TOAIN: A Throughput Optimizing Adaptive Index for Answering Dynamic kNN Queries on Road Networks

Siqiang Luo[†], Ben Kao[†], Guoliang Li[‡], Jiafeng Hu[†], Reynold Cheng[†], Yudian Zheng[*]
[†]The University of Hong Kong [‡]Tsinghua University [*]Twitter Inc.
[†]{sqluo, kao, jhu, ckcheng}@cs.hku.hk, [‡]liguoliang@tsinghua.edu.cn, [*]yudianz@twitter.com

ABSTRACT

We study the classical kNN queries on road networks. Existing solutions mostly focus on reducing query processing time. In many applications, however, system throughput is a more important measure. We devise a mathematical model that describes throughput in terms of a number of system characteristics. We show that query time is only one of the many parameters that impact throughput. Others include update time and query/update arrival rates. We show that the traditional approach of improving query time alone is generally inadequate in optimizing throughput. Moreover, existing solutions lack flexibility in adapting to environments of different characteristics. We propose TOAIN, which is a very flexible algorithm that can be easily trained to adapt to a given environment for maximizing query throughput. We conduct extensive experiments on both real and synthetic data and show that TOAIN gives significantly higher throughput compared with existing solutions.

PVLDB Reference Format:
Siqiang Luo, Ben Kao, Guoliang Li, Jiafeng Hu, Reynold Cheng, Yudian Zheng. TOAIN: A Throughput Optimizing Adaptive Index for Answering Dynamic *k*NN Queries on Road Networks. *PVLDB*, 11(5): 594 - 606, 2018.
DOI: https://doi.org/10.1145/3177732.3177736

1. INTRODUCTION

With the advances in mobile technologies and GPS-equipped devices, location-based services are becoming increasingly popular. Examples include taxi-hailing services such as Uber [5] and Lyft [4], and location-based games, such as *Pokémon GO*. Among all the operations supported by location-based systems, k-nearest-neighbor (or kNN) search on road networks is an important fundamental operation. For example, Uber needs to locate the cars in its fleet that are the closest to a customer's location who has issued a taxi request; A game server needs to find the *Pokémon*s that are the closest to a player in the *Pokémon GO* game.

In these applications, *distance* is often measured by the *traveling distance* or *traveling time* between two locations on a road network. Typically, a road network is modeled as a graph such that each node represents a road junction and each edge represents a road segment connecting two road junctions. For the kNN problem, we consider a set $\mathcal{M}$ of m objects, each is located at a point in a road segment. (For example, an object can represent a taxi in Uber or a *Pokémon* in the game.) Given a query q, which is represented as a point in the road network (e.g., the location of a commuter who has issued a taxi request, or the location of a *Pokémon* player), the problem is to locate the k nearest objects of q in $\mathcal{M}$ in terms of network distance.

The kNN problem on road networks has been studied extensively and many sophisticated data structures and algorithms have been proposed. Previous works, however, mostly focus on improving the query processing time (or query time for short). Query time, as we argue, is a *microscopic* view of the overall system performance. In the context of applications like taxi-hailing services and location-based games, other performance indicators, such as the *end-to-end query response time* and *system throughput*, are even more important. The objective of this paper is to provide a *macroscopic* view of the performance of a system that supports location-based services. In particular, we propose a mathematical model for optimizing system throughput subject to certain quality-of-service (QoS) constraints. We also propose a new flexible indexing scheme that allows such optimization to be realized in practice.

To facilitate our discussion, let us first explain some key concepts and system characteristics that are related to system performance.

[Queries and updates] We assume that kNN queries come from an extensive user base and that query arrivals are stochastic. For example, Didi, which is a taxi-hailing service in China, receives more than a thousand taxi-hailing requests per second during peak hours [6], and that it served about 7 million requests per day countrywide in 2015. Each such request generates a kNN query to locate the closest servicing taxis to a commuter. As another example, in the US, there were around 20 million daily active *Pokémon GO* players in 2016, each spent roughly 25 minutes playing each day on average. A player can activate the "nearby tracking" feature in the game, which helps the player locate the nearest *Pokémon*s. We note that kNN queries generated in these applications are stochastic, in particular, their arrival times are not pre-determined.

We assume that the set of objects $\mathcal{M}$ is *dynamic*, and consider the following two modes of changes: (1) The location of an object is changed. (2) An object is added to or removed from $\mathcal{M}$. Changes of the first kind model movements of cars (whose locations are continuously changing), while changes of the second kind model the *Pokémon* game in which the presence of a *Pokémon* at a location is transient (it shows up for a period of time and then disappears). Since a location-based service has to keep track of the whereabouts of objects, any changes made to the objects would trigger *updates* to the system. The number of updates generated in the system depends on the number of objects in $\mathcal{M}$ and the frequency at which the objects change. The *update load* could be substantial in large-scale systems. For example, there are about 4 million active drivers

Proceedings of the VLDB Endowment, Vol. 11, No. 5

DOI: https://doi.org/10.1145/3177732.3177736

in China providing Didi services. Each vehicle typically reports its location periodically with a periodicity of 3∼5 seconds.

[Query time] We use the term *query time* to refer to the amount of time taken by a system to process an *isolated* kNN query. With the state-of-the-art data structures and algorithms, one can achieve sub-millisecond and even down to tens of micro-second query time.

[Query Response time] The response time of a query q refers to the amount of time taken between q's arriving at the system and the time at which q's answer is computed. Queries and updates compete among themselves on data accesses and CPU cycles. These data and CPU contentions impose major delay in query processing, especially under high query and update arrival rates. In a high load situation, the server serializes the executions of queries and updates via a queuing policy. The response time of a query is the queuing delay plus its query (processing) time.

[Throughput] We define *throughput* as the number of queries that the system answers per unit time interval. Due to the stochastic nature of query arrivals and the variations of query and update times, throughput varies from one unit time interval to another. In the following discussion, the term *throughput* is used to refer to the average throughput of the system over an extensive period of time. For location-based services, throughput is an important performance measure. For example, Didi receives more than a thousand requests per second during peak hours, and yet it is reported that Didi has tapped into only about 1% of the potential market in China. As another example, *Pokémon GO* used to provide the "nearby tracking" feature. The feature was removed in July 2016 due to server overload [3]. In order to expand market share and to improve user experience in these applications, the system should be designed to maximize its throughput.

[Quality of Service (QoS)] The throughput of a system can be increased by admitting more queries. Doing so, however, would generally lower the quality of the service. As more queries are admitted to the system, queues start building up, which causes longer delays to queries and updates. This results in longer query response times and worse data freshness. In this paper we consider two QoS indicators: the average query response time R_q and the average update response time R_u. In particular, we assume that certain tolerance levels are given that serve as bounding constraints on R_q and R_u.

Existing solutions. There is an extensive number of previous works done on improving the query time of kNN search on road networks. However, query time is a microscopic view of system performance. From the perspective of users, the end-to-end query response time, which takes into account the queuing delay, is a more important measure of user experience. From the perspective of the service provider, throughput is a more important measure of the cost effectiveness of the system. It is a misconception that a smaller query time naturally translates into a higher throughput; After all, if each query is answered faster, more can be done. The fallacy is due to the oversight of the update load. Behind the ingenuity of a query-optimized algorithm often do we see an elaborate indexing scheme. Hence, a reduction in query time is often accompanied by a higher update cost. Given the contentions among queries and updates, response times and throughput can be inadvertently worsened. As an example, we compare the performance of a number of kNN algorithms in this paper. Our experiment shows that in some extreme but not unrealistic cases, G-tree [34], which is a state-of-the-art algorithm, is outperformed by the simple *Dijkstra* algorithm (which builds no indexes) in terms of throughput. As we will show later in this paper, we can mathematically model throughput with respect to a number of variables, including query/update times and their arrivals. These variables, in turns, depend on the algorithm (which induces an intricate tradeoff between query and update times) and the physical characteristics of a road network. Existing algorithms, however, provide little flexibility in the adjustment of the query/update tradeoff. This inflexibility leads to poor throughput and in-adaptability of the algorithms when they are applied to road networks of different physical characteristics.

Contributions. We summarize our contributions as follows:

- (Modeling) We consider different models concerning queries or updates arrivals and the queuing discipline that describe the operations of location-based services. We put forward a mathematical model that expresses the maximum throughput in terms of a number of key variables, subject to certain QoS constraints.
- (Data structure and algorithm) We propose the algorithm TOAIN, which considers the various variables and the mathematical model to estimate the maximum throughput. TOAIN uses a shortcut-based index called SCOB to speed up query processing. SCOB is a highly tunable structure that allows the query/update tradeoff to be adjusted adaptively. Given an application and its characteristics, TOAIN auto-tunes SCOB in order to achieve the best throughput.
- (Experimental evaluation) We evaluate the performance of TOAIN, comparing it against other state-of-the-art solutions. Our results show that TOAIN significantly outperforms the competitors over a wide spectrum of scenarios.

The rest of the paper is organized as follows. Section 2 discusses related works. Section 3 formally describes our models. Section 4 gives a mathematical analysis of throughput. Section 5 describes the TOAIN algorithm and the SCOB index. Section 6 presents experimental results. Finally, Section 7 concludes the paper.

2. RELATED WORKS

The problem of answering kNN queries on road networks have been extensively studied. In this section we briefly mention a few representative solutions. These include *Dijkstra* [14], IER [24], DisBrw [27], ROAD [20, 19], G-tree [34], and V-tree [30].

Dijkstra's algorithm [14] (or *Dijkstra*) is one of the best-known algorithms. The algorithm can be used to determine the shortest distances of all the nodes in a graph from a source node. Given a source node q, the algorithm expands the graph (which initially consists of only the node q) and visits other nodes in the order of their distances from q. *Dijkstra* can be used to answer kNN queries by expanding the graph just enough to locate the k closest objects to node q. *Dijkstra* is a simple baseline solution. In particular, it does not use an index and so object update costs are very low.

