2011 50th IEEE Conference on Decision and Control and
European Control Conference (CDC-ECC)
Orlando, FL, USA, December 12-15, 2011

Partitioning Graphs to Speed Up Point-to-Point Shortest Path Computations

Qing Song, and Xiaofan Wang, *Senior Member, IEEE*

Abstract—**Computing the shortest paths in graphs is a fundamental problem with numerous applications. The rapid growth of network in size and complexity has made it necessary to decrease the execution time of the shortest path algorithm. We develop an effective graph partition method to retrieve Balanced Traversing Distance partitions and constitute a hierarchical graph model based on the decomposed network for accelerating the path queries. We then propose a new heuristic hierarchical routing algorithm that can significantly reduce the search space by pruning unpromising subgraph branches. We evaluate our approach experimentally under different network partition schemes to show the gain in performance.**

I. Introduction

COMPUTING the shortest path between two points in a network is one of the most fundamental and well-studied problems in network algorithms. Numerous real-world applications can be transformed to this problem, which have attracted interests from many fields including geographic information systems (GIS), intelligent transportation systems (ITS), computer networks, and social networks. In 1959, Dijkstra [1] developed an elegant shortest path algorithm with a complexity of $O(|E|+|V|\log|V|)$, where $|V|$ is the number of vertices and $|E|$ is the number of arcs. Though Dijkstra algorithm computes the optimal solution in a theoretical sense, it is often too slow in practical applications, motivating several techniques for improving its response time [2].

In a typical application scenario, path queries have to be solved quickly and repeatedly for different node pairs on the same network, which stimulates the research on utilizing preprocessing techniques [3-7]. Naturally, pre-computing the shortest paths for all pairs of nodes would achieve extremely fast queries but is prohibited by its huge time and storage requirement. Thus, a better approach turns to extract and process some helpful hints that can effectively accelerate the queries. A lot of research has been tried to balance between preprocessing and query times, most of which uses graph partition techniques [3, 4, 5, 8] for the original problem decomposition. Möhring et al. [3] and Maue et al. [4] partition the graph into regions and employ goal-directed preprocessing techniques to eliminate unnecessary searches. Rajagopalan et al. [6] combine goal-directed heuristics with hierarchical and preprocessing techniques, and constitute an abstraction graph model for efficient storage and path computation on node-weighted graphs. Similarly, Jung et al. [7] develop a hierarchical graph model based on the spatial partitioning [5] of graphs, and then apply a variation of A* algorithm to accelerate the query process.

Intuitively, the choice of underlying partition methods may not affect the accuracy of a query algorithm, but the query execution can be effectively accelerated through appropriate partition of graphs. In addition, specific graph partition objectives should be employed to cater to different types of query algorithms. However, recent research related to path computations focuses more on the routing algorithm design, where the planar graph partition method is generally used just as a tool for decomposing the networks, with some simple partition objectives such as balance of subnetwork size or minimization of boundary nodes [5, 8, 9]. Less has been done to analyze the impact of graph partition objectives on the speedup of a shortest path algorithm.

In this work, we consider exact shortest path queries in large networks. The main goal is to accelerate the path queries based on an effective partition of graphs without a layout or an embedding, using fast preprocessing that maintains a small amout of auxiliary data. The network is abstracted in a hierarchical fashion, and the query algorithm is executed by combining hierarchical and goal-directed heuristics. The efficiency of the query algorithm towards different partitions is analyzed, with comparison experiments conducted on a large city road network.

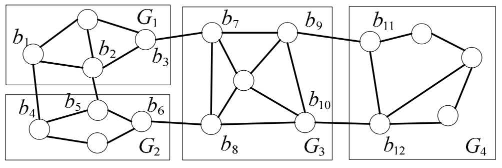
II. Hierarchical Graph Model

Let $G=(V, E)$ be a graph, where each node in V represents network objects, i.e., the intersecting points of roads in a road network, routers in the Internet, or individuals in a friendship network. Edges $E=\{(u, v)|(u, v\in V)\wedge(u\neq v)\}$ correspond to the connections between the preceding objects. Each node is assigned a weight by a function $w_v: V\rightarrow\Re^{\geq 0}$. The length of a path P is the sum of the weights of all nodes on the path, denoted by $w_v(P)$, and the distance $d_G(s,t)$ between two nodes s and t is defined by the length of the shortest path from s to t in G.

