Hierarchical Optimization of Optimal Path Finding for Transportation Applications [*]

Ning Jing[†]
Changsha Institute of Technology
jning@eecs.umich.edu

Yun-Wu Huang
University of Michigan
ywh@eecs.umich.edu

Elke A. Rundensteiner
University of Michigan
rundenst@eecs.umich.edu

Abstract

Efficient path query processing is a key requirement for advanced database applications including GIS (Geographic Information Systems) and ITS (Intelligent Transportation Systems). We study the problem in the context of automobile navigation systems where a large number of path requests can be submitted over the transportation network within a short period of time. To guarantee efficient response for path queries, we employ a path view materialization strategy for precomputing the best paths. We tackle the following three issues: (1) memory-resident solutions quickly exceed current computer storage capacity for networks of thousands of nodes, (2) disk-based solutions have been found inefficient to meet the stringent performance requirements, and (3) path views become too costly to update for large graphs. We propose the $HEPV$ (Hierarchical Encoded Path View) approach that addresses these problems while guaranteeing the optimality of path retrieval. Our experimental results reveal that $HEPV$ is more efficient than previously known path finding approaches.

1 Introduction

1.1 Motivation and Goals

The capability of computing path queries is an essential feature for advanced database applications such as geographical information systems [6] and computer networks. Our goal is to explore solutions for path finding in general with particular focus on addressing the problems inherent to navigation systems applications [6, 9, 10, 16]. This raises the following four issues: First, we are concerned with providing answers to path queries issued by a potentially large number of concurrent requests (e.g., during peak rush hour periods). Second, our solution must handle the dynamic nature of transportation networks, i.e., it must provide up-to-date query results even when the underlying transportation networks change frequently. Third, our solution must provide response at a near real-time level of performance (i.e., within seconds). Fourth, based on the requirements of *instruction-based* navigation systems [15], we are interested in efficiently determining the next link for the desired path rather than necessarily retrieving the complete path all at once. This is justified by the fact that a driver will need to know immediately *which next turn* to take while the retrieval of the *complete* path is less critical — especially as it may still be adjusted according to changing conditions during the travel period.

1.2 Path Query Optimization: Related Work

While much research has been conducted by both the theory and database communities on path finding, many suggested transitive closure algorithms [1, 3, 12] do not effectively handle path computation on large graphs in real-time application domains such as ITS (Intelligent Transportation Systems) systems. With new computer architectures and networks, the traditional algorithms were adapted to parallel and distributed transitive closure algorithms [7, 8]. As an alternative approach, [2] precompute all-pair shortest paths and store them in an encoded path view structure. Queries can be efficiently answered by performing simple lookups on the precomputed path view. This approach is a trade-off between computing paths from scratch and precomputing all paths. While this approach has been proven to be promising for relatively small map data sets, we determined [9] that it exceeds the memory capacity of typical computer systems for large graphs (e.g., graphs of 3,600 nodes or larger) and the performance of computing the path view deteriorates quickly (e.g., it takes 4 minutes for a graph with 3,600 nodes on a Sun SPARC-20.).

The hierarchical abstraction has been proposed by some researchers to overcome this problem. [8] for example divided a relation into fragments and introduced the notion of high-speed fragment. Unfortunately, the formation of high-speed fragments are very sensitive to the update of the underlying base relation. Therefore the authors recommended this approach only for rather stable base relations. In [10], we have presented a hierarchical graph model which classi-

[*]This work was supported in part by the University of Michigan ITS Center of Excellence grant (DTFH61-93-X-00017-Sub) sponsored by the U.S. Department of Transportation and by the Michigan Department of Transportation.

[†]On leave from Changsha Institute of Technology, China, is currently visiting the University of Michigan.

fies links according to road types and pushes up high speed roads such as highways to the next higher level of hierarchy. However, the paths retrieved from such hierarchical graphs are *not guaranteed to be optimal*. Similarly, [16] proposed a hierarchical A^* algorithm for road navigation systems, which — while more efficient than flat A^* — does not guarantee optimality either. [14] proposed another hierarchical multi-graph model by dividing graph into subgraphs and pushing up the precomputed paths as well as links between the boundary nodes. The paper did not discuss how the optimality can be achieved. Hierarchical graph refreshing — essential for road navigation to reflect the dynamic traffic condition — was also not handled.

1.3 Our Solution Approach: $HEPV$

Our proposed solution is the $HEPV$ (Hierarchical Encoded Path View) that addresses all of the issues outlined in Section 1.1 while guaranteeing path optimality. The basic idea is to divide a large graph into smaller graph fragments and organize them in a hierarchical manner by pushing up border nodes. We establish optimality theorems which guarantee that path retrieval on $HEPV$ returns globally optimal paths. These optimality theorems are used as basis for deriving the optimal hierarchical path search algorithm. We also present effective algorithms for accomplishing the following tasks: hierarchical graph generation, $HEPV$ encoding, and $HEPV$ incremental refreshing. The experiments confirm that the $HEPV$ approach represents an excellent compromise between the *compute-on-demand* and *precomputation* approaches.

Section 2 introduces the encoded path view structure. Section 3 proposes our hierarchical approach towards path encoding. Section 4 presents the optimality theorems and the optimal hierarchical path retrieval. Section 5 gives all associated algorithms. The experiments in Section 6 compare $HEPV$ to alternative path finding techniques. Section 7 concludes with a summary of contributions.

