On k-skip Shortest Paths

Yufei Tao† Cheng Sheng† Jian Pei‡

†Department of Computer Science and Engineering
Chinese University of Hong Kong
New Territories, Hong Kong
{taoyf, csheng}@cse.cuhk.edu.hk

‡School of Computing Science
Simon Fraser University
Burnaby, BC Canada
jpei@cs.sfu.ca

ABSTRACT

Given two vertices s, t in a graph, let P be the shortest path (SP) from s to t, and $P^\star$ a subset of the vertices in P. $P^\star$ is a *k-skip shortest path* from s to t, if it includes at least a vertex out of every k consecutive vertices in P. In general, $P^\star$ succinctly describes P by sampling the vertices in P with a rate of at least $1/k$. This makes $P^\star$ a natural substitute in scenarios where reporting every single vertex of P is unnecessary or even undesired.

This paper studies k-skip SP computation in the context of *spatial network databases (SNDB)*. Our technique has two properties crucial for real-time query processing in SNDB. First, our solution is able to answer k-skip queries *significantly faster* than finding the original SPs in their entirety. Second, the previous objective is achieved with a structure that occupies *less space* than storing the underlying road network. The proposed algorithms are the outcome of a careful theoretical analysis that reveals valuable insight into the characteristics of the k-skip SP problem. Their efficiency has been confirmed by extensive experiments with real data.

Categories and Subject Descriptors

H3.3 [**Information search and retrieval**]: Search process

General Terms

Algorithms

Keywords

k-skip, shortest path, road network

1. INTRODUCTION

Finding *shortest paths (SP)* in a graph is a classic problem in computer science. Formally, let $G = (V, E)$ be a graph where V is a set of vertices and E a set of edges. Each edge carries a non-negative weight. Define the *length* of a path P, represented as $\|P\|$, to be the total weight of the edges in P. Given two vertices $s, t \in V$, a *SP query* returns the path from s to t with the minimum length.

SIGMOD'11, June 12–16, 2011, Athens, Greece.

The SP problem has received considerable attention from the research community in the past few years (see the recent work [4, 24, 27] and the references therein), due to its profound importance in *Spatial Network Databases (SNDB)*. An SNDB manages geometric entities positioned in an underlying *road network*, and supports efficient data retrieval with predicates on network distances, and optionally also on spatial properties (a representative system is Google Maps). A standard modeling of a road network is a graph where each vertex corresponds to a junction and each edge represents a road segment. An edge's weight is often set to the length of the corresponding road segment or the average time for a vehicle to pass through the segment.

Traditionally, a SP query retrieves *every vertex* on the shortest path P, which is sometimes unnecessarily detailed in practice. Consider, for example, that a person leaves home for a retreat destination. Typically, the SP would first wind through her/his neighborhood R_1, continue onto a set of highways R_2, and eventually enter the neighborhood R_3 of the destination. The region, in which fine-detailed directions are imperative, is R_3. In R_1 and R_2, it is often sufficient to guide the user at a *coarse* level, in a manner similar to putting *sign-posts* along the way, for example, by naming some streets to be passed, and the highways to be taken in succession.

In fact, even the computation of turn-by-turn driving directions does not always demand all the vertices on P. This is because P may contain vertices at which *no turning is needed*. To illustrate, assume that P stays on the *Main Street* in an urban area for one kilometer, during which the street intersects another street every 100 meters. Each of those 10 intersections is a vertex in P, but only the first and last of them are enough to generate the instructions for turning into and away from the Main Street, respectively. The situation is similar if P involves a long highway, in which the vertices between the points where P enters and leaves the highway respectively can be ignored.

In this paper, we are interested in computing a *subset*, say $P^\star$, of the vertices in P. Instead of an arbitrary subset, however, we demand that P^* be *k-skip shortest path*, namely, it should contain *at least one* vertex out of every k *consecutive* vertices in P. Figure 1 shows an example with $k = 4$, where P consists of all the (black and white) vertices while $P^\star$ only the black ones. Note that there can be more than one black vertex in every 4 consecutive vertices in P, but it is impossible to have none.

$P^\star$ succinctly describes P by sampling its vertices with a rate of at least $1/k$. Such a sample set can replace P in answering queries like: what are the highways (alternatively, neighborhoods or cities) that P needs to go through? $P^\star$ is equally useful in discovering the gas stations (similarly, attractions or hotels) along P, because

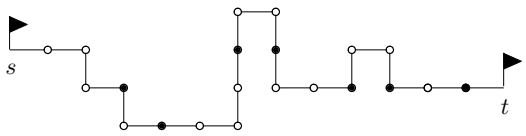


Figure 1: The black vertices constitute a 4-skip shortest path

it is often sufficient to find the stations close to the vertices in $P^\star$, as opposed to all the vertices of P. This is the idea of [18] in approaching the *continuous nearest neighbor* problem. Moreover, $P^\star$ is also adequate for estimating various statistics about P, as in the query: find the percentage of dessert coverage in the route from Los Angeles to Las Vegas. Finally, $P^\star$ is exactly what is needed to plot the original P in a digital map with a decreased resolution. For example, at Google Maps, only a subset of the vertices on a path need to be displayed according to the current zooming level.

For traditional mapping purposes, $P^\star$ has two notable advantages over P. First, it is (much) smaller in size, and hence, requires less bandwidth to transmit. This is a precious feature in mobile computing that will especially be appreciated by users charged by the amount of Internet usage. Second, using a clever algorithm, $P^\star$ can be faster to compute because, intuitively, fewer vertices need to be retrieved than P. Such an efficiency gain provides valuable opportunities for in-car navigation devices and routing websites such as Google Maps to support a great variety of on-route services in shorter time.

The concept of k-skip SP comes with a *zoom-in* operation. Given consecutive vertices u, v on $P^\star$, the operation finds the missing vertices on P from u to v. As there are at most $k-1$ missing vertices, for small k a zoom-in incurs negligible time, because it only needs to compute a *very short* path. This adds a nice *pay-as-you-go* feature in applying k-skip SP. First, a driver can request the least zoom-in's to complete the part of the route outside her/his knowledge. This allows her/him to pay for the most useful information only. Second, an algorithm preparing turn-by-turn directions can zoom-in only when necessary (i.e., a turning may exist between two adjacent vertices $P^\star$), thus saving the cost of locating the skipped vertices.

Contributions. Somewhat surprisingly, the notion of k-skip SP, or in general the idea of sampling (the vertices in) a SP, has not been mentioned previously to the best of our knowledge, let alone any algorithm dealing with the problem of k-skip SP computation. We fill the gap with a systematic study of this problem. In particular, our objectives are twofold:

1. Resolve a k-skip query *significantly faster* than calculating the original SP in entirety.
2. Achieve the previous purpose with a data structure that occupies *less space* than storing the input graph $G = (V, E)$ itself. This permits the structure to reside in memory even for the largest road network, as is crucial for real-time query processing in SNDB.

The first contribution of this paper is to formally establish the fact that, only a small subset $V^\star$ of the vertices in G need to be considered to attack the k-skip problem. In other words, the other vertices in $V \backslash V^\star$ are never needed to form any k-skip SP. Referring to $V^\star$ as a *k-skip cover*, we show that there always exists a $V^\star$ with size roughly proportional to $|V|/k$. This theoretical finding is of independent interest because it is not limited to road networks, but *actually holds for general graphs*.

Our second contribution is a full set of algorithms required to process k-skip SP queries within a stringent time limit. Specifically, these algorithms settle three sub-problems: (i) find a small $V^\star$ in time *linear* to the complexity of G, (ii) construct from $V^\star$ a space-economical structure, and (iii) answer a k-skip query by accessing only a small portion of the structure. We thoroughly evaluated these algorithms with extensive experiments on real data, including the massive road network of the US which has about 24 million vertices and 58 million edges. Our results prove that the proposed technique very well satisfy both requirements mentioned earlier in all settings.

Roadmap. The next section reviews the literature of SP computation. Section 3 formally defines the target problem, and gives an overview of our solutions. Section 4 elaborates the theoretical results on k-skip covers, and the algorithms for constructing the proposed structure. Section 5 clarifies how to answer a k-skip query and perform a zoom-in efficiently. Section 6 contains the experimental results, while Section 7 concludes the paper with directions for future work.

2. RELATED WORK

There is an extremely rich literature on the SP problem. In Section 2.1, we clarify the details of the *reach* algorithm, which is the state of the art for SP computation in road networks. Section 2.2 surveys the other recent progress in the database and theory communities.

2.1 *Dijkstra* and *reach*

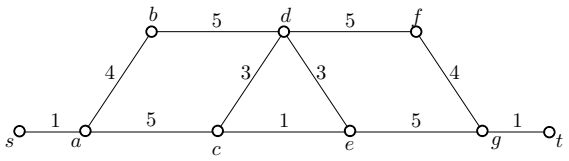
To facilitate discussion, given two vertices u, v in the input graph $G = (V, E)$, we denote by $SP(u, v)$ the SP from u to v. The length of $SP(u, v)$, i.e., $\|SP(u, v)\|$, is called the *distance* between u and v. If (u, v) is an edge in G, we represent its weight as $\ell(u, v)$. In case G is directed, (u, v) is an edge from u to v, and (v, u) means the opposite. To avoid unnecessary distraction, our examples use undirected graphs, but all the notations in our description are carefully written so that they are also correct for directed graphs.

