Shortest Distance and Path Queries on Spatial Networks

Victor Junqiu Wei
Supervisor: Raymond Chi-Wing Wong

Department of Computer Science and Engineering
Hong Kong University of Science and Technology

28th May, 2017

Abstract

Given two vertices of interest (POIs) s and t on a spatial network, a distance (path) query returns the shortest network distance (shortest path) from s to t. This query has far-reaching applications in practice and is a fundamental operation for many database and data mining algorithms.

Due to its importance, a variety of research efforts have been put into this problem. This paper surveys the existing works on the shortest distance and path query on the spatial networks. Most of the reviewed works pre-process an indexing structure to accelerate the query. We compared their performance in the perspectives of both theory and practice.

1 Introduction

The spatial network is a data model widely used in the geographical information system (GIS) and the scientific simulation [24, 36, 9]. A spatial network is a graph in which each vertex has a location and each edge has a weight, the physical meaning of which could be distance, travel time, energy cost, etc. In this network, the movement of an object is constrained to the network. Figure 1 shows an example of a spatial network. There are 8 vertices in the network, namely $v_1, v_2, ..., v_8$, each of which has a location. Each edge has an origin and a destination as well as a weight. A fundamental problem in this model is the shortest path (resp. distance) query. Specifically, given two vertices s and t in the spatial network, find the shortest network path (distance) from s to t. This query has a variety of applications in practice. For example, in the map service, a user gives his location A and his destination B and tries to find the shortest path from A to B. Besides, this query is also a building block for many database and data mining algorithms.

It is well-known that Dijkstra's shortest path algorithm [16] takes $O(n \log n + m)$ time, where n (m) is the number of vertices (edges) in the spatial network. Although Dijkstra's algorithm is simple and elegant, it is inefficient for the sizable networks and generate prominent delay in practice.

Thus, people proposed the preprocessing-query framework, in other words, an indexing structure, to accelerate the query by preprocessing the network. In this paper, we survey the existing works on the shortest distance and path query on the spatial network. We extracted the main idea of each reviewed technique and provide the deep insight into their performance.

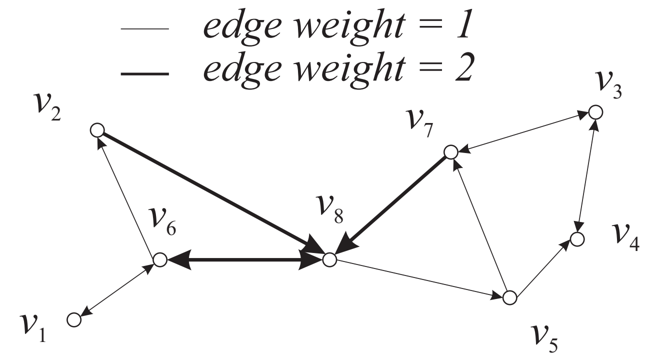


Figure 1: An Example of A Spatial Network

The remainder of the paper is organized as follows. Section 2 gives the key concepts and the problem definition and Section 3 gives the detailed descriptions, discussions and comparisons on the existing works. Section 4 concludes our work.

2 Problem Definition

Consider a spatial network $G(V, E)$, where V is a set of all vertices on the network and E is a set of all edges on G. The two incident vertices of each edge in E come from V. Let $n = |V|$ and $m = |E|$.

Given an edge (u, v) whose origin is u and destination is v, we denote the weight of (u, v) by $w_G(u, v)$. The subscript G indicates that the weight $w_G(u, v)$ is the weight of an edge in the original spatial network G. Consider the edge (v_5, v_4) in Figure 1 with its origin as v_5 and its destination as v_4 and its weight $w_G(v_5, v_4)$ as 1.

Given a network $G(V, E)$ and a node u, where $u \in V$, we define the *in-neighbors* of u, denoted by $N_{in}^G(u)$, to be the set of nodes $\{v|\text{there exits an edge } (v, u)\}$. Similarly, we define the *out-neighbors* of u, denoted by $N_{out}^G(u)$ to be the set of nodes $\{v|\text{there exits an edge } (u, v)\}$. Consider the node v_8 in Figure 1. Its in-neighbors $N_{in}^G(v_8)$ is $\{v_2, v_6, v_7\}$ and its out-neighbors $N_{out}^G(v_8)$ is $\{v_5, v_6\}$.

Given two points, namely s and t, in V, we simply call a path from s to t on $G(V, E)$ a *network path* from s to t denoted by $s \rightsquigarrow_G t$. The path $s \rightsquigarrow_G t$ consists of a sequence of nodes, each of which is in V. The subscript G denotes that all the edges along $s \rightsquigarrow_G t$ are from the spatial network G. The length of $s \rightsquigarrow_G t$, denoted by $w(s \rightsquigarrow_G t)$, is defined to be the sum of the weights of all edges involved in $s \rightsquigarrow_G t$.

Consider the path $v_4 \rightsquigarrow_G v_7 = (v_4, v_3, v_7)$ in Figure 1. Its weight $w(v_4 \rightsquigarrow_G v_7)$ is equal to $w_G(v_4, v_3) + w_G(v_3, v_7) = 2$.

Given two paths $s \rightsquigarrow_G t$ and $s' \rightsquigarrow_G t'$, where $t = s'$, we denote the *concatenation* of these two paths by $s \rightsquigarrow_G t \cdot s' \rightsquigarrow_G t'$. For example, $v_5 \rightsquigarrow_G v_4 = (v_5, v_4)$ and $v_4 \rightsquigarrow_G v_7 = (v_4, v_3, v_7)$ are two paths in Figure 1. The concatenation of these two paths, i.e., $v_5 \rightsquigarrow_G v_4 \cdot v_4 \rightsquigarrow_G v_7$, is (v_5, v_4, v_3, v_7).

Given two vertices, namely s and t, in V, we simply call the shortest path from s to t on $G(V, E)$ the *shortest network path* from s to t denoted by $s \rightarrowtail_G t$. For example, in Figure 1, $v_5 \rightsquigarrow_G v_7 = (v_5, v_4, v_3, v_7)$ is a path from v_5 to v_7 but it is not the shortest path from v_5 to v_7. The shortest path from v_5 to v_7, denoted by $v_5 \rightarrowtail_G v_7$, is (v_5, v_7). Besides, we let $d_G(s, t)$ denote the length of the shortest path $s \rightarrowtail_G t$ from s to t, i.e., $d_G(s, t) = w(s \rightarrowtail_G t)$. For example, the length of the path $v_8 \rightarrowtail_G v_3 = (v_8, v_5, v_4, v_3)$ in Figure 1, denoted by $d_G(v_8, v_3)$, is $1 + 1 + 1 = 3$. Note that the triangle inequality still holds under this distance function $d_G(\cdot, \cdot)$.

Problem 1 (Distance and Path Queries)**.** *Given a spatial network $G(V, E)$, and any two points in V, namely s and t, find an estimated value of $d_G(s, t)$ and the s-t shortest path $s \rightarrowtail_G t$ efficiently.*

3 Existing Work

In this section, we first present the related work about distance and path oracles on the static spatial networks in Section 3.1 - Section 3.10. The existing works in Section 3.1 and Section 3.2 proposed the algorithm for computing the shortest distance and shortest path on-the-fly. Whereas, the algorithms presented in Section 3.3 - Section 3.10 follow the preprocessing-query framework, where an indexing structure, namely *distance and path oracle*, is established to accelerate the query. After giving the algorithms on the static road networks, we then present the heuristic oracles on the dynamic spatial networks and on the dynamic graph in Section 3.11. We present the oracles on other types of networks in Section 3.12. We finally give the related works on some variants of the distance and path queries in Section 3.13. We omit the review on approximate oracles such as [41, 42] since the focus of this paper is on exact oracles. There are a large body of existing oracles on the static spatial networks. We cluster them into the following categories by their intuitions.