2 Background: Encoded Path View

2.1 Basic Graph Definitions

Definition 1 $G = (N, L, W)$ *is a graph, where* $N = \{N_i | 1 \leq i \leq n\}$ *is the set of nodes with* n *the total number of nodes.* $L = \{< N_i, N_j > | 1 \leq i, j \leq n \wedge i \neq j\}$ *is the set of links, where each link is a directed 2-tuple* $< N_i, N_j >$, *denoted simply as* L_{ij}. $W = \{LW_{ij} | LW_{ij} = f_c(L_{ij})\}$, *where* LW_{ij} *denotes the non-negative link weight of* L_{ij} *and* f_c *is the cost function of link* L_{ij}.

We call this graph *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 a sequence of nodes, where* $N_i = N_{k_1}$ *and* $N_j = N_{k_m}$, $N_{k_p} \in N$, $1 \leq p \leq m$, *and* $L_{k_q k_{q+1}} \in L$, $1 \leq q \leq m-1$. N_i, N_j *are source and destination nodes, respectively. The path weight* $PW_{ij} = \sum_{p=1}^{m-1} LW_{k_p k_{p+1}}$. *The shortest path* SP_{ij} *is the path from* N_i *to* N_j *that gives the minimum path weight* PW_{ij} *for all possible paths from* N_i *to* N_j. *The shortest path weight of* SP_{ij} *is denoted by* SPW_{ij}. $SPW_{ij} = \infty$ *if there is no path from* N_i *to* N_j. $SPW_{ij} = 0$ *if* $N_i = N_j$.

Definition 3 *The transitive closure of a graph* $G = (N, L, W)$ *is defined by* $CLOSURE(G) = \{< N_i, N_j > | L_{ij} \in L \vee (\exists N_k \in N)(< N_i, N_k > \in CLOSURE(G) \wedge L_{kj} \in L)\}$.

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

2.2 Encoded Path View Structure: $FEPV$

To find the shortest paths, we could apply one of the shortest path algorithms [4] to calculate this path *on-the-fly*. However, the high complexity for this computation prevents their use in real-time applications such as ITS. An alternative solution is to *precompute* all-pair shortest paths (i.e., *materialized path view*). Thus, the request for the shortest path is achieved by looking it up in the view, eliminating the need for path computation. Although this fully materialized path view is very efficient for path retrieval, it requires an unrealistically large amount of storage space. We thus utilize the encoded path view structure [2, 9], a partially materialized path view which stores the same path information using less storage space, and with only a linear increase in path retrieval time.

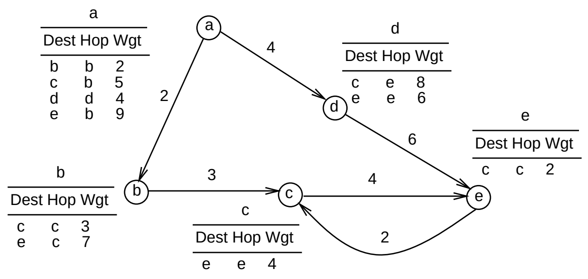


Figure 1: Encoded path view structure.

More specifically, the encoded path view, henceforth called $FEPV$ (Flat Encoded Path View), encodes paths by associating with each node a table that consists of all destination nodes reachable from this node and the direct successor node (called nexthop) on the shortest path from this node to the destination node, as well as the path weight of the shortest path. Therefore, $FEPV$ corresponds to a set of tables of 3-tuples $< destination, nexthop, pathweight >$, with each such table associated with a source node.

Figure 1 shows the encoded path structure of an example graph. Since the shortest path from N_a to N_e is $SP_{ae} = \{N_a, N_b, N_c, N_e\}$ and its path weight $SPW_{ae} = 9$, a tuple $< e, b, 9 >$ is thus in the table associated with N_a. We use $ENCODE(G)$ to denote the path encoding process for graph G. A complete discussion of the materialized path view approach can be found in [9].

2.3 Path Retrieval Over $FEPV$

To retrieve the shortest path from $FEPV$, we start from the table associated with the source node, identifying the needed tuple by searching the table using the destination node as the key. Once the tuple is found, its nexthop is the direct successor node on the shortest path, while its path weight represents the weight of the shortest path from source to destination. Next, we treat the nexthop node as the source node and repeat the search process recursively until the nexthop corresponds to the destination.

In Figure 1, to retrieve the shortest path from N_a to N_e, we first search the table of N_a, using N_e as the key. The result is $< e, b, 9 >$, thus $SPW_{ae} = 9$, and the nexthop is N_b. We then search the table of N_b, again using N_e as the key. The result is $< e, c, 7 >$, the nexthop is N_c. Searching the table of N_c and using N_e as the key, we get the tuple $< e, e, 4 >$. The nexthop N_e is the desired destination node. Thus we have retrieved the shortest path $SP_{ae} =< N_a, N_b, N_c, N_e >$, and its path weight $SPW_{ae} = 9$.

2.4 Discussion

The creation of $FEPV$, equal to calculating all-pair shortest paths, is computationally expensive [1], and the space requirement for the $FEPV$ is also high [2]. For small maps, the computational costs are acceptable. But if the map is large, encoding or updating the $FEPV$ may take a long time. Our experiments (see Section 6.1) show that the encoding time for a graph of 3,600 nodes is as long as 250 seconds on a Sun SPARC-20. Furthermore, the space requirement becomes so large (nearly 100M bytes) that to efficiently cache data in memory becomes less cost effective, forcing more usage of secondary storage.

3 Hierarchical Graph Model

3.1 Hierarchical Graph Definition

We now extend some of the notations in order to make them applicable to the hierarchical graph model.

$L_G(i, j)$ — the link from N_i to N_j in graph G.
$LW_G(i, j)$ — the link weight of $L_G(i, j)$.
$P_G(i, j)$ — any path from N_i to N_j in graph G.
$PW_G(i, j)$ — path weight of $P_G(i, j)$.
$SP_G(i, j)$ — 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 4 *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 $W^f = \{LW^f_{ij} | L^f_{ij} = L_{ij} \wedge LW^f_{ij} = LW_{ij}\}$. 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.

