW_G(a, b) =$$
$$MIN(SPW_{G_u^f}(a, b), MIN(\{SPW_{G_u^f}(a, i) + SPW_{G^s}(i, j) +$$
$$SPW_{G_u^f}(j, b) \mid (N_i, N_j \in BORDER(G_u^f)) \wedge N_i \ne N_j\})).$$

PROOF. All paths from N_a to N_b in G can be classified into two categories:

Case 1: The path consists solely of links from fragment G_u^f.

Case 2: The path consists of links from fragment G_u^f as well as other fragments.

For paths of Case 1, the path weight from N_a to N_b is denoted by $PW_{G_u^f}(a, b)$. If all paths from N_a to N_b are of this kind, then we know that the shortest path from N_a to N_b is also defined in G_u^f, i.e., $SPW_G(a, b) = SPW_{G_u^f}(a, b)$. This means that the shortest path can be computed using the local fragment information.

Any path of Case 2 consists of links of G which can be represented by the node sequence N_a, N_{i_1}, N_{i_1}, $\cdots$, N_{i_m}, N_b. As this path contains links of other fragments, it would leave this fragment G_u^f and ultimately come back. Therefore the path contains at least two border nodes N_i and N_j, such that N_i, $N_j \in BORDER(G_u^f)$ and $N_i \ne N_j$. Choose N_i to be the first such node and N_j to be the last such node on this path. The path weight of this particular path can be denoted by $PW_G(a, b) = PW_G(a, i) + PW_G(i, j) + PW_G(j, b)$.

Assume the node sequence of the shortest path is $N_a, \cdots, N_i, \cdots, N_j, \cdots, N_b$. By the principle of optimality [7], we have: $SPW_G(a, b) = SPW_G(a, i) + SPW_G(i, j) + SPW_G(j, b)$.

As the shortest path $SP_G(a, i)$ consists only of links of fragment G_u^f, it falls into Case 1. Thus we have $SPW_G(a, i) = SPW_{G_u^f}(a, i)$. Similarly, we have $SPW_G(j, b) = SPW_{G_u^f}(j, b)$.

As for $SPW_G(i, j)$, by Theorem 1 we have $SPW_G(i, j) = SPW_{G^s}(i, j)$.

Therefore, it follows that the shortest path weight from N_a to N_b in the context of the flat graph G is given by $SPW_G(a, b) = SPW_{G_u^f}(a, i) + SPW_{G^s}(i, j) + SPW_{G_u^f}(j, b)$.

From this equation, we know that the shortest path $SP_G(a, b)$ is in the following path set:

$$\{SP_{G_u^f}(a, i) + SP_{G^s}(i, j) +$$
$$SP_{G_u^f}(j, b) \mid (N_i, N_j \in BORDER(G_u^f)) \wedge N_i \ne N_j\}.$$

Thus the shortest path weight $SPW_G(a, b)$ can be represented by

$$SPW_G(a, b) =$$
$$MIN(\{SPW_{G_u^f}(a, i) + SPW_{G^s}(i, j) +$$
$$SPW_{G_u^f}(j, b) \mid (N_i, N_j \in BORDER(G_u^f)) \wedge N_i \ne N_j\}).$$

By combining Case 1 and Case 2, we have

$$SPW_G(a, b) =$$
$$MIN(SPW_{G_u^f}(a, b), MIN(\{SPW_{G_u^f}(a, i) + SPW_{G^s}(i, j) +$$
$$SPW_{G_u^f}(j, b) \mid (N_i, N_j \in BORDER(G_u^f)) \wedge N_i \ne N_j\})). \quad \square$$

By extending Theorem 2 to the hierarchical graph, we have the following corollaries.

COROLLARY 5. *Let $G^{hier} = \{P^0, P^1, \cdots, P^n\}$ be a hierarchical graph, where $P^0 = \{G_1^0, G_2^0, \cdots, G_{p_0}^0\}$ and $p_0 > 1$. For $N_a, N_b \in LOCAL(G_u^0)$, where $1 \le u \le p_0$, the following holds*:

$$SPW_G(a, b) =$$
$$MIN(SPW_{G_u^0}(a, b), MIN(\{SPW_{G_u^0}(a, i) + SPW_{G^1}(i, j) +$$
$$SPW_{G_u^0}(j, b) \mid (N_i, N_j \in BORDER(G_u^0)) \wedge N_i \ne N_j\})).$$

COROLLARY 6. *Let $G^{hier} = \{P^0, P^1, \cdots, P^n\}$ be a hierarchical graph, where $P^l = \{G_1^l, G_2^l, \cdots, G_{p_l}^l\}$, $0 < l < n$, and $p_l > 1$. For N_a, $N_b \in LOCAL(G_u^l)$, where $1 \le u \le p_l$, the following holds*:

$$SPW_{G^l}(a, b) =$$
$$MIN(SPW_{G_u^l}(a, b), MIN(\{SPW_{G_u^l}(a, i) + SPW_{G^{l+1}}(i, j) +$$
$$SPW_{G_u^l}(j, b) \mid (N_i, N_j \in BORDER(G_u^l)) \wedge N_i \ne N_j\})).$$

Theorem 3 presents a method to compute the shortest path weight over the partition and supergraph if source and destination nodes are in different fragments. The weight of the shortest path from node N_a to node N_b corresponds to the minimum of all possible concatenated shortest paths composed of three segments: from N_a to a border node N_i, from N_i to another border node N_j, and from N_j to N_b.

THEOREM 3. *Let $P = \{G_1^f, G_2^f, \cdots, G_p^f\}$ be a partition of graph G and G^s be the supergraph of P. For $N_a \in N_u^f$, $N_b \in N_v^f$, where $1 \le u, v \le p$ and $u \ne v$. Then*:

$$SPW_G(a,b) = MIN(\{SPW_{G_u^f}(a, i) + SPW_{G^s}(i, j) + SPW_{G_v^f}(j, b) \mid N_i \in BORDER(G_u^f) \wedge N_j \in BORDER(G_v^f)\}).$$

PROOF. Any path from N_a to N_b in G can be represented by the node sequence N_a, N_{i_1}, N_{i_1}, $\cdots$, N_{i_m}, N_b. It is obvious that two nodes N_i, N_j must exist in this path where $N_i \in BORDER(G_u^f)$ and $N_j \in BORDER(G_v^f)$ since the path must ultimately leave G_u^f and enter G_v^f.[5] Choose N_i to be the first such node and N_j to be the last such node (it is possible that $N_i = N_j$ if fragments G_u^f and G_v^f are adjacent). The path weight of the flat graph G is given by $PW_G(a, b) = PW_G(a, i) + PW_G(i, j) + PW_G(j, b)$.

Assume the node sequence of the shortest path from N_a to N_b in G is $N_a, \cdots, N_i, \cdots, N_j, \cdots, N_b$. By the principle of optimality [7], we have: $SPW_G(a, b) = SPW_G(a, i) + SPW_G(i, j) + SPW_G(j, b)$.

As the shortest path $SP_G(a, i)$ consists only of nodes from fragment G_u^f, and the shortest path $SP_G(j, b)$ consists only of nodes from fragment G_v^f, we have: $SPW_G(a, i) = SPW_{G_u^f}(a, i)$ and $SPW_G(j, b) = SPW_{G_v^f}(j, b)$.

As the nodes $N_i, N_j \in N^s$, by Theorem 1, we have $SPW_G(i, j) = SPW_{G^s}(i, j)$.

