Beyond Millisecond Latency kNN Search on Commodity Machine

Bailong Liao, Leong Hou U, Man Lung Yiu, Zhiguo Gong

Abstract—The k nearest neighbor (kNN) search on road networks is an important function in web mapping services. These services are now dealing with rapidly arriving queries, that are issued by a massive amount of users. While overlay graph-based indices can answer shortest path queries efficiently, there have been no studies on utilizing such indices to answer kNN queries efficiently. In this paper, we fill this research gap and present two efficient kNN search solutions on overlay graph-based indices. Experimental results show that our solutions offer very low query latency (0.1ms) and require only small index sizes, even for 10-million-node networks.

Index Terms—Nearest neighbor searches, Spatial databases, Overlay networks

1 Introduction

Due to the increasing availability of smartphones[1] and the growing Internet penetration[2] over the world, web mapping services (Google Maps, Bing Maps, Yahoo! Maps, MapQuest) are now dealing with rapidly arriving queries, that are issued by a massive amount of desktop and mobile users. In this paper, we focus on a typical query called the kNN query, e.g., "find k nearest restaurants to my location q (in terms of driving distance)". Formally, given a point-of-interest (POI) dataset P (e.g., restaurants, gas stations) and a road network G, the kNN query reports k points $p_i \in P$ such that their shortest path distances $dist_N(q, p_i)$ from q are minimized.

We believe that, like modern search engines [1]–[3], big (web mapping) providers have been deploying clusters of commodity servers to handle high query rate and scale with the usage. By designing better optimizations to reduce the processing time per query, each server can provide a higher query throughput, thus enabling the service provider to cut the number of servers (and operational cost). For instance, if the processing time per query can be shrunk by factor 10, then we only need 1/10 of the original number of servers to achieve the same query throughput.

The key in boosting performance is to replace disk-based solutions by main-memory solutions, since main memory has a page access latency (50ns) significantly lower than that of hard disk (5ms). In the relational database area, both the academia and the industry have developed main-memory relational database systems, like Oracle TimesTen, VoltDB, H-store[3], to offer high performance boosts. Recently, in the spatial database area, Nutanong and Samet [4] have proposed memory-efficient algorithms for processing kNN queries on road networks.

In this paper, we aim to develop a compact main-memory index for road network such that: (i) it can fit into main memory, and (ii) it can process kNN queries efficiently. This is important in reducing the number of servers and the operational cost at the service provider.

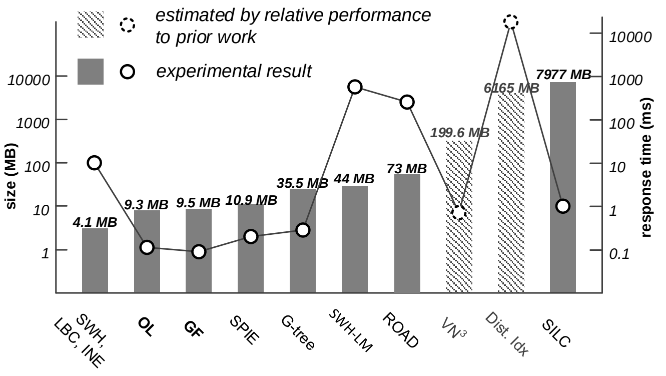


Fig. 1. Index overhead and response time of kNN solutions in the main memory scenario, for London road network (#nodes: 203,383, #edges: 267,628) and a dataset (2,040 POIs)

Except Ref. [4], most existing methods for road network kNN queries are disk-based solutions and do not carefully exploit a commodity machine's limited main memory size[4] (e.g., 8 GB). This raises an interesting question: "*Which existing method(s) are applicable in the main memory scenario?*" To answer this question, we plot the storage space and response time of existing methods (for a moderate-sized road network and a POI dataset) in Figure 1, where the measurements are obtained from our implementation or from previous experimental studies [5], [6]. Note that the storage space equates to the memory consumption in the main memory scenario. The bars indicate the main memory

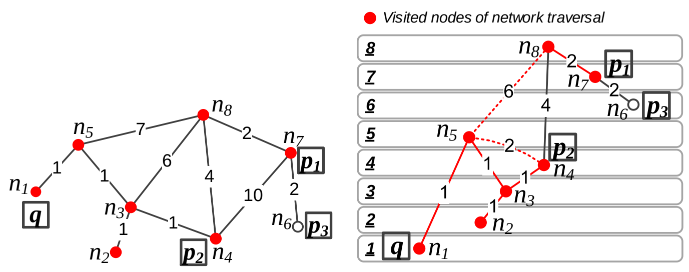
• *B.L. Liao, L.H. U, and Z.G. Gong are with the Department of Computer and Information Science, University of Macau, Macau. E-mails:* {`mb15573`, `ryanlhu`, `fstzgg`}`@umac.mo`
• *M.L. Yiu is with the Department of Computing, Hong Kong Polytechnic University, Hong Kong. E-mail:* `csmlyiu@comp.polyu.edu.hk`

1. http://www.go-gulf.com/blog/smartphone/
2. http://www.internetworldstats.com/stats.htm
3. http://voltdb.com/ http://hstore.cs.brown.edu/

4. Experiments in Ref. [4] used a commodity machine with 8 GB RAM.

size of 10 existing methods. For *no network indexing methods*[5], like INE [7], LBC [8], SWH [4], the space is only occupied by the road network G and POI dataset P (typically kept in an object index), shown as solid bars in the figure. Among these methods, SWH [4] outperforms LBC [8] and INE [7]. For *network indexing methods*, like G-tree [9], SWH-LM [4], SPIE [5], ROAD [6], SILC [10], the space is occupied by an index, indicated as slash bars in the figure. Observe that both the Distance Index [11] and the SILC Index [10] will soon exceed the main memory for larger moderate-sized road networks. In fact, SILC occupies $O(N^{1.5})$ space [10], which is super-linear to the the number of network nodes N. Moreover, ROAD [6] and SPIE [5] outperform Distance Index [11] and VN^3 [12], respectively, so we omit Distance Index and VN^3 from the experiments in this paper. We also test on the USA road network (#nodes: 23,947,347, #edges: 58,333,344), however, only few methods remain feasible (e.g., INE, LBC, SWH, G-tree, and SPIE). As a remark, SWH-LM is a variant of SWH that utilizes a landmark index [13], [14].

Contributions. In this work, we propose two memory efficient kNN search solutions, *Object-Last* (OL) and *Guide-Forest* (GF), that exploit the highway property of road networks [15], [16]. This property manifests a fact that a node is important if it is the *bridge* of many shortest paths, which is thoroughly explored in modern shortest path algorithms [16]–[20]. To utilize the highway property, these methods typically organize network nodes into a hierarchical graph structure, called an *overlay graph*.



(a) Bi-directional road network (b) Overlay graph

Fig. 2. kNN search on an overlay graph; node order: $n_1 \prec n_2 \prec n_3 \prec \cdots \prec n_8$

We elaborate how the overlay graph answers shortest path queries. In an overlay graph [16], every node is assigned into a hierarchy based on a node order. For example, suppose the node order is based on node IDs (e.g., $n_1 \prec n_2$), then we can transform the road network in Figure 2(a) into an overlay graph in Figure 2(b). The overlay graph excludes three existing edges $(e(n_3, n_8), e(n_4, n_7), e(n_5, n_8))$ that cannot fall on shortest paths, and includes two new shortcuts (i.e., dashed edges $e(n_4, n_5), e(n_5, n_8)$) that cover the necessary shortest paths passing through lower level nodes (e.g., $n_4 \rightarrow n_3 \rightarrow n_5$). To find the shortest path from n_s to n_t, it suffices to conduct a bi-directional search: a forward search from n_s, and a backward search from n_t.

5. For clarity, no network indexing means there is no road network index but a spatial index for objects (e.g., R-tree) may be used in these methods.

Unfortunately, such a method cannot support kNN search readily because the nodes of result objects (n_t) are unknown in advance. A naïve solution is to traverse the overlay graph in ascending network distance from q, i.e., visiting the nodes n_1, n_2, n_3, n_4, n_5, and n_8 in the above example. However, this approach would visit too many non-result nodes and defeat the purpose of using shortcuts.

In this work, we design two novel approaches that boost the performance of kNN queries by revisiting the structure of the overlay graphs. Our first approach, Object-Last (OL), reduces the search space by half through optimizing the overlay graph structure based on the objects' distribution. Our second approach, Guide-Forest (GF), adds guidance information into the overlay graph structure and searches on a heterogenous graph composed of both the original graph and the overlay graph. Finally, we propose to extend our solutions for queries on multiple object types and for range queries in Section 5.

We experimentally evaluate OL and GF on a set of real road networks. The experimental results demonstrate that our approach is an order of magnitude faster than most existing approaches (including SWH [4], G-tree [9], ROAD [6], and SPIE [5]) and the size overhead is negligible (just 30% to 60% times larger than the road network). This enables us to process kNN queries in very large road networks. To the best of our knowledge, we are the first work that offers **very low latency (0.1ms)** per query on **10-million-node networks** on a **commodity machine**. This translates to a query throughput of 10,000 queries per commodity machine per second.

The rest of the paper is organized as follows. We give the problem definition, preliminary background, and problem challenges in Section 2. In Section 3 and Section 4 we present our proposed solutions OL and GF respectively. Section 5 extends our solutions to support queries on multiple object types and for range queries. We experimentally evaluate our methods in Section 6 and discuss related work in Section 7. Finally, we conclude our work and discuss future work in Section 8.

2 Preliminaries

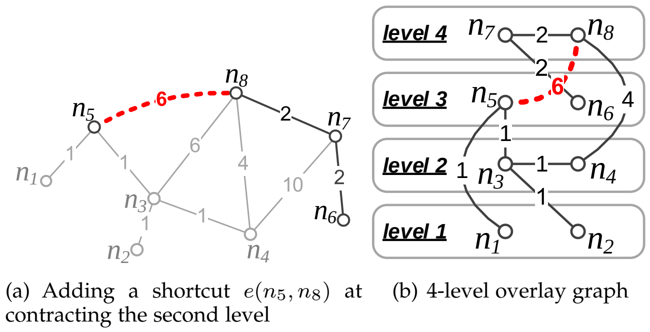
2.1 Problem Definition and Settings

In this work, we focus on the k nearest neighbor (kNN) search in a road network. Given a query location q, a set of objects P, and a road network $G = (N, E, W)$ where $w_i \in W$ indicates the weight of edge $e_i \in E$, our problem is to find a set of k objects P_{NN} where their network distances to q are not worse than any other object $p' \in P \setminus P_{NN}$. The network distance between q and p, denoted as $d_N(q, p)$, is the shortest path distance from q to p according to the edge weights W. Our objective is to develop a compact index so that (i) it can fit in main memory, and (ii) it supports efficient kNN search.

