Fast Nearest Neighbor Search on Road Networks*

Haibo Hu[1], Dik Lun Lee[1], and Jianliang Xu[2]

[1] Hong Kong Univ. of Science & Technology
{haibo, dlee}@cs.ust.hk
[2] Hong Kong Baptist University
xujl@comp.hkbu.edu.hk

Abstract. Nearest neighbor (NN) queries have been extended from Euclidean spaces to road networks. Existing approaches are either based on Dijkstra-like network expansion or NN/distance precomputation. The former may cause an explosive number of node accesses for sparse datasets because all nodes closer than the NN to the query must be visited. The latter, e.g., the Voronoi Network Nearest Neighbor (VN^3) approach, can handle sparse datasets but is inappropriate for medium and dense datasets due to its high precomputation and storage overhead. In this paper, we propose a new approach that indexes the network topology based on a novel network reduction technique. It simplifies the network by replacing the graph topology with a set of interconnected tree-based structures called SPIE's. An *nd* index is developed for each SPIE and our new (k)NN search algorithms on an SPIE follow a predetermined tree path to avoid costly network expansion. By mathematical analysis and experimental results, our new approach is shown to be efficient and robust for various network topologies and data distributions.

1 Introduction

Nearest neighbor (NN) search has received intensive attention in spatial database community in the past decade, especially in high-dimensional Euclidean spaces [10, 1, 13, 15]. Recently, the research focus is brought to spatial network databases (SNDB) where objects are restricted to move on predefined roads [11, 6, 9, 8]. In SNDB, a road network is modeled as a graph G ($< V, E >$), where a vertex (node) denotes a road junction and an edge denotes the road between two junctions; and the weight of the edge denotes the network distance. A nearest neighbor query on the road network is, given a query node q and a dataset (e.g., restaurants, gas stations) distributed on the nodes V, to find a data object that is the closest to q in terms of network distance.

Existing research falls into two categories. In the first category, NN search expands from the query node to adjacent nodes until a data object is found and further expansion cannot retrieve closer objects [6, 9]. Such network expansion originates from Dijkstra's algorithm that finds single-source shortest paths. The

---

advantage of this approach is that the network distance, the key to NN search, is automatically obtained during the expansion. However, the disadvantage is that the "unguided graph traversal" during network expansion may cause an explosive number of node accesses, especially for sparse datasets. In the second category, solution-based indexes are built on the datasets. Kolahdouzan et al. proposed VN^3 to partition the network into cells by the *Network Voronoi Diagram* (NVD) [8]. Each cell contains one data object that is the closest object to all the nodes in this cell. These cells are indexed by an R-tree in the Euclidean space, and thus finding the first NN is reduced to a point location problem. To answer k-nearest-neighbor (kNN) queries, they showed that the kth NN must be adjacent to some ith NN $(i < k)$ in the NVD. To speed up distance computation, they also precompute the distances between border points of adjacent cells. However, their approach is advantageous only for sparse datasets and small/medium k. Furthermore, if more than one dataset exists, NVD indexes and precomputed distances must be built and maintained separately for each dataset.

In this paper, we take a new approach by indexing the network topology, because compared with the datasets the topology is unique and less likely to change. To reduce index complexity and hence avoid unnecessary network expansion, we propose a novel technique called *network reduction* on road networks. This is achieved by replacing the network topology with a set of interconnected tree-based structures (called SPIE's) while preserving all the network distances. By building a lightweight nd index on each SPIE, the (k)NN search on these structures simply follows a predetermined path, i.e., the tree path, and network expansion only occurs when the search crosses SPIE boundaries. By analytical and empirical results, this approach is shown to be efficient and robust for road networks with various topologies, datasets with various densities, and kNN queries with various k. Our contributions are summarized as follows:

- We propose a topology-based index scheme for kNN search on road networks. To reduce the index complexity, a network reduction technique is developed to simplify the graph topology by tree-based structures, called SPIE's.
- We propose a lightweight nd index for the SPIE so that the (k)NN search in SPIE follows a predetermined tree path. With this index, the whole (k)NN search can avoid most of the costly network expansions.
- We develop cost models for the network reduction, NN and kNN search by our nd-based algorithms. These cost models, together with experimental results, show the efficiency of our approach and the performance impact of various parameters.

The rest of this paper is organized as follows. Section 2 reviews existing work of (k)NN search on SNDB. Section 3 presents the network reduction technique. Section 4 introduces the nd index on the SPIE and NN search algorithms on the reduced network. The algorithms are extended to kNN search in Section 5. Section 6 develops the cost models, followed by the performance evaluation in Section 7. Finally, Section 8 concludes the paper.

2   Related Work

Nearest neighbor (NN) search on road networks is an emerging research topic in recent years [11, 6, 9, 8]. It is closely related to the single-source shortest path problem, which has been studied since Dijkstra [4]. He proposed to use a priority queue to store those nodes whose adjacent nodes are to be explored. Besides the Dijkstra algorithm, A^* algorithm with various expansion heuristics was also adapted to solve this problem [5].

Among database researchers, Jensen et al. brought out the notion of NN search on road networks [6]. They proposed a general spatio-temporal framework for NN queries with both graph representation and detailed search algorithms. To compute network distances, they adapted the Dijkstra's algorithm to online evaluate the shortest path. Papadias et al. incorporated the Euclidean space into the road network and applied traditional spatial access methods to the NN search [9]. Assuming that Euclidean distance is the lower bound of network distance, they proposed *incremental Euclidean restriction* (IER) to search for NNs in the Euclidean space as candidates and then to compute their network distances to the query node for the actual NNs. However, IER cannot be applied to road networks where that distance bound does not hold, e.g., the network of transportation time cost. Although they proposed an alternative approach *incremental network expansion* (INE), it is essentially a graph traversal from the query point and thus performs poorly for sparse datasets.