3.1 Dijkstra's Algorithm

A well-known algorithm for computing the shortest distance and path query on-the-fly is the Dijkstra's algorithm [16]. It was originally designed for the graph where each edge has a non-negative weight and thus could naturally be applied to the spatial networks. The Dijkstra's algorithm starts the search from the source node s and expands the search region from s. In the initialization phase of the algorithm, it set the distance from s to all other nodes to be ∞ and creates a min-heap for containing all the visited nodes whose key is the distance to s. Then, it puts s into the heap and the key of s is 0. Next, it conduct the search in several iterations. In each iteration, it extracts the root o of the heap. For each out-neighbor o' of o, it update the s-o' shortest distance (resp. the predecessor of o') to be $d_G(s, o) + w_G(o, o')$ (o) and then insert o' into the heap if $d_G(s, o') > d_G(s, o) + w_G(o, o')$. The iterative procedure ends when the destination t is extracted from the heap. Finally, it returns $d_G(s, t)$. To find the s-t shortest path, it find a sequence of nodes $(o_1, o_2, ..., o_l)$, where $o_1 = s$, $o_l = t$ and o_i is the predecessor of o_i and

i is a positive integer at most l. The time complexity of the Dijkstra's algorithm is $O(n \log n + m)$. The correctness of the Dijkstra's algorithm is based on the following observation: any subpath of the s-t shortest path must be a shortest path. Thus, in the iterative procedure, for each node o visited, we only need to maintain the shortest s-o distance and the predecessor of o on the s-o shortest path on the graph induced by the visited nodes. Note that in the algorithm, there could be duplicate nodes in the heap but they must have different keys.

Bi-Dijkstra's algorithm is an advanced version of Dijkstra's algorithm where the search is performed from both the source and the destination. Specifically, in the initialization, both $d_G(s, s)$ and $d_G(t, t)$ are set to be 0 and for each node o in $V \setminus \{s\}$ (resp. $V \setminus \{t\}$), $d_G(s, o)$ (resp. $d_G(o, t)$) is set to be ∞. The algorithm then creates two heap, namely H_s and H_t, for the forward and backward search, respectively and insert s (resp. t) into H_s (resp. H_t). Next, the algorithm performs an iterative procedure. It extracts the root from H_s and H_t alternatively in the iterations. In each Iteration, if the root of H_s is extracted, then it follows the same steps as Dijkstra's algorithm and otherwise, for each in-neighbor o' of the root o of H_t, if $d_G(o', t) > d_G(o, t) + w_G(o', o)$, then it update $d_G(o', t)$ to be $d_G(o, t) + w_G(o', o)$ and insert o' into H_t. It also maintain the current shortest distance $\min_{v \in V}(d_G(s, v) + d_G(v, t)$. The Bi-Dijkstra's algorithm terminates when the current shortest distance is smaller than the sum of the keys of $r(H_s)$ and $r(H_t)$, where $r(H_t)$ (resp. $r(H_s)$) is the root of H_s (resp. H_t).

3.2 A^* Search

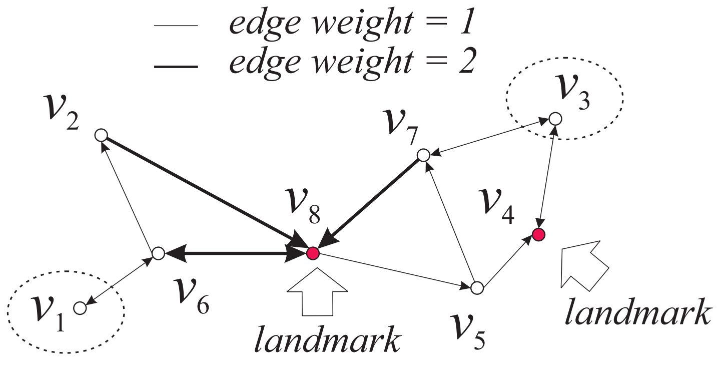
Dijkstra's algorithm visits all nodes in a circular manner from the source to the destination. However, it is possible to largely reduce the number of the visited nodes if the directional information regarding the source and destination could be provided. Specifically, the node with less distance to t should be propagated earlier than that with larger distance to t given that they have the same distance from s. Motivated by this observation, Hart, etc. [26] proposed a variation of Dijkstra's algorithm called 'A^* search'.

In this method, the key of a node u in the min-heap is an estimated lower bound of the length of the s-t shortest path passing through u. Mathematically speaking, $\forall u$ in the min-heap, the key of u in H is $d_G(s, u) + \underline{d_G}(u, t)$, $\underline{d_G}(\cdot, \cdot)$ denotes the lower bound of the shortest network distance $d_G(\cdot, \cdot)$. We will present later how to estimate $\underline{d_G}(\cdot, \cdot)$. The node with smaller estimated lower bound will be propagated earlier. In A^*, the key of a node u is the lower bound of the overall distance $d_G(s, u) + \underline{d_G}(u, t)$ while in Dijkstra's algorithm, the key of u is $d_G(s, u)$

which is the length of a partial path. Thus, A^* search provides tighter lower bound for the search and the bound contains the information of the destination t and it is a direction-aware search. This is the reason why A^* search is more efficient in practice. Similar to Dijkstra's algorithm, A^* search could have a bidirectional version where the search is conducted from both s and t.

Now, the only thing left is how to estimate $\underline{d_G}(\cdot,\cdot)$. Since the road network is a graph embedded in the Euclidean space, the Euclidean distance is a very natural selection. In the next section, we will present an A^* based algorithm which adopts more advanced lower bound.

3.3 ALT



$$\underline{d_G}(v_1, v_3)=\max\{d_G(v_1, v_8)\text{-}d_G(v_8, v_3), d_G(v_1, v_4)\text{-}d_G(v_4, v_3)\}$$
$$=d_G(v_1, v_3)\text{-}d_G(v_1, v_3)=6\text{-}1=5$$

Figure 2: An Example of ALT

[21] proposed an algorithm called $A^* + Landmarks + TriangleInequality$ (ALT in short). As could be noticed from its name, ALT follows the framework of A^* and computes the lower bound needed by using landmarks. In the pre-processing phase, it selects a set, denoted by L, of nodes to be the landmarks and precomputes the distances from each landmark to all nodes and the distances from all nodes to each landmark by the Dijkstra's shortest path algorithm. The cardinality k of L is given as a input. In the query phase, ALT adopts the precomputed distances as lower bound to reduce the search space. It simply estimates $\underline{d_G}(v, x)$), where v and x are two nodes in G, to be $\max_{o \in \mathrm{L}} |d_G(v, o) - d_G(o, x)|$. By the triangle inequality, the estimation must be correct. Consider the example as shown

in Figure 2. There are two landmarks v_4 and v_8. The lower bound of the distance from v_1 to v_3 is equal to $\max\{d_G(v_1, v_4) - d_G(v_4, v_3), d_G(v_1, v_8) - d_G(v_8, v_3)\} = 6 - 1 = 5$.

The key issue in ALT is the selection of the landmarks. [21] also proposed three heuristic-based landmark selection algorithm, namely *random landmark selection*, *farthest landmark selection* and *planar landmark selection*. The random landmark selection algorithm simply selects k nodes at random. The farthest landmark selection algorithms first initializes L to be $\emptyset$ and then works in k iterations, each of which selects a new node $\arg\max_{o \in V \setminus \mathrm{L}} \min_{o' \in \mathrm{L}} d_G(o, o')$ (i.e. the node in $V \setminus \mathrm{L}$ with the largest distance to L, where the distance between a node and a set of node is the minimum distance between the node and any node in the set). This selection algorithm tries to maximize the minimum pairwise distance of the landmarks. The third one, the planar landmark selection algorithm, first finds a vertex c closest to the geometric center of all nodes. Then, it divides the x-y plane into k pie-slice sectors centered at c, each of which contains roughly the same number of nodes. Finally, it selects a node from each sector which is farthest from c. The above three algorithms all have fairly good performance in practice.

3.4 Reach

Gutman, et al. [25] proposed an oracle called *REACH*. This technique is based on a concept called *Reach* of a vertex on a graph. Let me explain this concept as follows. Consider a vertex v on G and a shortest path $P = s \rightarrowtail_G t$ passing through v. The reach of v w.r.t. P, denoted by $r_P(v)$, is defined to be $\min\{d_G(s, v), d_G(v, t)\}$. The reach of v is defined to be $\max\{r_P(v) | P$ is a shortest path and P passes through $v\}$.

Based on the reach of each vertex, Gutman, et al. proposed a pruning method for the Dijkstra's algorithm. Consider a vertex u and one of its neighbor v which is visited by the Dijkstra's algorithm. u will be pruned if $r(u) < \min\{d_G(s, v) + w_G(v, u), \underline{d_G}(u, t)\}$, where $\underline{d_G}(u, t)\}$ is a lower bound of $d_G(u, t)\}$. In [25], the landmarks are utilized to measure the lower bound and several algorithms are proposed to compute the upper bound of the reach of each vertex since computing the exact reach of each vertex is costly. Goldberg et al. [22] then proposed several methods for accelerating the computing of the upper bound of the reach of each vertex.

