RESEARCH ARTICLE

Accurate and fast path computation on large urban road networks: A general approach

Qing Song1,*, Meng Li1, Xiaolei Li2

1 School of Electrical Engineering, University of Jinan, Jinan, P. R. China, 2 School of Control Science and Engineering, Shandong University, Jinan, P. R. China

* qsong83@gmail.com

Abstract

Accurate and fast path computation is essential for applications such as onboard navigation systems and traffic network routing. While a number of heuristic algorithms have been developed in the past few years for faster path queries, the accuracy of them are always far below satisfying. In this paper, we first develop an agglomerative graph partitioning method for generating high balanced traverse distance partitions, and we constitute a three-level graph model based on the graph partition scheme for structuring the urban road network. Then, we propose a new hierarchical path computation algorithm, which benefits from the hierarchical graph model and utilizes a region pruning strategy to significantly reduce the search space without compromising the accuracy. Finally, we present a detailed experimental evaluation on the real urban road network of New York City, and the experimental results demonstrate the effectiveness of the proposed approach to generate optimal fast paths and to facilitate real-time routing applications.

Introduction

In onboard navigation systems, a primary function is to find the route from the current location of a vehicle to a desired destination with a minimum expected travel time. Vehicle’s travel time on road is usually considered to be the single variable in determining the route. However, in applications of urban road networks it is often essential to take into account the delay time at road intersections, which may be caused by traffic signals or other interference factors such as people going by. Furthermore, most times the travel time on road is trivial while comparing with the delay at road intersections, creating the possibility to use the delay time for measuring the time taken on a route.

In network theory, this corresponds to the shortest path problem, where the delay time at road intersections can be mapped as a kind of weight associated with the node. In recent years, the popular of intelligent transportation system applications have stimulated widespread research interests [1–5], yielding in various types of shortest-path computation algorithms. From the classical Dijkstra algorithm [6], which is practically far too slow for real road network applications, to the Bidirectional search [7], and the A’ algorithm [8], strategies are used to reduce the search space from the initial one big circle to two smaller ones, and to an approximate ellipse in the A’ algorithm. Acceleration strategies in this first stage focus on the
improvement of the search strategy itself to restrict the number of nodes visited \[9, 10\], while using additional data of the road network such as geographical coordinates, and moreover focus on an optimization of the data structures \[11\].

The second stage of research focuses on very efficient path computation strategies that are practically usable in real-time onboard navigation systems. Auxiliary data are generated through a preprocessing task, and several special properties of road networks such as sparse, almost planar, and inherent hierarchical are exploited for design of high speed search techniques. Naturally, a pre-computation of the shortest paths between all node pairs would achieve extremely fast queries, however, it is unpractical for real road network applications due to its huge time and storage requirement. Plenty of research thus turned to extract and preprocess part of the helpful data that could effectively accelerate the queries \[12–17\]. Möhring et al. \[13\] partitioned the graph into regions and pre-computed a flag for each edge towards each region, indicating whether the edge lies on the shortest path to any node of this region, and the query algorithm only considered edges whose flag towards the target region was true. Maue et al. \[14\] pre-computed the shortest connection between all pairs of divided clusters and maintained an upper and lower bound during the query to prune the unnecessary searches.

Hierarchical strategy has proven to be very effective in reducing the search complexity of the algorithm, and there are different types of hierarchical models and heuristic search strategies for road network applications \[15,16,18,19\]. One makes use of the inherent hierarchical topologies of road networks \[18–21\], where the graph hierarchy is defined and constructed based on road categories, road lengths, speed limits, etc. The resulting path computation algorithm designed on such type of hierarchical model usually suffers from high computational errors, and it needs an assistance of a human expert to ensure the connectivity of the hierarchy as well as to improve the accuracy of path computation \[18\], which limits its applications. The other approach focuses more on how to partition and manage a network. The hierarchy is constructed based on the partition of the network, which are depicted in different types of models \[15, 16\]. The accuracy and the efficiency of the resulting path computation algorithm depends both on the model and on the search strategies. Jung and Pramanik \[15\] pre-computed all pairs of shortest paths between the boundary nodes of each subgraph and applied a variation of A* algorithm to accelerate the subsequent path computation. Rajagopalan et al. \[16\] present a sampling approach for estimating the risks of supernodes, and constituted a hierarchical abstraction graph model for fast near-optimal path computations.

In this work, we address the problem of efficient path computation on large urban road networks, where the delay time at road intersections is used for measuring the time taken on a route and can be treated as a kind of weight associated with the node. We focus on an accurate and more general approach, that is without any prior knowledge of node coordinates, road types, and without any prior knowledge of the network topology. First, we propose a new graph partitioning method with the objective of producing high balanced traverse distance partitions, and we constitute a three-level graph model based on the graph partition scheme for structuring the urban road network. Second, we develop a new hierarchical path computation algorithm that supports accurate and fast path queries in large node-weighted graphs. The algorithm benefits from the hierarchical graph model and utilizes a region pruning strategy to significantly reduce the search space without compromising the accuracy. We instantiate our approach with real-world data and study its properties, in particular the network partition effects and the performance of various heuristic algorithms, in a realistic and challenging urban environment.

