Finding All Nearest Neighbors with a Single Graph Traversal

Yixin Xu, Jianzhong Qi, Renata Borovica-Gajic, and Lars Kulik

School of Computing and Information Systems, The University of Melbourne
Melbourne, Australia
yixinx3@student.unimelb.edu.au
{jianzhong.qi,renata.borovica,lkulik}@unimelb.edu.au

Abstract. Finding the nearest neighbor is a key operation in data analysis and mining. An important variant of nearest neighbor query is the all nearest neighbor (ANN) query, which reports all nearest neighbors for a given set of query objects. Existing studies on ANN queries have focused on Euclidean space. Given the widespread occurrence of spatial networks in urban environments, we study the ANN query in spatial network settings. An example of an ANN query on spatial networks is finding the nearest car parks for all cars currently on the road. We propose VIVET, an index-based algorithm to efficiently process ANN queries. VIVET performs a single traversal on a spatial network to precompute the nearest data object for every vertex in the network, which enables us to answer an ANN query through a simple lookup on the precomputed nearest neighbors. We analyze the cost of the proposed algorithm both theoretically and empirically. Our results show that the algorithm is highly efficient and scalable. It outperforms adapted state-of-the-art nearest neighbor algorithms in both precomputation and query processing costs by more than one order of magnitude.

1 Introduction

Finding the nearest neighbor is an important query in spatial databases. Its variation includes reverse nearest neighbor search [1], continuous nearest neighbor search [2], all nearest neighbor search [3] and so forth. An important variant of the nearest neighbor query, the *all nearest neighbor query*, returns the nearest neighbor of each query object over a spatial network. Despite its importance, this query has not been addressed in the research literature on spatial networks.

ANN queries have many applications. We briefly discuss two of them: (i) ridesharing and (ii) carparks. For ridesharing, the average number of daily trips using Uber reached 5.5 million in 2016 [5], which shows the importance of highly scalable and efficient ANN algorithms to match cars with riders instantly. (ii) According to a study on parking spaces of 27 districts in the United States [4], the average oversupply ratio of parking spaces to cars requiring parking is 45% among districts that have identified parking shortages. The large oversupply ratio implies that building more parking spaces is not an effective solution to the perceived lack of parking spaces. Instead, this study shows that there is an increasing need for real-time parking management, which is able

to quickly report the locations of the nearest parking spaces for all drivers. This real-time parking management requires finding nearest neighbors (carparks) for all drivers in a road network. Both applications are examples of ANN queries in spatial networks. Figure 1 shows an example of an ANN query. Given two data objects o_1, o_2 and four query objects q_1, q_2, q_3, q_4, an ANN query is to compute the nearest data object for each query object, e.g., o_1 for q_1 and q_2, and o_2 for q_3 and q_4.

Existing studies on ANN mainly focus on the Euclidean space [3, 5–10], where the distance between two points is determined by their Euclidean distance. In the real world, movements of objects are usually restricted by the underlying road network. The traveling cost between two points is not only determined by their relative positions but also affected by the route between them. Take v_7 and v_{13} in Fig. 1 as an example. The travel distance between them is much larger than their Euclidean distance because the route must make a long detour to avoid the lake. In spatial networks, the distance between two points is measured by the length of their shortest path. Data structures and heuristics used by ANN algorithms in the Euclidean space, e.g., R-tree [11] and grid-partitioning [6], are not applicable to spatial networks due to the different distance concepts. Our study fills the need for an efficient ANN algorithm in spatial networks. To the best of our knowledge, this is the first study on ANN queries in spatial networks.

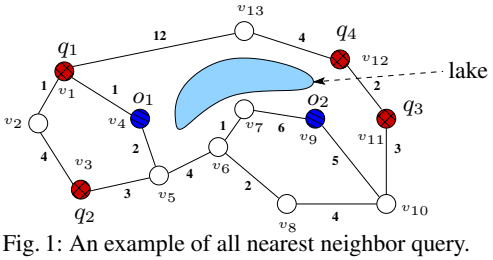


Fig. 1: An example of all nearest neighbor query.

A straightforward solution to find ANNs in spatial networks is to apply a state-of-the-art spatial network *nearest neighbor* (NN) algorithm for each query object individually. However, this solution is inefficient for large numbers of query objects. Besides, it does not scale to large networks due to high memory cost.

Applying NN algorithms straightforwardly is inefficient due to the overlap of the search regions of some query objects. For example, in Fig. 1, both the search regions of q_3 and q_4 cover the two edges between v_{10} and v_{11} and between v_9 and v_{10}, as these two edges are both on the shortest paths to their nearest neighbor the nearest neighbor of q_3 and q_4 (i.e., o_2). When the number of query objects is large, a large part of the network may be visited multiple times, thereby severely impacting query performance. Thus, an efficient ANN algorithm needs a careful design to avoid unnecessary visits.

The reason that the straightforward solution is not scalable is due to the index structures used by spatial network NN algorithms. Most of the recent spatial network NN algorithms improve their query performance by building indices during a precomputation phase. However, these indices are memory-intensive and thus do not scale to large networks. Table 1 depicts the average memory consumption of two state-of-the-art spatial network NN algorithms, G-tree [12] and IER-PHL [13], over five real-world road networks. The two algorithms consume rapidly increasing memory with the growing

network size, which renders them inapplicable to large networks. To illustrate, IER-PHL requires over 64 GB memory to index networks with roughly 14 million vertices.

Table 1: Memory consumption of G-tree, IER-PHL, and VIVET over five road networks.