Dijkstra. Let us first review the *Dijkstra*'s algorithm [5], which is the foundation of the *reach* method described shortly. *Dijkstra* finds $SP(s, t)$ by exploring the vertices in ascending order of their distances to the source s. At any time, each vertex $u \in V$ has a status chosen from {`unseen`, `labeled`, `scanned`}. Furthermore, u is associated with a label $l(u)$ equal to the distance of the SP from s to u found *so far* (i.e., an even shorter path may be discovered later). On this path, the vertex immediately preceding u is kept in $pred(u)$, referred to as the *predecessor* of u.

At the beginning of *Dijkstra*, all vertices $u \in V$ have status `unseen`, $l(u) = \infty$ and $pred(u) = \emptyset$. The only exception is s, whose status is `labeled`, with $l(s) = 0$ and $pred(u) = \emptyset$. At all times, the vertices of status `labeled` are managed by a min-heap Q, using their labels as the sorting key. In each iteration, the algorithm (i) de-heaps the vertex $u \in Q$ with the minimum $l(u)$, (ii) *relaxes* all edges (u, v) such that the status of v is *not* `scanned`, and (iii) changes the status of u to `scanned`. Specifically, relaxation of (u, v) involves the following steps:

---

relax (u, v)
1. **if** the status of v is `unseen` **then**
2. $\quad l(v) \leftarrow l(u) + \ell(u, v)$
3. $\quad$ the status of $v \leftarrow$ `labeled`
4. **else if** $l(v) > l(u) + \ell(u, v)$ **then**



(a) The input graph

a	b	c	d	e	f	g	s	t
1	0	6	5	6	0	1	0	0

(b) Global reaches of the vertices

Figure 2: Illustration of *reach*

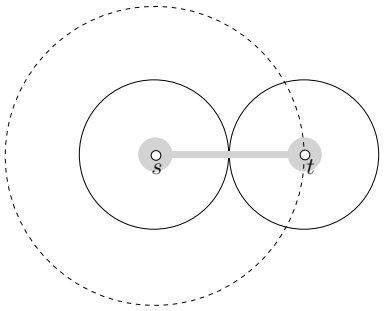


Figure 3: Comparison of *Dijkstra*, *bi-directional*, and *reach*

5. $l(v) \leftarrow l(u) + \ell(u, v)$
6. $pred(v) \leftarrow u$

The present $l(u)$ is guaranteed to be $\|SP(s, u)\|$. The algorithm terminates as soon as t, the destination, is selected by an iteration.

To illustrate, assume that we want to compute $SP(s, t)$ in Figure 2a. In the first iteration, *Dijkstra* scans s and relaxes (s, a), after which $Q = \{a\}$ and $l(a) = 1$. The next iteration de-heaps a and relaxes $(a, b), (a, c)$. Note that (a, s) is not relaxed because the status of s has become `scanned`. Now Q contains b, c with labels $5, 6$, respectively. The algorithm then scans b and relaxes (b, d), which labels d with 10. This is followed by de-heaping c, and relaxing $(c, d), (c, e)$. Note that the relaxation of (c, d) decreases $l(d)$ to 9. The rest of the algorithm proceeds in the same manner. By tracing the execution, one can verify that, at termination, all the vertices except t have already been scanned.

Dijkstra can be implemented in $O(m + n \log n)$ time [3], where n, m are the number of vertices and edges, respectively (i.e., $n = |V|, m = |E|$). In a road network, each vertex has an $O(1)$ degree, so the time complexity is essentially $O(n \log n)$.

Bi-directional. *Dijkstra* starts a graph traversal from s and gradually approaches t; call this a *forward* search. Immediately by symmetry, the SP problem can also be solved by a *backward* search from t to s (if G is directed, the *backward* search implicitly reverses the direction of each edge). The *bi-directional* algorithm [10, 20] achieves better efficiency by performing both searches synchronously, the effect of which is essentially to explore the vertices $u \in V$ in ascending order of $r_{s,t}(u)$, where

$$r_{s,t}(u) = \min\{\|SP(s, u)\|, \|SP(u, t)\|\}. \quad (1)$$

In fact, if u is closer to s (than to t), then it will first be touched in the forward search; otherwise, the backward search will find it first. The forward/backward search proceeds as in the standard *Dijkstra*'s algorithm (as if the other search did not exist). Let Q_f (Q_b) be the heap of the forward (backward) direction. Synchronization is carried out by advancing in each iteration the direction that has a smaller label at the top of the heap.

Bi-directional monitors the distance λ of the shortest s-to-t path found *so far*. λ is set to ∞ initially, and may be updated when the forward search (the backward case is symmetric) de-heaps a vertex u whose status in the backward direction is `labeled`[1]. Specifically, at this time, the algorithm computes $\lambda' = l_f(u) + l_b(u)$, where $l_f(u)$ and $l_b(u)$ are the labels of u in the forward and backward search, respectively. λ is reduced to λ' if $\lambda > \lambda'$, since it implies the existence of a shorter s-to-t path that concatenates $SP(s, u)$, (u, v), and $SP(v, t)$, where v is the current predecessor of u in the backward search. The algorithm terminates when either direction de-heaps a vertex already scanned in the other direction.

Let us demonstrate *bi-directional* on the graph in Figure 2a. After three iterations of each direction, which are the same as in *Dijkstra*, the forward (backward) search has scanned s, a, b (t, g, f), such that Q_f (Q_b) contains vertices c, d (e, d) with labels 6, 10, respectively. Currently, $\lambda = \infty$. Next, the forward search de-heaps $c \in Q_f$ and relaxes edges (c, d), (c, e), after which $l_f(e) = 7, l_f(d) = 9$. Similarly, the backward search then de-heaps $e \in Q_b$. As the status of e in the forward direction is `labeled`, the algorithm updates λ to $l_f(e) + l_b(e) = 7 + 6 = 13$, before it relaxes $(e, c), (e, d)$. The forward search continues by de-heaping $e \in Q_f$, which, however, has been scanned in the backward search. The algorithm therefore terminates, without de-heaping d in either direction.

Reach. Intuitively, if λ is the length of $SP(s, t)$, *Dijkstra* searches a ball that centers at s, and has radius λ, shown as the dashed circle in Figure 3. *Bi-directional*, on the other hand, explores two balls with radius $\lambda/2$ that center at s, t respectively (the two solid circles in Figure 3). The *reach* algorithm, which is the current state of the art, dramatically shrinks the search area to a small dumb-bell shape, as illustrated by the shaded region in Figure 3.

Let us start the explanation with the notion of *local reach*. Let u be a vertex on $SP(s, t)$. The *local reach of u in $SP(s, t)$* equals $r_{s,t}(u)$ as calculated in Equation 1. Note that this notion relies on a particular SP. Any vertex $v \notin SP(s, t)$ has no local reach defined in $SP(s, t)$. Now we extend the definition to *global reach*:

DEFINITION 1 (GLOBAL REACH [12]). *The **global reach**, denoted as $r(u)$, of a vertex u is the maximum local reach of u in all the shortest paths that pass u. Formally:*

$$r(u) = \max_{s,t|u \in SP(s,t)} r_{s,t}(u). \quad (2)$$

The reach $r(u)$ can be understood intuitively as follows. If u is on $SP(s, t)$, then *either s or t* must have distance at most $r(u)$ to u. In other words, in case neither s nor t is within distance $r(u)$ from u, u can be safely pruned in computing $SP(s, t)$.

Consider, for instance, vertex c in Figure 2a. To decide its global reach $r(c)$, first collect the set S of all the SPs that pass c, namely, $S = \{SP(s, t), SP(s, g), ..., SP(a, e), ...\}$. We then calculate the local reach of c in each SP of S. For example, its local reach in $SP(a, e)$ is $\min\{\|SP(a, c)\|, \|SP(c, e)\|\} = 1$. The final $r(c)$

[1]This status cannot be `scanned`; otherwise, the algorithm would have terminated right after u was de-heaped by the forward search, as will be clear shortly.

equals the maximum of all the local reaches, which can be verified to be 6 (as is its local reach in $SP(s,t)$). Figure 2b shows the global reaches of all the vertices.

In computing $SP(s,t)$, the *reach* algorithm [10, 12] proceeds in the same way as *bi-directional*, but may prune a vertex in relaxing an edge (u,v). Without loss of generality, suppose that the relaxation takes place in the forward search (the backward case is symmetric). This implies that the forward search just scanned u, but has never scanned v (otherwise the edge would not have been relaxed). The pruning of *reach* takes place as follows:

RULE 1. *Vertex v is pruned if both of the following hold:*

1. *v has status* `labeled` *or* `unseen` *in the backward direction*
2. $r(v) < l_f(u)$

where $l_f(u)$ is the label of u in the forward search.

In general, *reach* can be understood as the execution of *bi-directional* on the vertices that survive pruning. In Figure 2, for example, *reach* finds $SP(s,t)$ by scanning only s, a, c, e, g, t. As in *bi-directional*, *reach* first scans s (t) in the forward (backward) direction, after which Q_f (Q_b) includes a (g) with label 1. Next, the forward search de-heaps a from Q_f, and relaxes $(a,b), (a,c)$. The relaxation of (a,b) prunes b by Rule 1 because $r(b) = 0 < l_f(a) = 1$. The backward search then eliminates f in a similar fashion. The rest of the algorithm proceeds as in *bi-directional*. Vertex d does not need to be scanned for the reason explained earlier for *bi-directional*.

The space consumption of *reach* is very economical because, except G itself, only *one extra value* needs to be stored for each vertex. However, it can be expensive to calculate the exact reach of every vertex. Fortunately, there are fast algorithms [10, 12] to obtain approximate reaches that are guaranteed to *upper bound* their original values. Pruning remains the same except that $r(u)$ should be replaced by its upper bound. The outstanding efficiency of this algorithm on road networks has been theoretically justified [1].