3.5 Partition Techniques

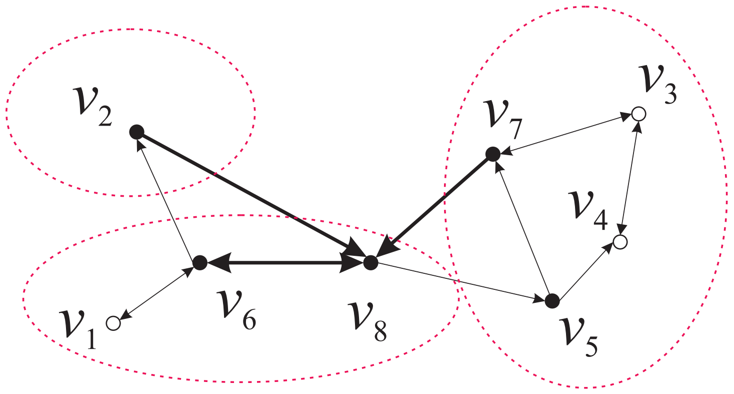


Figure 3: An Example of Partition

Hiti [29, 30] partitions the network into disjoint parts and precomputes the shortest distances between the boundary nodes of all parts to facilitate the query. Specifically, in the query phase, it invokes Dijkstra's algorithm but only the component containing the source and the component containing the destination involved. Note that if the source and the destination are in the same component, then only the component containing them will be involved. Thus, this technique will largely reduce the number of nodes visited by Dijkstra's algorithm and meanwhile preserve the correctness of the query processing due to the precomputation of the distances between the boundary nodes. Consider the example as shown in Figure 3. The network is partitioned into 3 components and they contain $\{v_2\}$, $\{v_1, v_6, v_8\}$ and $\{v_3, v_4, v_5, v_7\}$, respectively. Each black node is a boundary node. The distances between the boundary (i.e., black) nodes are precomputed. Consider the shortest distance query between v_1 and v_6. Since they are in the same component, only the nodes in the second component will be involved in the Dijkstra's algorithm (i.e., v_1, v_6 and v_8). Consider the shortest distance query between v_1 and v_2. Then, the query algorithm will only visits v_1, v_2, v_6 and v_8. The mostly used partition algorithm is *MERIT* [32] which partitions the graph into k disjoint components where k is a user-specified parameter. Note that each component has roughly the same number of nodes, i.e., the partition is 'balanced'. Besides, the number of border nodes is very small, i.e., the partition is 'light'.

3.6 Spatial Coherence

Samet et al. [38] proposed *Spatially Induced Linkage Cognizance* (SILC). SILC first indexes all vertices by a quadtree. It stores a shortest path map for each vertex v in the spatial network. The map partition all vertices by their spatial coherence, specifically, the first edge of their shortest path from v. In the shortest path map of v, each partition contains a set of disjoint quads instead of vertices and thus, much space could be saved. Given a source s and a destination t, it is easy to identify the first edge on the shortest s-t path though the shortest path map of s. Figure 4 illustrates the SILC built on the road network as shown in Figure 1. First, the 8 vertices are indexed in a quadtree. Then, consider the shortest path map of v_8. In the shortest path tree T rooted at v_8, v_8 has two children. One is v_6 and the other is v_5. Thus, the nodes in $V \setminus \{v_8\}$ are partitioned into two components. One is $\{v_1, v_2, v_6\}$ which contains the nodes on the sub-tree of T rooted at v_6 and it is represented by two disjoints quads (i.e., the two dark quads) as shown in Figure 4 in the shortest path map of v_8. The other component is $\{v_3, v_4, v_5, v_7\}$ which contains the nodes on the sub-tree of T rooted at v_5 and it is represented by two disjoints quads (i.e., the two gray quads) as shown in Figure 4 in the shortest path map of v_8.

In the query phase, SILC finds the sequence of edges in the shortest path through the map of all vertices on the path. Consider the shortest path query from v_8 to v_1 in the example as shown in Figure 4. The query algorithm first searches the shortest path map of v_8 and finds that the node v_1 is contained in a dark quad and this means that the first edge of the shortest path from v_8 to v_1 is (v_8, v_6). Thus, the first edge on the shortest path is obtained and then, the query algorithm search the shortest path map of v_6 and so on so forth.

The paper proved that the space complexity is $O(n^{1.5})$ which is significantly smaller than that of the materialization of all pairwise shortest paths (i.e., $O(n^3)$).

Jagan, et al. [43] proposed an other oracle called *Path Coherent Pairs Decomposition* (PCPD in short). This oracle contains a set of triplets, each of which is in the format $< X, Y, \psi >$, where X and Y are two disjoint quads and Ψ is a vertex or edge. In each triplet $< X, Y, \Psi >$ of PCPD, for any vertex $a \in X$ and any vertex $b \in Y$, the shortest path from a to b and the shortest path from b to a passes through Ψ. Figure 5 shows an example of a triplet $< X, Y, \Psi >$ in PCPD. In the example, the quad X contains v_3 and v_7 and the quad Y contains v_1 and v_6 and Ψ is v_8. It is very easy to see that the shortest path from a vertex in X to a vertex in Y must pass through Ψ. The paper [43] proved that for any two vertices s and t in G, there exists exactly one triplet $< X, Y, \Psi >$ in PCPD that s is in X and t is

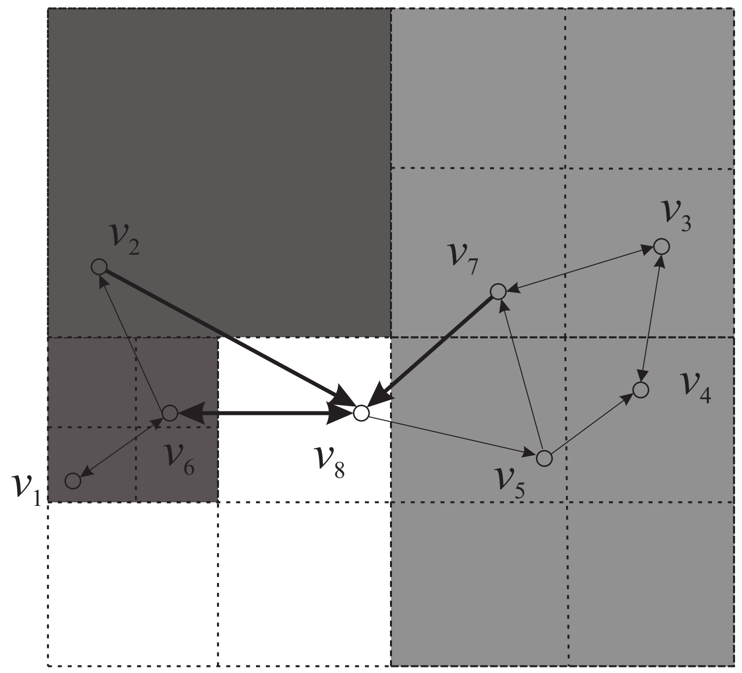


Figure 4: An Example of SILC

in Y.

In the query phase, given two vertices s and t, PCPD finds the shortest s-t path as follows in two steps.

Step 1: It first finds the triplet $< X, Y, \Psi >$ that s is in X and t is in Y. If Ψ is a vertex and (s, Ψ, t) is a path, it returns (s, Ψ, t) and terminate the algorithm. If Ψ is an edge (i.e., (u, v)) and (s, u, v, t) is a path, it returns (s, u, v, t) and terminate the algorithm.

Step 2: If Ψ is a vertex, it recursively finds the shortest path from s to Ψ and the shortest path from Ψ to t and concatenates them. If Ψ is an edge (i.e., (u, v)), it recursively finds the shortest path from s to u and the shortest path from v to t and concatenates them.

In the example as shown in Figure 5, consider the shortest path query from v_3 to v_1. In Step 1, it first find the triplet $< X, Y, v_8 >$. Since (v_3, v_8, v_1) is not a path, it performs Step 2 where it first recursively invokes the shortest path query from v_3 to v_8 and the shortest path query from v_8 to v_1 and concatenates the two paths.