The rest of the paper is organized as follows. In the next section, we introduce the basic definitions and notations used in the rest of the paper followed by a precise description of the
hierarchical graph model. Further, we present the graph partitioning method. Then, we introduce the hierarchical path computation algorithm and we examine the performance of the hierarchical optimization scheme in the further section. The paper finishes with a conclusions and future directions.

**Modelling of urban road network**

In this section, we present a three-level graph model for structuring the urban road network. First we give the basic definitions and notations used in the hierarchical graph model.

**Basic graph definitions**

Definition 1. Given two graphs \( G = (V,E,W) \) and \( G_u = (V_u,E_u,W_u) \) such that \( V_u \subseteq V, E_u \subseteq E \), and \( W_u \subseteq W \). Then, \( G_u \) is called the subgraph of \( G \).

Suppose that \( G \) is partitioned into a set of subgraphs \( G_1(V_1,E_1,W_1), G_2(V_2,E_2,W_2), \ldots, G_p(V_p,E_p,W_p) \), then the following relation satisfies:

\[
V_1 \cup V_2 \cup \cdots \cup V_p = V, \quad E_1 \cup E_2 \cup \cdots \cup E_p \subseteq E \\
V_i \cap V_j = \emptyset, \quad E_i \cap E_j = \emptyset \quad \text{where} \quad 1 \leq i, j \leq p \quad \text{and} \quad i \neq j.
\]

Definition 2. Given a graph \( G = (V,E,W) \), such that for any node \( i, j \in V \) there exists an edge \((i, j) \in E\), then \( j \) is called the adjacent node of \( i \), and vice versa. The adjacent nodes set of \( i \) is defined by:

\[
\text{Adjacent}(i) = \{ j | (i, j) \in E \quad \text{and} \quad j \in V \} 
\]

Definition 3. Given a partition \( P = \{G_1, G_2, \ldots, G_p\} \) of \( G \), such that for any node \( i \in V_u \) there exists an adjacent node \( j \in \text{Adjacent}(i) \) which satisfies \( j \in V_v \) and \( V_u \neq V_v, 1 \leq u, v \leq p \). Then, \( i \) is called the border node of subgraph \( G_u \). The border nodes set of subgraph \( G_u \) is denoted by \( \text{Border}(G_u) \). Subgraph \( G_i \) is called the neighbor subgraph of node \( i \), denoted by \( \text{Neighbor}(i) \).

Definition 4. Given a partition \( P = \{G_1, G_2, \ldots, G_p\} \) of \( G \), such that for any node \( i, j \in V \) there exists an edge \((i, j) \in E \) which satisfies \( i \in \text{Border}(G_u) \) and \( j \in \text{Border}(G_v) \), \( 1 \leq u, v \leq p \) and \( u \neq v \). Then, edge \((i, j) \) is called the cut edge between subgraphs \( G_u \) and \( G_v \). The cut edges set between \( G_u \) and \( G_v \) is defined by:

\[
\text{Cut}(G_u, G_v) = \{ (i,j) | (i,j) \in E \quad \text{and} \quad (i \in \text{Border}(G_u) \quad \text{and} \quad j \in \text{Border}(G_v)) \}
\]

Obviously, the cut edges separate the subgraphs to mutually disjoint sets with no overlapping nodes or edges, thus we have \( \bigcup_{u,v} \text{Cut}(G_u, G_v) = E - \bigcup_{u,v} E_{uv} \) where \( 1 \leq u, v \leq p, u \neq v \).

Definition 5. Given a partition \( P = \{G_1, G_2, \ldots, G_p\} \) of \( G \), for any subgraph \( G_u (1 \leq u \leq p) \) the connect edges set within the subgraph between all border nodes is defined by:

\[
\text{Connect}(G_u) = \{ (i,j) | (i,j) \in (\text{Border}(G_u) \times \text{Border}(G_u)) \quad \text{and} \quad (i \nrightarrow j) \quad \text{and} \quad (i \neq j) \},
\]

where the weight of any connect edge \((i,j)\) is computed by the shortest path distance from node \( i \) to \( j \) within subgraph \( G_u \).

Definition 6. Given a partition \( P = \{G_1, G_2, \ldots, G_p\} \) of \( G \), for any subgraph \( G_u (1 \leq u \leq p) \) the traverse distance set of the subgraph is defined by:

\[
\text{Traverse}(G_u) = \{ d_u(i,j) | (i,j) \in (\text{Border}(G_u) \times \text{Border}(G_u)) \quad \text{and} \quad (\text{Neighbor}(i) \neq \text{Neighbor}(j)) \quad \text{and} \quad (i \neq j) \},
\]

which reflects the accumulated distance increase when a shortest path passes through the subgraph \( G_u \). The ratio of the maximum value to the minimum one in the traverse distance set is
called the traverse distance ratio of subgraph $G_u$, denoted by $R_T(G_u)$, which measures the difference in traverse distance of the subgraph.