Like existing work [4]–[6], [10]–[12], we assume that both queries and objects are always located at the nodes of G. Nevertheless, our techniques can be extended to deal with objects located on edges E, by simply regarding each object as a network node.

2.2 Overlay Graph

Recent experimental studies [20], [21] show that the overlay graph is the most effective for answering shortest path queries, as it offers fast response time, low construction cost, and small index overhead. In general, an overlay graph can be viewed as a hierarchial structure of the original graph, where the hierarchy captures the importance of the nodes. For instance, Figure 3(b) illustrates an overlay graph G^o derived from Figure 3(a). To answer a shortest path query in G^o, we traverse only important nodes such that the search space is significantly reduced.



(a) Adding a shortcut $e(n_5, n_8)$ at contracting the second level (b) 4-level overlay graph

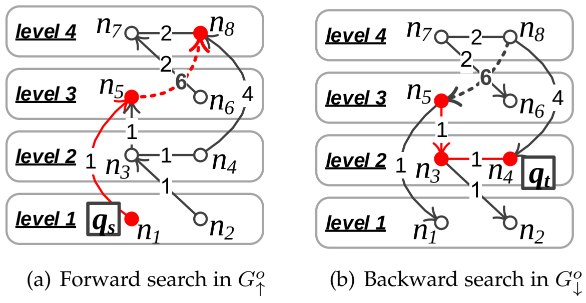
Fig. 3. Overlay graph

To construct an overlay graph G^o, a typical approach is to prioritize the nodes of G into a set of level layers $\mathcal{L}^1, \mathcal{L}^2, \cdots, \mathcal{L}^\ell$ and to iteratively *contract* the nodes in ascending level order. Specifically, the contraction process aims to remove unpromising edges from and add shortcuts into the overlay graph. When contracting a node n at level $\mathcal{L}^i$, an adjacent edge $e(n, n')$ of n is removed from the overlay graph if $e(n, n')$ is not the shortest path from n to n' (i.e., $w(n, n') > d_N(n, n')$). Besides, for any adjacent node pair n_i and n_j of layer $\mathcal{L}^i$, if their unique shortest path covers n and both n_i and n_j are located at a higher level than n (i.e., $n \prec n_i, n_j$), then we insert a shortcut $e(n_i, n_j)$ of weight $d_N(n_i, n_j)$ into the overlay graph.

We have shown an overlay graph with 8 levels in Figure 2(b). In the following example, suppose we categorize the nodes in Figure 3(a) into 4 levels (i.e., $\{n_1, n_2\} \prec \{n_3, n_4\} \prec \{n_5, n_6\} \prec \{n_7, n_8\}$), as shown in Figure 3(b). When contracting the second level $\mathcal{L}^2$, we retain $e(n_3, n_4)$, $e(n_3, n_5)$, $e(n_4, n_8)$ in the overlay graph. However, we discard $e(n_3, n_8)$ and $e(n_4, n_7)$ as they are not on any shortest path. In addition, we add a shortcut $e(n_5, n_8)$ (the dashed line in the figure) since both n_5 and n_8 are adjacent to $\mathcal{L}^2$ and n_3 and n_4 fall on the shortest path between n_5 and n_8. Note that we do not add $e(n_5, n_7)$ as a shortcut since the shortest path from n_5 to n_7 passes through an edge at a higher contraction level (i.e., $e(n_7, n_8)$).

Definition 1 (Upward and downward graphs)**.** *Given an overlay graph $G^o = (N^o, E^o)$ and the contraction order $\mathcal{L}$, an edge $e(n, n') \in E^o$ is in the upward overlay graph $G^o_\uparrow$ if and only if n is contracted no later than n'. Verse vice, an edge $e(n, n') \in E^o$ is in the downward overlay graph $G^o_\downarrow$ if and only if n is contracted no earlier than n'.*

Specifically, the overlay graph G^o consists of an *upward graph* $G^o_\uparrow$ and a *downward graph* $G^o_\downarrow$ (see Definition 1). For example, $e(n_3, n_4)$ is included in both upward and downward graphs since n_4 is neither contracted earlier nor later than n_3 (i.e., $n_3, n_4 \in \mathcal{L}^2$). To find a shortest path from a source n_s to a destination n_t, typical overlay graph query processing [17] conducts a bi-directional search with (i) a *forward search* from n_s (on $G^o_\uparrow$) and (ii) a *backward search* from n_t (on $G^o_\downarrow$). We illustrate the graphs $G^o_\uparrow$ and $G^o_\downarrow$ of the running example in Figure 4. Suppose that n_1 is the source and n_4 is the destination. Since both the forward and backward searches settle at n_5 and $d_N(n_1, n_5) + d_N(n_5, n_3) + d_N(n_3, n_4)$ is minimum, the bi-directional search returns $n_1 \to n_5 \to n_3 \to n_4$ as the shortest path. The correctness of the bi-directional search on the overlay graph is secured by the shortcuts (see [17] for a discussion).



(a) Forward search in $G^o_\uparrow$ (b) Backward search in $G^o_\downarrow$

Fig. 4. Bi-directional search in upward and downward graphs

Performance of overlay graph solutions. Bast et al. [22] observe that real road networks typically have small values of maximum node degree Δ[6] and low *highway dimension* h, where h is defined as the number of "important" nodes (which fall on many shortest paths) within a region. Thereby, the number of shortcuts is very small in practice. As reported in [17], the size overhead ratio is negative since the number of omitted edges (e.g., 2 edges are omitted in our running example) is more than newly added shortcuts (e.g., only 1 shortcut is added in Figure 3(b)). Thus, the overlay graph solutions incur fast response time and low storage overhead for the shortest path queries.

2.3 Challenges of kNN Search on Overlay Graphs

An overlay graph is a compact index for answering shortest path queries efficiently. This inspires us to investigate whether it can also be applied to answer kNN queries efficiently. Unfortunately, no existing work has attempted to answer kNN queries on overlay graphs, to the best of our knowledge.

We identify two challenges in conducting kNN search on the overlay graph: (i) the locations (nodes) of result objects are unknown in advance, and (ii) the construction of overlay graphs ignores the distribution of objects. For illustration, consider the example in Figure 2(b) and assume that $k = 1$, query point q is at n_1, and POIs are at n_4, n_6, n_7. Recall that the shortest path search on the overlay graph is a bidirectional search that requires both the source n_s and the destination n_t. Since the NN of q is unknown in advance, a naive solution is to run shortest path searches (from q)

6. The maximum node degree Δ is 9 of all evaluated road networks in our experiments.

to each POI (i.e., nodes n_4, n_6, n_7). Obviously, this is very expensive. In the worst case, some object (e.g., at n_6) resides at the leaf node of overlay graphs, causing the shortest path search to examine many nodes in overlay graphs.

A better approach is to run the Incremental Euclidean Restriction method (IER) [7], which incrementally retrieves candidate objects o in ascending order of their Euclidean distances $d_E(q, o)$ from q, and computes their shortest path distances $d_N(q, o)$ from q (by calling the overlay graph solution). However, the drawback of this approach is that it may examine some edges multiple times (during shortest path computation for different candidates). E.g., following the above example, edges $e(n_1, n_5), e(n_5, n_8)$ are examined multiple times. Also, the cost of this method is very sensitive to the number of results k.

In this paper, we aim to revisit the overlay graph structure in order to boost the performance of kNN queries. Our first approach, called Object Last (OL), addresses the second challenge. In the construction of the overlay graphs, we rearrange the node contraction order so that the nodes with objects are contracted later (and thus they appear in higher levels of overlay graphs). Thereby, the kNN result can be found by only forward search. Our second approach, called Guide Forest (GF), addresses the first challenge. Its idea is to add a guidance information to each node in overlay graphs, where the guidance information indicates the object existence of the corresponding subtree. During kNN search, these information guide the pruning of subtrees (that do not contain any objects). This elegant approach significantly reduces the search space of kNN queries. In our experiments, GF is an order of magnitude faster than the state-of-the-art competitors (including SWH, G-tree, ROAD, and SILC).

3 Object-Last Hierarchies

In this section, we develop a solution called *Object-Last hierarchies* (OL), which exploits the distribution of objects such that every node containing objects is rearranged to the last (top) level of the upward graph. This enables the kNN result of any query to be found by traversing edges in $G_\uparrow^{OL}$ only. OL reduces the query response time significantly since it reduces the search space (i.e., using only the upward graph instead of two overlay graphs).

In the following, we first introduce the construction of OL. Then, we show the query processing and correctness subsequently. Lastly, we discuss the effectiveness of OL.

3.1 Construction

Intuitively, if the network contains only one object p (located at node n), the ideal contraction order is to move n to the last level. In this way, the forward search always discovers p, regardless of the location of the query node n_q.

We can extend this idea to support multiple objects. Let $\mathcal{L} = \{\mathcal{L}^1, \cdots, \mathcal{L}^\ell\}$ be the node contraction layers (decided by the method in [15]). We contract a node n if it has no object. Otherwise, we omit n from contraction and rearrange it to the *object layer* $\mathcal{L}^{obj}$. This arrangement forces all objects to appear at the last (top) level of the overlay graph. As an example, suppose that we follow the contraction order in Figure 3. Our proposed object-last overlay graph $G_\uparrow^{OL}$

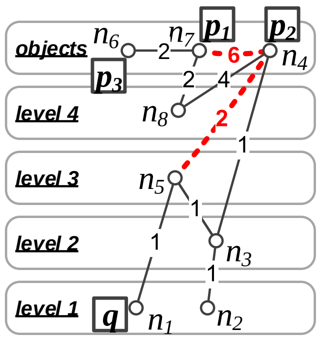


Fig. 5. Illustration of the Object-Last Overlay Graph

is illustrated in Figure 5. We rearrange the contraction of n_4, n_6, n_7 to the layer $\mathcal{L}^{obj}$ since they have objects inside. The object-last rearrangement constructs two shortcuts in total, where $e(n_5, n_4)$ and $e(n_4, n_7)$ are added when contracting $\mathcal{L}^3$ and $\mathcal{L}^4$, respectively.

