Finding Distance-Preserving Subgraphs in Large Road Networks

Da Yan[#1], James Cheng[*2], Wilfred Ng[#3], Steven Liu[#4]

[#] *Department of Computer Science and Engineering, Hong Kong University of Science and Technology*

{[1] yanda, [3] wilfred, [4] tm_lksac}@cse.ust.hk

[*] *Department of Computer Science and Engineering, The Chinese University of Hong Kong*

[2] jcheng@cse.cuhk.edu.hk

***Abstract*—Given two sets of points, S and T, in a road network, G, a *distance-preserving subgraph* (*DPS*) query returns a subgraph of G that preserves the shortest path from any point in S to any point in T. DPS queries are important in many real world applications, such as route recommendation systems, logistics planning, and all kinds of shortest-path-related applications that run on resource-limited mobile devices. In this paper, we study efficient algorithms for processing DPS queries in large road networks. Four algorithms are proposed with different tradeoffs in terms of DPS quality and query processing time, and the best one is a graph-partitioning based index, called *RoadPart*, that finds a high quality DPS with short response time. Extensive experiments on large road networks demonstrate the merits of our algorithms, and verify the efficiency of RoadPart for finding a high-quality DPS.**

I. Introduction

Given two point sets S and T in a road network G, a **distance-preserving subgraph** (**DPS**) query returns a subgraph of G that preserves the shortest path distance between any two points $s \in S$ and $t \in T$. The query set S (or T) may be specified by a region (e.g., a district or a city), in which case S (or T) contains any point in G that is within the region.

In this paper, we assume that a server maintains a large road network, and processes DPS queries posed by different applications. The applications may then perform different tasks locally on the DPS obtained from the server. Here we describe two possible real world applications of DPS queries:

Example 1: Consider a French logistics company providing services between Paris and three other European cities: Munich, Rome, and Madrid. Given the European road network, the company can pose three DPS queries with S being the set of involved locations in Paris, and T being the set of involved locations in Munich, Rome, and Madrid, respectively. The query answers are three small subgraphs, which are then merged as a small graph. The company can then arrange the delivery routes efficiently using the graph. ■

Example 2: Consider the development of a route search engine for people who travel in Southern California. Given the USA road network, the search engine may pose a DPS query with $S = T$ being the set of travel spots in Southern California. The obtained subgraph can then be used by the search engine to process route queries posed by travelers. ■

Using a DPS in an application has many benefits compared with using the original large road network, including smaller space requirement and faster query processing.

Space reduction. Disk-resident graph traversal is costly since it requires many random I/O operations. As a result, many studies advocate to hold the graph in memory [14], [17]. However, an entire road network is often too large to fit in the limited memory space of a mobile device.

A recent study [6] suggests to have mobile devices answer navigational queries locally, as a way of addressing the scalability limitations of servers that may need to answer heavy workloads of location queries online. In their model, the server partitions the road network into regions (or graph fragments), and repeatedly broadcasts the regions on the air. Given a source and a destination, the clients only receive the necessary regions, and store the resulting subgraph for query processing.

Note that [6] requires a mobile device to receive the relevant regions to process each point to point shortest path (PPSP) query individually. On the other hand, if a mobile device downloads a DPS, it can be used to compute any shortest path between points of interest locally.

Query efficiency. Processing a PPSP query often accesses a large portion of vertices that are irrelevant (i.e., they are not on the shortest path), especially when the source and the destination are far apart. Computing the shortest paths in a DPS avoids visiting a large number of irrelevant vertices in the original road network.

Most state-of-the-art shortest path indices on road networks rely on pre-computing all-pair shortest paths [7], [8], [9], [10], which is not practical for large road networks. If the region of interest is constrained, one can issue a DPS query and build the indices on the DPS returned by the query. Since the subgraph is distance-preserving, the shortest paths between points of interest are correctly obtained from the indices.

In addition to answering shortest path queries, the DPS can also be used to efficiently process many other queries whose definitions are based on the network distance, such as optimal location queries [2], aggregate nearest neighbor queries [3], and optimal meeting point queries [4].

Contribution. In this paper, we propose the DPS query. We develop efficient algorithms to answer the DPS queries. We first propose two basic algorithms, one quality-centric algorithm that finds the smallest DPS and one efficiency-centric algorithm that finds a loose DPS in one pass over the graph. Then, we develop a graph-partitioning based index, called **RoadPart**, that considers both query answer quality

and query efficiency. Finally, we propose a convex hull based method to further improve the quality of the DPS obtained by RoadPart. Our experiments show that our algorithms scale to road networks with tens of millions of vertices, and RoadPart is able to compute a high quality DPS efficiently.

Organization. The rest of the paper is organized as follows. We formally define the DPS query in Section II. Section III introduces two basic algorithms. Section IV presents the RoadPart index for planar road networks, while Section V extends RoadPart to handle general road networks. Section VI discusses the convex hull method. We report the experimental results in Section VII, review the related work in Section VIII, and conclude the paper in Section IX.

II. Notations and Problem Definition

In this paper, we model a road network as an undirected, weighted and connected graph, $G = (V, E)$, where each vertex $v \in V$ is associated with Cartesian coordinates $(v.x, v.y)$, and each edge $(u, v) \in E$ has a weight equal to the physical length of (u, v). In addition, G has bounded vertex degree, i.e., the maximum vertex degree in G is a small constant, and $|E| = O(|V|)$. Many existing works also treat G as a planar graph. However, we do not make this assumption, since a real road network is only a near-planar graph with a small portion of crossover edges (called **bridges**) modeling flyovers and tunnels.

We denote the length of an edge (u, v) by $|uv|$, and the Euclidean distance between points u and v by $\|uv\|$. We also denote the shortest path between vertices u and v by $sp(u, v)$, and the length of $sp(u, v)$ by $dist(u, v)$.

Problem definition. Given a road network, $G = (V, E)$, and two query point sets, S and T, where the points are locations in G, find a subset of vertices $V' \subseteq V$ such that for any two points, $s \in S$ and $t \in T$, $sp(s, t)$ exists in the subgraph G' of G induced by V'.

The subgraph G' is called a **distance-preserving subgraph** (**DPS**) of G for (S, T). We call such a query an ***(S,T)*-DPS query**. In the special case when $S = T = Q$, we call the query a ***Q*-DPS query**.

From now on, we assume $S, T \subseteq V$. If a query point q is on an edge (u, v), we only need to include both u and v into the query set instead of q.

To handle some spatial operations efficiently in our query processing algorithms, we construct two R-trees, $Rtree(V)$ and $Rtree(E)$, bulkloaded over the vertex set V and edge set E, respectively [12]. Note that each vertex in a road network is a 2D point with coordinates and each edge is a line segment. The R-trees are built once for all as a pre-processing step, and they can be used in the processing of any DPS query.

The DPS G' is not unique, and our objective is to find a tight DPS efficiently, which we will discuss in the following sections.

III. Baseline Algorithms for Finding a DPS

There are two main performance metrics for answering a DPS query: (1) *the quality of query answer* (i.e., the DPS), and (2) *the efficiency of query processing*. Based on these two performance metrics, we develop two baseline algorithms for answering DPS queries, one focusing on attaining the smallest DPS, while the other focusing on finding a DPS in short processing time.

A. Quality Centric Baseline

The smallest DPS for query (S, T) is the DPS $G' = (V', E')$ such that V' contains only the vertices on $sp(s, t)$ for all $s \in S$ and $t \in T$. We propose a baseline algorithm, named **BL-Quality (BL-Q)**, to compute this DPS, which requires $\min\{|S|, |T|\}$ rounds of *single-source shortest path* (**SSSP**) computation.

Without loss of generality, assume $|S| \leq |T|$. Then, we need to run the SSSP algorithm (such as Dijkstra's algorithm) for all $s \in S$. To avoid accessing the entire graph, we can terminate the SSSP computation as soon as the shortest paths from s to all vertices in T are computed.

BL-Q takes $O(\min\{|S|, |T|\} \cdot |V| \log |V|)$ time, since for road networks, each SSSP computation takes only $O(|V| \log |V|)$ time. The algorithm finds the smallest DPS; however, it is very slow when $|S|$ and $|T|$ are large.

Vertex collection. After running SSSP computation for each $s \in S$, we need to include the vertices on $sp(s, t)$ into V' for all $t \in T$. Here we show that this operation can be carried out in $O(|E|) = O(|V|)$ time, and thus does not increase the time complexity of BL-Q: We maintain a set C containing all the vertices that are already added to V'. For each vertex $t \in T$, we add the vertices on $sp(s, t)$ to V' starting with $v = t$, and stop either when a vertex $v \in C$ is reached (since the vertices on $sp(s, v)$ are already added to V'), or when s is reached. The $O(|E|)$ time complexity follows from the fact that each edge in G is visited at most once.

B. Query Efficiency Centric Baseline

To achieve high efficiency in query processing, we want to minimize the number of rounds of the SSSP computation. We present another baseline algorithm, called **BL-Efficiency (BL-E)**, that uses only one round of SSSP computation.

BL-E first locates a vertex v_c in the middle of Q (we set $Q = S \cup T$ for query (S, T)). Then, we run SSSP from v_c until $sp(v_c, q)$ is computed for all $q \in Q$. Let r be the length of the longest $sp(v_c, q)$. We continue to run the SSSP from v_c to obtain all the other shortest paths from v_c with length no greater than $2r$. Finally, we include into V' the vertices on all shortest paths from v_c with length no greater than $2r$.