Definition 5 *A partition $P_G = \{G^f_1, G^f_2, \cdots, G^f_p\}$ of 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: $L^f_u \cap L^f_v = \emptyset$.*

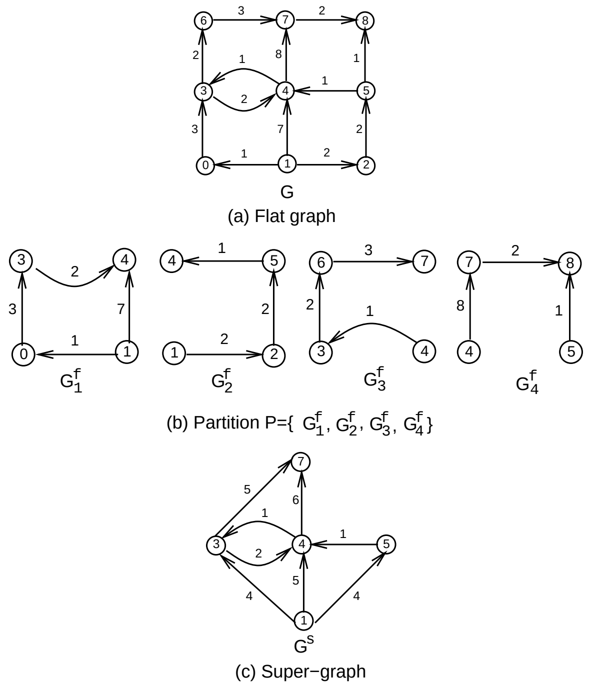


Figure 2: Flat graph, partition and super-graph.

In Definition 5, the first two requirements state that the fragments as a whole must contain all nodes and links of G. The third requirement states 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. Figure 2(b) depicts a partition of the flat graph from Figure 2(a).

Definition 6 *Given a partition $P_G = \{G^f_1, G^f_2, \cdots, G^f_p\}$, the intersection of nodes in fragment G^f_u with nodes in all other fragments of P_G is called the border node set for G^f_u. $BORDER(G^f_u) = N^f_u \cap (\bigcup^p_{v=1 \wedge u \ne v} N^f_v)$, where $1 \le u \le p$. If $BORDER(G^f_u) \cap BORDER(G^f_v) \ne \emptyset$ 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 [4]. 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.

$1 \leq u, v \leq p$ and $u \neq v$, then the fragments G^f_u and G^f_v are said to be adjacent. Conversely, the non-border node set is defined by $LOCAL(G^f_u) = N^f_u - BORDER(G^f_u)$.

A border node of a fragment appears in at least one of the other fragments, while a local node appears in only one fragment. Two fragments are said to be adjacent if they have at least one common border node. In Figure 2(b), $BORDER(G^f_1) = \{N_1, N_3, N_4\}$ because N_1 and N_4 also appear in G^f_2, N_3 also appears in G^f_3. $LOCAL(G^f_1) = \{N_0\}$ since N_0 is only in G^f_1.

Definition 7 *Given $P_G = \{G^f_1, G^f_2, \cdots, G^f_p\}$ a partition of G, a super-graph $G^s = (N^s, L^s, W^s)$ is defined by:*
(1) $N^s = \bigcup_{u=1}^{p} BORDER(G^f_u)$.
(2) $L^s = \{L_{ij} | (N_i, N_j \in N^s) \wedge (\exists u)((1 \leq u \leq p) \wedge (N_i, N_j \in N^f_u) \wedge (< N_i, N_j > \in CLOSURE(G^f_u))\}$.
(3) For any link $L_{ij} \in L^s$, its link weight $LW_{G^s}(i,j) = MIN(\{SPW_{G^f_u}(i,j) | (1 \leq u \leq p) \wedge (< N_i, N_j > \in CLOSURE(G^f_u))\})$, where MIN is the minimum function.

Intuitively, the super-graph of a partition P_G consists of all the border nodes of P_G. If there exists a pair of border nodes N_i and N_j in a fragment and N_j is reachable from N_i in the context of that fragment, there is a link from N_i to N_j in the super-graph G^s. Its link weight is the minimum path weight of the shortest paths from N_i to N_j. Note that these are just minimal within the context of one fragment, but not necessarily globally minimal.

For example, Figure 2(c) depicts the super-graph of the partition from Figure 2(b). $L_{G^s}(1,3) \in L^s$ because there is a path $P_{G^f_1}(1,3) = \{N_1, N_0, N_3\}$. While $L_{G^s}(5,7) \notin L^s$ because there is no path in any of the fragments. $LW_{G^s}(1,4) = 5$ because there are two shortest paths in two fragments: $SP_{G^f_1}(1,4) = \{N_1, N_0, N_3, N_4\}$ with $SPW_{G^f_1}(1,4) = 6$, and $SP_{G^f_2}(1,4) = \{N_1, N_2, N_5, N_4\}$ with $SPW_{G^f_2}(1,4) = 5$. By Definition 7, we choose the one with minimum shortest path weight.