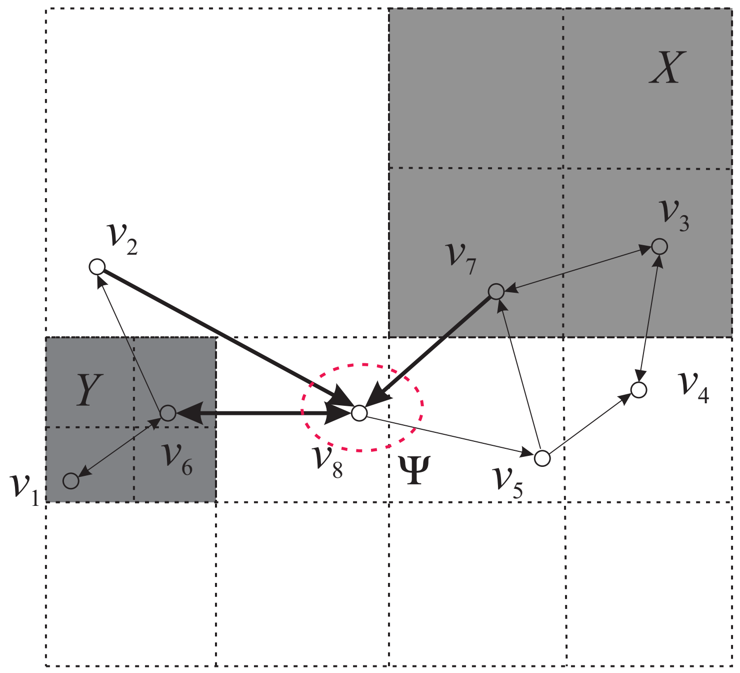


Figure 5: An Example of PCPD

3.7 Transit Node Routing

Bast et al. [10, 11] observed that there are a small number of nodes, namely *Transit Nodes*, such that any long-range shortest path passes one of them. Bast et al. developed a heuristic query algorithm based on this observation. However, [47] showed that the solution [10] may not return the correct answer.

3.8 2-Hop Labeling Techniques

Cohen et al. [14] first proposed the *2-Hop Labeling* (2HL) for the distance query on graphs. It preprocesses a set $\mathcal{L}(u)$ of labels for each node u. In the query phase, the shortest path from the source s and the destination t could be computed by searching $\mathcal{L}(s)$ and $\mathcal{L}(t)$. Ittai and Denial, etc. [2], studied a variant of 2-Hop Labeling, namely *Hierarchical Hub Labeling* (HHL), for the shortest path query on graphs. The best theoretical bounds on the preprocessing time of *2-Hop Labeling* is shown to be $\Omega(n^4)$ [14] which is impractical even for a small network. [14] also showed that the space of any *2-Hop Labeling* is lower-bounded by $\Omega(n\sqrt{m})$. Jin et al. [28] proposed a *Highway Centric Labeling* (HCL) which is more efficient than *2-Hop Labeling* on large sparse graph. The preprocessing time (resp. space) is $O(nm)$ (resp. $O(n^2)$).

3.9 Hierarchy Technique

[19] proposed *Contraction Hierarchy* (CH) Intuitively, the technique is to transform the network by adding a set of assistant edges, namely *shortcuts*, into the original network $G(V, E)$. The shortcuts bridges distant nodes and thus accelerates the query processing.

We proceed to present the details of Contraction Hierarchy (CH in short). We begin with several concepts as the preliminary.

Given a permutation P of the n nodes in the spatial network $G(V, E)$ and a node v, where $v \in V$, we call the position of v in P the rank of v, denoted by $R_P(v)$.

Consider the 7 vertices as shown in Figure 1 and a permutation $I = (v_1, v_2, v_3, v_4, v_5, v_6, v_7)$. The rank $R_I(v_1)$ of v_1 is equal to 1. The rank $R_I(v_2)$ of v_2 is equal to 2 and so on so forth.

Given a permutation P of the n nodes in a spatial network $G(V, E)$ and a path $u \rightsquigarrow_G v$, where u and v are two nodes in G, $u \rightsquigarrow_G v$ is called a *trivial path* if it has only one edge or each node x on $u \rightsquigarrow_G v$ excluding v and u has a smaller rank than both u and v in P (i.e., $R_P(x) < \min\{R_P(u), R_P(v)\}$). Given a path on a network, the path is called a *shortest trivial path* if it is a shortest path and it is also a trivial path.

For example, in Figure 1, both (v_5, v_4, v_3, v_7) and (v_5, v_7) are trivial paths because (v_5, v_7) has only one edge and on (v_5, v_4, v_3, v_7), the nodes v_4 and v_3 have smaller ranks than v_5 and v_7. (v_7, v_8, v_5) is not a trivial path because it has more than 1 edge and v_8 has a higher rank (i.e., 8) than $\min\{R_I(v_5), R_I(v_7)\} = 5$. (v_5, v_4, v_3, v_7) and (v_7, v_8, v_5)) are not shortest trivial paths since (v_5, v_4, v_3, v_7) (resp. (v_7, v_8, v_5)) is not a shortest path (resp. trivial path). But (v_5, v_7) is a shortest trivial path since it is a shortest path and a trivial path.

CH [19, 20] first imposes a permutation of all nodes and then contract each node in the ascending order of their ranks. It maintains a transformed network which is initially equal to G. The contraction of each node v adds shortcuts into the transformed network. A shortcut is an edge in the transformed network G' which is not in the original network G. We denote a shortcut whose origin is u and destination is v by $\langle u, v\rangle$.

Given a permutation of all nodes, we call a contraction operation, in the procedure of which some shortcuts are created, a *distance-preserving contraction* if it has three properties: (1) it contracts all nodes in the ascending order of their ranks. (2) in the final transformed network, the weight of any shortcut $\langle x, y\rangle$ is equal to $d_G(x, y)$. (3) there must be a shortcut from x to y whose weight is $d_G(x, y)$ if and only if there is a trivial shortest path from x to y on the original network G. The contraction of CH is a distance-preserving contraction which we will show in details later.

Given a permutation of all nodes in a network, a path is called an "ascending path" (resp. a "descending path") if each node excluding the first node (resp. the last node) has a higher (resp. smaller) rank than its predecessor. Then, we present an important property.

Observation 1 (ascending-descending property)**.** *Consider any two nodes s and t and the nodes o on the $s \rightarrowtail_G t$ with the highest rank and a transformed network G' obtained by a distance-preserving contraction, there exits one s-o (resp. o-t) shortest path on the transformed network G' which is an 'ascending path' (resp. a 'descending path') if s (resp. t) is not o.*

Proof. Since all nodes are contracted by a distance-preserving contraction, there must be a shortcut from u to v whose weight is $d_G(u, v)$ if the nodes on the $u \rightarrowtail_G v$ excluding u and v have lower ranks than u and v. We proceed to justify that there exists an s-o shortest path on G' which is an ascending path if s is not o. Consider a sequence $(o_0, o_1, o_2, ..., o_m)$ of nodes on the $s \rightarrowtail_G o$, where $o_0 = s$, $o_m = o$ and $o_i (i \in [1, m-1])$ is the node with the highest rank on the s-o_{i+1} shortest path on G. Then, there must exist a shortcut $\langle o_i, o_{i+1}\rangle$ on G' whose weight is $d_G(o_i, o_{i+1})$ due to the property (3) of the distance-preserving contraction. Thus, there exists an s-o shortest path on G' must be an ascending path if s is not o. Similarly, we could prove that there exists an o-t shortest path on G' which is a descending path if t is not o. $\square$

By this property, the query processing is accelerated by imposing a rank-based pruning method on the bidirectional Dijkstra's algorithm which we will show later in detail (Roughly speaking, in both the forward and backward search, each node only needs to expand the adjacent nodes with higher order).

Next, we present the contraction procedure first and then in the end of this section, we discuss the ordering of all nodes. We begin with a concept.

Given an edge (u, v), we denote by $\mathrm{o}(u, v)$ the node with smaller rank than u and v such that in the final transformed network G', $w_{G'}(u, \mathrm{o}(u, v)) + w_{G'}(\mathrm{o}(u, v), v)$ is the smallest.