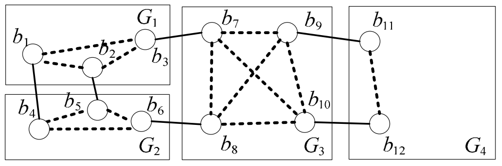
Given a graph $G=(V, E)$, a collection $\kappa=\{G_1(V_1, E_1), \ldots, G_k(V_k, E_k)\}$ of pairwise disjoint sets $V_i\subseteq V$, $E_i\subseteq E$, $1\leq i\leq k$, such that $\bigcup_{i=1}^{k} V_i = V$ and $\bigcup_{i=1}^{k} E_i \subset E$ is called a partition of G and each set $G_i(V_i, E_i)$, $1\leq i\leq k$ a subgraph of G. For any

Manuscript received March 5, 2011. This work was supported by the Major State Basic Research Development Program of China (973 Program) under Grant 2010CB731400 and the NSF of China under Grant 61074125.

The authors are with the Department of Automation, Shanghai Jiao Tong University, and Key Laboratory of System Control and Information Processing, Ministry of Education of China, Shanghai 200240, China (e-mail: qsong@sjtu.edu.cn; xfwang@sjtu.edu.cn).



(a)



(b)

Fig. 1. Two-level graph hierarchy (G, G^H). (a) Original network G with four subgraphs with boundary node sets $\{b_1, b_2, b_3\}$, $\{b_4, b_5, b_6\}$, $\{b_7, b_8, b_9, b_{10}\}$, and $\{b_{11}, b_{12}\}$, respectively. (b) High-level graph G^H constructed from subgraphs G_1, G_2, G_3, and G_4, where dashed lines denote community edges, and solid lines denote intercommunity edges.

node $v \in V$, let $Sub(v)$ denote the subgraph to which v belongs to. A node $u \in V_i$ is called a border node of G_i if there exists an edge $(u, v) \in E$ with $v \in V_j$ and $i \neq j$, and an inner node of G_i otherwise; G_j is then called a neighbor subgraph of u, denoted by $Ns(u)$, and the subgraphs G_i and G_j are said to be adjacent. The set of all border nodes of G_i is denoted by $B(G_i)$. An edge $(u, v) \in E$ is called an intercommunity edge if u, v belong to adjacent subgraphs G_i and G_j respectively. The inter-community edge set between G_i and G_j is denoted by $I(G_i, G_j)=\{(u, v) \in E | (u \in B(G_i)) \wedge (v \in B(G_j)) \wedge (i \neq j)\}$. Obviously, $\bigcup_{i,j} I(G_i, G_j) = E - \bigcup_i E_i$, with $1 \leq i, j \leq k$ and $i \neq j$.

Definition 1: Given a partition $\kappa = \{G_1, G_2, \ldots, G_k\}$ of G, the community edge set of subgraph G_i ($1 \leq i \leq k$) is defined by

$$C(G_i) = \{(u,v) \,|\, (u,v) \in (B(G_i) \times B(G_i)) \wedge (u \xrightarrow{d_{G_i}(u,v)} v) \wedge (u \neq v)\}.$$

Definition 2: Given a partition $\kappa = \{G_1, G_2, \ldots, G_k\}$ of G, the high-level graph of G is defined by $G^H=(V^H, E^H, W^H)$.

1) $V^H = \bigcup_{i=1}^{k} B(G_i)$.

2) $E^H = (\bigcup_{i,j} I(G_i, G_j)) \cup (\bigcup_i C(G_i))$, with $1 \leq i, j \leq k$ and $i \neq j$.

3) For any edge $(u, v) \in E^H$, its edge weight $w_e^H(u,v) := d_{G_i}(u,v)$ if u, v belong to the same subgraph, and 0 otherwise. For all nodes in V^H, the weight $w_v^H := w_v$.