2.2 More results on SP computation

The $A^\star$ algorithm [13] is a classic SP solution that captures *Dijkstra* as a special case. The effectiveness of $A^\star$ relies on a method to calculate, typically in constant time, a lower bound of $\|SP(u,v)\|$ for any two vertices u, v. Apparently, 0 can be a trivial lower bound, but in that case $A^\star$ degenerates into *Dijkstra*. In general, the tighter the lower bounds are, the faster $A^\star$ is than *Dijkstra*.

Motivated by this, Agrawal and Jagadish [2] proposed to precompute the distances between each vertex and a special set of vertices called *landmarks*. In answering a SP query, those distances are deployed to derive lower bounds based on triangle inequality. This idea is also the basis of the *ALT* algorithm developed by Goldberg and Harrelson [9], which has lower query time (than [2]) at the tradeoff of consuming more space. The experiments of [10] indicate that *ALT* is slower than the *reach* method described in Section 2.1. Note, however, that *ALT* and *reach* are compatible, in the sense that their heuristics can be applied at the same time to maximize efficiency [10].

Sanders and Schultes [8, 25, 26] presented a *highway hierarchy (HH)* technique, whose rationale is to identify some edges that mimic the role of highways in reality. In computing a SP, the algorithm of *HH* modifies *Dijkstra* so that the search can skip as many non-highway edges as possible, and thus, terminate earlier. Based on the empirical evaluation of [10, 25], *HH* has comparable performance with respect to *reach*.

The *separator* technique is another popular approach [15, 16, 17] to preprocess a graph G for efficient SP retrieval. The idea is to divide the vertices of G into several components, and for each component, extract a set of *boundary vertices* such that the SP between two vertices in different components must go through a boundary vertex. Query efficiency benefits from the fact that, most computation of a SP can be restricted only to the boundary vertices. According to [25], however, separator-based methods are not as efficient as *HH* on road networks. Another disadvantage is that these methods often have expensive space consumption. For example, the space of [15] is at the order of $n^{1.5}$ (where n is the number of vertices), which is prohibitive for massive graphs.

In [28], Wagner et al. described a *geometric container* technique, which associates each edge (u,v) in G with a geometric region. The region covers all the vertices t such that the SP from u to t goes through v. In running *Dijkstra*, such regions can be used to prune many vertices that do not appear in the SP. Lauther [19] integrated a similar idea with the *bi-directional* algorithm discussed in Section 2.1. Samet et al. [24] proposed to break the geometric regions into smaller disjoint pieces that can be indexed by a quadtree. Their solution lowers the cost of SP calculation (compared to [19, 28]), but entails $O(n^{1.5})$ space. A common shortcoming of [19, 24, 28] is that their preprocessing essentially determines the SPs between all pairs of vertices. The all-SP problem is notoriously difficult. Even on planar graphs, the fastest solution requires $O(n^2)$ time [7], which is practically intractable for large n.

Recently, Sankaranarayanan et al. [27] proposed a *path oracle* that is constructed from *well-separated pair decomposition*, and can be used to accelerate SP retrieval. Such an oracle consumes $O(s^2 n)$ space in 2-d space, where s is shown to be around 12 for practical data. Xiao et al. [30] showed that SP queries can be accelerated by exploiting symmetry in the data graph. Their approach, however, is limited to the case where all edges have a unit weight. Wei [29] developed an alternative solution by resorting to *tree decomposition* [23]. Rice and Tsotras [22] studied the shortest path problem with label restrictions. Some other work considers how to *estimate* shortest path distances, e.g., [11, 21].

We emphasize that all the works aforementioned calculate traditional, complete, SPs. The concept of k-skip SP has not appeared previously, and is formalized in the next section for the first time in the literature.

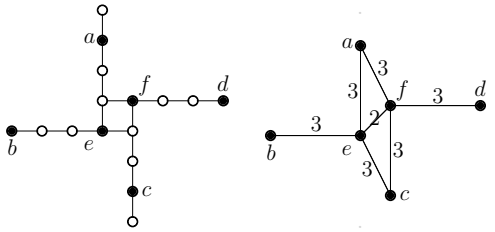
3. K-SKIP SHORTEST PATHS

For simplicity, we assume that the data graph $G = (V, E)$ is undirected, and will discuss directed graphs only when the extension is not straightforward. The following definition formalizes k-skip SP:

DEFINITION 2 (k-SKIP SHORTEST PATH). *Let $SP(s,t)$ be the shortest path from source s to destination t. Label the vertices in $SP(s,t)$ in the order they appear: $v_1, v_2, ..., v_l$ (i.e., $v_1 = s, v_l = t$), where l is the total number of vertices in the path. If $l \geq k$, let $P^\star$ be an ordered subset of $\{v_1, ..., v_l\}$, where the ordering respects that in $SP(s,t)$. $P^\star$ is a* ***k-skip shortest path*** *from s to t if*

$$P^\star \cap \{v_i, ..., v_{i+k-1}\} \neq \emptyset \quad (3)$$

for every $1 \leq i \leq l - k + 1$.



(a) Black vertices constitute a 3-skip cover (b) 3-skip graph

Figure 4: Preprocessing for 3-skip SP computation

Even for fixed s and t, there can be multiple $P^\star$ satisfying the above condition. In other words, k-skip SPs are *not* unique, although all of them have to be subsets of $SP(s,t)$. Our objective is to preprocess G into a space-economical structure such that, given any s, t at run time, we can find a k-skip SP between any s and t efficiently.

The first step of our preprocessing is to eliminate the redundant vertices in G. The intuition is that, since most vertices in a SP need *not* be reported under the k-skip definition, certain vertices would end up being *never* returned. For example, consider the graph G in Figure 4a where all edges have a unit weight. Observe that, for $k = 3$, any SP with 3 vertices must contain at least one black vertex. In other words, it suffices to form k-skip SPs using *just* the black vertices, whereas the other (white) vertices can be discarded.

We refer to the set of black vertices in the above example as a *3-skip cover*. The next definition generalizes the notion to *k-skip cover*, which contains a subset of the vertices that need to be considered for k-skip SPs.

DEFINITION 3 (k-SKIP COVER). *Let $V^\star$ be a subset of the vertices in G. $V^\star$ is a **k-skip cover** if it has the property that, for arbitrary $s, t \in V$ such that $SP(s,t)$ has at least k vertices,*

$$V^\star \cap SP(s,t)$$

is a k-skip SP from s to t, after ordering the vertices of $V^\star \cap SP(s,t)$ appropriately.

As will be clear in Section 5, the efficiency of our k-skip SP algorithm crucially depends on the fact that, it does not need to process the vertices of $V \setminus V^\star$. Hence, the minimization of $|V^\star|$ is essential, but it turns out to be rather challenging on an arbitrarily complex G. In fact, it is non-trivial even if one simply wants to know *whether* a small $V^\star$ always exists. These issues will be resolved in the next section.

We want to capture the adjacency of the vertices in $V^\star$ as far as k-skip SPs are concerned. For example, the black vertex a in Figure 4a can be consecutive only to e and f in a 3-skip SP (observe that e and f have blocked all the possible ways that can lead a to any other black vertex). We represent this by generating two "super-edges" that link a to e, f respectively. After this, the original edges (of G) incident on a can be ignored in computing any k-skip SP beginning from a.

Let us formalize the rationales behind the preceding paragraph in the next two definitions.

DEFINITION 4 (k-SKIP NEIGHBOR). *Let u, v be two vertices in a k-skip cover $V^\star$. We say that v is a **k-skip neighbor** of u if $SP(u,v)$ (namely, the shortest path from u to v in G) does not pass any other vertex in $V^\star$.*

It is easy to see that the SP from u to any of its k-skip neighbor v can have at most k edges. In an undirected G, the relation implied by the above definition is symmetric, i.e., u is a k-skip neighbor of v if and only if v is a k-skip neighbor of u. This is not necessarily true for directed graphs. In any case, let $N_k(u)$ be the set of all k-skip neighbors of u. For instance, in Figure 4a, $N_3(a) = \{e, f\}$.

DEFINITION 5 (k-SKIP GRAPH). *A **k-skip graph** of G is a graph $G^\star = (V^\star, E^\star)$, where*

- *$V^\star$ is a k-skip cover of G;*
- *for every vertex $u \in V^\star$, $E^\star$ has an edge (u, v) for each k-skip neighbor v of u, namely, $v \in N_k(u)$.*

The weight of $(u, v) \in E^\star$ is $\|SP(u,v)\|$.

We call each edge $(u, v) \in E^\star$ a *super-edge*. Figure 4b demonstrates the 3-skip graph $G^\star$ of the graph G in Figure 4a. The weight of (a, e) in $G^\star$ equals 3, which is the distance of a and e in G.

As shown later, finding a k-skip SP on the original graph G can be reduced to computing a (traditional) SP on a k-skip graph $G^\star$. In the next section, we will delve into the properties of $G^\star$, and give a method to construct it efficiently. Then, the reduction and the accompanied algorithms will be elaborated in Section 5.

4. *K*-SKIP GRAPH

The effectiveness of our pre-computed structure, namely a k-skip graph $G^\star = (V^\star, E^\star)$, relies on the size of the k-skip cover $V^\star$. No performance gain would be possible if a small $V^\star$ could not be guaranteed. Fortunately, we will show that such a good $V^\star$ always exists (Section 4.1). Sections 4.2 and 4.3 then elaborate the algorithms for building $G^\star$.