Inspired by the Voronoi Diagram in vector spaces, Kolahdouzan et al. proposed a solution-based approach for kNN queries in SNDB, called *Voronoi Network Nearest Neighbor* (VN^3) [8]. They precompute the Network Voronoi Diagram (NVD) and approximate each Voronoi cell by a polygon called Network Voronoi Polygon (NVP). By indexing all NVP's with an R-tree, searching the first nearest neighbor is reduced to a point location problem. To answer kNN queries, they prove that the kth NN must be adjacent to some ith $(i < k)$ NN in NVD, which limits the search area. To compute network distances for an NN candidate, they precompute and store the distances between border nodes of adjacent NVP's, and even the distances between border nodes and inner nodes in each NVP. By these indexes and distances, they showed that VN^3 outperforms INE, by up to an order of magnitude. However, VN^3 heavily depends on the density and distribution of the dataset: as the dataset gets denser, both the number of NVP's and the number of border points increase, causing higher precomputation overhead and worse search performance. Given that NN search by network expansion on dense datasets is efficient, VN^3 is only useful for sparse datasets.

Shahabi et al. applied graph embedding techniques to kNN search on road networks [11]. They transformed a road network to a high-dimensional Euclidean space where traditional NN search algorithms can be applied. They showed that KNN in the embedding space is a good approximation of the KNN in the road network. However, this technique involves high-dimensional (40-256) spatial indexes, which leads to poor performance. Further, the query result is approximate and the precision heavily depends on the data density and distribution.

Continuous nearest neighbor (CNN) query is also studied recently. Besides an efficient solution for NN query, CNN query on road network also requires to efficiently determine the network positions where the NN(s) change. Various approaches such as UBA [7], UNICONS [2] are proposed to solve this problem.

3   Reduction on Road Networks

The objectives for network reduction are: (1) to reduce the number of edges while preserving all network distances, and (2) to replace the complex graph topology with simpler structures such as trees. To achieve the objectives, we propose to use the *shortest path trees* (SPT). The basic idea is to start from a node (called *root*) in the road network G and then to grow a shortest path tree from it by the Dijkstra's algorithm. During the execution, when a new node n is added to the tree, we additionally check if its distances in G to all the other tree nodes are preserved by the tree. This is completed by checking if there is any edge adjacent to n in G that connects n to a tree node closer than the tree path. Such an edge is called a *shortcut*. If n has no shortcuts, it is inserted to the tree as in Dijkstra's algorithm; otherwise n becomes a new root and a new SPT starts to grow from it. The new SPT connects with some existing SPT's through the shortcuts of n. The whole process continues until the SPT's cover all nodes in network G. These SPT's form a graph— called an *SPT graph*— whose edges are the shortcuts from the root of an SPT to some node in another SPT. Figure 1 illustrates an SPT graph. Obviously the SPT graph is much simpler than the graph of road network. It is noteworthy that the reduction from a graph to a set
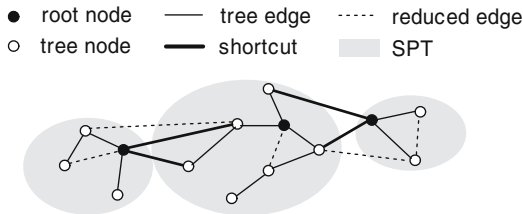

Fig. 1. An Example of Shortest Path Tree Graph

of interconnected trees is not generally beneficial. Nonetheless, road networks exhibit the following two properties that justify this approach: (1) the degree of a junction in the road network is normally equal to or greater than 3, some junctions serving as "hubs" of the network may have even higher degrees; (2) the *girth* of the network, i.e., the length of the shortest circuit in the network, is long, because small circuit means redundant paths between close-by nodes, which is normally avoided in network design. We show in the cost model (in Section 6.2) that these two properties lead to effective reduction on road networks.

In order to further reduce the shortcuts and thus the number of SPT's, we augment the SPT's to allow sibling-to-sibling edges. These edges are called *horizontal edges* and the SPT's that support such edges are called **s**hortest **p**ath *tree with* **h***orizontal edges* (SPH). With horizontal edges, SPH's can store shortcuts between siblings and thus no new SPH needs to be created in such cases. Later in Section 4.2 we will prove that SPH still forms a hierarchy and the shortest path between any two nodes is still easy to allocate.

Algorithm 1. Network Reduction by SPH

Input: a network G and a starting root r
Output: an SPH graph Γ
Procedure:

 1: *starting_node* $= r$;
 2: **while** there is node in G that is not covered in any SPH of Γ **do**
 3:     build a blank SPH T and insert T as a vertex into Γ;
 4:     insert all the non-sibling shortcuts of *starting_node* as edges to Γ;
 5:     build a blank priority queue H and insert $< starting\_node, 0 >$ to H;
 6:     **while** H is not empty **do**
 7:         node $n = H.pop()$;
 8:         **if** n has no shortcuts to non-sibling tree nodes **then**
 9:             insert n into T;
10:         **else**
11:             break;
12:         relax the distances in H according to Dijkstra's algorithm;

Algorithm 1 shows the procedure of network reduction. The inner *while* loop is modified from the Dijkstra's algorithm to build an individual SPH. Different from Dijkstra's algorithm, the inner loop stops whenever there are shortcuts to non-sibling tree nodes and then a new SPH starts to grow from this node. These shortcuts are stored as the edges in the SPH graph Γ.