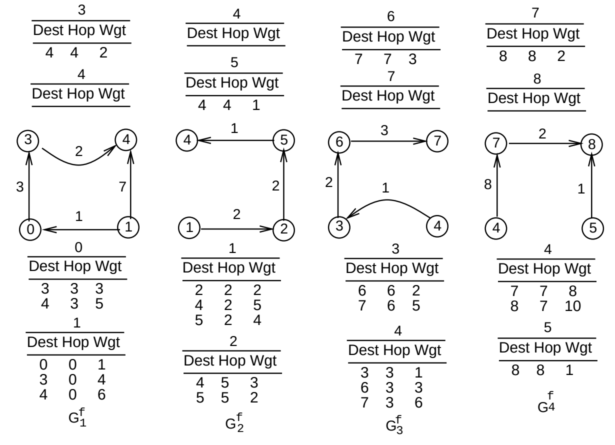
Definition 8 *A hierarchical graph of a (flat) graph G is defined by $G^h = (P_G, G^s)$, where P_G is a partition of the graph G and G^s is the super-graph of P_G.*

The hierarchical graph described above consists of two levels of graphs[3], the first level graphs are fragments of a partition, and the second level graph is the super-graph of the partition. The hierarchical graph transforms a flat graph into several small fragmented graphs and a super-graph. The advantage of such a hierarchical graph is that the number of nodes within each fragment can be effectively controlled by the number of partition. As discussed in the previous section, the computational complexity of the path algorithm largely depends on the number of nodes in the graph, thus savings are expected.

[3]The hierarchical graph has been extended to multi-levels [13]. Due to space limitation, we restrict the hierarchical graph to two levels.

3.2 Hierarchical Encoded Path View: $HEPV$

Given a hierarchical graph $G^h = (P, G^s)$ with $P = \{G^f_1, G^f_2, \cdots, G^f_p\}$ a partition of the flat graph G and $p > 1$, we now extend the $FEPV$ structure to the hierarchical graph. For each fragment, we create and maintain the encoded path view as described in Section 2.2. Each link in the super-graph can be treated as a virtual link that consists of the following information: $fragmentID$, $nexthop$, and $weight$. $fragmentID$ identifies a fragment, say G^f_u, which contains the shortest path $SP_{G^f_u}(i,j)$ that $L_{G^s}(i,j)$ represents. $nexthop$ is the direct successor of the shortest path $SP_{G^f_u}(i,j)$ in G^f_u. $weight$ is the link weight $LW_{G^s}(i,j)$ which is equal to $SPW_{G^f_u}(i,j)$. Consequently, the extended encoded path view structure of the super-graph corresponds to tables of 4-tuples $< destination, fragmentID, nexthop, pathweight >$, with each such table associated with one node in the super-graph. Therefore, the encoded path view structure of the hierarchical graph consists of a set of tables associated with each node of all fragments and of the super-graph. We refer to this as $HEPV$ (*Hierarchical Encoded Path View*).



G^f_1

3

Dest	Hop	Wgt
4	4	2

4

Dest	Hop	Wgt

0

Dest	Hop	Wgt
3	3	3
4	3	5

1

Dest	Hop	Wgt
0	0	1
3	0	4
4	0	6

G^f_2

4

Dest	Hop	Wgt

5

Dest	Hop	Wgt
4	4	1

1

Dest	Hop	Wgt
2	2	2
4	2	5
5	2	4

2

Dest	Hop	Wgt
4	5	3
5	5	2

G^f_3

6

Dest	Hop	Wgt
7	7	3

7

Dest	Hop	Wgt

3

Dest	Hop	Wgt
6	6	2
7	6	5

4

Dest	Hop	Wgt
3	3	1
6	3	3
7	3	6

G^f_4

7

Dest	Hop	Wgt
8	8	2

8

Dest	Hop	Wgt

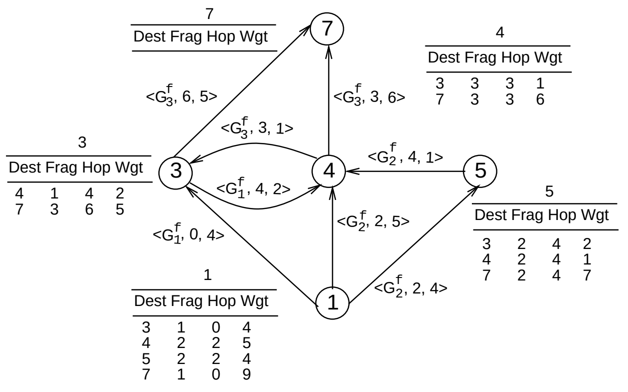
4

Dest	Hop	Wgt
7	7	8
8	7	10

5

Dest	Hop	Wgt
8	8	1

(a) Partition P={ $G^f_1, G^f_2, G^f_3, G^f_4$ }



7

Dest	Frag	Hop	Wgt

4

Dest	Frag	Hop	Wgt
3	3	3	1
7	3	3	6

3

Dest	Frag	Hop	Wgt
4	1	4	2
7	3	6	5

5

Dest	Frag	Hop	Wgt
3	2	4	2
4	2	4	1
7	2	4	7

1

Dest	Frag	Hop	Wgt
3	1	0	4
4	2	2	5
5	2	2	4
7	1	0	9

(b) The super−graph

Figure 3: $HEPV$ of the hierarchical graph.

In Figure 3, we show the $HEPV$ structure of the example hierarchical graph from Figure 2. The tables associated with each node of the fragments have the same meaning as we described in Section 2.2. In the super-graph, the link $L_{G^s}(1,3)$ is associated with three attributes $< \; G_1^f, 0, 4 \; >$, which means: the corresponding shortest path of this link is in the fragment G_1^f, the nexthop of the shortest path is N_0, and the link weight $LW_{G^s}(1,3) = 4$. In $HEPV$, the table of node N_1 in the super-graph, denoted by the tuple $< 7, 1, 0, 9 >$, tells us that the shortest path weight $SPW_{G^s}(1,7) = 9$, the nexthop of the shortest path $SP_{G^s}(1,7)$ is the node N_0 in the fragment G_1^f.