**Graph hierarchy definitions**

**Definition 7.** Given a urban road network, the level-0 graph model is defined by $G^0 = (V^0, E^0, W^0)$, where

1. $V^0$ is the set of nodes, with each node $i \in V^0$ corresponding to a road intersection in the network.
2. $E^0 = \{(i, j) | (i, j) \in V^0 \land (i \neq j)\}$ is the set of edges, with each edge corresponding to a road.
3. For any node $i \in V^0$, its node weight $w^0(i) \in W^0$ corresponds to the average delay time of vehicles at $i$ during a certain computing period of time.

**Definition 8.** Given a partition $P = \{G_1, G_2, \ldots, G_p\}$ of level-0 graph $G^0$, the level-1 graph model is defined by $G^1 = (V^1, E^1, W^1)$, where

1. $V^1 = \bigcup_{u=1}^{p} \text{Border}(G_u)$.
2. $E^1 = (\bigcup_{u,v} \text{Cut}(G_u, G_v)) \cup (\bigcup_u \text{Connect}(G_u)), 1 \leq u, v \leq p$ and $u \neq v$.
3. For any edge $(i, j) \in E^1$, its edge weight

   $$w^1(i, j) = \begin{cases} 
   d_{G_u}(i, j) & (i, j) \in \text{Connect}(G_u) \\
   0 & \text{otherwise}
   \end{cases}.$$

**Definition 9.** Given a partition $P = \{G_1, G_2, \ldots, G_p\}$ of level-0 graph $G^0$, the level-2 graph model is defined by $G^2 = (V^2, E^2, W^2)$, where

1. Each node $i \in V^2$ corresponds to a subgraph.
2. Each edge $(i, j) \in E^2$ corresponds to the collection of cut edges between the subgraphs.
3. For any node $i \in V^2$, its node weight

   $$w^2(i) = \min \text{Traverse}(i).$$

**Fig 1** is an illustration of the model. The level-0 graph is the original urban road map that consists of all nodes and edges. A fragmentation of 8 partitions is applied on the level-0 graph, and all border nodes at this level are extracted as the nodes at the next higher level (level-1). The level-2 graph consists of 8 nodes, which is far less than the number of nodes at the original level (level-0).

**Graph partitioning algorithm**

In this section, we present an agglomerative graph partitioning algorithm for producing high balanced traverse distance partitions. The basic objective is to reduce the difference in traverse distance, that is to reduce the traverse distance ratio of each subgraph, while keeping the subgraph size to a setting range $[\delta_L, \delta_U]$.

The algorithm is composed of the following two steps:

Step 1: Initialization of subgraphs. In this step, the initial network (level-0 graph) is divided into a series of small subgraphs, where each border node has only one neighbor subgraph. This can be achieved by checking the nodes one by one: for each unassigned node $i$ (all nodes are unassigned with the subgraph number in the beginning), we proceed as follows...
1. Judge whether \( i \) has more than one adjacent node. If so, assign a new subgraph number \( Num \) to node \( i \), and maintain a set \( Sub \) for the nodes assigned to this subgraph in each iteration. \( Sub = \{ i \} \) in the beginning, then we repeat adding the unassigned adjacent nodes of the node in \( Sub \), that is

\[
Sub = \{ \text{unassigned adjacent nodes of } Sub \},
\]

and assign the subgraph number \( Num \) to the nodes in \( Sub \). This process is repeated until the subgraph size of \( Num \) is greater than \( \delta_i \). If \( \delta_i \) cannot be achieved after several iterations, move all nodes of \( Num \) to a neighbor subgraph and reassign the subgraph number.

2. Delete the first element in \( Sub \), i.e. node \( j \), judge whether \( j \) has an adjacent node \( x \) whose subgraph number is not \( Num \). If not, go to 3); Otherwise, add all unassigned adjacent nodes of \( j \) and \( x \) to the end of the set \( Sub \), and assign them the subgraph number \( Num \), and go to 4).

3. For any unassigned node \( x \), judge whether \( x \) has an adjacent node whose subgraph number is not \( Num \), add the node \( x \) to the end of the set \( Sub \) and assign it the subgraph number \( Num \) only if the condition holds, and then go to 4).

4. Exit if the set \( Sub \) becomes null; Otherwise, go to 2) and continue.

In the end, we move the unassigned degree one node (with only one adjacent node) to its neighbor subgraph and assign them with the subgraph number. At this point, all nodes are assigned to a subgraph, and each border node has only one neighbor subgraph.