4   Nearest Neighbor Search on SPH Graph

In this section, we present our solution to NN search on the reduced network, i.e., the SPH graph. The search starts from the SPH where the query node is located. By building a local index nd on each SPH, this search is efficient. Searching into the adjacent SPH's in the graph continues until the distance to the SPH's already exceeds the distance to the candidate NN. In what follows, we first show the nd-based NN search on a tree. Then we extend it to the SPH and the SPH graph. Finally we present the index construction and maintenance algorithms. The extension to kNN search is shown in the next section.

4.1   NN Search on Tree

We begin with the NN search on a tree. To avoid network expansion that recursively explores the parent and child nodes from the current searching node, we
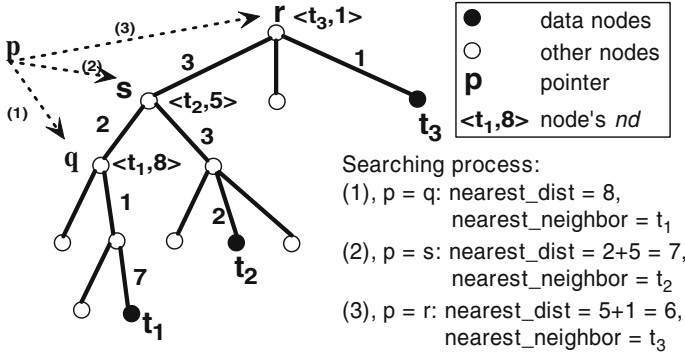
Fig. 2. Nearest Neighbor Search on Tree

store, for each node v, the nearest data object in its descendants (*nearest descendant* or *nd* for short). The object is denoted by $v.nd.object$ and its distance to v is denoted by $v.nd.dist$. For example, in Figure 2, $s.nd$ is set to $< t_2, 5 >$, as the nd of s is t_2 which is 5 units away. If a node have no data object in its descendants, its nd is set to $<null, \infty>$.

The pointer to the current searching node, p, starts from the query node q. Based on the nd index, if $p.nd$ is closer to q than the current NN, $p.nd$ becomes the new NN and the current nearest distance, $nearest\_dist$, is updated. Then p proceeds to q's parent, grandparent, $\cdots$, etc., until the distance between p and q exceeds $nearest\_dist$ or p reaches the root. Figure 2 shows an example where p starts at q and then moves to s and r, until it finds the NN. With the nd index, the search path is at most as long as the tree path to the root. Therefore the number of node accesses is bounded by the height of the tree. In the next subsections, we extend the nd index to the SPT and SPT graph.

4.2 SPIE: SPH with Triangular Inequality

An SPH is more complicated than a tree because there are multiple paths from the source to the destination. In this subsection, our objective is to modify the SPH obtained from Section 3 so that the weight of each edge (tree edge or horizontal edge) represents the shortest distance between the two adjacent nodes. In other words, we modify the SPH to satisfy the *triangular inequality*, that is, $\forall$ three edges ab, bc, ac $\in SPH.E$, $w(ac) \leq w(ab) + w(bc)$. The modified SPH is called an SPH with triangular **i**nequality **e**dges (SPIE).

The conversion from an SPH into an SPIE is a local operation. For each node u, we obtain its child nodes; the set of tree edges and horizontal edges between these nodes forms a weighted graph. We perform the *Floyd-Warshall* algorithm [3] to find all-pairs shortest paths in this graph. The distances of these shortest paths form the weights of tree edges and horizontal edges in the SPIE. The following theorem proves that SPIE guarantees that the shortest path of any two nodes u and v comprises one and only one horizontal edge which connects one of u's ancestors and one of v's ancestors.

Theorem 1. *For two nodes u and v in SPIE that are not descendant/ancestor of each other, their shortest path consists of the following nodes sequentially, $u_0, u_1, u_2, \cdots, u_s, v_t, \cdots, v_2, v_1, v_0$, where $u_0 = u$, $v_0 = v$, and u_i (v_i) is the parent of u_{i-1} (v_{i-1}); u_s and v_t are the child nodes of $lca_{u,v}$, the lowest common ancestor of u and v.*

PROOF. In order to prove the theorem, we first introduce Lemma 1.

Lemma 1. *Any path from node u to its descendant v in an SPIE must include all the tree edges from u to v. In other words, v's parent, grandparent, $\cdots$, till u, must exist in any path from u to v.*

PROOF. Let $level(i)$ denote the depth of node i ($level(root) = 0$), and n denote $level(v) - level(u) - 1$. By mathematical induction,

1. For $n = 1$, v's parent node must be included in the path because otherwise there are more than one parent for node v, which is prohibited in an SPIE;
2. Assume the lemma holds for $n = k$. Thus for $n = k+1$, we only need to prove t, u's child and v's ancestor, is included in the path. By the assumption, all ancestors of v that are below t are already in the path, especially s, t's child and v's ancestor. Since s is in the path, by the same reasoning as in 1, t must be in the path.

Hereby, 1 and 2 complete the proof.   □

We now prove the theorem. Let p denote $lca_{u,v}$ for simplicity.

1. First, we prove that if all sibling edges among p's children are removed, p must exist in $path(u,v)$. Consider the two subtrees that are rooted at p's two children and contain u and v respectively. Since the only edges linking them with the rest of the SPIE are the two tree edges adjacent to p, p must exist in any path between the two subtrees. Thus, p must exist in $path(u,v)$.
2. From Lemma 1, $u_1, u_2, \cdots, u_s$ must exist in $path(u,v)$. We only need to prove that they are the only nodes in the path.[1] By contradiction, if there were one node x between u_i and u_{i+1}, x must be a sibling node of u_i. However, since all edge weights satisfy triangular inequality, i.e., $w(u_i, u_{i+1}) \leq w(u_i, x) + w(x, u_{i+1})$, removing node x results in an even shorter path, which contradicts the shortest path condition. Therefore, $u_1, u_2, \cdots, u_s$ are the only nodes in the path.
3. Finally we prove that when adding back the sibling edges removed in 1, the path is the same except that p is removed from $path(u,v)$. On the one hand, due to triangular inequality, $w(u_s, v_t) \leq w(u_s, p) + w(p, v_t)$, so p should be removed from the shortest path. On the other hand, since all added edges are sibling edges, if any new node is to be added to the path, only sibling nodes are possible choices; but from 2, adding sibling nodes only increases the path distance. Therefore, no nodes should be added.