Thus, the shortest path weight $SPW_G(a, b)$ can be represented by

$$SPW_G(a, b) = SPW_{G_u^f}(a, i) + SPW_{G^s}(a, b) + SPW_{G_v^f}(j, b).$$

From the above equation, we know that the shortest path $SP_G(a, b)$ is in the following path set

$$\{SP_{G_u^f}(a, i) + SP_{G^s}(i, j) + SP_{G_v^f}(j,b) \mid N_i \in BORDER(G^f_u) \wedge N_j \in BORDER(G_v^f)\}.$$

Accordingly, the shortest path weight $SPW_G(a, b)$ can be represented by

$$SPW_G(a, b) = MIN(\{SPW_{G_u^f}(a, i) + SPW_{G^s}(i, j) + SPW_{G_v^f}(j, b) \mid N_i \in BORDER(G_u^f \wedge N_j \in BORDER(G_v^f\}). \quad \square$$

Again we get the following corollaries by extending Theorem 3 to the hierarchical graph. First, we can easily substitute P^0 and G^1 for the arbitrary partition P and the resulting supergraph G^s for some graph G.

COROLLARY 7. *Let $G^{hier} = \{P^0, P^1, \cdots, P^n\}$ be a hierarchical graph, where $P^0 = \{G_1^0, G_2^0, \cdots, G_{p_0}^0\}$ and $p_0 > 1$. For $N_a \in N_u^0$, $N_b \in N_v^0$, where $1 \le u, v \le p_0$ and $u \ne v$. The following holds*:

$$SPW_G(a, b) = MIN(\{SPW_{G_u^0}(a, i) + SPW_{G^1}(i, j) + SPW_{G_v^0}(j, b) \mid N_i \in BORDER(G_u^0) \wedge N_j \in BORDER(G_v^0)\}).$$

Next, we apply Theorem 3 to a fragment at level l of a hierarchical graph.

COROLLARY 8. *Let $G^{hier} = \{P^0, P^1, \cdots, P^n\}$ be a hierarchical graph, where $P^l = \{G_1^l, G_2^l, \cdots, G_{p_l}^l\}$, $0 < l < n$, and $p_l > 1$. For $N_a \in N_u^l$, $N_b \in N_v^l$, where $1 \le u, v \le p_l$ and $u \ne v$. The following holds*:

$$SPW_{G^l}(a, b) = MIN(\{SPW_{G_u^l}(a, i) + SPW_{G_u^{l+1}}(i, j) + SPW_{G_v^l}(j, b) \mid N_i \in BORDER(G_u^l) \wedge N_j \in BORDER(G_v^l)\}).$$

These corollaries state how to compute the optimal shortest path weight from the hierarchical graph. In fact, given a source and destination node pair, we can recursively use these corollaries to calculate the shortest path weight. We will discuss the shortest path retrieval algorithms in detail in the next section.

5.2 Path Retrieval Over *HEPV*

Based on the structure of hierarchical path views (*HPVs*) (Section 4) and the optimality theorems (Section 5.1), we now present the ***S**hortest **P**ath **R**etrieval **A**lgorithm* (*SPR)* in Fig. 10.

To retrieve the shortest path weight $SPW_G(a, b)$ and its nexthop N_c in the *HEPV* system, the algorithm (Fig. 10) first checks the subpaths combined by all border node pairs, each of which consists of border nodes from fragments to which N_a and N_b belong (line 2), respectively. The function $SSPR(N_i, N_j, 1)$ (***S**upergraph **S**hortest **P**ath **R**etrieval*) retrieves the shortest path weight $SPW_{G^1}(i, j)$ of the supergraph G^1 and the nexthop N_c of the path $SP_{G^1}(i, j)$ (line 3). By Corollary 7, the shortest path weight $SPW_G(a, b)$ of the flat graph G is the minimum of all concatenated path weights (lines 4

5. If $N_a \in BORDER(G_u^f)$, then $N_a = N_i$. If $N_b \in BORDER(G_v^f)$, then $N_b = N_j$. It is also possible that the two fragments G_u^f and G_v^f are adjacent, in which case $N_i = N_j$.

ALGORITHM $SPR(N_a, N_b)$: **return** N_c, $SPW_G(a, b)$
// **Retrieve the shortest path weight** $SPW_G(a, b)$ **and the nexthop** N_c
// **of the shortest path** $SP_G(a, b)$ **over the hierarchical graph** $G^{hier} = \{P^0, P^1, \ldots, P^n\}$,
// **where** $P^0 = \{G_1^0, G_2^0, \ldots, G_{p_0}^0 \{0\}, N_a \in G_u^0, N_b \in G_v^0$, **and** $1 \le u, v \le p_0$.

01 $SPW_G(a, b) := \infty;$
02 **for all** $N_i \in BORDER(G_u^0) \wedge N_j \in BORDER(G_v^0)$
03 $(N_c, SPW_{G^1}(i, j)) := SSPR(N_i, N_j, 1);$
04 **if** $(SPW_{G_u^0}(a, i) + SPW_{G^1}(i, j) + SPW_{G_v^0}(j, b)) < SPW_G(a, b)$ **then**
05 $SPW_G(a, b) := SPW_{G_u^0}(a, i) + SPW_{G^1}(i, j) + SPW_{G_v^0}(j, b);$
06 **if** $N_a \neq N_i$ **then**
07 $N_c := EP_{G_u^0}(a, i).nexthop;$
08 **endif endif endfor**
09 **if** $u = v \wedge SPW_{G_u^0}(a, b) < SPW_G(a, b)$ **then**
10 $N_c = :EP_{G_u^0}(a, b).nexthop;$
11 $SPW_G(a, b) := SPW_{G_u^0}(a, b);$
12 **endif**

Fig. 10. The shortest path retrieval algorithm *SPR*.

and 5). In the case of the source and destination nodes being in the same fragment (line 9), by Corollary 5, we also need to check the local shortest path $SP_{G_u^0}(a, b)$ (lines 9 to 11). In other words, this algorithm is a direct implementation of the path retrieval strategies shown by Corollary 5 and Corollary 7.

In Fig. 11, the function $SSPR(N_a, N_b, l)$ retrieves the shortest path weight $SPW_{G^l}(a, b)$ and the nexthop of the supergraph G^l at level l. If $l = n$, the $SPW_{G^n}(a, b)$ is retrieved directly from the *encoded path view* of G^n (lines 1, 2, and 16). Otherwise by Corollary 6, we need to compare all the paths concatenated by border nodes of involved fragments (line 5) and find the path with minimum path weight (lines 6 to 8). In case of the source and destination nodes in the same fragment, by Corollary 8 we also need to check the local shortest path (lines 12 to 14). This function is recursively called (line 6) until $l = n$ (line 1).

The correctness of the shortest path retrieved by *SPR* algorithm is guaranteed by the optimality theorems. The algorithm can be optimized if the source and destination nodes of the query appear in the top graph G^n of the hierarchy. By Corollary 4, the query can be answered directly from the *encoded path view* of graph G^n. For the same reason, the recursive function *SSPR* can be optimized if the source and destination nodes, say N_a and N_b, of the query also appear in the same graph at a level, say l. By Corollary 3, $SPW_{G^l}(a, b) = SPW_G(a, b)$, thus we can start to retrieve the shortest path at level l, which eliminates having to call the *SSPR* function recursively at lower levels.

Although the retrieval of the shortest path in *HEPV* with more than one level is less efficient than the *FPV* approach, a simple look-up of the *FPV*, due to calling the recursive function *SSPR* and comparing the concatenated local and global paths, our experiments (see Section 6) show that it is still significantly faster than the compute-on-the-fly approach. The algorithm only needs to compare the local shortest paths concatenated by border nodes of two involved fragments. Given an appropriate partition of a flat graph, the number of border nodes of a fragment is much smaller than the total number of nodes of that fragment.