Definition 3: Given a partition $\kappa = \{G_1, G_2, \ldots, G_k\}$ of G. For any shortest path $P=(s, \ldots, s', u, \ldots, v, t', \ldots, t)$ that passes a subgraph G_i, $1 \leq i \leq k$, the traversing distance is defined by

$$Td(G_i) := d_G(u, v),$$

where u is the first node and v is the last node from G_i on P, $Sub(s') \neq Sub(t')$, $G_i \subseteq Ns(s')$, and $G_i \subseteq Ns(t')$. Note that u, v may be duplicated when the path passes G_i via just one border node. In addition, the traversing distance $Td(G_i)$:=0 if G_i has only one neighbor subgraph.

The definition of traversing distance makes sense when each border node of the graph is connected to only one intercommunity edge. Then, the minimum traversing distance of subgraph G_i implies the minimum length increase that may happen when a shortest path passes the subgraph G_i through a community edge. For the subgraph G_i which has more than one neighbor, the traversing distance set can be calculated by

$$Td(G_i) = \{d_G(u,v) \,|\, (u,v) \in (B(G_i) \times B(G_i)) \wedge (Ns(u) \neq Ns(v)) \wedge (u \neq v)\}.$$

Definition 4: Given a partition $\kappa = \{G_1, G_2, \ldots, G_k\}$ of G, the abstraction graph of G is defined by $G^A=(V^A, E^A, W^A)$, where

1) each node $u \in V^A$ is called the supernode representing the subgraph formed by $\{v \in V | Sub(v)=u\}$;

2) each edge $(u, v) \in E^A$ represents the collection of edges $\{(u', v') \in E \,|\, (Sub(u') = u) \wedge (Sub(v') = v)\}$;

3) the weight of any node $u \in V^A$ is defined by $w_v^A := \min Td(u)$.

For example, Fig.1(a) shows a graph G and its four subgraphs G_1, G_2, G_3, and G_4. Fig.1(b) shows the high-level graph constructed from G_1 to G_4, where each subgraph is represented as a complete graph composed of its border node set and the community edge set. The intercommunity edge can be thought of as forming bottlenecks between subgraphs. The corresponding abstraction graph is shown in Fig.2, which contains far fewer nodes and edges. Thus, it is possible to perform a search first on the abstraction graph for an estimate of the path length, which is crucial in eliminating unnecessary computations. The estimate gets to the real shortest path value when the maximum traversing distance equals the minimum one for every subgraph, and the query execution is most efficient at this ideal case. This observation inspires the partition method with the objective of balancing traversing distance to be presented in the next section.

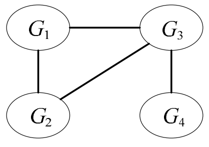


Fig. 2. Abstraction graph G^A for graph in Fig. 1, with supernodes denoting the subgraphs and edges representing the connections between subgraphs.

III. Graph Partitioning

In this section, we present our graph partition model and the associated notations followed by a description of the BTD partitioning algorithm for pursuing Balanced Traversing Distance partitions.

A. Mathematical Model

Suppose that the original network is partitioned into k subgraphs $G_1, G_2, \ldots, G_k$. Let $s(G_i)$ be the size of subgraph G_i, and R_i the ratio of the maximum traversing distance to the minimum traversing distance for subgraph G_i,

$$R_i := max(Td(G_i))/min(Td(G_i)),$$

with $1 \le i \le k$. The ratio R_i:=1 if G_i has only one neighbor subgraph. For any node $u \in B(G_i)$, $Num(u)$ represents the number of intercommunity edges incident to u. The permitted upper and lower bounds for subgraph size are denoted by δ_U and δ_L, respectively.

Inspired by the ideal case of graph partition and path queries, we propose a mathematical model as following:

$$min \{ R_i | \forall\ 1 \le i \le k \}$$

subject to

$$Num(u)=1,\ u \in B(G_i) \quad (1)$$

$$\delta_L \le s(G_i) \le \delta_U \quad (2)$$

Here, the traversing distance ratio R_i approaches its minimum value 1 when the maximum traversing distance gets very close to the minimum one for each subgraph. Constraint (1) ensures that the shortest path will pass through a subgraph via at least one community edge rather than just one border node based on Definition 3. Constraint (2) aims to avoid too large/small partitions and thereby balances the path searching within each subgraph.

B. BTD Algorithm

To get the graph decomposition satisfying the objective and constraints in (1)-(2), the BTD algorithm comprises two phases.

Phase 1. Subgraph initialization. During this phase, the network is divided into a series of small subgraphs. Initially, all nodes are unmarked. Then, nodes are considered one by one, and for each unmarked node u of degree two or more, we proceed as follows.