Algorithm 1 Object-Last Hierarchies Construction

```
   $H$: heap, $N_{Obj}$: set of object nodes, $G_\uparrow^{OL}$: forward graph
   **ConstructOL** ( Graph $G = (N, E)$, Objects $P$ )
1: decide a node contraction order $\mathcal{L} = \{\mathcal{L}^1, \cdots, \mathcal{L}^\ell\}$ ▷ by [15]
2: $\mathcal{L}^{obj} := \{\emptyset\}$                                ▷ object level
3: for $i$:= 1 to $\ell$ do
4:     for $n \in \mathcal{L}^i$ do                      ▷ adding edges in $E^o$
5:         if $n$ has object(s) then add $n$ into $\mathcal{L}^{obj}$
6:         else, contract $n$ and add edges into $G_\uparrow^{OL}$
7: contract $n \in \mathcal{L}^{obj}$ and add edges into $G_\uparrow^{OL}$
8: return $G_\uparrow^{OL}$
```

Algorithm 1 shows the pseudo code of the OL construction. We only need to construct the upward overlay graph $G_\uparrow^{OL}$ of OL since the query processing of OL only utilizes the *forward search* (to be discussed shortly). The construction contracts nodes by the contraction order (i.e., constructing shortcuts and adding necessary adjacent edges into $G_\uparrow^{OL}$) and adds object nodes into the object level $\mathcal{L}^{obj}$. Finally, it contracts the object nodes in $\mathcal{L}^{obj}$ (i.e., only adding adjacent edges into $G_\uparrow^{OL}$ but not constructing any new shortcut).

3.2 Query processing

To process a kNN query, OL applies the forward search in the OL upward graph $G_\uparrow^{OL}$ only. The pseudo code is shown in Algorithm 2. We prove its correctness in Lemma 1.

Figure 5 illustrates how this algorithm answers the 2NN query at the query node n_1. Initially, the heap content is: $H = (n_1, 0)$. First, we deheap n_1 and enheap entry $(n_5, 1)$ for its (outgoing) neighbor. Second, we deheap n_5 and enheap entry $(n_4, 3)$ for its (outgoing) neighbor. Third, we deheap n_4 and discover object p_2; also we enheap $(n_7, 9)$ for its (outgoing) neighbor. Finally, we deheap n_7 and discover object p_1; also the search terminates. In this example, OL obtains the 2NN result by only visiting 4 nodes.

Lemma 1. *OL computes the correct kNN result.*

Proof. We first show that OL always returns correct 1st NN by the forward search in $G_\uparrow^o$. Consider two cases for the

Algorithm 2 OBJECT-LAST HIERARCHIES kNN

```
   $H$: heap, $R$: $k$NN result
   **QueryOL** ( Query node $n_q$, OL graph $G_{\uparrow}^{OL}$, Size $k$ )
1: add $(n_q, 0)$ into $H$
2: **while** $H$ is not empty or $|R| < k$ **do**
3:     pop $(n_i, \delta)$ from $H$; visit $n_i$    ▷ in ascending order to $\delta$
4:     **if** $n_i$ has object(s) **then** add $n_i$'s object(s) into $R$
5:     **for all** $e(n_i, n_j) \in E_{\uparrow}^{OL}$ and $n_j$ is not visited **do**
6:         add or update $(n_j, \delta + w(n_i, n_j))$ into $H$
7: **return** the first $k$ result of $R$
```

location of query q: (i) q at object node and (ii) q at non-object node. The former is trivial and the latter is secured since there is no object node having lower hierarchy level than the node of q.

Regarding the remaining neighbors (e.g., the k^{th} NN p_{kNN}), the search can be viewed as one of the following cases.

If $SP(q, p_{kNN})$ **does not cover any prior neighbor(s),** then the shortest path can be found by the forward search in $G_{\uparrow}^{o}$.

Otherwise, $SP(q, p_{kNN})$ is composed of two parts $SP(q, p_{iNN})$ and $SP(p_{iNN}, p_{kNN})$. $SP(p_{iNN}, p_{kNN})$ is correct since every possible path at the object layer is secured by the *shortcuts* (cf. the contraction process). Thus, $SP(q, p_{kNN})$ must be the shortest path.

□

3.3 Effectiveness of OL

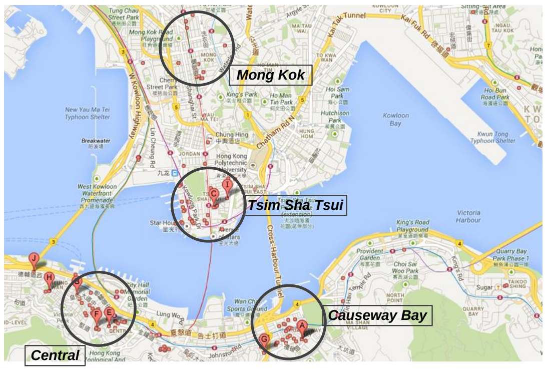


Fig. 6. Skewed distribution of POIs; distribution of 'boutique' in Hong Kong (using GoogleMap)

In the worst case, the number of additional shortcuts can reach $|N_{Obj}|^2$ (i.e., every object node pair has a shortcut connected.), where N_{Obj} is set of nodes containing object(s). In this section, we explain why the number of object shortcuts is practically small due to the clustered distribution of objects.

Observe that the distribution of shops in real world follows a clustered distribution. This cluster effect[7] occurs because a cluster of shops can attract much more customers than individual shops. We verify this effect in GoogleMap search; Figure 6 illustrates that 'boutique' shops in Hong Kong follow a clustered distribution.

7. https://en.wikipedia.org/wiki/Cluster_effect

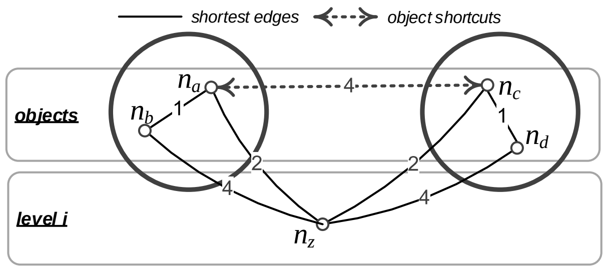


Fig. 7. The effect of clustered distribution of objects

For a clustered distribution of objects, not every pair of object nodes has to build a shortcut. In the example of Figure 7, 4 object nodes are distributed into 2 clusters: $\{n_a, n_b\}, \{n_c, n_d\}$. We may build at most four bi-directional shortcuts (i.e., $e(n_a, n_c)$, (n_a, n_d), $e(n_b, n_c)$, and $e(n_b, n_d)$) since their shortest paths fall on a non-object node n_z at level i. However, the shortcuts in-between two clusters can be viewed as the connection paths for internal object nodes. Thus, $e(n_a, n_d)$, $e(n_b, n_c)$, and $e(n_b, n_d)$ need not be built since their shortest paths pass through shortcut $e(n_a, n_c)$. In our experiments, the number of additional shortcuts is 1.5% of $|N_{Obj}|^2$ in the Los Angeles road network when the object ratio is 1%.

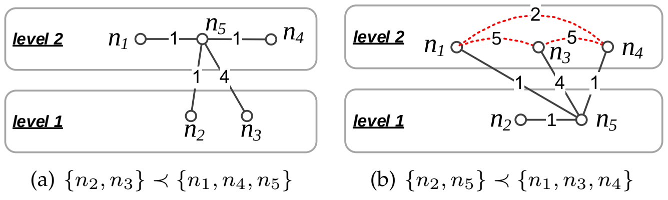


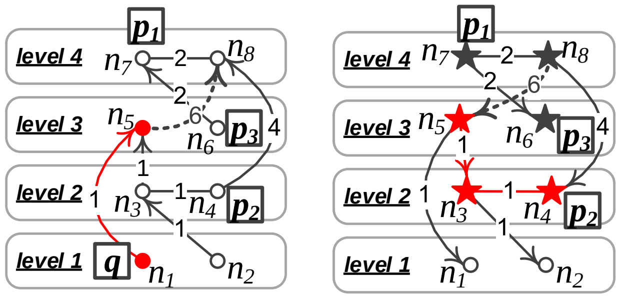
Fig. 8. The effect of contraction order

OL seems to be a promising solution to process kNN queries as the entire search space is cut in half (i.e., the search only traverses in $G_{\uparrow}^{OL}$). However, the contraction of the object-last arrangement may no longer follow the principle of the overlay graph solutions (i.e., contracting nodes according to their importance). We use a concrete example to demonstrate the effect of contraction order in Figure 8. According to the topology of the road network, n_5 is an important node (cf. [17]) as the majority of the shortest paths in this network go through n_5. Suppose the contraction order is $\{n_2, n_3\} \prec \{n_1, n_4, n_5\}$, there is no need to add any shortcut since n_5 is contracted at the last level. However, if we swap n_5 and n_3, then the contraction adds 3 shortcuts to $G_{\uparrow}^{OL}$. This example shows the importance of the contraction order in the overlay graph construction.

Summary. OL offers low response time for kNN queries since it successively applies the forward search paradigm of the overlay graph by introducing the concept of object-last arrangement. However, it suffers from two drawbacks. First, the index size (e.g., number of shortcuts) is higher than the original overlay graph. Second, the index maintenance is costly since the entire structure may be affected by a few object updates.

4 Guide Forest Hierarchies

Although OL provides low response time for kNN queries, it loses some promising characteristics of the overlay graph based solutions, e.g., light index overhead and fast construction/maintenance. We proceed to investigate an alternative approach that avoids the drawback of OL.