Road network	**number of vertices**	**Memory consumption**		
		G-tree	**IER-PHL**	**VIVET**
Northwest US	1 million	88.4 MB	845.1 MB	9.2 MB
East US	3.6 million	339.8 MB	6.4 GB	27.5 MB
Western US	6.3 million	543.3 MB	10.4 GB	47.8 MB
Central US	14.1 million	1.5 GB	>64 GB	107.4 MB
Full US	23.9 million	2.4 GB	>64 GB	182.7 MB

We propose **VIVET** (**Vi**rtual **ve**rtex **t**raversal), a spatial network ANN algorithm that overcomes the above limitations. In the precomputation phase, VIVET runs Dijkstra's algorithm starting with a virtual vertex. The virtual vertex is created by connecting it to every data object with an edge of weight *zero* (as shown in Fig. 2). After the traversal, the shortest path from the virtual vertex to each vertex in the network is obtained. For each vertex v_i, we observe that there is always *one* data object o_j on the shortest path from the virtual vertex to v_i, and o_j is the nearest neighbor of v_i. We store the nearest neighbors of all vertices in an array N. For query processing, VIVET reports the nearest neighbor of every query object by a simple lookup to N.

VIVET significantly outperforms solutions adapted from state-of-the-art nearest neighbor algorithms described above in terms of precomputation and query cost. The precomputation of VIVET is efficient and easy to implement compared with other nearest neighbor indices because it only requires a *single traversal* over the network. Furthermore, the memory consumption of the VIVET index (the array N) is linear to the number of vertices in the network, which makes it scalable to large networks. Taking Table 1 as an example, the memory consumption of VIVET is more than an order of magnitude lower compared with the index of G-tree and two orders of magnitude lower compared with the index of IER-PHL. In query processing, VIVET refers to the array N directly to report the query results and thus outperforms the state-of-the-art NN algorithms by almost two orders of magnitude. For example, VIVET needs less than 0.02 seconds to answer 500,000 query objects while existing NN algorithms require more than 6 seconds under the same setting.

To summarize, our contributions are as follows:

- To the best of our knowledge, this is the first study on all nearest neighbor queries in spatial networks.
- We propose a simple and efficient algorithm called VIVET for ANN queries in spatial networks. VIVET is applicable to both undirected and directed networks.
- Our theoretical analysis proves the advantage of VIVET. The precomputation of VIVET requires $O((|E| + |V| + n) \log |V|)$ time and $O(|V|)$ space, where n represents the number of data objects and $|E|$, $|V|$ represent the number of edges and vertices, respectively. The overall query complexity is linear to the number of query objects m, i.e., $O(m)$.

– We conduct experiments on both real-world and synthetic data, showing that VIVET outperforms the state-of-the-art algorithms by one to two orders of magnitude in terms of query times and precomputation costs.

2 Preliminaries

We start with a few basic concepts, based on which we define the all nearest neighbor query in spatial networks.

We consider a set of n data objects $\mathcal{O} = \{o_1, o_2, ..., o_n\}$ and a set of m query objects $\mathcal{Q} = \{q_1, q_2, .., q_m\}$. Both the data objects and the query objects are represented by points on a spatial network.

The *spatial network* is modeled by a graph $G = \langle V, E \rangle$, where V is a set of vertices and E is a set of edges. We consider both directed and undirected graphs. For ease of presentation, we assume an undirected graph by default, and will discuss how our techniques and algorithms can be adapted to directed graphs in Section 3.3. An edge $e_{i,j} \in E$ connects two vertices v_i and v_j in V. Such two vertices are called *adjacent vertices*. Every edge $e_{i,j}$ is associated with a *weight*, denoted by $w(e_{i,j})$, which represents the cost of traveling between v_i and v_j. A *path* between two vertices v_i and v_j is an ordered list of edges between the two vertices, denoted by $P_{i,j}$. We use $|P_{i,j}|$ to denote the number of edges in the path, and $l(P_{i,j})$ to denote the *length* of the path, which is the sum of the weights of the edges in the path. The *shortest path* between v_i and v_j is the path between them with the smallest length. This smallest length is called the *shortest path distance* between v_i and v_j, denoted by $d_n(v_i, v_j)$. We further use $d_e(v_i, v_j)$ to denote the Euclidean distance between v_i and v_j.

For simplicity, we assume that the data objects and the query objects are located at the graph vertices. This assumption can be easily met by adding vertices that represent the data objects or query objects to the graph.

Nearest neighbor query in a spatial network. Given a query object q and a set of data objects $\mathcal{O}$ in a spatial network G, a *nearest neighbor* query finds the nearest data object $o_i \in \mathcal{O}$ with the smallest shortest path distance to q, denoted by $NN(q)$:

$$NN(q) = \{o_i \in \mathcal{O} | \forall o_j \in \mathcal{O} : d_n(o_i, q) \leq d_n(o_j, q)\}$$

All nearest neighbor query in a spatial network. Given a set of query objects $\mathcal{Q}$ and a set of data objects $\mathcal{O}$ in a spatial network G, an *all nearest neighbor* query finds the nearest data object $o_j \in \mathcal{O}$ with the smallest shortest path distance to every query object $q_i \in \mathcal{Q}$. The query answer is a set of tuples of a query object and its nearest data object, denoted by $ANN(\mathcal{Q}, \mathcal{O})$. Formally,

$$ANN(\mathcal{Q}, \mathcal{O}) = \{\langle q_i, o_j \rangle | q_i \in \mathcal{Q}, o_j \in \mathcal{O}, o_j = NN(q_i)\}$$

In Fig. 1, $ANN(Q, O) = \{\langle q_1, o_1 \rangle \langle q_2, o_1 \rangle, \langle q_3, o_2 \rangle, \langle q_4, o_2 \rangle\}$.