We find v_c by first computing the *minimum bounding rectangle* (**MBR**) of Q, based on the Cartesian coordinates of the vertices in Q. Let us denote the center of the MBR by p_c. We then find v_c as the vertex nearest to p_c, through a nearest neighbor query over the R-tree $Rtree(V)$.

We now prove the correctness of BL-E.

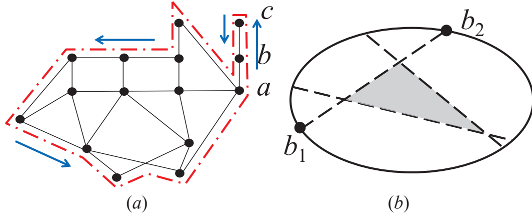


Fig. 1. A contour of a road network and the cuts

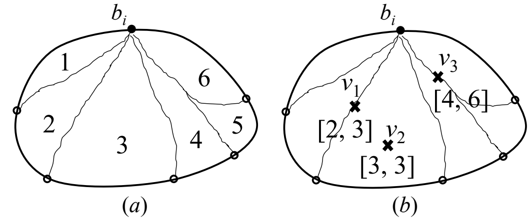


Fig. 2. Vertex labeling in one dimension, w.r.t. b_i

Lemma 1: $\forall s, t \in Q$, $dist(s,t) \leq 2r$.

Proof: $dist(s,t) \leq dist(s,v_c) + dist(v_c,t) \leq 2r$. ■

Theorem 1: $\forall v \in V$, if $dist(v_c,v) > 2r$, then v is not on $sp(s,t)$ for any $s,t \in Q$.

Proof: Since $dist(v_c,v) \leq dist(v_c,s) + dist(s,v)$, we have $dist(s,v) \geq dist(v_c,v) - dist(v_c,s) > 2r - r = r$. Similarly, we have $dist(t,v) > r$. Suppose on the contrary that $\exists s,t \in Q$, such that v is on $sp(s,t)$, then $dist(s,t) = dist(s,v) + dist(v,t) > 2r$, which contradicts Lemma 1. ■

According to Theorem 1, it is not necessary to add v to the DPS if $dist(v_c,v) > 2r$. However, this DPS is at least $((2r)^2/r^2) = 4$ times as large as the smallest DPS; thus, it is a low-quality query answer. On the other hand, the algorithm is very efficient since it only takes $O(|V|\log|V|)$ time.

IV. An Index for Answering DPS Queries

In Section III, we introduced the two main performance metrics for answering DPS queries. We also presented two baseline algorithms and identified their weaknesses. In this section, we propose an index to answer DPS queries efficiently with high answer quality.

We name our index as **RoadPart**, since our approach is based on road network partitioning. In this section, we first present our solution for planar road networks. Then in Section V, we extend our solution to handle non-planar road networks.

A. An Overview of RoadPart

We first present an overview of RoadPart, which consists of two phases: (1) **offline indexing phase**, which partitions the input road network into small regions and assigns vertices with region IDs; and (2) **online querying phase**, which answers any DPS query using the region IDs of the vertices in a query.

In the indexing phase, we first compute a contour of the road network $G = (V, E)$ and partition G by shortest paths between vertices on the contour (such shortest paths are called *cuts*). A **contour** of G is an ordered sequence of vertices, $C = \langle v_1, \ldots, v_k, v_{k+1} = v_1\rangle$, where each $v_i \in V$, such that all vertices in G are contained in the polygon formed by linking each v_i to v_{i+1}, for $1 \leq i \leq k$. A **cut** is a shortest path (in G) connecting two vertices in C. For example, the dashed outline shown in Figure 1(a) represents the contour of the road network, and the shortest path $sp(b_1, b_2)$ shown in Figure 1(b) is a cut.

We select a set of **border vertices** $B = \{b_1, \ldots, b_\ell\} \subseteq C$ evenly on the contour C. Then, we use the cuts $sp(b_i, b_j)$, where $b_i, b_j \in B$ to partition G. As shown in Figure 2(a), the cuts from b_i to the other border vertices partition the whole road network into **zones**. We assign each zone with a unique label. Given the border vertex b_i, we then assign a label $[l(v), h(v)]$ to each vertex $v \in V$, indicating that v belongs to Zones $l(v)$-$h(v)$ derived by the cuts from b_i. For example, in Figure 2(b), v_2 is assigned a label $[3,3]$ since v_2 is contained in Zone 3, while v_3 is assigned a label $[4,6]$ since v_3 is on the boundaries of Zones 4, 5 and 6.

For each border vertex $b_i \in B$, each vertex v is assigned a label determined by the zones that are derived by the cuts from b_i; thus, altogether we assign $|B|$ labels to each vertex. We associate each vertex $v \in V$ with a $|B|$-dimensional label vector $vec(v)$, such that its i-th dimension $vec(v)[i]$ is the label determined by b_i.

Note that zones may overlap with each other if they are partitioned by cuts from different border vertices. Moreover, a vertex may belong to different zones with respect to different border vertices. Thus, we define a **region** R as a set of vertices that have the same label vector, i.e., $\forall u, v \in R$, $vec(u) = vec(v)$.

The output of our partitioning algorithm is a set of regions, $\mathcal{R}$, such that each vertex $v \in V$ belongs to exactly one region $R \in \mathcal{R}$. Since all the vertices in a region R have the same label vector, we assign each region $R \in \mathcal{R}$ a unique ID and keep only the region ID, instead of $vec(v)$, with v. On the other hand, we keep the label vector with R, denoted by $vec(R)$, i.e., $vec(R) = vec(v)$ for all $v \in R$. In this way, we reduce the space cost for keeping the label vectors from $O(|B| \cdot |V|)$ to $O(|V| + |B| \cdot |\mathcal{R}|)$, which is a significant reduction since $|\mathcal{R}| \ll |V|$.

Let $R(v)$ be the region that a vertex v belongs to. Then, the label vector of v can be obtained by $vec(R(v))$. For simplicity, we simply use $vec(v)$ to mean $vec(R(v))$ in the remainder of this paper.

Finally, given a DPS query, Q or (S, T), we retrieve $vec(v)$ for all $v \in Q$ or $v \in (S \cup T)$, and then compute the DPS for the query from these label vectors.

B. Indexing by Graph Partitioning

We now discuss the details of the indexing algorithm, which consists of three main parts: **contour computation**, **border vertex selection**, and **vertex labeling**.

1) Contour Computation: A contour of a road network G is like a bounding polygon of G. Thus, a tighter bounding polygon captures the shape of the entire road network more

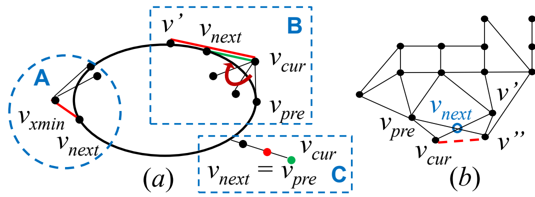


Fig. 3. Contour computation

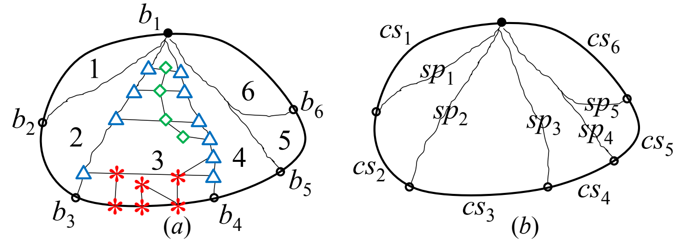


Fig. 4. Vertex labeling

accurately and hence gives a partitioning of higher quality. We compute a tight contour of G as follows.

Let v_{xmin} be the vertex in G with the minimum x-coordinate. Let v_{pre}, v_{cur} and v_{next} be the *previous*, *current* and *next* vertices processed, respectively.

We start with $v_{cur} = v_{xmin}$, and keep finding the next vertex v_{next} on the contour until v_{cur} becomes v_{xmin} again. When $v_{cur} = v_{xmin}$, we choose the edge, (v_{cur}, v_{next}), that is the most downward in the plane, as shown in Part A of Figure 3(a).

When $v_{cur} \neq v_{xmin}$, we pick the edge (v_{cur}, v_{next}) that maximizes the clockwise angle $\angle v_{pre}v_{cur}v_{next}$, where tie is broken by choosing the shortest edge, as shown in Part B of Figure 3(a). A special case is when v_{cur} is a dangling point as shown in Part C of Figure 3(a), in which case we set $v_{next} = v_{pre}$. This case finds a sub-sequence in the contour, such as $\langle a, b, c, b, a\rangle$ in Figure 1(a).

Extension to non-planar graphs. For a non-planar graph, which we will discuss in Section V, v_{next} may not be a vertex adjacent to v_{cur} due to the presence of bridges. For example, in Figure 3(b), the edge (v_{cur}, v') does not imply that the correct choice of v_{next} is v'. The correct v_{next} that gives a tight contour, however, is not a vertex of the graph, but the point '$\circ$' as shown in Figure 3(b).