Step1: Mark the node u with a new subgraph number "*sn*", and maintain a set "*temp*" for the nodes added to this subgraph in turn. Set

$$temp=\{unmarked\ neighbors\ of\ u\}.$$

Generally, we repeat this process of adding unmarked neighbors several times to avoid too small subgraphs produced during initialization.

Step2: Remove the first node v from *temp*, and judge whether v has a neighbor node x in a different subgraph. If not, go to *Step3*; Otherwise, add all unmarked neighbors of v and x to the end of the set *temp*, mark them with the subgraph number *sn*, and go to *Step4*.

Step3: For any unmarked neighbor node x, judge whether x has a neighbor node in a different subgraph, add the neighbor x to the end of *temp* and mark it with the subgraph number *sn* if the condition holds, and then go to *Step4*.

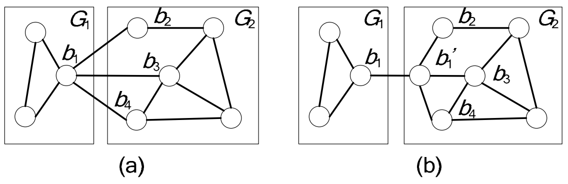


Fig. 3. Node adding process. (a) Original graph partitions with boundary node sets $\{b_1\}$ and $\{b_2, b_3, b_4\}$. (b) Modified graph partitions with boundary node sets $\{b_1\}$ and $\{b_1'\}$.

Step4: Exit if the set *temp* becomes null; Otherwise, go to *Step2* and continue.

At the end of the sweep, we randomly put the unmarked degree one node in its neighbor subgraph and mark it with the subgraph number. At that point, all nodes are marked, and each node is adjacent to one or two subgraphs (for inner nodes and border nodes, respectively). Then, for any border node which is adjacent to more than one intercommunity edge, we turn to add a zero-weighted node to replace the multiple nodes adjacent to it. In Fig.3(a), node b_1 is connected to three intercommunity edges. By adding a node b_1' in its neighbor subgraph and linking up the endpoints of the original intercommunity edge via b_1', we get a modified graph where each border node is connected to only one intercommunity edge, as shown in Fig.3(b).

Phase 2. Subgraph agglomeration. During this phase, a heuristic agglomeration process is performed on the graph partition produced in phase 1, with the purpose of reducing the ratio R_i and regulating the subgraph size to $[\delta_L, \delta_U]$.

First, we need to compute the ratio R_i for each subgraph. This can be fulfilled by a local shortest path tree construction from each node in the high-level graph (similar strategies can be found in [10]). At the end of the loop, we get the traversing distance set for each subgraph, and the ratio R_i is computed by dividing the maximum traversing distance by the minimum one. Then, for each subgraph G_i we evaluate the degradation of R_i that would take place by merging a neighboring subgraph to G_i. Combine G_i with the neighbor for which this degradation is maximal, but only if the combined subgraph size is below δ_U. This process is repeated for all subgraphs until no further improvement can be achieved (or as soon as the ratio R_i is below an acceptable value) and until the subgraph size is within the region of $[\delta_L, \delta_U]$ for all subgraphs. This may consume a fair amount of time but is worthy since the graph partition usually need not be applied repeatedly. The constructed community edges are employed to facilitate the computation of the traversing distance ratio R_{com} for a combined subgraph G_{com}. Still, more techniques need to be introduced for further acceleration.

Note that the subgraph combination may simultaneously affect the traversing distance ratio R_j of a neighboring subgraph G_j, thus we need to update the neighbor information and recompute R_j for the affected subgraphs. In addition, the subgraph agglomeration process will certainly not affect the number of intercommunity edges incident to a border node so that Constraint (2) satisfies.

IV. Hierarchical Routing Algorithm

In this section, we introduce an efficient Hierarchical Subgraph Pruning (HSP) algorithm which will benefit greatly from the BTD method for route computation on large graphs. Before formally presenting the algorithm, we introduce the precomputation task for the static and dynamic scenarios.