3 VIVET

In this section, we present our VIVET algorithm for ANN queries in spatial networks. The VIVET algorithm precomputes and stores the nearest data object of every vertex in

the network. When an ANN query is issued, we simply lookup for the vertices where the query objects lie on and return the corresponding nearest data object. Next, we detail the precomputation process of VIVET, which computes the nearest neighbors for all the vertices in a spatial network with a single traversal over the network.

3.1 Precomputation

To compute the nearest neighbors for all the vertices, a straightforward method is to run a graph shortest path search algorithm such as Dijkstra's algorithm [14] starting from every vertex in the network. However, this algorithm may traverse the network too many times and access the same vertices and edges repetitively.

To avoid such repetitive computation and overlapping network traversals, we propose to traverse the network starting from a virtual vertex which connects to all data objects. The traversal will go through every vertex in the network. When the traversal reaches a vertex, the corresponding path reaching the vertex must pass a data object and this data object will be recorded as the nearest neighbor of the vertex.

We first augment the graph G with a virtual vertex v^* and connect it to every data object $o_i \in \mathcal{O}$ with a directed edge $\overrightarrow{e_{*,i}}$ of weight 0. As we assume that the data objects are all on the vertices, this process effectively connects the virtual vertex to every vertex v_i in V on which a data object lies. We denote the resulting graph as G^*, $G^* = \langle V^*, E^* \rangle$, where $V^* = V \cup \{v^*\}$ and $E^* = E \cup \{\overrightarrow{e_{*,i}} | \overrightarrow{e_{*,i}}$ connects v^* to $o_i \in \mathcal{O}\}$.

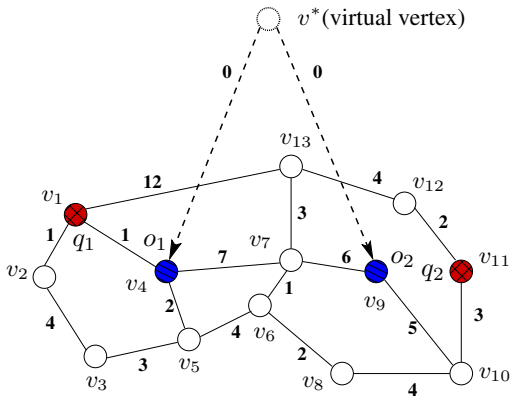


Fig. 2: An example of VIVET.

We call G^* the *augmented graph*. Figure 2 illustrates such a graph. The virtual vertex v^* is connected to vertices v_4, v_9 where there are data objects o_1 and o_2. Note that even though the original graph G is undirected, the augmented graph G^* contains directed edges that only allows traveling from v^* to the vertices (data objects) in V. The directed edges here are used to ensure that the graph traversal starting from v^* will not go back to v^* (note the zero weight for the edges connecting v^* to the data objects), so as to guarantee the validity of our precomputation algorithm.

Once the augmented graph G^* is computed, we run a single-source graph shortest path algorithm (e.g., Dijkstra's algorithm) starting from the virtual vertex v^* to find the shortest path to every vertex in V. We record the data object that a path goes through.

When the traversal reaches a vertex v_i, the data object on the path to v_i is recorded. We show that there is always one and only one data object on the shortest path from v^* to v_i and this data object is the nearest data object of v_i with the following two lemmas.

Lemma 1. ***Given a connected graph*** $G = \langle V, E \rangle$ ***and an augmented graph*** $G^* = \langle V^*, E^* \rangle$ ***created from*** G***, for every vertex*** $v_i \in V$***, there must be one and only one data object on the shortest path from*** v^* ***to*** v_i***.***

Proof. First, we prove that there must be at least one data object on the shortest path to v_i. Since G is connected, there must be a path that connects v^* to v_i by the design of the augmented graph G^*. Since v^* is only connected to the data objects, any path including the shortest path from v^* to v_i must go through at least one data object.

Next, we prove by contradiction that there is at most one data object on the shortest path to v_i. Let $\langle P = v^*, o_j, ..., v_i \rangle$ be the shortest path from v^* to v_i. The second vertex on by the path must be a vertex on which a data object o_j lies by design of G^*. Suppose that there is another data objects o_k in the path, i.e., $P = \langle v^*, o_j, ...o_k, ..., v_i \rangle$. Then, the distance between o_j and v_i must be larger than that between o_k and v_i, i.e., $d_n(o_j, v_i) > d_n(o_k, v_i)$. Since there is an edge that connects from v^* to every data object, there must be another path $P' = \langle v^*, o_k, ..., v_i \rangle$. The edge between v^* to every data object has a zero weight. Thus, the path length $l(P) = d_n(o_j, v_i) > d_n(o_k, v_i) = l(P')$, which contradicts that P is the shortest path between v^* and v_i. □

For example, in Fig 2, there is only one data object o_2 on the shortest path between v^* and v_{11}, which goes through v^*, o_2, v_{10}, v_{11}.

Lemma 2. ***Given a connected graph*** $G = \langle V, E \rangle$ ***and an augmented graph*** $G^* = \langle V^*, E^* \rangle$ ***created from*** G***, the data object on the shortest path from*** v^* ***to every vertex*** $v_i \in V$ ***is the nearest data object of*** v_i***.***

Proof. The proof is similar to the second half of Lemma 1's proof and omitted. □

Lemmas 1 and 2 guarantee the correctness of using a single-source shortest path algorithm to compute the nearest neighbor for every vertex. Any single-source graph shortest path algorithms can be used. We use Dijkstra's algorithm for its simplicity and efficiency [14].

Once the nearest neighbors of the vertices are computed, we store them as an array of vertex-NN pairs (together with the shortest path distance) for fast retrieval at query processing. We call this array the *NN array*. Table 2 illustrates the NN array built for the example shown in Fig. 2.