(a) upward graph $G^o_\uparrow$ (b) downward graph $G^o_\downarrow$ (with guidance labels '$\star$')

Fig. 9. Guide forest hierarchies

In this section, we reuse both overlay graphs $G^o_\uparrow, G^o_\downarrow$ to guide the search of kNN queries. We design a method called *Guide Forest* (GF) since it utilizes guidance information to accelerate kNN search in a forest (i.e., overlay graphs). To obtain such guidance information, we use the reachability of the downward graph $G^o_\downarrow$ to define the concept of *object-reachable* node (see Definition 2). In Figure 9(b), each object-reachable node is labeled with '$\star$', e.g., $n_3, n_4, n_5, n_6, n_7, n_8$. Note that n_2 is not object-reachable since it does not have any outgoing edge in $G^o_\downarrow$.

Definition 2 (Object reachability in $G^o_\downarrow$)**.** *Given a downward graph $G^o_\downarrow$, a node n is called object-reachable if there exists a path (in $G^o_\downarrow$) from n to some object node.*

Intuitively, during kNN search, we can skip relaxing an edge $e(n_i, n_j)$ if n_j is not object-reachable. We illustrate this idea with the kNN query q in Figure 9. First, we begin the forward search at q in the upward graph $G^o_\uparrow$. When we reach n_5, we check the downward graph $G^o_\downarrow$ and finds an edge of n_5: $e(n_5, n_3)$. Suppose n_3 is the next node to traverse. We only relax edge $e(n_3, n_4)$ but not edge $e(n_3, n_2)$ since n_2 is not object-reachable. The correctness of this approach is obvious since no object exists in n's downward graph if n is not object-reachable (by Definition 2). This elegant approach guides the search towards object nodes, thereby significantly boosting the performance of kNN search.

However, there are some unresolved issues in GF: ($\mathcal{I}1$) *How do we construct the guidance information and maintain it for updates?*, ($\mathcal{I}2$) *How to exploit such guidance information to speedup kNN search?*, ($\mathcal{I}3$) *Can we optimize the overlay graph for kNN queries?*, and ($\mathcal{I}4$) *Can we adopt other guidance information?* In the remaining of this section, we carefully address these concerns and thoroughly analyze our solution.

4.1 Bit-array construction and maintenance

In this work, we use a concise data structure called *bit-array* $\mathcal{B}$ as the guidance information. We indicate the object reachability of a node n_i by the bit $\mathcal{B}(n_i)$ (i.e., '$\star$' in Figure 9(b)). This bit-array is a compact structure that only requires $N/8$-byte space overhead.[8] Its size is negligible as compared to other existing kNN solutions (cf. Figure 1). In addition, this bit-array also provides good construction time and low maintenance cost, which are discussed shortly.

Bit-array construction. First we suppose the basic upward and downward graphs are constructed in the same way as in other overlay graph solutions (cf. Section 2.2). In this subsection, we discuss how to compute the bit-array $\mathcal{B}$ for object-reachable nodes. Given the downward graph $G^o_\downarrow$, we can determine the bit value $\mathcal{B}(n_i)$ based on the spread of influence. Initially, we set each bit value $\mathcal{B}(n_i)$ to *false*. Then we access the object nodes in ascending contraction order. To examine the object reachability of n, we apply a reverse network traversal from n. We first add n into a queue S and set $\mathcal{B}(n)$ to *true*. In each iteration, we dequeue a node n_i from S. For each incoming edges $e(n', n_i)$ in $G^o_\downarrow$, we update $\mathcal{B}(n')$ to *true* and add n' into S only if $\mathcal{B}(n')$ is *false* (i.e., n' is not yet discovered by any object). We repeat the above process until S becomes empty. In Figure 9(b), n_4 is the first object node to be examined according to the contraction order and its reverse search marks five nodes, i.e., n_3, n_4, n_5, n_7, and n_8, as *true*. The next object node to be examined is n_6 and its reverse search relaxes only one edge $e(n_7, n_6)$ since n_7 is already marked as *true*. The time complexity of the bit-array construction is the same as a complete reverse network traversal, which takes $O(E+S)$ time where E indicates the number of edges in the road network and S indicates the number of shortcuts created by the overlay graph solutions (cf. Section 2.2).

Algorithm 3 Bit-array Construction

```
    $S$: Queue; $\mathcal{B}$: Bit-array
    **ConstructBitArray** ( Overlay Graph $G^o_\downarrow$, Objects $P$ )
 1: initialize each value of $\mathcal{B}$ to $false$
 2: sort $P$ according to the contraction order
 3: **for** $p \in P$ in this order **do**
 4:     $n_i$ := the node containing $p$
 5:     $\mathcal{B}(n_i) := true$; add $n_i$ into $S$
 6:     **while** $S$ is not empty **do**
 7:         $n_i := pop(S)$
 8:         **for** $n_i$'s incoming edge $e(n', n_i) \in E^o_\downarrow$ **do**
 9:             **if** $\mathcal{B}(n') = false$ **then**
10:                 $\mathcal{B}(n') := true$; add $n'$ into $S$
11: **return** bit-array $\mathcal{B}$
```

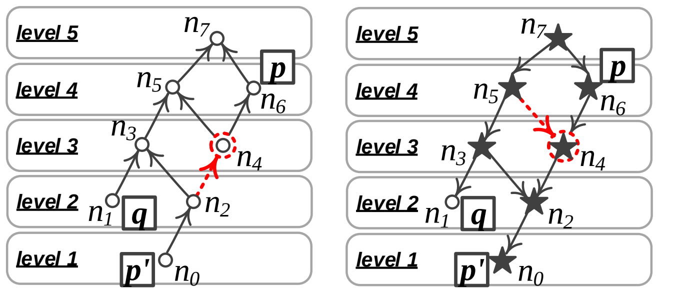
Bit-array updates. The bit-array can be maintained incrementally with respect to object updates. For the insertion of a new object, we first check whether the newly inserted object affects the bit value of its located node. If the node is affected, we add it into S and apply a partial reverse traversal to discover the influenced nodes. This procedure terminates when there is no more influenced node found. Similarly, we can handle the object deletion in similar approach but for each traversed node we need to further check whether the deleted object is the only object it can reach. If yes, we unmark it and enqueue it; otherwise, we do nothing. Note that the worst-case time complexity of the maintenance is the same as the construction time. In

8. This bit-array is compact in practice. Even for a very large road network (US, #node 23,947,347), the bit-array size is just 2.85 MB.

practice, the maintenance cost is much smaller than the construction time since the structure is maintained for the set of nodes influenced by the update.

4.2 Query processing

To process a kNN query, a simple approach is to traverse the edges in both upward and downward graphs based on the bit-array. However, this approach searches back and forth in graphs $G^o_\uparrow$ and $G^o_\downarrow$, causing unnecessary network traversal. Figure 10 shows a worst-case example where most of the nodes are object-reachable (those labeled with '$\star$'). Suppose p is the first NN of the query q (at n_1). In this example, the network traversal (at n_1) discovers n_4 via two paths: $n_1 \rightarrow n_3 \rightarrow n_2 \rightarrow n_4$, and $n_1 \rightarrow n_3 \rightarrow n_5 \rightarrow n_4$. Recall that, in Section 2.2, based on the bi-directional search of the overlay graph solution, the shortest path between n_1 and n_4 is settled at n_5. It means that the traversal from n_2 to n_4 is unnecessary, and it should be omitted during kNN search.



(a) upward graph $G^o_\uparrow$ (b) downward graph $G^o_\downarrow$ (with bit-array)

Fig. 10. Query processing in GF

Algorithm 4 GUIDE-FOREST HIERARCHIES kNN

```
    $H$: heap, $R$: $k$NN result
    **QueryGF** ( Query node $n_q$, GF graphs $G^o_\uparrow, G^o_\downarrow$, Bit-array $\mathcal{B}$, Size $k$ )
1:  add $(n_q, 0, \uparrow)$ into $H$
2:  while $H$ is not empty and $|R| < k$ do
3:      pop $(n_i, \delta, \rho)$ from $H$; visit $n_i$  ▷ in ascending order to $\delta$
4:      if $n_i$ has object(s) then add $n_i$'s object(s) into $R$
5:      if $\rho = \uparrow$ then  ▷ preceding path on upward edges only
6:          for all $e(n_i, n_j) \in E^o_\uparrow$ and $n_j$ is not visited do
7:              add or update $(n_j, \delta + w(n_i, n_j), \uparrow)$ into $H$
8:      for all $e(n_i, n_j) \in E^o_\downarrow$ and $n_j$ is not visited do
9:          if $\mathcal{B}(n_j)$ is true then
10:             add or update $(n_j, \delta + w(n_i, n_j), \downarrow)$ into $H$
11: return the first $k$ result of $R$
```

Algorithm 4 shows the complete pseudo code of querying kNN in GF, which applies network traversal on two overlay graphs. Each heap entry (n_i, δ, ρ) is associated with a boolean flag ρ that determines the way of relaxing edges. Only when ρ is $\uparrow$, we relax edges in the upward graph (lines 5–7). For edges in the downward graph, we relax them when the adjacent node n_j is object-reachable (line 9). We prove its correctness in Lemma 2.

As an example, suppose that all edges have the same length (1 unit) in Figure 10. We illustrate the running steps of Algorithm 4 in the following table. The search terminates when we deheap the first object node n_0.

iteration	*heap content*	*settled nodes*
1	$(n_1, 0, \uparrow)$	/
2	$(n_3, 1, \uparrow)$	n_1
3	$(n_5, 2, \uparrow), (n_2, 2, \downarrow)$	n_1, n_3
4	$(n_2, 2, \downarrow), (n_4, 3, \downarrow), (n_7, 3, \uparrow)$	n_1, n_3, n_5
5	$(n_0, 3, \downarrow), (n_4, 3, \downarrow), (n_7, 3, \uparrow)$	n_1, n_3, n_5, n_2
6	$(n_4, 3, \downarrow), (n_7, 3, \uparrow)$	n_1, n_3, n_5, n_2, n_0

Lemma 2. *GF computes the correct kNN result.*

Proof. As shown in Section 2.2, the network traversal in the overlay graph always returns the proper shortest path from q to any node in the network. Thereby, the correctness of the query processing can be viewed as a problem: *Do we miss any object by applying our downward pruning (in line 9) and upward pruning (in line 5)?*

The correctness of the downward pruning is trivial; if $\mathcal{B}(n_j)$ is *false*, then there is no object in any reachable node from n_j in $G^o_\downarrow$.