Hereby, 1, 2 and 3 complete the proof.   □

---

[1] By symmetry, the proof is the same for the $v_1, v_2, \cdots, v_t$, and hence omitted.

4.3   NN Search on SPIE and SPIE Graph

By Theorem 1, a shortest path in an SPIE is the same as that in a tree except that a horizontal edge replaces two tree edges adjacent to the lowest common ancestor. Therefore, NN search in SPIE still starts from the query node q and moves upward to its ancestors. The only difference is that, instead of p's nd, the nd's of p's child nodes (except for the node pointed by the last p), are examined during the search. This is because if p is the lowest common ancestor of q and some possible NN, according to Theorem 1, one of p's children, instead of p, appears in the path. Figure 3 illustrates the NN search on an SPIE. In this example, when $p = s$, the nd of u, instead of s, is examined.   Regarding
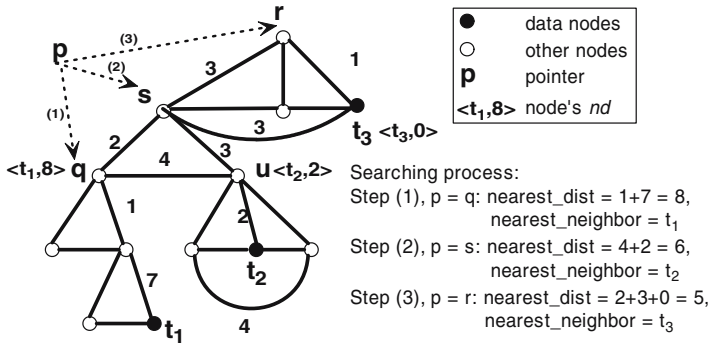

Fig. 3. NN search on SPIE

NN search on the SPIE graph, once the search is completed on the current SPIE, its shortcuts need to be considered. More specifically, the search should be propagated to those nodes that are adjacent to the shortcuts and are closer to q than the current NN. With these nodes being query points, new NN searches on the SPIE's are started. These searches are processed similarly except that the distance is accumulated from q. Algorithm 2 shows the pseudo-code of the NN search on one SPIE. For NN search on the SPIE graph, this algorithm is invoked with the SPIE that contains q.

4.4   *nd* Index Construction

The nd index is independently constructed on each SPIE. As aforementioned, the nd data structure of each node n stores both the nearest descendant and its shortest distance to n. In addition, based on the fact that the nearest descendant of n is also the nearest descendant of all nodes along the path from $n.nd$ to n, $n.nd$ also stores the child node of n in this path to record the path to the nearest descendant. To build the nd index, a bottom-up fashion is applied: the nd's of n's children are built and then the nearest nd among them is elected as the nd for n. Algorithm 3 shows the pseudo-code of the bottom-up nd construction.

Algorithm 2. NN Search on an SPIE

Input: an SPIE Γ, a query point q, accumulated distance D from the global query point

Global: the candidate NN r, also the output when the entire search terminates

Procedure: NN_search_on_SPIE(Γ,q,D)

1: $p = q$;
2: **while** $dist_{p,q} < dist_{r,q}$ **do**
3:    find the best NN object u in p's child nodes's nd;
4:    **if** u is better than r **then**
5:       update r;
6:    $p = p.parent$;
7: **for** each shortcut s, t ($s \in \Gamma, t \in \Phi$) **do**
8:    **if** $D + dist_{q,t} < dist_{r,q}$ **then**
9:       NN_search_on_SPIE(Φ,t,$D + dist_{q,t}$);

Algorithm 3. Build nd index on an SPIE

Input: an SPIE Γ, a node p

Operation: Build p's nd recursively

Procedure: build_nd(Γ,p)

1: **if** p is a data object **then**
2:    set $p.nd = p$;
3: **else if** p is a leaf node **then**
4:    set $p.nd = null$;
5: **else**
6:    **for** each p's child v **do**
7:       build_nd(Γ,v);
8:    find the nearest descendant v^* among p's child nodes' nd;
9:    set $p.nd = v^*$;

Regarding disk paging, the nd index is paged in a top-down manner [14]: starting from the root, the SPIE is traversed in a breadth-first order, where nd structure is greedily stored in a disk page until it is full. The breadth-first traversal guarantees that topologically close nodes are physically close on disk.

4.5 Handling Updates

This subsection copes with updates on both network topology and data objects.

Updates on Network Topology. Network updates include the insertion/deletion of nodes, insertion/deletion of edges, and change of edge weights.

- **node insertion:** the node is inserted to the SPIE that contains the adjacent node.
- **node deletion:** only the SPIE that contains this node needs to be rebuilt by Dijkstra's algorithm[2].

---

[2] If the SPHIE is no longer connected, the SPIE is split.

- **edge insertion:** if the edge is an intra-SPIE edge and provides a shorter distance between the two adjacent nodes, only this SPIE is rebuilt by Dijkstra's algorithm; otherwise if the edge is a shortcut, it is added to the SPIE graph, otherwise no operation is needed.
- **edge deletion:** if the edge is an intra-SPIE edge, this SPIE is rebuilt; otherwise if it is a shortcut, it is removed from the SPIE graph; otherwise no operation is needed.
- **increase edge weight:** same as *edge deletion*.
- **decrease edge weight:** same as *edge insertion*.