Note that the performance of the *SPR* algorithm is *independent* of the number of nodes on the shortest path, while some other algorithms such as A^* are not. This makes *SPR* preferable over other algorithms for long path retrievals. However, the computational complexity of the *SPR* algorithm, an exhaustive search over all possible paths, is exponential in the number of hierarchical levels. Given an l level hierarchical graph G^{hier}, let us assume for simplicity that each fragment has the same number of border nodes B. To retrieve the shortest path weight $SPW_G(a, b)$, the *SPR* algorithm needs to compare all the B^2 possible concatenated local paths at level 0 and the supergraph paths at level 1 (line 2 in *SPR* algorithm); to calculate the shortest path $SPW_{G^1}(i, j)$ of the supergraph at level 1, the *SSPR* function again needs to compare all the B^2 possible concatenated

FUNCTION SSPR(N_a, N_b, l): **return** N_c, $SPW_{G^l}(a, b)$

// **Retrieve the shortest path weight** $SPW_{G^l}(a, b)$ **and the nexthop** N_c

// **of the shortest path** $SP_{G_u^l}(a, b)$ **over the hierarchical graph** $Ghier = \{P^0, P^1, \ldots, P^n)$,

// **where** $P^l = \{G_1^l, G_2^l, \ldots, G_{p_l}^l$, $1 \le l \le n$, $p_l \ge 1$, $N_a \in G_u^l$, $N_b \in G_v^l$, and $1 \le u, v \bullet p_l$.

01 **if** $l = n$ **then**

02 $\quad\quad N_c := EP_{G_u^l}(a, b).nexthop;$

03 **else**

04 $\quad\quad SPW_{G^l}(a, b) := \infty;$

05 $\quad\quad$ **for all** $N_i \in BORDER(G_u^l) \wedge N_j \in BORDER(G_v^l)$

06 $\quad\quad\quad\quad (N_c, SPW_{G_u^{l+1}}(i, j)) := SSPR(N_i, N_j, l + 1);$

07 $\quad\quad\quad\quad$ if $(SPW_{G_u^l}(a, i) + SPW_{G_u^{l+1}}(i, j) + SPW_{G_v^l}(j, b)) < SPW_{G^l}(a, b)$ then

08 $\quad\quad\quad\quad\quad\quad SPW_{G^l}(a, b) := SPW_{G_u^l}(a, i) + SPW_{G_u^{l+1}}(i, j) + SPW_{G_v^l}(j, b);$

09 $\quad\quad\quad\quad\quad\quad$ **if** $N_a \ne N_i$ **then**

10 $\quad\quad\quad\quad\quad\quad\quad\quad N_c := EP_{G_u^l}(a, i).nexthop;$

11 $\quad\quad\quad\quad\quad\quad$ **endif endif endfor**

12 $\quad\quad$ **if** $u = v \wedge SPW_{G_u^l}(a, b) < SPW_{G^l}(a, b)$ **then**

13 $\quad\quad\quad\quad N_c := EP_{G_u^l}(a, b).nexthop;$

14 $\quad\quad\quad\quad SPW_{G^l}(a, b) := SPW_{G_u^l}(a, b);$

15 $\quad\quad$ **endif enfif**

16 **return** $(N_c, SPW_{G^l}(a, b));\}$

Fig. 11. The supergraph shortest path retrieval function *SSPR*.

local paths at level 1 and the supergraph paths at level 2 (line 5 in *SSPR* function), and so on, until the *SSPR* function retrieves the shortest path weight from the top graph of the hierarchy. Therefore, the computational complexity of *SPR* algorithm is $O((B^2)^{l-1})$, with l the number of hierarchical levels. This indicates that there is a balance between the number of levels versus the number of fragments in each level that define the hierarchical graph. Our empirical evaluation in Section 6 confirms this analysis.

6 Experimental Evaluation

We now present a comprehensive set of experiments evaluating of performance of *HEPV*. We used three kinds of graphs:

Grid graph: The synthetic grid graphs corresponds to grid patterns with randomly assigned link weights. We use grid graphs for experiments because, since they lend themselves nicely to controlling graph characteristics, such as the sizes and the numbers of fragments.

Random graph: The random graph is a more general graph. Each node is randomly located within a prespecified area and is randomly assigned a degree. A link is created between two randomly picked nodes. The random graphs we have generated are strongly connected graphs, i.e., every node is reachable from every other node.

Real map: We had available on-line street maps of Troy County and the surrounding areas in suburban Detroit. We used this real data set to verify the validity of the other two kinds of synthetic graphs.

All experiments are conducted on a Sun SPARC-20 workstation with 128MB main memory. Every experiment was repeated 10 times and the results presented here correspond to an average over these 10 runs. Because out test computer is a dedicated machine not used by others, and our experiments are always conducted during night while no other tasks are being processed, machine variations therefore were minimized. For path retrieval tests, the 10 runs are based on results of 10 different paths. Result variations between the 10 runs are caused by the length difference between the 10 paths tested. The comparative results are very similar in terms of their relative performance for all 10 runs (no outliers).

Although we have conducted experiments up to four levels, we found that three levels are sufficient for the sizes of the maps we are interested in although a hierarchy of four levels can be advantageous for very large network

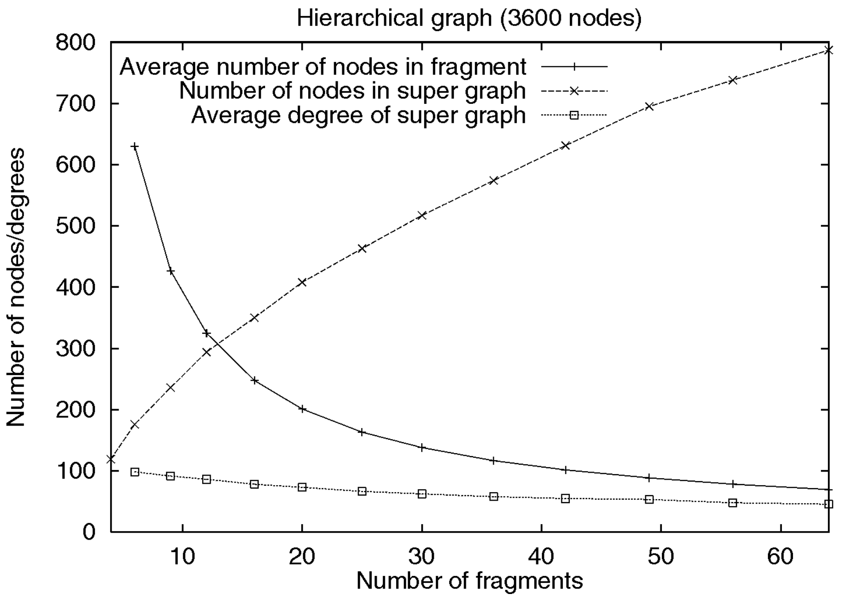


Fig. 12. Features of the hierarchical graph.

(>10,000 nodes). In this section, we only present experimental results based on hierarchies up to three levels.

6.1 Characteristics of the Hierarchical Graph

The purpose of this first experiment is to show the relationship between features of the hierarchical graph and the number of fragments. This then lays the foundation for understanding the behavior of *HEPV* with respect to retrieval, encoding and storage, and thus interpreting our experiments in the following sections. In the following, we thus vary the number of fragments in a partition and measure the effects on the hierarchical graph.