We compute this v_{next} as follows. We first find the edge (v_{cur}, v') with maximum $\angle v_{pre}v_{cur}v'$. Then, we find all the edges that cross (v_{cur}, v'), among which we find the edge (u_{int}, v_{int}) whose intersection point v_{int} is closest to v_{cur}. We set $v_{next} = v_{int}$ if (u_{int}, v_{int}) exists, and set $v_{next} = v'$ otherwise. The edges crossing (v_{cur}, v') are found by an intersection query on the R-tree $Rtree(E)$.

Note that the new vertex v_{int} is only added temporarily to compute the correct contour; however, since v_{int} is not a vertex in G, it cannot be a border vertex for graph partitioning and is hence removed at the end of contour computation.

Complexity analysis. The contour contains approximately $O(\sqrt{|V|})$ vertices (as implied from the relationship between circumference and area). Let us use d to denote the maximum vertex degree of G, and use d' to denote the maximum number of edges checked in an intersection query on $Rtree(E)$. Note that d and d' are usually small constants in a road network, and thus, it takes $O((d+d' \log |E|)\sqrt{|V|}) = O(\sqrt{|V|} \log |V|)$ time to compute the contour.

2) Border Vertex Selection: The contour computation returns a vertex sequence, i.e., the contour $C = \langle v_1 = v_{xmin}, \ldots, v_k, v_{k+1} = v_{xmin}\rangle$, which constitutes a polygon. We define the circumference of the contour to be $L = \sum_{i=1}^{k} \|v_i v_{i+1}\|$, where we use Euclidean distance rather than edge length because the edge (v_i, v_{i+1}) may not exist. For example, in Figure 3(b), v_{cur} and v'' are two consecutive vertices of the contour but (v_{cur}, v'') is not an edge in G. The length of a subsequence $\langle v_i, v_{i+1}, \ldots, v_j\rangle$ is defined similarly.

We select ℓ border vertices, $B = \{b_1, \ldots, b_\ell\}$, evenly on the contour. We apply the *equi-length* method to divide the contour into disjoint subsequences such that each subsequence has length close to L/ℓ, and select the first vertex in each subsequence as a border vertex. We apply the equi-length method instead of the *equi-frequency* method[1] because road networks are distance-based.

3) Vertex Labeling: We now present how to compute the labels of all the vertices determined by a border vertex b. Consider b_1 in Figure 4(a), where the cuts, i.e., the shortest paths $sp(b_1, b_i)$ for $2 \leq i \leq 6$, divide the graph into 6 zones. We compute the cuts using the A* algorithm [13].

From now on, given a border vertex b, we use sp_i to denote the cut from b that divides Zone i and Zone $(i+1)$, for $1 \leq i < \ell$, which is illustrated in Figure 4(b).

The border vertices also divide the contour into disjoint vertex subsequences $\{cs_1, \ldots, cs_\ell\}$, where cs_i belongs to Zone i. For example, in Figure 4(a), cs_3 corresponds to the contour segment between b_3 and b_4.

Since a vertex v may belong to several zones, if v already has label $[l(v), h(v)]$, and it is newly found to belong to Zone i, we need to **insert** i into $[l(v), h(v)]$. The insertion operation is defined as follows:

- **Case 1:** no update is necessary if $i \in [l(v), h(v)]$.
- **Case 2:** if $i < l(v)$, update $l(v)$ to be i.
- **Case 3:** if $i > h(v)$, update $h(v)$ to be i.

Given a border vertex b, we assign the labels determined by b to all the vertices in the three steps given below:

Step 1: for $1 \leq i < \ell$, we process each vertex v on sp_i as follows: if v is not labeled, we assign label $[i, i+1]$ to v; otherwise, v is already labeled by another cut, and we insert i and $i+1$ to the label of v.

Step 2: for each vertex subsequence cs_i of the contour, we assign label $[i, i]$ to the unlabeled vertices in cs_i, and put them in a queue $\mathcal{Q}$. Then, we start an *in-zone BFS* from $\mathcal{Q}$, i.e., a bread-first search that only visits the vertices within Zone i, and assign label $[i, i]$ to each unlabeled vertex reached.

[1] Equi-frequency divides the contour into disjoint subsequences such that each subsequence contains k/ℓ vertices.

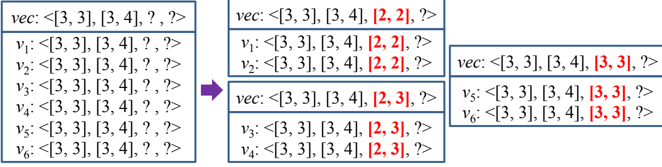


Fig. 5. Region splitting

In-zone BFS avoids visiting a vertex in another zone by stopping BFS propagation at labeled nodes, including those on sp_{i-1} and sp_i. Furthermore, it ignores bridges to ensure that label $[i, i]$ does not propagate across sp_{i-1} and sp_i through a bridge.

Steps 1 and 2 may not label all the vertices in Zone i. For example, in Zone 3 of Figure 4(a), Step 1 assigns label to the vertices (marked by '$\triangle$') on sp_2 and sp_3, while the in-zone BFS from cs_3 will only reach the vertices that are marked by '$*$' within Zone 3, leaving all the other vertices that are marked by '$\diamond$' unlabeled.

Step 3: Since Zone i is a polygon consisting of the edges of sp_{i-1}, cs_i and sp_i, we can determine whether a vertex v is in Zone i using the ray casting algorithm.

To ensure that all vertices are marked, for each unlabeled vertex v, we determine its zone using the ray casting algorithm (let it be Zone i), and assign label $[i, i]$ to each unlabeled vertex reached from v by the in-zone BFS. Note that computing the zone that a vertex falls in is more expensive than labeling it through in-zone BFS from other vertices. For example, in Zone 3 of Figure 4(a), when we find any '$\diamond$' vertex, we can assign the label $[3, 3]$ to all the other '$\diamond$' vertices reachable from it via in-zone BFS.

Vertex labeling, along with graph partitioning, is performed in ℓ rounds, where round i is for border vertex $b_i \in B$.

In round 1, we partition G into ℓ regions by $(\ell - 1)$ cuts, $sp(b_1, b_j)$ for $2 \leq j \leq \ell$, and assign $vec(v)[1]$ for each $v \in V$. Then, in round 2, we further partition G into smaller regions by $(\ell - 1)$ cuts, $sp(b_2, b_j)$ for $1 \leq j \leq \ell$ and $j \neq 2$, and assign $vec(v)[2]$ for each $v \in V$. This process repeats until we process all the ℓ border vertices.

In fact, instead of keeping $vec(v)$ for each $v \in V$, we only need to keep $vec(R)$ for each region R obtained in each round, since $vec(v) = vec(R)$ if v is in R. When R is partitioned into k smaller regions in a following round, we split $vec(R)$ into k label vectors by adding another dimension. For example, Figure 5 illustrates the splitting of a region in round 2 into three regions in round 3, where we add the 3-rd dimension to the label vectors to distinguish among the new regions created in round 3.

At the end of round ℓ, we obtain the region set $\mathcal{R}$, where each region R has a distinct $vec(R)$ with ℓ dimensions.

Complexity analysis. Recall that computing the contour takes $O(\sqrt{|V|}\log|V|)$ time. Since we have ℓ border vertices, computing all the cuts from the border vertices takes $O(\ell^2|V|\log|V|)$ time. For the vertex labeling, since most vertices are assigned label by in-zone BFS, it takes approximately $O(|V| + |E|) = O(|V|)$ time; and since we have ℓ rounds of vertex labeling, we need $O(\ell|V|)$ time in total. Therefore, the overall time complexity of graph partitioning is approximated by $O(\ell^2|V|\log|V|)$.

C. Query Processing by Finding Regions

Given an (S, T)-DPS query (similarly for a Q-DPS query), we use the region set $\mathcal{R}$ to construct a DPS $G' = (V', E')$ that preserves $dist(s, t)$ for any $s \in S$ and $t \in T$. Since we consider only planar road networks in this section, $sp(s, t)$ does not contain a bridge edge.

The DPS G' is computed by the following two steps:

1) Compute an ℓ-dimensional label vector, called a **window** W, from the query sets S and T, such that all points in $(S \cup T)$ are contained in the region represented by W.
2) Find all the regions in $\mathcal{R}$ that are contained in the region represented by W, and add their vertices to V'.

While all the vertices in a region $R \in \mathcal{R}$ have the same label vector, we define the region represented by W differently. Let us use the label vector W to directly refer to the region represented by W. Then, a vertex v is in W, if and only if for each dimension i (let $W[i] = [l_W, h_W]$), v belongs to at least one of Zones l_W–h_W determined by border vertex b_i. Thus, vertices in W can have different label vectors.

Before we discuss the details of query processing, we first give the following two label operations that are used in the subsequent discussion. Let $[l, h]$ and $[l', h']$ be two zone labels.

- **Label union**: $[l, h] \cup [l', h'] = [min(l, l'), max(h, h')]$.
- **Label intersection**: if $max(l, l') \leq min(h, h')$, $[l, h] \cap [l', h'] = [max(l, l'), min(h, h')]$; otherwise, $[l, h] \cap [l', h'] = \emptyset$.

Next, we present a pruning strategy employed in our query processing. A region $R \in \mathcal{R}$ can be pruned by checking $vec(R)$ against W as stated in Theorem 2 below.