A. Preprocessing

Suppose the original network G has been partitioned into k subgraphs G_1, G_2, …, G_k. The high-level graph G^H can be easily constructed by extracting the border nodes, the inter-community edges, and adding the community edges between every pair of border nodes of each subgraph. As introduced previously in the BTD algorithm (Phase 2), the traversing distance set $Td(G_i)$ can be obtained via a local search on G^H. And the abstraction graph G^A is then constructed by mapping each subgraph G_i as a node, with $\min\{Td(G_i)\}$ as its weight, and adding an edge between any pair of nodes whose corresponding subgraphs are adjacent. In fact, most of the preprocessing has been finished at the graph partition stage when BTD algorithm is employed as the partition tool.

In dynamic scenarios, the preprocessed data have to be updated temporarily before a path query. Naturally, we will not re-partition the graphs every time since the weight changes may not significantly affect the partition quality of a graph, which is measured by the average traversing distance ratio $\overline{R}$. For the case that the additional computation caused by an increasing $\overline{R}$ is still lower than that of the graph re-partitioning, we turn to use the initial partitions and reconstruct the related high-level subgraph and abstraction subgraph for the affected areas, which yields a low update cost. In case that the weight of a community edge of G_i is decreasing, we need to compute the optimal community edge length by performing a local search on G^H from each border node of G_i, and then update the traversing distance set $Td(G_i)$ and the corresponding supernode weight on G^A simultaneously; For other cases, we will neither perform the local tree construction nor update the supernode weight. The real weight value is definitely no less than the current weight. Thus, we will never overestimate the path length by using a relatively lower supernode weight, though this may somewhat affect the query efficiency.

B. HSP Algorithm

The HSP algorithm generally covers two scenarios. Here, we mainly discuss the case that the source and destination nodes are in two distinct subgraphs. For those node pairs in the same subgraph, we can compute the shortest path either on the original network G, or on a rebuilding search area defined by [10].

Phase 1: Path length evaluation. During this phase, the accumulated minimum traversing distance from t to several other subgraphs is computed, and the path length is estimated

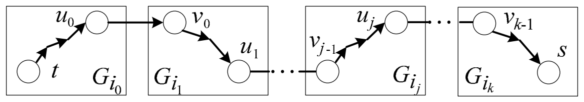


Fig. 4. Path length evaluation for the subgraphs of path SP over the high-level graph G^H. s is the source node, and t is the destination node. (u_j, v_j) is an intercommunity edge between subgraphs G_{i_j} and $G_{i_{j+1}}$ with $1 \le j \le k$-1.

based on a practical route between s and t, which provides an initial upper bound on the length of the shortest path.

The algorithm first grows a shortest path tree over G^A, starting from the destination supernode G_t (which contains the destination node t). At the initialization stage, we assign a distance value $\underline{d}(G_i, t)$ to every supernode G_i. Set it to zero for the destination supernode G_t and infinity for all other supernodes. Then, we perform an ordinary Dijkstra search from G_t, and prune the search temporarily when the source supernode G_s is settled. Let SP= (G_t (G_{i_0}) => G_{i_1} => G_{i_2} ··· =>G_s (G_{i_k})) represent the optimal path between supernodes G_s and G_t on G^A. For any settled supernode G_i, $\underline{d}(G_i, t)$ gives a minimum traversing distance from G_i to t. To compute an upper bound on the path length between s and t, we set $\hat{d}(G_i,t) := \infty$ for every subgraph G_i, representing the upper distance from G_i to t, and update such distance for the subgraphs on path SP. For each subgraph pair (G_{i_j} , $G_{i_{j+1}}$) on SP, where $1 \le j \le k$-1, we proceed as follows:

Find an intercommunity edge (u_j, v_j) between subgraphs G_{i_j} and $G_{i_{j+1}}$ on G^H, where u_j, v_j are border nodes of G_{i_j} and $G_{i_{j+1}}$, respectively. The upper distance $\hat{d}(G_{i_j},t)$ is then updated sequentially by adding the weight of the community edge ($v_{j\text{-}1}$, u_j) to $\hat{d}(G_{i_{j-1}},t)$, where $\hat{d}(G_{i_0},t)$ is assigned with the maximum weight of the community edges incident to u_0.