4.1 Size of *k*-skip cover

This subsection will establish:

THEOREM 1. *Let $G = (V, E)$ be a graph with $n = |V|$ vertices. For any $1 \leq k \leq n$, G has a k-skip cover $V^\star$ of size $O(\frac{n}{k} \log \frac{n}{k})$.*

Our proof leverages the theory of *Vapnik-Chervonenkis (VC) dimension*, which quantifies how decomposable a search problem is. Specifically, let $\mathcal{D}$ be a dataset and $\mathcal{Q}$ be a set of queries that can be issued on $\mathcal{D}$. Given a query $q \in \mathcal{Q}$, define $q(\mathcal{D})$ to be the result of q on $\mathcal{D}$. A *shatterable* set $S \subseteq \mathcal{D}$ is such that, for any $S' \subseteq S$, there is always a query $q \in \mathcal{Q}$ with $q(\mathcal{D}) \cap S = S'$. In other words, any subset $S' \subseteq S$ needs to have the property that, a query $q \in \mathcal{Q}$ retrieves *all* the items of S', and *nothing* from $S \setminus S'$ (the result of q, however, may also include items from $\mathcal{D} \setminus S$). The VC dimension of $(\mathcal{D}, \mathcal{Q})$ is the size of the largest shatterable subset of $\mathcal{D}$. Note that the VC dimension is *not* defined on a dataset, but instead, on a *pair* of a dataset and a query set.

Now let us analyze the VC dimension of the SP problem. Here, we have an input graph $G = (V, E)$. The $\mathcal{D}$ mentioned earlier corresponds to V. A SP query $q_{s,t}$ is uniquely defined by a source vertex s and a destination vertex t. Hence, the result $q_{s,t}(V)$ of $q_{s,t}$ is $SP(s,t)$. Let Q be the union of all the possible SP queries, namely, $Q = \bigcup_{s,t \in V} q_{s,t}$ (thus, $|Q| = n^2$). Next, we give a crucial lemma:

LEMMA 1. *For any graph G, the VC dimension of* (V, Q) *is 2.*

PROOF. Assume for contradiction that the VC dimension d of (V, Q) is at least 3 (note that d must be an integer). Hence, there is a shatterable set S with size d, which belongs to the SP returned by a query $q \in Q$. Let $u_1, u_2, ..., u_d$ be the vertices of S ordered by their appearance on the SP. Hence, u_2 is on the SP from u_1 to u_d. In this case, however, $S' = \{u_1, u_d\}$ cannot be in any SP that does not contain u_2, contradicting the fact that S is shatterable.

It remains to verify that the VC dimension can be 2. We omit the details as this is trivial. □

The rest of the proof (for Theorem 1) concerns *ϵ-net*. Let $\mathcal{D}, \mathcal{Q}$ be as defined earlier in our introduction to VC dimension. A set $S \subseteq D$ is an *ϵ-net* of $(\mathcal{D}, \mathcal{Q})$ if $S \cap q(\mathcal{D}) \neq \emptyset$ for any q satisfying $|q(\mathcal{D})| \geq \epsilon|\mathcal{D}|$. In other words, if q retrieves no less than $\epsilon|\mathcal{D}|$ items, at least one of these items must appear in S. The lemma below draws the correspondence between ϵ-net and k-skip cover:

LEMMA 2. *A* $(\frac{k}{n})$*-net* $V^\star$ *of* (V, Q) *is a k-skip cover of G.*

PROOF. Let Q' be the set of queries $q \in Q$ such that the SP $q(V)$ returned by q has *exactly* k vertices. By the definition of (k/n)-net, for each $q' \in Q'$, $V^\star \cap q'(V) \neq \emptyset$. Now consider a query $q \in Q \setminus Q'$ whose result $q(V)$ has more than k vertices. Clearly, any sub-path of $q(V)$ including k vertices is the result $q'(v)$ of some $q' \in Q'$, from which $V^\star$ contains at least a vertex. Therefore, $V^\star$ is a k-skip cover. □

The *ϵ-net theorem* [14] dictates that any $(\mathcal{D}, \mathcal{Q})$ with VC dimension d has an ϵ-net of size $O(\frac{d}{\epsilon} \log \frac{1}{\epsilon})$. This, combined with Lemmas 1 and 2, gives Theorem 1.

Remark 1. Effectively, the proof of the ϵ-net theorem [14] shows that a random subset of $\mathcal{D}$ with size $O(\frac{d}{\epsilon} \log \frac{1}{\epsilon})$ is an ϵ-net with high probability. Hence, we can find a k-skip cover $V^\star$ by simply taking $O(\frac{n}{k} \log \frac{n}{k})$ vertices from V randomly.

Remark 2. There is a trivial lower bound of n/k on the size of a k-skip cover. Hence, the upper bound in Theorem 1 is asymptotically tight, up to only a logarithmic factor.

4.2 Computing a k-skip cover

As mentioned in the previous subsection, a k-skip cover $V^\star$ can be obtained by taking $O(\frac{n}{k} \log \frac{n}{k})$ random vertices from V. It is well-known that randomized techniques generally work much better in practice than predicted by theory. Therefore, the $V^\star$ of a real graph would be much smaller, rendering a sample set of size $O(\frac{n}{k} \log \frac{n}{k})$ most probably unnecessarily large. Of course, we could artificially reduce the number of samples, but a good sample size appears to be difficult to decide. A large size gives only marginal improvement, whereas a small size has the risk that the sample set may no longer be a k-skip cover.

We propose an *adaptive sampling (AS)* algorithm to resolve the above issue. Before explaining the algorithm, we need a few more definitions. Let the *λ-hop neighbor set* of a vertex $u \in V$, denoted as $V_\lambda(u)$, be the set of all vertices $v \in V$ that can be reached from u by crossing *at most* λ edges. Each $v \in V_\lambda(u)$ is called a *λ-hop neighbor* of u. For example, assume an input graph G as shown in Figure 5a, where all edges, except those explicitly labeled, have weight 1. The 2-hop neighbor set $V_2(u)$ of u contains all the vertices in the shaded diamond. Note that $V_2(u)$ is decided *without* taking the edge weights into account.

Denote by $G_\lambda(u)$ the graph induced by $V_\lambda(u)$, which is referred to as the *λ-hop vicinity* of u. That is, $G_\lambda(u)$ includes *all and only* the edges in G that are between the vertices of $V_\lambda(u)$. For instance, in Figure 5a, the edges of $G_2(u)$ are those that fall *entirely* in the diamond.

By computing the SPs *inside* $G_\lambda(u)$ from u to all the other vertices in $V_\lambda(u)$ (that is, each SP uses only the edges of $G_\lambda(u)$), one obtains a *λ-hop shortest path tree* $T_\lambda(u)$ rooted at u. For every vertex v in $G_\lambda(u)$, the u-to-v path in $T_\lambda(u)$ is the SP from u to v *inside* $G_\lambda(u)$. Figure 5b demonstrates the 2-hop SP tree $T_2(u)$ of u. It is worth noting the difference between a SP inside the tree and a SP in the whole graph. For example, the u-to-c path in $T_2(u)$ has length 5. Although this is the shortest when only the edges of $G_2(u)$ are considered, there is an even shorter path $u \to f \to e \to d \to c$ which uses two edges $(e, d), (d, c)$ outside the 2-vicinity of u.

We now arrive at the most crucial concept underlying the *AS* algorithm. Let $V^\star$ be a subset of the vertices in G. We say that a λ-hop SP tree $T_\lambda(u)$ is *covered by* $V^\star$, if every u-to-v path in $T_\lambda(u)$ having at least λ edges goes through at least one vertex in $V^\star$, where v is a vertex in $T_\lambda(u)$. In Figure 5a, for example, let $V^\star$ be the set of black vertices. Then, the $T_2(u)$ in Figure 5b is *not* covered because the path u-to-b has 2 edges but passes no black vertex.

The next lemma gives an important property:

LEMMA 3. $V^\star$ *is a k-skip cover if it covers the* $T_{k-1}(u)$ *of all* $u \in V$.

PROOF. Assume for contradiction that a $V^\star$ fulfilling the if-condition of the lemma is not a k-skip cover. It follows that there exist two vertices $u, v \in V$ such that $SP(u, v)$ has k vertices none of which is in $V^\star$. Since $SP(u, v)$ has $k-1$ edges, by the construction of $T_{k-1}(u)$, we know (i) all the vertices on $SP(u, v)$ are in $T_{k-1}(u)$, and (ii) the u-to-v path in $T_{k-1}(u)$ is exactly $SP(u, v)$. This means, however, that $T_{k-1}(u)$ has not been covered by $V^\star$, thus reaching a contradiction. □

We are ready to clarify the *AS* algorithm:

---

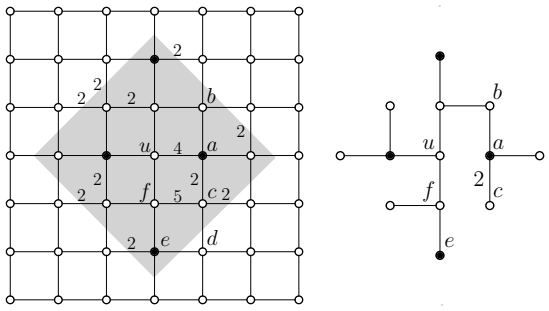
adaptive-sampling

1. $V^\star = \emptyset$
2. randomly permutate the vertices of V
3. **for** each vertex $u \in V$
4. **if** $T_{k-1}(u)$ is not covered by $V^\star$ **then**
5. add u to $V^\star$

The correctness of the algorithm follows from Lemma 3, and the fact that, if $T_{k-1}(u)$ has not been covered by $V^\star$ yet, including u itself to $V^\star$ immediately makes $T_{k-1}(u)$ covered. For instance, as shown earlier, the $T_2(u)$ in Figure 5b is not covered, but it will be, once u is added to $V^\star$.

