k-Nearest Neighbors on Road Networks: A Journey in Experimentation and In-Memory Implementation

Tenindra Abeywickrama, Muhammad Aamir Cheema, David Taniar
Faculty of Information Technology, Monash University, Australia
{tenindra.abeywickrama,aamir.cheema,david.taniar}@monash.edu

ABSTRACT

A k nearest neighbor (kNN) query on road networks retrieves the k closest points of interest (POIs) by their network distances from a given location. Today, in the era of ubiquitous mobile computing, this is a highly pertinent query. While Euclidean distance has been used as a heuristic to search for the closest POIs by their road network distance, its efficacy has not been thoroughly investigated. The most recent methods have shown significant improvement in query performance. Earlier studies, which proposed disk-based indexes, were compared to the current state-of-the-art in main memory. However, recent studies have shown that main memory comparisons can be challenging and require careful adaptation. This paper presents an extensive experimental investigation in main memory to settle these and several other issues. We use efficient and fair memory-resident implementations of each method to reproduce past experiments and conduct additional comparisons for several overlooked evaluations. Notably we revisit a previously discarded technique (IER) showing that, through a simple improvement, it is often the best performing technique.

1. INTRODUCTION

Cisco reports that more than half a billion mobile devices were activated in 2013 alone, and 77% of those devices were smartphones. Due to the surge in adoption of smartphones and other GPS-enabled devices, and cheap wireless network bandwidth, map-based services have become ubiquitous. For instance, the GlobalWebIndex reported that Google Maps was the most used smartphone app in 2013 with 54% of smartphone users having used it [1]. Finding nearby facilities (e.g., restaurants, ATMs) are among the most popular queries issued on maps. Due to their popularity and importance, k nearest neighbor (kNN) queries, which find the k closest points of interest (objects) to a given query location, have been extensively studied in the past.

While related to the shortest path problem in many ways, the kNN problem on road networks introduces new challenges. Since the total number of objects is usually much larger than k it is not efficient to compute the shortest paths (or network distances) to all objects to determine which are kNNs. The challenge is to not only ignore the objects that cannot be kNNs but also the road network vertices that are not associated with objects. Recently, there has been a large body of work to answer kNN queries on road networks. Some of the most notable algorithms include *Incremental Network Expansion* (INE) [18], *Incremental Euclidean Restriction* (IER) [18], *Distance Browsing* [20], *Route Overlay and Association Directory* (ROAD) [16, 17], and *G-tree* [24, 25]. In this paper, we conduct a thorough experimental evaluation of these algorithms.

1.1 Motivation

1. Neglected Competitor. IER [18] was among the first kNN algorithms on road networks. It has often been the worst performing method and as a result is no longer included in comparisons. The basic idea of IER is to compute shortest path distances using Dijkstra's algorithm to the closest objects in terms of Euclidean distance. Although many significantly faster shortest path algorithms have been proposed in recent years, surprisingly, IER has never been compared against other kNN methods using any algorithm other than Dijkstra. To ascertain the true performance of IER it must be integrated with state-of-the-art shortest path algorithms.

2. Discrepancies in Existing Results. We note several discrepancies in the experimental results reported in some of the most notable papers on this topic. ROAD is seen to perform significantly worse than Distance Browsing and INE in [24]. But according to [16], ROAD is experimentally superior to both Distance Browsing and INE. The results in both [16] and [24] show Distance Browsing has worse performance than INE. In contrast, Distance Browsing is shown to be more efficient than INE in [20]. These contradictions identify the need for reproducibility.

3. Implementation Does Matter. Similar to a recent study [22], we observe that simple implementation choices can significantly affect algorithm performance. For example, G-tree utilizes *distance matrices* that can be implemented using either hash-tables or arrays and, on the surface, both seem reasonable choices. However the array implementation in fact performs more than an order of magnitude faster than the hash-table implementation. We show that this is due to data locality in G-tree's index and its impact on cache performance. In short, seemingly innocuous choices can drastically change experimental outcomes. We also believe discrepancies reported above may well be due to different choices made by the implementers. Thus it is critical to provide a fair comparison of existing kNN algorithms using careful in-memory implementations.

4. Overlooked Evaluation Measures/Settings. All methods studied in this paper decouple the road network index from that of the set of objects, i.e. one index is created for the road network and another to store the set of objects. Although existing studies evaluate the road network indexes, no study evaluates the behaviour of each individual object index. The construction time and storage cost

Proceedings of the VLDB Endowment, Vol. 9, No. 6

for these *object indexes* may be critical information for developers when choosing methods, especially for object sets that change regularly. Additionally kNN queries have not been investigated for travel time graphs (only travel distance), which is also a common scenario in practice. Finally the more recent techniques (G-tree and ROAD) did not include comparisons for real-world POIs.

1.2 Contributions

Below we summarize the contributions we make in this paper.

1. Revived IER: We investigate IER with several efficient shortest path techniques for the first time (see Section 5). We show that the performance of IER is significantly improved when better shortest path algorithms are used. This occurs to the point that IER is the best performing method in most settings, including travel time road networks where Euclidean distance is a less effective lower bound.

2. Highly Optimised Algorithms Open-Sourced: We present efficient implementations of five of the most notable methods (IER, INE, Distance Browsing, ROAD and G-tree). Firstly we have carefully implemented each method for efficient performance in main memory as described in Section 6. Secondly we thoroughly checked each algorithm and made various improvements applicable in any setting, as documented in Appendix A and [6]. The source code and scripts to run experiments have been released as open-source [2], making our best effort to ensure it is modular and re-usable.

3. Reproducibility Study: With efficient implementations of each algorithm, we repeat many experiments from past studies on many of the same datasets in Section 7. Our results provide a deeper understanding of the state-of-the-art with new insights into the weaknesses and strengths of each technique. We also show that there is room to improve kNN search heuristics by demonstrating that G-tree can be made more efficient by using Euclidean distances.

4. Extended Experiments and Analysis: Our comprehensive experimental study in Section 7 extends beyond past studies by: 1) comparing object indexes for the first time; 2) revealing new trends by comparing G-tree with another advanced method (ROAD) on larger datasets for the first time; 3) evaluating all methods (including ROAD and G-tree) on real-world POIs; and 4) evaluating applicable methods on travel time road networks.

5. Guidance on Main-Memory Implementations: In Section 6 we also demonstrate how simple choices can severely impact algorithm performance. We share an in-depth case study to give insights into the relationship between algorithms and in-memory performance with respect to data locality and cache efficiency. Additionally we highlight the main choices involved and illustrate them through examples and experimental results, to provide hints to future implementers. Significantly, these insights are potentially applicable to any problem, not just those we study here.

2. BACKGROUND

2.1 Problem Definition

We represent a road network as a connected undirected graph $G = (V, E)$ where V is the set of vertices and E is the set of edges. For two adjacent vertices $u, v \in V$, we define the edge between them as $e(u, v)$, with weight $w(u, v)$ representing any positive measure such as distance or travel time. We define the shortest path distance, hereafter network distance, between any two vertices $u, v \in V$ as $d(u, v)$, the minimum sum of weights connecting u and v. For conceptual simplicity, similar to the existing studies [20, 24], we assume that each object (i.e., POI) and query is located on some vertex in V. Given a query vertex q and a set of object vertices O, a kNN query retrieves the k closest objects in O based on their network distances from q.

2.2 Scope

We separate existing kNN techniques into two broad categories based on the indexing they use: 1) blended indexing; and 2) decoupled indexing. Techniques that use blended indexing [9, 13, 15] create a single index to store the objects as well the road network. For example, VN^3 [15] is a notable technique that uses a network Voronoi diagram based on the set of objects to partition the network. In contrast, decoupled indexing techniques [16, 18, 20, 24] use two separate indexes for the object set and road network, which is more practical and has several advantages as explained below.

Firstly, a real-world kNN query may be applied to one of many object sets, e.g., return the k closest restaurants or locate the nearest parking space. Blended indexing must repeatedly index the road network for each type of object, entailing huge space and preprocessing time overheads. But decoupled indexing requires only one road network index regardless of the number of object sets, resulting in lower storage and pre-processing cost. Secondly, if there is any change in an object set, blended indexing must update the whole index and reprocess the entire road network, whereas decoupled techniques need only update the object index. For example, the network-based Voronoi diagram must be updated resulting in expensive re-computations [15]. Conversely, in decoupled indexing, the object indexes (e.g., R-tree) are typically much cheaper to update. The problem is more serious for object sets that change often, e.g., if the objects are the nearest *available* parking spaces.

Due to these advantages, all recent kNN techniques use decoupled indexing. In this paper, we focus on the most notable kNN algorithms that employ decoupled indexing. These algorithms either employ an expansion-based method or a heuristic best-first search (BFS). The expansion-based methods encounter kNNs in network distance order. Heuristic BFS methods instead employ heuristics to evaluate the most promising kNN candidates, not necessarily in network distance order, potentially terminating sooner. We study the five most notable methods which include two expansion-based methods, INE [18] and ROAD [16], and three heuristic BFS methods, IER [18], Distance Browsing (DisBrw) [20] and G-tree [24].

Given the rapid growth in smartphones and the corresponding widespread use of map-based services, applications must employ fast in-memory query processing to meet the high query workload. In-memory processing has become viable due to the increases in main-memory capacities and its affordability. Thus, we limit our study to in-memory query processing. However, we remark that disk-based settings are also important but are beyond the scope of this paper mainly due to the space limitation.