4 Optimal Path Retrieval Over $HEPV$

In this section, we first present three optimality theorems[4] which indicate how the optimal shortest path over the hierarchical graph can be computed. Based on the $HEPV$ structure and optimality theorems, we then develop the optimal hierarchical path retrieval algorithm.

Theorem 1 *Let $G^h = (P, G^s)$ be a hierarchical graph. For any node pair $N_a, N_b \in G^s$, we have $SPW_{G^s}(a,b) = SPW_G(a,b)$.*

The first optimality theorem shows that the shortest path weight computed on the super-graph solely corresponds to the actual shortest path weight in the flat graph.

Theorem 2 *Let $G^h = (P_G, G^s)$ be a hierarchical graph with partition $P_G = \{G_1^f, G_2^f, \cdots, G_p^f\}$ and $p > 1$. For $N_a, N_b \in LOCAL(G_u^f)$, where $1 \le u \le p$, this holds: $SPW_G(a,b) = MIN(SPW_{G_u^f}(a,b), MIN(\{SPW_{G_u^f}(a,i) + SPW_{G^s}(i,j) + SPW_{G_u^f}(j,b) | (N_i, N_j \in BORDER(G_u^f)) \wedge N_i \neq N_j\}))$.*

Theorem 2 indicates how the shortest path weight can be computed over the hierarchical graph if both source and destination nodes are in the same fragment. The shortest path weight is the smaller one between the local shortest path inside the fragment and the minimum path weight among all possible cross-level shortest paths from the source node to a border node N_i, from N_i to a border node N_j, and from N_j to the destination node.

Theorem 3 *Let $G^h = (P, G^s)$ be a hierarchical graph with partition $P = \{G_1^f, G_2^f, \cdots, G_p^f\}$ and $p > 1$. For $N_a \in N_u^f, N_b \in N_v^f$, where $1 \le u, v \le p$ and $u \neq v$, this holds: $SPW_G(a,b) = MIN(\{SPW_{G_u^f}(a,i) + SPW_{G^s}(i,j) + SPW_{G_v^f}(j,b) | N_i \in BORDER(G_u^f) \wedge N_j \in BORDER(G_v^f)\})$.*

Theorem 3 indicates how the shortest path weight over the hierarchical graph can be computed if source and destination nodes are in different fragments. In this case, the shortest path weight is the minimum among all possible cross-level shortest paths described above.

[4] We omit the proofs here due to space limitation. Instead, complete proofs are given in [13].

Based on the optimality theorems, we develop the optimal path retrieval algorithm which is depicted in Figure 4. In the algorithm, we use $ESP_{G_u^f}(i,j)$ to denote the encoded shortest path $SP_{G_u^f}(i,j)$ for the fragment G_u^f. The shortest path $SP_{G^s}(i,j)$ of the super-graph G^s is denoted by $ESP_{G^s}(i,j)$. Path retrieval over the hierarchical graph consists of four parts according to the four cases of the source and destination nodes:

Case 1: Both the source and destination nodes are border nodes (line 1 in Figure 4).

Case 2: The source node is a border node, but the destination node is a local node (line 4 in Figure 4).

Case 3: The source node is a local node, but the destination node is a border node (line 10 in Figure 4).

Case 4: Both the source and destination nodes are local nodes (line 16 in Figure 4).

```
ALGORITHM $PathRetrieval(N_a, N_b)$
01 if $N_a, N_b \in N^s$
02   $SPW_G(a,b) = ESP_{G^s}(a,b).pathweight$;
03 else
04   if $N_a \in N^s \wedge N_b \in LOCAL(G_u^f) \wedge 1 \le u \le p$
05     $SPW_G(a,b) = \infty$;
06     $\forall N_i \in BORDER(G_u^f)$
07       if $(SPW_{G^s}(a,i) + SPW_{G_u^f}(i,b)) < SPW_G(a,b)$
08         $SPW_G(a,b) = SPW_{G^s}(a,i) + SPW_{G_u^f}(i,b)$;
09   else
10     if $N_a \in LOCAL(G_u^f) \wedge 1 \le u \le p \wedge N_b \in N^s$
11       $SPW_G(a,b) = \infty$;
12       $\forall N_i \in BORDER(G_u^f)$
13         if $(SPW_{G_u^f}(a,i) + SPW_{G^s}(i,b)) < SPW_G(a,b)$
14           $SPW_G(a,b) = SPW_{G_u^f}(a,i) + SPW_{G^s}(i,b)$;
15     else
16       if $N_a \in LOCAL(G_u^f) \wedge N_b \in LOCAL(N_v^f) \wedge$
–            $1 \le u, v \le p$
17         $SPW_G(a,b) = \infty$;
18         $\forall N_i \in BORDER(G_u^f) \wedge N_j \in BORDER(G_v^f)$
19           if $(SPW_{G_u^f}(a,i) + SPW_{G^s}(i,j) + SPW_{G_v^f}(j,b))$
–                $< SPW_G(a,b)$
20             $SPW_G(a,b) = SPW_{G_u^f}(a,i) + SPW_{G^s}(i,j) +$
–                $SPW_{G_v^f}(j,b)$;
21         if $u = v \wedge SPW_{G_u^f}(a,b) < SPW_G(a,b)$
22           $SPW_G(a,b) = SPW_{G_u^f}(a,b)$;
```

Figure 4: Retrieve the shortest path from $HEPV$.

Theorem 1 applies to Case 1. In other cases, Theorem 2 and Theorem 3 indicate what is needed to achieve optimal path retrieval. Although the retrieval of the shortest path over $HEPV$ is less efficient than the path retrieval over $FEPV$ approach, we will show (see Section 6) that it is still significantly faster than the compute *on-the-fly* approach. The algorithm only needs to retrieve the local shortest paths and compute the cross-level shortest paths concatenated by border nodes.