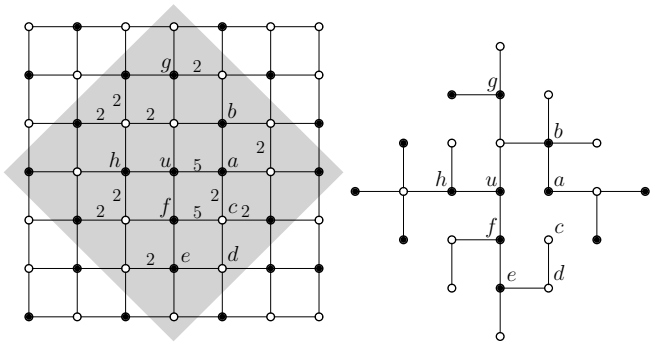
Heuristic. We can further reduce the size of $V^\star$ by first sorting the vertices of V in descending order of their degrees, and then, randomly permute the vertices with an identical degree. This increases the chance of sampling a vertex with a higher degree, which is beneficial because such a vertex tends to lie on more SPs, and therefore, have stronger covering power.

Time complexity. We close the subsection by analyzing the execution time of the algorithm, assuming that each vertex has an $O(1)$ degree, which is true in road networks. The randomization at Line 1 can be implemented in $O(n)$ time [6] where $n = |V|$ (the sorting in the high-degree-favoring heuristic can be done in $O(n)$



(a) 2-hop vicinity of u and its (b) 2-hop SP tree of u

Figure 5: Deciding whether to sample u into a 3-skip cover



(a) 3-hop vicinity of u (b) 3-hop SP tree of u

Figure 6: Deciding $M_3(u)$

time when there are only a constant number of possible degrees). The λ-hop vicinity of a node u can be found by a standard *breath first traversal (BFT)* initiated at u, which terminates in $O(\sigma_\lambda(u))$ time where $\sigma_\lambda(u)$ is the number of λ-hop neighbors of u, i.e., $\sigma_\lambda(u) = |V_\lambda(u)|$. Then, the λ-hop SP tree $T_\lambda(u)$ can be extracted from $G_\lambda(u)$ using the *Dijkstra*'s algorithm in $O(\sigma_\lambda(u) \log \sigma_\lambda(u))$ time. Checking whether $T_\lambda(u)$ is covered by $V^\star$ demands a single traversal of $T_\lambda(u)$ in $O(\sigma_\lambda(u))$ time. Hence, the total cost of the algorithm is $O(n\bar{\sigma}_{k-1} \log \bar{\sigma}_{k-1})$, where $\bar{\sigma}_{k-1}$ is the average number of $(k-1)$-hop neighbors of the vertices in V.

The value of $\bar{\sigma}_{k-1}$ depends on the structure of the road network, but not the number n of the vertices. For example, consider two simple road networks that are a 100×100 and 1000×1000 grid, respectively. Although the second grid has 100 times more vertices, the two grids have the same $\bar{\sigma}_{k-1} = \Theta(k^2)$. Hence, when k is far smaller than n, $O(n\bar{\sigma}_{k-1} \log \bar{\sigma}_{k-1})$ grows linearly with n.

4.3 Computing a k-skip graph

Recall that the goal of our preprocessing is to produce a k-skip graph $G^\star = (V^\star, E^\star)$. $V^\star$ is simply a k-skip cover, whose derivation has been clarified in Section 4.2. Next we complete the puzzle by explaining the derivation of $E^\star$. Our discussion concentrates on the subproblem that, given a vertex $u \in V^\star$, how to calculate the set $N_k(u)$ of its k-skip neighbors (Definition 4). Once this is done, it is trivial to obtain $E^\star$ according to Definition 5, because we only need to add to $E^\star$ a super-edge from each u to every vertex $v \in N_k(u)$.

We will call each vertex of $V^\star$ a *sample* from now on, due to the sampling nature of the *AS* algorithm. A naive approach to acquire $N_k(u)$ is to first find the SP from u to every other sample $v \in V^\star$, and then add v to $N_k(u)$ if the path passes no other sample. However, since $|V^\star| \geq n/k$ (see Remark 2 of Section 4.1), doing so for all $u \in V^\star$ would incur $\Omega(n^2/k)$ time, which is prohibitive for large n. We circumvent the performance pitfall by aiming at a *superset* $M_k(u)$ of $N_k(u)$. As will be clear shortly, despite that $M_k(u)$ may result in a $G^\star$ with more edges, it has the advantage of being computable in significantly less time, thus allowing our technique to support gigantic graphs.

$M_k(u)$ can be conveniently defined by recycling the notations of the previous subsection. Let us first have the k-hop SP tree $T_k(u)$ of u (*as opposed to* the $T_{k-1}(u)$ in the *AS* algorithm). Then, $M_k(u)$ includes all the samples $v \neq u$ in $T_k(u)$ such that the u-to-v path in $T_k(u)$ does not pass any other sample. For illustration, Figure 6a shows a 3-skip cover $V^\star$ (the black vertices) on the data graph of Figure 5a. To determine $M_3(u)$, we first extract the 3-hop SP tree $T_3(u)$ as in Figure 6b. Then, it becomes clear that $M_3(u) = \{b, f, g, h\}$. Note that a, for example, is not in $M_3(u)$ due to the presence of b that blocks the path from u to a.

The lemma below establishes the relationship of $M_k(u)$ and $N_k(u)$.

LEMMA 4. $N_k(u) \subseteq M_k(u)$.

PROOF. By Definition 4, the fact $v \in N_k(u)$ implies that (i) $SP(u, v)$ does not pass any other sample, and (ii) $SP(u, v)$ has no more than k edges. Property (ii) indicates that all the vertices of $SP(u, v)$ must be in the k-hop vicinity of u, and hence, are present in $T_k(u)$. It follows that the u-to-v path in $T_k(u)$ is exactly $SP(u, v)$. Property (i) further shows that the path cannot contain any other sample. Therefore, by the construction of $M_k(u)$, we know $v \in M_k(u)$. □

We call $M_k(u)$ a *super neighbor set* of u. After acquiring it, we create a super-edge (u, v) in $E^\star$ from u to each vertex $v \in M_k(u)$, and set its weight to the length of the u-to-v path in $T_k(u)$. In other words, the super-edge represents a path in the original graph. The final $E^\star$ is complete after carrying out the above procedure for all the vertices $u \in V$.

Time complexity. After $T_k(u)$ is ready, $M_k(u)$ can be easily obtained by a single traversal of $T_k(u)$ in $O(\sigma_k(u))$ time, where $\sigma_k(u)$ is the number of the k-hop neighbors of u (i.e., the number of vertices in $T_k(u)$). A straightforward adaptation of the analysis in Section 4.2 shows that the cost of processing all $u \in V^\star$ is $O(n\bar{\sigma}_k \log \bar{\sigma}_k)$, where $\bar{\sigma}_k$ is the average number of k-hop neighbors size of the vertices in V. For k far lower than n, $O(n\bar{\sigma}_k \log \bar{\sigma}_k)$ is linear in n, due to the reasons explained at the end of Section 4.2.

5. QUERY ALGORITHM

Given vertices s, t in the data graph G, next we will explain how to obtain a k-skip SP from s to t using the k-skip graph $G^\star$ precomputed. Section 5.1 first gives an overview of our algorithm. A part of the algorithm needs to retrieve a traditional SP from $G^\star$, for which Section 5.2 presents an improved version of the *reach* method. Finally, Section 5.3 will discuss how to perform a zoom-in operation efficiently.

5.1 High-level description

Recall that the vertex set of $G^\star$ is a k-skip cover $V^\star$. The task of k-skip SP calculation is simple when both s and t are samples, namely, they belong to $V^\star$. In this case, we only need to find the

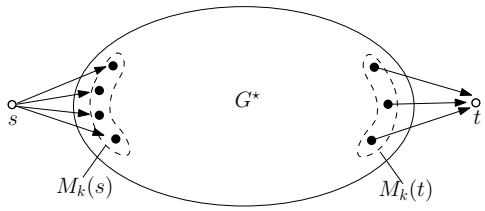


Figure 7: In-place sampling of s and t

SP from s to t in $G^\star$, that is, traveling only on the super-edges. By the definition of $G^\star$, this SP is guaranteed to be a k-skip SP in the original graph G.

Let us focus on the situation where neither s nor t is a sample. Our solution is to sample them into $G^\star$ *right away*, so that the case can be converted to the previous scenario where s and t are samples. The inclusion of s, t as samples is *temporary*; after query processing, they will be removed from $G^\star$, whose size therefore does not change.

Incorporation of s, t in $G^\star$ involves two steps. First, s and t are inserted in $V^\star$. Second, some super-edges are created to reflect the appearance of s, t, in the same way the existing super-edges are computed. That is, given s (similarly for t), we first obtain the super neighbor set $M_k(s)$ of s, and then add to $E^\star$ a super-edge from s to each $u \in M_k(s)$, all exactly as described in Section 4.3. This process is illustrated in Figure 7. The analysis of Section 4.3 shows that, the above strategy runs in $O(\sigma_s(k) \log \sigma_s(k) + \sigma_t(k) \log \sigma_t(k))$ time. Remember that $\sigma_s(k)$, the number of k-hop neighbors of s, is low when k is small (similarly for $\sigma_t(k)$). Hence, sampling-in-place s and t incurs insignificant overhead.