For the source subgraph G_s (which contains the source node s), we update its upper distance $\hat{d}(G_s,t)$ by adding the maximum weight of the community edges incident from $v_{k\text{-}1}$ to $\hat{d}(G_{i_{k-1}},t)$. Naturally, R= (t, u_0, v_0, …, $u_{k\text{-}1}$, $v_{k\text{-}1}$, s) gives a practical route from t to s, as shown in Fig.4. The weight of the route R is certainly not more than $\hat{d}(G_s,t)$ since we use a maximum weight at the source and destination subgraphs during the upper distance computation. Thus, the upper bound of the shortest path between s and t can be initialized to

$$\hat{d}(s,t) := \hat{d}(G_s,t) .$$

Continue the preceding shortest path search from G_t on G^A, until the distance of the currently visited supernode $\underline{d}(G_i, t)$ is not less than $\hat{d}(s,t)$, or else terminate the search if all supernodes have been marked. The sub areas denoted by the unsettled supernode can not appear on the final shortest path since any path passing them will yield in a larger path length.

Phase 2: Hierarchical search with pruning. In the second phase we perform a shortest path search starting from the source node s. During initialization, we assign a distance value $d(s, u)$ to every node u. Set it be the weight of s for node s and infinity for all other nodes. Record the previously visited node $Pre(u)$ on the optimal path for every node u and set it to null in the beginning. Mark the source node s as current and begin the search.

Step 1: For current node u (u=s in the beginning), judge whether u is in the source or destination subgraph. If not, go to *Step 2*; Otherwise, relax the nodes v adjacent to u on the original network G, which amounts to replace $d(s, v)$ with a new value $d(s, u)+w_v(v)$, but only if this value is smaller. Here, $w_v(v)$ denotes the weight of node v. Overwrite the predecessor $Pre(v)$ if the distance to v is updated, and go to *Step 4*.

Step 2: Judge whether the current node u and its predecessor $Pre(u)$ are located in the same subgraph: If not, go to *Step 3*; Otherwise, compute the lower bound $\underline{d}(s, u, t)$ on the length of the shortest path between s and t by

$$\underline{d}(s, u, t)= d(s, u)+ \underline{d}(Ns(u), t),$$

where $Ns(u)$ represents the neighbor subgraph of u. For $Ns(u)$, the subgraph pruning condition satisfies only when this lower estimate exceeds the upper bound $\hat{d}(s,t)$. The condition holds thereafter because any node marked later will have a larger $d(s, u)$ value. Mark the subgraph $Ns(u)$ as pruned, and since then all searches grow into that subgraph will be pruned. Go to *Step 4* without performing any relaxation if subgraph $Ns(u)$ has been pruned; Otherwise, relax the endpoint v of the intercommunity edge incident to u, and replace the distance to v by $d(s, u)+w_v(v)$ if this yields a lower $d(s, v)$ value. Update the predecessor of v, and then go to *Step 4*.

Step 3: Tighten the upper bound $\hat{d}(s,t)$ as follows if u is a node of subgraph $G_{i_j} \in SP$ $(1 \le j \le k)$:

$$\hat{d}(s,t) = \min[\hat{d}(s,t), d(s, Pre(u)) + \hat{d}(Sub(u),t) + |d_{Sub(u)}(v_{j-1},u) - d_{Sub(u)}(v_{j-1},u_j)|].$$

Here, $Sub(u)$ denotes the subgraph to which u belongs to, and $d_{Sub(u)}(x, y)$ gives the weight of the community edge (x, y) of subgraph $Sub(u)$.

Relax the community edges incident to u, and for each endpoint v (except u) of the community edge, replace $d(s, v)$ with $d(s,u)+d_{Sub(u)}(u, v)-w_v(u)$ if this achieves an improvement. Update the predecessor of v, and go to *Step 4*.

Step 4: Choose the node u' with minimum $d(s,u')$ value from all the unmarked nodes. Exit if the node $u'=t$; Otherwise, mark u' as current, and go to *Step 1* and continue.

The HSP algorithm can be used effectively to address problems in dynamic scenarios. For example, a new event in the environment may affect the weights of several nodes on the path previously obtained, ever since the path traversal has been initiated from a source node s. The dynamic path planning problem involves a path recomputation from the current node c to the destination node t. In case that the node on $path_{c\to t}$ has an increased weight or the node outside the path has a decreased weight, we need to update the preprocessed data temporarily and then compute a new path from c to t, where the preceding algorithm applies with s replaced by c. For other cases, we will neither update the preprocessed data nor recompute the shortest path, as the path retrieved is already the optimal one.