Step 2: Subgraph agglomeration. In this step, an agglomeration process is performed on the decomposed small subgraphs produced in Step 1, with the objective of reducing the traverse distance ratio and regulating the subgraph size to the setting range. This can be achieved by greedily merging two neighboring subgraphs which yields the smallest traverse distance ratio, as long as the combined subgraph size is below \( \delta_U \).
1. Traverse distance ratio computation for each subgraph. Based on the decomposed subgraphs in Step 1, we can easily construct the level-1 graph $G^1$ by extracting the border nodes, the cut edges, and adding the connect edges between the border nodes of each subgraph. Then, a local shortest path tree is constructed from each node of $G^1$: starting with the border node as the root, a Dijkstra search on $G^1$ is stopped as soon as the current distance is already greater than the maximal weight of the connect edges incident to the root. Then we get the traverse distance set for each subgraph, and the traverse distance ratio can be computed by dividing the maximum value by the minimum one.

2. Heuristic agglomeration process. For each subgraph $G_u$, we evaluate the traverse distance ratio decrease that would happen by merging a neighboring subgraph to $G_u$. Combine $G_u$ with the subgraph which yields the maximum ratio decrease, but only if the combined subgraph size is below $\delta_U$. This process is repeated until all subgraphs are within the size of $[\delta_L, \delta_U]$ and no further improvements can be achieved. Note that the merging of subgraphs may simultaneously affect the traverse distance ratio of a neighboring subgraph, thus we need to update the ratio for such potentially affected subgraphs.

**Hierarchical path computation algorithm**

Based on the preceding graph model, we develop a hierarchical path computation algorithm using a region pruning strategy (HiARP) for accurate and fast path computations. We mainly discuss the case that the source and destination nodes are far away from each other; for the case that the source and destination are within the same subgraph or in adjacent subgraphs, we turn to compute the shortest path directly using the level-0 graph.

**HiARP algorithm**

The HiARP algorithm generally follows the Dijkstra search, where the search first starts on the level-2 graph for an estimation of the shortest path length between source node $s$ and destination node $d$, then the search switches to the level-0 graph, starting from the source node $s$, and once the border nodes are reached, the search will switch to the next higher level (level-1). A close upper bound on the shortest path length between $s$ and $d$ is maintained and tightened repeatedly during the search. Whenever the search on level-1 reaches a subgraph, a lower estimate will be calculated and the whole subgraph region will be pruned if this lower estimate exceeds the upper bound.

The algorithm is composed of the following steps:

**Step 1:** Search on $G^2$. Identify the source subgraph $G_s$ which contains the source node $s$ and the destination subgraph $G_d$ which contains the destination node $d$. Initialize a distance value $d_2(G_u, G_d)$ for every subgraph $G_u$, set it to zero for the destination subgraph $G_d$ and infinity for all the other subgraphs. Then start search from $G_d$ on the level-2 graph $G^2$ following a standard Dijkstra search, until all subgraphs are settled. Suppose that the shortest path between the destination subgraph $G_d$ and the source subgraph $G_s$ is

$$ SP_2(G_s, G_d) = \langle G_s(G_u) \rightarrow G_1 \rightarrow \cdots \rightarrow G_k(G_s) \rangle. \quad (5) $$

To compute an upper bound on the shortest path length from subgraph $G_u$ ($G_u \in SP_2(G_d, G_s)$) to the destination $d$, we utilizes the level-1 graph, and extracts the nodes and edges within the subgraph area of $SP_2(G_d, G_s)$ and extracts the cut edges between these subgraphs. Add an edge from $d$ to every border node $j_0$ ($j_0 \in \text{Border}(G_d)$), where the weight of the edge is assigned with the maximum weight of the connect edges incident to $j_0$, and the same edge adding
process from the border node \( i_k \) \( (i_k \in \text{Border}(G_j)) \) to the source \( s \). Then the shortest path from \( d \) to \( s \) within this constructed area can be easily obtained, as shown in Fig 2, where the shortest one is denoted by red lines. Let \( \tilde{d}(i, d) \) denote the path length from \( d \) to any node \( i \) on the shortest path. Based on this practical route, we get an upper bound on the shortest path length from every subgraph \( G_u \in \text{SP}_2(G_d, G_i) \) to \( d \), and from the source \( s \) to \( d \), that is:

\[
Upper_{s-u-d} = \tilde{d}(j, d), u = 0, 1, \cdots, k
\]

\[
Upper_{s-d} = \tilde{d}(s, d)
\]

Step 2: Search on \((G^0, G^1)\). Initialize a distance value \( d(s, i) \) for every node \( i \in V^0 \), set it to \( w^0(s) \) for the source node \( s \) and infinity for all the other nodes. Record the predecessor node \( P(i) \), which precedes \( i \) in an optimal shortest path from the source. Set \( P(s) = s \) for the source \( s \) and null for the other nodes. Then mark the source \( s \) as the current node and start the search.