Let $G^\star_{s,t}$ be the resulting $G^\star$ with the new super-edges, and $SP^\star(s, t)$ be the SP from s to t on $G^\star$. The rest of our algorithm returns directly $SP^\star(s, t)$, whose correctness is shown in the lemma below:

LEMMA 5. *$SP^\star(s, t)$ is a k-skip SP of $SP(s, t)$.*

PROOF. Let u be the first sample (counting also t) when we walk from s along $SP(s, t)$. By the definition of k-skip SP, $SP(s, u)$ has at most k edges, all of which appear in the k-hop SP tree of s. Hence, $u \in M_k(s)$ (otherwise, there would be another sample on $SP(s, u)$, contradicting the choice of u), which means (s, u) is a super-edge in $G^\star_{s,t}$.

Let v be the last sample (counting also s) on $SP(s, t)$ before we arrive at t. A similar argument shows that (v, t) is also a super-edge. The correctness of the lemma then follows from the fact that every super-edge has a weight equal to the length of the path it represents. □

Remark on directed graphs. Let S be the set of super-edges on t newly computed for $G^\star_{s,t}$. In the above, we computed S by first finding the super neighbor set $M_k(t)$ of t, and then inserting a super-edge (u, t) for each $u \in M_k(t)$. If G is directed, the derivation of S is slightly different, in the sense that we need to first *reverse* the directions of the edges in G, before proceeding as described earlier. Intuitively, $M_k(t)$ should contain the samples that can reach t "directly" (without passing another sample). Reversing directions allows us to apply the same algorithm in Section 4.2 to extract $M_k(t)$, originally designed to find samples reachable *from* t directly. The reversing incurs no additional execution time, because a direction change can take place only when the relevant edge is touched by the algorithm.

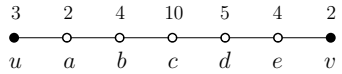


Figure 8: Super reach calculation

5.2 *Reach⋆*

Now that we have converted k-skip SP computation to finding $SP^\star(s, t)$ on $G^\star$ (in case s, t are not samples, simply treat $G^\star_{s,t}$ as $G^\star$), many SP algorithms such as *Dijkstra*, *bi-directional*, and *reach*, can be plugged in to obtain $SP^\star(s, t)$. However, as those methods are not designed for our context, they may be improved by taking into account the characteristics of $G^\star$. Next, we achieve the purpose for *reach*.

As discussed in Section 2.1, *reach* prunes a vertex v in relaxing an edge (u, v), if the global reach $r(v)$ of v is small, compared to the distance that the algorithm has traveled in the forward/backward search (see Rule 1). On a k-skip graph, using only $r(v)$ for pruning may miss plenty of pruning opportunities. The reason is that, a super-edge (u, v) implicitly captures a path in the original graph, and hence, can be pruned as long as *any vertex* on that path has a low global reach. Motivated by this, we formulate a new notion:

DEFINITION 6 (SUPER REACH). *Let (u, v) be a super-edge in $G^\star$, and P be the path in G that (u, v) represents. The **super reach** of (u, v), denoted as $sr(u, v)$, equals the minimum $h(w)$ of all the vertices $w \in P$, where*

$$h(w) = r(w) - \min\{\|P_1\|, \|P_2\|\} \quad (4)$$

where $r(w)$ is the global reach of w, and P_1 (P_2) is the path on P from u to w (w to v).

To illustrate, assume that the path P captured by a super-edge (u, v) is as shown in Figure 8. The number above each vertex is its global reach (e.g., $r(u) = 3$); and suppose for simplicity all the edges have weight 1. To decide, for example, $h(b)$, we first observe $\|P_1\| = 2$ and $\|P_2\| = 4$, and then calculate by Equation 4 $h(b) = 4 - \min\{2, 4\} = 2$. After $S = \{h(u), h(a), ..., h(e), h(v)\}$ is ready, the super reach $sr(u, v)$ can be determined as the minimum of S, i.e., $h(a) = 1$.

Apparently, $sr(u, v)$ can be computed in time linear to the number of vertices in P, i.e., at most k. Also, preserving all the super reaches entails small space, as only one extra value per super edge is stored. Next, we propose a new rule to enhance the pruning power of *reach*.

RULE 2. *When the forward search is about to relax a super edge (u, v), prune the edge if $sr(u, v) < l_f(u)$, where $l_f(u)$ is the label of u. A similar rule applies to the backward search.*

More precisely, pruning the super-edge (u, v) means that (i) the relaxation is not performed, and (ii) v is not en-heaped at this time (it is possible for v to get en-heaped due to another later relaxation though). Also note that the pruning happens *regardless of the status of v in the other direction* (unlike Rule 1 which requires v to have status `labeled` or `unseen` in the opposite search). This turns out to be a valuable property that permits the development of a crucial heuristic for maximizing efficiency, as discussed shortly.

Our algorithm, named *reach⋆*, for SP computation over $G^\star$ is identical to *bi-directional* (see Section 2.1), except that Rule 2 is

checked prior to every edge relaxation in an attempt to avoid the relaxation. The following theorem establishes the correctness of the algorithm.

THEOREM 2. *Reach* finds a SP on $G^\star$ correctly.*

PROOF. If the forward or backward search applies Rule 2 to prune a super-edge on $SP^\star(s,t)$, we say that a *blow* occurs. The lemma is obvious if no blow ever happens, so the subsequent discussion considers that there was at least one blow. Our argument proceeds in two steps. First, we will show that there can be only one blow during the execution of *reach**. This implies that every super-edge in $SP^\star(s,t)$ must be eventually relaxed in *at least one* direction, since two blows are needed to eliminate a super-edge from both directions. Equipped with these facts, we will prove the lemma in the second step.

Step 1. Without loss of generality, assume that the first blow occurred in the forward search, and eliminated super-edge $(u,v) \in SP^\star(s,t)$. It is easy to see that, when the blow happened, (i) the forward direction had scanned all the vertices in $SP^*(s,u)$, and (ii) $sr(u,v) < l_f(u)$ (by Rule 2), where $l_f(u)$ equals $\|SP^\star(s,u)\| = \|SP(s,u)\|$. Thus,

$$sr(u,v) < \|SP(s,u)\|. \tag{5}$$

Let P be the path in G that (u,v) represents, and w be the vertex in P that minimizes $h(w)$ (see Equation 4), i.e., $sr(u,v) = r(w) - \min\{\|SP(u,w)\|, \|SP(w,v)\|\}$. Hence,

$$r(w) - \|SP(u,w)\| \leq sr(u,v) \tag{6}$$

$$r(w) - \|SP(w,v)\| \leq sr(u,v). \tag{7}$$

Inequalities 5 and 6 lead to $r(w) < \|SP(s,u)\| + \|SP(u,w)\| = \|SP(s,w)\|$. By definition, $r(w)$ is at least $\min\{\|SP(s,w)\|, \|SP(w,t)\|\}$. So we know $r(w) \geq \|SP(w,t)\|$ which, together with Inequality 7, gives:

$$sr(u,v) \geq \|SP(w,t)\| - \|SP(w,v)\| = \|SP(v,t)\|. \tag{8}$$

Inequalities 5 and 8 indicate $\|SP(s,u)\| > \|SP(v,t)\|$. Due to (i) the way *bi-directional* synchronizes the two directions and (ii) the choice of (u,v), the backward search must have scanned all the vertices on $SP(v,t)$ before the blow happened.

If there was a second blow, either the forward search needed to de-heap a vertex in $SP(v,t)$, or the reverse search needed to de-heap a vertex in $SP(s,u)$. But both events would have terminated the algorithm immediately, because *bi-direction* ends when the forward/backward search de-heaps a vertex already scanned in the other direction. Therefore, no second blow could have occurred.

Step 2. The analysis of Step 1 shows that, at the moment the blow took place, the status of v was `scanned` in the backward search. This means that, u had a status of `labeled` in the backward direction. Consequently, when the forward search de-heaped u (right before the blow), as in *bi-directional*, *reach** updated λ, which records the length of the SP found so far, to $l_f(u) + l_b(u) = \|SP^\star(s,u)\| + \|SP^\star(u,t)\| = \|SP^\star(s,t)\|$. In other words, *reach** found the correct SP successfully. □

After the forward (or backward) search de-heaps a vertex u, our current *reach** attempts to prune each out-going super-edge at u with Rule 2. Hence, the rule has to be applied numerous times if many super-edges out of u can be eliminated. This can harm the efficiency because each vertex in $G^\star$ may have a large degree (unlike G, where each vertex's degree is bounded), the result of which is that we may end up applying the rule a huge number of times during the entire algorithm.

The next heuristic allows us to significantly reduce the cost, while still eliminating as many super-edges as before, with *no* increase in the space assumption at all. The idea is to store the outgoing super-edges of each vertex u in $G^\star$ in *descending* order of their super reaches. After de-heaping u in the algorithm, we attempt to prune those edges in the sorted order. The benefit is that once an edge has been eliminated by Rule 2, we can assert that all the remaining edges can be pruned as well, because all of them must have *lower* super reaches (than the one pruned), and therefore, will satisfy Rule 2 for sure.

The above heuristic is made possible by the fact that, to prune an edge (u,v), Rule 2 does *not* require checking the status of v (in the search opposite to the one where the pruning happens). If this was not the case, the heuristic would virtually promise no performance gain as checking the status of v takes nearly the same amount of time as applying Rule 2 on (u,v).