TABLE I
NETWORK PARTITION SCHEMES INVOLVED IN PERFORMANCE EVALUATION

Partition Schemes	p	$\overline{R}$	n^H	m^H
Scheme 1	2543	8.36	43282	1068699
Scheme 2	1712	9.11	36240	971335
Scheme 3	894	9.31	22960	789790
Scheme 4	510	13.64	19115	761172
Scheme 5	894	16.50	25607	840928

p represents the number of subgraphs; n^H and m^H represent the number of nodes and edges in the corresponding high-level graph.

V. EXPERIMENTAL EVALUATION

To verify the validity of our hierarchical routing algorithm, we consider the New York City road network with 366923 nodes and 1557956 edges [11], where nodes and edges denote the roads and the intersecting points of roads respectively. Each node is assigned a weight representing the cost of the road, thus the queries compute the minimum cost route on such network. We evaluate the performances of HSP using different partition schemes, compared with two well-known approaches, i.e., HIPLA [6] and hierarchical Dijkstra algorithm (Hi-dijkstra), and analyze the impact of the average traversing distance ratio and the number of subgraphs on the query efficiency. All the algorithms were developed in Matlab 7.8.0 (R2009a) and conducted on an Intel Xeon X5482 Dual Core processor with 32GB of RAM. The system ran Microsoft Windows Vista.

A. Test Generation and Description

A total of five network partition schemes are employed in our testing, as shown in Table I. Schemes 1 to 4 are produced at different stages of a BTD algorithm, accompanied by the subgraph agglomeration. Generally, the average traversing distance ratio $\overline{R}$ follows a downward trend as the subgraph agglomerates, though it may fluctuate slightly. Here, we select the subgraph partition with a tentative rising $\overline{R}$ value for analyzing the effect of the number of subgraphs and $\overline{R}$ on a query algorithm. For completeness, another partition scheme (Scheme 5) satisfying the constraints in (1)-(2) is generated for comparison, which has the same size as scheme 3 but with a larger $\overline{R}$ value. Each scheme is made to solve a set of 1000 problems using the same randomly generated source and destination nodes.

B. Performances of Various Algorithms

Table II shows the average execution time and accuracy of HSP, HIPLA, and Hi-dijkstra on Schemes 1 to 5, where the numbers presented are average values over 1000 problems. We observe that HSP requires much less computation time

TABLE II
COMPARISON OF AVERAGE EXECUTION TIMES AND ACCURACY FOR SCHEMES 1-5

Algorithm	Scheme 1			Scheme 2			Scheme 3			Scheme 4			Scheme 5		
	t	$\overline{E}$ %	E_{max} %	t	$\overline{E}$ %	E_{max} %	t	$\overline{E}$ %	E_{max} %	t	$\overline{E}$ %	E_{max} %	t	$\overline{E}$ %	E_{max} %
Hi-dijkstra	3.51	0	0	3.17	0	0	2.60	0	0	2.36	0	0	3.05	0	0
HSP	0.84	0	0	0.74	0	0	0.54	0	0	0.69	0	0	1.00	0	0
HIPLA	0.20	38.16	154.57	0.19	40.13	242.50	0.22	36.09	164.36	0.17	36.34	164.48	0.19	39.66	141.47

t represents the average execution time (in seconds); $\overline{E}$ and E_{max} represent the average and maximum errors observed in various algorithms.

compared with Hi-dijkstra and it computes the optimal path for all node pairs. Though HIPLA achieves the best efficiency in all the partition schemes, the large errors produced are inevitable due to its naive heuristics. It is noted that the path identified by HIPLA is on average 38% longer than the optimal path (much larger than that is reported in [6]), with an average maximum error of up to 173%, which makes HIPLA unsuitable for high-precision path queries.

C. *Effects of $\overline{R}$ vs. p*

We analyze the execution time of HSP on Schemes 1-4 with different values of $\overline{R}$ and p. We find that the efficiency of HSP is affected simultaneously by these two factors. Naturally, the efficiency of a hierarchical algorithm will be enhanced with a decreasing number of subgraphs, as the high-level graph contains fewer nodes and edges. Thus, the execution time of HSP drops in Schemes 1 to 3 when the increase in $\overline{R}$ has not become a leading factor. However, when $\overline{R}$ exceeds a certain value, the efficiency of HSP will be greatly weakened since the increase in $\overline{R}$ leads to an even larger increase in the search space. Hence, the execution time of HSP increases in Scheme 4 though its number of subgraphs is the smallest.