Updates on Data Objects. Updates on data objects include object insertion/deletion. These changes affect the nd index only; the SPIE graph is not affected. Therefore, data objects updates are less costly than network updates. Moreover, the inserted/deleted object only affects the nd index of this node and its ancestors in the SPIE. So the index update starts from the node where the object insertion/deletion occurs, and repeatedly propagates to the parent until the nd no longer changes.

5   K-Nearest-Neighbor Search

To generalize NN search to KNN search, every time p points at a new node, we not only examine the nd of p (or more precisely the nd's of p's children), but also search downwards to examine the nd of p's descendants for candidate NN farther than $p.nd$. The downward search terminates when all (or k, whichever is smaller) data objects in p's descendants are found, or when the accumulated distance from q exceeds the kth NN candidate distance from q. During the downward search, a priority queue L is used to store the nodes to be examined, sorted by their accumulated distances from q.

Figure 4 shows an example of a 2NN search on the same SPIE as in Figure 3. r denotes the current set of 2NN candidates, where $< t_1, 8 >$ means a candidate t_1 is 8 units from the query node q. In priority queue L, $< x, 1 >$ means that the nd of node x that is 1 unit from q is to be examined. Every time p moves upwards to a new node (e.g., s), the priority queue L is initialized with nd of p's children (e.g., u). Then we repeatedly pop up the first node from L, examine its nd, and push its children to L until L is empty, or two objects have been found, or the accumulated distance exceeds the second NN distance to q. Afterwards p moves upwards to its parent and the same procedure is repeated. The entire NN search terminates, as in Algorithm 2, when the distance from p to q already exceeds that from the k-th NN candidate to q.

6   Cost Models

In this section, we analyze the effectiveness of our proposed network reduction and nd-based nearest neighbor search algorithms. We develop cost models for the number of edges removed during the reduction and nodes accesses (NA)
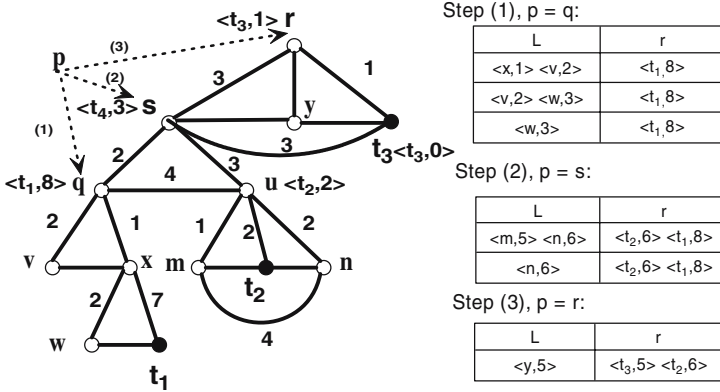
Fig. 4. KNN search on SPIE

during the NN and kNN search. We then compare the latter with the number of nodes accesses by the naive Dijkstra-like network expansion algorithm. Before we start the analysis, we first introduce some assumptions and notations.

6.1   Analytical Assumptions and Notations

To simplify the analytical model, we make the following assumptions on the network topology:

- The degree of each node is equal to f;
- The weight of each edge is equal to 1;
- There are N nodes in the network and M data objects are uniformly distributed in the network. Let $p = \frac{M}{N}$.

Table 1 summarizes all the notations, including those defined in the sequel.

Table 1. Notations for Cost Models

Notation	Definition
f	degree of each node
N	number of nodes
M	number of data objects
p	probability of a node is an object, $p = \frac{M}{N}$
g	average length of the circuits in the network
r	radius of the network
NA	number of nodes accesses in the search
$\mathcal{D}$	average distance between a node and its NN
$\mathcal{D}_k$	average distance between a node and its kth NN
n_d	cardinality of the d-th layer
$\mathcal{C}_d$	sum of cardinality of all layers within the d-th layer
P_i	probability that the NN is i units away

6.2   Cost Model for Network Reduction

We first estimate the number of edges remained after the network reduction. Let g denote the average length of the circuits in the network. During the reduction process, each circuit leads to a new shortest path tree with two shortcuts (ref. Figure 5). Since there are $f \cdot N/2$ edges, the number of circuits is $\frac{fN}{2g}$. So the
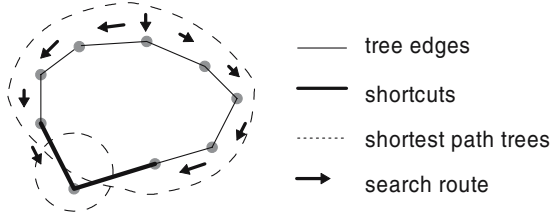

Fig. 5. A Circuit Leads to A New SPT with Two Shortcuts

number of tree edges and the number of shortcuts after reduction are $N - \frac{fN}{2g}$, and $\frac{2fN}{2g}$, respectively. Therefore, the ratio of the number of edges in the SPIE graph to the road network $\mathcal{R}$ is:

$$\mathcal{R} = \frac{(N - \frac{fN}{2g}) + \frac{2fN}{2g}}{fN/2} = \frac{2}{f} + \frac{1}{g} \tag{1}$$

An immediate observation from Equation 1 is that increasing f and g reduces the ratio and hence enlarges the performance gain of the network reduction. Nonetheless, the reduction is beneficial even when f and g are small. For example, in a network of 2D uniform grid, f equals to 4 and g equals to 4, $\mathcal{R} = 3/4 < 1$.

It is also noteworthy that, although SPIE does not further reduce the edges from SPT, it helps convert a shortcut to a tree edge, which reduces the NN search cost since the nd index is built on tree edges.