We first give Lemma 2 which is used in the proof of Theorem 2.

Lemma 2: Given a cut sp that divides a road network into two sides, let s and t be two vertices that are both on one side of the cut, then there exists a shortest path $sp(s, t)$ that does not go across sp to the other side.

Proof: Consider the example in Figure 6(a) where s and t are on the right side of the cut sp_2. Suppose on the contrary that $sp(s, t)$ (i.e., the bold path) goes across sp_2 to the other side of the cut. Since in this section we consider G as a planar graph, $sp(s, t)$ and sp_2 must intersect at two points m_1 and m_2 as shown in Figure 6(a). Since sp_2 is a shortest path, the sub-path between m_1 and m_2 on sp_2 is also a shortest path, denoted by $sp(m_1, m_2)$. Therefore, the path composed of $sp(s, m_1)$, $sp(m_1, m_2)$ and $sp(m_2, t)$ is not longer than $sp(s, t)$, which gives another shortest path from s to t that does not go across sp_2 to the other side. ∎

Theorem 2: A region $R \in \mathcal{R}$ can be pruned if there exists a dimension i such that $vec(R)[i] \cap W[i] = \emptyset$.

Proof: We prove that if $vec(R)[i] \cap W[i] = \emptyset$, then $\forall s \in S$ and $t \in T$, $sp(s, t)$ does not contain any vertex in R.

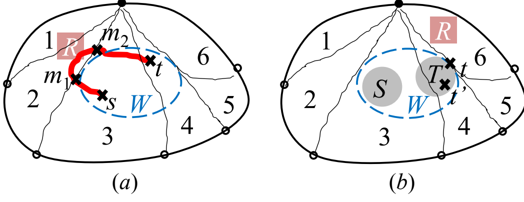


Fig. 6. Query processing by finding regions

Let $W[i] = [l_W, h_W]$ and $vec(R)[i] = [l_R, h_R]$, then W is bounded by cuts sp_{l_W-1} and sp_{h_W} with respect to b_i. For example, in Figure 6(a), $W[i] = [3, 4]$ is bounded by cuts sp_2 and sp_4.

Since $vec(R)[i] \cap W[i] = \emptyset$, we have $h_R < l_W$ or $h_W < l_R$. Without loss of generality, let us assume $h_R < l_W$. Since W covers all points in $S \cup T$, $\forall s \in S$ and $t \in T$, s and t are one side of sp_{l_W-1} and all vertices in R are on the other side. According to Lemma 2, $sp(s, t)$ does not contain any vertex in R. ■

To illustrate, Figure 6(b) shows the process of computing the i-th dimension of the window W: since all points in S and T are contained in Zones 3 and 4, we have $W[i] = [3, 4]$. We can prune a region R if $vec(R)[i] = [2, 2]$, since $sp(s, t)$ does not pass through Zone 2. However, we cannot prune R if $vec(R)[i] = [2, 3]$, since R is contained in Zone 3.

Theorem 2 implies that a region $R \in \mathcal{R}$ is not pruned only if $\forall i, vec(R)[i] \cap W[i] \neq \emptyset$, i.e., R is contained by W.

The above discussion shows that it is critical to compute a tight window. Given query Q (we set $Q = (S \cup T)$ for query (S, T)), let $R(Q) = \cup_{q \in Q} R(q)$ be the set of regions in $\mathcal{R}$ that contain some vertices in Q. A simple way to compute the window is given below:

$$W[i] = \cup_{R \in R(Q)} vec(R)[i], \forall i. \quad (1)$$

However, the window computed by Equation (1) can be quite loose. Consider the example shown in Figure 6(b), where there are two vertices $t, t' \in T$, with $vec(t)[i] = [4, 6]$ and $vec(t')[i] = [4, 4]$. Due to the presence of t, by Equation (1) we have $W[i] \supseteq [4, 6]$, i.e., $W[i]$ contains Zones 4, 5 and 6. As a result, the region R with $vec(R)[i] = [6, 6]$ cannot be pruned using Theorem 2 since $[4, 6] \cap [6, 6] \neq \emptyset$. However, $W[i] = [3, 4]$ is sufficient for the example in Figure 6, and R can be safely pruned since $[3, 4] \cap [6, 6] = \emptyset$. Although $vec(t)[i] = [4, 6]$, it is not necessary to include Zones 5 and 6 into W since t is contained in Zone 4. On the other hand, Zone 4 must be included in W because $vec(t')[i] = [4, 4]$.

We propose to obtain a tight window by the following steps:

1) **Window initialization:** For each dimension i, if there exists $R \in R(Q)$ such that $vec(R)[i] = [l, l]$, we set $W[i] = vec(R)[i]$; otherwise, we pick an arbitrary region $R \in R(Q)$ with $vec(R)[i] = [l, h]$ and set $W[i] = [l, l]$. Let $W[i] = [l_W, h_W]$. In either case, we initialize $l_W = h_W$ for each dimension i.
2) **Window expansion:** given the current window W and a region $R \in R(Q)$, let $vec(R)[i] = [l_R, h_R]$. We expand W for each $R \in R(Q)$ in turn as follows:

   Case 1: if $W[i] \cap vec(R)[i] \neq \emptyset$, $W[i]$ remains unchanged. For example, consider the case when $W[i] = [3, 4]$ and $vec(R)[i] = [4, 6]$.
   Case 2: if $l_W > h_R$, we set $W[i] = [h_R, h_W]$. For example, when $W[i] = [3, 4]$ and $vec(R)[i] = [1, 2]$, $W[i]$ is updated to [2, 4].
   Case 3: if $l_R > h_W$, we set $W[i] = [l_W, l_R]$. For example, when $W[i] = [3, 4]$ and $vec(R)[i] = [5, 6]$, $W[i]$ is updated to [3, 5].

The following theorem proves the correctness of query processing.

Theorem 3: Given a Q-DPS query (set $Q = (S \cup T)$ for an (S, T)-DPS query), let G' be the subgraph of G induced by the set of vertices that are in the regions contained by W. Then, G' is a DPS for Q when G is planar.

Proof: The correctness of this method follows from the fact that all regions in $R(Q)$ are contained in W. ■

Complexity analysis. It takes $O(|Q|) = O(|V|)$ time to compute $R(Q)$, $O(\ell|R(Q)|) = O(\ell|\mathcal{R}|)$ time to compute W, $O(\ell|\mathcal{R}|)$ time to find the regions contained by W, and $O(V)$ time to add the vertices in these regions into V'. Thus, query processing takes $O(\ell|\mathcal{R}| + |V|)$ time.

V. Handling Non-Planar Road Networks

In this section, we extend RoadPart to process DPS queries in non-planar road networks.

A. Finding Bridges

A real road network usually contains a small portion of **bridges**, i.e., edges that cross other edges (modeling flyovers, tunnels, etc.). We need to find all the bridges in the indexing phase of RoadPart, which are then used for answering DPS queries.

Given a non-planar road network $G = (V, E)$, the problem of finding all the bridges of G is actually a self-spatial-join of the edge set E, where the spatial predicate is edge intersection.

We use *indexed-nested-loop* join to find the bridges, where the index is the R-tree $Rtree(E)$. We check each edge $(u, v) \in E$, and if it is not yet marked as a bridge, we find its crossing edges by an intersection query on $Rtree(E)$; if there exists a crossing edge, we mark (u, v) and its crossing edges as bridges.

B. DPS Construction Using Bridges

Referring to Lemma 2 in Section IV-C again, we can see that as long as $sp(s, t)$ does not pass through a bridge (and thus m_1 and m_2 in its proof exist), $sp(s, t)$ does not need to go across cut sp. Thus, we have the following corollary of Lemma 2 in the context of a non-planar road network.

Corollary 1: Given a cut sp, let s and t be two vertices on one side of sp. If $sp(s, t)$ does not pass through a bridge, it does not go across sp to the other side of sp.

Corollary 2: Given a cut sp, let s and t be two vertices inside W. If $sp(s, t)$ does not pass through a bridge, then $sp(s, t)$ is inside W.

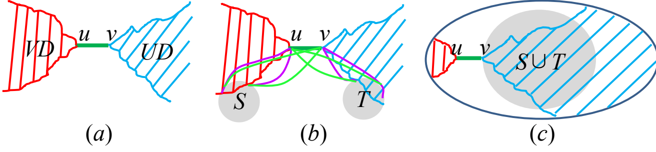


Fig. 7. Bridge domain

Proof: Suppose that $sp(s,t)$ does not pass through a bridge. For any cut sp that forms a boundary of W, $sp(s,t)$ does not go across sp according to Corollary 1. The corollary follows since sp is an arbitrary cut that forms a boundary of W. ■

We now compute a DPS G' such that G' contains $sp(s,t)$ for all $s \in S$ and $t \in T$, whether or not $sp(s,t)$ passes through a bridge. We assume that all the vertices contained in the window W computed in Section IV-C are already added into V', before we consider the bridges.

1) Bridge Domain: To include the shortest paths passing through a bridge (u,v) into the DPS G', we need the concept of **bridge domain**: a bridge (u,v) has two domains $UD = \{x \in V | dist(x,u) = dist(x,v) + |vu|\}$ and $VD = \{x \in V | dist(x,v) = dist(x,u) + |uv|\}$.