3. METHODS

We now describe the main ideas behind each method evaluated by our study. Some methods propose a road network index and a kNN query algorithm to use it. In some cases, such as G-tree, we refer to both the index and kNN algorithm by the same name.

3.1 Incremental Network Expansion

Incremental Network Expansion (INE) [18] is a method derived from Dijkstra's algorithm. As in Dijkstra, INE maintains a priority queue of the vertices seen so far (initialised with the query vertex q). The search is expanded to the nearest of these vertices v. If $v \in O$ then it is added to the result set as one of the kNNs and if v is the kth object then the search is terminated. Otherwise the edges of v are used to relax the distances to its neighbors and the expansion continues. As in Dijkstra's algorithm, relaxation involves

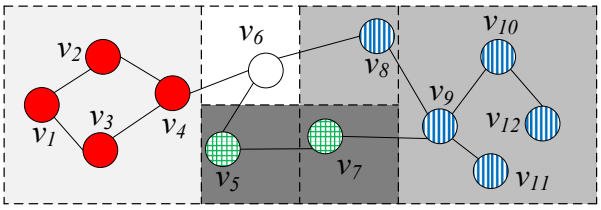


Figure 1: SILC: Coloring Scheme and Quadtree for v_6

updating the minimum network distances to the neighbors of v using the network distance through v. The disadvantage of INE is that it visits all nodes closer to q than the kth object, which may be considerable if this object is far from q.

3.2 Incremental Euclidean Restriction

Incremental Euclidean Restriction (IER) [18] uses Euclidean distance as a heuristic to retrieve candidates from O, as it is a lower bound on network distance for road networks with travel distance edges. Firstly, IER retrieves the Euclidean kNNs, e.g., using an R-tree [19]. It then computes the network distance to each of these k objects and sorts them in this order. This set becomes the candidate kNNs and the network distance to the furthest candidate (denoted as D_k) is an upper bound on the distance to the true kth nearest neighbor. Now, IER retrieves the next nearest Euclidean neighbor p. If the Euclidean distance to p is $d_E(q,p)$ and $d_E(q,p) \geq D_k$, then p cannot be a better candidate by network distance than any current candidate. Moreover, since it is the *nearest* Euclidean neighbor, the search can be terminated. However, if $d_E(q,p) < D_k$ then p may be a better candidate. In this case, IER computes the network distance $d(q,p)$. If $d(q,p) < D_K$, p is inserted into the candidate set (removing the furthest candidate and updating D_k). This continues until the search is terminated or there are no more Euclidean NNs.

3.3 Distance Browsing

Distance Browsing (DisBrw) [20] uses the Spatially Induced Linkage Cognizance (SILC) index proposed in [21] to answer kNN queries. [21] proposed an incremental kNN algorithm, which DisBrw improves upon by making fewer priority queue insertions.

SILC Index. We first introduce the SILC index used by DisBrw. For a vertex $s \in V$, SILC pre-computes the shortest paths from s to all other vertices. SILC assigns each adjacent vertex of s a unique color. Then, each vertex $u \in V$ is assigned the same color as the adjacent vertex v that is passed through in the shortest path from s to u. Figure 1 shows the coloring of the vertices for the vertex s=v_6 where each adjacent vertex of v_6 is assigned a unique color and the other vertices are colored accordingly. For example, the vertices v_9 to v_{12} have the same color as v_8 (blue vertical stripes) because the shortest path from v_6 to each of these vertices passes through v_8 (for this example assume unit edge weights).

Observe that the vertices close to each other have the same color resulting in several contiguous regions of the same color. These regions are indexed by a region quadtree [19] to reduce the storage space. The color of a vertex can be determined by locating the region in the quadtree that contains it. SILC applies the coloring scheme and creates a quadtree for each vertex of the road network. This requires $O(|V|^{1.5})$ space in total and, due to the all-pairs shortest path computation, $O(|V|^2 \log |V|)$ pre-processing time.

To compute the shortest path from s to t, SILC uses the quadtree of s to identify the color of t. The color of t determines the first vertex v on the shortest path from s to t. To determine the next vertex on the shortest path, this procedure is repeated on the quadtree

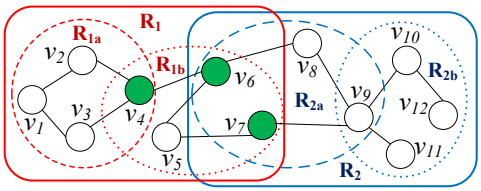


Figure 2: ROAD

of v. For example, in Figure 1, the first vertex on the shortest path from v_6 to v_{12} is v_8 because v_{12} has the same color as v_8. The color of v_{12} is found by locating the quadtree block containing v_{12}. The shortest path can be computed in $O(m \log |V|)$ where m is the number of edges on the shortest path [20].

kNN Algorithm. To enable kNN search, DisBrw stores additional information in each quadtree. For each vertex v contained in a quadtree block b, it computes the ratio of the Euclidean and network distances between the quadtree owner s and v. It then stores the minimum and maximum ratios, λ^- and λ^+ respectively, with b. Now, given any vertex t, DisBrw computes a *distance interval* $[\delta^-, \delta^+]$ by multiplying the Euclidean distance from s to t by the λ^- and λ^+ values of the block containing t. This interval defines a lower and upper bound on the network distance from s to t and can be used to prune objects that cannot be kNNs. The interval is *refined* by obtaining the next vertex u in the shortest path from s to t (as described earlier), computing an interval for u to t, and then adding the known distance from s to u to the new interval. By refining the interval, it eventually converges to the network distance.

DisBrw used an *Object Hierarchy* in [20] to avoid computing distance intervals for all objects. The basic idea was to compute distance intervals for regions containing objects, then visit the most promising regions (and recursively sub-regions) first. We found this method did not use the SILC index to its full potential. Instead we retrieve Euclidean NNs as candidate objects for which intervals are then computed. Otherwise, the DisBrw kNN algorithm proceeds exactly as in [20]. We refer the reader to [6] for full details and experimental comparisons with the original method.

3.4 Route Overlay & Association Directory

The search space of INE can be considerably large depending on the distance to the kth object. Route Overlay and Association Directory (ROAD) [16, 17] attempts to remedy this by bypassing regions that do not contain objects by using *search space pruning*.

An *Rnet* is a partition of the road network G=(V, E), with every edge in E belonging to at least one Rnet. Thus, an Rnet R represents a set of edges $E_R \subseteq E$. V_R is the set of vertices that are associated with edges in E_R. To create Rnets, ROAD partitions the road network G into $f \geq 2$ Rnets, recursively partitioning resulting Rnets until a hierarchy of $l > 1$ levels is formed (with G being the root at level 0). Figure 2 shows Rnets (for l=2) for the graph in our running example. The enclosing boxes and ovals represent the set V_R of each Rnet. Specifically, R_1=$\{v_1, \cdots, v_7\}$ and R_2=$\{v_6, \cdots, v_{12}\}$ are the child Rnets of the root G. Each of R_1 and R_2 are further divided into Rnets, e.g., R_1 is divided into R_{1a}=$\{v_1, v_2, v_3, v_4\}$ and R_{1b}=$\{v_4, v_5, v_6, v_7\}$.

For an Rnet R, a vertex $b \in V_R$ with an adjacent edge $e(b,v) \notin E_R$ is defined as a *border* of R. For instance, v_4 is a border of R_{1b} but v_5 is not. These borders form the set $B_R \subseteq V_R$, e.g., the border set of R_{1b} consists of v_4, v_6 and v_7. ROAD computes the network distance between every pair of borders $b_i, b_j \in B_R$ in each Rnet and stores each as the *shortcut* $S(b_i, b_j)$. Now any shortest path between two vertices $s, t \notin V_R$ involving a vertex $u \in V_R$ must enter R through a border $b \in B_R$ and leave through a border $b' \in B_R$.

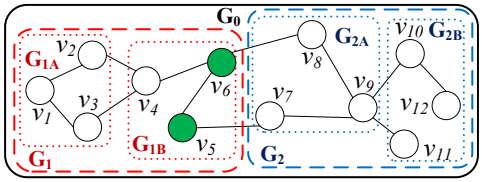


Figure 3: G-tree

So if a search reaches a border $b \in B_R$ the shortcuts associated with b, $S(b, b') \forall\, b' \in B_R$, can be traversed to bypass the Rnet R while preserving network distances. For example, in Figure 2, the borders of R_{1b} are v_4, v_6 and v_7 (the colored vertices) and ROAD precomputes the shortcuts between all these borders. Suppose the query vertex is v_1 and the search has reached the vertex v_4. If it is known that R_{1b} does not contain any object, the algorithm can bypass R_{1b} by quickly expanding the search to other borders of R_{1b} without the need to access any non-border vertex of R_{1b}. E.g., using the shortcut between v_4 and v_7, the algorithm can compute the distance between v_1 to v_7 without exploring any vertex in R_{1b}.

Since child Rnets are contained by their parent Rnet, a border b of an Rnet must be a border of some child Rnet at each lower level. For example, v_6 in Figure 2 is a border for R_{1b} and its parent R_1. This allows the shortcuts to be computed in a bottom-up manner, where shortcuts at level i are computed using those of level $i{+}1$, greatly reducing pre-computation cost. Only leaf Rnets require a Dijkstra's search on the original graph G.