Algorithm 1: BuildCH(V, E)

```
Data: A spatial network $G(V, E)$
Result: A transformed spatial network $G'(V', E')$
1  // Step 1: Initialization;
2  $E' \leftarrow E, V' \leftarrow V$;
3  Initialize a temporary network $G_t(V_t, E_t)$;
4  $E_t \leftarrow E, V_t \leftarrow V$;
5  // Step 2: Random Permutation;
6  Impose a permutation $I$ of all nodes in $V$;
7  // Step 3: Contraction;
8  while $I$ is not empty do
9      Extract the first node $u$ in $I$;
10     for each node $v \in N_{in}^{G_t}(u)$ do
11         for each node $x \in N_{out}^{G_t}(u)$ do
12             if There is no such an edge $(v, x)$ in $G_t$ then
13                 Find $d_G(v, x)$ by Dijkstra's algorithm;
14                 if $d_G(v, x) = w_{G_t}(v, u) + w_{G_t}(u, x)$ then
15                     Create an edge (shortcut) $\langle v, x \rangle$ in $G'$ and $G_t$;
16                     $w_{tmp} \leftarrow w_{G_t}(v, u) + w_{G_t}(u, x)$;
17                     $\mathrm{o}(v, x) \leftarrow u$;
18                     $w_{G_t}(v, x) \leftarrow w_{tmp}$.;
19                     $w_{G'}(v, x) \leftarrow w_{tmp}$.;
20     remove $u$ and its incident edges from $G_t$;
21 // Step 4: Output;
22 return $G'(V', E')$.
```

Algorithm 2: RealPath($\langle u, v\rangle$)

```
    Data: edge $\langle u, v\rangle$ in the transfromed network $G'$
    Result: The real path of $\langle u, v\rangle$ in the original network $G$
1   if there is an $\langle u, v\rangle$ in $G$ and $w_G(u, v) = w_{G'}(u, v)$ then
2       return $(u, v)$;
3   return RealPath($u, \mathrm{o}\langle u, v\rangle$)·RealPath($\mathrm{o}\langle u, v\rangle, v$).
```

Algorithm 1 shows the pseudocode of the contraction algorithm. In Algorithm 1, the input is the original network $G(V, E)$ and the output is the transformed network $G'(V', E')$. We first initialize G' to be G (Line 1-2). We also maintain a temporary network $G_t(V_t, E_t)$ which is initialized to be G (Line 3-4). It imposes a permutation I of all the nodes (Line 5-6). The loop contracts the nodes one by one following the order of I (Line 7-21). Consider the contraction of the node u. For any two nodes v and x, where $v \in N_{in}^{G_t}(u)$ and $x \in N_{out}^{G_t}(u)$, we perform the following operations (line 10-11). If there is no such an edge $\langle u, x\rangle$ in G_t and (v, u, x) a trivial shortest path from u, CH simply creates an edge (i.e., shortcut) $\langle u, x\rangle$ in G_t and G'(Line 12-15). Then, we assign $\mathrm{o}(v, x)$ to be u and assign $w_{G_t}(v, x)$ and $w_{G'}(v, x)$ to be w_{tmp} (Line 15-18). Note that the edge (v, u) (resp. (u, x)) could be either an edge copied from E or a shortcut. Besides, the path (v, u, w) must be a trivial path, since we contract the nodes one by one in the order of the permutation. The final step of the outer loop is to remove the node u and its incident edges from G_t (Line 19). It is obvious that our contraction method is a distance-preserving contraction.

We proceed to illustrate an example of the preprocessing algorithm. Consider a random permutation $I = (v_1, v_2, ..., v_7)$ of the seven objects as shown in Figure 1. Initially, G_t is a copy of the original network G. Figure 6(a) - (e) shows the contractions of the first five objects and the temporary network G_t in each step, and the contractions of v_6, v_7 and v_8 are trivial and thus they are shown in the figure. In the first step as shown in Figure 6(a), v_1 is contracted. Since $N_{in}^{G_t}(v_1) = \emptyset$, in this step, we simply remove v_1 and its incident edges. In Figure 6(b), the node v_2 is contracted, in the procedure of which, we found $N_{in}^{G_t}(v_2) = \{v_6\}$ and $N_{out}^{G_t}(v_2) = \{v_8\}$. Since (v_6, v_8) is an edge in G, we update the weight of (v_6, v_8) in G_t and G' to be $\min\{w_G(v_6, v_8), w_{G_t}(v_6, v_2) + w_{G_t}(v_2, v_8)\} = 2$. Then, we update $\mathrm{o}(v_6, v_8)$ to be v_2. In Figure 6(c), the node v_3 is contracted. Note that $N_{in}^{G_t} = N_{out}^{G_t} = \{v_4, v_7\}$. Since there are no edges (v_4, v_7) and (v_7, v_4) in G_t, we create the shortcut $\langle v_4, v_7\rangle$ (resp. $\langle v_7, v_4\rangle$) in G' and G_t

with the weight equal to $\min\{w_G(v_4, v_7), w_{G_t}(v_4, v_3) + w_{G_t}(v_3, v_7)\} = 2$ (resp. $\min\{w_G(v_7, v_4), w_{G_t}(v_7, v_3)+w_{G_t}(v_3, v_4)\} = 2$). Then, we update $\mathrm{o}(v_4, v_7)$ (resp. $\mathrm{o}(v_7, v_4)$) to be v_3 (resp. v_3). The contractions of v_4 and v_5 are similar.

The final transformed network G' is shown in Figure 6(f). Figure 7(a) shows a shortest path from v_4 to v_1 in the original network. The shortest path is $(v_4, v_3, v_7, v_8, v_6, v_1)$. As could be noticed, there is a shortest cut $\langle v_4, v_7\rangle$ in the transformed network whose weight is equal to $w_G(v_4, v_3) + w_G(v_3, v_7)$. Thus, there is a shortest path from v_4 to v_3 in the transformed network (i.e., $(v_4, v_7, v_8, v_6, v_1)$). It consists of an ascending path (i.e., (v_4, v_7, v_8)) and a descending path (i.e., (v_8, v_6, v_1)) as Figure 7(b) shows.

Now, the remaining issue in the preprocessing is the ordering of all nodes. There are exponential orders of all nodes. It is obvious that the number of the shortcuts in the final transformed network is highly related to the order. However, the problem of finding the order with the minimum number of shortcuts is NP-hard[34]. Motivated by this, several heuristic-based orders were proposed and the most popular one is called *edge difference* which has the best empirical performance.

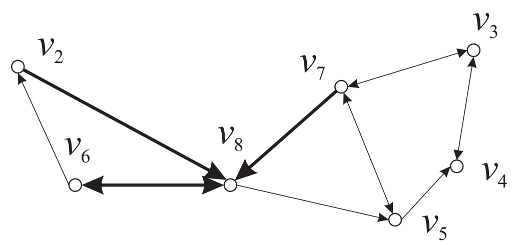
For the distance between a given source s and destination t, we perform a bidirectional Dijkstra's shortest path algorithm in the graph $G'(V', E')$. Specifically, the algorithm performs a forward (resp. backward) Dijkstra's algorithm from s (resp. t). The forward (resp. backward) search picks one visited node in each iteration by their priorities (which we will elaborate later) and expands its out-neighbors (resp. in-neighbors) and update their distances from s (resp. to t). The bidirectional search is enhanced with the following rule.

Rank Constraint. We call an edge (u, v) a *Forward Constrained Edge* (resp. *Backward Constrained Edge*) if u is a visited node and $R_I(v) > R_I(u)$ (resp. $R_I(v) < R_I(u)$). The forward (resp. backward) Dijkstra's algorithm only relaxes Forward (resp. Backward) Constrained Edges.

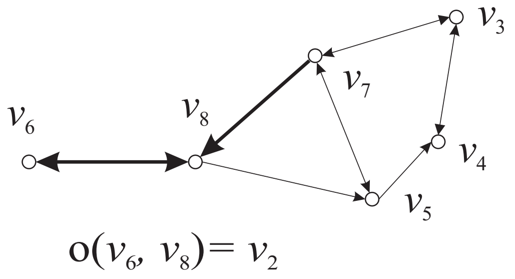
The above query algorithm finds an s-t shortest path on the transformed network G' and thus the shortest distance is obtained. However, for the shortest path query, we need to find the correspond path in G.

As we could notice, every edge in the transformed network G' has a corresponding path in the original network G. Algorithm 2 shows the pseudocode of an algorithm called *RealPath* for computing the corresponding path of a given edge $\langle u, v\rangle$ in G'. We call the output of the RealPath$\langle u, v\rangle$ function the *real path* of a shortcut $\langle u, v\rangle$. Consider the shortcut $\langle v_4, v_7\rangle$ as shown in Figure 6(f). The real path of $\langle v_4, v_7\rangle$ is $\langle v_4, v_3\rangle \cdot \langle v_3, v_7\rangle$.