1. Judge whether the current node \( i \) \( (i = s \) in the beginning\) is in the level-0 graph \( G^0 \). If not, go to 2); Otherwise, relax the adjacent node \( j \) of \( i \) on \( G^0 \), which amounts to replace \( d(s, j) \) with a new value \( (d(s, i) + w^0(j)) \), but only if this value is smaller. Overwrite the predecessor of \( j \) to \( P(j) = i \) if the distance to \( j \) is updated, and then go to 4).

2. Judge whether the current node \( i \) and the predecessor \( P(i) \) are in the same subgraph. If not, go to 3); Otherwise, compute the lower bound on the shortest path length from \( s \) to \( d \) via subgraph \( \text{Neighbor}(i) \), that is

\[
Lower_{s-i-d} = d(s, i) + d_j(\text{Neighbor}(i), G_d)
\]

The region pruning condition will act only if this lower estimate exceeds the current upper bound \( Upper_{s-i-d} \). If the condition satisfies, mark the subgraph \( \text{Neighbor}(i) \) as pruned and go to 4), and since then all searches growing into that subgraph region will be pruned; Otherwise, relax the cut edge \((i, j) \in E^1\), which amounts to replace the distance to \( j \) by \((d(s, i) + w^0(j)) \) if this yields a smaller value. Update the predecessor of \( j \), and then go to 4).

3. Tighten the upper bound \( Upper_{s-i-d} \) if \( i \) is a border node of subgraph \( G_u \in \text{SP}_2(G_d, G_i) \) \((1 \leq u \leq k)\):

\[
Upper_{s-i-d} = \min\{Upper_{s-u-d}, (d(s, P(i)) + w^1(i, j) + Upper_{u-i-d})\}
\]

Relax the connect edges incident to \( i \): for each endpoint \( j \), replace the distance to \( j \) by \((d(s, i) + w^1(i, j) - w^0(i)) \) if this yields a smaller value. Update the predecessor of \( j \), and go to 4).

4. Extract the node \( i \) with the minimum \( d(s, i) \) value from all the unmarked nodes. Exit if the node \( i = d \); Otherwise, mark the node \( i \) as current, and go to 1) and continue.
At the end of the algorithm, we get the shortest path length \( d(s, d) \) from \( s \) to \( d \), and the path can be concatenated through the record of predecessor nodes. The HiARP algorithm can be used efficiently to address the path computation problem for very large node-weighted graphs. The hierarchical strategy decomposes the complex search problem into several light searches on the graph hierarchies, and the region pruning strategy greatly reduces the search space by eliminating unnecessary subgraph regions, moreover instead of relaxing all adjacent nodes, the relaxation operation in Step 2 is also simplified, which makes HiARP much faster than classical hierarchical path computation algorithms, while the path computed by HiARP is optimal.

### Optimality of HiARP

We give a theoretical proof for the optimality of the HiARP algorithm from the following two aspects. First, it is natural that the upper bound will never underestimate the optimal shortest path length, and the lower bound will never overestimate that. Thus, for any pruned subgraph region \( \text{Neighbor}(i) \), the shortest path will certainly bypass \( i \) due to the fact that \( \text{Lower}_{s,d} > \text{Upper}_{s,d} > d(s,d) \), hence there is no need to relax the edge incident to \( i \), which can never appear on the shortest path; as the region pruning condition holds thereafter, all search branches growing into the subgraph region later will be pruned, and this will not affect the optimality of the algorithm. Second, the relaxation operation in Step 2 will not affect the nature of optimality in a Dijkstra’s search. In Step 2–1), we perform relaxation for all nodes adjacent to the current node on \( G^0 \), which is equivalent to a standard Dijkstra’s search. Step 2–2) and 3) denote a hierarchical search on the level-1 graph \( G^1 \), where in 3) we also relax all edges incident to the current node; while in 2) we only perform relaxation on the cut edge incident to the current node \( i \). There is no need to relax the connect edges \( (i,j) \in E^1 \) since \( i \) and \( P(i) \) are located in the same subgraph, thus \( w^1(P(i), j) \) already gives the optimal path length for the connect edges \( (P(i), j) \in E^1 \), and any relaxation towards connect edges just equals to a weight modification inside the subgraph, which has already been done by the relaxation operation at \( P(i) \). Therefore, we have proven the optimality of the HiARP algorithm. □

### Experimental evaluation

To verify the validity of our approach, we consider the real urban road network of New York City (freely available data from [22]). The network is transformed to a graph with 366923 nodes and 1557956 edges, where nodes and edges correspond to the roads and the intersecting points of roads respectively, and the travel time data is used to simulate the average delay time of vehicles at nodes. We present numerical evaluations of HiARP algorithm using different graph partitions, compared to the well-known HIPLA algorithm [16], and hierarchical Dijkstra algorithm in order to analyze the main influencing factors on the algorithm performance. Also, we study the performance of various algorithms in terms of computational time and accuracy in order to verify the effectiveness of the proposed heuristics to real-time routing applications.