ROAD [20] is based on *Dijkstra*. It speeds up kNN query processing for cases where objects are sparsely located in a network. Specifically, ROAD partitions a graph into many subgraphs (called Rnets). These Rnets are merged to form larger Rnets in a hierarchical fashion. An indicator is associated with each Rnet signaling whether the Rnet contains any objects. During a *Dijkstra* expansion, if an Rnet with no objects is to be explored, the search inside the Rnet is skipped. Compared to *Dijkstra*, ROAD gives a faster query time at the expense of an update cost; when an object is updated, the indicators of some Rnets have to be updated accordingly.

G-tree [34] builds a similar subgraphs hierarchy like ROAD, but instead of an indicator, each subgraph is associated with an *Occurrence-List (OL)*. G-tree records the objects that are located in each leaf sub-graph; The OL of each non-leaf subgraph is a collection of the lists of IDs of its descendants in the hierarchy. An efficient assembly method using the OLs to answer kNN queries is given in [34]. The algorithm is further improved in [8]. We remark that the OLs store more object information than the indicators used in ROAD. Thus, G-tree is generally faster than ROAD in terms of query time but slower in update time.

V-tree [30] employs a similar hierarchical structure as G-tree. V-tree identifies *border nodes* that are at the *boundaries* of subgraphs. By maintaining the lists of nearest objects to these border nodes, efficient techniques are devised to answer kNN queries.

In this paper we evaluate *Dijkstra*, ROAD, G-tree, and V-tree against our proposed solution TOAIN. There are other existing solutions, such as IER [24] and DisBrw [28, 27]. Since these solutions have been shown to be generally outperformed by the other algorithms we mentioned, we do not consider them in this paper.

There are also previous works that tackle the continuous kNN query problem on road networks [29, 11, 10, 18, 23, 13, 33, 17]. Many of these studies (e.g., [29, 11, 10, 18, 33, 17]) assume moving query points and stationary objects. These studies are thus orthogonal to ours. The moving-objects model is studied in [23, 13]. Their focus, however, is efficient maintenance of kNN results of static *standing queries*. Under our model, queries are one-shots and they arrive in random fashion. Our model follows more closely applications like taxi-hailing and location-based gaming.

Finally, our solution is based on the Contraction Hierarchy (CH) algorithm [15], which was designed for efficient shortest distance queries. We will introduce the CH algorithm in Section 5.1.

3. MODELS

In this section we present the models that describe a location-based system that answers kNN queries on a road network.

3.1 Road Network Model

Following previous studies (e.g., [34, 22, 31, 35]), we model a road network as a directed graph $G(V, E)$, where a node $u \in V$ represents a road junction and a directed edge $(u, v) \in E$ represents a road segment from junction u to junction v. Each edge is associated with a weight $w(u, v)$ which indicates the distance of the road segment (edge) (u, v). We consider a set of objects $\mathcal{M}$. Each object $o \in \mathcal{M}$ is located at a point on a road segment. Objects in $\mathcal{M}$ are dynamic in that their presence/absence in $\mathcal{M}$ as well as their locations could change. Given two nodes $s, t \in V$, we use $s \rightsquigarrow t$ to denote a shortest path from s to t in the graph G. We use $d_G(s, t)$ to denote the distance of the corresponding path in G. Given an object o located on a road segment (u_o, v_o) such that o is at a distance of w_o from the node u_o, we map o to node u_o with the offset w_o registered. Given a query q, which is located on a road segment (u_q, v_q) such that q is at a distance of w_q from node v_q, we map q to node v_q with an offset w_q. We extend the distance function d_G so that it measures the network distance between points on road segments. Specifically, the distance from query q to object o is defined as $d_G(q, o) = d_G(v_q, u_o) + w_q + w_o$.

DEFINITION 1 (kNN). *Given a query q, a set of objects $\mathcal{M}$ on a road network G, and a constant k, the kNN query returns k objects $o_1, \ldots, o_k \in \mathcal{M}$ such that $d_G(q, o_i)$ $1 \leq i \leq k$ are the k smallest ones among all objects in $\mathcal{M}$.*

We assume that queries/objects are mapped to network nodes and that distances are computed with the offsets properly taken into account as discussed above. In the following discussion, for simplicity, we ignore these offsets and assume that queries and objects are located at network nodes. This saves us from the minor details of distance computation and greatly simplifies our discussion.

3.2 System Models

Figure 1 illustrates a system serving kNN queries. There are two sources of tasks, namely queries and updates. Tasks that arrive at the system are first put into a queue, and they are served by the system under certain queuing policy.

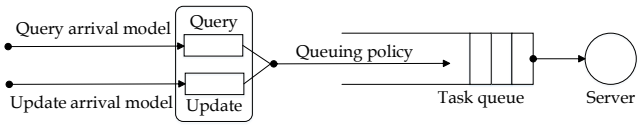


Figure 1: System model.

Arrival models. We assume that queries arrive at the system as a Poisson process. For updates, we consider two scenarios:

[*Batch Update Arrival (BUA)*] Note that in the taxi-hailing application, each object in $\mathcal{M}$ reports its location periodically (a.k.a. heartbeat protocol). In this scenario, we divide time into periods, each of a duration of T seconds. We assume that $m = |\mathcal{M}|$ updates are collected at the beginning of each period, one from each object. If an update collected in a period is not installed in the database by the end of that period, the update is dropped because the update is superseded by the one of the same object in the next period.

[*Random Update Arrival (RUA)*] This corresponds to the location-based game scenario in which objects appear/disappear/relocated in a random fashion. In this scenario we model update arrivals as another Poisson process.

Queuing models. We consider two queuing policies:

[*First-come-first-served (FCFS)*] Queries and updates are served in the order of their arrival times.

[*Query-first (QF)*] Queries have higher priorities over updates. In particular, all queries in the queue are placed in front of all updates, and queries are served FCFS among themselves. Moreover, when a query reaches the head of the queue and if the system is currently serving an update, the update will be *preempted* by the query. QF is often employed in real-time databases for applications such as programmed stock trading [16] and for prioritizing query/update in web databases [25].

Note that the QF policy favors queries over updates. It thus reduces query queuing delay but extends that of updates. In some applications, QF is justifiable because reducing query response time is more important than update timeliness. Take taxi-hailing application as an example. Query response time delay has a direct impact on the user experience in the responsiveness of the system. Even seconds of delay can be directly felt. On the other hand, a few second delay in installing an update means the system is computing a shortest distance based on data that is a few second stale. The estimated distance (e.g., in terms of traveling time) is generally off by an insignificant amount (e.g., a few seconds of inaccuracy in estimating a 5-minute ride is relatively insignificant).

In this paper we study two specific system models, namely, (1) BUA+QF and (2) RUA+FCFS to illustrate our idea of a flexible and adaptive kNN algorithm. As we have explained, the first combination closely models the taxi-hailing application. We also chose to study the second model because it is quite general. We remark that other arrival/queuing models are possible. Due to space limitations, we confine this study to combinations (1) and (2).

4. THROUGHPUT ANALYSIS

In this section we present a mathematical model that expresses throughput in terms of the arrivals and computations of queries and updates under the two system models. Table 1 lists the symbols we use.

4.1 BUA+QF model

Under this model, time is divided into periods, each of length T seconds. Within each period, $m = |\mathcal{M}|$ updates are collected

Table 1: Notations.

Notation	Description
(t_q, V_q)	expected/variance-of query time
(t_u, V_u)	expected/variance-of update time
λ_q, λ_u	query/update arrive rate
R_q	average query response time
R_q^*	average query response time bound, a QoS measure
λ_q^*	largest average throughput subject to an R_q^* constraint
T	update periodicity (under the QF model)
$m = \|\mathcal{M}\|$	number of objects
$G = (V, E)$	graph representing a road network
$d_G(u, v)$	shortest distance between u and v in G
$u \rightsquigarrow v$	a shortest path from u to v
$u \curvearrowright v$	a shortcut from u to v
$u_\downarrow$	the set of downhill objects of u

at the beginning of the period. Let λ_q be the query arrival rate (in number per second). Furthermore, let t_q, V_q be the average and the variance of query (processing) time, respectively; and t_u, V_u be those of updates. If the queuing system is *stable*, i.e, the workload arrival rate does not exceed the system's servicing rate, the expected response time of queries, R_q, is given by the following lemma.

LEMMA 1. *Under the BUA+QF model,* $R_q = \frac{\lambda_q(t_q^2+V_q)}{2(1-\lambda_q t_q)} + t_q$, *if the queuing system is stable.*

PROOF. Since queries always preempt updates under BUA+QF, the response times of queries are unaffected by the presence of updates. If one considers only the queuing and execution of queries, the system is an M/G/1 queue. By the Pollaczek-Khinchine formula [12], we have, $R_q = \frac{\rho+\lambda_q\mu V_q}{2(\mu-\lambda_q)} + t_q$, where μ is the service rate and $\rho = \lambda_q/\mu$ is the utilization of system. The lemma immediately follows by substituting $\mu = 1/t_q$. □

From Lemma 1, we see that R_q increases with λ_q. In other words, as throughput (i.e., λ_q) increases, so is the average query response time. From our discussion in the introduction, to ensure a good QoS, one option is to limit how large R_q gets. Specifically, we assume a QoS requirement that R_q is not more than an average response time bound R_q^*. Hence, the maximum average throughput (i.e., largest query arrival rate), denoted by λ_q^*, supported by the system under the QoS requirement is given by,

$$\left(\frac{\lambda_q^*(t_q^2+V_q)}{2(1-\lambda_q^* t_q)} + t_q \le R_q^*\right) \Rightarrow \left(\lambda_q^* \le \frac{2(R_q^*-t_q)}{V_q + 2R_q^* t_q - t_q^2}\right) \quad (1)$$

Moreover, as the system admits more queries, less time is left for installing updates. To avoid dropping too many updates, the system should reserve enough time in a period to process updates that are collected in that period. Hence, $T - T \cdot \lambda_q^* \cdot t_q \ge m \cdot t_u$, and so:

$$\lambda_q^* \le (T - mt_u)/(T \cdot t_q). \quad (2)$$

Combining Equations 1 and 2, we have,

$$\lambda_q^* \le \min\left\{\frac{2(R_q^*-t_q)}{V_q + 2R_q^* t_q - t_q^2}, \quad (T-mt_u)/(T\cdot t_q)\right\}. \quad (3)$$