6.3   Cost Model for NN Search

To derive the number of node accesses in an NN search, we first derive the *average distance* ($\mathcal{D}$) between a node and its NN. Let us define the d-th *layer* of node q as the set of nodes that are d units away from q. Let n_d denote the cardinality of the d-th layer, and $\mathcal{C}_d$ denote the sum of cardinality of all layers within the d-th layer, i.e., $\mathcal{C}_d = \sum_{i=1}^{d} n_i$. Then we have:

$$\mathcal{C}_d = \sum_{i=1}^{d} n_i = 1 + f + f(f-1) + f(f-1)^2 + ... + f(f-1)^{d-1} \approx \frac{(f-1)^d}{f-2} \tag{2}$$

Let P_i denote the probability that the NN is i units away, and r denote the radius of the network. Then we have:

$$\mathcal{D} = \sum_{i=0}^{r} i \times P_i \tag{3}$$

Since the data objects are uniformly distributed, we have:

$$P_i = (1 - p)^{\mathcal{C}_{i-1}} (1 - (1 - p)^{\mathcal{C}_i - \mathcal{C}_{i-1}}) \tag{4}$$

Replacing P_i in (3) with (4) and $\mathcal{C}_i$ with (2), we get:

$$\mathcal{D} \approx \sum_{i=0}^{r-1} (1 - p)^{\mathcal{C}_i} \approx \sum_{i=0}^{r-1} (1 - p)^{\frac{(f-1)^i}{f-2}} \tag{5}$$

Now we estimate the number of node accesses in the NN search. The naive algorithm searches all nodes within the $\lceil \mathcal{D} \rceil$-th layer. Therefore, NA_{naive} is given by:

$$NA_{naive} = \mathcal{C}_{\lceil \mathcal{D} \rceil} \approx \frac{(f-1)^{\lceil \mathcal{D} \rceil}}{f - 2} \tag{6}$$

Recall that in our nd-based algorithm, the pointer p starts from q, examines the nd's of p's children (except for the child that p previously points at), and moves upward (and possibly to other SPIE's through the shortcuts) until the distance from p to q exceeds $\lceil \mathcal{D} \rceil$. Therefore,

$$NA_{nd} = \sum_{i=0}^{\lceil \mathcal{D} \rceil} (f - 1) = (f - 1)(\lceil \mathcal{D} \rceil + 1) \tag{7}$$

By comparing (6) and (7), NA_{naive} is exponential to the average NN distance $\mathcal{D}$ while NA_{nd} is linear to $\mathcal{D}$.

6.4   Cost Model for KNN Search

Similar to the derivation of NN search, we start by estimating $\mathcal{D}_k$, the *average distance* of the kth NN to q. Let P_i denote the probability that the kth NN is i units away. Then,

$$P_i = \binom{\mathcal{C}_i}{k} p^k (1 - p)^{\mathcal{C}_i - k} \approx \frac{\mathcal{C}_i^k p^k (1 - p)^{\mathcal{C}_i}}{k!(1 - p)^k} \tag{8}$$

Different from the NN search, we use the *maximum likelihood* (ML) estimation to derive $\mathcal{D}_k$, i.e., $\mathcal{D}_k = argmax_i P_i$. To get the maximum value of P_i in 8, it is equivalent to solve the following equation on the derivatives.

$$\frac{\partial \mathcal{C}_i^k (1 - p)^{\mathcal{C}_i}}{\partial i} = 0 \implies \frac{\partial \mathcal{C}_i^k (1 - \mathcal{C}_i p)}{\partial i} = 0 \tag{9}$$

The above derivation requires an additional assumption that $p << 1$. Solving (9) and replacing $\mathcal{C}_i$ by (2), we obtain,

$$\mathcal{D}_k = argmax_i P_i = \frac{\log k(f - 2) - \log p(k + 1)}{\log(f - 1)} \tag{10}$$

Now we estimate the number of node accesses in the KNN search. For the naive algorithm, similar to (6), we have:

$$NA_{naive} = C_{\lceil \mathcal{D}_k \rceil} \approx \frac{(f-1)^{\lceil \mathcal{D}_k \rceil}}{f-2} \tag{11}$$

Recall that in our nd-based algorithm, the pointer p starts from q, examines the nd's of p's children (except for the child that p previously points at), searches downwards, and moves upward (and possibly to other SPIE's through the short-cuts) until the distance from p to q exceeds $\lceil \mathcal{D} \rceil$. For each downward search, the number of node accesses, NA_{down}, is equivalent to the total length of the paths from the k nearest descendants to p. Let β denote the distance from the kth nearest descendant to p. We have the following two equations,

$$\sum_{i=1}^{\beta} (f-1)^i p = k$$

$$\sum_{i=1}^{\beta} (f-1)^i p \cdot i = NA_{down}$$

Solving these two equations, we have

$$NA_{down} \approx \frac{f \cdot \beta \cdot k}{p} \approx \frac{f \cdot k(\log k(f-2) - \log p)}{p \log(f-1)} \tag{12}$$

Therefore,

$$NA_{nd} = \sum_{i=0}^{\lceil \mathcal{D}_k \rceil} NA_{down} \approx \frac{f \cdot k(\lceil \mathcal{D}_k \rceil + 1)(\log k(f-2) - \log p)}{p \log(f-1)} \tag{13}$$

By comparing (11) and (13), we come to a similar conclusion as in Section 6.3 that $NA_{nd} << NA_{naive}$.