For a $n = m \times m$ grid graph, we evenly divide it into $p = f \times f$ fragments. On average, each fragment has n/p nodes. The total number of border nodes[6] is $2m(f-1) \approx 2\sqrt{np}$. On each side of a fragment, there are at most $\sqrt{n}/\sqrt{p} = \sqrt{n/p}$ border nodes. Thus a fragment can have up to $4\sqrt{n/p}$ border nodes. Because every border node can have links to all the other border nodes in the supergraph that belong to the same fragment, the average node degree is approximately $4\sqrt{n/p}$.

Fig. 12 compares three features of the hierarchical graph: the number of nodes in a fragment versus the number of fragments, the number of nodes in the supergraph versus the number of fragments, and the average node degree of the supergraph versus the number of fragments. With the increase of the number of fragments, the fragment size goes down, but the supergraph size goes up. The average degree of the supergraph decreases slowly. These three curves match the above analysis.

Based on these characteristics, when the number of fragments increases we expect the computational and memory cost of the hierarchical graph to

1) decrease for local individual fragments, and
2) increase for the supergraph. Our experiments in the following sections analyze this trade-off.

6.2 Encoding the *HPV*

In this experiment, our goal is to compare the performance gain achieved for path encoding due to our hierarchical graph model. In the following, we use the Dijkstra algorithm [9] to do the encoding, though any other all-pair shortest path algorithm would work similarly. The computational complexity of all-pair shortest paths by the Dijkstra algorithm is $O(n^2 log(n))$ for an ITS graph.

For a two-level hierarchical graph, we partition a flat grid graph of n nodes into p fragments of the same size. From the analysis of Section 6.1, each fragment has n/p nodes and the supergraph has $2\sqrt{np}$ nodes. Consequently, the encoding time for all the fragments is $p * c_1 (n/p)^2 log(n/p) = c_1 n^2 log(n/p)/p$ with c_1 a constant. The encoding time for the supergraph is $c_2 (2\sqrt{np})^2 log(2\sqrt{np}) = c_3 nplog(2\sqrt{np})$ with c_2 and c_3 constants. Thus the encoding time of the *HPV* is $c_1 n^2 log(n/p)/p + c_3 nplog(2\sqrt{np})$. The encoding time of the *FPV* is $c_4 n^2 log(n)$ with c_4 a constant. It is expected that the *HPV* encoding time of the hierarchical graph is less than the *FPV* encoding time of the corresponding flat graph for sufficiently large n and $1 < p < n$. Our experiments on real and grid maps confirmed this.

6.2.1 Encoding Time of HPV Versus Graph Size

Fig. 13 shows the results of experiments on grid, random, and real ITS graphs. We find that the encoding time of the *HPV* (two levels, four fragments at level 0) is smaller than the encoding time of the *FPV* for all three types of graphs. The encoding time of the real graph is very close to the encoding time of the grid graph for the same graph size. This confirms that the transportation network is similar to a grid graph. With the increase of the graph size, the encoding

6. For simplicity, we assume $f >> 1$, which will not affect our discussion.

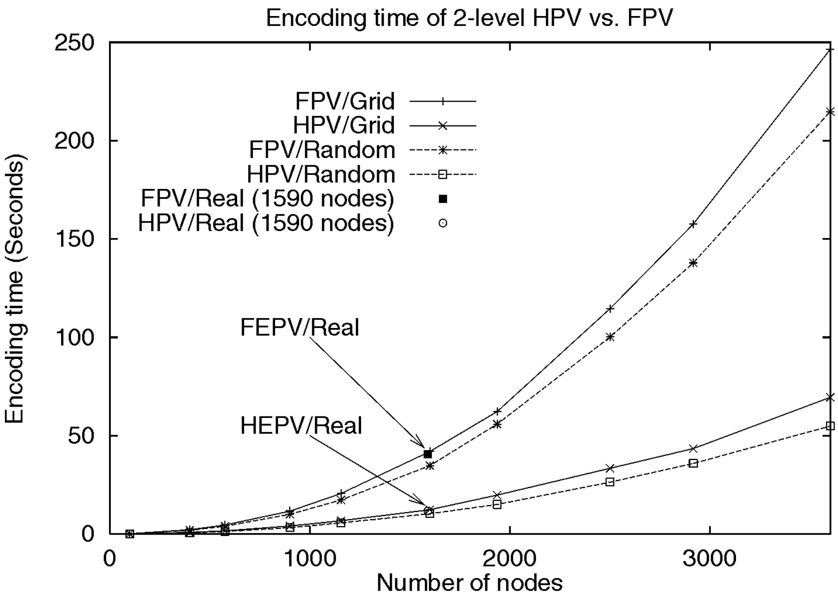


Fig. 13. Encoding two-level *HPV* versus *FPV*.

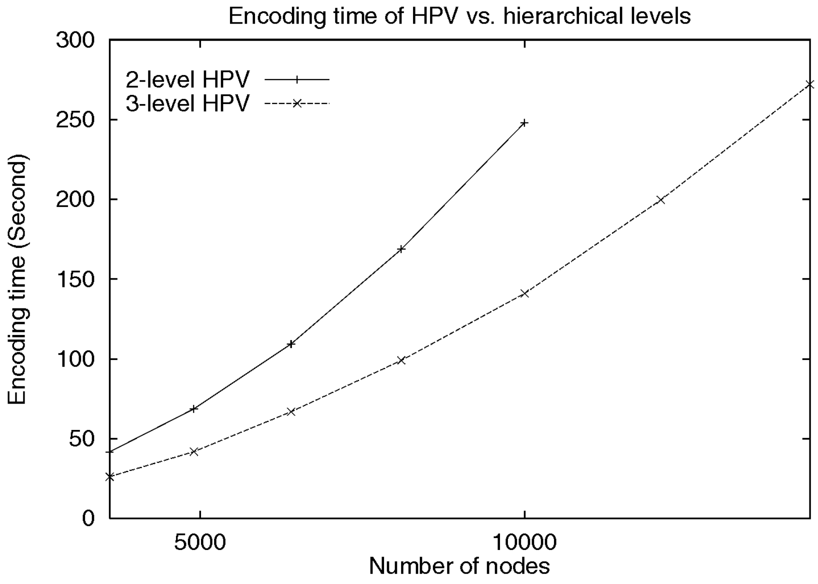


Fig. 14. Encoding *HPV* versus hierarchical levels.

time of the *FPV* increases sharply, while the encoding time of the *HPV* increases relative slowly. Our experimental results thus confirm our analytical evaluation and clearly demonstrate the superiority of our proposed *HPV* approach over the *FPV* approach.

With the increase of graph size, the encoding time of the two-level *HPV* also increases. To optimize the encoding time of *HEVP* for large graphs, we could create hierarchical graphs of more than two levels. In Fig. 14, we measure the *HPV* encoding time of hierarchical graphs with two and three levels. The two-level *HPV* has 16 fragments at level 0, while the three-level *HPV* has 36 fragments at level 0 and four fragments at level 1. The results in Fig. 14 indicate that, for large graphs (>3,000 node), three-level *HPV* is more efficient to encode than two-level *HPV*.