Let us simplify Equation 3 for an easier interpretation of the formulae. Let

$$\alpha = R_q^*/t_q; \qquad \beta = mt_u/T; \qquad \gamma = V_q/t_q^2.$$

α gives the ratio between the response time tolerance (R_q^*) and the average query time (t_q). For example, if t_q = 1ms and users accept a query response time of within 1s, then α = 1s/1ms = 1,000. β gives the fraction of time that the system is processing updates. γ is the squared coefficient of variation of query time. For the various index structures and algorithms we tested, γ ranges from 0.1 to 0.9. Equation 3 can be rewritten as:

$$\lambda_q^* \le \begin{cases} \frac{1}{t_q}\cdot\frac{2\alpha-2}{\gamma+2\alpha-1}, & \text{if } (\gamma+2\alpha-1)\beta < \gamma+1; \\ (1-\beta)/t_q, & \text{if } (\gamma+2\alpha-1)\beta \ge \gamma+1. \end{cases} \quad (4)$$

For typical cases where $2\alpha \gg 1 \gg \gamma$, Equation 4 can be approximated by,

$$\lambda_q^* \le \begin{cases} 1/t_q, & \text{if } \alpha\beta < 1/2 \quad \text{(QoS-bound mode)} \\ (1-\beta)/t_q, & \text{if } \alpha\beta \ge 1/2 \quad \text{(Update-bound mode)} \end{cases} \quad (5)$$

From Equation 5, we see that the system can be operating in either one of two modes: (1) If the update load is small (small β) and/or the response time QoS requirement is stringent (small R_q^*, leading to a relatively small α) such that $\alpha\beta < 1/2$, the system is essentially constrained by the response-time QoS requirement and it is operating as if it is serving only queries. In this case, the maximum average throughput is limited by the servicing rate ($1/t_q$) of queries. We call this the *QoS-bound mode*. On the other hand, if update load is high (large β) and/or the QoS requirement is loose such that $\alpha\beta \ge 1/2$, the effect of update processing kicks in. The system is left with only $1-\beta$ of its capacity to process queries and so λ_q^* is limited by $(1-\beta)/t_q$. We call this the *update-bound mode*.

An interesting observation is that if a system is operating in the QoS-bound mode, we should opt for a query-efficient algorithm (so that t_q is small) in order to achieve a large maximum throughput. However, if we scale the system up to serve more objects (e.g., more taxis, thus a larger m), β will become larger, and at some point, $\alpha\beta$ will exceed 1/2 and the system will switch to the update-bound mode. In this case, the maximum throughput is also limited by the update cost. In other words, a system that is engineered only towards fast query time may not scale well.

4.2 RUA+FCFS model

Under this model, updates arrive as a Poisson process with arrival rate λ_u. The system can be described by a 2-class (corresponding to queries and updates) FIFO queue. The average query response time is given by the following lemma (A proof of the lemma is given in our technical report [1].):

LEMMA 2. *Under the RUA+FCFS model,*

$$R_q = \frac{\lambda_u(V_u+t_u^2) + \lambda_q(V_q+t_q^2)}{2(1-\lambda_q t_q - \lambda_u t_u)} + t_q \quad (6)$$

if the queuing system is stable.

In order for the system to be stable, the input workload cannot exceed the system's servicing capacity. Therefore,

$$\lambda_q t_q + \lambda_u t_u \le 1. \quad (7)$$

Again, R_q (Equation 6) is an increasing function of λ_q. Bounding R_q by R_q^* and considering Equation 7, the maximum average throughput λ_q^* under the RUA+FCFS model is given by,

$$\lambda_q^* \le \min\left\{\frac{2(R_q^*-t_q)(1-\lambda_u t_u) - \lambda_u(V_u+t_u^2)}{V_q + 2R_q^* t_q - t_q^2}, \frac{1-\lambda_u t_u}{t_q}\right\}. \quad (8)$$

In addition to the query response time QoS, one can also impose an update response time QoS. Let R_u denote the average update response time, by symmetry, we have, similar to Equation 6:

$$R_u = \frac{\lambda_u(V_u+t_u^2) + \lambda_q(V_q+t_q^2)}{2(1-\lambda_q t_q - \lambda_u t_u)} + t_u. \quad (9)$$

We can bound the quantity in Equation 9 by a maximum tolerable value R_u^* as an update response time QoS requirement. This will further restrain λ_q^* with a quantity similar to the one shown in Equation 8. For simplicity, we do not consider this update QoS requirement in the rest of this paper.

From Equations 3 and 8, we see that λ_q^* depends not only on the query time t_q, but on a number of other variables, in particular t_u. As we have discussed, there is generally a tradeoff between t_q and t_u. A smaller query time t_q is likely achieved at the expense of a longer update time t_u (e.g., due to a more elaborate indexing scheme). Take the last quantity, $(1-\lambda_u t_u)/t_q$, shown in Equation 8 as an example. This term shows that if updates arrive at a high rate, λ_q^* will be significantly suppressed due to a large λ_u. In this case, a lower update cost (smaller t_u) is preferable to a lower query time (smaller t_q). From the equations, it can be seen that a flexible algorithm that can fine tune its performance in terms of t_u and t_q is very desirable in achieving the highest throughput possible.

5. ALGORITHMS

In this section we describe our algorithm TOAIN, which answers kNN queries with an objective of maximizing throughput. TOAIN is inspired by the shortcut-based *Contraction-Hierachy algorithm* (or CH for short) [15], which is designed for answering shortest-path distance queries. Based on *shortcuts*, we propose an adaptive index called SCOB, which is operated on by TOAIN to answer kNN queries. An important feature of SCOB is that it is highly *tunable* in that we can tilt it towards more query-efficient or towards more update-efficient. As we will see later, adjusting SCOB allows TOAIN to adapt to different application environments while pushing for a higher throughput. In the following, we first describe our adaptation of the CH search (Section 5.1). Then, we introduce our SCOB index, explaining how it is constructed and maintained (Section 5.2). After that, we describe how TOAIN adjusts SCOB given an application environment (Section 5.3).

5.1 CH Search

The CH search enables an efficient shortest distance computation with the help of *shortcuts*. A shortcut, denoted by $u \curvearrowright v$, is an edge *derived* from graph G that connects nodes $u, v \in V$. Shortcut $u \curvearrowright v$ is given a weight $d_G(u, v)$, which is the shortest distance from u to v in G. The shortest-distance query of a u-v pair can be answered immediately if $u \curvearrowright v$ and its weight is computed. There are $O(|V|^2)$ possible shortcuts. To limit the number, we define a ranking function $r(u), \forall u \in V$, and define the *shortcut set*, $SC_<$.

DEFINITION 2 (SHORTCUT SET $SC_<$). *Given a graph $G = (V, E)$ and a ranking function r on V, a shortcut $u \curvearrowright v \in SC_<$ iff (1) $u, v \in V$, (2) $r(u) < r(v)$, and (3) $r(z) < r(u)$ for any intermediate node z in a shortest path $u \rightsquigarrow v$ from u to v.*

Verbally, Condition (3) in Defintion 2 means that as one traverses from u en route to v via a shortest path in G, all nodes encountered other than the terminal ones are of lower ranks than that of u. In the following discussion, we will also use $SC_<$ to denote the *shortcut graph*, which is one that consists of *only* shortcut edges. In particular, we write $d_{SC_<}(u, v)$ as the shortest path length from u to v in the shortcut graph. The subscript $(<)$ in $SC_<$ indicates that a shortcut links a node u to another node v that is of a *straightly* higher rank (see Condition 2 of Definition 2). Since the weights of shortcuts are derived directly from G, we have,

LEMMA 3. $d_G(u, v) \leq d_{SC_<}(u, v) \quad \forall u, v \in V$.

Before we describe how the CH algorithm works with the shortcut set, we make two simplifying assumptions (to be relaxed in Section 5.2.2):

A1: G is undirected. A2: Ranks of nodes are all distinct.
These assumptions will greatly simplify our discussions and allow us to stay focused on the main idea of TOAIN and SCOB. We will present the technical details of relaxing the assumptions in later sections of the paper. We will also discuss how to design the ranking function $r()$ in Section 5.3.

As an illustration, Figure 2 shows a shortest path $s \rightsquigarrow t = (s, a, b, x, c, d, t)$ from a node s to a node t. The numbers in brackets give the ranks of the nodes. In the figure, nodes of higher ranks are drawn closer to the top; lower-rank ones are closer to the bottom. The figure illustrates 4 shortcuts: $(s \curvearrowright a)$, $(a \curvearrowright x)$, $(t \curvearrowright d)$, and $(d \curvearrowright x)$. (We assume that the graph is undirected and so are paths. Hence, we have the shortcuts such as $d \curvearrowright x$.) Note that node b is bypassed by the shortcut $a \curvearrowright x$ because $r(b) = 4 < 5 = r(a)$.

To see how shortcuts help determine shortest path distances, we consider the following definition of a *summit node*.

DEFINITION 3 (SUMMIT NODE). *Given a shortest path $s \rightsquigarrow t$ from a node s to a node t, the summit node x of $s \rightsquigarrow t$ is the node with the highest rank in $s \rightsquigarrow t$.*

By Assumption A2, all ranks are distinct, and hence the summit node x is unique. The shortest path $s \rightsquigarrow t$ thus consists of two segments, $s \rightsquigarrow x$ and $x \rightsquigarrow t$. The fact that x has the highest rank in $s \rightsquigarrow t$ implies that one can reach x from s via only shortcuts. For example, referring to Figure 2, s is connected to x by shortcuts $(s \curvearrowright a)$ and $(a \curvearrowright x)$. Likewise, one can reach x from t via shortcuts. We have the following lemma on summit nodes (see [1] for a proof):

LEMMA 4. *If x is the summit node of a shortest path $s \rightsquigarrow t$ in graph G, then $d_G(s, t) = d_{SC_<}(s, x) + d_{SC_<}(t, x)$.*

Given two nodes s and t, to compute their shortest distance, $d_G(s, t)$, the CH algorithm performs two *Dijkstra* explorations on the shortcut graph $SC_<$, one originates from s and another from t. The two *Dijkstra* searches are executed concurrently in alternating lockstep fashion. The idea is to "discover" the shortcut path from s to the summit x and the one from t to x. Once the two *Dijkstra* searches meet at the summit, the path length $d_G(s, t)$ is obtained by the sum of the shortcut weights as given by Lemma 4. From Figure 2, we see that the two *Dijkstra* searches are figuratively like climbing a hill. We thus call the *Dijkstra* search that originates from the source (destination) to the summit an *s-climb* (*t-climb*).