### Settings

Four partition schemes are employed in our testing, which are generated at different stages of our graph partitioning algorithm, as shown in Table 1. The following parameters are counted:

- \( p \): the number of subgraphs in a graph partition
\[ \bar{R}_T = \frac{1}{p} \sum_{i=1}^{p} \frac{\text{max Traverse}(G_i)}{\text{min Traverse}(G_i)} \]

- \( R_T \): the average traverse distance ratio of a graph partition, defined as
- \( R_N \): the ratio of maximum number of nodes in the subgraph to the minimum one (which measures the difference in subgraph size)
- \(|V^1|\): the number of nodes in the level-1 graph
- \(|E^1|\): the number of edges in the level-1 graph

Generally, the average traverse distance ratio \( \bar{R}_T \) follows a downward trend with the agglomeration process of subgraphs, though it may fluctuate slightly. Here we select the partition scheme with a tentative rising \( \bar{R}_T \) value to facilitate the performance analysis and comparison of the query algorithm. As we can see from Table 1, the level-1 graph contains fewer nodes and edges as the subgraph agglomerates, the number of nodes drops, and the average degree increases slightly.

For each partition scheme, we conduct five tests each using the HiARP, the HPLA, and the hierarchical Dijkstra algorithm as the search rule. Each test is made to solve a set of 200 route requests using the same randomly generated source and destination nodes. It should be pointed out that during the execution of HPLA, we employ some pre-computed data to facilitate the path retrieval within subgraphs, which makes it much faster than the initial method. All the algorithms are implemented in Matlab 7.8.0 on an Intel Xeon X5482 Dual Core processor with 32GB of RAM and the system ran Microsoft Windows Vista.

**Performance analysis**

Fig 3 shows the average computational costs of HiARP, HPLA, and hierarchical Dijkstra algorithm (Hi-dijkstra) under Schemes 1 to 4, where the numbers presented are average values over 200 route requests. Here, similar trends are observed in the tests of each algorithm.

In Fig 3A, the execution time of HiARP algorithm reaches a minimum value in Scheme 3. It is noted that the efficiency of HiARP is affected simultaneously by the average traverse distance ratio \( \bar{R}_T \) and by the number of subgraphs \( p \). Naturally, the efficiency of a hierarchical routing algorithm will be enhanced with a decreasing number of subgraphs, as the corresponding level-1 graph contains fewer nodes and edges. Thus, the execution time of HiARP drops in Schemes 1 to 3 when the increase in \( \bar{R}_T \) has not become a leading factor. However, when \( \bar{R}_T \) exceeds a certain value, the efficiency of HiARP will be greatly weakened since the increase in \( \bar{R}_T \) leads to an even larger increase in the search space, compared to the size reduce of the level-1 graph. Hence, the execution time of HiARP increases in Scheme 4 though its number of subgraphs is the smallest.
In Fig 3B, we count another variable, the average number of subgraph regions that are pruned during the search, which depicts the efficiency of HiARP from another point of view. We observe that the number of regions pruned drops as \( p \) reduces in Schemes 1 to 4. Generally, the more regions pruned at an early search stage, the more efficient the HiARP would be. Thus, the number of regions pruned can only reflect the efficiency of HiARP to some extent, whereas the stage when the region is pruned is more important.

In Fig 3C, we find a similar fluctuation on the execution time of HIPLA. The efficiency of HIPLA is generally improved with the decrease of \( p \), as the corresponding abstraction graph [16] contains fewer nodes and edges, which leads to the downward trend in Schemes 1 to 2. Moreover, the balance in subgraph size can also affect the query efficiency, as HIPLA will search routes within the subgraph of path \( SP \) [16]. Hence, the execution time of HIPLA increases in Scheme 3 due to the rising ratio of \( R_N \), and then drops in Scheme 4 as a result of the decreasing \( R_N \) and \( p \).

In Fig 3D, the execution time of hierarchical Dijkstra algorithm decreases in all the five tests. It is reasonable that the efficiency of hierarchical Dijkstra algorithm depends particularly on the graph size, though the average degree of nodes may also affect it more or less. With a dropping number of subgraphs, the constructed level-1 graph contains fewer nodes and edges, thereby leading to a decreasing trend on the execution time over Schemes 1 to 4.

**Performance comparison**

Table 2 shows the average computational time and accuracy of HiARP, HIPLA, and hierarchical Dijkstra algorithm under Schemes 1–4, where the numbers presented are average values.
over all five test sets (an average of 1000 route requests to be precise). We observe that the execution time of HiARP is much less than that of hierarchical Dijkstra algorithm. As we can see the average execution time of HiARP on Scheme 1 is 0.72s, whereas the average execution time of hierarchical Dijkstra algorithm is 3.51s. Both HiARP and hierarchical Dijkstra algorithm compute the accurate optimal path for all route requests, which is consistent with our theoretical predictions in the Optimality of HiARP Section. We also notice that the HIPLA achieves the best time efficiency in all partition schemes, e.g., the average execution time of HIPLA on Scheme 1 is 0.20s; however, large errors are observed in its path computation results based on our testing graph. The average error observed in HIPLA is around 37.68%, with a maximum error of up to 242.5%, which is inapplicable to high-precision routing applications.