To further illustrate the effect of $\overline{R}$ on HSP, we compare the query execution on Schemes 3 and 5 involving the same number of subgraphs. From Table II, we can observe that the execution time of HSP is much lower on Scheme 3 than on Scheme 5. We also notice that the high-level graphs formed by Schemes 3 and 5 are almost the same in size from Table I. Thus, the performance degradation on Scheme 5 can only be caused by the increasing value of $\overline{R}$. Fixing the number of subgraphs, the efficiency of HSP will be improved via the decrease of $\overline{R}$, which coincides with the starting point of the BTD partitioning method.

D. *Discussion*

We analyze the best and worst case runtime complexity of HSP to show the gain in performance. The best case happens when $\overline{R}$ approaches the minimum value 1; then, HSP will search routes only within the subgraph of path *SP*. All the other subgraphs will be pruned by the subgraph pruning condition, which yields a similar search space as HIPLA. While on the contrary when $\overline{R}$ is quite large, few subgraphs will be pruned during the search of HSP and the computational complexity gets close to the Hi-dijkstra. Thus, the runtime complexity of HSP is between that of Hi-dijkstra and HIPLA, which can be viewed as two extremes of HSP.

VI. CONCLUSION

In this paper, we develop an effective graph partition method for accelerating the path queries on large node-weighted networks. We propose a new heuristic hierarchical routing algorithm based on our hierarchical graph model, which could compute optimal routes in both static and dynamic environments. The proposed method can also be applied to edge-weighted graphs through several conversions and is focused in another piece of our work. As part of future research, it would be beneficial to quantify the effect of the number of subgraphs and the average traversing distance ratio on the performance improvement of a query algorithm so as to determine the optimum values. Also, it is worth developing more fast and effective partition methods to further reducing the traversing distance ratio.

REFERENCES

[1] E. W. Dijkstra, "A note on two problems in connexion with graphs", *Numer. Math.*, vol. 1, pp. 269-271, 1959.

[2] L. Fu, D. Sun, L. R. Rilett, "Heuristic shortest path algorithms for transportation applications: state of the art", *Comput. Oper. Res.*, vol. 33, pp. 3324–3343, 2006.

[3] R. Möhring, H. Schilling, B. Schütz, D. Wagner, and T. Willhalm, "Partitioning graphs to speed up Dijkstra's algorithm", *ACM J. Exp. Algor.*, vol.11, article no.2.8, pp. 1-29, 2006.

[4] J. Maue, P. Sanders, and D. Matijevic, "Goal directed shortest path queries using precomputed cluster distances", *ACM J. Exp. Algor.*, vol.14, article no.3.2, pp. 1-27, 2009.

[5] Y. W. Huang, N. Jing, and E. A. Rundensteiner, "Effective graph clustering for path queries in digital map database", in *Proc. CIKM*, Rockville, 1996, pp. 215-222.

[6] R. Rajagopalan, K. G. Mehrotra, C. K. Mohan, and P. K. Varshney, "Hierarchical path computation approach for large graphs", *IEEE Trans. Aerosp. Electron. Syst.*, vol. 44, no. 2, pp. 427-440, 2008.

[7] S. Jung and S. Pramanik, "An efficient path computation model for hierarchically structured topographical road maps", *IEEE Trans. Knowl. Data Eng.*, vol. 14, no. 5, pp. 1029-1046, 2002.

[8] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs", *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359-392, 1998.

[9] M. B. Habbal, H. N. Koutsopoulos, and S. R. Lerman, "A decomposition algorithm for the all-pairs shortest path problem on massively parallel computer architectures", *Transp. Sci.*, vol. 28, no. 4, pp. 292-308, 1994.

[10] Q. Song and X. F. Wang, "Efficient routing on large road networks using hierarchical communities", *IEEE Trans. Intell. Transp. Syst.*, vol. 12, no. 1, pp. 132-140, Mar. 2011.

[11] [Online]. Available:http://www.dis.uniroma1.it/~challenge9/download.shtml