ROAD uses a *Route Overlay* index and an *Association Directory* to efficiently compute kNNs. Recall that a vertex v may be a border of more than one Rnet. The Route Overlay index stores, for each vertex v, the Rnets for which it is a border along with the *shortcut trees* of v. The Association Directory provides a means to check whether a given Rnet or vertex contains an object or not. The kNN algorithm proceeds incrementally from the query vertex q in a similar fashion to INE. However, when ROAD expands to a new vertex v, instead of inspecting its neighbors, it consults the Route Overlay and Association Directory to find the highest level Rnet associated with it that does not contain any object. ROAD then relaxes all the shortcuts in this Rnet in a similar way to edges in INE, to bypass it. Of course when v is not a border of any Rnet or if all Rnets associated with v contain an object, it relaxes the edges of v exactly as in INE. The search terminates when k objects have been found or there are no further vertices to expand.

3.5 G-tree

G-tree [24, 25] also employs graph partitioning to create a tree index that can be used to efficiently compute network distances through a hierarchy of subgraphs. The partitioning occurs in a similar way to that of ROAD where the input graph G is partitioned into $f \geq 2$ subgraphs. Each subgraph is recursively partitioned until it contains no more than $\tau \geq 1$ vertices. For any subgraph G_i, $V_i \subseteq V$ is defined as the set of road network vertices contained within it. Any vertex $b \in V_i$ with an edge $e(b, v)$ where $v \notin V_i$ is defined as a border of G_i and all such vertices form the set of borders B_i. Figure 3 shows an example where the colored vertices v_5 and v_6 are borders for the subgraph $G_1{=}\{v_1, \cdots, v_6\}$.

The partitioned subgraphs naturally form a tree hierarchy with each node in the G-tree associated with one subgraph. Note that we use *node* to refer to the G-tree node while *vertex* refers to road network vertices. Notably a non-leaf node G_i does not need to store subgraph vertices, but only the set of borders B_i and a *distance matrix*. For non-leaf nodes, the distance matrix stores the network distance from each child node border to all other child node borders. For leaf nodes, it stores the network distance between each of its borders and the vertices contained in it.

Similar to the bottom-up computation of shortcuts in ROAD, the distance matrix of nodes at tree level i can be efficiently computed by reducing the graph to only consist of borders at level $i{+}1$ using the distance matrices of that level. Only leaf nodes require a Dijkstra's search on the original graph. Given a planar graph and optimal partitioning method, G-tree is a height-balanced tree with a space complexity of $O(|V| \log |V|)$. The similarities with ROAD are clear. One major difference is that G-tree uses its border-to-border distance matrices to "assemble" shortest path distances by the path through the G-tree hierarchy. We refer the reader to the original paper [24] for the details of the assembly method.

Another key difference is the kNN algorithm. To support efficient kNN queries, G-tree introduces the *Occurrence List*. Given an object set O, the Occurrence List of a G-tree node G_i lists its children that contain objects, allowing empty nodes to be pruned. The kNN algorithm begins from the leaf node that contains q, using an Dijkstra-like search to retrieve leaf objects. However, we found this leaf search could be further optimised and describe our improved leaf search algorithm in [6]. The algorithm then incrementally traverses the G-tree hierarchy from the source leaf. Elements (nodes or objects) are inserted into a priority queue using their network distances from q. The network distance to a G-tree node is computed using the assembly method by finding its nearest border to q. Queue elements are dequeued in a loop. If the dequeued element is a node, its Occurrence List is used to insert its children (nodes or object vertices) back into the priority queue. If the dequeued element is a vertex, it is guaranteed to be the next nearest object. The search terminates when k objects are dequeued.

A useful property of assembling distances is that, given a path through the G-tree hierarchy, distances can be *materialized* for already visited G-tree nodes. For example, given a query vertex q and two kNN objects in the same leaf node, after locating one of them, the distances to the borders of this leaf need not be recomputed.

Name	Region	# Vertices	# Edges
DE	Delaware	48,812	119,004
VT	Vermont	95,672	209,288
ME	Maine	187,315	412,352
CO	Colorado	435,666	1,042,400
NW	North-West US	1,089,933	2,545,844
CA	California & Nevada	1,890,815	4,630,444
E	Eastern US	3,598,623	8,708,058
W	Western US	6,262,104	15,119,284
C	Central US	14,081,816	33,866,826
US	United States	23,947,347	57,708,624

Table 1: Road Network Datasets

4. DATASETS

Here we describe the datasets used to supply the road network $G{=}(V, E)$ and set of object vertices $O \subseteq V$ for kNN querying.

4.1 Real Road Networks

We study kNN queries on 10 real-world road network graphs as listed in Table 1. These were created for the 9th DIMACS Challenge [3] from data publicly released by the US Census Bureau. Each network covers all types of roads, including local roads, and contains real edge weights for travel distances and travel times (both are used in our experiments). We also conduct in-depth studies for the United States (US) and North-West US (NW) road networks. The US dataset, covering the entire continental United

Object Set	United States		North-West US	
	Size	Density	Size	Density
Schools	160,525	0.007	4,441	0.004
Parks	69,338	0.003	5,098	0.005
Fast Food	25,069	0.001	1,328	0.001
Post Offices	21,319	0.0009	1,403	0.001
Hospitals	11,417	0.0005	258	0.0002
Hotels	8,742	0.0004	460	0.0004
Universities	3,954	0.0002	95	0.00009
Courthouses	2,161	0.00009	49	0.00005

Table 2: Real-World Object Sets

States, is the largest with 24 million vertices. The NW road network (with 1 million vertices), covering Oregon and Washington, represents queries limited to a smaller region or country. Notably this is the first time DisBrw has been evaluated on a network with more than $500,000$ vertices, previously not possible due to its high pre-processing cost (in terms of both space and time).

4.2 Real and Synthetic Object Sets

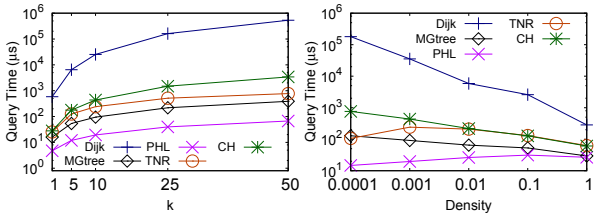
We created object sets based on both real-world points of interest (POIs) and synthetic methods as described below.

Real-World POI Sets. We created 8 real-world object sets (listed in Table 2) using data extracted from OpenStreetMap (OSM) [4] for locations of real-world POIs in the United States. Each object set is associated with a particular type of POI, e.g., all fast food outlets. POIs were mapped to road network vertices on both the US and NW road networks using their coordinates. While real POIs can be obtained freely from OSM, it is not a propriety system. As a result the data quality can vary, e.g., the largest object sets in OSM may not be representative of the true largest object sets and the completeness of POI data may vary between regions. So, in addition to real-world object sets, we generate synthetic sets to make generalizable and repeatable observations for all road networks.

Uniform Object Sets. A uniform object set is generated by selecting uniformly random vertices from the road network. As these objects are randomly selected road network vertices, they are likely to simulate real POIs, e.g., areas with more vertices have more POIs (e.g., cities) while those with fewer roads have less (e.g., rural areas). The density of objects sets d is varied from 0.0001 to 1, where d is the ratio of the number of objects $|O|$ to the number of vertices $|V|$ in the road network. High densities can simulate larger object sets which are common occurrences, e.g., ATM machines, parking spaces. Low densities correspond to the sparsely located POIs, e.g., post offices or restaurants in a particular chain. By decreasing the density we can simulate more difficult queries, as fewer objects imply longer distances and therefore larger search spaces. Uniform objects were used to evaluate G-tree in [24, 25].

Clustered Object Sets. While some POIs may be uniformly distributed other types, such as fast food outlets, occur in clusters. To create such clustered object sets, given a number of clusters $|C|$, we select $|C|$ central vertices uniformly at random (as above). For each central vertex, we select several vertices (up to a maximum cluster size C_{max}) in its vicinity, by expanding outwards from it. This distribution was used to evaluate ROAD in [16].

Minimum Object Distance Sets. The worst-case kNN query occurs when the query location is remote. To simulate this we create minimum distance object sets as follows. We choose an approximate centre vertex v_c by using the nearest vertex to the Euclidean centre of the road network. We find the furthest vertex v_f from v_c and set D_{max} as the network distance from v_c to v_f. For an object set R_i, $i \in [1, m]$, we choose $|O|$ objects such that the network distance from v_c to each object in R_i is at least $\frac{D_{max}}{2^{m-i+1}}$. For example for m=5, the set R_1 contains objects within the range $(\frac{D_{max}}{32}, D_{max}]$. Thus we investigate the effect of increasing minimum object distance by comparing query time on R_i with increasing i.