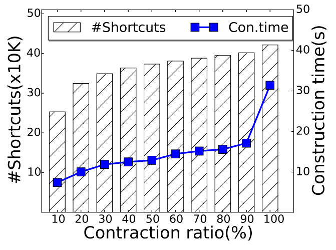
We show the correctness of the upward pruning as follows. According to the bi-directional search of the overlay graph, the shortest path between n_q and n_j must settle at a node n_z with a non-lower order, i.e., $n_q \preceq n_z \wedge n_j \preceq n_z$. Moreover, the node order of the paths from n_q to n_z and from n_j to n_z is monotonic increasing. For any node n_k whose node order is higher than n_j, the shortest path from n_q to n_k cannot pass through n_j due to the order monotonicity. Thereby, the upward pruning preserves the proper shortest path from q to any node. $\square$

Complexity analysis. For the ease of the analysis, we simply reuse the notations of [15], where N indicates the number of nodes in the graph, h denotes the highway dimension (cf. Section 2.2), Δ indicates the maximum degree of the road network, and D donates the diameter of the road network. According to [15], the number of visited nodes for any shortest path query in an overlay graph is bounded by $O(h \log N \log D)$. Suppose the degree of a node in the overlay graph is F (i.e., $\Delta + h \log N \log D$ [15]), the time complexity of the forward search in GF is bounded by $O(F \cdot h \log N \log D)$. Thus processing a kNN query in GF takes $O(F \cdot (h \log N \log D)^2)$ time as every relaxed node of the forward search may issue an individual downward search in the worst case.

4.3 Optimizing the overlay graph for kNN queries

In this section, we optimize the overlay graph structure by two techniques, *contraction early termination* and *local contraction rearrangement*. These two techniques not only reduce the index size but also boost the query performance.

Contraction early termination. First, we observe that the guidance information at the last few contraction levels are not effective since almost every node is object reachable, i.e., $\mathcal{B}(n)$=*true* (cf. Section 4.1). In other words, the query processing must relax edges along both search directions (i.e., $G^o_\downarrow$ and $G^o_\uparrow$) at the top of the overlay graph such that no search space can be pruned. To make things worse, the construction becomes costly at the tail-end since more shortcuts are inserted into the overlay graph due to the graph density. We demonstrate this finding in Figure 11 that plots the construction time and the number of added shortcuts along the contraction processing (in terms of the contraction ratio) for the Los Angeles and California road networks. For instance, the last 10% contractions in the Los Angeles road network takes around 50% of the total construction time.


(a) LA

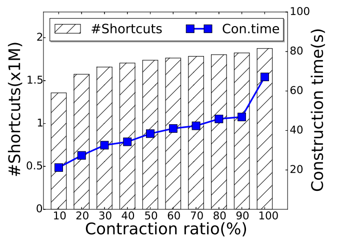

(b) CA

Fig. 11. #Shortcuts and construction time among the contraction ratio

Obviously, the effectiveness of the bit-array is inversely proportional to the contraction ratio. This leads to an interesting idea: *Can we early terminate the contraction processing?*

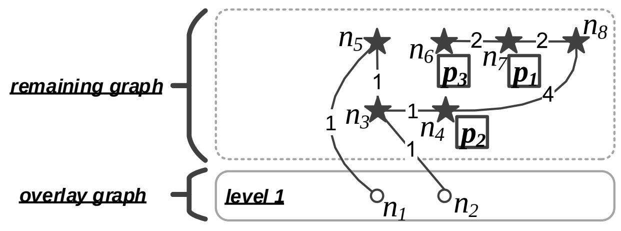


Fig. 12. A heterogenous structure

Inspired by the above discussion, we attempt to terminate the contraction process early subject to the effectiveness of the guidance information. An illustration is given in Figure 12 based on our running example, where the contraction process is terminated while the nodes of the remaining levels (e.g., $\mathcal{L}^2$, $\mathcal{L}^3$, and $\mathcal{L}^4$) are all object reachable. This strategy returns a heterogenous network that consists of two graphs, (1) a partial overlay graph and (2) a partial original graph with shortcuts.

Regarding the query processing, relaxing a node in the overlay graph and the remaining graph follows the procedure of Algorithm 4 and Dijkstra's algorithm, respectively. The correctness of the query processing is easily verified since the individual traversals in the overlay graph and the remaining graph are secured by Lemma 2 and Dijkstra's algorithm.

According to the effect of the contraction ratio (cf. Figure 11), the early contraction termination offers lower construction time and smaller index overhead. In this work, we recommend to terminate at level i where the remaining nodes (from level i to ℓ) are all object reachable. This termination strategy likely optimizes the query performance which is shown as follows.

- **After level i:** Suppose the process contracts level i and creates ξ shortcuts. Note that every node after level i must be object reachable so that these ξ shortcuts cannot help to skip the search at level i (i.e., each node must relax the edges till level i due to the downward reachability). However, the network traversal has to relax ξ more edges. Thus, the performance of this strategy is worse than or equal to the level i termination.
- **Before level i:** If we terminate to contract at a level lower than i, then there are a group of non-reachable nodes in the remaining graph. We cannot use the reachability to prune their relaxations in the remaining graph since they are at the same level with other object reachable nodes. However, if we postpone our contraction, some of these nodes are marked as *false* (i.e., not object reachable) at a level lower than i so that some of their relaxations can be omitted. Thus, the level i termination likely outperforms this strategy especially when the number of additional shortcuts are moderate.

To early terminate the contraction process, we progressively maintain the bit-array along the contraction process. When contracting an object node, we reversely search all object reachable nodes and mark them as *true*. For instance, when contracting n_4 in Figure 12, all remaining nodes (n_3, n_5, n_6, n_7, and n_8) are marked as *true* by the reverse search process so that the contraction process is terminated.

Local contraction rearrangement. The second optimization is a heuristic which attempts to postpone the contraction of object nodes. The main difference from OL is that we only arrange the contraction order locally instead of pushing every object to the last level (cf. Section 3). According to our discussion, the effectiveness of GF is sensitive to the number of *true* values in the bit-array. However, if we simply minimize the number of *true* values (i.e., equivalent to OL), then this approach may construct many additional shortcuts which significantly affects the query performance and increases the space overhead of GF. We arrive at a compromise that we only rearrange the contraction order in a local scope. Given the contraction levels, every node is only allowed to rearrange within λ levels such that the number of *true* values in the bit-array is minimized. The effectiveness of this approach is shown in our experimental study.

4.4 Additional guidance information

In this subsection, we present an additional guidance information, called distance-array $\mathcal{D}$, to improve the performance of GF.

Distance-array $\mathcal{D}$ construction. Like the bit-array, the distance-array $\mathcal{D}$ indicates the reachability information but the distance-array keeps the distance of the nearest reachable object (in both downward and upward graphs) instead of the bit-flags. For instance, $\mathcal{D}(n_5)$ in Figure 9(b) is 2 (i.e., the shortest distance from n_5 to p_2 in G^o). $\mathcal{D}$ is constructed by two network traversals, i.e., a *bottom-up traversal* in $G^o_\downarrow$ and a *top-down traversal* in $G^o_\uparrow$. In the bottom-up traversal, we use the same strategy of Algorithm 3 to find the distance of the nearest reachable object in $G^o_\downarrow$. More specifically, we first initialize all $\mathcal{D}(n_i)$ values to ∞. When updating $\mathcal{D}(n')$ (Line 10 of Algorithm 3), we set $\mathcal{D}(n')$ to $\mathcal{D}(n_i) + w(n', n_i)$

only if $\mathcal{D}(n') > \mathcal{D}(n_i) + w(n', n_i)$ (i.e., a nearer downward reachable object is found). In the top-down traversal, we traverse $G^o_\uparrow$ and update the reachability information from the highest contraction order to the lowest order. More specifically, we update $\mathcal{D}(n')$ (i.e., $e(n', n_i) \in E^o_\uparrow$) only if $\mathcal{D}(n') > \mathcal{D}(n_i) + w(n', n_i)$ (i.e., a nearer upward reachable object is found). After these two traversals, $\mathcal{D}(n_i)$ records the distance to the nearest object reachable from n_i.

Query processing. Next we show how Algorithm 4 is extended to support the distance-array. When adding n_j into H (Line 10 of Algorithm 4), we update the weight of n_j by additionally summing the downward reachable distance $\mathcal{D}(n_j)$, i.e., $\delta + w(n_i, n_j) + \mathcal{D}(n_j)$. This change does not affect the correctness since the weight of n_j is bounded by the distance to the first downward reachable object.

Comparison with the bit-array. Obviously the distance-array approach outperforms the bit-array approach since the distance-array offers more informative information to discover the nearest neighbor objects. The tradeoff is that the distance-array occupies more space[9]. In practices, the bit-array only approach is not far behind the distance-array approach since the downward search unlikely traverses too many nodes to discover the NNs (due to the distribution of the objects). According to our experimental study, the distance-array approach is 20% faster than the bit-array approach on average. As the improvement is not significant, the bit-array is still our recommended guidance information due to its conciseness and the ease of maintenance. However, if the response time is the most important factor in the system, then the distance-array is recommended.

4.5 Implementation detail

In order to save memory usage, only one graph structure is used to represent both the upward and downward graphs as their hierarchical structures are identical.

For bi-directional graphs. We can decide the existence of an edge $e(n_i, n_j)$ in upward or downward graph by the edge-node level of n_i and n_j. For instance, $e(n_5, n_1)$ in Figure 9(b) exists in the downward graph since n_5 is located at higher level than n_1.

For general cases. For each edge, we store a 2-bit structure where the first (second) bit indicates the existence of an edge in the upward (downward) graph. This overhead is negligible as compared to the graph structure size.

5 Extensions

5.1 Supporting Multiple Types of Objects

In this section, we study how to support multiple types of objects in kNN queries. This extension has useful applications but it was neglected in the past. An example query looks like: *show me the nearest Chinese restaurant or Japanese restaurant*. Surprisingly, this important extension is not well addressed in existing work [4]–[8], [10]–[12].