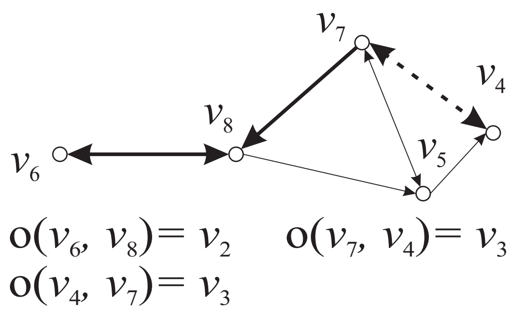
— *edge weight = 1* **—** *edge weight = 2* - - - *edge weight = 2* (*shortcut*)



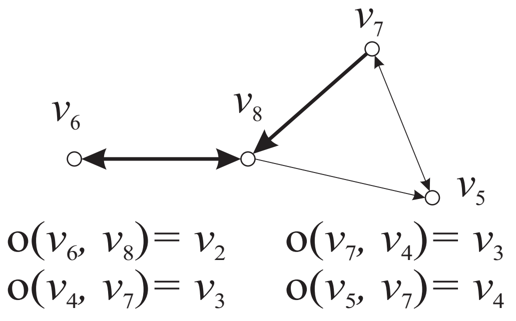
(a) Step 1 of Preprocessing



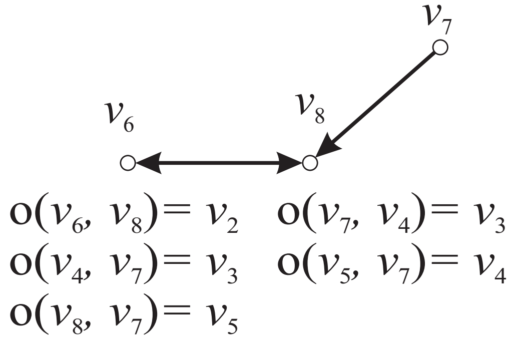
(b) Step 2 of Preprocessing



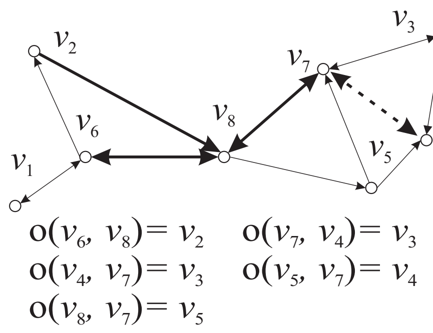
(c) Step 3 of Preprocessing



(d) Step 4 of Preprocessing



(e) Step 5 of Preprocessing



(f) Illustration of Transformed Network $G'(V', E')$

Figure 6: Illustration of Preprocessing of CH

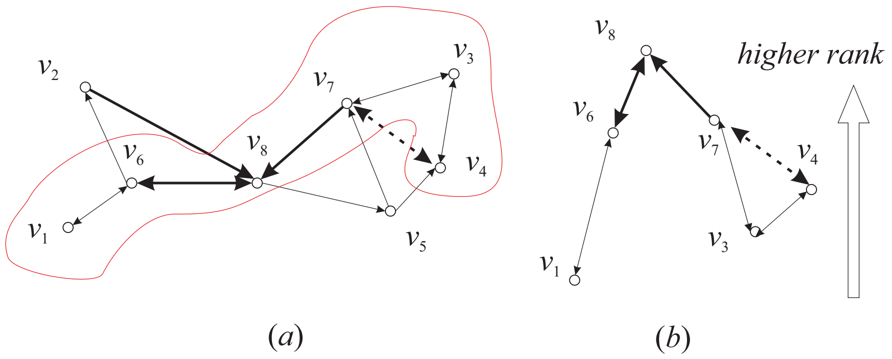


Figure 7: An Example of Ascending-Descending Property of CH

3.10 Hybrid

Transit Node Routing + Hierarchy Technique. Abraham et al. [3] formalized the observation of Bast et al. [10, 11] and derived an oracle, namely *Transit Node Variant* (TNV), with worst-case bounds. It has the same preprocessing algorithm as CH except that it imposes a different order of all vertices. We present the permutation generation next. In the construction phase, TNV first materializes all pairwise shortest path and then sorted them in the descending order of their length. Let l_{max} (resp. l_{min}) denote the maximum (resp. minimum) pairwise shortest distance. Then, it puts all the shortest path into k different groups. The group i contains all shortest paths whose lengths are in the range $[\frac{l_{max}}{2^i}, \frac{l_{max}}{2^{i-1}}]$, where i is a positive integer. Note that the shortest path with the minimum length must be in the group k, i.e., $l_{min} \in [\frac{l_{max}}{2^k}, \frac{l_{max}}{2^{k-1}}]$. Note that by the above procedure, we obtain that k is $O(\log \alpha)$, where α is defined to be $\frac{l_{max}}{l_{min}}$. For each i in $\{1, 2, ..., k\}$, TNV finds a set H_i of vertices such that any shortest path in group i contains one vertex in H_i. We will present the details of the selection algorithm later in details. Then, TNV imposes a partial oder on all vertices. For each vertex contained in some selected set, its order is assigned to be the minimum integer j such that H_j contains the vertex. For each vertex that is not selected, its order is $k + 1$. Next, TNV converts the partial order into a total order. After the conversion, a node u has a lower order than a node v in the total order if u has a lower order than v in the partial order. For all nodes with the same order in the partial order will be ordered randomly in the total order. The remaining steps in the construction is the contraction of all nodes in the descending order and the contraction of each node is the same as that of CH.

Now, we delve into the details of the vertices finding problem left above. Ideally, the set of vertices selected from each group should be as small as possible. However, finding the smallest set of vertices such that each path in a given group of paths passes through one of them is NP-hard. Thus, TNV simply find $O(\log n)$-approximate solution by greedy algorithm.

The query algorithm is a Bi-Dijkstra's algorithm enhanced with two constraints. One is the Rank Constraint which is the same as that of CH. The other is Range Constraint. Specifically, in the Bi-Dijkstra's algorithm, an edge (or shortcut), denoted by (u, v), is expanded only if the weight of the edge is smaller than $\frac{l_{max}}{2^x}$, where x is the smaller order of v and u in the partial order.

The paper introduced a key concept, namely *highway dimension*, which is defined to be the minimum positive real value h such that for each i in $\{1, 2, ..., k\}$, there exits a set of vertices with cardinality at most h and any path in the group i passes through one vertex in it. Equipped with this concept, the paper derived theoretical bounds on TNV which is shown in Table 2. Although this oracle is theoretically decent, an all-pairwise shortest path query is needed to build such an oracle which is inefficient and precludes its usage in practice.

Zhu et al. [48] proposed an oracle, namely *Arterial Hierarchy* (AH), which is also based on the observation of Bast et al. [10, 11]. It is a hybrid of *Hierarchy Technique* and *Transit Node Routing.* Note that its preprocessing time is $O(n^2 \log n)$. AH [48] has similar theoretical results to that of TNV [3]. This oracle does not require the materialization of all-pairwise shortest paths and thus are deployable in practice. The only differences of AH and TNV are two issues: 1. the partial order generated in the construction phase 2. the Range Constraint in the query phase. Now, we are ready to delve into the details of the partial order in AH. As the preliminaries, we present several concepts first. Given a network and a 4x4 grids imposed on the network, the strip containing the 4 upper (resp. lower) cells is called *North Strip* (resp. *South Strip*). Similarly, the strip containing the 4 right (resp. left) cells is called *East Strip* (resp. *West Strip*). Figure 8 illustrates the 4x4 grids imposed on the network as shown in Figure 1 and the four strips of the grids.

Given a network and a 4x4 grids, an edge is called an *Arterial Edge* if one of the following statements is true: 1. The edge intersects with the vertical bisector of the grids and there is a shortest path passing through it whose endpoints are in different sides of the vertical bisector but are not in the cells adjacent to the vertical bisector. 2. The edge intersects with the horizontal bisector of the grids and there is a shortest path passing through it whose endpoints are in different sides of the horizontal bisector but are not in the cells adjacent to the horizontal bisector. Consider the example as shown in Figure 8. (v_8, v_5) and (v_7, v_8) are

Arterial Edges since they intersects with the vertical bisector and v_3-v_1 (resp. v_2-v_4) shortest path passes through (v_8, v_5) (resp. (v_7, v_8)).

AH first impose 4x4, 8x8, 16x16,...... boundary-aligned grids on the road network. In the last grid, each cell contains at most one point and in the second last grid, there exists one cells containing more than 1 point. Suppose that there are k grids. Next, AH inspects each 4x4 sub-grid in each grid imposed and find all the Arterial Edges of them. We call an edge a level-i edge if it is an Arterial Edge of a sub-grid of the ith grid, where i is a positive integer at most k. Then, the order of each node is assigned to be the smallest i such that there exists a level i edge incident to it. We finished the partial order of AH and next, we present the range constraint adopted in the query phase. In the Bi-Dijkstra's algorithm, an edge (or shortcut), denoted by (u, v), is expanded only if there exists a 3x3 sub-grid of the ith grid containing both u and v , where i is the smaller order of v and u in the partial order. The remaining parts of AH are exactly the same as TNV.