5 $HEPV$ Creation And Maintenance

5.1 $HEPV$ Creation

Assuming we are given the partitions of a graph[5], we can then create the super-graph and the $HEPV$ of the hierarchical graph by the algorithm outlined in Figure 5. For simplicity, we use $L_{G^s}(i,j)$ to denote the 3-attributes $< fragmentID, nexthop, weight >$ associated with the link of the super-graph from N_i to N_j. In the algorithm, the super-graph contains all the border nodes (line 1). Next, each fragment is encoded by an all-pair shortest path algorithm (line 3). For any border node pair of each fragment (line 4), if there is a path between these two border nodes (line 5), then there should be a link between the two border nodes in the super-graph. If there is no link between these two border nodes in the super-graph, the algorithm creates a new link (lines 12 to 15). Otherwise it updates the existing link (lines 8 to 10) to guarantee the link weight in the super-graph is the minimum path weight among all the shortest paths of the involved fragments. Finally, the super-graph is encoded (line 16).

```
ALGORITHM $HEPVCreate(\{G^f_0, G^f_1, \cdots, G^f_{p-1}\})$
01 $N^s = \bigcup_{u=0}^{p-1} BORDER(G^f_u)$; $L^s = \emptyset$;
02 $\forall (0 \leq u < p)$
03   $ENCODE(G^f_u)$;
04   $\forall (N_i, N_j \in BORDER(G^f_u)) \wedge N_i \neq N_j$
05     if $SPW_{G^f_u}(i,j) < \infty$
06       if $L_{G^s}(i,j) \in L^s$
07         if $LG^s(i,j).weight > SPW_{G^f_u}(i,j)$
08           $L_{G^s}(i,j).fragmentID = G^f_u$;
09           $L_{G^s}(i,j).nexthop = ESP_{G^f_u}(i,j).nexthop$;
10           $L_{G^s}(i,j).weight = SPW_{G^f_u}(i,j)$;
11       else
12         $L^s = L^s \cup \{< N_i, N_j >\}$;
13         $L_{G^s}(i,j).fragmentID = G^f_u$;
14         $L_{G^s}(i,j).nexthop = ESP_{G^f_u}(i,j).nexthop$;
15         $L_{G^s}(i,j).weight = SPW_{G^f_u}(i,j)$;
16 $ENCODE(G^s)$;
```

Figure 5: The algorithm to create the $HEPV$.

5.2 $HEPV$ Update

Whenever a link weight of a fragment is changed, the corresponding link weight of the super-graph may also be affected. This would happen only if the super-graph is no longer representing the original shortest path of the underlying flat graph. It is obvious that we only need to update the encoded path view of the directly affected fragments and the super-graph, while all other fragments are not touched. We present the $HEPV$ update algorithm in Figure 6.

For any border node pair of the changed fragment (line 3), we need to find its new minimum shortest path weight (line

[5] In [13] we discuss the issues of graph partitioning and propose a link-sorting partition algorithm which we used in the experiments in this paper. [11] studies the graph clustering techniques which could also be used for partitioning.

```
ALGORITHM $HEPVUpdate(\{G^f_{v_0}, G^f_{v_1}, \cdots, G^f_{v_{q-1}}\})$
// $\{G^f_{v_0}, G^f_{v_1}, \cdots, G^f_{v_{q-1}}\}$ are changed.
// $\{G^f_{v_0}, G^f_{v_1}, \cdots, G^f_{v_{q-1}}\} \subseteq \{G^f_0, G^f_1, \cdots, G^f_{p-1}\}$.
01 $\forall G^f_u \in \{G^f_{v_0}, G^f_{v_1}, \cdots, G^f_{v_{q-1}}\}$
02   $ENCODE(G^f_u)$;
03   $\forall (N_i, N_j \in BORDER(G^f_u)) \wedge N_i \neq N_j$
04     $LW_{G^s}(i,j) = \infty$;
05 for $0 \leq u < p$
06   $\forall (N_i, N_j \in (BORDER(G^f_u) \wedge$
–          $(\vee_{k=0}^{q-1} BORDER(G^f_{v_k}))) \wedge N_i \neq N_j$
07     if $SPW_{G^f_u}(i,j) < \infty \wedge LW_{G^s}(i,j) > SPW_{G^f_u}(i,j)$
08       $L_{G^s}(i,j).fragmentID = G^f_u$;
09       $L_{G^s}(i,j).nexthop = ESP_{G^f_u}(i,j).nexthop$;
10       $L_{G^s}(i,j).weight = SPW_{G^f_u}(i,j)$;
11 $ENCODE(G^s)$;
```

Figure 6: The algorithm to update $HEPV$.

4). In the loop, for any border node pair of the changed fragment (which may also appear in other unchanged fragments), we find the new minimum shortest path weight among all the fragments which have this border node pair (lines 8 to 10).

6 Experimental Evaluation

We have performed experiments on synthetic grid graphs as well as a street map of Troy County and vicinity areas in suburban Detroit. All experiments are conducted on a SUN SPARC-20 with 128MB main memory.

6.1 Encoding the $HEPV$ vs. the $FEPV$

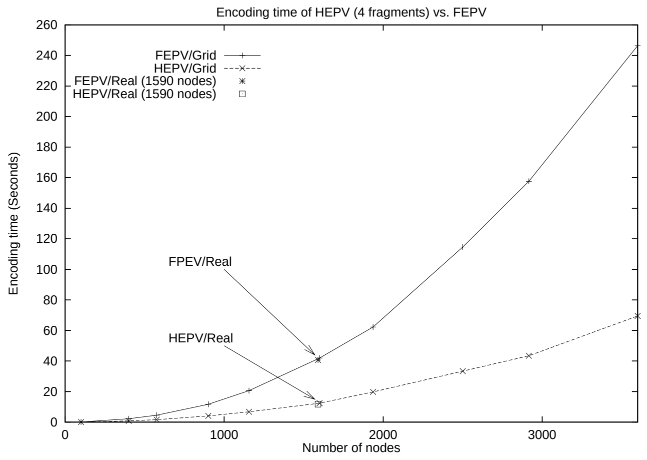


Figure 7: Encoding time of the $HEPV$ vs. $FEPV$.