Table 2: VIVET index of Fig. 2.

	$\boldsymbol{v_1}$	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	$\boldsymbol{v_{11}}$	v_{12}	v_{13}
NN	$\boldsymbol{o_1}$	o_1	o_1	o_1	o_1	o_1	o_2	o_1	o_2	o_2	$\boldsymbol{o_2}$	o_2	o_2
distance	**1**	2	5	0	2	6	6	8	0	5	**8**	10	9

Algorithm 1 summarizes the precomputation procedure of VIVET. The algorithm starts with creating the augmented graph G^* based on the spatial network G (Lines 1

Algorithm 1: Precomputation

Input : $G = \langle V, E \rangle$, object set $\mathcal{O}$
Output: NN array indexing the nearest object of every vertex $v_i \in V$.

```
1  create a virtual vertex v*;
2  E* = E, V* = V ∪ {v*};
3  for o_i ∈ O do
4      create a virtual edge e_{*,o_i.vid};          // o_i.vid is the vertex ID of o_i
5      w(e_{*,o_i.vid}) = 0;
6      E* = E* ∪ {e_{*,o_i.vid}};
7  initialize an array N with size |V|;
8  for o_i ∈ O do
9      N[o_i.vid].nndistance = 0;
10     N[o_i.vid].nnid = o_i.oid;                    // o_i.oid is the object ID of o_i
11 initialize a priority queue PQ;
12 PQ.insert (v*);
13 while PQ ≠ ∅ do
14     v_i = the first element in PQ;
15     if v_i has not been visited before then
16         for each adjacent vertex v_j of v_i that have not been visited before do
17             if N[j].nndistance > N[j].nndistance + w(e_{i,j}) then
18                 N[j].nndistance = N[j].nndistance + w(e_{i,j});
19                 N[j].nnid = N[i].nnid;
20                 PQ.insert(v_j);
21         mark v_i as visited;
22 return N;
```

to 6). Then, it initializes an array N of size $|V|$ to store the NN pairs (Line 7). The data objects located at vertices are the nearest data objects of those vertices, which yield a nearest neighbor distance of 0 (Lines 8 to 10). Next, the graph traversal starts. We use a priority queue PQ to facility the traversal (Line 11). Each element in the queue is a vertex in G^* to be visited, which is prioritized by its distance to the nearest data object computed so far in the NN array. The virtual vertex v^* is inserted into PQ to initialize the traversal (Line 12). A loop is run to keep popping out vertices from PQ (Lines 13 to 21). The vertex with the smallest distance to the nearest data object in PQ is popped out first (Line 14). When a vertex v_i is popped out and visited for the first time, vertices connected to it that have not been visited before are inserted into PQ (Lines 16 to 20). For each such vertex v_j, if the path through v_i is shorter than the existing shortest path to v_j, we update the distance to nearest data object of $v_j(N[j].nndistance)$ to be the nearest neighbor distance of v_i plus the weight of the edge between v_i and v_j (Line 18), and the nearest data object of $v_j(N[j].nnid)$ is updated to be that of v_i (Line 19). When PQ becomes empty, all vertices will have been visited and their nearest data objects are computed and stored in the NN array N. The array N is returned and the algorithm terminates (Line 22).

3.2 Query Processing

Once the NN array is computed, an ANN query can be processed by first locating the vertex v_j on which every query object q_i lies and then retrieving the nearest data object of v_j from the NN array, which is returned as the nearest data object of q_i. If a query object is lying on an edge, we locate both vertices of the edge and retrieve their nearest data objects. We compare the distances of the two retrieved data objects to q_i and return the closer one as the nearest data object of q_i. We omit the pseudocode of the query processing procedure for conciseness.

Continuing with the example shown in Fig. 2, where there are two query objects q_1, q_2 represented by the two red circles, q_1 is at v_1 and q_2 is at v_{11}. The nearest neighbor of v_1 is o_1 and the nearest neighbor of v_{11} is o_2 as shown in the NN array listed in Table 2. VIVET reports o_1 and o_2 as the nearest neighbors of q_1 and q_2, respectively.

3.3 Generalizing the Algorithm

VIVET can be generalized to directed networks and to process ANN queries without precomputation with small changes.

When applied to directed networks, we need to update the traversal for single-source graph shortest path computation as follows. When a vertex is visited (Line 15 of Algorithm 1), we retrieve its inbound edges and add the vertices connected by these edges to the priority queue PQ to be visited next, i.e., we update Line 16 of Algorithm 1 to be "for every vertex v_j that has an edge pointing to v_i". We need to reverse the direction of the edges of the virtual vertex v^* such that they point from the data object vertices to v^* instead of from v^* to the data objects. By doing so, we find the shortest "reverse" paths from the data objects to the vertices in the network, which are the shortest paths from the vertices to the data objects. This approach is correct because we still use Dijkstra's algorithm graph expansion procedure, but restricting the direction of the edges to ensure that the paths found are going from the vertices to the data objects. Our experiments show similar behavior of VIVET for both undirected and directed graphs.

When processing an ANN query without the precomputed NN array, we run the single-source graph shortest path computation online. We find the shortest paths from the virtual vertex v^* to all query objects instead of all network vertices. When the shortest paths are found, the data object o_j on the shortest path to query object q_i is returned as the nearest data object of q_i. The correctness of doing so is guaranteed by Lemmas 1 and 2 above straightforwardly. Our experiments verify the efficiency of VIVET in dynamic scenarios, especially when the number of query objects is large. For example, when the network has over one million vertices and 2^{11} data objects, dynamic VIVET requires 0.5 seconds to answer an ANN query with 2^{16} query objects while the other state-of-the-art algorithms requires at least 0.8 seconds.