Readers are referred to [15] for the details of the CH algorithm and [1] for our shortcut creation algorithm.

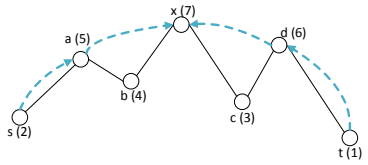


Figure 2: Shortcuts and CH search illustration.

5.2 The SCOB Index

The CH algorithm is designed to answer shortest-distance queries. A straightforward way to apply CH to find the kNN of a given query node q is to determine $d_G(q, o_i)$ using CH for every object $o_i \in \mathcal{M}$, and return the k objects with the shortest distances. This amounts to doing $m = |\mathcal{M}|$ s-climbs from node q and m t-climbs, one from each node where an object o_i resides. This approach requires $2m$ *Dijkstra* searches and is inefficient.

To improve query time, we should reduce the number of *climbs* done. We observe that all s-climbs originate from q and hence they can be combined into one single *Dijkstra* search. To avoid the m t-climbs, we propose the SCOB index. The idea of the SCOB index is to store at each summit node (e.g., node x in Figure 2) the id's and distances of some objects that are located *downhill* of the summit in the shortcut graph (e.g., an object located at node t and its distance from node x). By doing so, as the s-climb from q reaches a summit node, say x, the downhill objects of x can be retrieved and their distances from q can be computed. As we will see shortly, only k downhill objects need to be kept at each summit node in SCOB.

DEFINITION 4 (DOWNHILL OBJECTS). *An object o located at a node v is a downhill object of a node u if there is a path from v to u in the shortcut graph $SC_<$. We use $u_\downarrow$ to represent the set of downhill objects of u.*

We further define *top-k downhill nearest neighbors* (kDNNs) and a lemma concerning them:

DEFINITION 5 (kDNNS). *Given a node $u \in G$, the top-k downhill nearest neighbors (kDNNs) of u, denoted by kDNN(u), is a set of objects such that (1) if the number of downhill objects of u is less than or equal to k, then kDNN(u) contains all the downhill objects of u; (2) otherwise, kDNN(u) contains the k downhill objects o_i's of u that give the smallest $d_{SC_<}(o_i, u)$.*[1]

LEMMA 5. *Given a node s and an object o located at a node t, if object o is one of the kNNs of node s, then $o \in k$DNN(x), where x is the summit node on a shortest path $s \rightsquigarrow t$.*

PROOF. First, since x is the summit node of $s \rightsquigarrow t$, t is connected to x via only shortcuts (see, e.g., Figure 2). Hence, o, which is located at node t, is a downhill object of x. We assume by contradiction that $o \notin k\text{DNN}(x)$. Then, there are at least k objects $o_1, ..., o_k$, such that $d_{SC_<}(o_i, x) < d_{SC_<}(o, x)$, $\forall i \in [1, k]$. Based on Lemmas 3 and 4, we have,

$$d_G(s, o_i) \leq d_G(s, x) + d_G(x, o_i) \leq d_{SC_<}(s, x) + d_{SC_<}(o_i, x) < d_{SC_<}(s, x) + d_{SC_<}(o, x) = d_G(s, o).$$

Hence, the k o_i's are all closer to s than o is. So, object o cannot be one of the kNNs of s. Contradiction ensues. □

Our SCOB index stores for each node u its kDNN(u). For our target applications, such as taxi-hailing and location-based games, where k is typically small, kDNN(u) is implemented as a vector of object-distance pairs $[(o_1, d_{SC_<}(o_1, u)), \ldots, (o_k, d_{SC_<}(o_k, u))]$ sorted in ascending order of distances. We remark that for applications where k is large, kDNN(u) can be implemented with more efficient priority queue structures, such as a binary max-heap. From Lemma 5, we know that the kNNs of a query q can be obtained from the kDNNs of certain summit nodes. As a result, we only need to perform one s-climb (from q) and no t-climbs are needed. Algorithm 1 outlines the procedure for answering a kNN query using the SCOB index. Specifically, we maintain a result set $\mathcal{R}$, which contains the best-known kNN objects found so far. As we perform *Dijkstra* search from q on the shortcut graph $SC_<$, when a node p is visited, if $d_{SC_<}(q, p)$ is larger than the k-th longest distance of the objects in $\mathcal{R}$, no further exploration through p will be done; otherwise, we retrieve kDNN(p) and check if any objects in there let us refine the best-known results.

[1]In case there are multiple objects that are tie as the k-th smallest-distance objects, we break the tie, for example, by their object id's. For simplicity, we do not explicitly mention the tie-breaking in the rest of the paper but assume that it is done.

Algorithm 1: `Query(`$SC_<$`, q, k)`

```
1  R ← ∅; d_0 ← longest distance of an object in R from q, initially ∞;
   /* Conduct Dijkstra search from q on SC_<, with
      the following operations.                          */
2  for each node p being visited in the Dijkstra search do
3      if ||R|| ≥ k and d_{SC_<}(q, p) > d_0 then
4          break ;
5      for object o ∈ kDNN(p) do
6          add [o, d_{SC_<}(q, p) + d_{SC_<}(o, p)] into R;
7          if ||R|| > k then
8              remove from R the object with longest distance from q.
9          update d_0;
10 return R;
```

5.2.1 SCOB Construction and Updates

We next show how objects are inserted to and deleted from SCOB. SCOB construction can be considered as a sequence of inserts. An object update can be treated as a delete followed by an insert.

Algorithm 2: `Insert(`$SC_<$`, o, k)`

```
   /* Suppose o is located at node t; Conduct
      Dijkstra search from t on SC_<, with the
      following operations.                              */
1  for each node p being visited in the Dijkstra search do
2      add [o, d_{SC_<}(t, p)] into kDNN(p);
3      if ||kDNN(p)|| > k then
4          delete from kDNN(p) the object with longest distance from
           p;
```

[Insert] To insert an object o that is located at a node t, we need to update kDNN(u) for all nodes u of which o is a downhill object. By Definition 4, u is reachable from t in the shortcut graph $SC_<$. The insert can thus be achieved by a *Dijkstra* search from t on $SC_<$. Specifically, when a node u is visited in the search, we compute $d_{SC_<}(t, u)$. If $d_{SC_<}(t, u)$ cracks top k in kDNN(u), object o and its distance $d_{SC_<}(t, u)$ is added to kDNN(u), ousting another object if appropriate.

[Delete] We first consider two properties of downhill objects.

PROPERTY 1. *An object o located at a node u is a downhill object of u.*

PROPERTY 2. *Given a node u, $u_\downarrow = (\bigcup_{(v \curvearrowright u) \in SC_<} v_\downarrow) \cup \{$all objects located at $u\}$.*

Property 1 follows immediately from Defintion 4 by considering the (null) path from node u to itself. For Property 2, if an object o located at a node t is a downhill object of a node v (i.e., $o \in v_\downarrow$), then by Definition 4, there is a path from t to v in $SC_<$. Concatenate that path with $(v \curvearrowright u)$ gives us a path from t to u in $SC_<$. Hence, $o \in u_\downarrow$. Combine this with Property 1, Property 2 follows.

Property 2 says that the downhill objects of u are given by the objects located at u plus all the downhill objects of those nodes v's, where each v is linked to u by a shortcut $v \curvearrowright u$. Hence, kDNN(u) can be determined directly from kDNN(v)'s and the objects located at u. Considering that the v's are of lower ranks than u (by the fact that they are linked to u by shortcuts), this suggests that the kDNNs of nodes can be updated recursively in a node-rank order. The object delete procedure makes use of this observation. Specifically, to delete an object o originally located at a node t, if $o \notin k$DNN(t), no updates to SCOB is needed. Otherwise, o is removed from kDNN(t) and we perform an (*uphill*) *Dijkstra* search

Algorithm 3: `Delete(`$SC_<$`, o, k)`

```
   /* Suppose o is located at node t;                          */
1  if o ∈ kDNN(t) then
2      F ← ∅;
       /* Conduct Dijkstra search from t; lines 3∼6
          show the operations done in the search */
3      for each node u being visited in the Dijkstra search do
4          if o ∈ kDNN(u) then
5              remove o from kDNN(u);
6              F ← F ∪ {u};
7      F* ←sort nodes of F in increasing ranks;
8      while F* is not empty do
9          p ← lowest ranked node in F*;
10         kDNN(p) ← at most k objects located at p;
11         for (v ⌢ p) ∈ SC_< do
12             for object o* ∈ kDNN(v) do
13                 add [o*, d_SC<(o*, v) + d_SC<(v, p)] to
                   kDNN(p);
14                 if ||kDNN(p)|| > k then
15                     delete from kDNN(p) the object with longest
                       distance from p;
16         remove p from F*;
```

on $SC_<$ to locate all the kDNN lists that contain o, and remove o from them. Let p be a node whose kDNN list is so updated. The vacancy made by the removal of o from kDNN(p) is re-filled by finding $p_\downarrow$ and determining the object in $p_\downarrow$ that is the top-k nearest to p. To find $p_\downarrow$, we perform a *Dijkstra* search from p on $SC_<$ going *only downhill*. That is, we follow the shortcuts in $SC_<$ in a reverse manner. By Property 2, the downhill searches from the p's can be integrated and ordered according to the p's ranks.

Algorithms 2 and 3 summarize the insert and delete procedures, respectively. We remark that with SCOB, the *Dijkstra* searches on the shortcut graph $SC_<$ required in query, object insert and object delete are generally much more efficient compared with doing *Dijkstra* searches on the original graph G.

5.2.2 Relaxing Assumptions

In this section we relax assumptions A1 and A2.