6.2.2 Encoding Time of HPV Versus Number of Fragments

We have analytically determined in Section 6.2 that the encoding time for all fragments is $c_1 n^2 log(n/p)/p$ and the encoding time for the supergraph is $c_3\ np\ log(2\sqrt{np})$. Increasing the number of fragments p will cause the encoding time for all the fragments to go down since the fragments become smaller. However, the encoding time for the supergraph now increases because it becomes larger. We are interested in determining the optimal number of fragments that minimizes the total encoding cost. Fig. 15 shows our experimental results of measuring the encoding time for two-level *HPV* as we vary the number of fragments. The optimal number of fragments for a grid graph of 3,600

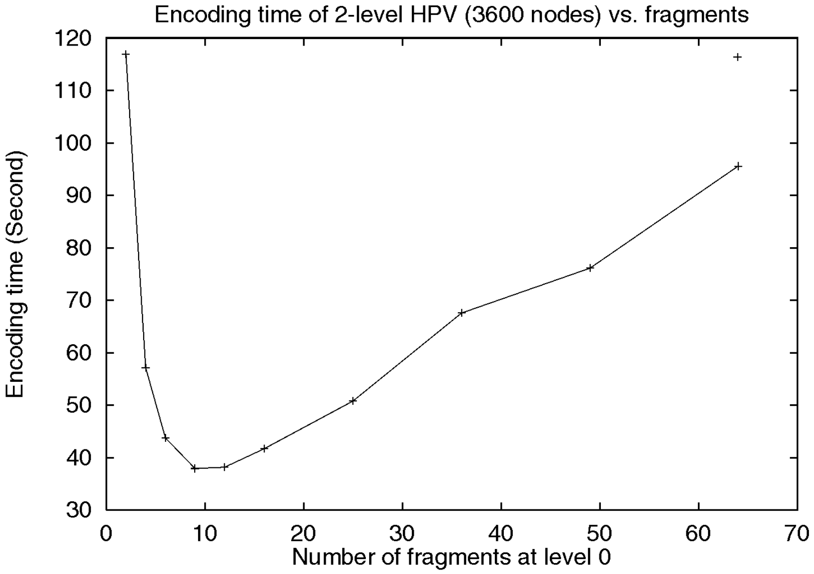


Fig. 15. Encoding time of two-level *HPV* versus number of fragments.

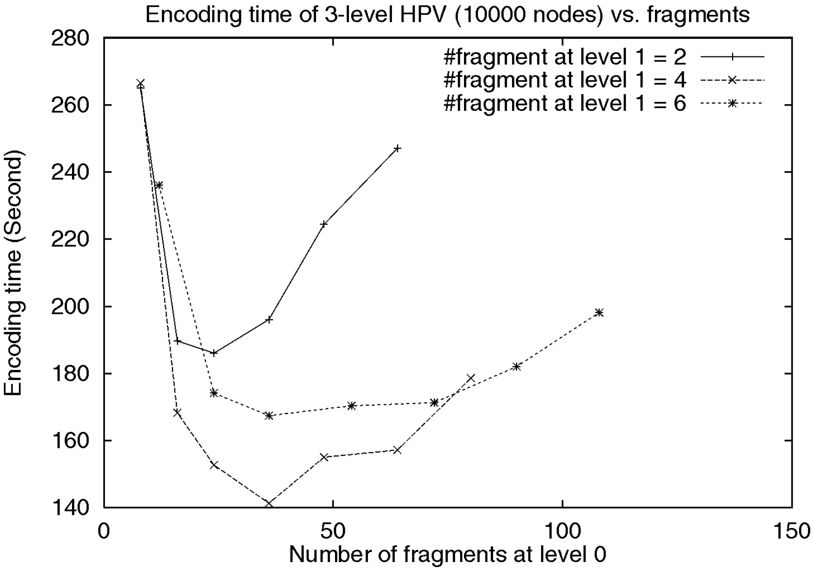


Fig. 16. Encoding time of three-level *HPV* versus number of fragments.

nodes is around 10. This indicates that it is important to run such a test when building a path finding system for a particular data set to guarantee optimal setup.

In Fig. 16, we measure the encoding time of the three-level *HPV* for a graph of 10,000 nodes. We vary the number of fragments at level 0 from 16 to 108 and set the number of fragments at level 1 to 2, 4, and 6. The results (Fig. 16) show that the optimal setup is when the number of level-0 fragments is set to 36 and the number of level-1 fragment is set to 4.

6.3 Updating the *HPV*

Three steps are needed to incrementally update the *HPV* of a two-level hierarchical graph: first, re-encoding the changed fragments; second, updating the affected links of the supergraph; and third, reencoding the supergraph. Therefore, the cost of updating the *HPV* is $i * C_e^f + C_u^s + C_e^s$, where $i \geq 1$ is the number of fragments affected by an update, C_e^f is the cost of reencoding a fragment assuming all fragments are of same size, C_u^s is the cost of updating the supergraph, and C_e^s is the cost of reencoding the supergraph. The cost function thus is linear with respect to the number of changed fragments i.

Our experiments with two-level *HPV* (Fig. 17) match our analytical results. This experiment also indicates that the influence of changing a link weight is limited to the confines of the fragment in which the link is contained and, if necessary, propagation of the change to the supergraph. All other fragments are unaffected. For the flat graph, the

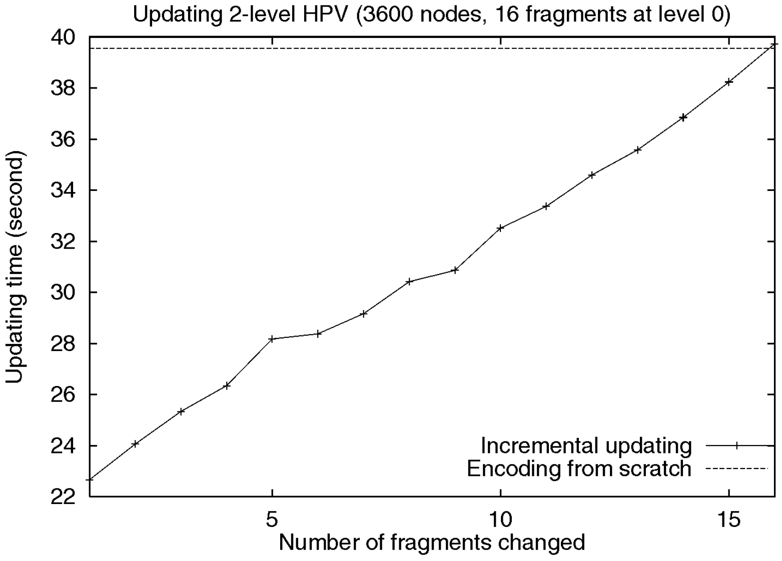


Fig. 17. Updating time of two-level *HPV*.

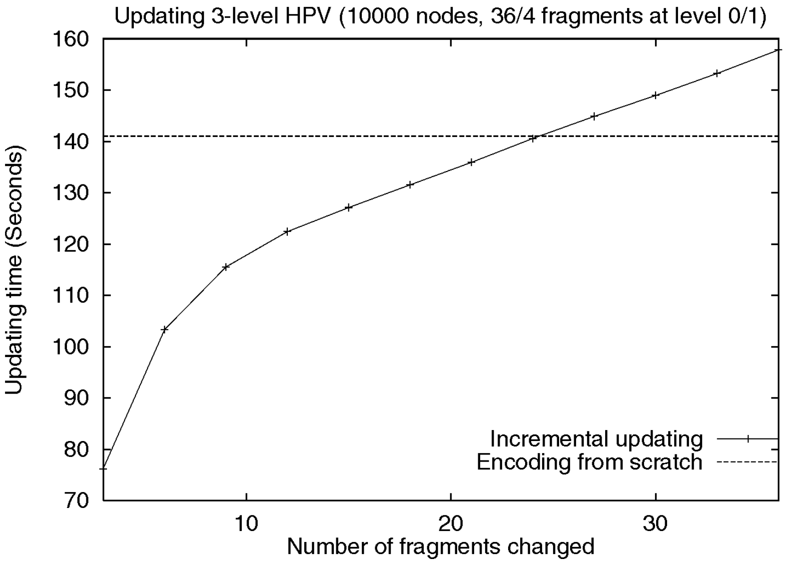


Fig. 18. Updating time of three-level *HEPV*.

whole graph is typically affected even if only few links are changed [15].