In this experiment, our goal is to compare the performance gain achieved for path encoding of the hierarchical graph model. Figure 7 shows the experimental results of measuring encoding times on grid and real ITS graphs with 4 fragments. The encoding time of $HEPV$ is smaller than that of $FEPV$ for all graphs. The encoding time of the real graph is close to the encoding time of the grid graph for the same graph size. With the increase of the graph size, the encoding time of $FEPV$ increases sharply, while the encoding time of $HEPV$ increases slowly. The experiment clearly shows the superiority of $HEPV$ over $FEPV$ in terms of encoding time.

6.2 Updating the $HEPV$

There are three steps to update the $HEPV$: first, reencoding all the changed fragments; second, updating the super-graph; and third, reencoding the super-graph. Therefore, the cost of updating the $HEPV$ is $i \times C_e^f + C_u^s + C_e^s$, 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 super-graph, and C_e^s is the cost of reencoding the super-graph. The cost function thus is linear with respect to the number of changed fragments i.

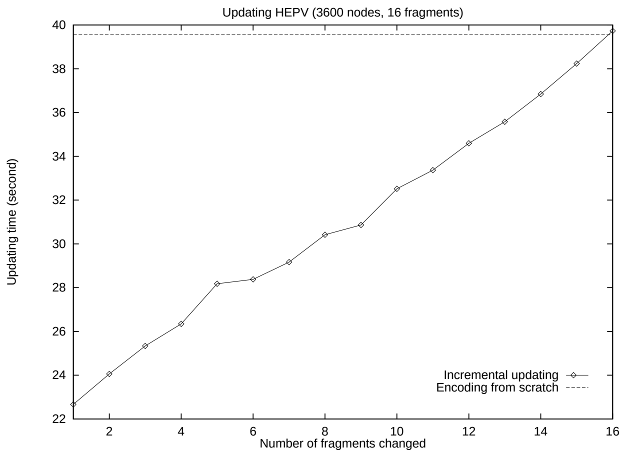


Figure 8: Updating time of the $HEPV$.

Our experiment (Figure 8) matches the above analysis. This experiment also indicates that the influence of changing a link weight is limited to the confines of the fragment the link is contained in, and if necessary, propagation of the change to the super-graph. All other fragments are un-affected. For the flat graph, the whole graph is typically affected even if only few links are changed [9].

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 super-graph is more than 22 seconds. If all the fragments are changed, the cost of incrementally updating the $HEPV$ 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.

6.3 Memory Requirement

Figure 9 depicts the memory requirements of $FEPV$ and $HEPV$ for different numbers of fragments. In the experiment, we included all the memory (static and dynamic) used by each method. For a graph of 3600 nodes, the $FEPV$ approach requires almost 100M bytes memory, while the $HEPV$ approach only needs 10M bytes for 16 fragments. Therefore the $HEPV$ approach is more memory efficient than $HEPV$ approach.

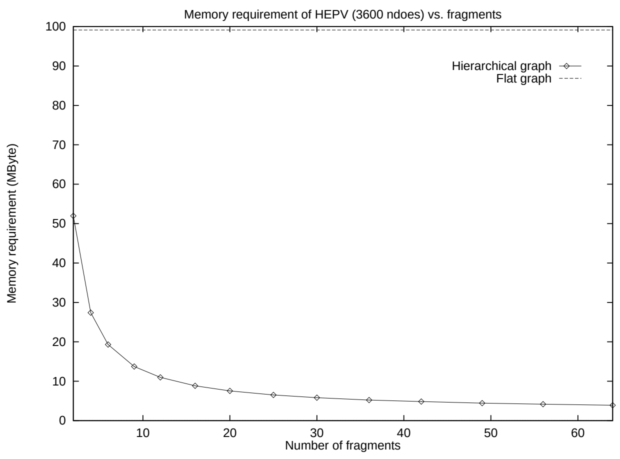


Figure 9: Memory requirement of $HEPV$ vs. fragments

6.4 Path Retrieval

Compared to the flat graph, the savings we gain from the $HEPV$ include a smaller encoding time (Section 6.1), a smaller updating time (Section 6.2), and also smaller memory requirements (Section 6.3). However, the price of these benefits is an increase of the cost for path retrieval. As a comparison, we have implemented A^* algorithm [16], because it has been quite influential to the research of the shortest path problems due to its good performance. In addition, we have also implemented the hierarchical A^* algorithm [10]. The hierarchical A^* algorithm is a three step A^* search: First, starting from the fragment to which the source node belongs to find a path that leads to a border node. Second, starting from the border node in the super-graph to find a path that leads to a node which is also a border node in the fragment of the destination node. Last, starting from the border node in the fragment to which the destination node belongs to find the path to the destination. We should note that the hierarchical A^* does *not* guarantee to find the optimal shortest path. This is contrast to our hierarchical approach, $HEPV$, which is *optimal*.