A *multi-source Dijkstra's algorithm* has been proposed in the literature [15] that starts by adding multiple source vertices into the priority queue PQ. The focus of [15] is to run the multi-source Dijkstra's algorithm to test the reachability of different points and find out the most time-consuming shortest path in the graph for emergency services. Another study [16] shares a similar idea and uses a multi-source shortest path approach for location privacy. To the best of our knowledge, we are the first to apply this technique for finding ANNs in spatial networks.

Table 3: Road networks.

Name	# vertices	# edges	Description
NY	264,346	733,846	New York (Undirected)
COL	435,666	1,057,066	Colorado (Undirected)
FLA	1,070,376	2,712,798	Florida (Undirected)
NW	1,207,945	2,840,208	Northwest USA (Undirected)
CAL	1,890,815	4,657,742	California & Nevada (Undirected)
E	3,598,623	8,778,114	Eastern USA (Undirected)
W	6,262,104	15,248,146	Western USA (Undirected)
CTR	14,081,816	34,292,496	Central USA (Undirected)
Europe	18,010,173	42,188,664	Europe (Directed)
USA	23,947,347	58,333,344	Full USA (Undirected)

3.4 Algorithm Complexity

Next, we analyze the complexity of VIVET. We denote the number of data objects as n and the number of query objects as m. We also denote the numbers of vertices and edges in G^* as $|V^*|$ and $|E^*|$, which equals to $|V|$+1 and $|E| + n$, respectively.

Precomputation. Creating the augmented graph G^* takes $O(|V|+1+|E|+n)$ time. The time for traversing G^* to compute the nearest data objects is determined by the time of the single-source shortest path algorithm. We use Dijkstra's algorithm, which has a time complexity of $O((|E^*| + |V^*|) \log |V^*|)$ in the worst case by using a binary heap for the priority queue PQ, which is equivalent to $O((|E| + |V| + n) \log |V|)$. Overall, the time complexity of the precomputation of VIVET is $O((|E| + |V| + n) \log |V|)$. The size of the NN array is linear to the number of vertices, i.e., $O(|V|)$.

Query processing. The query time complexity of VIVET is linear to the number of query objects. For each query object, the nearest neighbor is computed in constant time from the NN array. Therefore, the query time complexity of VIVET is $O(m)$.

4 Experiments

We experimentally compare the performance of VIVET against the state-of-the-art NN algorithms, IER-PHL [17], G-tree [12] and INE [18]. All algorithms are implemented in C++ and run on a 64-bit virtual node with a 1.8 GHz CPU and 64 GB memory from an academic computing cloud (Nectar [19]) running on OpenStack.

4.1 Experimental Setup

We run ANN queries on real-world road network datasets as listed in Table 3, which are created for the 9^{th} DIMACS Challenge [20]. Each undirected network has two datasets, a travel time dataset and a travel distance dataset, the edge weight of which correspond to the travel distance and the travel time between vertices, respectively. Note that the directed network *Europe* has only the travel time dataset. As the experimental results

on the travel distance dataset are consistent with that on the travel time dataset in most cases, we focus on showing experiments on the travel time dataset due to the space limit. We use two methods to create data object sets: mapping real-world POIs and synthetically sampling. We use eight types of real-world POIs extracted from OpenStreetMap by Abeywickrama et al. [17]. We also synthetically sample vertices of the networks to be the data objects and query objects following two distributions, uniform and clustered. The uniform distribution simulates scenarios where areas with more vertices tend to have more objects, while the clustered distribution simulates scenarios where objects may be clustered in some areas. The maximum number of vertices is 50 in every cluster.

Table 4: Experiment settings.

Parameters	**Values**	**Default**
Road Networks	Refer to table 3	*NW*
number of data objects	2^7 to 2^{16}	2^{11}
number of query objects	2^7 to 2^{16}	2^{10}
Real-world POIs	Refer to Table 2 in [17]	*Parking*
Synthetic data objects distributions	Uniform, Clustered	*Uniform*
Synthetic query objects distributions	Uniform, Clustered	*Uniform*

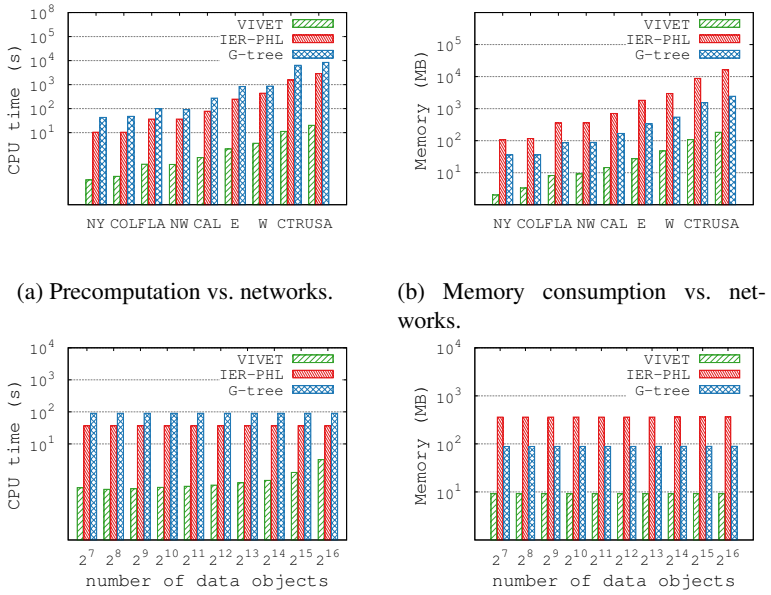
Table 4 shows the range of variables we use in our experiments. In a default setting, we run queries on 2^{11} uniformly distributed data objects and 2^{10} uniformly distributed query objects over the network *NW*. *Park* is the default POI type in experiments using real-world POIs as data objects. We first show the algorithm performance on undirected graphs and then compare algorithms on directed graphs.