Relaxing A2: ranks are distinct. The CH algorithm determines the shortest distance $d_G(s,t)$ from a node s to a node t by performing an s-climb and a t-climb. In our discussion, we assumed that the summit node x of the shortest path $s \rightsquigarrow t$ is unique. This allows the two climbs to meet at x. If nodes are allowed to have equal ranks, then there could be more than one summit node on $s \rightsquigarrow t$. In this case, the two climbs will be disconnected, and the shortest distance cannot be determined. Figure 3(a) shows an example. In the figure, both nodes x and y are summits of the path $s \rightsquigarrow t$.

This issue can be resolved by a modification to the definition of shortcuts. Specifically, we change the second requirement of a shortcut $u \curvearrowright v$ in Defintion 2 from "$r(u) < r(v)$" to "$r(u) \leq r(v)$". We use $SC_\leq$ to denote the set of shortcuts so defined. Figure 3(d) shows such shortcuts along the path $s \rightsquigarrow t$. We make two observations: (1) Climbs based on $SC_\leq$ are slower than those based on $SC_<$. For example, the climb from s to x in Figure 3(d) takes 3 hops while that in Figure 3(a) takes 2. (2) Only one climb needs to be based on $SC_\leq$, the other can be based on $SC_<$. For example, in Figure 3(b), the s-climb uses shortcuts in $SC_<$ and the t-climb uses those in $SC_\leq$; Figure 3(c) shows the other way round. For description purpose, we call a climb that uses $SC_<$ a "*straight climb*", while a climb that uses $SC_\leq$ a "*gentle climb*".

With SCOB, one can consider the *Dijkstra* search executed for a query q as an s-climb from node q, while the search for executing an object insert/delete as a t-climb from the node that the object is located. We therefore can adjust the relative efficiency of queries vs. updates by making one climb gentle and the other one straight. Figures 5 illustrates this tradeoff.

The construction and maintenance of the SCOB index will have to be modified if we make t-climbs gentle. For example, all mentioning of $SC_<$ in Definitions 4 and 5, Property 2, Algorithm 2 (Insert) and Algorithm 3 (Delete) are replaced by $SC_\leq$. Moreover, if s-climbs are made gentle instead, then in Algorithm 1 (Query), $d_{SC_<}(q,p)$ is replaced by $d_{SC_\leq}(q,p)$. (See [1] for more details.)

Relaxing A1: G is undirected. Road networks are generally represented by directed graphs. If G is directed, then the shortcuts used in t-climbs should be based on the reversed edges in G. The shortcut graph used for performing t-climbs (either $SC_<$ or $SC_\leq$) should therefore be constructed based on the reversed version of G.

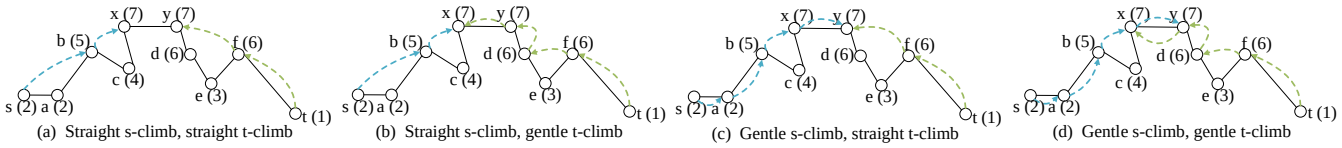
5.3 TOAIN

In this section we present TOAIN. We propose an interesting strategy of designing the ranking function $r()$. We also show how TOAIN adjusts the ranking function in search for the best query/update tradeoff, and hence achieves the best throughput.

In [15], where the CH algorithm is studied, it is hinted that structurally more important nodes in a graph should be given higher ranks. It is claimed that such rank assignments would generally make CH more efficient. A choice of a node-importance measure is *betweenness centrality*, which is defined as the number of shortest paths that pass through a given node. Computing betweenness for all the nodes, however, takes $O(|V||E|)$ time [9], which is expensive. One could consider applying a more efficient algorithm for computing approximate betweenness, such as [26, 32]. In this paper we propose an alternative way of measuring node importance (and hence to derive r()) based on the concepts of cover nodes and cover dimension [21]. There are two advantages of our cover-node-based method: First, it is efficient. Second, our experiments show that constructing the shortcut set using cover-node-based ranking is more efficient than doing so using betweenness-based ranking.

DEFINITION 6 (COVER NODE, COVER DIMENSION). *Consider a road network on a spatial map. Let us superimpose a $K \times K$ grid on it partitioning the map into K^2 cells. Given a cell C, consider its 3 × 3 and 5 × 5 neighborhoods as illustrated in Figure 4. Now, consider any node u in C and any node v outside the 5 × 5 region. The shortest path $u \rightsquigarrow v$ must cross the perimeter of the 3 × 3 region. Let (x, y) be the edge in $u \rightsquigarrow v$ that crosses the 3 × 3 boundary. We call node x a cover node of cell C. The number of cover nodes of a cell is called the cover dimension ζ of the cell.*

For example, in Figure 4, the cover nodes of cell C are $\{d, i, o\}$ and the cover dimension of the cell is 3. Intuitively, if one travels from any node within C to somewhere outside the 5 × 5 neighborhood (i.e., for a non-trivial distance), the shortest path taken must pass through at least one of the cover nodes. In [21], some interesting empirical observations about cover nodes are made. In particular, it is found in many road networks: (1) the cover dimensions of cells are very small and they vary over a very small range of values; (2) the cover dimensions of cells stay relatively the same *regardless* of the grid resolution (K). For example, there are about 264,346 road junctions in the New York road network. If we put a 50 × 50 grid on it, each cell contains about 106 nodes on average. The cover dimensions, ζ, of the 2,500 cells have an average of around 10. That means, on average, a shortest path that connects any of (an average of) 106 nodes in a cell to somewhere outside the node's 5 × 5 vicinity has to pass through one of 10 specific cover



(a) Straight s-climb, straight t-climb (b) Straight s-climb, gentle t-climb (c) Gentle s-climb, straight t-climb (d) Gentle s-climb, gentle t-climb

Figure 3: Climbing with equal ranks.

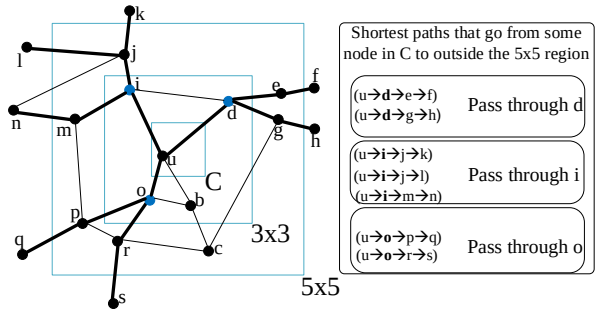


Figure 4: Cover nodes (i, d, o in this example).

nodes. This shows that the cover nodes of a cell act like a small number of gateways to the outside world for all the nodes inside the cell. Cover nodes are thus very important. Moreover, we found that if we change the grid resolution (e.g., to 25×25 and so making each cell 4 times larger), the average cover dimension remains close to 10. Since a larger cell (due to a smaller grid resolution K) contains more nodes than a smaller cell, we argue that cover nodes of larger cells (smaller K) are more important than the cover nodes of smaller cells (larger K).

With the above discussion, we design the ranking function $r()$ based on cover nodes and grid resolutions using the following procedure: First, we pick an integer h such that the range of $r()$ is $[0...h]$, and a high-resolution $K_1 \times K_1$ grid (i.e., large K_1). Each node is initially given a rank of 0. With respect to the K_1 grid, all cover nodes of all the cells are collected into a set Υ_1. The ranks of these nodes are *promoted* to 1. Then, we iterate the following steps: During the i-th iteration ($2 \leq i \leq h$), (1) lower the grid resolution by half, i.e., set $K_i = K_{i-1}/2$. (2) For each cell C in the $K_i \times K_i$ grid, find all the rank-(i-1) nodes in C and consider the shortest paths that originate from them. (3) Determine the cover nodes of C from those shortest paths. (4) The cover nodes of all the cells found in the iteration are collected in Υ_i; these nodes are promoted to rank i. Algorithm 4 gives the pseudo code of the procedure.

Algorithm 4: `ComputeRank` (G, h)

```
1  initialize r(u) to 0 for every node u ∈ G;
2  S ← V /* V is the node set of G.                    */
3  for grid level i from 1 to h do
4      impose K_i × K_i grid on G;
       /* Refer to text for the value of K_i          */
5      Υ_i ← ∅;
6      for node v ∈ S do
7          conduct Dijkstra search from v within the 5 × 5 sub-grid,
           whose central cell contains v;
8          Υ_{i,v} ← cover nodes found during the Dijkstra search;
9          Υ_i ← Υ_i ∪ Υ_{i,v};
10     update r(z) to i for every node z ∈ Υ_i;
11     S ← Υ_i;
```

In Section 5.2.2 we mentioned that TOAIN can control the tradeoff between query and update times by making either the s-climbs or the t-climbs straight and the other gentle. Another way to control the tradeoff is by adjusting h. To see this, recall that node ranks range from 0 to h. Figuratively, h is the *height of the hill* (see Figure 5 for an illustration). We see that a short hill (small h) favors a straight climb (because it takes few steps to reach the top of the hill) and disfavors a gentle climb (because the hill is flat and so there is much level-wise exploration). These two controls provide us with a wide *configuration space* of SCOB. For example, very fast query processing can be achieved by straight s-climbs and small h (see Figure 5(a)), while comparable query/update times can be achieved by using a large h (see Figures 5(c) and (d)).

5.3.1 Picking a Configuration

TOAIN determines a SCOB configuration based on the objective of maximizing throughput. From Section 4, Equations 3 and 8, the maximum throughput attainable (λ_q^*) is dependent on a number of variables, among which (t_q, V_q) and (t_u, V_u) are the (processing time average, variance) of queries and updates, respectively. The relative values of these variables can be controlled by adjusting the SCOB configuration. Given an application environment (abstracted by e.g., a road network G, a set of moving objects $\mathcal{M}$, a query/update arrival model or a real workload of them, the value of k, etc.), TOAIN constructs the SCOB index under various configurations and evaluates (t_q, V_q, t_u, V_u) for each configuration by performing a simulation analysis. These values are then substituted into Equations 3 and 8 to determine the maximum throughputs of the SCOB configurations[2]. Finally, the configuration that gives the highest throughput will be picked for deployment.