Intuitively, UD contains all the vertices $x \in V$ with $sp(x,u)$ passing through v, and VD contains all the vertices $x \in V$ with $sp(x,v)$ passing through u. Figure 7(a) shows an example of UD and VD of the bridge (u,v).

We now present two properties of the bridge domains.

Theorem 4: UD and VD are disjoint, i.e. $UD \cap VD = \emptyset$.

Proof: We first prove that if $x \in UD$, then $x \notin VD$: if $x \in UD$, $dist(x,u) = dist(x,v)+|vu|$, and thus $dist(x,v) < dist(x,u) < dist(x,u)+|uv|$, i.e., $x \notin VD$. Similarly, we can prove that $x \notin UD$ if $x \in VD$, and thus $UD \cap VD = \emptyset$. ■

Theorem 5: If $s \notin VD$ or $t \notin UD$, $sp(s,t)$ does not pass through (u,v).

Proof: If $sp(s,t)$ passes through (u,v), then (1)$sp(s,v)$ must pass through (u,v), and thus $s \in VD$; and (2)$sp(u,t)$ must pass through (u,v), and thus $t \in UD$. ■

Since the graph is undirected, we also have: if $s \notin UD$ or $t \notin VD$, $sp(s,t)$ does not pass through $(v,u) = (u,v)$.

We use UD^* to denote $UD \cap (S \cup T)$, and use VD^* to denote $VD \cap (S \cup T)$. Theorem 5 implies that if $UD^* = \emptyset$ or $VD^* = \emptyset$, the bridge (u,v) can be safely **pruned**, i.e., we do not need to consider (u,v) in computing the DPS.

Figure 7(c) shows an example where $VD^* = \emptyset$ and thus (u,v) is pruned. Figure 7(b) shows an example where $VD^* \neq \emptyset$ and $UD^* \neq \emptyset$, in which case bridge (u,v) cannot be pruned.

Since the graph is undirected, there are two cases for each bridge (u,v): (1)$s \in VD$ and $t \in UD$, and (2)$s \in UD$ and $t \in VD$. We include the paths $sp(x,u)$ and $sp(v,x)$, for all $x \in UD^* \cup VD^*$, into the GPS G' computed in Section IV-C. This operation can be done in $O(|V|)$ time using the vertex collection method presented in Section III-A.

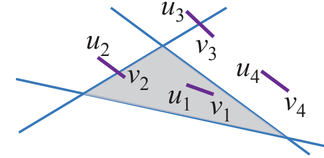


Fig. 8. Bridge pruning

2) Domain Computation: Given a bridge (u,v), we compute UD^* and VD^* simultaneously by Dijkstra's algorithm. In Dijkstra's algorithm, we use a min-heap to keep those vertices whose distance from the source vertex has not been determined, where the key is the estimated distance of each vertex in the heap. We maintain two min-heaps, $\mathcal{Q}_u$ and $\mathcal{Q}_v$, for running Dijkstra's algorithm from u and v, respectively, and compute UD^* and VD^* as follows.

Each time, we compare the minimum keys of $\mathcal{Q}_u$ and $\mathcal{Q}_v$. Assume that $\mathcal{Q}_u$ has a smaller minimum key than $\mathcal{Q}_v$ (the case when $\mathcal{Q}_v$ has a smaller minimum key is symmetric), then we remove the vertex, x, that has the minimum key from $\mathcal{Q}_u$. The exact distance of x from u, i.e., $dist(x,u)$, is then computed. If $dist(x,v)$ has been computed, i.e., x is removed from $\mathcal{Q}_v$ before it is removed from $\mathcal{Q}_u$ due to $dist(x,v) \leq dist(x,u)$, then $x \notin VD$ and we only check whether $x \in UD$, i.e., $dist(x,u) = dist(x,v) + |vu|$. If so, x is put into UD^* only if $x \in S \cup T$.

Since we only compute UD^* and VD^*, we may stop running Dijkstra's algorithm as soon as the distance from u and v to all the vertices in $S \cup T$ are computed.

C. Bridge Categorization and Pruning

So far, we prune a bridge (u,v) only if UD^* or VD^* is empty. However, it is too expensive to compute UD^* and VD^* for every bridge (u,v). We now show that only a small fraction of the bridges needs to be examined, while the others can be pruned directly without computing UD^* and VD^*.

Bridge categorization. With respect to the window W computed in Section IV-C, a bridge (u,v) belongs to one of the following three types:

- **Interior bridge**: (u,v) is inside W, e.g., (u_1,v_1) in Figure 8.
- **Cut bridge**: (u,v) crosses a cut that forms a boundary of W, e.g., (u_2,v_2) and (u_3,v_3) in Figure 8.
- **Exterior bridge**: (u,v) is neither an interior bridge nor a cut bridge, e.g., (u_4,v_4) in Figure 8.

We can determine the bridge type using $vec(u)$, $vec(v)$, and W. Consider the case when the cuts from b_i divide the road network into zones. Given a vertex v with label $vec(v)[i] = [l_v, h_v]$, and suppose that $W[i] = [l_W, h_W]$, we define a comparison operation $comp(vec(v)[i], W[i])$ as follows:

- If $l_v > h_W$, $comp(vec(v)[i], W[i]) = 1$:
  - e.g., if $vec(v)[i] = [5,6]$ and $W[i] = [3,4]$, then $comp(vec(v)[i], W[i]) = 1$, meaning that $vec(v)[i]$ is *strictly larger* than $W[i]$.

- If $l_W > h_v$, $comp(vec(v)[i], W[i]) = -1$:
  - e.g., if $vec(v)[i] = [1, 2]$ and $W[i] = [3, 4]$, then $comp(vec(v)[i], W[i]) = -1$, meaning that $vec(v)[i]$ is *strictly smaller* than $W[i]$.
- Otherwise, $comp(vec(v)[i], W[i]) = 0$:
  - e.g., if $vec(v)[i] = [2, 3]$ and $W[i] = [3, 4]$, then $comp(vec(v)[i], W[i]) = 0$, meaning that v is in Zone 3, which is contained by $W[i]$.

We now present how to determine whether a bridge (u, v) is a cut bridge, using $vec(u)$, $vec(v)$, and W:

Observation 1: Given a bridge (u, v), Let us use $comp_u$ and $comp_v$ to denote $comp(vec(u)[i], W[i])$ and $comp(vec(v)[i], W[i])$, respectively. We also use sp_1 and sp_2 to denote the two cuts that form the boundary of $W[i]$ in dimension i. Then, (u, v) is a cut bridge if one of the following three cases is true for some dimension i: (1) $comp_u \cdot comp_v = -1$; or (2) $comp_u = 0$ and $comp_v \neq 0$; or (3) $comp_u \neq 0$ and $comp_v = 0$.

Proof: Note that the cuts sp_1 and sp_2 (from a border vertex b_i) divide the space into three parts, assuming that the three sub-spaces are: Space 1 (which is the side of sp_1 that does not contain W), Space 2 (which is between sp_1 and sp_2, i.e., $W[i]$), and Space 3 (which is the side of sp_2 that does not contain W).

In Case (1), u falls in Space 1 (or Space 3) and v falls in Space 3 (or Space 1). In Case (2), u falls in Space 2 and v falls in either Space 1 or Space 3. In Case (3), v falls in Space 2 and u falls in either Space 1 or Space 3. Since u and v are in two different spaces in each of the three cases, (u, v) is a cut bridge crossing sp_1 or/and sp_2. ∎

Bridge pruning. Although we can also find interior and exterior bridges, we show that it is not necessary to find these bridges: they can be pruned directly since only cut bridges need to be examined.

Lemma 3: Given a cut sp, let s and t be two vertices on one side of sp. If $sp(s, t)$ needs to go across sp to the other side, then $sp(s, t)$ passes through a bridge crossing sp.

Proof: Referring to the proof of of Lemma 2: since $sp(s, t)$ needs to go across sp to the other side, m_1 and m_2 cannot both occur. Therefore, $sp(s, t)$ passes through a bridge crossing sp. ∎

Theorem 6: Interior and exterior bridges can be pruned.

Proof: We first prove that interior bridges do not need to be examined. For any $s \in S$ and $t \in T$, suppose that $sp(s, t)$ passes through n interior bridges $(u_1, v_1), (u_2, v_2), \ldots, (u_n, v_n)$. Let us define $SP = \{sp(s, u_1), sp(v_1, u_2), \ldots, sp(v_n, t)\}$, then all the sources and destinations of the paths in SP are inside W. Case 1: if any path in SP passes through a non-interior bridge (u', v'), $sp(s, t)$ is already included into G' when we process (u', v'). Case 2: otherwise, any path in SP does not pass though a bridge (interior or non-interior), and according to Corollary 2, all the paths in SP are inside W. Therefore, $sp(s, t)$ is inside W and thus inside G'.

Next, we prove that external bridges do not need to be examined. For any $s \in S$ and $t \in T$, suppose that $sp(s, t)$ passes through an external bridge (u, v), meaning that u and v are on one side of some cut (which can be any cut that forms a boundary of W), while W is (and thus s and t are) on the other side of the cut. According to Lemma 3, $sp(s, t)$ passes through a bridge (u', v') crossing the cut. Note that (u', v') is a cut bridge, and thus $sp(s, t)$ is already included into G' when we process the cut bridge (u', v'). ∎

For the cut bridges, we further show that only a small portion of them need to be examined. We identify those cut bridges by running the BL-E algorithm presented in Section III-B, as stated by the following corollary of Theorem 1:

Corollary 3: A cut bridge (u, v) can be pruned if $dist(v_c, u) > 2r$ or $dist(v_c, v) > 2r$.