4.2 Precomputation Costs

We compare the precomputation costs of VIVET with the index-based NN algorithms IER-PHL and G-tree by measuring their time and memory consumption.

Effect of the network size. Figures 3a and 3b show the precomputation costs over different networks. All algorithms require longer time and larger memory to build indices when the network has more nodes and edges. Compared with IER-PHL and G-tree, VIVET reduces the precomputation time by two orders of magnitude and saves the memory consumption by one order of magnitude due to a single traversal over the network. Compared to the precomputation costs on the travel distance dataset as shown in Table 1, both G-tree and VIVET require consistent precomputation costs on the two datasets. IER-PHL, however, requires less memory on the travel time dataset by taking advantage of the travel speed (geometrical length divided by travel time) to improve the effectiveness of the highway decomposition, thereby reducing the index size.

Effect of the number of data objects. Figures 3c and 3d show the effect of the number of data objects on the precomputation costs. Varying the number of data objects has little effect on the precomputation costs of both IER-PHL and G-tree because their precomputation costs are dominated by the process of building network indices. As for VIVET, its precomputation time increases with the growing number of data objects,



(a) Precomputation vs. networks.

(b) Memory consumption vs. networks.

(c) Precomputation time vs. # data objects.

(d) Memory consumption vs. # data objects.

Fig. 3: Precomputation costs.

which is caused by the increasing number of virtual edges added in the augmented network G^*. Even though the precomputation costs of VIVET are impacted by the number of data objects, the number of data objects in real world scenarios usually lies within a reasonable range. For example, the number of parking spaces in NW is 5098 [17], which lies in the range between 2^{12} and 2^{13} as shown in Fig. 3c. The precomputation time of VIVET in this range is approximately 1.5% of that of IER-PHL and 0.5% of that of G-tree. In terms of memory consumption VIVET has a constant index size when the number of data objects changes as its index size is determined only by the number of vertices. Its index size is at least an order of magnitude smaller than those of IER-PHL and G-tree.

4.3 Query Costs

We further analyze the query performance of IER-PHL, G-tree, INE, and VIVET by comparing their query times.

Effect of the network size. Figure 4 shows the query times of the four algorithms on different networks. The query times of G-tree and INE increases rapidly with the growing network size due to their larger search region, while those of IER-PHL and VIVET are much less impacted by the network size. However, IER-PHL consumes large size of memory for large networks as shown in Fig.3b. VIVET outperforms the other three algorithms by more than two orders of magnitude over all networks, which shows the efficiency and scalability of VIVET in large networks.

Effect of the number of data objects. Figure 6 shows the query times of uniform and clustered data objects when varying number of data objects. VIVET again outperforms the state-of-the-art by more than two orders of magnitude. Furthermore, the query times of VIVET is unaffected by the size and distribution of data objects as it only performs a simple lookup to answer an ANN query.

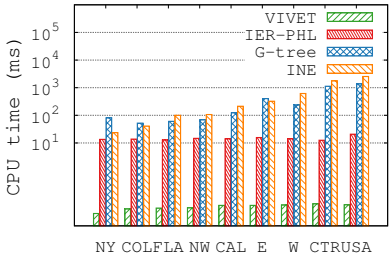


Fig. 4: Query time vs. network size.

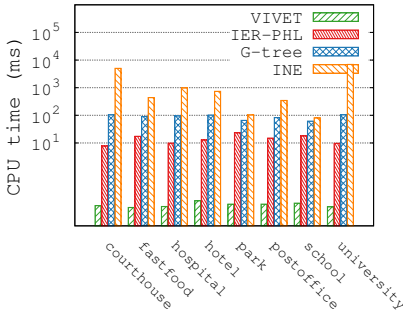


Fig. 5: Query time vs. real data objects.

Effect of the number of query objects. Figure 7 shows the effect of the number of query objects on the query performance. Query objects in Fig. 7a are generated following the uniform distribution while those in Fig. 7b are generated following a clustered distribution. As expected, the query times of all algorithms grow with the increasing number of query objects. VIVET is two orders of magnitude faster than the most efficient baseline IER-PHL. Furthermore, our scalability experiments show that VIVET can answer an ANN query with 10 million query objects within 0.3 seconds, while the other state-of-the-art algorithms require more than 2 seconds to answer such large number of query objects.

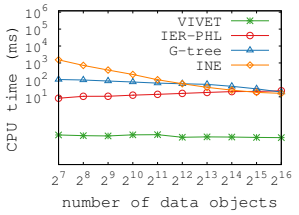
Real-world object sets. Figure 5 shows the query times of different algorithms when data objects are generated based on real-world POIs. VIVET outperforms other algorithms by more than two orders of magnitude on all types of POIs examined.

4.4 Experiments on Directed Graphs

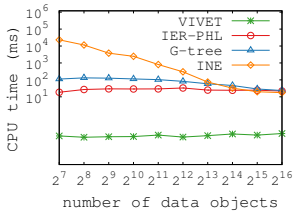
Since G-tree requires undirected graphs for its graph partitioning phase while IER-PHL assumes undirected graphs for indexing, we only compare the performance of VIVET against INE on the directed network *Europe*.

Precomputation cost. Figure 8 shows the precomputation time of VIVET when the number of data objects varies. The precomputation time required by VIVET increases slightly with increasing number of data objects, which is consistent with the experiments in undirected graphs. The memory consumption of VIVET on *Europe* is 137 MB, which remains linear to the number of vertices in the network.