5.3.2 Complexity Analysis

We end this section with a summary of a complexity analysis on TOAIN, using the maximum cover dimension ζ^*. Due to space limitations, readers are referred to [1] for the proofs.

LEMMA 6 (COMPUTING RANKS). `ComputeRank` *(Algorithm 4) costs* $\mathcal{O}(h\zeta^*|V|\log|V|)$ *time.*

LEMMA 7 (CREATING SHORTCUTS). *Creating the shortcut set* $SC_<$ *(or* $SC_\leq$*) costs* $\mathcal{O}(h\zeta^*|V|\log|V|)$ *time.*

LEMMA 8 (SIZE OF SHORTCUT SET). *Given a grid hierarchy, the number of shortcuts created in* $SC_<$ *(or* $SC_\leq$*) is* $\mathcal{O}(\zeta^*|V|)$*, regardless of* h*.*

As shown in [21], the maximum cover dimension ζ^* is small. Therefore, the number of shortcuts created is approximately linear to $|V|$, i.e., $\mathcal{O}(\zeta^*|V|) \approx \mathcal{O}(|V|)$. We also note that the height h is small in real datasets. For example, in all of our tested datasets,

[2] We remark that throughput can also be estimated using simulation. However, that would require multiple simulation runs, each testing one specific λ_q, in order to find the maximum throughput (λ_q^*) that does not cause system overloading. Instead, TOAIN applies Equations 3 and 8 so that λ_q^* can be directly estimated.

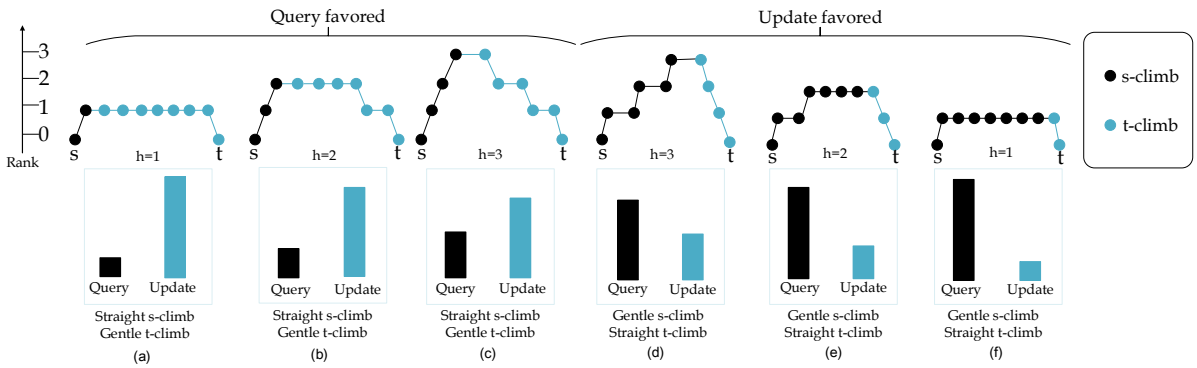


Figure 5: Adjusting the relative costs of queries and updates.

$h \leq 12$. Consequently, the preprocessing, i.e., computing ranks and creating shortcuts, can be done in close to $\mathcal{O}(|V| \log |V|)$ time.

6. EXPERIMENT

In this section we present experiment results evaluating the performances of various algorithms under a wide spectrum of application settings. Our objectives are twofold. First, existing algorithms were typically evaluated on their query processing efficiency. As we have argued, from the perspective of service providers, throughput is a more relevant measure. We thus re-evaluate existing algorithms based on their abilities in supporting large throughput systems. Second, our algorithm TOAIN is designed to adapt to different application characteristics, such as system model, query/update arrival rates, and network scale. We thus evaluate TOAIN's adaptability to a wide range of scenarios as compared with existing techniques. In Section 6.1 we first describe our experiment setups. These include the real/synthetic datasets used, the query/update generators, algorithm implementations, and performance measurements. In Section 6.2 we discuss two case studies analyzing the algorithms under two illustrative scenarios. Finally, in Section 6.3, we show the performances of the algorithms under a wide range of application settings to evaluate their adaptabilities.

6.1 Setup

Data. We conduct our experiments on three real road networks, namely, Beijing (BJ), North West America (NW), and New York City (NY). Table 2 shows the characteristics of these networks. For BJ, we obtained from UCAR [2], which is a popular taxi hailing service in Beijing, trajectories of 3,000 taxis. These trajectories are given as a single stream of location updates. There are 8.74 million updates in the stream. For NW, we obtained a real dataset of points of interests (POIs) in NW, such as restaurants, hospitals, and schools. These POIs help us model location-based games in which the occurrences of objects are usually clustered at POIs. Moreover, to evaluate the impact of a road network's size on the algorithms' performances, we extract sub-networks from BJ. Specifically, we start with a *center node* in BJ that is located closest to the geographical center of Beijing city. We gradually expand the network from the center node until certain number of nodes are collected in the expansion. These sub-networks are named BJz in Table 2, where z indicates the network size in multiples of 10,000 nodes.

For each road network, we model two applications: taxi hailing (i.e., the BUA+QF model) and location-based game (i.e., the RUA+FCFS model). We refer to a scenario by X-Y, where X is a road network and Y is either BUA or RUA indicating the update model. For each scenario, we need to generate (1) queries and (2) an object set and object updates. Queries are generated as a Poisson process at an arrival rate of λ_q. Given a size m, an object set $\mathcal{M}$ is generated by randomly selecting m nodes in the network at each of which an object is created and placed. With the BUA model, we divide time into periods, each of T seconds long. At the start of each period, an object o located at a node u updates its location to another node v, where v is randomly picked from u's immediate neighbors. With the RUA model, updates are generated as another Poisson process with arrival rate λ_u. Each update is either an insert or a delete with equal probability. For an insert, a new object o is created and a node is randomly pick at which o is placed; For a delete, an object is randomly picked and removed.

Update generations for the scenarios BJ-BUA and NW-RUA are exceptions to the above rules because we have real data to help us better generate the updates. Specifically, for BJ-BUA, updates are given by the real UCAR trajectory data. For NW-RUA, an *insert* update will only place a newly created object at one of the POIs.

We vary other parameters to model various application characteristics. These parameters are summarized in Table 3. In particular, we define *object density* $\theta = m/|V|$ to control the scale of the object set $\mathcal{M}$ normalized against the road network size. We also vary λ_u/m as the update rate of objects. Default values of the parameters are highlighted in boldface.

Table 2: Road networks.

Symbol	Network	#Edges	#Nodes	Additional info.
BJ	Beijing	2,690,296	1,285,215	3,000 taxi trajectories
NW	US North West	2,840,208	1,207,945	13,132 POIs
NY	New York City	733,846	264,346	
BJ8	Beijing sub-networks	170,236	80,000	
BJ16		341,352	160,000	
BJ32		682,152	320,000	
BJ64		1,358,564	640,000	
BJ128		2,677,984	1,280,000	

Table 3: Parameters (default values in bold).

Parameter	Description	Values
k	Number of NNs returned	1, 10, **20**, 30, 40
R_q^*	Response time QoS (ms)	0.2, 0.4, **0.8**, 1.6, 3.2, 6.4
θ	Object density ($m/\|V\|$)	0.1, 0.01, **0.001**
T	Update periodicity (secs)	1, 2, **4**, 8, 16
λ_u/m	Object update rate (s^{-1})	1/16, 1/8, **1/4**, 1/2, 1, 2, 4, 8, 16

Implementation. We conduct experiments on TOAIN and four other representative algorithms, namely, *Dijkstra* [14], ROAD [20], G-tree [34], and V-tree [30]. These algorithms are described in

Section 2. We use the codes provided by [8] for the implementations of ROAD and G-tree. G-tree in [8] is an optimized version of the original algorithm, and we refer to it as G-tree* in this section. The V-tree implementation is based on the source code provided by the authors of V-tree. We implemented *Dijkstra* and TOAIN using C++. Experiments are conducted on an Intel i7-4870HQ 2.5GHz CPU with 16GB RAM running Mac OS X (10.10.4).

Measurements. We measure the maximum average throughput (λ_q^*) for the system under various road networks, models, and parameter settings. In each case, λ_q^* is estimated as follows. We run the system for 200 seconds with a certain query arrival rate λ_q. We gradually increase λ_q and repeat the run until one of two conditions is met: (Query QoS violation): If the average query response time exceeds the QoS requirement R_q^*. (Overloading): If the system is overloaded. We distinguish two cases. (1) Under the BUA model, each object generates an update every T seconds. If an update cannot be installed in the system within T seconds, the update loses its value (because another update of the same object would have arrived). We consider the system overloaded. (2) Under the RUA model, we compute the total servicing time of all queries and updates generated in the 200s period. If the amount exceeds 200s, we consider the system overloaded. The largest λ_q in a run that does not trigger any of the above conditions is registered. This process is repeated 20 times. The average of the largest λ_q's obtained is taken as a measurement of λ_q^*.

6.2 Case Studies

In this section we report the results of two illustrative cases.

Case Study 1. [*Location-based game with high update rates*] We model a location-based game setting such as *Pokémon GO*. We use the Beijing road network (BJ). We set m = 5,000 objects (*Pokémon*s), k = 9 (the *nearby* search in *Pokémon GO* shows 9 nearest *Pokémon*s), a high update arrival rate at λ_u = 100,000 per second, R_q^* = 0.8ms, and the system model is RUA+FCFS.

Table 4 shows the results of the five algorithms. For each algorithm, we show the query time (t_q), update time (t_u) and the maximum average throughput (λ_q^*). First, let us consider the four existing methods. As discussed in Section 2, *Dijkstra* does not build an index, while the others use elaborate indexes. From Table 4, we see that in terms of query time, the index-based methods (i.e., V-tree, G-tree* and ROAD) are faster. Update efficiency is in general higher if more elaborate index is used. In particular, the update cost of *Dijkstra* is negligible. In this update-heavy case, the update cost plays an important role in determining the throughput. As shown in Table 4, the basic *Dijkstra* algorithm gives a higher throughput than G-tree* and V-tree because of a very small update cost. At an update arrival rate of λ_u = 100,000 per second, G-tree* spends 8.3μs $\times$ 100,000 = 0.83s, or 83% of its time processing updates, leaving only 17% of its capacity in answering queries. This leads to a small throughput. Interestingly, although V-tree gives a much better t_q than *Dijkstra*, it takes 10μs $\times$ 100,000 = 1s to process all the updates generated in a second. This results in 0 throughput.