Finally, we present our last pruning rule for cut bridges. Let $\mathcal{L}$ be a list of cut pairs (sp_1^i, sp_2^i) that form the boundary of W, where sp_1^i and sp_2^i are the two cuts from the border vertex b_i that bounds $W[i]$. Assume that the cut pairs in $\mathcal{L}$ are in the same order as the label dimension index, i.e. $\mathcal{L} = \{(sp_1^1, sp_2^1), \ldots, (sp_1^\ell, sp_2^\ell)\}$. Then, we have the following pruning rule for cut bridges:

Theorem 7: Given a cut bridge (u, v), let (sp_1^j, sp_2^j) be the first cut pair in $\mathcal{L}$ such that (u, v) crosses sp_1^j (or sp_2^j). Then, (u, v) can be pruned if there exists $i < j$, such that u and v are both on the side of sp_1^i (or sp_2^i) that does not contain W.

Proof: Let S_k be the set of cut bridges (u', v') that are not pruned by Theorem 7, such that (sp_1^k, sp_2^k) is the first cut pair in $\mathcal{L}$ with (u', v') crossing sp_1^k (or sp_2^k). Thus, S_1 consists of all the cut bridges crossing sp_1^1 or sp_2^1.

Now consider a cut bridge (u, v) described in Theorem 7 and assume that u and v are both on the side of sp_1^i that does not contain W (the case for sp_2^i is symmetric). For any $s \in S$ and $t \in T$, suppose that $sp(s, t)$ passes through (u, v). Then, according to Lemma 3, $sp(s, t)$ passes through a bridge (u_1, v_1) crossing the cut sp_1^i. If $(u_1, v_1) \in S_i$, then $sp(s, t)$ is already included into G' when (u_1, v_1) is processed. Otherwise, there must exist $i' < i$, such that u_1 and v_1 are both on the same side of $sp_1^{i'}$ (or $sp_2^{i'}$) that does not contain W. A recursive analysis will find a cut bridge (u', v') in some $S_{i'}$ such that (u', v') includes $sp(s, t)$ into G' when it is processed, since when $i' = 1$, we have all the cut bridges crossing sp_1^1 or sp_2^1 in S_1. ∎

We can determine that u and v are both on the side of sp_1^i (or sp_2^i) that does not contain W, if $comp(vec(u)[i], W[i]) \cdot comp(vec(v)[i], W[i]) = 1$.

Theorem 7 also implies that we can further reduce the number of cut bridges to examine by ordering the cut pairs in $\mathcal{L}$ in non-decreasing number of cut bridges crossing either cut in a pair, instead of being simply ordered by the label dimension index.

D. Complexity Analysis

Let d be the maximum number of edges checked in an intersection query on $Rtree(E)$. Then, finding the bridges

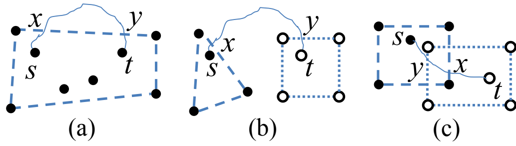


Fig. 9. An illustration of the convex hull based algorithms

by the indexed-nested-loop join takes $O(|E| \cdot (d \log |E|)) = O(|V| \log |V|)$ time, since d is a small constant for road networks. Thus, the overall $O(\ell^2 |V| \log |V|)$ time complexity of index construction obtained in Section IV-B remains.

Let b be the number of bridges that require domain computation. Since examining a bridge takes $O(|V| \log |V|)$ time, computing W takes $O(\ell|\mathcal{R}|)$ time and constructing G' from W takes $O(V)$, processing a DPS query with bridges takes $O(\ell|\mathcal{R}| + b|V| \log |V|) \approx O(b|V| \log |V|)$ time. Note that b is a small number since only a small portion of cut bridges need to be examined after pruning, as verified by our experiments.

VI. A Convex Hull Method

In this section, we propose a convex hull based method to further tighten the DPS obtained by RoadPart.

The convex hull of a finite point set P in a plane, denoted $hull(P)$, is the minimal convex polygon containing P, and $hull(P)$ can be computed in $O(|P| \log |P|)$ time, using Andrew's Monotone Chain algorithm [11].

A. Processing Q-DPS Queries

Algorithm 1 shows our convex hull method for processing Q-DPS queries. The input H is a DPS obtained by RoadPart, and the output is a tightened DPS G'. The input H can also be the original road network, but the computation is expensive when the road network is large.

The algorithm first computes $hull(Q)$ by Andrew's Monotone Chain algorithm [11]. For example, in Figure 9(a) the 4-sided polygon is $hull(Q)$. We add into V' those vertices of H that are contained in polygon $hull(Q)$. (e.g., the vertices inside the 4-sided polygon such as s and t). Then, we add the vertices of the polygon to $border(Q)$ (e.g., the four vertices at the four corners of the 4-sided polygon). We also find the edges in H that intersect with each edge $q_i q_j$ on the polygon, and add the intersection points into $border(Q)$. We use $q_i q_j$ to denote a polygon edge to distinguish it from a graph edge (u, v). Finally, we compute the shortest paths $sp(s, t)$ for all $s, t \in border(Q)$, and add the vertices on the paths to V'. We return the subgraph of H induced by V' as the DPS G'.

We now prove the correctness of Algorithm 1.

Theorem 8: The graph G' returned by Algorithm 1 is a DPS for Q.

Proof: We prove that $\forall s, t \in Q$, the vertices on $sp(s, t)$ are in V'. We have two cases. Case 1: if all the vertices on $sp(s, t)$ are contained in polygon $hull(Q)$, then they are included in V' in Line 2. Case 2: otherwise, there exists a vertex x (and y) being the first (and last) point on $sp(s, t)$ that passes across $hull(Q)$ (see Figure 9(a)). The vertices on $sp(s, x)$ and $sp(y, t)$ are added to V' in Line 2, and the vertices on $sp(x, y)$ are added to V' in Lines 7-8. Thus, the vertices on $sp(s, t)$ are all in V'. ■

Algorithm 1 Convex Hull Method for Q-DPS Query

Input: a road network H, and a query set Q
Output: a DPS $G' = (V', E')$ for Q

1: Compute a convex polygon $hull(Q)$;
2: Add vertices of H that are contained in polygon $hull(Q)$ to V';
3: $border(Q) \leftarrow$ the set of vertices of polygon $hull(Q)$;
4: **for each** edge $q_i q_j$ on polygon $hull(Q)$ **do**
5: Find the edges in H intersecting with $q_i q_j$;
6: Add the intersection points to $border(Q)$;
7: **for** any two points $s, t \in border(Q)$ **do**
8: Add the vertices on $sp(s, t)$ to V';
9: Return G' as the subgraph of H induced by V';

Algorithm 2 Convex Hull Method for (S, T)-DPS Query

Input: a road network H, and query sets S and T
Output: a DPS $G' = (V', E')$ for (S, T)

1: Compute convex polygons $hull(S)$ and $hull(T)$;
2: Add vertices of H that are contained in polygon $hull(S)$ to V';
3: Add vertices of H that are contained in polygon $hull(T)$ to V';
4: Compute $border(S)$ and $border(T)$ using Lines 3-6 of Algorithm 1, by replacing Q with S and T, respectively;
5: **for** any two points $s \in border(S)$, $t \in border(T)$ **do**
6: Add the vertices on $sp(s, t)$ to V';
7: Return G' as the subgraph of H induced by V';

The time complexity of Algorithm 1 is dominated by that of the SSSP computations in Lines 7-8, i.e., computing the shortest paths from each $v \in border(Q)$ to all other vertices in $border(Q)$, which takes $O(|border(Q)| \cdot |V| \log |V|)$ time. Since $|border(Q)|$ is approximately $\sqrt{|Q|}$, Algorithm 1 takes $O(\sqrt{|Q|} \cdot |V| \log |V|)$ time.

B. Processing (S, T)-DPS Queries

The convex hull algorithm for processing (S, T)-DPS queries, as given in Algorithm 2, is similar to Algorithm 1, except that we process two point sets S and T.

The time complexity of Algorithm 2 is dominated by that of computing the shortest paths from each $s \in border(S)$ to each $t \in border(T)$ in Lines 5-6, which takes $O(\min\{|border(S)|, |border(T)|\} \cdot |V| \log |V|)$ time. Thus, Algorithm 2 takes $O(\sqrt{\min\{|S|, |T|\}} \cdot |V| \log |V|)$ time.

We now prove the correctness of Algorithm 2.

Theorem 9: The graph G' returned by Algorithm 2 is a DPS for (S, T).