Remark. It is worth pointing out that, the mechanism behind *reach**, namely the integration of *bi-directional* and the no-status-checking pruning strategy of Rule 2, actually extends the algorithmic paradigm for SP computation as illustrated in Section 2.1. In retrospect, *reach** is an immediate benefit of this extension.

5.3 Zoom-in

As mentioned in Section 1, the concept of k-skip SP is naturally accompanied by a *zoom-in* operator. Given consecutive vertices u, v on a k-skip $SP^\star(s,t)$, the operator finds all the vertices between u and v on the full $SP(s,t)$. A naive way to zoom-in is to run *Dijkstra* to extract the SP from u to v. A faster solution applies *bi-directional*. In fact, one can do even better using *reach*. However, simply executing *reach* afresh to compute $SP(u,v)$ is not likely to outperform *bi-directional* much. This is because the pruning of *reach* (i.e., Rule 1 in Section 2.1) is effective only if the vertex u de-heaped in the, for example, forward search has a large label $l_f(u)$. This requires the forward search to have come a long way from the source, a situation that will not happen between u and v, because they are at most k vertices apart on their SP.

There is a simple remedy to significantly boost the efficiency. The main idea is to *pretend* as if we were running *reach* to compute $SP(s,t)$ (as opposed to $SP(u,v)$), and that the algorithm had just come to u and v in the forward and backward searches, respectively. We "continue" the forward direction by setting $l_f(u) = \min\{\|SP(s,u)\|, \|SP(v,t)\|\}$, and making u the only vertex in the heap Q_f (which implies giving u the status `labeled`). Note that both $\|SP(s,u)\|$ and $\|SP(v,t)\|$ are available in the k-skip $SP^\star(s,t)$ already calculated. Similarly, the backward direction is also continued by setting $l_b(v) = l_f(u)$ and creating a heap Q_b with only v inside. We then start a normal iteration and proceed as in *reach*.

6. EXPERIMENTS

In this section, we empirically evaluate the performance of the proposed solutions. Our experimentation used four spatial networks[2] whose specifications are summarized in Table 1. Specifically, NY, BAY, CA-NV, and USA contain the road networks in New York city, San Francisco Bay area, California and Nevada combined, and the entire US, respectively. The weight of an edge

[2]All datasets can be downloaded from *http://www.dis.uniroma1.it/∼challenge9/download.shtml*.

dataset	num. of vertices	num. of edges
NY	264,346	733,846
BAY	321,270	800,172
CA-NV	1,890,815	4,657,742
USA	23,947,347	58,333,344

Table 1: Dataset specifications

	k						
dataset	**4**	**6**	**8**	**10**	**12**	**14**	**16**
NY	51%	38%	31%	26%	22%	20%	18%
BAY	46%	33%	26%	22%	18%	16%	14%
CA-NV	46%	33%	21%	19%	16%	16%	15%
USA	45%	32%	25%	21%	18%	16%	15%

(a) Vertex ratio

dataset	4	6	8	10	12	14	16
NY	72%	65%	61%	58%	56%	53%	52%
BAY	66%	57%	52%	48%	45%	43%	42%
CA-NV	63%	54%	49%	45%	43%	41%	40%
USA	61%	51%	47%	44%	41%	39%	38%

(b) Edge ratio

Table 2: Sizes of k-skip graphs

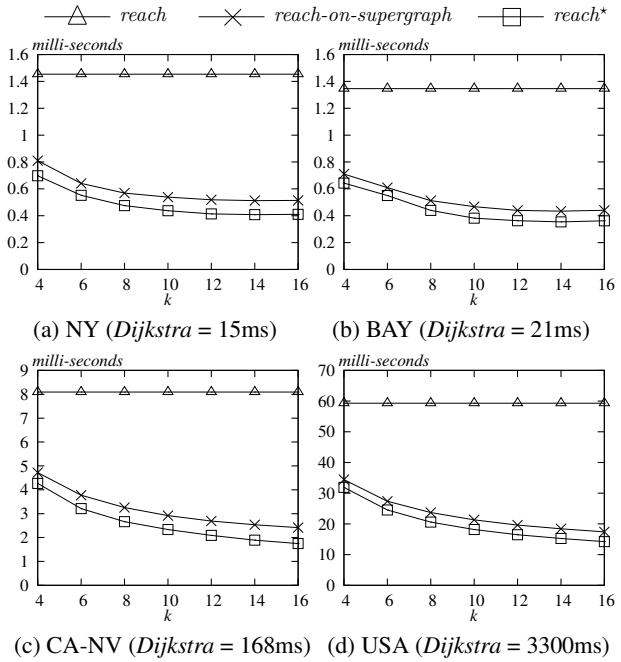
equals the travel time on the corresponding road segment. All of our results were obtained on a computer equipped with an Intel Core 2 DUO 3.0Ghz CPU and 2 Giga bytes memory, running Fedora Linux 13.

Size of the pre-computed structure. Our technique has the feature of demanding only a structure with *sub-linear* size, i.e., a k-skip graph $G^\star = (V^\star, E^\star)$ occupies *less space* than the underlying road network $G = (V, E)$. The first set of experiments demonstrates this by proving that $G^\star$ has fewer vertices and edges than G. Equivalently, if we define the *vertex ratio* to be $|V^\star|/|V|$ and the *edge ratio* to be $|E^\star|/|E|$, the goal is to show that both ratios are below 1 by a comfortable margin. Moreover, remember that $V^\star$ is a k-skip cover (Definition 3). Hence, the vertex ratio also reflects the effectiveness of the *AS* algorithm in Section 4.2.

Table 2a presents the vertex ratios of each dataset as k varies from 4 to 16. Interestingly, we noticed that the ratio roughly equals $2/k$ in all cases, that is, *AS* finds a k-skip cover with size about $2|V|/k$. In the same style, Table 2b shows the corresponding edge ratios, which are also much lower than 1, and decrease with the increase of k. A general observation is that, both ratios tend to be smaller (i.e., greater size reduction) when the underlying network is sparser (NY is the densest among all the datasets).

Query characteristics. Ideally, a k-skip SP query should be answered much faster than its traditional counterpart (that retrieves the whole path). Next, we compare the *reach** algorithm developed in Section 5.2 against *reach*, which is the state of the art for the (traditional) SP problem, as reviewed in Section 2.1. We also examined a method *reach-on-supergraph* that can be regarded as a compromise of the two algorithms. As mentioned in Section 5.2, any SP algorithm can be applied to produce a k-skip SP by finding a SP on the k-skip graph $G^\star$. Following this rationale, *reach-on-supergraph* is a k-skip solution that simply employs *reach* on $G^\star$. As a reference, we also report the performance of *Dijkstra*.

The cost of each method is measured as its average response time in processing a *workload* of 1000 queries whose source and destination are both randomly selected from the vertices of the original network G. Figures 9a-9d plot the costs of alternative solutions as a function of k for the four datasets, respectively. The overhead of



(a) NY (*Dijkstra* = 15ms) (b) BAY (*Dijkstra* = 21ms)

(c) CA-NV (*Dijkstra* = 168ms) (d) USA (*Dijkstra* = 3300ms)

Figure 9: Cost of k-skip and traditional SP queries

Dijkstra is given separately beside the dataset name, because it is substantially slower than all other competitors.

Both k-skip algorithms *reach** and *reach-on-supergraph* outperform *reach* significantly. Furthermore, their performance gains over *reach* become increasingly obvious as k grows larger (*reach* is irrelevant to k as it always reports the entire SP). This, therefore, validates k-skip graph as an effective methodology to attack the k-skip SP problem. Between the two k-skip algorithms, the consistent superiority of *reach** ascertains the necessity of designing a new algorithm that is able to leverage the characteristics of k-skip graphs. Remember that *reach** gleans its performance advantages at very little space overhead, because only one extra value needs to be stored for each edge in $G^\star$.

For the subsequent discussion, we define the *SP-cardinality* of query as the number of vertices in its complete SP. Figure 9 does not shed much light on each method's behavior in handling queries with different SP-cardinalities. Figure 10 remedies the drawback with a detailed version of the results for $k = 10$ (*Dijkstra* is omitted due to its poor efficiency). To obtain Figure 10a, we first decided the range ρ of the SP-cardinalities of all the queries in a workload. Then, for each algorithm, we measured its average cost in answering the queries whose SP-cardinalities were among the first 10% of ρ, second 10%, ..., and the last 10%, respectively. The collection of these 10 values correspond to the dots on a curve in Figure 10a. The other diagrams in Figure 10 were acquired in the same manner. The relative performance of all algorithms remains stable throughout the whole spectrum of SP- cardinality. The bell-shape of each curve is a characteristic of the *reach* algorithm (and hence, also its variant *reach**).