A simple solution is to build a unified index for all objects (ignoring their types) and then check the types of objects on-the-fly at the query time. However, the object cardinality reduces the effectiveness of the index. To make things worse, the target group of kNN objects may be far away from the query node, which may report many false fits and render the index ineffective. Besides the straightforward solution, some approaches (e.g., SWH [4], G-tree [9], ROAD [6], and SILC [10]) can build an individual index for each type of objects since these approaches separately index the road network and the objects. However, this approach needs to search multiple indices if the kNN query is interested in more than one single type of objects (e.g., Chinese restaurants or Japanese restaurants).

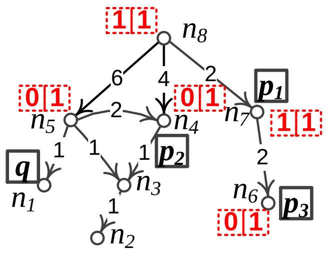


Fig. 13. Support of multiple type of objects

We find that GF supports multiple object types in a very convenient way. We simply employ m bit-arrays for m types of objects. Suppose we use two bits to index two types of objects, e.g., type-I: $\{p_1\}$ and type-II: $\{p_2, p_3\}$, in the running example. Figure 13 shows their bit-flag values in the downward graph. To construct this structure, we iteratively execute lines 2–11 of Algorithm 3 for each object type.

The size overhead of GF increases linearly with the number of object types m. To reduce the overhead, we can merge multiple types of objects into a super type and use one bit per node for it. Its effectiveness is justified by these observations: (1) the object-to-node ratio is very small in practice (e.g., the largest group of objects in our California dataset falls on 9.5% of the network nodes), (2) the distribution of different type of objects may be similar (e.g., the restaurants and shopping malls fall on similar distributions), and (3) some object types are likely searched together in kNN queries (e.g., hotels and restaurants). These observations suggest that we can group the object types by standard clustering techniques [23] based on their distribution similarity, where the number of clusters is subject to an index size constraint. More specifically, the similarity of two object types, t_i and t_j, can be defined as

$$sim(\mathcal{B}_{t_i}, \mathcal{B}_{t_j}) = \frac{one(\mathcal{B}_{t_i} \cap \mathcal{B}_{t_j})}{one(\mathcal{B}_{t_i} \cup \mathcal{B}_{t_j})}$$

where $one(\mathcal{B})$ returns the number of 1's in $\mathcal{B}$ and $sim(\cdot)$ falls into [0,1]. Two object types of high similarity are likely grouped into the same super type in the clustering process. In the experiments, we will show the effectiveness of this grouping approach.

5.2 Supporting Range Queries

The range queries are also popular in daily life applications. For instance, people are interested in finding restaurants within walking distances to their current location. To process the range queries using GF, we traverse the graph using

9. The overhead is still negligible. For instance, the distance-array size in US road network is just 45.66 MB.

the same strategy of Algorithm 4 until the weight of the first heap element is larger than the range threshold r. A minor optimization is that we ignore an relaxed edge if its weight is already larger than r.

6 Experimental Study

In this section, we experimentally compare our proposed solutions with existing kNN solutions, including INE [7], SWH [4], SWH+LM [4], SPIE [5], G-tree [9], ROAD [6], and SILC [10]. Some solutions are omitted from our experimental study because they were shown to be dominated by the above solutions, e.g., (i) INE [7], LBC [8] $\prec$ SWH [4], and (ii) Distance Index [11] $\prec$ ROAD [6], and (iii) VN^3 [12] $\prec$ SPIE [5]. We use the implementations from G-tree [9] (including G-tree and ROAD) and implement other methods by ourselves. We have verified the correctness of every implemented method and their relative performance is similar or even better than the result reported in G-tree [9] and ROAD [6]. All experiments are conducted on a 64-bit Ubuntu machine with Intel Core i7 3.2GHz CPU and 103GB RAM. To fairly measure the response time of different methods, we secure enough memory to store the entire index and execute kNN search in the main memory. In other words, the entire query process does not incur any disk access.

6.1 Datasets and parameter setting

TABLE 1
Statistics of the roadmaps

city	abbr.	#vertices	#edges	#objs	
Colorado	CO	435,666	1,057,066	-	synthetic
California	CA	1,890,815	4,657,742	-	synthetic
Eastern US	E-US	3,598,623	8,778,114	-	synthetic
Western US	W-US	6,262,104	15,248,146	-	synthetic
United States	US	23,947,347	58,333,344	-	synthetic
California	CAL	1,984,828	2,583,212	391,604	real
New York	NY	831,938	1,068,548	98,539	real
Los Angeles	LA	336,630	448,964	18,326	real

We conduct our experiments on 8 public available roadmap datasets [24]–[26] as shown in Table 1. We extracted the last three road maps (California/CAL, New York/NY, Los Angeles/LA) and their corresponding points-of-interest datasets from OpenStreetMap[10]. For every object, we map the object to the closest road segment by their latitude and longitude based on Euclidean distance. As shown in Section 2.1, we deal with the edge objects by replicating them to two end nodes of their located edge and keeping the distances from two end nodes to their actual locations. For the real roadmaps (CAL, NY and LA), we randomly pick objects according to an object-to-node ratio. For other roadmaps, we randomly generate the objects on the nodes according to an object-to-node ratio.

Table 2 shows the ranges of the investigated parameters, and their default values (in bold). In each experiment, we vary a single parameter, while setting the others to their default values. For each method, we report its average performance over 10,000 kNN queries.

10. http://www.openstreetmap.org

TABLE 2
Range of parameter values

Parameter	Values
Result size k	1, 5, **10**, 20, 50
Object-to-node ratio	0.05%, 0.1%, 0.25%, 0.5%, **1%**, 2.5%, 5%, 10%, 20%, 50%
Length of shortest paths	short, avg., above avg., long, **mixed**
#Object types per index (for real object experiments only)	**1**, 4, 8, 12, 16, 20
Range distance r (for range queries)	1, 2, **5**, 10, 20 (times NN distance)
Local rearrangement λ	0, 10, 15, **20**, 25, 30, 35

In this work, the overlay graph contraction order and the number of levels are based on the suggestion of [17] as the overhead of their overlay graph is the smallest among other competitors [20]. Note that our work are applicable to other overlay graph solutions (e.g., [20]). We omit the investigation of the overlay graph since we concentrate here on evaluating the proposed kNN query processing techniques.

6.2 Scalability experiments

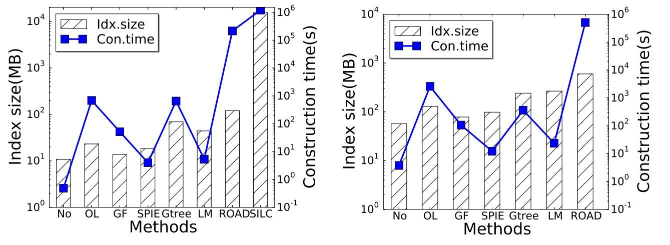
The effect of graph size. Table 3 shows the response time of all methods on these road networks by setting all parameters to their default values (e.g., object-to-node ratio=1% and $k = 10$). A promising finding is that our methods (OL and GF) scale very well, and it seems that they are almost insensitive to the road network sizes. This is because the overlay graph is flattened in general (i.e., a minority of important nodes have large fan-out due to the highway property.). The upward search remains efficient due to the flattened index structure and the fact that the downward search of GF is well guided by the bit-flag array. Among all competitors, SPIE is the only method which is not sensitive to the graph sizes; nevertheless, it is still one order of magnitude slower than our proposed methods. Besides, SPIE does not scale well with other factors (e.g., k) as its search strategy is based on a local shortest tree. To make things worse, SPIE requires to build one graph index for one specific type of objects which limits its applicability in real world applications. G-tree does not scale well with the graph sizes. For instance, the response time of G-tree in the entire US roadmap (US) is around 5.65 times slower than in the western portion of US roadmap (W-US), although the number of edges is just 3.82 times larger. SILC is the closest method to ours in terms of response time. However, since it occupies $O(N^{1.5})$ space, the index sizes for larger road networks (CA, E-US, W-US, US) exceed the memory size of a commodity machine. Besides SPIE, G-tree, and SILC, the response times of other competitors are two order of magnitude slower than our proposed methods. Some result are omitted due to their long construction time and memory demand, e.g., the result of ROAD in E-US, W-US, and US.

For clarity, we also demonstrate the construction time and index size[11] of all discussed methods in Figure 14 on two datasets, LA and CA. Obviously, our proposed methods, OL and GF, have a reasonable index size and construction time. Specifically, the index size of GF is just 26.7% and

11. The index size indicates the size of both network index and object index.

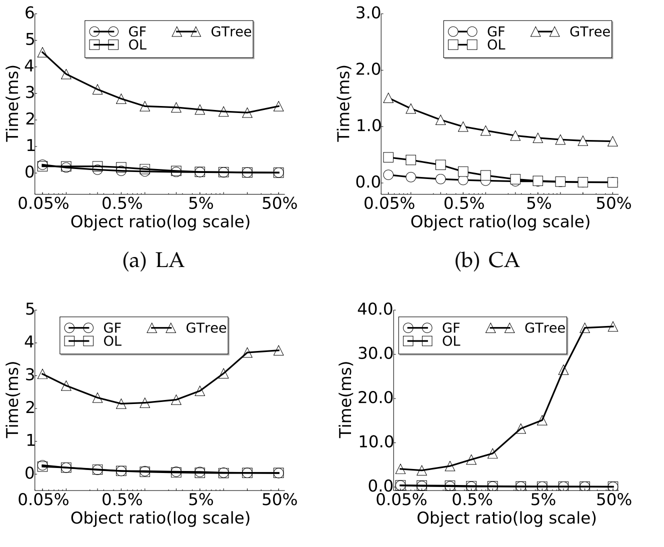
TABLE 3
The effect of graph size

	City	Method								
		no-network indexing		**network indexing**						
		INE	SWH	OL	GF	G-tree	SPIE	SWH-LM	ROAD	SILC
Response time(ms)	CO	37.38	69.06	0.20	**0.070**	0.79	1.97	89.73	384.09	0.63
	CA	60.87	68.49	0.25	**0.081**	1.31	2.08	95.73	1082.49	n/a
	E-US	74.98	73.96	0.24	**0.067**	3.62	2.12	94.67	n/a	n/a
	W-US	158.14	79.12	0.27	**0.055**	3.24	1.89	120.37	n/a	n/a
	US	412.35	134.80	0.27	**0.078**	18.31	2.05	236.49	n/a	n/a
	CAL	115.85	667.82	0.17	**0.074**	6.52	4.09	1912.08	n/a	n/a
	NY	54.56	34.35	0.094	**0.057**	2.29	1.88	109.73	623.04	n/a
	LA	41.53	11.33	**0.014**	0.06	2.74	2.23	125.04	227.15	0.142