Proof: We prove that $\forall s \in S, t \in T$, the vertices on $sp(s, t)$ are in V'. We have two cases. Case 1: if all the vertices on $sp(s, t)$ are inside the regions bounded by polygons $hull(S)$ or $hull(T)$, then they are added to V' in Lines 2-3. Case 2: otherwise, there exists a vertex x (and y) being the first (and last) point on $sp(s, t)$ that passes across $hull(S)$ (and $hull(T)$). This may happen as shown in Figure 9(b), i.e., y is not on $sp(s, x)$, in which case the vertices on $sp(s, x)$ and $sp(y, t)$ are added to V' in Lines 2-3, and the vertices on $sp(x, y)$ are added to V' in Lines 5-6. Or it may happen as in Figure 9(c),

TABLE I
REAL ROAD NETWORK DATASETS AND INDEX CONSTRUCTION RESULTS

Name	Data Size	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert E_b\rvert$	$\lvert E_b\rvert/\lvert E\rvert$	$\ell = \lvert B\rvert$	Indexing Time (sec)	Index Size	$\lvert\mathcal{R}\rvert$
COL (Colorado)	32.7 MB	435,666	521,200	2,691	0.516%	20	22.8	3.1 MB	969
NW (Northwest USA)	76.1 MB	1,207,945	1,410,387	10,530	0.747%	50	372.5	9.5 MB	3014
EAST (Eastern USA)	235.7 MB	3,598,623	4,354,029	15,919	0.366%	45	1066.8	28.9 MB	4242
USA (Full USA)	1.63 GB	23,947,347	28,854,312	108,808	0.377%	70	26341	210 MB	11629

i.e., y is on $sp(s,x)$, in which case the vertices on $sp(s,y)$ and $sp(y,t)$ are added to V' in Lines 2 and 3, respectively. Thus, in all cases, the vertices on $sp(s,t)$ are added to V'. ■

VII. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of our algorithms using large real road networks. All the experiments are run on a Linux server with eight 3GHz Intel CPU and 32GB memory.

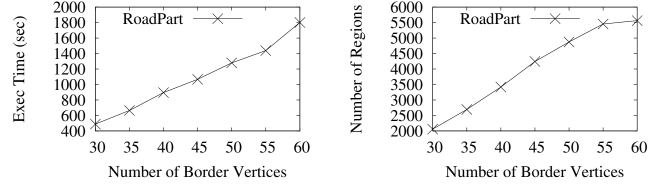
Datasets. The experiments are conducted on four real road network datasets from [18]. Table I summarizes the datasets, where $|E|$ corresponds to the number of *undirected* edges, and $|E_b|$ corresponds to the number of bridges. We can see that only a very small fraction of the edges are bridges. Furthermore, we scale the edge weights to ensure that $|uv| \geq \|uv\|$ for each edge (u,v) [3], which is required by the A* algorithm used for cut computation.

A. Results on Graph Partitioning for RoadPart Indexing

To partition a road network, we first need to determine the number of border vertices ℓ (or $|B|$). We measure the evenness of the region size, by the size of the largest region, M, where the size of a region refers to the number of vertices it contains. Starting from $\ell = 30$, we increase ℓ by 5 each time and partition the road network using ℓ. As ℓ increases, the maximum region size M decreases sharply when ℓ is small, and becomes stable when ℓ is reasonably large.

We report the performance of graph partitioning for various values of ℓ. Figure 10(a) shows the partitioning time of RoadPart on the EAST dataset for various values of ℓ, and Figure 10(b) shows the number of regions $|\mathcal{R}|$ obtained by partitioning. The results on other datasets are similar and thus omitted. Although the theoretical time complexity of our partitioning algorithm is quadratic in ℓ, in practice our results show that the partitioning time and the number of regions increase almost linearly as ℓ increases. This is because, in each of the ℓ rounds of vertex labeling, the time for computing ℓ cuts using the A* algorithm is insignificant compared with the time for the in-zone BFS based vertex labeling.

We increase ℓ until the maximum region size M becomes stable, and choose the value of ℓ that leads to the smallest M for partitioning. The obtained region set $\mathcal{R}$ is then used for later experiments. Table I reports the value of ℓ we set for each dataset, the indexing time (mostly for graph partitioning), the index size, and the number of regions partitioned. We can see that the offline indexing time and the index size are acceptable even for a road network with tens of millions of vertices.



(a) Running Time (b) Number of Regions $|\mathcal{R}|$

Fig. 10. Effect of ℓ on graph partitioning for RoadPart indexing

Compared with the size of the road networks, the size of our index is about an order of magnitude smaller.

B. Results on DPS Query Processing

To assess the performance of our algorithms for processing DPS queries, we first discuss how we generate the queries and the measures we use to record the performance. Then, we present the performance results.

Query generation. To study the scalability of our algorithms for answering DPS queries, for each road network dataset $G = (V,E)$, we generate a query set S and T (or Q) as follows. Let us denote the MBR of all the vertices in V by $mbr(V)$, and denote the width (height) of $mbr(V)$ by W (H). We first generate a $\varepsilon W \times \varepsilon H$ rectangular window over G, where $\varepsilon < 1$ is a parameter that controls the window size, and then put all the vertices in the window into the query set.

For an (S,T)-DPS query, we generate both S and T using the same ε. Furthermore, we use another parameter ε' to control how far S and T are: the distance between the window centers is equal to $\varepsilon' W$.

Measures. During the query processing of RoadPart, we call a bridge (u,v) as *examined* if it is not pruned by the rules described in Section V-C, and thus UD^* and VD^* are computed as discussed in Section V-B2; we call (u,v) *valid* if it introduces vertices into V' (i.e., when $UD^* \neq \emptyset$ and $VD^* \neq \emptyset$).

We use the following measures to evaluate the performance of our algorithms: (1) query processing time; (2) DPS size; (3) the number of examined bridges; and (4) the number of valid bridges.

Query processing time. Table II shows some representative results about the performance of our algorithms on the real datasets. The complete results are given in an online appendix [19]. For Q-DPS queries, we vary ε (and thus $|Q|$); while for (S,T)-DPS queries, we fix ε and vary ε' (and thus, how far apart S and T are).

TABLE II
RESULTS ON QUERY PROCESSING TIME (WALL CLOCK TIME IN SEC) AND DPS QUALITY

ε	$\|Q\|$	BL-E		RoadPart				Convex Hull Method			BL-Q	
		Time	$\|V'\|$	Time	b	b_v	$\|V'\|$	Time	$\|border\|$	$\|V'\|$	Time	$\|V'\|$
Q-DPS queries on USA												
2%	16,562	0.4	194,095	2.3	3	0	103,633	102 (13.9)	369	18,497	3,838	17,561
4%	66,443	0.8	930,999	42.7	25	0	345,524	441 (120)	853	74,018	29,455	71,542
6%	177,876	1	1,450,931	86.2	40	0	536,824	1,567 (336)	1,701	190,441	129,088	184,809
8%	347,308	1.9	2,693,966	154	49	0	729,538	2,400 (476)	1,708	363,336	419,529	355,026
10%	507,807	2.7	3,892,420	319	76	0	894,506	3,534 (696)	1,833	528,258	866,048	521,651
Q-DPS queries on EAST												
5%	21,190	0.3	423316	7	10	0	106,750	49 (17.9)	448	24,787	1,831	22,974
10%	90,567	1	1369625	18.1	18	0	154,456	347 (89.4)	691	96,432	29,773	95,009
15%	241,118	1.5	2606948	45.9	45	5	503,897	920 (233)	1,134	251,748	164,159	245,934
20%	448,735	1.7	3010128	35.7	37	1	865,989	1,644 (645)	1,774	466,701	403,426	457,645
25%	665,259	1.8	3193448	82.1	72	2	1,136,570	2,378 (882)	1,956	688,826	737,853	677,231
Q-DPS queries on COL												
10%	6,112	0.1	98,251	1.4	14	0	44,909	3.1 (1)	156	6,999	96.5	6,665
20%	30,968	0.2	286,347	5.7	37	1	108,670	52 (17.1)	598	37,679	2,364	34,089
30%	81,574	0.2	399,698	4.9	30	1	162,404	86.9 (36.8)	787	94,122	9,875	86,310
40%	156,485	0.2	435,079	4.9	29	2	263,346	93.4 (59.9)	749	165,820	23,533	159,623
50%	204,812	0.2	435,666	4.9	28	1	287,775	95.6 (60.3)	709	212,930	38,011	208,885

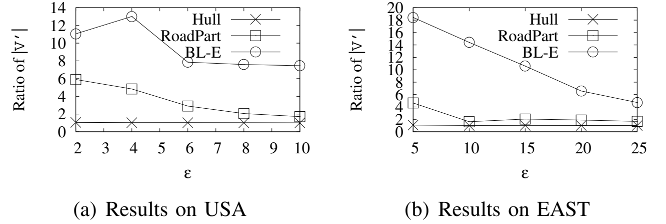
ε'	$\|S\|$	$\|T\|$	BL-E		RoadPart				Convex Hull Method			BL-Q	
			Time	$\|V'\|$	Time	b	b_v	$\|V'\|$	Time	$\|border\|$	$\|V'\|$	Time	$\|V'\|$
(S,T)-DPS queries on USA (ε=0.04)													
2%	60,011	84,939	1	1,255,634	55	30	0	419,391	487 (108)	818	120,747	40,400	116,744
4%	54,413	87,808	1.6	2,062,571	141	52	0	606,434	619 (173)	738	152,790	41,170	147,808
6%	47,055	110,622	2.2	3,101,715	149	52	1	729,539	4,409 (171)	689	203,071	58,828	172,256
8%	40,416	102,353	2.7	3,959,442	346	79	0	854,078	1,016 (194)	688	168,401	61,957	162,493
10%	30,361	88,547	3.4	4,951,731	499	85	1	913,247	2,082 (274)	711	147,233	60,415	141,731

For RoadPart, we report the number of examined bridges, denoted by b, and the number of valid bridges, denoted by b_v, in Table II. For the convex hull method, we report its running time on the original road network, as well as that on the DPS obtained by RoadPart (in the parentheses). We also report $|border(Q)|$ (or $\min\{|border(S)|, |border(T)|\}$), denoted by $|border|$, in Table II. We find in our experiments that $|border|$ and $|V'|$ are the same for the convex hull method whether the input is the original road network or the DPS, which shows that the DPS found by the convex hull method is strictly tighter than the one found by RoadPart.