7 Performance Evaluation

In this section, we present the experimental results on network reduction, nd index construction and (k)NN search. We used two road networks in the simulation. The first is synthetic for controlled experiments, which was created by generating 183,231 planar points and connecting them through edges with random weights between 1 and 10. The degree of nodes follows an exponential distribution with its mean denoted as f. f is tuned to evaluate its effect on network reduction. The second is a real road network obtained from Digital Chart of the World (DCW). It contains 594,103 railroads and roads in US, Canada, Mexico. Among these line segments, we identified 430,274 unique nodes, and thus the average degree of nodes, f, is about 2.7. Similar to [9], we used the connectivity-clustered access method (CCAM) [12] to sort and store the nodes and their adjacent lists. The page size was set to 4K bytes. The testbed was implemented in C++ on a Win32 platform with 2.4 GHz Pentium 4 CPU, 512 MB RAM.

We compare our nd-based NN search algorithm with two competitors. The first is the Dijkstra-based naive network expansion algorithm which uses a priority queue to store the nodes to be searched and increasingly expands to their adjacent nodes on the network. The second is the Voronoi-based Network Nearest Neighbor (NV^3) algorithm [8] which computes the Network Voronoi Diagram for each dataset. So far, it is known to be the best algorithm for NN search in road networks.

Regarding the performance metrics, we measured the CPU time, the *number of disk page accesses* and the *number of node accesses*. The first two show the search cost while the last metric indicates the pruning capability of the network reduction and nd index.

7.1   Network Reduction

We evaluated the performance of the network reduction by measuring the number of edges before and after the reduction. In Figure 6, the result from the synthetic networks shows the same trend as Equation 1: when f increases from 2 to 10, the reduced edges increases from 5% to 60% of the total edges. However, when f gets even larger, the average length g of a circuit decreases, which partially cancels out the effect of f. Therefore, we expect the proportion of reduced edges to stabilize when $f > 10$. For the real road network, the average node degree f is reduced from 2.7 to 2.05, which is very close to a tree structure. In fact, only 1571 shortest path trees were created out of the 430,274 nodes. These results confirm the feasibility and effectiveness of network reduction on large road networks.
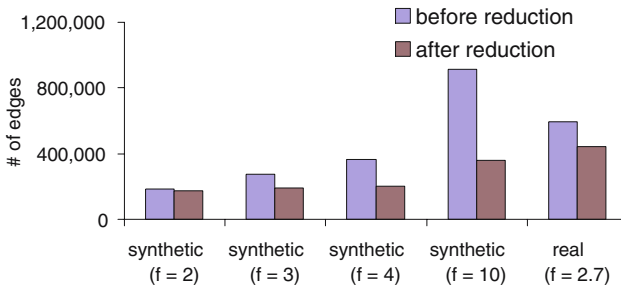

**Fig. 6.** Effect of Network Reduction

7.2   *nd* Index Construction

We created three randomly distributed datasets with their cardinality set to 0.001, 0.01, 0.1 (denoted as p) to the total number of nodes on the real road network. We then built both VN^3 index (including the NVP R-tree, NVD's, $Bor-Bor$ distances, and OPC distances) and nd index on these datasets. Table 2 shows the index sizes and the clock time for index construction. Note that for the nd index, we do not count the size and construction time for the SPIE graph, which is 7.5 MB and 303 seconds respectively, because this one-time cost is shared by all datasets. The result shows that our nd index has a constant size and almost constant construction time. It is more efficient to build than VN^3 index.

Table 2. Comparison on Index Construction

Size (MB)	$p = 0.001$	$p = 0.01$	$p = 0.1$	Time (s)	$p = 0.001$	$p = 0.01$	$p = 0.1$
VN^3	347	92	67	VN^3	2748	765	512
nd	5.16	5.16	5.16	nd	12	12	14

7.3   NN Search Result

We conducted experiments of NN search on the real road network for the three datasets and measured the CPU time, page accesses and node accesses[3]. All statistics were obtained from an average of 2,000 trials. In Figure 7(a), we observe that the number of page accesses for both the naive and nd algorithms decreases as the density of the dataset p increases, whereas the number of page accesses for VN^3 is almost constant. This is because the first two algorithms are based on graph traversal while VN^3 is based on point location on the NVP R-tree. Even though VN^3 precomputes the Network Voronoi Diagram and is thus efficient in finding the first nearest neighbor, our nd-based algorithm still outperforms it when $p > 0.01$, because more queries can be answered by visiting the nd of a few nodes on a single SPIE. In this sense, nd is more robust than VN^3 for datasets with various densities. Figure 7(b) confirms this argument: the nd-based algorithm reduces the node accesses of the naive algorithm by 2 orders of magnitude when $p = 0.001$ but it still reduces the nodes accesses by half when $p = 0.1$.

(a) Page Access

(b) Node Access
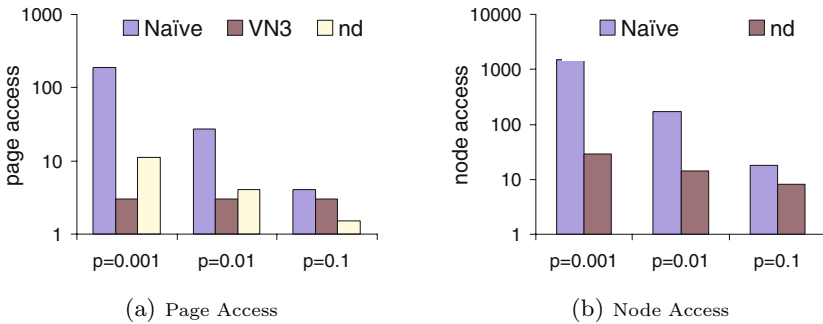
Fig. 7. Performance Comparison for NN Search

7.4   KNN Search Result

We conducted the kNN search for the $p = 0.01$ dataset on the real road network, where k ranges from 1 to 50. We measured the page accesses and CPU time and plotted the results in Figures 8(a) and 8(b). The results show that when $k = 1$, VN^3 requires the fewest page accesses and the least CPU time, because VN^3