(a) Varying k (b) Varying Density

Figure 4: IER Variants (NW, d=0.001, k=10, uniform objects)

5. IER REVISITED

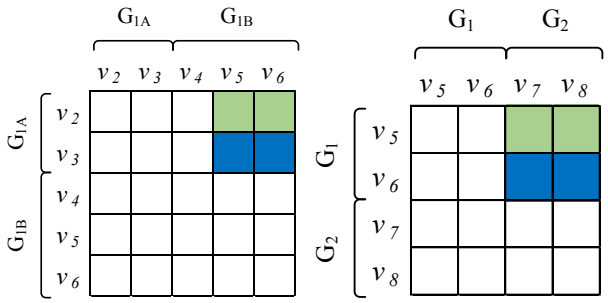
Network distance computation is a critical part of IER. However, to the best of our knowledge, all existing studies [16–18, 20] employ Dijkstra's algorithm to compute network distances. Dijkstra's algorithm is not only slow but it must also revisit the same vertices for subsequent network distance computations. Even if Dijkstra's algorithm is suspended and resumed for subsequent Euclidean NNs, this is necessarily no better than INE, which uses Dijkstra-like expansion until kNNs are found.

To understand the true potential of IER, we combined it with several fast techniques. *Pruned Highway Labelling* [7] is amongst the fastest techniques. It boasts fast construction times despite being a labelling method, but has similarly large index sizes. The G-tree assembly-based method mentioned earlier can also compute network distances. Notably, in a similar manner to G-tree's kNN search, the "materialization" property can be used to optimise repeated network distance queries from the same source (as in IER). The Dijkstra-like leaf-search can also be suspended and resumed. This is doubly advantageous for IER, as it becomes more robust to "false hits" (Euclidean NNs that are not real kNNs), especially if they are in the vicinity of a real kNN. We refer to this version of G-tree as MGtree. Finally we combined IER with *Contraction Hierarchies* (CH) [11] and *Transit Node Routing* (TNR) [8] using implementations made available by a recent experimental paper [23]. We use a grid size of 128 for TNR as in [23].

We compare the performance of IER using each method in Figure 4. PHL is the consistent winner, being 4 orders of magnitude faster than Dijkstra and an order of magnitude better than the next fastest method at its peak. G-tree, assisted by materialization, is the next best method. All methods converge with increasing density, as the search space becomes smaller. Note that CH is the technique used to answer local queries in TNR, which explains why TNR and CH are so similar for high densities as the distances are too small to use transit nodes. At lower densities, transit nodes are used more often, leading to a larger speed up. Given these results, in our main experiments, we include the two fastest versions of IER, i.e., PHL and MGtree. Note that the superiority of PHL and MGtree is also observed for other road networks and object sets.

6. IMPLEMENTATION IN MAIN MEMORY

Given the affordability of memory, the capacities available and the demand for high performance map-based services, memory-resident query processing is a realistic and often necessary requirement. However, we have seen in-memory implementation efficiency can affect performance to the point that algorithmic efficiency becomes irrelevant [22]. Firstly, this identifies the need to



(a) G_1 (b) G_0

Figure 5: Distance Matrices

understand how this can happen so that guidelines for efficient implementation may be developed. Secondly, it implies that some algorithms may possess intrinsic qualities that make them superior in-memory. The utility of the latter cannot be ignored. We first illustrate both aforementioned points using a case study and then outline typical choices and our approach to settle them.

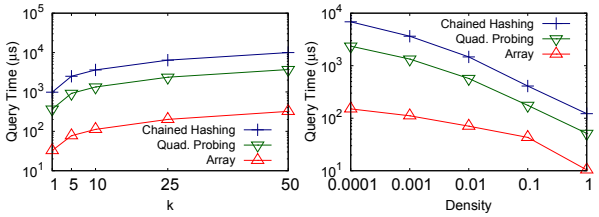
6.1 Case Study: G-tree Distance Matrices

G-tree's distance matrices store certain pre-computed graph distances (between borders of sub-graphs), allowing "assembly" of longer distances in a piece-wise manner. We firstly describe the G-tree assembly method below, then show how the implementation of distance matrices can significantly impact its performance.

Every G-tree node has a set of borders. From our running example in Figure 3, v_5 and v_6 are borders of G_1. Each non-leaf node also has a set of children, for example G_{1A} and G_{1B} are the children of G_1. These in turn have their own borders, which we refer to as "child borders" of G_1. A distance matrix stores the distances from every child border to every other child border. For example for G_1, its child borders are v_2, v_3, v_4, v_5, v_6, and its distance matrix is shown in Figure 5(a). But recall that a border of a G-tree node must necessarily be a border of a child node, e.g., the borders of G_1, v_5 and v_6, are also borders of G_{1B}. This means the distance matrix of G_1 repeatedly stores some border-to-border distances already in the distance matrix of G_{1B}, a redundancy that can become quite large for bigger graphs. To avoid this repetition and utilise, in general, $O(1)$ random retrievals, a practitioner may choose to implement the distance matrix as a hash-table. This has the added benefit of being able to retrieve distances for any two arbitrary borders.

Given a source vertex s and target t, G-tree's assembly method firstly determines the *tree path* through the G-tree hierarchy. This is a sequence of G-tree nodes starting from the leaf node containing s through its immediate parent and each successive parent node up to the least-common ancestor (LCA) node. From the LCA, the path traces through successive child nodes until reaching the leaf node containing t. The assembly method then computes the distances from all borders from the ith node in the path, G_i, to all borders in i+1th node, G_{i+1}. These two nodes are necessarily either both children of the LCA or have a parent-child relationship. In either case the parent node's distance matrix contains values for all border-to-border distances. Assuming we have computed all distances from s to the borders of G_i, we compute the distances to the borders of G_{i+1} by iterating over each border of G_i and computing the minimum distance through them to each border of G_{i+1}.

From our running example in Figure 3, let v_1 be the source and v_{12} be the target. In this case the beginning of the tree path will contain the child node G_{1A} and then its parent node G_1. Assume we have computed the distances to the borders of G_{1A} (easily done by using the distance matrix of leaf node G_{1A}, which stores leaf vertex



(a) Varying k (b) Varying Density

Figure 6: Distance Matrix Variants (NW, d=0.001, k=10)

to leaf border distances). Now we compute the distance to each border of G_1 from v_1, by finding the minimum distance through one of G_{1A}'s borders. To do this, for each of G_{1A}'s borders, we iterate over G_1's borders, retrieving distance matrix values for each pair (updating the minimum when a smaller distance is found). This is shown by the shaded cells in Figure 5(a). Similarly G_1 and its sibling G_2 are the next nodes in the tree path, and we again retrieve distance matrix values by iterating over two lists of borders. These values are retrieved from the matrix of the LCA node, G_0, and the values accessed are shaded in Figure 5(b).

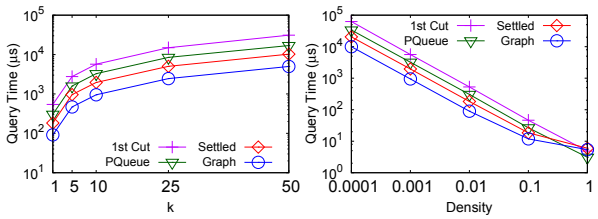
As we are iterating over lists (i.e., arrays) of borders, the distance matrix does not need to be accessed in an arbitrary order, as we observed in the G-tree authors' implementation. This is made possible by grouping the borders of child nodes as shown in Figure 5 and storing the starting index for each child's borders. Additionally we create an offset array indicating the position of the nodes' own borders in its distance matrix. For example, the offset array for G_1 indicates its borders (v_5 and v_6) are at the 3rd and 4th index of each row in its distance matrix shown in Figure 5(a). While Figure 5 shows the distance matrix as a 2D array, it is best implemented as a 1D array. This and the previously described accessed method, allow all shaded values to be accessed from sequential memory locations, thus displaying excellent spatial locality. This is shown in Figure 5 as the shaded cells are either contiguous or very close to being so. Spatial locality makes the code cache-friendly, allowing the CPU to easily predict and pre-fetch data into cache that will be read next. Otherwise the data would need to be retrieved from memory, which is 20−200× slower than CPU cache (depending on the level). This effect is amplified in real road networks as they contain significantly larger numbers of borders per node.

We compare three implementations of distance matrices, including the 1D array described above and two types of hash-tables: chained hashing [10] (STL `unordered_map`); and quadratic probing [10] (Google `dense_hash_map`). In Figure 6, chained hashing is a staggering 30 times slower than the array. While quadratic probing is an improvement, it is still an order of magnitude slower. Had we used either of the hash-table types, we would have unfairly concluded that G-tree was the worst performing algorithm.

Distance Matrix	INS	Cache Misses (Data)		
		L1	L2	L3
Chained Hashing	953 B	28.8 B	20.5 B	13 B
Quadratic Probing	1482 B	11.2 B	7.5 B	5.3 B
Array	151 B	1.5 B	0.4 B	0.3 B

Table 3: Hardware Profiling: 250,000 Queries on NW Dataset

We investigate the cache efficiency of each implementation in CPU cache misses at each level in billions in Table 3 (also showing INS, no. of instructions in billions) using `perf` hardware profiling of 250,000 varied queries on NW. Chained hashing uses indirection to access data, resulting in poor locality and the highest number of cache misses. Quadratic probing improves locality at the expense



(a) Varying k (b) Varying Density

Figure 7: INE Improvement (NW, d=0.001, k=10)

of more costly collision resolution, hence it uses more instructions than chained hashing. However, it cannot achieve better locality than storing data in an array sorted in the order it will be accessed. This ordering means the next value we retrieve from the array is far more likely to be in some level of cache. Unsurprisingly, it suffers from the fewest cache misses. This is a unique strength of G-tree's distance matrices and shows, while in-memory implementation is challenging, it is still possible to design algorithms that work well.