It is interesting to see that TOAIN is able to achieve a higher throughput than others. In particular, TOAIN's throughput is about 8 times higher than that of G-tree*. Recall that TOAIN intelligently chooses the best performing SCOB configuration. In this case study, TOAIN considers a total of 24 SCOB configurations. Four examples of such configurations are shown in Table 5. To understand these configurations, let us summarize our discussion in Section 5.2: (1) A query involves an s-climb, while an update involves a t-climb. (2) A straight climb is faster than a gentle climb. (3) A shorter hill (smaller h) favors straight climbs, while a higher hill favors gentle climbs. (4) TOAIN tunes query/update costs by adjusting h and the steepness of the climbs. From Table 5, we see that TOAIN picks the configuration which employs a 6-level SCOB index with gentle s-climb and straight t-climb. In this update-heavy case, straight t-climb gives small update costs (as evidenced by the relatively small t_u = 5.8μs of Configuration 1 in the table). Increasing the hill height to 8 (Configuration 2) disfavors straight-climbs and as a result, t_u increases to 30μs in Configuration 2. The update cost is too high and that leaves the system with no capacity to handle queries, resulting in a 0 throughput. Shortening the hill to h = 4 (Configuration 3) favors straight-climbs. This makes update faster (3.2μs) at the expense of a larger query time (105μs). The overall result is a smaller throughput (5,540). Finally, in Configuration 4, t-climbs are gentle. This makes updates very slow (≈4,000μs). The system cannot handle the heavy stream of updates leading to 0 throughput.

Table 4: Algorithms' performance (Case Study 1).

Algo.	Query time t_q (μs)	Update time t_u (μs)	Throughput λ_q^*
Dijkstra	338.1	≈ 0	2,049
ROAD	135.6	2.9	4,650
G-tree*	140.4	8.3	936
V-tree	200.9	10.0	0
TOAIN	44.2	5.8	7,743

Table 5: Example configurations of TOAIN (Case Study 1).

Configuration	t_q (μs)	t_u (μs)	λ_q^*	h	s-climb	t-climb
1 (selected)	**44.2**	**5.8**	**7,743**	6	gentle	straight
2	34.6	30.0	0	8	gentle	straight
3	105.8	3.2	5,540	4	gentle	straight
4	14.0	3,948.9	0	10	straight	gentle

Case Study 2. [*Taxi hailing*] We model a taxi hailing service. We use the New York road network (NY). We set m = 15,000 (taxis), k = 1 (only the closest taxi is located), T = 4s, R_q^* = 0.8ms, and the system model is BUA+QF.

Table 6 shows the result of the algorithms. (The row labeled "Config. A" refers to one particular SCOB configuration considered but not picked by TOAIN. We will discuss more on that shortly.) Comparing against the first case study, in our second case study, we have more objects (15,000 vs. 5,000). However, each object updates its location only once in every T = 4 seconds. This gives an update arrival rate of λ_u = 3,750 per second, which is a small rate. The case is thus update-light. In such a setting, one would expect G-tree* and V-tree, which favor fast query processing, to excel. Contrary to our intuition, Table 6 shows that G-tree* and V-tree are dominated by *Dijkstra* and ROAD on all t_q, t_u, and λ_q^*. In particular, *Dijkstra* is the best performing algorithm among the four existing methods.

Table 6: Algorithms' performance (Case Study 2).

Algo.	t_q (μs)	t_u (μs)	λ_q^*
Dijkstra	4.3	≈ 0	222,655
ROAD	8.3	1.0	112,100
G-tree*	28.7	2.1	30,305
V-tree	22.8	4.9	36,883
TOAIN	0.8	44.1	1,027,712
Config. A	5.6	0.9	176,141

Dijkstra has a very small update cost because it does not build elaborate indices like ROAD, G-tree* or V-tree. The reason why *Dijkstra* also gives a very small query time is because the object density in this case study is relatively large — there are relatively many objects (15K) over a relatively small network (264K nodes).

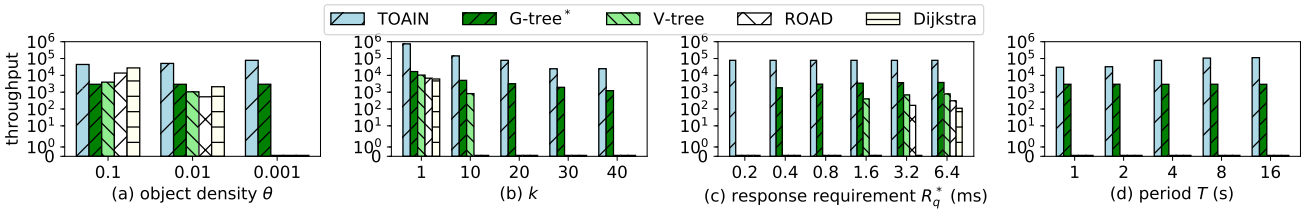


Figure 8: Throughput (NY, BUA+QF).

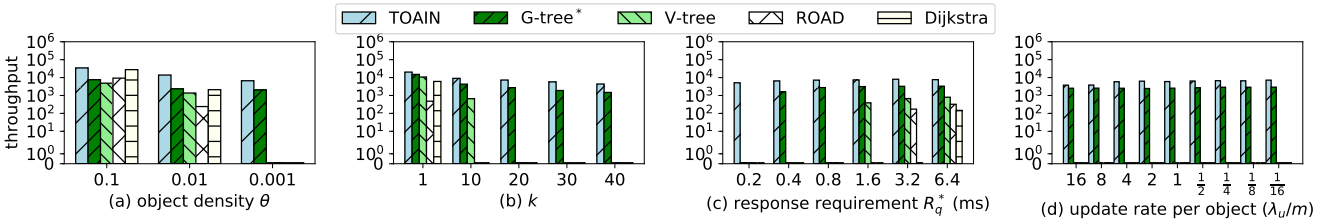


Figure 9: Throughput (NY, RUA+FCFS).

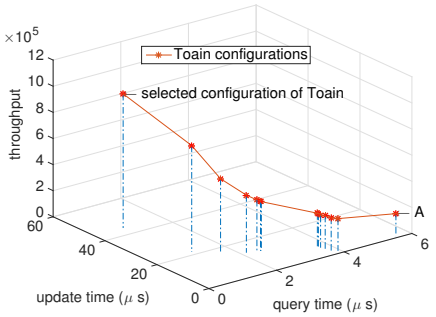


Figure 7: SCOB configurations (Case Study 2).

For example, if objects were uniformly distributed across the network nodes, then given a query, we expect that, on average, the closest object can be located within a neighborhood of 264K/15K = 17.6 nodes. The very localized search gives *Dijkstra* a very fast query time. G-tree* and V-tree, on the other hand, need to inspect sophisticated index structures in answering queries. The cost outweighs the benefit of using the index, giving them a relatively long query time. For TOAIN, although its update cost is high, it has a very small query time. The result is a higher throughput than the other algorithms. For example, TOAIN's throughput is 27 times larger than V-tree, 34 times larger than G-tree*.

In this case study, TOAIN considers 14 SCOB configurations. Figure 7 gives a 3-d illustration of them displaying their (t_q, t_u, λ_q^*) values. As an example, the row marked "Config. A" in Table 6 shows the values of the configuration marked "A" in Figure 7. From the data, we make a few interesting observations. (1) The SCOB index is very flexible. It covers a wide spectrum of query/update tradeoff. (2) TOAIN is successful in picking the configuration with the highest throughput. (3) From Table 6, we see that the performance with configuration A, in terms of t_q and t_u, straightly dominates those of ROAD, G-tree*, and V-tree. Yet, TOAIN opts for another configuration that is not clearly dominating the others in both (t_q, t_u), but is the best performer in throughput. This shows that a straightforward comparisons of algorithms based on t_q and t_u is inadequate in identifying the highest-throughput solution.

6.3 Adaptability

To evaluate the adaptability of the algorithms, we evaluate them over a wide spectrum of system settings and environments. Specifically, for each road network and system model, we vary the parameters shown in Table 3 over the ranges of values shown in the table. For example, with the NY network, we evaluated the algorithms over the combinations of (k, R_q^*, θ, T) under the BUA+QF model, and other combinations of (k, R_q^*, θ, λ_u/m) under the RUA+FCFS model. Due to space limitations, we summarize the representative results in this section. Our observations are generally applicable to other cases, and some of these results can be found in [1].

In Figure 8 (Figure 9), we show the algorithms' throughputs with the NY road network under the BUA+QF (RUA+FCFS) model. We vary 4 parameters, one at a time. When we vary a parameter, other parameters assume their default values shown in Table 3. We display, in total, the results of 42 cases in 8 sub-figures. Note that: (1) If an algorithm gives 0 throughput, its corresponding bar is not displayed in the figures. (2) The y-axis (throughput) is in log scale.

From the figures, we see that TOAIN is highly adaptable to changing system settings and environments. It significantly outperforms the other algorithms in throughput for all the cases. Let us say that an algorithm is "*applicable*" in a certain case if it produces a non-zero throughput for the case. We see that TOAIN is much more applicable than others. In particular, TOAIN is applicable to all the cases. Moreover, there are two cases (at R_q^* = 0.2 in Figures 8(c) and 9(c)) in which the response time requirement is so stringent that only TOAIN can adjust to the most query-efficient SCOB index and it is the only applicable solution.

In general, a smaller object density θ and a larger k require the algorithms to explore a larger neighborhood of a query q in order to locate all kNN objects of q. Also, a smaller R_q^* means a more stringent query QoS requirement. These situations thus require more sophisticated index structures to facilitate fast query processing and to sustain a high throughput. This explains why the throughputs of *Dijkstra* and ROAD (which use no or simple indices) drop drastically as θ decreases, k increases, and R_q^* decreases (see Figures 8(a)(b)(c) and 9(a)(b)(c)).