The next experiment studies the number of vertices in a k-skip SP. We present the number as a percentage of the query's SP-cardinality (e.g., 50% means that the k-skip path contains half of the vertices in the entire SP), which is referred to as the *output percentage*. For each dataset, Table 3 reports the average output percentages of a workload under different values of k. As expected,

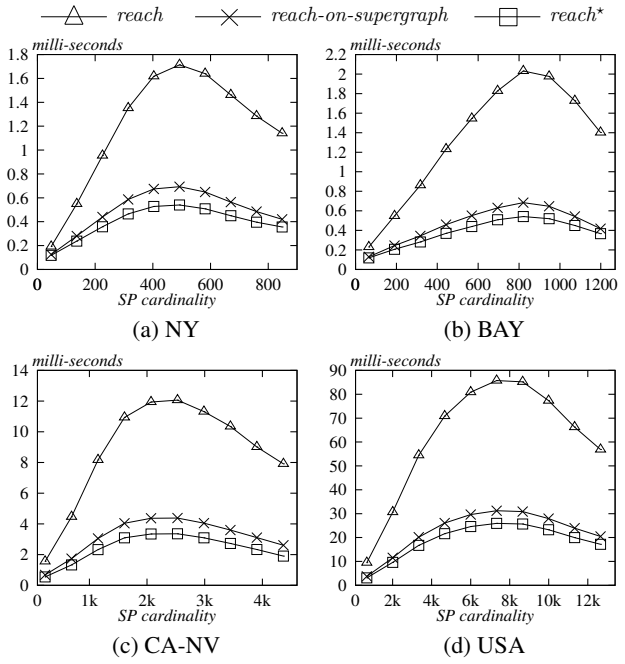


Figure 10: Query cost vs. the number of vertices in the complete SP ($k = 10$)

	k						
dataset	**4**	**6**	**8**	**10**	**12**	**14**	**16**
NY	55%	41%	34%	29%	27%	24%	22%
BAY	52%	36%	29%	25%	22%	20%	18%
CA-NV	52%	36%	29%	25%	22%	21%	19%
USA	51%	35%	28%	24%	22%	20%	19%

Table 3: Number of vertices in a k-skip SP (in percentage of the SP-cardinality)

the percentage decreases as k increases. It is not hard to observe a strong correlation between the output percentages and the vertex ratios in Table 2.

We proceed to assess the efficiency of the *zoom-in* algorithm in Section 5.3. The k-skip SP $P^\star$ of a query naturally defines $l - 1$ possible zoom-in operations if $P^\star$ has l vertices. For each road network and a particular k, we randomly performed 10% of all the zoom-in's determined by 50,000 queries with random sources and destinations, and gauged their average time. Figure 11 illustrates the results of all datasets. Note that the y-axis is in μ-seconds (1μs = 10^{-6} sec). In general, for $k \leq 16$, the cost of a zoom-in is at the order of 10μs, and is thus negligible in practice.

Pre-computation overhead. The last set of experiments examines the time of constructing a k-skip graph $G^\star$. Given an input graph G with all vertices' reaches already computed (the time of reach computation has been thoroughly evaluated in [10, 12]), the cost of building $G^\star$ has two components: the time of (i) finding a k-skip cover $V^\star$ (Section 4.2), and (ii) determining the super-edges of $G^\star$ (Section 4.3). For each road network, Figure 12 reports the overhead of these components as a function of k. The overall pre-processing time is dominated by the cost of (i) because it has to process all vertices in G, whereas (ii) only needs to process the vertices of $G^\star$. Consulting the specifications in Table 1, we observe that the pre-processing overhead increases linearly with the number of edges in the original graph. For example, for the same k, the

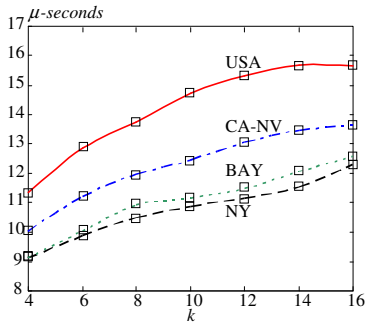


Figure 11: Zoom-in time vs. k

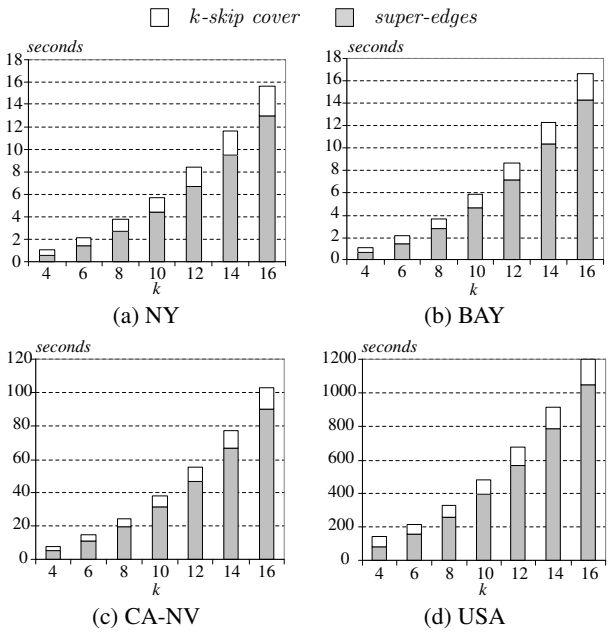


Figure 12: Pre-processing cost vs. k

overhead on USA is about 12 times that of CA-NV, where 12 is roughly the ratio between the numbers of edges in USA and CA-NV.

7. CONCLUSIONS

This paper introduced a novel concept called *k-skip shortest path*, which is a subset of the vertices in a traditional SP P, subject to the constraint that the subset must include at least one vertex in every k consecutive vertices of P. We carried out a systematic study on the problem of k-skip SP computation, where the goal is to pre-process a road network into a structure of economical size such that a k-skip query on the original network can be processed significantly faster than finding the entire SP. We settled the problem with a new methodology that constructs a *k-skip graph* from a set of selected vertices of G constituting a *k-skip cover*. The efficiency of the proposed technique has been verified with extensive experiments.

Our work also points to several promising directions for future research on the k-skip SP problem. One, for example, is to design a deterministic algorithm for finding the *minimum* k-skip cover, which does not appear to be trivial at all even for $k = 2$. Another interesting problem is to prove (possibly by resorting to the

highway dimension [1]) a theoretical upper bound on the number of super-edges in a k-skip graph, which will immediately lead to a bound on the overall space consumption of our technique. Finally, it is certainly an important direction to explore the semantics of k-skip SP as well as the corresponding algorithms on other types of graphs, such as social networks.

Acknowledgements

Yufei Tao and Cheng Sheng were supported by grants GRF 4173/08, GRF 4169/09, and GRF 4166/10 from HKRGC. Jian Pei was supported by NSERC Discovery grants, NSERC Discovery Accelerator Supplement grants, and NRAS Research Team Program. We thank the anonymous reviewers for their constructive suggestions on improving the paper.

8. REFERENCES

[1] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. F. Werneck. Highway dimension, shortest paths, and provably efficient algorithms. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 782–793, 2010.

[2] R. Agrawal and H. V. Jagadish. Algorithms for searching massive graphs. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 6(2):225–238, 1994.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[4] K. Deng, X. Zhou, H. T. Shen, S. W. Sadiq, and X. Li. Instance optimal query processing in spatial networks. *The VLDB Journal*, 18(3):675–693, 2009.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] R. Durstenfeld. Algorithm 235: Random permutation. *Communications of the ACM (CACM)*, 7(7):420, 1964.

[7] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal of Computing*, 16(6):1004–1022, 1987.

[8] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of Workshop on Experimental Algorithms (WEA)*, pages 319–333, 2008.

[9] A. V. Goldberg and C. Harrelson. Computing the shortest path: A^* search meets graph theory. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 156–165, 2005.

[10] A. V. Goldberg, H. Kaplan, and R. F. Werneck. Reach for A^*: Shortest path algorithms with preprocessing. In *Proceedings of Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 38–51, 2006.

[11] A. Gubichev, S. J. Bedathur, S. Seufert, and G. Weikum. Fast and accurate estimation of shortest paths in large graphs. In *Proceedings of Conference on Information and Knowledge Management (CIKM)*, pages 499–508, 2010.

[12] R. J. Gutman. Reach-based routing: A new approach to shortest path algorithms optimized for road networks. In *Proceedings of Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 100–111, 2004.

[13] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.

[14] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete & Computational Geometry*, 2:127–151, 1987.

[15] D. A. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. *Discrete Applied Mathematics*, 126(1):55–82, 2003.

[16] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical encoded path views for path query processing: An optimal model and its performance evaluation. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 10(3):409–432, 1998.

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

[18] M. R. Kolahdouzan and C. Shahabi. Alternative solutions for continuous k nearest neighbor queries in spatial network databases. *GeoInformatica*, 9(4):321–341, 2005.

[19] U. Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, 22:219–230, 2004.

[20] T. A. J. Nicholson. Finding the shortest route between two points in a network. *The Computer Journal*, 9:275–280, 1966.

[21] M. Potamias, F. Bonchi, C. Castillo, and A. Gionis. Fast shortest path distance estimation in large networks. In *Proceedings of Conference on Information and Knowledge Management (CIKM)*, pages 867–876, 2009.

[22] M. N. Rice and V. J. Tsotras. Graph indexing of road networks for shortest path queries with label restrictions. *Proceedings of the VLDB Endowment (PVLDB)*, 4(2):69–80, 2010.

[23] N. Robertson and P. D. Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[24] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 43–54, 2008.

[25] P. Sanders and D. Schultes. Highway hierarchies hasten exact shortest path queries. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 568–579, 2005.

[26] P. Sanders and D. Schultes. Engineering highway hierarchies. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 804–816, 2006.

[27] J. Sankaranarayanan, H. Samet, and H. Alborzi. Path oracles for spatial networks. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):1210–1221, 2009.

[28] D. Wagner, T. Willhalm, and C. D. Zaroliagis. Geometric containers for efficient shortest-path computation. *ACM Journal of Experimental Algorithmics*, 10, 2005.

[29] F. Wei. Tedi: Efficient shortest path query answering on graphs. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 99–110, 2010.

[30] Y. Xiao, W. Wu, J. Pei, W. Wang, and Z. He. Efficiently indexing shortest paths by exploiting symmetry in graphs. In *Proceedings of Extending Database Technology (EDBT)*, pages 493–504, 2009.