6.2 Guidelines for Implementation Choices

In-memory implementation requires careful consideration, or experimental outcomes can be drastically affected as seen with G-tree's distance matrices and in [22]. Many choices are actually quite simple, but their simplicity can lead to them being overlooked. Here we outline several choices and options to deal with them to assist future implementers. To illustrate the impact of these choices we progressively improve a first-cut in-memory implementation of INE. Each plot line in Figure 7 shows the effect of one improved choice. Each roughly halves the query time, with the final implementation of INE being $6-7\times$ faster.

1. Priority Queues. All methods in our study employ priority queues. In particular, INE and ROAD involve many queue operations and thus rely on their efficient implementation. Binary heaps are most commonly used, but we must choose whether to allow duplicate vertices in the queue or not. Without duplicates, the queue is smaller and queue operations involve less work. But this means the heap index of each vertex must be looked up to update keys e.g., through a hash-table. On degree-bounded graphs, such as road networks, the number of duplicates is small, and removing them is simply not worth the lost locality and increased processing time incurred with hash-tables. As a result, we see a $2\times$ improvement when INE is implemented without decreasing keys (see *PQueue* in Figure 7). Note that we use this binary heap for all methods.

2. Settled Vertex Container. Recall INE and ROAD must track vertices that have been dequeued from their priority queues (i.e., settled). The scalable choice is to store vertices in a hash-table as they are settled. However we observe an almost $2\times$ improvement, as shown by *Settled* in Figure 7 by using a *bit-array* instead. This is despite the need to allocate memory for $|V|$ vertices for each query. The bit-array has the added benefit of occupying $32\times$ less space than an integer array, thus fitting more data in cache lines. This does add a constant pre-allocation overhead for each query, which is proportionally higher for small search spaces (i.e, for high density). But the trade-off is worth it due to the significant benefit on larger search spaces (i.e., low density).

3. Graph Representation. A disk-optimised graph data structure was proposed for INE in [18]. In main memory, we may choose to replace it with an array of node objects, with each object containing an adjacency list array. However by combining all adjacency lists into a single array we are able to obtain another $2\times$ speed-up (refer to *Graph* in Figure 7). Firstly, we assign numbers to vertices from 0 to $|V|-1$. An *edges* array stores the adjacency list of each vertex consecutively in this order. The *vertices* array stores the starting index of each vertex's adjacency list in *edges*, also in order. Now for any vertex u we can find the beginning of its adjacency list in *edges* using *vertices*$[u]$ and its end using *vertices*$[u+1]$. This contiguity increases the likelihood of a cache hit during expansion. We similarly store ROAD's shortcuts in a global shortcut array, with each shortcut tree node storing an offset to this array. The principle demonstrated here is that recommended data structures in past studies cannot be used verbatim. It is necessary to replace IO-oriented data structures e.g., we replaced the B$^+$-trees, recommended in the originally disk-based DisBrw and ROAD, with sorted arrays.

Parameter	Values
Road Networks	DE, VT, ME, CO, **NW**, CA, E, W, C, **US**
k	1, 5, **10**, 25, 50
Density (d)	1, 0.1, 0.01, **0.001**, 0.0001
Synthetic POIs	**uniform**, clustered, min. obj. distance
Real POIs	Refer to Table 2

Table 4: Parameters (Defaults in Bold)

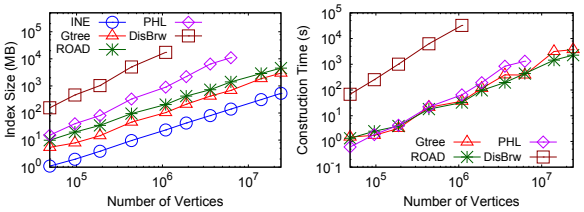
4. Language. C++ presently allows more low-level tuning, such as specifying the layout of data in memory for cache benefits, making it preferable in high performance applications. Implementers may consider other languages such as Java for its portability and design features. But when we implemented INE with all aforementioned improvements in Java (Oracle JDK 7), we found it was at least $2\times$ slower than the equivalent C++ implementation. One possible reason is that Java does not guarantee contiguity in memory for collections of objects. Also, the same objects take up more space in Java. Both factors lead to lower cache utilisation, which may penalise methods that are better able to exploit it.

7. EXPERIMENTS

7.1 Experimental Setting

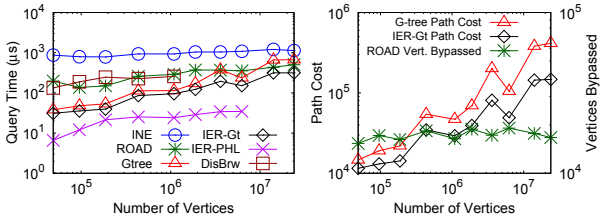
Environment. We conducted experiments on a 3.2GHz Intel Core i5-4570 CPU and 32GB RAM running 64-bit Linux (kernel 4.2). Our program was compiled with g++ 5.2 using the O3 flag, and all query algorithms use a single thread. To ensure fairness, we used the same subroutines for common tasks between the algorithms whenever possible. We implemented INE, IER, G-tree and ROAD from scratch. We obtained the authors code for G-tree, which we used to further improve our implementation, e.g., by selecting the better option when our choices disagreed with the authors' choice of data structures. For Distance Browsing, we partly based our SILC index on open-source code from [23], but being a shortest path study this implementation did not support kNN queries. As a result, we implemented the kNN algorithms ourselves from scratch, modifying the index to support them, taking the opportunity to make significant improvements (e.g., as discussed in Appendix A). We used a highly efficient open-source implementation of PHL made available by its authors [7]. All source code and scripts to generate datasets, run experiments, and draw figures have been released as open-source [2] for readers to reproduce our results or re-use in future studies.

Index Parameters. The performance of the G-tree and ROAD indexes are highly dependent on the choice of leaf capacity τ (G-tree), hierarchy levels l (ROAD) and fanout f (both) [16, 17, 24]. We experimentally confirmed trends observed in those studies and computed parameters for new datasets. As such, we use fanout f=4 for both methods. For G-tree we set τ to 64 (DE), 128 (VT, ME, CO), 256 (NW, CA, E), and 512 (W, C, US). For ROAD, we set l to 7 (DE), 8 (VT, ME), 9 (CO, NW), 10 (CA, E) and 11 (W, C,



(a) Index Size (b) Construction Time

Figure 8: Effect of Road Network Size $|V|$



(a) Query Time (b) G-tree & ROAD Stats

Figure 9: Effect of Road Network Size $|V|$ (d=0.001, k=10)

US). We chose values of l for ROAD in accordance with the results reported in [17] that show query performance of ROAD improves for larger l. Specifically, for each dataset, we increased l until either the query performance did not improve or further partitioning was not possible due to too few vertices in the leaf levels.

Query Variables. Table 4 shows the range of each variable used in our experiments (defaults in bold). Similar to past studies [24], we vary k from 1 to 50 with a default of 10. We used 8 real-world object sets as discussed Section 4. We vary uniform object set density d from 0.0001 to 1 where d=$|O|/|V|$ with a default value of 0.001. We choose this default density as it closely matches the typical density for real-world object sets as shown in Table 2. Furthermore this density creates a large enough search space to reveal interesting performance trends for methods. We vary over 10 real road networks (listed in Table 1) with median-sized NW and largest US road networks as defaults. We use distance edge weights in Sections 7.2 and 7.3 for comparison with past studies, and because IER and DisBrw were developed for such graphs. But we repeat experiments on travel times later in Section 7.5.

Query and Object Sets. All query times are averaged over 10,000 queries. For real-world object sets, we tested each set with 10,000 random query vertices. For uniform and clustered object sets, we generate 50 different sets for each density and number of clusters (resp.) combined with 200 random query vertices. For minimum distance object sets (described in Section 4.2), we generated 50 sets for each distance set R_i with $i \in [1, m]$. We also chose 200 random query vertices with distances from the centre vertex in range $[0, \frac{D_{max}}{2^m})$ (i.e., vertices closer than R_1) for use with all sets. We use m=6 for NW and m=8 for US to ensure there were enough objects in each set to satisfy the default density 0.001.

7.2 Road Network Index Pre-Processing Cost

Here we measure the construction time and size of the index used by each technique for all road networks in Table 1.

Index Size. Figure 8(a) shows the index size for each algorithm. INE only uses the original graph data structure, so its size can be seen as the lower bound on space. DisBrw could only be built for the first 5 road networks before exceeding our memory capacity. This is not surprising given the $O(|V|^{1.5})$ storage complexity. However, in our implementation, we were able to build DisBrw for an index with 1 million vertices (NW) consuming 17GB. PHL also exhibits large indexes, however it can still be built for all but the 2 largest datasets. We note that PHL experiences larger indexes on travel distance graphs because they do not exhibit prominent hierarchies needed for effective pruning (on travel time graphs we were able to build PHL for all indexes [6]). G-tree consumed less space than ROAD. E.g., for the US dataset G-tree used 2.9GB compared to ROAD's 4.4GB. As explained in past studies [24], ROAD's Route Overlay contains significant redundancy as multiple shortcut trees repeatedly store a subset of the Rnet hierarchy.