To further illustrate TOAIN's adaptability, we investigate the 450 cases (all 450 combinations of (k, R_q^*, θ, T) in Table 3) of (NY, BUA+QF) experiment. We found that TOAIN considered 14

Table 7: Percentage of cases in which a configuration is picked by TOAIN (NY, BUA+QF).

Configurations	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Frequency	0%	6.7%	13.3%	16.2%	21.5%	9.3%	5.1%	0.8%	2.3%	6.5%	3.8%	2.6%	5.1%	6.6%

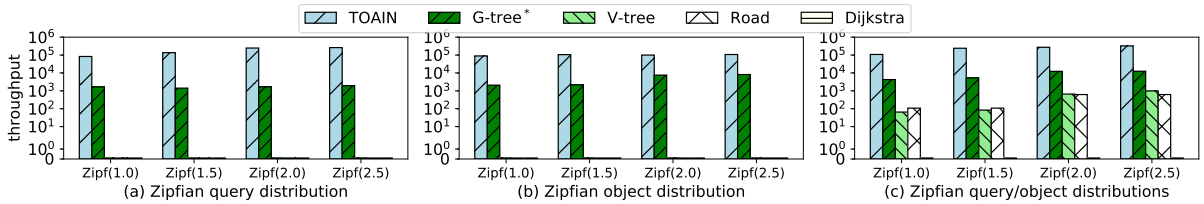


Figure 12: Throughput under Zipfian object/query distributions (NY, BUA+QF).

SCOB configurations corresponding to h = 1 to 7 combined with straight/gentle s-climb and t-climb. Table 7 shows the relative frequencies that these 14 configurations are picked over the 450 cases. We see that, except for 1 configuration, the selection of the configurations is very spread-out. This again highlights the importance of an adaptive and configurable algorithm. Figures 10(a) and (b) show the algorithms' throughputs for the networks NY, NW and BJ under the models BUA+QF and RUA+FCFS, respectively. Figure 11 shows the results for the BUA+QF model as we vary the size of the BJ network from $|V|$ = 80,000 nodes (BJ8) to $|V|$ = 128,000 nodes (BJ128) (see Table 2). In these experiments, parameters are set to their default values (Table 3), except that for Figure 11, we keep the number of objects m = 5,000. From the results, we draw similar conclusions as those mentioned in the previous experiments: TOAIN outperforms the other algorithms and it is highly adaptable to different road networks.

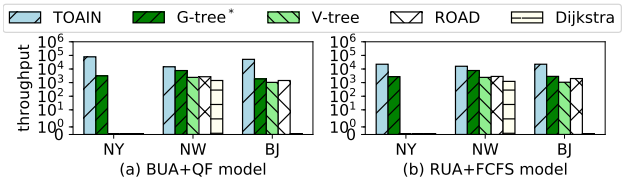


Figure 10: Throughput on different road networks.

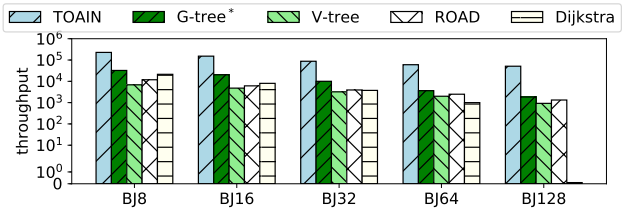


Figure 11: Throughput vs. network size (BJ, BUA+QF).

In our previous experiments, objects and queries are distributed to the network nodes uniformly. Our last set of experiments study the effect of skewed distributions of objects and queries. From [7], we obtain yellow-cab pickup locations in NY. We compute a *popularity rank* for each node in NY based on the nodes' pickup frequencies. (The most popular node is ranked 1st.) We model object/query locations as Zipfian distributions. Specifically, the node with the r-th popularity rank is given a probability of $f(r;\alpha,|V|) = (1/r^{\alpha})/(\sum_{n=1}^{|V|} 1/n^{\alpha})$ for being an object or query location, where α is a parameter that controls the skewness of the distribution (a larger α gives a skewer distribution). Figure 12 shows the algorithms' performances under the BUA+QF model for (a) Zipfian query, uniform object distributions, (b) uniform query, Zipfian object distributions, and (c) Zipfian query and object distributions. From the figure, we again see that TOAIN is applicable to all the scenarios, while other algorithms register 0 throughputs in many cases. TOAIN is thus highly adaptable and it significantly outperforms other algorithms. The experiment is repeated for the RUA+FCFS model for which we draw similar conclusions.

Finally, the indexing process of TOAIN takes about 2 minutes for a small network like NY to 20 minutes for a large network like NW. The amount of memory consumption depends on the network size as well as k. (A larger k implies bigger kDNN lists.) As an example, the memory consumption for the NY network with k = 10 is 228MB. The largest case is the NW network with k = 50, which requires 1.13GB of memory.

7. CONCLUSION

In this paper we study the problem of optimizing the throughput of processing kNN queries on road networks. We consider different query/update arrival models and queuing models that describe the operations of location-based services. We propose a mathematical model that expresses the maximum throughput in terms of a number of key variables, subject to certain QoS constraints. We propose TOAIN, which employs a highly tunable index structure called SCOB to achieve throughput maximization. The salient feature of SCOB is that it is highly auto-configurable, making it possible to adapt to different environments and settings that require delicate adjustments in query/update tradeoffs. We evaluate the performance of TOAIN, comparing it against other state-of-the-art solutions. We conduct experiments and simulations on real road networks and real/synthetic workloads. Our results show that TOAIN outperforms the competitors over a wide spectrum of cases.

Acknowledgement

Ben Kao and Siqiang Luo were supported by the Hong Kong Research Grants Council grant GRF 17254016. Guoliang Li was supported by 973 Program of China (2015CB358700) and NSF of China (61632016,61472198,61521002,61661166012). Reynold Cheng, Jiafeng Hu, and Yudian Zheng were supported by the Hong Kong Research Grants Council (RGC Projects HKU 17229116 and 17205115) and the University of Hong Kong (Projects 104004572, 102009508,104004129). We would like to thank the reviewers for their insightful comments.

8. REFERENCES

[1] https://sites.google.com/view/roadknn-throughput-tr.

[2] https://www.10101111.com/.

[3] https://www.forbes.com/sites/insertcoin/2016/07/30/the-new-pokemon-go-update-has-killed-nearby-tracking-completely/#1ecb56341787.

[4] https://www.lyft.com/.

[5] https://www.uber.com/.

[6] http://tech.sina.com.cn/i/2016-06-21/doc-ifxtfrrc4037422.shtml.

[7] http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[8] T. Abeywickrama, M. A. Cheema, and D. Taniar. k-nearest neighbors on road networks: a journey in experimentation and in-memory implementation. *PVLDB*, 9(6):492–503, 2016.

[9] U. Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[10] 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.

[11] H.-J. Cho, S. J. Kwon, and T.-S. Chung. A safe exit algorithm for continuous nearest neighbor monitoring in road networks. *Mobile Information Systems*, 9(1):37–53, 2013.

[12] J. W. Cohen. *The single server queue*, volume 8. Elsevier, 2012.

[13] U. Demiryurek, F. Banaei-Kashani, and C. Shahabi. Efficient continuous nearest neighbor query in spatial networks using euclidean restriction. In *SSTD*, pages 25–43, 2009.

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

[15] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Experimental Algorithms*, pages 319–333. Springer, 2008.

[16] B. Kao, K.-y. Lam, and B. Adelberg. Updates and view maintenance. *Real-Time Database Systems*, pages 185–202, 2002.

[17] M. Kolahdouzan and C. Shahabi. Voronoi-based k nearest neighbor search for spatial network databases. In *VLDB*, pages 840–851, 2004.

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

[19] K. C. Lee, W.-C. Lee, and B. Zheng. Fast object search on road networks. In *EDBT*, pages 1018–1029, 2009.

[20] K. C. 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.

[21] S. Luo, R. Cheng, X. Xiao, B. Kao, S. Zhou, and J. Hu. Fast matching of detour routes and service areas. Technical Report TR-2016-03, The University of Hong Kong, 2016. http://www.cs.hku.hk/research/techreps/document/TR-2016-03.pdf.

[22] S. Luo, Y. Luo, S. Zhou, G. Cong, J. Guan, and Z. Yong. Distributed spatial keyword querying on road networks. In *EDBT*, pages 235–246, 2014.

[23] K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *VLDB*, pages 43–54, 2006.

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

[25] H. Qu and A. Labrinidis. Preference-aware query and update scheduling in web-databases. In *ICDE*, pages 356–365, 2007.

[26] M. Riondato and E. M. Kornaropoulos. Fast approximation of betweenness centrality through sampling. *DMKD*, 30(2):438–475, 2016.

[27] H. Samet, J. Sankaranarayanan, and H. Alborzi. Scalable network distance browsing in spatial databases. In *SIGMOD*, pages 43–54, 2008.

[28] J. Sankaranarayanan, H. Alborzi, and H. Samet. Efficient query processing on spatial networks. In *GIS*, pages 200–209, 2005.

[29] C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. *GeoInformatica*, 7(3):255–273, 2003.

[30] B. Shen, Y. Zhao, G. Li, Q. Y. Zheng, Weimin, B. Yuan, and Y. Rao. V-tree: Efficient knn search on moving objects with road-network constraints. In *ICDE*, pages 871–882, 2016.

[31] S. Wang, X. Xiao, Y. Yang, and W. Lin. Effective indexing for approximate constrained shortest path queries on large road networks. *PVLDB*, 10(2):61–72, 2016.

[32] Y. Yoshida. Almost linear-time algorithms for adaptive betweenness centrality using hypergraph sketches. In *KDD*, pages 1416–1425, 2014.

[33] B. Zheng, K. Zheng, X. Xiao, H. Su, H. Yin, X. Zhou, and G. Li. Keyword-aware continuous knn query on road networks. In *ICDE*, pages 871–882, 2016.

[34] R. Zhong, G. Li, K.-L. Tan, L. Zhou, and Z. Gong. G-tree: An efficient and scalable index for spatial search on road networks. *TKDE*, 27(8):2175–2189, 2015.

[35] 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*, pages 857–868, 2013.