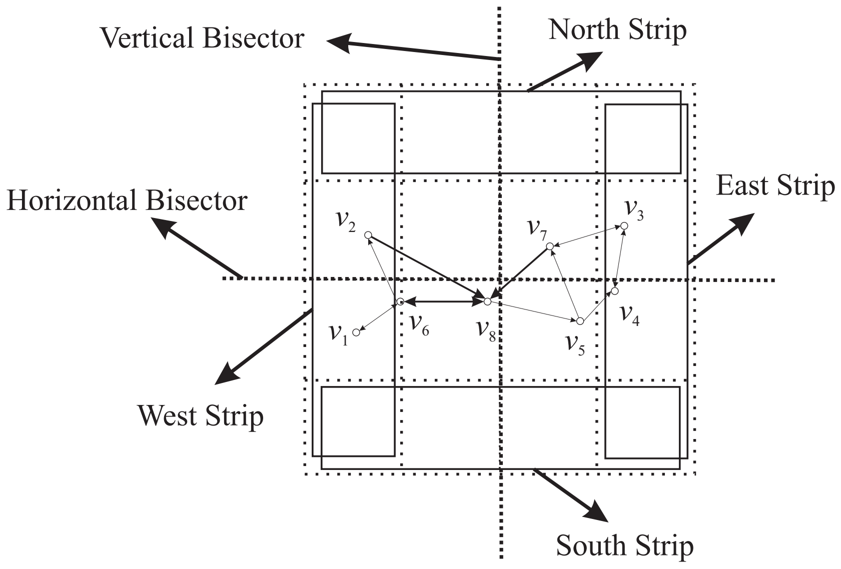


Figure 8: An Example of Arterial Hierarchy

2-Hop Labeling + Tree-Decomposition. Takuya et al. [6] proposed *Pruned Landmark Labeling* (PLL) which enhances the *2-Hop Labeling* with some pruning techniques. It is a hybrid of Tree-Decomposition, and 2-Hop Labeling. As shown in the empirical study of [6], PLL significantly outperform 2-Hop Labeling techniques [14, 2, 28] and HHL [2] in terms of pre-computation, space and query efficiency in theory and in practice. In a grid network which is common in Manhattan, ω is as large as $\sqrt{n}$ [1]. The experiments of [5] also verified that PLL is not scalable and much worse than CH in terms of building time and space overhead on the spatial networks. [4] extended *Pruned Landmark Labeling* to Top-k distance query.

Remark. Among of these techniques, some [19, 30, 21, 10, 39, 40, 2] focus on practical performance, and they are mostly heuristic-based. Thus, their worst case shortest distance or path query time is the same as that of Dijkstra's shortest path algorithm. Others [38, 48, 6, 7, 20, 3] have theoretical improvements on the query time, and Table 2 shows the complexities of these techniques. Note that those of the 2-hop labeling techniques are not shown since they are significantly outperformed by the PLL oracle [6]. As stated in [47], *CH* is the one with the best practical query efficiency which incurring the least space and precomputation overhead. The oracles with theoretical bounds as shown in Table 2 all have a larger-than $O(n^2)$ preprocessing time which is not scalable. Besides, the above oracles apply to static networks only which prevents their usage in many applications which are based on the dynamic spatial networks.

3.11 Oracles on Dynamic Spatial Networks and Dynamic Graph

[44] presented a dynamic routing oracle based on *Highway Hierarchy* (HH) [39, 40] which is a path oracle on the static networks. It also studied the dynamic version of the heuristics except CH on the static spatial networks and illustrated in the experiments that the dynamic HH has the best empirical performance. Similarly, [20] extended CH [19] to the dynamic spatial networks and their experiments showed that dynamic CH outperforms dynamic HH. Experiments showed that they are efficient in practice. However, their theoretical bounds are not attractive. Many papers [34, 13, 18] tried to give an analysis to their preprocessing time and query time whereas the best analysis [13, 18] of their preprocessing time (resp. query time) is $O(n^2 \log n)$(resp. $O(\sqrt{n} \log n)$) which is not scalable. As stated in [48], CH [19] has an $O(n^2)$ space. Besides, there is no analysis on the update time. In other words, although they perform well in practice, there is no analysis showing why they are efficient and which property of the road networks makes them efficient. [7] extended PLL [6] to the dynamic networks. However, [7] only supports vertex insertion and edge insertion operation and its theoretical bound on the update time is $O(n^2 \log^3 n)$ which is impractical on spatial networks.

3.12 Oracles on Other Types of Graphs

In this section, we survey the oracles on the other types of graphs. They all have assumptions on the property of the graphs which are obviously not suitable for the spatial networks. We divide them to the following types.

Small Tree-Width Graph. Two papers proposed oracles, namely *TEDI* [46] and *TD* [8] based on the tree-decomposition. This technique performs well on the graph with small tree width [1], denoted by ω. However, it is shown that in a square grid spatial network which is common in Manhattan district, the tree width is as large as $\sqrt{n}$ [1]. As a result, the above two oracles has an $O(n^2)$ (resp. $O(n)$) preprocessing time (resp. distance query time) which is non-scalable. As shown in the experiment of [6], they are significantly outperformed by PLL.

Planar Graph. There are several works [23, 33, 35] which studied the distance and path query on the planar graphs. Among them, [23] only applies to the network with integer weights. However, there are strong empirical evidence [17, 9] illustrating that the spatial networks are quite non-planar, and therefore, these techniques cannot be applied on spatial networks.

Scale-Free Graph. We call a graph scale-free if the distribution of its vertex degree follows the power-law, i.e., $Prob(\text{a vertex has degree } k) \propto k^{-\beta}$, where β is a positive real number and typically, $2 \leq \beta \leq 3$. Intuitively, this means that there are a small number of nodes with very high degree. Jiang et al. [27] proposed a structure for the distance query on Scale-free graph. However, as stated in [27, 17], the vertices in the spatial network all have very small degree (typically 2-4), since in practice, a very small number of road segments intersects at a junction due to the space limit. Thus, the distribution of the vertex degree in a spatial network is nearly a uniform distribution. Besides, the oracle proposed in [27] is disk-based and has a worse performance than PLL when the data could be fit in memory.

3.13 Variants of Path Queries on Spatial Networks

Rice and Isotras [37] proposed a novel model of a road network where each edge has a label. They studied the problem of finding a shortest path with the label constraint and developed an index based on the CH. Wang et al. [45] study the shortest path problem on the public transportation network where each edge has a timetable showing its available time window. They developed a distance and path oracle based on the HHL. Thus, the normal spatial network is a special case of the network they studied and their oracles reduce to CH and HHL on the normal spatial networks. There are existing works [31, 12, 15] on the distance and path queries on the time-dependent road networks. They all assume that the changes of the edge weights are periodical and are represented as a function of time in their data model. However, this assumption is not realistic in many scenarios in practice where the changes of the weights are caused by traffic jams, bad weathers and so on. In these applications, the changes are neither periodical nor predictable and thus the edge weight could not be formulated as a function of time or time series.

Comparison of related works. Among all the surveyed works, Dijkstra's algorithm, Bi-Dijkstra's algorithm and A^* compute the shortest distance/path on-the-fly. ALT and REACH has some preprocessing but they all have an $O(n \log n)$ running time. The remaining oracle have improvement in the worst case query time. We summarize them and highlight their results in Table 2. As shown in the table, [48, 42, 3, 19, 14] all have a worse-than $O(n^2)$ building time. By the result of [47] and [48], the State-of-Art are PLL, CH and AH. PLL has the smallest empirical distance query time and the smallest empirical path query time. AH has the best theoretical result. Specifically, compared with others in Table 2, it has a comparable preprocessing time ($O(n^2)$), a smaller space ($O(n)$), a smaller distance query time ($O(1)$) and a smaller path query time ($O(l)$). (Note that we ignore the very small parameters for better clarity). Besides, in the empirical side, it has the second smallest distance query time and the second smallest path query time and the second smallest preprocessing time and space among the existing oracles. CH has the best overall empirical performance. It has the smallest preprocessing time and the smallest space and the third smallest distance query time and the third smallest path query time. Although its query time is not the best, its query time for both distance query and path query is less than 900ms in a network with more than 14M vertices.