Query cost. The query times of VIVET and INE on directed network are compared in Fig. 9. VIVET outperforms INE by up to four orders of magnitude in this set of experiments. When the number of data objects increases, the query performance of INE improves due to the smaller size of the search region. However, even for dense data objects, VIVET still outperforms INE by three orders of magnitude.

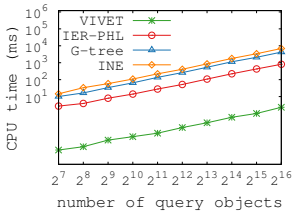


(a) uniform data objects

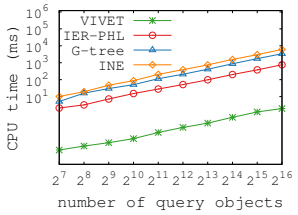


(b) clustered data objects

Fig. 6: Effect of the number of data objects on query time.



(a) uniform query objects



(b) clustered query objects

Fig. 7: Effect of the number of query objects on query time.

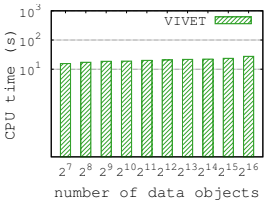


Fig. 8: Precomputation time (directed graph).

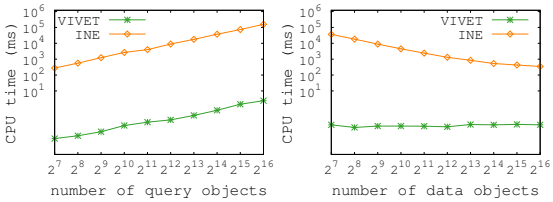


Fig. 9: Query time (directed graph).

5 Related Work

Nearest neighbor queries are studied extensively under various data spaces including spatial network spaces, while all nearest neighbor queries are studied mainly in the Euclidean spaces. We review studies on nearest neighbor queries in spatial networks and all nearest neighbor queries in the Euclidean space.

Nearest neighbor queries in spatial networks. A key issue in NN query processing in spatial networks lies in the high cost of computing the shortest path distance between two objects, which may require a graph traversal. Studies of NN queries in spatial networks thus explore various techniques to reduce the shortest path distance computation. Papadias et al. [18] propose two spatial network NN algorithms: *IER* and *INE*. The IER algorithm is based on the observation that the spatial network distance between two objects must be no smaller than their Euclidean distance. The INE algorithm gradually expands the search region from the query object so that the first data object reached when expanding is the query answer. Kolahdouzan et al. [21] precomputes a *network Voronoi diagram* over the spatial network, which partitions the network into sections

(*network Voronoi cells*). An NN query can then be answered simply by locating the network Voronoi cell containing the query object q . More recent studies use index structures to help to improve the query efficiency. The *distance browsing* algorithm [22] uses the *spatially induced linkage cognizance* (SILC) index, which stores the network shortest path distance between every pair of vertices. The *ROAD* [23] algorithm hierarchically partitions the spatial network and precomputes the spatial network distance between border vertices within every partition, where border vertices of a partition are the vertices connecting to other partitions. The *G-tree* [12] algorithm also partitions the network but differs from ROAD on the tree structures and searching paradigms.

An experimental paper [17] compares the performance of various network NN algorithms. They also proposed an algorithm named IER-PHL that combines the IER algorithm with a shortest path computation technique named *pruning highway labelling* (PHL) [13]. It is found that the query time of IER-PHL outperforms the other NN algorithms in most cases. G-tree is also competitive because it requires lower precomputation costs than IER-PHL, while ranks the second in terms of the query time. INE, on the other hand, is the most efficient algorithm when the data objects are densely distributed.

All Nearest Neighbour in Euclidean Space. ANN algorithms in the Euclidean space can be grouped into index-free and index-based algorithms. We start with the index-free algorithms. Clarkson et al. [24] consider the case where query and data objects belong to the same set. and split the space into small cubic cells of equal size. The distance from a query object to its nearest neighbor is hence bounded by the distance to the nearest cell occupied by a data object. Vaidya et al. [25] use a similar idea but optimize the splitting scheme. When query objects and data objects are in two different sets, the ANN query is also called the *nearest neighbor join* (NN-join) query. Xia et al. [6] propose the *Gorder* algorithm to process the ANN-join query. Gorder divides query objects and data objects into several blocks and schedules the searching order of data objects' blocks so that promising nearest neighbor candidates are visited first. Zhang et al. [7] propose a hash-based algorithm that hashes the query objects together with the data objects and divides them into buckets. For query objects in a bucket, nearest neighbors only need to be searched from data objects in the same or overlapping buckets. Chen et al. [10] use the Hilbert curve to hash the data objects into grid cells.

Index-based ANN algorithms compute the query with a traversal over a precomputed index structure. Böhm et al. [3] propose an R-tree based algorithm named *MuX*. They optimize the I/O cost by organizing input data using large pages and take advantage of a secondary search structure within pages to optimize the efficiency. Zhang et al. [7] propose two algorithms for the case where the data objects are indexed with an R-tree. Their first algorithm named *multiple nearest neighbor* (MNN) finds the nearest neighbor of every query object by computing an NN query on the R-tree of data objects. The processing order of the query objects is optimized so that close query objects can be handled consecutively. Their second algorithm named *batched nearest neighbor* (BNN) finds nearest neighbors of multiple query objects at a time. BNN first groups multiple query objects and traverses the R-tree of data objects once for finding the nearest neighbors of a group. Chen et al. [5] use a Quad-tree variant called the *MBRQT* to index the data objects and propose a metric called *NXNDIST* (MINMAXMINDIST) to prune the search space during index traversal. Sankaranarayanan et al. [26] propose another prun-

ing metric called *MAXMAXDIST*. Yu et al. [8] use *iDistance* [27] as the index structure. They propose an algorithm named *iJoin* that takes advantage of the data partitioning strategy of iDistance. Emrich et al. [9] propose to index the data objects with an *SS-tree* and use trigonometric relationships to prune the search space during index traversal.