To further illustrate the performance improvements, we give a theoretical analysis on the computational complexity of HiARP, compared to the hierarchical Dijkstra algorithm and HIPLA. In a network with \(|V|\) nodes and \(p\) subgraphs, the average number of nodes in each subgraph is \(\lambda = |V|/p\). As the Dijkstra’s search runs in \(O(|V|\log|V|)\) time for sparse networks, the computational complexity of HIPLA is \(O(p\log p) + O(\lambda \log \lambda)\). Suppose that the level-1 graph contains \(|V|\) nodes and \(|E_1|\) edges, thereby the average number of border nodes in each subgraph is \(b = |V|/p\), and the path computational cost of hierarchical Dijkstra algorithm is of complexity \(O(|V|\log|V|)+O(\lambda \log \lambda)\). In our HiARP algorithm, the path computation is broken into two subprocesses, that is path search over the level-2 graph \(G^2\) with \(p\) nodes which incurs \(O(p\log p)\) time, and path search over \((G^0, G^1)\) which incurs \(O(\lambda \log \lambda)+O(kb \log (kb))\) time, where \(k\) is the average number of subgraph regions visited on the level-2 graph. Thus, the computational complexity of HiARP is \(O(p\log p) + O(\lambda \log \lambda) + O(kb \log (kb))\). For a well-partitioned graph which gives a good initial heuristics on the region pruning, \(k\) is quite small compared to \(p\), specifically when \(k < \max(p/b, \lambda/b)\), the complexity of HiARP is approximately \(O(p\log p) + O(\lambda \log \lambda)\), which equals to that of the HIPLA. In this case, the HiARP will search routes only within the subgraph area of \(SP_2(G_b, G_s)\), and all the other subgraphs will be pruned as a result of the region pruning condition. While on the contrary when the graph is not so well-partitioned, i.e., with large value of \(R_T\), few subgraph regions will be pruned during the execution of HiARP and the computational complexity will approximate to that of the hierarchical Dijkstra algorithm. This case can not happen anymore under our partition scheme, as the region pruning condition will always act since \(kb < |V|\). Therefore, the complexity of HiARP is much lower than that of the hierarchical Dijkstra algorithm and time efficiency is greatly improved.

Conclusions
In this paper, we have addressed the problem of efficient path computation on large urban road networks. We have developed a new graph partitioning method that can help significantly reduce the search space and thereby accelerate the path computation. By modelling the

Table 2. Computational times and accuracy comparisons of various algorithms on Schemes 1–4.

| Algorithm | Scheme 1 | Scheme 2 | Scheme 3 | Scheme 4 |
|-----------|----------|----------|----------|----------|
|           | t (s)    | E%       | Emax %   | t (s)    | E%       | Emax %   | t (s)    | E%       | Emax %   |
| Hi-dijkstra | 3.51     | 0        | 0        | 3.17     | 0        | 0        | 2.60     | 0        | 0        |
| HiARP     | 0.72     | 0        | 0        | 0.60     | 0        | 0        | 0.41     | 0        | 0        |
| HIPLA     | 0.20     | 38.16    | 154.57   | 0.19     | 40.13    | 242.50   | 0.22     | 36.09    | 164.36   |
|           |          |          |          |          |          |          |          |          |          |

\(t\) represents the average computational time (in seconds); \(E\) and \(E_{\text{max}}\) represent the average and maximum computational errors.

https://doi.org/10.1371/journal.pone.0192274.t002
road network in a three-level graph structure, we have decomposed the complex search problem into several light searches on the graph hierarchies. We have then proposed a new hierarchical path computation algorithm HiARP using a region pruning strategy, and we have formally proven that the shortest path computed by HiARP is optimal. Our HiARP algorithm is computationally very efficient and is readily applicable for path computation problems for very large node-weighted graphs. We have instantiated our approach with real-world data to study the algorithm performance under different graph partition schemes, compared with two famous algorithms viz., HIPLA and hierarchical Dijkstra algorithm, both theoretically and experimentally. The experimental evaluation has confirmed the effectiveness of the proposed graph partitioning and path computation algorithms to generate optimal fast paths so as to facilitate real-time routing applications.

The hierarchical model framework proposed in this paper including the graph partitioning and the hierarchical path computation algorithm provides a general approach for accurate optimal path computation on large node-weighted graphs, and this can be easily extended to edge-weighted graphs. Thus, as part of future research, it would be beneficial to see the application of the proposed approach to other types of networks, e.g., social networks, communication networks. Also, it is worth quantifying the influences of the number of subgraphs and the average traverse distance ratio on the query times, in order to determine the optimum value regions. And more complicated traffic conditions can be added to the model, and we will further study the algorithm performance on such networks.