4 Conclusion

In this paper, we surveyed the existing works on the shortest distance/path queries on the spatial network database. In each technique that we investigated, we presented its major idea and provided the insights on the reasons why it works. We

Reference	Preprocessing Time	Distance Query Time
SILC [38]	$\Theta(n^2 \log n)$	$O(l \log n)$
PCPD [43]	$\Theta(n^2 \log n)$	$O(l \log n)$
AH [48]	$O(hn^2\lambda^2)$	$O(h\lambda^2 \log h + h\lambda^4)$
TNV [3]	$\Omega(n^2 \log n \log \alpha)$	$O(\Lambda^2 \log^2 n \log^2 \alpha)$
CH [20]	$O(n^2 \log n)$	$O(\sqrt{n} \log n)$
PLL [6, 7]	$O(n^2 \log^2 n)$	$O(\sqrt{n} \log n)$

Reference	Space	Shortest Path Query Time
SILC [43]	$O(n\sqrt{n})$	$O(l \log n)$
PCPD [38]	$O(s^d \cdot n)$	$O(l \log n)$
AH [48]	$O(hn\lambda^2)$	$O(h\lambda^2 \log h + h\lambda^4 + l)$
TNV [3]	$O(\Lambda n \log n \log \alpha)$	$O(\Lambda^2 \log^2 n \log^2 \alpha + l)$
CH [20]	$O(n^2)$	$O(\sqrt{n} \log n + l)$
PLL [6, 7]	$O(n\sqrt{n} \log n)$	$O(l\sqrt{n} \log n)$

Remark: l is the number of edges in the shortest path. s is around 12 in practice and d is the dimensionality of the network. h (resp. λ) is the height of the hierarchy which is equal to $\log \frac{\max_{u,v \in V} d_G(u,v)}{\min_{u,v \in V} d_G(u,v)}$ (resp. the *Arterial Dimension* of the spatial networks) and they were proposed in [48]. h (resp. λ) is at most 26 (resp. 100) in the datasets tested in [48]. α is defined to be $\frac{\max_{u,v \in V} d_G(u,v)}{\min_{u,v \in V} d_G(u,v)}$. λ and Λ are all $\Omega(\sqrt{n})$ in a grid spatial network like that in Manhattan District, where Λ is a concept called *Highway Dimension*.

Table 1: Comparison of Distance and Path Oracles on Spatial Networks

also made a comparison of the existing works. One interesting research topic in the future is the distance/path queries on the dynamic spatial networks, where there are frequent updates of the weights of the edges. Although CH and HH have the update algorithm, they are not efficient enough to meet the real-time requirement in a highly dynamic network. It is also very interesting to propose a distance/path oracle with favorable theoretical guarantees on the preprocessing time, the space, the query time and the update time.

Reference	Preprocessing Time	Distance Query Time
SILC	$\Theta(n^2 \log n)$	$O(l \log n)$
PCPD	$\Theta(n^2 \log n)$	$O(l \log n)$
2-Hop	$O(n^2 \log^2 n)$	$O(\sqrt{n} \log n)$
CH	$O(n^2 \log n)$	$O(\sqrt{n} \log n)$
AH	$O(hn^2)$	$O(h \log h)$

Reference	Space	Shortest Path Query Time
SILC	$O(n\sqrt{n})$	$O(l \log n)$
PCPD	$O(s^d \cdot n)$	$O(l \log n)$
2-Hop	$O(n\sqrt{n} \log n)$	$O(l\sqrt{n} \log n)$
CH	$O(n^2)$	$O(\sqrt{n} \log n + l)$
AH	$O(hn)$	$O(h \log h + l)$

Remark: l is the number of edges in the shortest path. s is around 12 in practice and d is the dimensionality of the network. h (resp. λ) is the height of the hierarchy which is equal to $\log \frac{\max_{u,v \in V} d_G(u,v)}{\min_{u,v \in V} d_G(u,v)}$ (resp. the *Arterial Dimension* of the spatial networks) and they were proposed in [48]. h (resp. λ) is at most 26 (resp. 100) in the datasets tested in [48]. α is defined to be $\frac{\max_{u,v \in V} d_G(u,v)}{\min_{u,v \in V} d_G(u,v)}$. λ and Λ are all $\Omega(\sqrt{n})$ in a grid spatial network like that in Manhattan District, where Λ is a concept called *Highway Dimension*.

Table 2: Comparison of Distance and Path Oracles on Spatial Networks

References

[1] http://mathworld.wolfram.com/treewidth.html.

[2] I. Abraham, D. Delling, A. V. Goldberg, and R. F. Werneck. Hierarchical hub labelings for shortest paths. In *Algorithms–ESA 2012*. 2012.

[3] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *SODA*, 2010.

[4] T. Akiba, T. Hayashi, N. Nori, Y. Iwata, and Y. Yoshida. Efficient top-k shortest-path distance queries on large networks by pruned landmark labeling. In *AAAI*, 2015.

[5] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *ALENEX*, 2014.

[6] T. Akiba, Y. Iwata, and Y. Yoshida. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *SIGMOD*, 2013.

[7] T. Akiba, Y. Iwata, and Y. Yoshida. Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling. In *WWW*, 2014.

[8] T. Akiba, C. Sommer, and K.-i. Kawarabayashi. Shortest-path queries for complex networks: exploiting low tree-width outside the core. In *EDBT*, 2012.

[9] M. Barthélemy. Spatial networks. *Physics Reports*, 2011.

[10] H. Bast, S. Funke, and D. Matijević. Transit: ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge—Shortest Path*, 2006.

[11] H. Bast, S. Funke, P. Sanders, and D. Schultes. Fast routing in road networks with transit nodes. *Science*, 2007.

[12] G. V. Batz, D. Delling, P. Sanders, and C. Vetter. Time-dependent contraction hierarchies. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, 2009.

[13] R. Bauer, T. Columbus, I. Rutter, and D. Wagner. Search-space size in contraction hierarchies. In *Automata, Languages, and Programming*.

[14] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 2003.

[15] U. Demiryurek, F. Banaei-Kashani, C. Shahabi, and A. Ranganathan. Online computation of fastest path in time-dependent spatial networks. In *SSTD*, 2011.

[16] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1959.

[17] D. Eppstein and M. T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *SIGSPATIAL*, 2008.

[18] S. Funke and S. Storandt. Provable efficiency of contraction hierarchies with randomized preprocessing. In *Algorithms and Computation*, pages 479–490. Springer, 2015.

[19] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms*. 2008.

[20] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 2012.

[21] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, 2005.

[22] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for a*: Efficient point-to-point shortest path algorithms. In *ALENEX*, 2006.

[23] S. Gupta, S. Kopparty, and C. Ravishankar. Roads, codes, and spatiotemporal queries. In *PODS*, 2004.

[24] R. H. Güting. An introduction to spatial database systems. *VLDBJ*, 1994.

[25] R. J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. *ALENEX/ANALC*, 2004.

[26] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 1968.

[27] M. Jiang, A. W.-C. Fu, R. C.-W. Wong, and Y. Xu. Hop doubling label indexing for point-to-point distance querying on scale-free networks. *VLDB*, 2014.

[28] R. Jin, N. Ruan, Y. Xiang, and V. Lee. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *SIGMOD*, 2012.

[29] S. Jung and S. Pramanik. Hiti graph model of topographical road maps in navigation systems. In *ICDE*, 1996.

[30] S. Jung and S. Pramanik. An efficient path computation model for hierarchically structured topographical road maps. *TKDE*, 2002.

[31] E. Kanoulas, Y. Du, T. Xia, and D. Zhang. Finding fastest paths on a road network with speed patterns. In *ICDE*, 2006.

[32] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 1998.

[33] P. N. Klein, S. Mozes, and O. Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space o (n log 2 n)-time algorithm. *TALG*, 2010.

[34] N. Milosavljević. On optimal preprocessing for contraction hierarchies. In *SIGSPATIAL*, 2012.

[35] S. Mozes and C. Sommer. Exact distance oracles for planar graphs. In *SODA*, 2012.

[36] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, 2003.

[37] M. Rice and V. J. Tsotras. Graph indexing of road networks for shortest path queries with label restrictions. *VLDB*, 2010.

[38] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, 2008.

[39] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *European Symposium on Algorithms (ESA)*. 2005.

[40] P. Sanders and D. Schultes. Engineering highway hierarchies. In *European Symposium on Algorithms (ESA)*. 2006.

[41] J. Sankaranarayanan and H. Samet. Distance oracles for spatial networks. In *ICDE*, 2009.

[42] J. Sankaranarayanan and H. Samet. Query processing using distance oracles for spatial networks. *TKDE*, 2010.

[43] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *VLDB*, 2009.

[44] D. Schultes and P. Sanders. Dynamic highway-node routing. In *Experimental Algorithms*. 2007.

[45] S. Wang, W. Lin, Y. Yang, X. Xiao, and S. Zhou. Efficient route planning on public transportation networks: A labelling approach. In *SIGMOD*, 2015.

[46] F. Wei. Tedi: efficient shortest path query answering on graphs. In *SIGMOD*, 2010.

[47] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *VLDB*, 2012.

[48] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou. Shortest path and distance queries on road networks: towards bridging theory and practice. In *SIGMOD*, 2013.