The results show that BL-E answers a Q-DPS query in seconds, RoadPart in minutes, the convex hull method in less than an hour, and BL-Q in many hours. For RoadPart, only a small number of bridges need to be examined, which is a very small fraction of all the bridges (see the "$|E_b|$" column in Table I). Besides, at most a few of these bridges introduce vertices into V'. The results thus verify the effectiveness of our bridge pruning rules.

RoadPart is over one order of magnitude faster than the convex hull method on the original road network, and scales well with $|Q|$. This result is because examining a bridge (u, v) requires SSSP computation with sources u and v, for which RoadPart performs $2b$ rounds of SSSP computation if b bridges are examined, while the convex hull method requires $|border|$ rounds of SSSP computation. As shown in Table II, $|border|$ is much larger than $2b$ and thus the convex hull method is much more expensive.

We also note that running the convex hull method on the DPS obtained by RoadPart is several times faster than running it on the original road network, even if we include the query processing time of RoadPart in the total running time.



(a) Results on USA (b) Results on EAST

Fig. 11. DPS quality comparison

DPS quality/size. We can see from Table II that $|Q|$ is quadratic in ε. This is because, the size of the query window is $\varepsilon W \times \varepsilon H$, which is quadratic in ε.

Recall that BL-Q returns the smallest DPS; thus, we compare the DPS size of the other three algorithms using that of BL-Q as the baseline. Given a query set Q, suppose that algorithm $\mathcal{A}$ returns a DPS $G'_{\mathcal{A}} = (V'_{\mathcal{A}}, E'_{\mathcal{A}})$ and BL-Q returns a DPS $G'_* = (V'_*, E'_*)$, then we define the *V-ratio* of algorithm $\mathcal{A}$ as $|V'_{\mathcal{A}}|/|V'_*|$ (≥ 1). Smaller V-ratio implies higher DPS quality.

Figure 11 shows the V-ratio of our algorithms on datasets USA and EAST for Q-DPS queries, where the V-ratio of all the algorithms decreases as ε increases. The V-ratio of BL-E is large, which shows that although BL-E finds a DPS in just several seconds, the DPS is usually too loose to be useful.

On the other hand, the V-ratio of the convex hull method is

always close to 1 (never exceeds 1.1), which verifies that the method finds a very high-quality DPS. Note that the convex hull method always finds a DPS within an hour, while BL-Q takes over a week to find the smallest DPS when $|Q|$ is moderately large.

As for RoadPart, the DPS returned is already tight (the V-ratio is smaller than 2) when ε is up to 10%. But when $|Q|$ is too small, the DPS returned by RoadPart is not sufficiently tight. This is because each region $R \in \mathcal{R}$ has certain granularity, and as long as a vertex $v \in R$ is in Q (or $S \cup T$), all the vertices in R are included in V'. For (S, T)-DPS queries, the DPS returned is not as tight as the convex hull method, especially when S and T are far apart (see Table II). This is because all the vertices in the window W determined by (S, T) are included in V', although for any $s \in S, t \in T$, $sp(s, t)$ only go through several highway paths between S and T.

C. Results on Query Processing over a DPS

We now compare the performance of query processing on the DPS with that on the original road network. Table II has already shown that the convex hull method is much more efficient on the DPS returned by RoadPart than on the original road network.

We now compare the performance of PPSP computation (i.e., the A* algorithm) on a DPS with that on the original road network. We randomly generate 1000 vertex pairs (s, t) according to the DPS query set, and compare the time for computing the shortest paths $sp(s, t)$ for all the pairs.

We find that PPSP computation is much faster on a DPS. For Q-DPS queries on the USA dataset, when ε is 2%, it takes 173 seconds to find all the 1000 paths on the road network, 4.2 seconds on the DPS returned by RoadPart, and 1.8 seconds on the DPS returned by the convex hull method. When ε is 6%, it takes 394 seconds on the original road network, 55 seconds on the DPS returned by RoadPart, and 31 seconds on the DPS returned by the convex hull method. The complete results are reported in our online appendix [19].

Shortest path computation is faster on a DPS because vertices in $(V - V')$ are neither initialized (by setting the distance estimations to $+\infty$) nor visited. Since network distance computation is basic to many other queries in road networks [2], [3], [4], we expect that it is also much faster to process these queries on the DPSs than on the original road network.

VIII. Related Work

Road network partitioning has been used for point to point shortest path query indexing [6] and monitoring proximity relations [1]. [6] proposed to partition the road network by a vertex k-d tree, while [1] proposed to partition the road network by cuts like RoadPart. However, [1] adopts a grid-like partitioning scheme that only allows graph pruning on two dimensions, while RoadPart allows pruning on ℓ dimensions. Furthermore, the partitioning scheme of [1] may not be correct for real road networks since it assumes that a road network is planar. Finally, [1] does not give an algorithm for finding a contour and labeling the vertices.

Convex hull has been applied in [4] to process optimal meeting point queries. In this work, we use convex hull to help find a tight DPS.

A large number of shortest path indices have been proposed for road networks, including [7], [8], [9], [10] that require pre-materialization of all-pair shortest paths, and [15], [16] that focus on techniques for processing queries online efficiently. Our work is orthogonal to these works, since we find a DPS for a query point set, and the existing shortest path indices can be built on the DPS.

IX. Conclusions

In this paper, we propose four algorithms to answer *distance-preserving subgraph* (DPS) queries. Among the algorithms, BL-E is fast but the DPS it computes is too loose, while BL-Q computes the smallest DPS but is too slow. On the contrary, our RoadPart framework provides a nice tradeoff between DPS quality and query processing time, and can be used at the server-end for finding DPS, which fits many real life application scenarios as we discussed in Section I. At the client-end, we recommend to use the convex hull method to refine the DPS returned from the server, which is considerably faster than BL-Q and returns a very tight DPS.

References

[1] Z. Xu and H.-A. Jacobsen. "Processing Proximity Relations in Road Networks". In *SIGMOD*, 2010.
[2] X. Xiao, B. Yao and F. Li. "Optimal Location Queries in Road Network Databases". In *ICDE*, 2011.
[3] M. L. Yiu, N. Mamoulis and D. Papadias. "Aggregate Nearest Neighbor Queries in Road Networks". In *TKDE*, 2005.
[4] D. Yan, Z. Zhou and W. Ng. "Efficient Algorithms for Finding Optimal Meeting Point on Road Networks". In *VLDB*, 2011.
[5] Z. Xu. "Efficient Location Constraint Processing for Location-Aware Computing". Ph.D. Thesis, Univ. of Toronto, 2009.
[6] G. Kellaris and K. Mouratidis. "Shortest Path Computation on Air Indexes". In *VLDB*, 2010.
[7] H. Samet, J. Sankaranarayanan and H. Alborzi. "Scalable Network Distance Browsing in Spatial Databases". In *SIGMOD*, 2008.
[8] J. Sankaranarayanan, H. Samet and H. Alborzi. "Path Oracles for Spatial Networks". In *VLDB*, 2009.
[9] E. Cohen, E. Halperin, H. Kaplan and U. Zwick. "Reachability and Distance Queries via 2-Hop Labels". In *SODA*, 2002.
[10] R. Jin, N. Ruan, Y. Xiang and V. E. Lee. "A Highway-Centric Labeling Approach for Answering Distance Queries on Large Sparse Graphs". In *SIGMOD*, 2012.
[11] F. P. Preparata and M. I. Shamos. "Computational Geometry: An Introduction". *Springer-Verlag*, 1985.
[12] S. T. Leutenegger, J. M. Edgington and M. A. Lopez. "STR: A Simple and Efficient Algorithm for R-tree Packing". In *Technical Report, Institute for Computer Applications in Science and Engineering*, 1997.
[13] S. Shekhar, A. Kohli and M. Coyle. "Path Computation Algorithms for Advanced Traveller Information System (ATIS)". In *ICDE*, 1993.
[14] S. Yang, X. Yan, B. Zong and A. Khan. "Towards Effective Partition Management for Large Graphs". In *SIGMOD*, 2012.
[15] R. Geisberger, P. Sanders, D. Schultes and D. Delling. "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks". *Experimental Algorithms*, 2008.
[16] Y. Tao, C. Sheng and J. Pei. "On k-skip Shortest Paths". In *SIGMOD*, 2011.
[17] http://research.microsoft.com/en-us/projects/trinity
[18] http://www.dis.uniroma1.it/challenge9/download.shtml
[19] http://www.cse.ust.hk/~yanda/papers/partApp.pdf