For a hierarchical grid graph of 3,600 nodes and 16 fragments, the encoding time of one fragment is around 1 second, whereas the encoding time of the supergraph is more than 22 seconds. If all the fragments are changed, the cost of incrementally updating the *HPV* is very close to the cost of encoding it from scratch. This is reasonable because in this extreme case both of them have to do approximately the same amount of work.

Fig. 18 shows the experimental results of updating a three-level *HPV*. The updating time goes up sharply for less than 12 fragments updated. This is because more and more fragments at higher levels are affected due to the increased number of fragments updated at lower levels. The *HPV* updating time rises less sharply when more than 12 fragments are updated at level 0. This can be explained by the fact that for more than 12 fragments updated at level 0, all the fragments at higher levels are affected, and the updating time for any one fragment at level 0 is a constant. Fig. 18 also indicates that for more than 24 fragments updated, the *HPV* updating time exceeds its encoding time. Since the *HPV* incremental updating algorithm (Fig. 9) needs to identify the possibly affected fragments and to propagate the update to the next higher level for every updated fragment at each level, more updating costs are incurred than encoding from scratch if many fragments are updated. Thus, for more than 24 fragments updated in our experiment, we prefer to re-encode the *HPV* rather than to incrementally update each of them.

6.4 Memory Requirement

Now we study the *HPV* memory requirements of the *HEPV* model as compared to the flat graph. Given a flat graph *G*

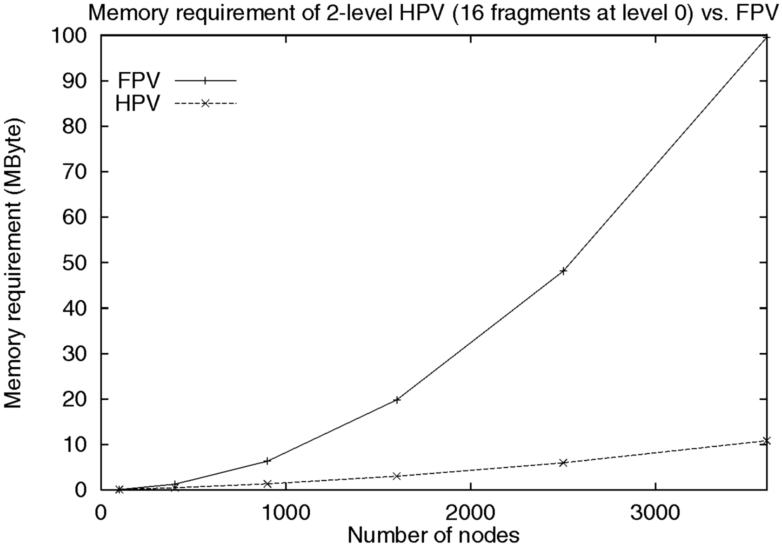


Fig. 19. Memory requirement: *HPV* versus *FPV*.

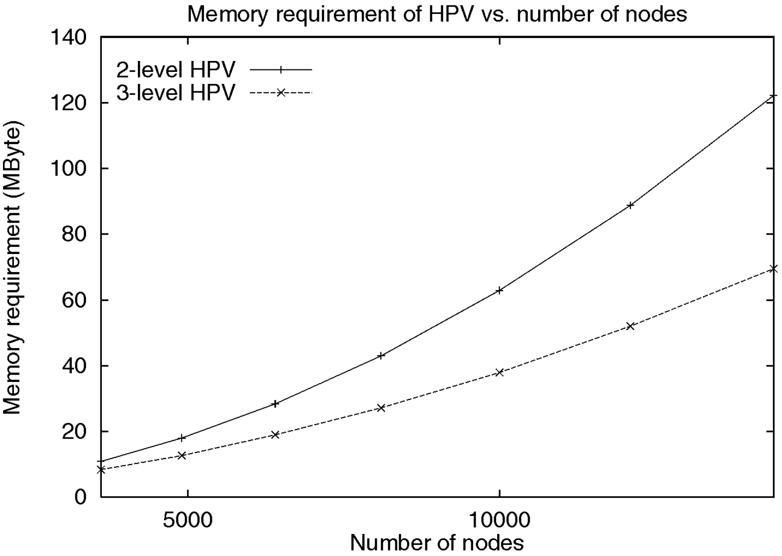


Fig. 20. Memory requirement of *HPV* versus hierarchical levels.

of n nodes and a partition of G into p fragments of similar size, each fragment would have approximately n/p nodes and the supergraph would have $2\sqrt{np}$ nodes (Section 6.1). Assume we use a matrix to store all-pair shortest path weight tables of our encoded path structure. The memory required to store *FPV*, all-pair shortest path weight over the flat graph, is c_1n^2 with c_1 a small constant. However, the memory required to store all-pair shortest path weight for all the fragments is $p * c_2(n/p)^2 = c_2n^2/p$ with c_2 a small constant. The memory required for encoding the supergraph is $c_3\,(2\sqrt{np})^2 = c_4np$, where c_3 and c_4 are small constants. Thus the total memory requirement of a two-level *HPV* is $c_2n^2/p + c_4np$, which is less than the memory requirement of *FPV* for some p.

In Fig. 19, we compare the memory requirements of *HPV* with *FPV*. The memory requirement of *FPV* increases sharply with the increase of the number of nodes, which is in accordance with our above analysis that the memory requirement of the *FPV* is proportional to the square of the number of nodes. The memory requirement of *HPV* increases slowly with the increase of the number of nodes, which confirms that the *HPV* requires less memory than *FPV*.

Fig. 20 shows the memory requirements of *HPV* with two and three hierarchical levels. For large graphs, three-level *HPV* requires less memory than two-level *HPV*.

6.5 Path Retrieval Performance Evaluation

Compared to the flat graph, the savings we gain from the *HEPV* model include a smaller path view encoding time (Section 6.2), a smaller path view updating time (Section 6.3), and also smaller memory requirements to store path views (Section 6.4). However, the price of these benefits is an increase of the cost for path retrieval. To retrieve the shortest path weight from the *HPV*, we have to compare all the concatenated local shortest paths of the involved fragments with the shortest paths in the supergraph (Theorem 2 and Theorem 3). This clearly incurs more computational cost.

As a comparison, we have implemented the heuristic A^* algorithm [31], because it has been quite influential to the research of the shortest path problems due to its good performance. The estimate function used by our A^* algorithm is:

$$estimate = \sqrt{\left(x_1 - x_2\right)^2 + \left(y_1 - y_2\right)^2} \times \text{minimum link weight per unit}.$$

The *estimate* is the Euclidean distance between two nodes times the minimum link weight per unit distance. This estimate will always underestimate the actual shortest path weight and thus gives the correct result.

In Fig. 21, we vary the graph size and measure the path retrieval time of the *FPV* approach, our *HEPV* model with a two-level *HPV* and a three-level *HPV*, and the A^* algorithm. The *FPV* is most efficient because a simple look up of the *encoded path tables* suffices to retrieve the requested shortest path. The two-level *HPV* is also very efficient because only minimal computation is needed to retrieve the shortest path. The three-level *HPV* is less efficient than the two-level *HPV* because the paths precomputed by the three-level *HPV* represents shorter segments of shortest paths than those precomputed by the two-level *HPV*. Therefore, path computation in the three-level *HPV* requires a higher number of expansions of these precomputed path segments than the two-level *HPV*. Both two-level *HPV* and three-level *HPV* outperform A^* significantly because the A^* algorithm, without precomputing shortest path segments, searches paths by expanding each individual link. The number of link expansions during an A^* search is much larger than that of the path retrieval over the hierarchical path views in the *HEPV* model. In conclusion, the *HEPV* model is highly efficient in path retrieval as compared to the *compute-on-demand* approaches.