Figure 10 compares the four approaches of path retrieval. In the experiment, each approach retrieves the *complete* shortest path of two diagonal nodes of the grid graph of 3,600 nodes varying the number of fragments. The path retrieval time of the $FEPV$ is close to 0 because it only needs to look up the encoded structure. $HEPV$ is also very efficient because only minimal computation is needed to retrieve the shortest path. The next best is the hierarchical A^* algorithm because it searches on three smaller graphs, but the path it finds is not guaranteed to be optimal. The U-curve of the hierarchical A^* shows the relationship between the path retrieval

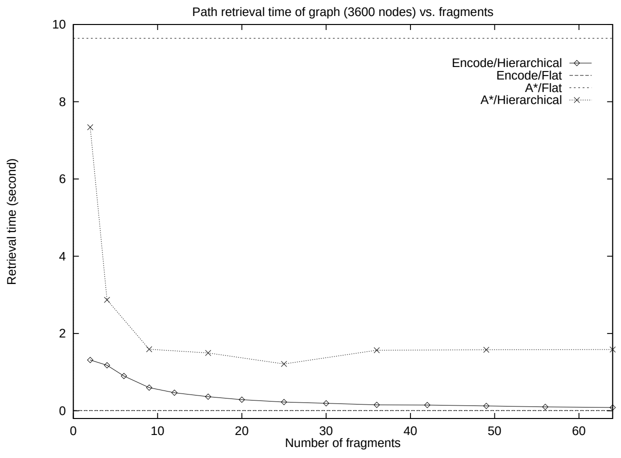


Figure 10: Path retrieval times.

time and the number of fragments. With the increase of fragments, the retrieval time for fragments decreases but the retrieval time for the super-graph increases. The flat A^* has the least efficient path retrieval time of more than 9 seconds because it calculates directly on the oversized graph.

7 Conclusion

In this paper, we proposed a hierarchical graph model that supports efficient optimal path retrieval. By extending the *encoded path view* [2, 9] to the hierarchical graph model, we achieve an excellent compromise between the complete computation of paths *on-demand* versus *precomputing* all paths. We conduct experiments for our hierarchical path view approach $HEPV$ using synthetic graphs and real maps. The experiments confirm that a link update now has a limited sphere of influence, therefore update costs of $HEPV$ are significantly reduced. Path retrieval is still significantly faster than A^*. In addition, the memory requirement of $HEPV$ is also much less compared to $FEPV$.

On a Sun SPARC-20, for a large hierarchical graph of 3600 nodes and 16 fragments, the experiments are very promising: only 40 seconds are needed to update the $HEPV$ even if all fragments are changed; the time to retrieve the next link of the desired path is too short to be measurable; and even the complete path retrieval takes less than 1 second; the memory requirement is also kept to a small 10M bytes compared to the conventional path view approach of nearly 100M bytes.

The contributions of our paper can be summarized as follows: (1) We have proposed the hierarchical graph model ($HEPV$) which exploits path materialization strategies; (2) We have developed algorithms to create and maintain $HEPV$; (3) We have developed a shortest path retrieval algorithm which can be shown to be optimal; and (4) Our experimental results demonstrate the superiority of our $HEPV$ approach over other alternative solutions.

References

[1] R. Agrawal, S. Dar and H. V. Jagadish, "Direct Transitive Closure Algorithms: Design and Performance Evaluation," *ACM TODS*, Vol. 15, No. 3, Sep. 1990, pp. 427 – 458.

[2] R. Agrawal and H. V. Jagadish, "Materialization and Incremental Update of Path Information", *Proc. of the 5th Int. Conf. on Data Engineering*, 1989, pp. 374 – 383.

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

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

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

[6] M. J. Egenhofer, "What's Special about Spatial? Database Requirements for Vehicle Navigation in Geographic Space," *Proc. of the 1993 ACM SIGMOD Conf.*, 1993, pp. 398 – 402.

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

[8] M. A. W. Hustma, F. Cacace, and S. Ceri, "Parallel Hierarchical Evaluation of Transitive Closure Queries," *Proc. of the 1st Int. Conf. on Parallel and Distributed Information Systems*, 1990, pp. 130 – 137.

[9] Y. W. Huang, N. Jing, and E. A. Rundensteiner, "A Semi-Materialized View Approach for Route Maintenance in IVHS," *Proc. of the 2nd ACM Workshop on Geographic Information Systems*, 1994, pp. 144 – 151.

[10] Y. W. Huang, N. Jing, and E. A. Rundensteiner, "Hierarchical Path Views: A Model Based on Fragmentation and Transportation Road Types," *Proc. of the 3rd ACM Workshop on Geographic Information Systems*, 1995, pp. 93 – 100.

[11] Y. W. Huang, N. Jing, and E. A. Rundensteiner, "Effective Graph Clustering for Path Queries in Digital Map Databases," *Proc. of the 5th Int. Conf. on Information and Knowledge Management*, 1996.

[12] Y. Ioannidis, R. Ramakrishnan, and L. Winger, "Transitive Closure Algorithms Based on Graph Traversal," *ACM TODS*, Vol. 18, No. 3, Sep. 1993, pp. 512 – 576.

[13] N. Jing, Y. W. Huang, and E. A. Rundensteiner, "Hierarchical Encoded Path View Approach and Its Evaluation," *ITS Research Center of Excellence, University of Michigan*, UMTRI-88499, 1995.

[14] S. Jung, S. Pramanik, "HiTi Graph Model of Topological Road Maps in Navigation Systems," *Proc. of the 12th Int. Conf. on Data Engineering*, 1996, pp. 76 – 84.

[15] Loral Federal Systems, "IVHS Architecture Phase One Final Report", *Sponsored by Federal Highway Administration, DTFH61-93-C-00211*, 1994.

[16] S. Shekar, A. Kohli, and M. Coyle, "Path Computation Algorithms for Advanced Traveler Information Systems," *Proc. of the 9th Int. Conf. on Data Engineering*, 1993, pp. 31 – 39.