Construction Time. Figure 8(b) compares the construction time of each index for increasing network sizes. DisBrw again stands out as its index (SILC [21]) requires an all-pairs shortest path computation. However, the computation of each SILC quadtree is independent and can be easily parallelized. We observed a speed-up factor of very close to 4× with our quad-core CPU using OpenMP. Note that other methods cannot be so easily parallelized. Despite this DisBrw still required 9 hours on NW, while parallelization is useful it does not change the asymptotic behaviour. PHL takes longer than G-tree and ROAD but surprisingly not significantly so, thanks to pruned labelling [7]. IER's index performance depends on the network distance method it employs (i.e., G-tree or PHL).

Recall that both ROAD and G-tree must partition the road network. Since the network partitioning problem is known to be NP-complete, ROAD and G-tree both employ heuristic algorithms. As both methods require the same type of partitioning we use the same algorithm, the multilevel graph partitioning algorithm [14] used in G-tree. This method uses a much faster variant of the Kernighan-Lin algorithm recommended in ROAD [16]. Consequently, we are able to evaluate ROAD for much larger datasets for the first time, with ROAD being constructed in less than one hour for even the largest dataset (US) containing 24 million vertices. The construction time of ROAD is comparable to G-tree, because both use the same partitioning method, and employ bottom-up methods to compute shortcuts and distance matrices, respectively.

We remark that, while most existing studies have focused on improving query processing time, there is a need to develop algorithms and indexes providing comparable efficiency with a focus on reducing memory usage and construction time.

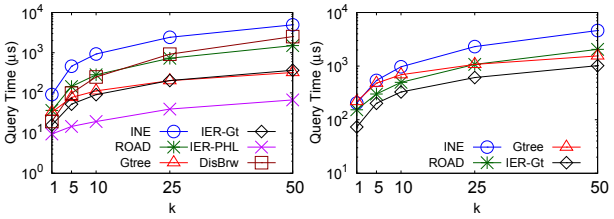
7.3 Query Performance

We investigated kNN query performance over several variables: road network size, k, density, object distance, clusters, and real-world POIs. Implementations have been optimized according to Section 6. We have applied numerous improvements to each algorithm, as detailed in Appendix A and [6]. IER network distances are computed using both PHL (when its index fits in memory) and G-tree with materialization (shown as IER-PHL and IER-Gt, resp.).

7.3.1 Varying Network Size

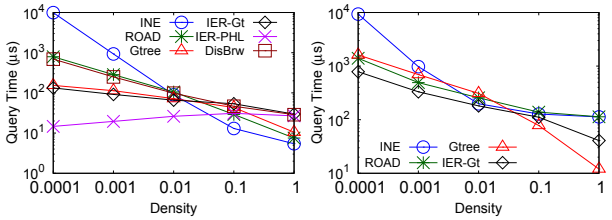
Figure 9(a) shows query times with increasing numbers of road network vertices $|V|$ for all 10 road networks in Table 1 on uniform objects. We observe the consistent superiority of IER-based methods. Figure 9(a) clearly shows the reduced applicability of DisBrw. Even though its performance is close to ROAD, its large index size makes it applicable on only the first 5 datasets.

Surprisingly G-tree's advantage over ROAD decreases with increasing network size $|V|$. Recall that ROAD can be seen as an optimisation on INE, where the expansion can bypass object-less regions (i.e., Rnets). Thus ROAD's relative improvement over INE depends on the time saved bypassing Rnets versus additional time spent descending shortcut trees. In general, given the same density, we can expect a similar sized region to contain the same number of objects irrespective of the network size $|V|$. This explains why



(a) NW Dataset (b) US Dataset

Figure 10: Effect of k (d=0.001)



(a) NW Dataset (b) US Dataset

Figure 11: Effect of Density (k=10)

INE remains relatively unaffected by $|V|$. It also means that regions without objects are similarly sized. Although Rnets may grow, the size of the Rnets we do bypass also grow, so ROAD bypasses similar numbers of vertices. So the time saved bypassing regions does not increase greatly. Thus ROAD's query time with increasing $|V|$ mainly depends on the depth of shortcut trees. But the depth is bounded by l, which we know does not increase greatly, and as a result ROAD scales extremely well with increasing $|V|$.

G-tree's non-materialized distance computation cost is a function of the number of borders of G-tree nodes (i.e., subgraphs) involved in the tree path to another node or object. With increasing network size, a G-tree node at the same depth has more borders and the path cost is consequently higher. Thus, we see G-tree "catch-up" to ROAD on the US dataset. These trends are demonstrated in 9(b). G-tree's path cost (in border-to-border computations) increases while the number of vertices ROAD bypasses remains stable with increasing $|V|$ (note these are not directly comparable).

7.3.2 *Varying k*

Figures 10(a) and 10(b) show the results for varying k for the NW and US datasets, respectively, on uniform objects. Significantly, IER-PHL is 5× faster than any other method on NW. While PHL could not be constructed for the US dataset for travel distances, IER-Gt takes its place as the fastest method, being twice as fast as G-tree. Interestingly, this is despite both using the same index, also materializing intermediate results, and IER-Gt having the additional overhead of retrieving Euclidean NNs. So this is really an examination of heuristics used by G-tree. Essentially G-tree visits the closest subgraph (i.e., by one of its borders) while IER-Gt visits the subgraph with the next Euclidean NN. IER-Gt can perform better because its heuristic incorporates an estimate on distances to objects within subgraphs while G-tree does not. Each time G-tree visits a subgraph not containing a kNN it pays a penalty in the cost of non-materialized distance computations. We have seen this cost increases with network size, which explains why the improvement of IER-Gt is greater on the US than on NW. This is verified in Figure 9(b), which shows IER-Gt involves fewer computations than G-tree and the gap increases with network size.

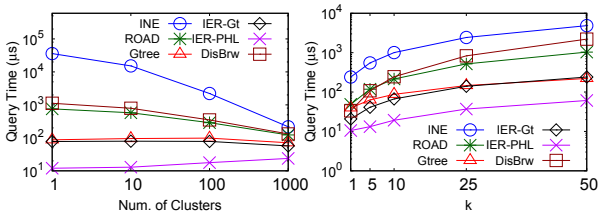
We observe that G-tree outperforms ROAD, DisBrw and INE on NW, with a trend similar to previous studies [24]. INE is the slowest as it visits many vertices. For $k = 1$ the ROAD, DisBrw and G-tree methods are indistinguishable as a small area is likely to contain the NN. ROAD and DisBrw scale very similarly with k. G-tree scales better than both, at its peak nearly an order of magnitude better than ROAD and DisBrw. As more objects are located, more paths in the G-tree hierarchy are traversed, allowing greater numbers of subsequent traversals to be materialized. As explained in Section 7.3.1, we again see G-tree's relative improvement over ROAD decrease in Figure 10(b) for the larger US dataset.

7.3.3 *Varying Density*

We evaluate performance for varying uniform object densities in Figure 11. With increasing density the average distance between objects decreases and in general query times are lower. The rate of improvement for heuristic-based methods (DisBrw, G-tree, IER) is slower because they are less able to distinguish better candidates. For IER this means more false hits, explaining why IER-PHL's query times increase (slightly) as it has no means to re-use previous computations like IER-Gt does. The rate of improvement is higher for expansion-based methods as their search spaces become smaller. ROAD falls behind INE beyond density 0.01 indicating the tipping point at which the time spent traversing shortcut trees exceeds the time saved bypassing Rnets (if any). The query times plateau at high densities on the US dataset for ROAD and INE because it is dominated by the bit-array initialization cost (refer to Section 6.2). G-tree performs well at high densities as more kNNs are found in the source leaf node. In this case it reverts to a Dijkstra-like search (which we improved [6]) providing comparable performance to INE and ROAD on NW. G-tree exceeds them on the US as a bit-array is not required due to G-tree's leaf search being limited to at most τ vertices.

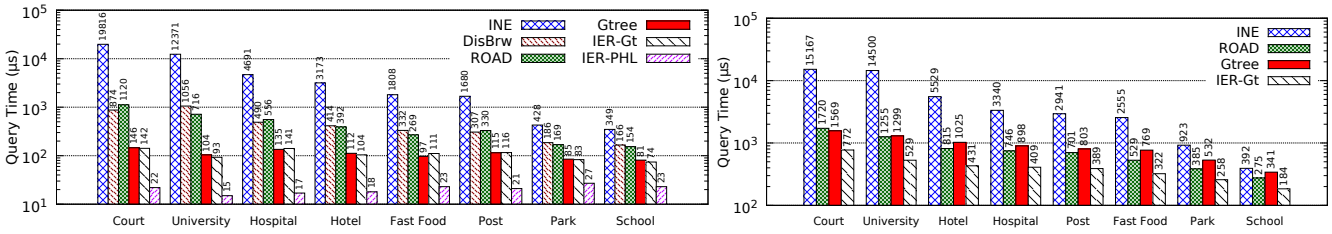
7.3.4 *Varying Clusters*

In this section we evaluate performance on clustered object sets proposed in Section 4.2. Figure 12 shows the query time with increasing numbers of clusters and varying k. In both cases cluster size is at most 5. Figure 12(b) uses an object density of 0.001. As the number of clusters increases the average distance between objects decreases leading to faster queries. This is analogous to increasing density, thus showing the same trend as for uniform objects. IER-PHL's superiority is again apparent. One difference to uniform objects is IER-based methods find it more difficult to differentiate between candidates as the number of clusters increases, and query times increase (but not significantly). Similarly in Figure 12(b), as k increases, IER-PHL visits more clusters, causing its performance lead to be slightly smaller than for uniform objects. IER-Gt on the other hand is more robust to this, as it is able to materialize most results. G-tree again performs better than DisBrw and ROAD. Due to clustering, objects in the same cluster will likely be located in the same G-tree leaf node. After finding the first object, G-tree can quickly retrieve other objects without recomputing distances to the leaf node, thus remaining relatively constant.