---

[3] Since CPU time was found neglectable in 1NN search, we omit it in this subsection.

optimizes the 1NN search by only requiring the search to locate the NVP that contains the query node. However, as k increases, VN^3 still needs to traverse the NVD graph to search for candidate NNs; a major factor that contributes to the high cost of a kNN search by VN^3 is that the distance computation between each candidate and the query node is carried out separately and from scratch, while for network-expansion-based algorithms such as the naive and nd-based algorithms, the distance is computed accumulatively. This argument is supported by Figures 8(a) and 8(b) where the gap between VN^3 and the naive algorithm decreases as k increases. On the other hand, the nd-based algorithm performs consistently well for a wide range of k. The reasons are four-folded. Firstly, recall that after network reduction, each SPIE contains hundreds of nodes on average, which means that for small k it is likely that the search ends in one or two SPIE's. This explains why nd outperforms VN^3 even for small k. Secondly, although kNN search on nd index requires searching for the nd of p's descendants, these nd's are likely stored in the same disk page that p resides. Thirdly, since there are only 1571 SPIE's in the SPIE graph, looking for adjacent SPIE's to search is efficient. Last but not the least, thanks to the nd index that avoids naive expansion within one SPIE, the nd algorithm is the least affected by the increase of k. In Figures 8(a) and 8(b), both page accesses and CPU time of the nd algorithm are sublinear to k.

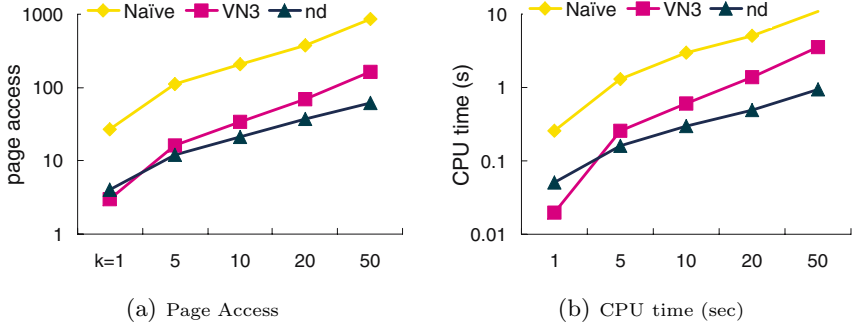(a) Page Access   (b) CPU time (sec)

Fig. 8. Performance Comparison for KNN Search: $p = 0.01$ Dataset

To summarize the results, the network reduction and nd-based (k)NN search algorithms exhibit the following advantages: (1) the network topology is significantly simplified and the reduction is a one-time cost for multiple datasets; (2) the nd index is lightweight in terms of storage and construction time; (3) the (k)NN search algorithm performs well for a wide range of datasets with different densities; (4) the kNN search algorithm performs well for a wide range of k.

8   Conclusion and Future Work

In this paper, we proposed a new kNN search technique for road networks. It simplifies the network by replacing the graph topology with a set of interconnected tree-based structures called SPIE's. An nd index was devised on the SPIE

so that our proposed kNN search on the SPIE follows a predetermined tree path. Both cost models and experimental results showed that our approach outperforms the existing network-expansion-based and solution-based kNN algorithms for most of the network topologies and data distributions.

In future work, we plan to devise structures other than SPIE to reduce the network topology. By striking a balance between the topological complexity of the structure and the kNN searching complexity on it, we can further improve the performance of our approach.

References

1. Stefan Berchtold, Daniel A. Keim, Hans-Peter Kriegel, and Thomas Seidl. Indexing the solution space: A new technique for nearest neighbor search in high-dimensional space. *TKDE*, 12(1):45–57, 2000.
2. Hyung-Ju Cho and Chin-Wan Chung. An efficient and scalable approach to cnn queries in a road network. In *VLDB*, 2005.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 2nd Edition*. McGraw Hill/MIT Press, 2001.
4. E. W. Dijkstra. A note on two problems in connection with graphs. *Numeriche Mathematik*, 1:269–271, 1959.
5. Eric Hanson, Yannis Ioannidis, Timos Sellis, Leonard Shapiro, and Michael Stonebraker. Heuristic search in data base systems. *Expert Database Systems*, 1986.
6. Christian S. Jensen, Jan Kolarvr, Torben Bach Pedersen, and Igor Timko. Nearest neighbor queries in road networks. In *11th ACM International Symposium on Advances in Geographic Information Systems (GIS'03)*, pages 1–8, 2003.
7. M. Kolahdouzan and C. Shahabi. Continuous k-nearest neighbor queries in spatial network databases. In *STDBM*, 2004.
8. Mohammad Kolahdouzan and Cyrus Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB Conference*, pages 840–851, 2004.
9. D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB Conference*, pages 802–813, 2003.
10. Nick Roussopoulos, Stephen Kelley, and Frdric Vincent. Nearest neighbor queries. In *SIGMOD Conference, San Jose, California*, pages 71–79, 1995.
11. C. K. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for knearest neighbor search in moving object databases. In *10th ACM International Symposium on Advances in Geographic Information Systems (GIS'02)*, 2002.
12. S. Shekhar and D.R. Liu. Ccam: A connectivity-clustered access method for networks and network computations. *IEEE Transactions on Knowledge and Data Engineering*, 1(9):102–119, 1997.
13. Roger Weber, Hans-Jorg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 194–205, 1998.
14. J. Xu, X. Tang, and D. L. Lee. Performance analysis of location-dependent cache invalidation schemes for mobile environments. *IEEE Transactions on Knowledge and Data Engineering*, 15(2):474–488, 2003.
15. Cui Yu, Beng Chin Ooi, Kian-Lee Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *VLDB Conference, Roma*, pages 421–430, 2001.