6 Conclusion

We studied all nearest neighbor queries in spatial networks and proposed a scalable and efficient algorithm named VIVET. Compared with the methods adapted from state-of-the-art nearest neighbor algorithms, VIVET reduces the precomputation and query costs by one to two orders of magnitude. The improvements are achieved via a shared computation technique that computes the nearest neighbors for all query objects at the same time with a single graph traversal. Extensive experiments using real road networks confirm the advantages of VIVET in terms of precomputation time, storage space, and query time compared to the state-of-the-art network NN algorithms.

Since existing index structures for nearest neighbor queries suffer due to large memory consumption, while VIVET effectively overcomes this limitation, it is worth to explore the applicability of the VIVET index structure on other variants of the nearest neighbor problems in spatial networks in the future. Our preliminary results already confirmed the advantage of VIVET on NN queries compared to the state-of-the-art in terms of precomputation costs and query performance.

Acknowledgment

This work is supported in part by Australian Research Council (ARC) Discovery Project DP180103332.

References

1. Safar, M., Ibrahimi, D., Taniar, D.: Voronoi-based reverse nearest neighbor query processing on spatial networks. Multimedia systems 15(5), 295–308 (2009)
2. Mouratidis, K., Yiu, M.L., Papadias, D., Mamoulis, N.: Continuous nearest neighbor monitoring in road networks. In: VLDB. pp. 43–54 (2006)
3. Böhm, C., Krebs, F.: The k-nearest neighbour join: Turbo charging the kdd process. Knowledge and Information Systems 6(6), 728–749 (2004)
4. Weinberger, R.R., Karlin-Resnick, J.: Parking in mixed-use us districts: Oversupplied no matter how you slice the pie. Transportation Research Record: Journal of the Transportation Research Board (2537), 177–184 (2015)
5. Chen, Y., Patel, J.M.: Efficient evaluation of all-nearest-neighbor queries. In: ICDE. pp. 1056–1065 (2007)
6. Xia, C., Lu, H., Ooi, B.C., Hu, J.: Gorder: an efficient method for knn join processing. In: VLDB. pp. 756–767 (2004)
7. Zhang, J., Mamoulis, N., Papadias, D., Tao, Y.: All-nearest-neighbors queries in spatial databases. In: SSDBM. pp. 297–306 (2004)
8. Yu, C., Cui, B., Wang, S., Su, J.: Efficient index-based knn join processing for high-dimensional data. Information and Software Technology 49(4), 332–344 (2007)

9. Emrich, T., Graf, F., Kriegel, H.P., Schubert, M., Thoma, M.: Optimizing all-nearest-neighbor queries with trigonometric pruning. In: SSDBM. pp. 501–518. Springer (2010)
10. Chen, H.L., Chang, Y.I.: All-nearest-neighbors finding based on the hilbert curve. Expert Systems with Applications 38(6), 7462–7475 (2011)
11. Guttman, A.: R-trees: A dynamic index structure for spatial searching. In: SIGMOD. pp. 47–57 (1984)
12. Zhong, R., Li, G., Tan, K.L., Zhou, L.: G-tree: An efficient index for knn search on road networks. In: CIKM. pp. 39–48 (2013)
13. Akiba, T., Iwata, Y., Kawarabayashi, K.i., Kawata, Y.: Fast shortest-path distance queries on road networks by pruned highway labeling. In: ALENEX. pp. 147–154 (2014)
14. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische mathematik 1(1), 269–271 (1959)
15. Eklund, P.W., Kirkby, S., Pollitt, S.: A dynamic multi-source dijkstra's algorithm for vehicle routing. In: ANZIIS. pp. 329–333 (1996)
16. Duckham, M., Kulik, L.: A formal model of obfuscation and negotiation for location privacy. In: Pervasive. pp. 152–170. Springer (2005)
17. Abeywickrama, T., Cheema, M.A., Taniar, D.: K-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. PVLDB 9(6), 492–503 (2016)
18. Papadias, D., Zhang, J., Mamoulis, N., Tao, Y.: Query processing in spatial network databases. In: VLDB. pp. 802–813 (2003)
19. http://nectar.org.au/research-cloud/
20. http://www.dis.uniroma1.it/challenge9/
21. Kolahdouzan, M., Shahabi, C.: Voronoi-based k nearest neighbor search for spatial network databases. In: VLDB. pp. 840–851 (2004)
22. Samet, H., Sankaranarayanan, J., Alborzi, H.: Scalable network distance browsing in spatial databases. In: SIGMOD. pp. 43–54. ACM (2008)
23. Lee, K.C., Lee, W.C., Zheng, B., Tian, Y.: Road: A new spatial object search framework for road networks. TKDE 24(3), 547–560 (2012)
24. Clarkson, K.L.: Fast algorithms for the all nearest neighbors problem. In: FOCS. pp. 226–232 (1983)
25. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: STOC. pp. 135–143 (1984)
26. Sankaranarayanan, J., Samet, H., Varshney, A.: A fast all nearest neighbor algorithm for applications involving large point-clouds. Computers & Graphics 31(2), 157–174 (2007)
27. Yu, C., Ooi, B.C., Tan, K.L., Jagadish, H.: Indexing the distance: An efficient method to knn processing. In: VLDB. vol. 1, pp. 421–430 (2001)