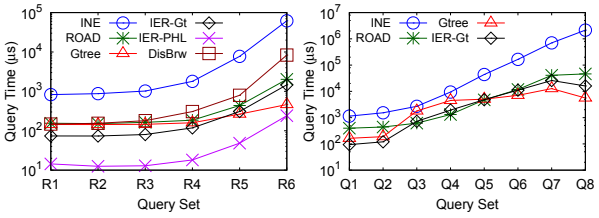
(a) Varying No. of Clusters (b) Varying k

Figure 12: Effect of Clustered Objects (NW, d=0.001, k=10)



(a) NW Road Network (b) US Road Network

Figure 13: Varying Real-World Object Sets (Defaults: k=10)



(a) NW Dataset (b) US Dataset

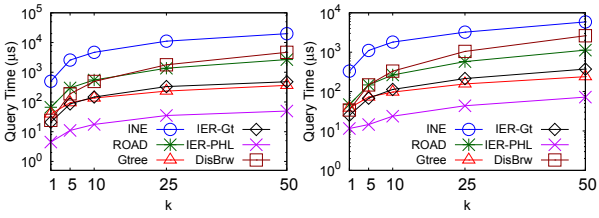
Figure 14: Varying Min. Obj. Distance (d=0.001, k=10)

7.3.5 *Varying Minimum Object Distance*

Each set R_i in Figure 14 represents an exponentially increasing network distance to the closest object with increasing i, as described in Section 4.2. For the smallest sets, objects still tend to be found further away, as there are fewer closer vertices. However as distance increases further, we see the effect of "remoteness". INE scales badly due to the increasing search space. IER-based methods scale poorly as the Euclidean lower bounds becomes less accurate with increasing network distance. This is particularly noticeable in Figure 14(b) as G-tree eventually overtakes IER-Gt on the US. But IER-PHL still outperforms all methods on NW. DisBrw performs poorly for a similar reason, making many interval refinements. G-tree scales extremely well in both cases, as more paths are visited through the G-tree hierarchy, more computations can be materialized for subsequent traversals.

7.3.6 *Real-World Object Sets*

Varying Object Sets. In Figure 13, we show query times of each technique on typical real-world object sets from Table 2. These are ordered by decreasing size, which is analogous to decreasing density, showing the same trend as in Figure 11. Schools represent the largest object set and all methods are extremely fast as seen for high density. A more typical POI, like hospitals, are less numerous and show the differences between methods more clearly. Regardless, IER-PHL on NW and IER-Gt on US consistently and significantly outperform other methods on all real-world object sets. Also note query times for G-tree are higher on US than NW for the same sets, confirming our observations in Section 7.3.1.



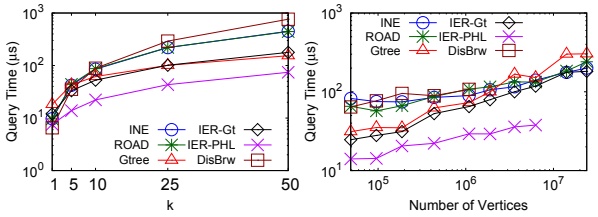
(a) Hospitals (b) Fast Food

Figure 15: Varying k for Real-World Objects (NW, k=10)

Varying k. Figure 15 shows the behaviour of two typically searched POIs, fast food outlets and hospitals, on the NW dataset. Hospitals display a trend similar to that of uniform objects for increasing k, as they tend to be sparse. IER-PHL is again significantly faster than G-tree. Although still fastest, IER-PHL has slightly lower performance for fast food outlets as these tend to appear in clusters where Euclidean distance is less able to distinguish better candidates, similar to synthetic clusters in Figure 12(b). Thus trends observed for equivalent synthetic object sets in previous experiments are also observed for real-world POIs.

7.3.7 *Original Settings*

A recent experimental comparison [24] used a higher default density of d=0.01. While we choose a more typical default density, we reproduce results using d=0.01 in Figure 16 for varying k and network size. Note that we use the smaller Colorado dataset in Figure 16(a) for direct comparison with [24]. Firstly, all methods compared in [24] now answer queries in less than 1ms. While our CPU is faster, it cannot account for such a large difference. This suggests our implementations are indeed efficient. Secondly, most methods are difficult to differentiate, as such a high density implies a very small search space (i.e., queries are "easy" for all methods).



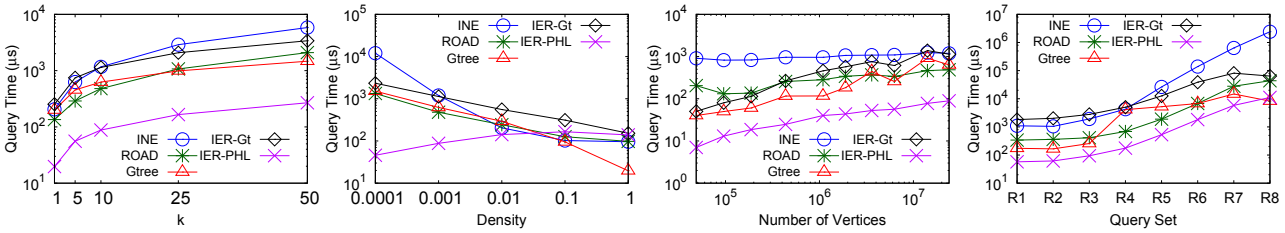
(a) Varying k (b) Varying $|V|$

Figure 16: kNN Queries (CO, d=0.01, k=10)

7.4 Object Set Index Pre-Processing Cost

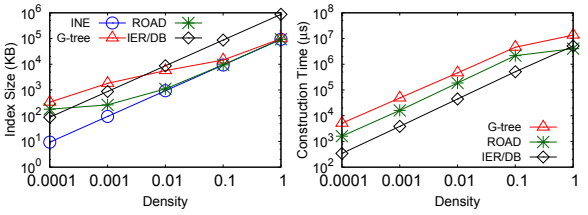
The original ROAD paper [16] included pre-processing of a fixed object set in its road network index statistics. But there may be many object sets (e.g., one for each type of restaurant) or objects may need frequent updating (e.g., hotels with vacancies). So we are interested in the performance of individual object indexes over varying size (i.e., density). We evaluate 3 object indexes on the US dataset, namely: *R-trees* used by IER, *Association Directories* used by ROAD and *Occurrence Lists* used by G-tree. Note that in our study DisBrw also uses R-trees [6].

Index Size. In practice object indexes for all object sets would be constructed offline, loaded into memory and the appropriate one injected at query time. We investigate the index sizes (in KB) in Figure 18(a) to gauge what effect each density has on the total size. The size of the input object set used by INE is the lower bound storage cost. ROAD's object index is smaller than G-tree's because it



(a) Varying k (b) Varying Density d (c) Varying $|V|$ (d) Varying Min. Obj. Distance

Figure 17: Query Performance on Travel Time Graphs (US, k=10, d=0.001, uniform objects)



(a) Index Size vs. d (b) Build Time vs. Density d

Figure 18: Object Indexes for US (uniform objects)

need only store whether an Rnet contains an object or not, which is easily done in a low memory bit-array. G-tree's object index must additionally store the child nodes containing objects. Both indexes must however store the actual objects, which gradually dominates the index size with increasing density. Note that we chose R-tree parameters (e.g., node capacity) for best performance. As a result R-trees fall behind after density 0.01, but this can be remedied by increasing the node capacity at the expense of Euclidean kNN performance. We note that object indexes are much smaller than road network indexes, as they are simpler data structures, and real-world object sets with high densities are less frequent.

Construction Time. Figure 18(b) similarly shows the object index construction times. Again, they are all constructed much faster than road network indexes, due to being simpler data structures. The ROAD and G-tree object indexes incur the largest build time due to bottom-up propagation of the presence of objects through their respective hierarchies. However, the R-trees used by IER are significantly faster to build. As R-trees support updates, this suggests the possibility of use in real-time settings.

7.5 Travel Time Road Networks

kNNs may just as commonly be required in terms of travel time. In this section we reproduce query results for the US road network with travel time edge weights. Results for other experiments on travel times can be found in [6], but we note any differences here.

Extending IER. IER uses the Euclidean distance as a lower bound on the network distance between two points for travel distance edge weights. This can easily be extended for other edge weights. Let w_i (resp. d_i) represent the edge weight (resp. Euclidean length) of an edge e_i. We compute $S = max_{\forall e_i \in E}(d_i/w_i)$. E.g., if w_i represents travel time, S corresponds to the maximum speed on any edge in the network. Let $d_E(p, q)$ be the Euclidean distance between two points p and q. It is easy to see that $d_E(p, q)/S$ is a lower bound on the network distance between p and q, e.g., the time it takes to travel the Euclidean distance at the maximum possible speed. Thus, we compute S for the network and use the new lower bound in IER. Landmarks are known to provide better lower bounds on travel time graphs [12]. However there is no equivalent data structure, such as an R-tree, to incrementally retrieve candidates by their lower bound, making them undesirable for use here.