(a) LA (b) CA

Fig. 14. Construction time and index size of methods



(a) LA (b) CA (c) NY (d) CAL

Fig. 15. The effect of object-to-node ratio

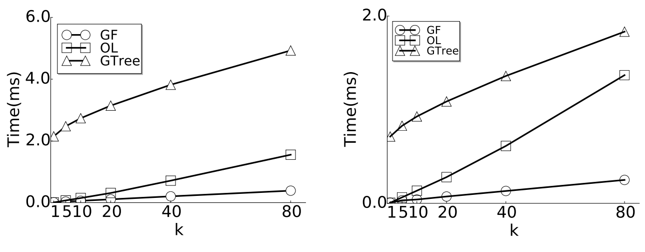
37% larger than the size of no-network indexing methods on LA and CA, respectively. Among these methods, SPIE is the only method that offers competitive index size to OL and GF which also provides very fast construction time. However, SPIE does not scale with respect to other scalability factors (e.g., object cardinality) since every object is required to keep a local search tree in SPIE. Other methods like G-tree and SWH-LM are only 4-6 times larger than GF. ROAD and SILC are definitely infeasible for large road networks as their sizes are one to two orders of magnitude larger than GF. Also, the construction time of ROAD and SILC is very slow since these methods require many pairwise shortest path computations during index construction. For instance, ROAD and SILC requires 59.6 and 336.0 hours, respectively, to construct the index of LA while the construction of GF only takes 24.5 seconds. In summary, our proposed methods are superior to all competitors with respect to all three performance factors (including response time, construction time, and index size). More importantly, we are the first work that process kNN queries within 0.1ms in 10-million-node networks on a commodity machine. This translates to a query throughput of 10,000 queries per commodity machine per second.

In the remaining experiments, we omit less competitive methods (like INE, SWH, SWH-LM, SPIE, ROAD, and SILC) as they are far behind our proposed methods or their applicability is limited. Moreover, we only report the results on two datasets, LA (using real object datasets) and CA (using synthetic object datasets). The results on other datasets exhibit similar trends.

The effect of object-to-node ratio. Figure 15 shows the average query time of k-NN queries on the LA, CA, NY, and CAL datasets varying on object-to-node ratio. For the real object roadmaps (i.e., LA, NY, and CAL) experiments, we sample the testing object sets from the entire dataset according to different object-to-node ratios. If the object-to-node ratio is higher than the amount of an object set (e.g., there are only 5.4% nodes containing object(s) in the LA dataset), we randomly sample nodes from the roadmap as object nodes. In general, the search executes faster when the object-to-node ratio increases since the kNN result becomes closer to the query location. Our method scales better on real objects since both OL and GF are more effective on skewed distributions (i.e., OL may have fewer object shortcuts and GF may have fewer nodes having *true* bit-flag value). G-tree is more sensitive to the object-to-node ratio due to its query processing strategy. G-tree partitions the road network into a set of subgraphs and the search space depends on the *number* of subgraphs that have object(s) inside. G-tree necessarily traverses more subgraphs if the object-to-node ratio increases.

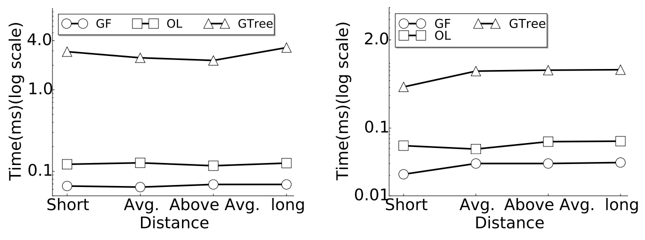
The effect of k. Figure 16 shows the average query time of k-NN queries on the LA and CA datasets varying on k. The response time of every method grows linearly with the value of k. Again, our proposed methods are less sensitive in LA than CA due to the mentioned effect of object-to-node ratios. GF is superior to G-tree as it is 12.7 and 7.3 times faster at $k = 80$ on LA and CA, respectively.

The effect of query distance. To evaluate the effect of



(a) LA (b) CA

Fig. 16. The effect of k



(a) LA (b) CA

Fig. 17. The effect of query distance

different query distances, we first calculate the diameter of the road network, denoted as d, and then generate 4 query sets (containing 10,000 queries for each) based on the average distance to their kNN object using d as a metric reference. Specifically, the maximum object distances in these 4 query sets are $\frac{d}{8}$, $\frac{d}{4}$, $\frac{d}{2}$, and $\frac{3d}{4}$, respectively, which are denoted as *short*, *avg.*, *above avg.*, and *long* in Figure 17. The methods generally become more costly when the query-object distances increase. OL and GF are 20-47 times faster than G-tree in different query sets.

Object distributions. Next, we report the average response

TABLE 4
The effect of object distributions (LA)

object type (object ratio)	**bank** (0.12%)	**cafe** (0.2%)	**restaurant** (0.48%)	**school** (1.3%)	**place of worship** (2%)
SPIE	13.70ms	10.93ms	5.63ms	1.61ms	1.87ms
G-tree	3.19ms	3.12ms	2.86ms	2.45ms	2.33ms
OL	0.134ms	0.142ms	0.122ms	0.078ms	0.064ms
GF	0.081ms	0.077ms	0.057ms	0.035ms	0.034ms

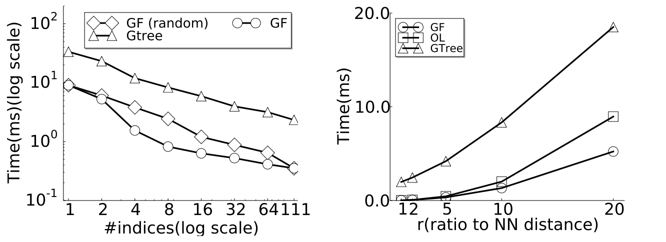
time of the methods in different object distributions, on the LA dataset. Our methods (both OL and GF) outperform G-tree and SPIE by at least an order of magnitude since real objects follow skewed distributions.

Summary. According to our experimental results, it is clear that GF outperforms all competitors in terms of response time and index size. More specifically, GF computes kNN queries by one order of magnitude faster than existing solutions. The index size of GF is negligible which is just 33.5% to 50.1% larger than the size of no-network indexing methods in all evaluated datasets under the default settings. The construction time of GF is also fast: for instance, the index of the US roadmap can be constructed in 12.2 minutes (while the construction of OL and G-tree takes 9.3 and 3.39 hours, respectively). The runner-up method OL is not far behind GF in terms of response time; however, its construction time is one order of magnitude slower than GF. Thus, we recommend GF as the best method for kNN search.

6.3 Extension experiments

The effect of multiple types. We also evaluate our grouping strategy for multiple type of objects (cf. Section 5.1). We note that the multiple type extension is also applicable to G-tree which only requires to keep some additional information in every subgraph without affecting the road network index. However, we omit to adopt this change to OL since it requires to rearrange many contraction orders for handling an extra type of objects. Thereby, we only evaluate GF and G-tree in this experiment.

We group all 111 object types from the object dataset of LA. To evaluate our grouping strategy, these 111 types are grouped into 64, 32, 16, 8, 4, 2 and 1 indices by a standard clustering algorithm based on the similarity equation discussed in Section 5.1. In this experiment, each query searches a random type of objects. Figure 18(a) shows that both GF and G-tree scale well with the number of indices. It demonstrates the effectiveness of our grouping strategy in Section 5.1. In addition, we also report the performance of a method, GF (random), that groups object types arbitrarily. Both GF and GF (random) outperform G-tree by a visible margin.



(a) Multiple types (LA) (b) Range queries (LA)

Fig. 18. Supplementary experiments on real objects

Range queries. We also evaluate the processing performance of range queries by varying the range threshold r. In this work, we simply multiply r by the first NN distance. When r is set to 20, the range queries return several hundred results on average. Figure 18(b) compares GF with two closest competitors, G-tree and OL, in LA road network. GF is clearly superior to the other competitors where it is at least 3.5 times and 1.16 times faster than G-tree and OL, respectively.

6.4 The effect of GF optimizations

We proceed to investigate the effectiveness of our optimization techniques for GF proposed in Section 4.3 and 4.4.

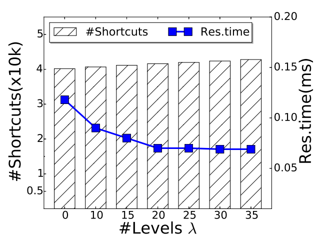
Contraction early termination. Table 5 shows the effect of the contraction early termination. As the query performance is secured by the termination strategy discussed in Section 4.3 so that we only demonstrate the gain of the index size and construction time. Table 5 compares two methods, where GF-Full searches in the fully contracted overlay graph and GF searches in the heterogenous graph. GF saves 7.2% index size and 50% construction time on average among

TABLE 5
The effect of early termination

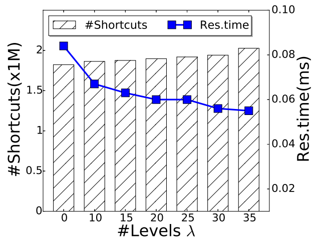
City	Index size (MB)			Construction time (s)		
	GF-Full	GF	gain	GF-Full	GF	gain
CO	17.46	16.34	6.4%	10.83	8.15	24.7%
CA	78.32	72.71	7.2%	90.51	51.60	42.99%
E-US	148.72	137.07	7.8%	177.48	89.48	49.58%
W-US	256.27	238.4	7.0%	273.54	164.59	39.83%
US	996.11	914.19	8.2%	1667.15	734.77	55.93%
CAL	75.52	72.03	4.61%	239.56	132.10	44.86%
NY	31.87	30.39	4.64%	79.10	44.07	44.29%
LA	13.66	12.95	5.20%	42.03	20.35	51.57%

all evaluated road networks which matches the analysis in Section 4.3.