7 Related Work

Considerable effort has been devoted to investigate the transitive closure problems [8]. Some algorithms, such as Washall's algorithm [1], [8], are based on matrix representation and calculation; while others, such as Dijkstra's algorithm [9], are based on graph traversal.

Various methods have been proposed to improve the standard transitive closure algorithms for databases. Agrawal and Jagadish [4] presented the disk-based hybrid algorithms which combine the features of both matrix-based and graph-based algorithms. A special access structure for transitive closure queries was proposed by [5] to improve the disk I/O. Agrawal and Jagadish [3], Ausiello and Italiano [6], and Huang et al. [15] studied incremental algorithms for path computation. With the new computer architecture and computer network, the traditional algorithms were adapted to parallel and distributed transitive closure algorithms [11], [12], [14], [22]. The capability of efficiently calculating the shortest path transitive closure is an active research topic in some advanced application domains such as the route guidance of ITS [10], [15], [16], [17], [18], [27], [28], [32], [33].

On the other hand, the materialized view of the transitive closure is an alternative approach of precomputing and maintaining the transitive closure. In the path view approach, the request for the shortest path is achieved by looking it up in the view, eliminating the need for compute it on the fly. Such a path precomputation approach was

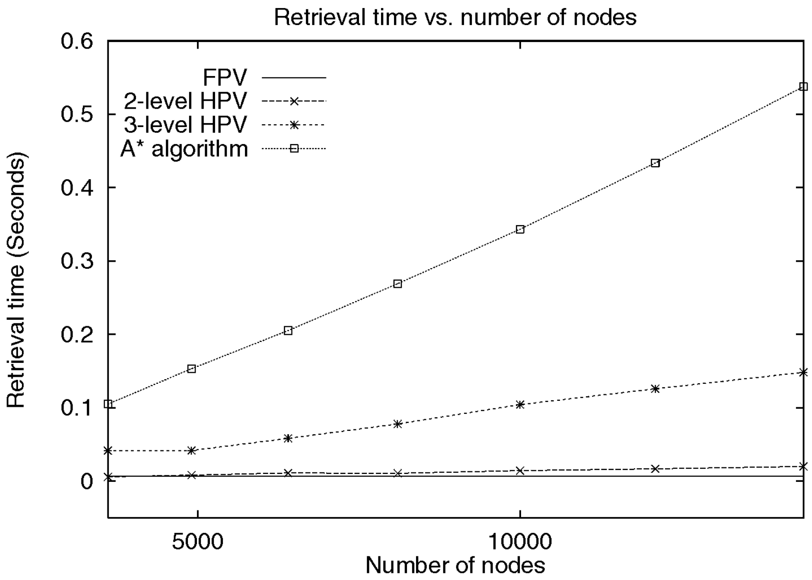


Fig. 21. Path retrieval times.

proposed by [3]. They have shown that the *semimaterialized encoding structure* has an acceptable storage overhead compared to maintaining all possible paths. In our previous work [15], we studied the trade-off between *precomputing* and *compute-on-demand* approaches in the context of ITS. We applied the encoded structure to ITS type graphs and developed incremental algorithms to handle cyclic graphs. In a related paper [17], we studied the (nonhierarchical) disk-based path view algorithms.

Several previous efforts have also focused on the idea of hierarchical structures. Houstma et al. [11], [12] proposed the idea of distributed and parallel transitive closure computation. Their approach divides a relation into fragments. To answer a path query, it first needs to determine the fragments which the path traverses, and then to perform the computation over these fragments. The approach prefers the fragmentation graph of the fragmented relation to be acyclic. The cost of the path query depends on the number of fragments the path traverses. Houstma et al. [14] was the continuous work of [11], [12] with the introduction of the notion of high-speed fragment. Unfortunately, the formation of the high-speed fragment is very sensitive to the update of the underlying base relation and therefore the authors recommended their approach only for a base relation which is rather stable.

Shekhar et al. [32] proposed a hierarchical A^* algorithm for navigation systems, which—while more efficient than flat A^*—does not guarantee optimality of the retrieved paths. Jung and Pramanik [26] proposed a hierarchical multigraph model by dividing the graph by nodes and pushing up the precomputed paths as well as links between the boundary nodes. This work did not address the issue of graph partitioning to break a map into smaller submaps—which is important to the hierarchical graph creation and influential to the performance of path retrieval. Hierarchical graph refreshing—inherent to road navigation to reflect dynamic traffic conditions—was also not handled.

Data decomposition techniques were studied in [2] where they proposed three heuristics to divide a large database into domains. Our experimental evaluation found that the optimal data clustering and partition technique proposed in [30] is not acceptable due to its exponential complexity, while the suggested suboptimal algorithm [13], [30] results in fragments of the flat graph with unnecessarily many border nodes which has an adverse effect on the efficiency of path retrieval over *HEPV*. The center-based algorithm [30] raises the problem of choosing center nodes for each fragment which may be difficult for large ITS graphs. In a related effort, [19] we studied the data clustering techniques for map databases and evaluated current clustering algorithms. Based on these evaluation, we devised the partition algorithm proposed in this paper for creating the hierarchical graph.

8 CONCLUSION

In this paper, we propose the Hierarchical Encoded Path Views (*HEPV*) model which optimizes shortest path query processing. The *HEPV* divides a large graph into smaller graph fragments and organizes them in a hierarchical manner. A path view storing all-pair shortest paths within each fragment is precomputed and frequently updated. Path computation in *HEPV* corresponds to expanding the shortest path segments stored in the path views. Because each all-pair shortest path view is confined to a small graph fragment, the *HEPV* is much more efficient in the path view update and storage costs than the shortest path precomputation approach based on the entire flat graph. Because *HEPV* computes paths by concatenating shortest path segments that are already computed, it is much more efficienct in path computation than the *compute-on-demand* approach (e.g., A^*) which searches paths by expanding individual graph links. In this paper, we prove that paths computed by *HEPV* are optimal.

We conducted a comprehensive set of experiments based on a Sun SPARC-20 workstation using grid graphs, randam graphs, and real map data sets. Our experimental results confirm the superiority of our *HEPV* model. For example, for a graph of 3,600 nodes, it takes 40 seconds to encode path views for a two-level *HEPV* and more than 200 seconds for the flat graph approach. For memory requirement, the former needs about 10 MBytes whereas the latter takes up close to 100 MBytes. To retrieve a path on a graph of 14,400 nodes, the average time needed by a two-level *HEPV* is less than 20 milliseconds, which is about 27 times shorter than the A^* approach. For large graphs (>3,000), a three-level *HEPV* is more efficient in path view encoding and storage but less efficient than a two-level *HEPV*. In general, our *HEPV* model represents an excellent compromise between *compute-on-demand* versus *precomputing* all paths. The appropriate number of levels in *HEPV* is determined by the graph sizes and the required path query time constraint.

The contributions of our paper are:

1) we have proposed the *HEPV*;
2) we have developed the link-sorting based partition algorithm which is very efficient for large map fragmentation;
3) we have developed algorithms to create and maintain the *HEPV*, as well as an algorithm to retrieve the *optimal* shortest path;
4) we have proven the optimality of the shortest path retrieved by *HEPV*; and
5) we have conducted extensive experiments of evaluating our *HEPV* approach and contrasting it with alternative solutions.