Unlike travel distances, we were able to construct the PHL index for *all* datasets, with the largest requiring 16GB [6]. This is due to "highway" properties exhibited in travel time graphs (e.g., an edge with a higher speed is more likely to be on a shortest path) leading to smaller label sizes. Figure 17 shows the query times for travel times with varying k, density, network size $|V|$ and object distance on the US dataset. In general, IER experiences more false hits due to the looser lower bound on travel times, explaining why IER-Gt is now significantly outperformed by G-tree. But, surprisingly, IER-PHL still remains the fastest method in most situations. The penalty in false hits is partly offset by the reduced label sizes for PHL. The looser lower bound also aggravates cases where Euclidean distance was less effective on travel distances. For example, IER was already less able to distinguish better candidates with increasing density, and as result IER-PHL degrades faster on travel times in Figure 17(b). This is similarly observed for increasing network distance in Figure 17(d) for the same reason. Despite this, IER-PHL remains the fastest method in most cases. Other trends observed for travel distances are similarly observed for travel times (e.g., G-tree degrades with increasing $|V|$ in Figure 17(c)).

Criteria	INE	G-tree	ROAD	IER	DisBrw
Query Performance					
Default Settings	5th	2nd	=3rd	1st	=3rd
Small k	5th	=3rd	=3rd	1st	2nd
Large k	5th	2nd	3rd	1st	4th
Low Density	5th	2nd	=3rd	1st	=3rd
High Density	1st	3rd	2nd	4th	5th
Small Networks	5th	2nd	=3rd	1st	=3rd
Large Networks	4th	=3rd	2nd	1st	N/A
Network and Object Index Pre-Processing					
Time (Network)	1st	3rd	2nd	4th	5th
Time (Objects)	1st	5th	4th	=2nd	=2nd
Space (Network)	1st	2nd	3rd	4th	5th
Space (Objects)	1st	5th	2nd	=3rd	=3rd

Table 5: Ranking of Algorithms Under Different Criteria

8. CONCLUSIONS

We have presented an extensive experimental study for the kNN problem on road networks, settling unanswered questions by evaluating object indexes, travel time graphs and real-world POIs. We verify that G-tree generally outperforms INE, DisBrw and ROAD, but the relative improvement is much smaller and at times reversed, demonstrating the impact of implementation efficiency. Table 5 provides the ranking of the algorithms under different criteria.

Our most significant conclusions are regarding IER, which we investigated with fast network distance techniques for the first time. IER-PHL significantly outperformed every competitor in all but a few cases, even on travel time graphs where Euclidean distance is less effective. IER provides a flexible framework that can be combined with the fastest shortest path technique allowed by the users'

memory capacity and must be included in future comparisons. Additionally, on travel distances, we saw that IER-Gt often outperformed the original G-tree kNN algorithm despite using the same index. As this suggests Euclidean NN can be a better heuristic, it identifies room for improvement in kNN search heuristics. Perhaps more information can be incorporated into object indexes.

Finally, we investigated the effect of implementation choices using G-trees distance matrices and data structures in INE. By investigating simple choices, we show that even small improvements in cache-friendliness can significantly improve algorithm performance. As such there is a need to pay careful attention when implementing and designing algorithms for main memory, and our insights are applicable to any technique not just those we study.

Acknowledgements. We sincerely thank the authors of G-tree [24], PHL [7] and the shortest path experimental paper [23] for providing source code of their algorithms, in particular Y. Kawata for valuable assistance in improving the code of PHL. We also thank the anonymous reviewers for their feedback through which we significantly improved our work. The research of Muhammad Aamir Cheema is supported by ARC DE130101002 and DP130103405.

9. REFERENCES

[1] https://www.globalwebindex.net/blog/top-global-smartphone-apps.
[2] https://github.com/tenindra/RN-kNN-Exp.
[3] http://www.dis.uniroma1.it/%7Echallenge9/.
[4] http://www.openstreetmap.org.
[5] http://www.cs.utah.edu/%7Elifeifei/SpatialDataset.htm.
[6] T. Abeywickrama, M. A. Cheema, and D. Taniar. k-nearest neighbors on road networks: A journey in experimentation and in-memory implementation. *CoRR*, abs/1601.01549, 2016. http://arxiv.org/abs/1601.01549.
[7] T. Akiba, Y. Iwata, K.-i. Kawarabayashi, and Y. Kawata. Fast shortest-path distance queries on road networks by pruned highway labeling. In *ALENEX*, pages 147–154, 2014.
[8] H. Bast, S. Funke, D. Matijevic, P. Sanders, and D. Schultes. In transit to constant time shortest-path queries in road networks. In *WEA*, pages 46–59, 2007.
[9] H.-J. Cho and C.-W. Chung. An efficient and scalable approach to cnn queries in a road network. In *VLDB*, pages 865–876, 2005.
[10] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.
[11] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *WEA*, pages 319–333, 2008.
[12] A. V. Goldberg and C. Harrelson. Computing the shortest path: A search meets graph theory. In *SODA*, 2005.
[13] H. Hu, D. L. Lee, and V. C. S. Lee. Distance indexing on road networks. In *VLDB*, pages 894–905, 2006.
[14] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.
[15] M. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB*, pages 840–851, 2004.
[16] K. Lee, W.-C. Lee, B. Zheng, and Y. Tian. Road: A new spatial object search framework for road networks. *TKDE*, 24(3):547–560, 2012.
[17] K. C. K. Lee, W.-C. Lee, and B. Zheng. Fast object search on road networks. In *EDBT*, pages 1018–1029, 2009.
[18] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *VLDB*, pages 802–813, 2003.
[19] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2005.
[20] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, 2008.
[21] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *GIS*, pages 200–209, 2005.
[22] D. Šidlauskas and C. S. Jensen. Spatial joins in main memory: Implementation matters! *PVLDB*, 8(1):97–100, 2014.
[23] L. Wu, X. Xiao, D. Deng, G. Cong, A. D. Zhu, and S. Zhou. Shortest path and distance queries on road networks: An experimental evaluation. *PVLDB*, 5(5):406–417, Jan. 2012.
[24] R. Zhong, G. Li, K. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *TKDE*, 27(8):2175–2189, Aug 2015.
[25] R. Zhong, G. Li, K.-L. Tan, and L. Zhou. G-tree: An efficient index for knn search on road networks. In *CIKM*, pages 39–48, 2013.

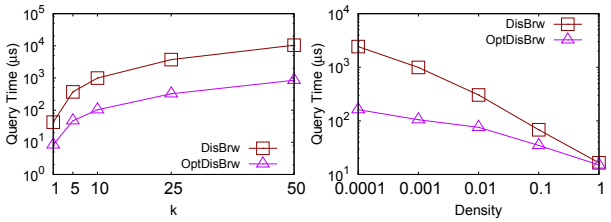
APPENDIX

A. IMPROVED ALGORITHMS

We made numerous improvements to all algorithms that, in contrast to efficient implementation, are applicable in any setting. Here we describe an improvement of DisBrw that may be useful for future studies. All other improvements (including updated pseudocode for each algorithm) can be found in [6].

Real road network graphs consist of large numbers of degree-2 vertices. Generally 30% of vertices have degree-2 for road networks in Table 1, e.g., on the US dataset 30.3% of vertices have degree-2 (another 19.9% have degree-1). These may exist to capture details such as varying speed limits or curvature. This degree distribution can have a significant impact on computing shortest paths, and we demonstrate the potential improvement on DisBrw.

SILC uses the quadtrees and coloring scheme described in Section 3.3 to iteratively compute the vertices in a shortest path, at a cost of $O(\log |V|)$ for each vertex. We use *chain* to refer to a path consisting only of vertices with degree-2 or less, e.g., a section of motorway with no exits. Let v be the current vertex in the shortest path from s to t and u be the previous vertex in the shortest path. If v is on a chain, we do not need to consult the quadtree because the next vertex in the shortest path *must* be the neighbor of v that is not u. This saves $O(\log |V|)$ for each degree-2 vertex in the shortest path. In fact, if target t is not on the chain, we can directly "jump" to the last vertex in the chain saving several $O(\log |V|)$ lookups. This observation can be easily exploited by storing the two ends of the chain for each vertex with degree less than 2.



(a) Varying k (b) Varying Density

Figure 19: Deg-2 Optimisation (NA-HWY, d=0.001, k=10)

This optimisation can significantly improve DisBrw query times. We refer to this version as OptDisBrw. For our default NW dataset this results in a 30% improvement [6] coinciding with the number of degree-2 vertices quoted above. However some road networks have an even larger proportion of degree-2 vertices, such as the highway road network for North America used in past studies [16, 17, 20] with 175,813 vertices [5], 95% of which are degree-2. In this case OptDisBrw is up to an order of magnitude faster than DisBrw as shown in Figure 19, as the average chain length is significantly higher resulting in longer jumps. Accordingly future work must keep degree-2 vertices in mind for potential optimisations. Given these results we use chain optimised refinement for DisBrw in our experiments.