Local contraction rearrangement. Next we demonstrate the



(a) LA

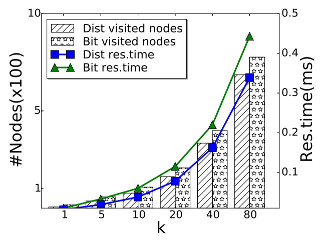


(b) CA

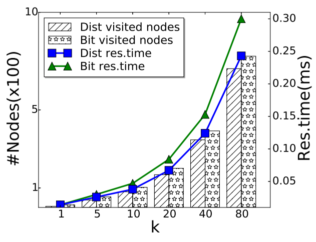
Fig. 19. Local rearrangement

effect of local contraction rearrangement subject to different level constraints λ by 19. When λ is set to 0, we do not allow any contraction rearrangement. When we slightly increase λ (e.g., $\lambda = 10$), the response time of the queries decreases since the bit-array becomes more effective (i.e., less *true* values). However, the improvement becomes less obvious when λ rises above 20 since more shortcuts are added into the graph due to the local rearrangement. We set λ to 20 as our default setting as it achieves a good trade-off.

The effect of guidance information. Figure 20 shows the



(a) LA



(b) CA

Fig. 20. The effect of guidance information

effect of adopting different guidance information, including the bit-array and the distance-array. The response time of GF with the distance-array is around 20% faster than the bit-array on both road networks. Moreover, the distance-array reduces the number of visited nodes by 20% and 10% when $k = 10$ and $k = 80$, respectively, since it postpones the relaxation of unpromising edges (cf. our discussion in Section 4.4). This turns out to reduce the number of elements kept in the heap. However, we still recommend the bit-array as the default guidance information due to its conciseness and the ease of maintenance.

7 Related Work

No-network indexing methods for kNN. The methods in this category only access the given data (the road network G and POI index) and employ priority queues (i.e., heaps) for processing queries. All of them can fit in main memory.

Incremental Euclidean Restriction (IER) [7] utilizes the property that the Euclidean distance between two nodes lower-bounds their network distance. IER incrementally retrieves candidate POIs p_i in ascending order of their Euclidean distances $d_E(q, p_i)$ from q, and computes their network distances $d_N(q, p_i)$ from q (e.g., by calling A* algorithm). It terminates when the next Euclidean NN distance exceeds the k-th nearest network distance found so far. However, IER executes the network distance computation multiple times, which may visit some nodes in the network repeatedly. We can boost the performance of IER by replacing the A* algorithm with modern shortest path algorithms [14], [17]–[20]. However, it still suffers from multiple network distance computations and visiting some network nodes repeatedly.

Incremental Network Expansion (INE) [7] employs a heap H to examine network nodes in ascending order of their network distance $d_N(q, p_i)$ from q. It guarantees that each network node is visited at most once. However, since it does not exploit the Euclidean lower-bound, it may visit some irrelevant nodes that cannot lead to the kNN results.

Lower Bound Constraint (LBC) [8] integrates IER with A* search such that each network node is visited at most once. Specifically, for each candidate p_i, LBC maintains a heap H_{p_i} to store the lower-bound distances from q via node n to p_i. In each iteration, LBC identifies the node n' with the smallest bound of all heaps, explores the neighbors of n', and updates all heaps H_{p_i}. Although LBC requires fewer accesses to the road network, it incurs higher overhead in maintaining multiple heaps H_{p_i} for candidates. Single Wavefront Heuristics (SWH) [4] employs a single heap to manage the lower bound distances above, while achieving the same road network access cost as in LBC. Since SWH has smaller overhead on maintaining a heap, it performs better than LBC in terms of CPU time.

Network indexing methods for kNN. The methods in this category require significant storage space for storing precomputed information (i.e., the index), which can be utilized to accelerate kNN search significantly. Due to large index size, most of these indexes can only be stored in hard disk instead of main memory, thereby outweighing the benefit offered by the index.

For example, VN^3 [12] requires precomputing the network Voronoi diagram with respect to a POI dataset P. SILC [10] requires $O(N^{1.5})$ storage space, an amount that is super-linear in the number of network nodes. Among existing methods, ROAD [6] and SPIE [5] outperform Distance Index [11] and VN^3 [12], respectively. G-tree [9] is the latest network indexing method for kNN queries and its query response time is shown to outperform that of ROAD and SILC. First, it partitions the road network into sub-networks, and then maintains a shortest path distance matrix for every pair of border nodes within each sub-network. At query time, this information can be effectively utilized to accelerate kNN search. Observe that these shortest path distance matrices occupy much more memory space and incur higher response time when compared to our proposed solutions.

8 Conclusion

To the best of our knowledge, this is the first work that offers very low latency (0.1ms) per kNN query on 10-million-node networks on a commodity machine. This translates to a query throughput of 10,000 queries per second per commodity machine. Our experimental studies on large scale road networks show that our solutions are 1-3 orders of magnitudes faster than existing methods while our indexes are compact and can fit into main memory. In the future, we plan to further enhance the performance of kNN search with keywords on objects.

References

[1] L. A. Barroso, J. Dean, and U. Hölzle, "Web search for a planet: The google cluster architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003.

[2] R. A. Baeza-Yates, C. Castillo, F. Junqueira, V. Plachouras, and F. Silvestri, "Challenges on distributed web retrieval," in *ICDE*, 2007, pp. 6–20.

[3] R. A. Baeza-Yates, "Towards a distributed search engine," in *CIAC*, 2010, pp. 1–5.

[4] S. Nutanong and H. Samet, "Memory-efficient algorithms for spatial network queries," in *ICDE*, 2013, pp. 649–660.

[5] H. Hu, D. L. Lee, and J. Xu, "Fast nearest neighbor search on road networks," in *EDBT*, 2006, pp. 186–203.

[6] K. C. K. Lee, W.-C. Lee, B. Zheng, and Y. Tian, "Road: A new spatial object search framework for road networks," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 3, pp. 547–560, 2012.

[7] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query processing in spatial network databases," in *VLDB*, 2003, pp. 802–813.

[8] K. Deng, X. Zhou, H. T. Shen, S. W. Sadiq, and X. Li, "Instance optimal query processing in spatial networks," *VLDB J.*, vol. 18, no. 3, pp. 675–693, 2009.

[9] R. Zhong, G. Li, K.-L. Tan, and L. Zhou, "G-tree: an efficient index for knn search on road networks," in *CIKM*, 2013, pp. 39–48.

[10] H. Samet, J. Sankaranarayanan, and H. Alborzi, "Scalable network distance browsing in spatial databases," in *SIGMOD Conference*, 2008, pp. 43–54.

[11] H. Hu, D. L. Lee, and V. C. S. Lee, "Distance indexing on road networks," in *VLDB*, 2006, pp. 894–905.

[12] M. R. Kolahdouzan and C. Shahabi, "Voronoi-based k nearest neighbor search for spatial network databases," in *VLDB*, 2004, pp. 840–851.

[13] A. V. Goldberg and C. Harrelson, "Computing the shortest path: *A* search meets graph theory," in *SODA*, 2005, pp. 156–165.

[14] J. M. Kleinberg, A. Slivkins, and T. Wexler, "Triangulation and embedding using small sets of beacons," *J. ACM*, vol. 56, no. 6, 2009.

[15] I. Abraham, A. Fiat, A. V. Goldberg, and R. F. F. Werneck, "Highway dimension, shortest paths, and provably efficient algorithms," in *SODA*, 2010, pp. 782–793.

[16] R. Mertens, T. Steffens, and J. Stachowiak, "Path searching with transit nodes in fast changing telecommunications networks," in *GI Jahrestagung (1)*, 2008, pp. 158–163.

[17] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, "Contraction hierarchies: Faster and simpler hierarchical routing in road networks," in *WEA*, 2008, pp. 319–333.

[18] R. J. Gutman, "Reach-based routing: A new approach to shortest path algorithms optimized for road networks," in *ALENEX/ANALC*, 2004, pp. 100–111.

[19] R. Bauer and D. Delling, "Sharc: Fast and robust unidirectional routing," in *ALENEX*, 2008, pp. 13–26.

[20] A. D. Zhu, H. Ma, X. Xiao, S. Luo, Y. Tang, and S. Zhou, "Shortest path and distance queries on road networks: towards bridging theory and practice," in *SIGMOD Conference*, 2013, pp. 857–868.

[21] R. Bauer, D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner, "Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm," *ACM Journal of Experimental Algorithmics*, vol. 15, 2010.

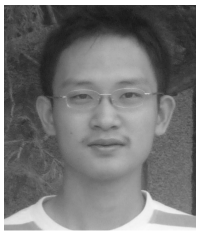
[22] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes, "In transit to constant time shortest-path queries in road networks," in *ALENEX*, 2007.

[23] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, no. 8, pp. 888–905, 2000.

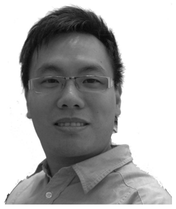
[24] http://www.dis.uniroma1.it/challenge9/download.shtml.

[25] http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm.

[26] http://www.idi.ntnu.no/~joao/publications/EDBT2012/.



Bailong Liao is a Master student at the Department of Computer and Information Science, University of Macau, under the supervision of Prof. Zhiguo Gong and Dr. Leong Hou U. His current research focuses on spatial database and data mining.



Leong Hou U is now an Assistant Professor at the University of Macau. His research interest includes spatial and spatio-temporal databases, advanced query processing, web data management, information retrieval, data mining and optimization problems.

Man Lung Yiu is now an assistant professor in the Department of Computing, Hong Kong Polytechnic University. Prior to his current post, he worked at Aalborg University for three years starting in the Fall of 2006. His research focuses on the management of complex data, in particular query processing topics on spatiotemporal data and multidimensional data.

Zhiguo Gong is currently an Associate Professor in the Department of Computer and Information Science, University of Macau, Macau, China. His research interests include databases, web information retrieval, and web mining.