**Supporting information**

S1 Code. Source code of the HiARP algorithm. (ZIP)

**Acknowledgments**

The authors would like to thank the Dr Qingjie Kong and Guilin Zhang for their time and advice in the preparation of this manuscript.

**Author Contributions**

Conceptualization: Qing Song, Xiaolei Li.

Data curation: Qing Song, Meng Li.

Formal analysis: Qing Song, Xiaolei Li.

Funding acquisition: Qing Song, Meng Li.

Investigation: Xiaolei Li.

Methodology: Qing Song, Meng Li, Xiaolei Li.

Supervision: Xiaolei Li.

Validation: Qing Song, Meng Li.

Writing – original draft: Qing Song.

**References**

1. Bast H, Delling D, Goldberg AV, Müller-Hannemann M, Pajor T, Sanders P, et al. Route planning in transportation networks. Algorithm Engineering: Selected Results and Surveys. 2016; 19–80.

2. Sommer C. Shortest-path queries in static networks. ACM Computing Surveys. 2014; 46(4): Article 45.
3. Yang J, Chen Y. Fast computing betweenness centrality with virtual nodes on large sparse networks. PLoS ONE. 2011; 6(7): e22557. https://doi.org/10.1371/journal.pone.0022557 PMID: 21818337

4. Gøni J, Avena-Koenigsberger A, Velez de Mendizabal N, van den Heuvel MP, Betzel RF, Sporns O. Exploring the morphospace of communication efficiency in complex networks. PLoS ONE. 2013; 8(3): e58070. https://doi.org/10.1371/journal.pone.0058070 PMID: 23505455

5. Zhu L, Zhang Y, Su F, Chen L, Huang T, Cai Y. A shortest-path-based method for the analysis and prediction of fruit-related genes in arabidopsis thaliana. PLoS ONE. 2016; 11(7): e0159519. https://doi.org/10.1371/journal.pone.0159519 PMID: 27434024

6. Dijkstra EW. A note on two problems in connexion with graphs. Numerical Mathematics. 1959; 1: 269–271.

7. Pohl I. Bi-directional search. Machine Intelligence. 1971; 6: 127–140.

8. Hart EP, Nilsson NJ, Raphael B. A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics. 1968; SSC-4(2): 100–107.

9. Lysgaard J. A two-phase shortest path algorithm for networks with node coordinates. European Journal of Operational Research. 1995; 87(2): 368–374.

10. Karimi HA. Real-time optimal route computation: A heuristic approach. ITS Journal. 1996; 3(2): 111–127.

11. Cherkassky BV, Goldberg AV, Radzik T. Shortest path algorithms: Theory and experimental evaluation. Mathematical Programming. 1996; 73: 129–174.

12. Wagner D, Willhalm T. Geometric containers for efficient shortest-path computation. ACM Journal of Experimental Algorithmics. 2005; 10(1.3): 1–30.

13. Möhring R, Schilling H, Schütz B, Wagner D, Willhalm T. Partitioning graphs to speed up Dijkstra’s algorithm. ACM Journal of Experimental Algorithmics. 2006; 11(2.8): 1–29.

14. Maue J, Sanders P, Matijevic D. Goal directed shortest path queries using precomputed cluster distances. ACM Journal of Experimental Algorithmics. 2009; 14(3.2): 1–27.

15. Jung S, Pramanik S. An efficient path computation model for hierarchically structured topographical road maps. IEEE Transactions on Knowledge and Data Engineering. 2002; 14(5): 1029–1046.

16. Rajagopalan R, Mehrotra KG, Mohan CK, Warshney PK. Hierarchical path computation approach for large graphs. IEEE Transactions on Aerospace and Electronic Systems. 2008; 44(2): 427–440.

17. Chondrogiannis T, Gamper J. Exploring graph partitioning for shortest path queries on road networks. Proceedings of the 26 GI-Workshop on Foundations of Databases. 2014; 1–6.

18. Liu B. Route finding by using knowledge about the road network. IEEE Transactions on Systems, Man, and Cybernetics, Part A. 1997; 27(4): 436–448.

19. Jagadeesh GR, Srikanthan T, Quek KH. Heuristic techniques for accelerating hierarchical routing on road networks. IEEE Transactions on Intelligent Transportation Systems. 2002; 3(4): 301–309.

20. Fu L, Sun D, Rilett LR. Heuristic shortest path algorithms for transportation applications: State of the art. Computers and Operations Research. 2006; 33: 3324–3343.

21. Chou YL, Romeijn HE, Smith RL. Approximating shortest paths in large-scale networks with an application to intelligent transportation systems. INFORMS Journal on Computing. 1998; 10(2): 79–163.

22. Nineth DIMACS implementation challenge-shortest paths. 2006; Available from: http://www.dis.uniroma1.it/~challenge9/download.shtml