We plan to extend our hierarchical graph model to a parallel and distributed environment. The extension of the hierarchical graph model to constrained (e.g., spatial, temporal constraints) path findings also needs to be addressed [20].

ACKNOWLEDGMENTS

This work was supported, in part, by the University of Michigan ITS Center of Excellence under Grant no. DTFH61-93-X-00017-Sub, and was sponsored by the U.S. Department of Transportation and the Michigan Department of Transportation. We are also grateful for partial support from National Science Foundation NYI Grant no.

1R1 94-57609, Intel, AT&T, and Illustra. Ning Jing, who made his contribution to this research while he was visiting the Department of Electrical Engineering and Computer Science at the University of Michigan, was also supported, in part, by the State Education Commission of the Peoples Republic of China. Elke A. Rundensteiner was with the Department of Electrical Engineering and Computer Science at the University of Michigan when this work was performed.

REFERENCES

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

[2] R. Agrawal and H.V. Jagadish, "Efficient Search in Very Large Databases," *Proc. 14th VLDB Conf.* , pp. 407–418, 1988.

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

[4] R. Agrawal and H.V. Jagadish, "Hybrid Transitive Closure Algorithms," *Proc. 16th VLDB Conf.,* Brisbane, Australia, pp. 326–334, 1990.

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

[6] G. Ausiello and G.F. Italiano, "Incremental Algorithms for Minimal Length Paths," *J. Algorithms,* vol. 12, pp. 615–638, 1991.

[7] R.E. Bellman, "On A Routing Problem," *Quarterly Applications Math.,* vol. 16, pp. 87–90, 1958.

[8] T. Cormen, C. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, 1993.

[9] E.W. Dijkstra, "A Note on Two Problems in Connection with Graph Theory," *Numerische Mathematik,* vol. 1, pp. 269–271, 1959.

[10] M.J. Egenhofer, "What's Special About Spatial? Database Requirements for Vehicle Navigation in Geographic Space," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 398–402, 1993.

[11] M.A.W. Houstma, P.M.G. Apers, and S. Ceri, "Complex Transitive Closure Queries on a Fragmented Graph," *Proc. Third Int'l Conf. Data Theory, Lecture Notes in Computer Science*, Springer-Verlag, pp. 470–484, 1990.

[12] M.A.W. Houstma, P.M.G. Apers, and S. Ceri, "Distributed Transitive Closure Computations: The Disconnection Set Approach," *Proc. 16th VLDB Conf.*, pp. 335–346, 1990.

[13] M.A.W. Houstma, P.M.G. Apers, and G.L.V. Schipper, "Data Fragmentation for Parallel Transitive Closure Strategies," *Proc. Ninth Int'l Conf. Data Eng.*, pp. 447–456, 1993.

[14] M.A.W. Houstma, F. Cacace, and S. Ceri, "Parallel Hierarchical Evaluation of Transitive Closure Queries," *Proc. First Int'l Conf. Parallel and Distributed Information Systems*, pp. 130–137, 1990.

[15] Y.-W. Huang, N. Jing, and E.A. Rundensteiner, "A Semi-Materialized View Approach for Route Maintenance in Intelligent Vehicle Highway Systems," *Proc. Second ACM Workshop Geographic Information Systems,* pp. 144–151, Nov. 1994.

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

[17] Y.-W. Huang, N. Jing, and E.A. Rundensteiner, "Path Queries for Transportation Networks: Dynamic Reordering and Sliding Window Paging Techniques," *Proc. Fourth ACM Workshop Geographic Information Systems*, Nov. 1996.

[18] Y.-W. Huang, N. Jing, and E.A. Rundensteiner, "Evaluation of Hierarchical Path Finding Techniques for ITS Route Guidance," *Proc. Sixth Ann. Meeting ITS-Am.,* 1996.

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

[20] Y.-W. Huang, N. Jing, and E.A. Rundensteiner, "Integrated Query Processing Strategies for Spatial Path Queries," *Proc. 13th Int'l Conf. Data Eng.,* 1997, to appear.

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

[22] M.A.W. Houstma, A.N. Wilslchut, and J. Flokstra, "Implementation and Performance Evaluation of a Parallel Transitive Closure Algorithm on PRISMA/DB," *Proc. 19th VLDB Conf.*, pp. 206–217, 1993.

[23] Y. Ioannidis and R. Ramakrishnan, "Efficient Transitive Closure Algorithms," *Proc. 14th VLDB Conf.*, pp. 382–394, 1988.

[24] Y. Ioannidis, R. Ramakrishnan, and L. Winger, "Transitive Closure Algorithms Based on Graph Traversal," *ACM Trans. Database Systems,* vol. 18, no. 3, pp. 512–576, Sept. 1993.

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

[26] S. Jung and S. Pramanik, "HiTi Graph Model of Topological Road Maps in Navigation Systems," *Proc. 12th Int'l Conf. Data Eng.*, pp. 76–84, 1996.

[27] J. Krozel and D. Andrisani II, "Intelligent ε-Optimal Path Prediction for Vehicular Travel," *IEEE Trans. Systems, Man, and Cybernetics,* vol. 25, no. 2, pp. 345–353, Feb. 1995.

[28] C.-K. Lee, "A Multiple-Path Routing Strategy for Vehicle Route Guidance Systems," *Transportation Research,* vol. 2, no. 3, pp. 185–195, 1994.

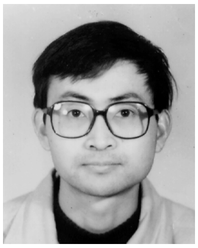
[29] Loral Federal Systems, *IVHS Architecture Phase One Final Report*, Federal Highway Admin., *DTFH61-93-C-00211*, 1994.

[30] W.T. McCormick Jr., P.J. Schweitzer, and T.W. White, "Problem Decomposition and Data Reorganization by a Clustering Technique," *Operations Research,* vol. 20, no. 5, pp. 993–1,009, 1972.

[31] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison Wesley, Reading, Mass., 1994.

[32] S. Shekhar, A. Kohli, and M. Coyle, "Path Computation Algorithms for Advanced Traveler Information Systems," *Proc. Ninth Int'l Conf. Data Eng.*, pp. 31–39, 1993.

[33] T.A. Yang, S. Shekhar, B. Hamidzadeh, and P.A. Hancock, "Path Planning and Evaluation in IVHS Databases," *VNIS*, pp. 283–290, 1991.



Ning Jing received the BS and MS degrees in electrical engineering, and the PhD degree in computer science, all from the Changsha Institute of Technology, Changsha, Peoples Republic of China. He is currently a faculty member in the Department of Electrical Engineering at the Changsha Institute of Technology. From 1994 to 1996, he was 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, multimedia databases, databases for Internet information services, and geographic information systems. Dr. Jing received a High Education Award from the State Department in 1992 and an Outstanding Visiting Scholar Grant from the State Education Commission in 1994.

Yun-Wu Huang received the BS degree in management science from National Chiao-Tung University in 1982 and the MS degree in computer science from Indiana University in 1989. Currently, he is a PhD candidate in the Department of Electrical Engineering and Computer Science at the University of Michigan. He worked as a computer professional in the areas of database and computer network from 1989 to 1995. His current research interests include spatia databases, multimedia databases, and geographic information systems.

Elke A. Rundensteiner received a BS degree (Vordiplom) from Johann Wolfgang Goethe University, Frankfurt, West Germany, and a PhD degree from the University of California, Irvine. 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, multimedia databases, and geographic information systems. She has received numerous honors and awards, including a Fulbright Scholarship, a National Science Foundation Young Investigator Award in databases in 1994, an Intel Young Investigator Engineering Award, and an IBM Partnership Award. She is member of the IEEE and the ACM.