TASK: An Efficient Framework for Instant Error-tolerant Spatial Keyword Queries on Road Networks

Chengyang Luo[♮1], Qing Liu[♮2], Yunjun Gao[♮3], Lu Chen[♮4], Ziheng Wei[♯5], Congcong Ge[♯6]
[♮]Zhejiang University, [♯]Huawei Cloud Computing Technologies Co., Ltd
{[1]luocy1017, [3]gaoyj, [4]luchen}@zju.edu.cn, [2]lqzju2010@gmail.com, {[5]ziheng.wei, [6]gecongcong1}@huawei.com

ABSTRACT

Instant spatial keyword queries return the results as soon as users type in some characters instead of a complete keyword, which allow users to query the geo-textual data in a *type-as-you-search* manner. However, the existing methods of instant spatial keyword queries suffer from several limitations. For example, the existing methods do not consider the typographical errors of input keywords, and cannot be applied to the road networks. To overcome these limitations, in this paper, we propose a new query type, i.e., instant error-tolerant spatial keyword queries on road networks. To answer the queries efficiently, we present a framework, termed as TASK, which consists of index component, query component, and update component. In the index component, we design a novel index called reverse 2-hop label based trie, which seamlessly integrates spatial and textual information for each vertex of the road network. Based on our proposed index, we devise efficient algorithms to progressively return and update the query results in the query component and update component, respectively. Finally, we conduct extensive experiments on real-world road networks to evaluate the performance of our presented TASK. Empirical results show that our proposed index and algorithms are up to 1-2 orders of magnitude faster than the baseline.

PVLDB Reference Format:
Chengyang Luo, Qing Liu, Yunjun Gao, Lu Chen, Ziheng Wei, and Congcong Ge. TASK: An Efficient Framework for Instant Error-tolerant Spatial Keyword Queries on Road Networks. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Geo-textual objects associated with both geographical and textual information are ubiquitous in daily life, such as restaurants, hotels, shopping malls, etc. In the literature, many types of spatial keyword queries have been studied [7, 11], e.g., top-k spatial keyword queries [12, 34], reverse spatial keyword queries [16, 27], why-not spatial keyword queries [6, 45], continuous spatial keyword queries [14, 36], to name but a few. In this paper, we focus on instant spatial keyword queries [19, 32, 47].

Instant queries, a.k.a., *search-as-you-type* or *type-ahead search*, return query results on-the-fly when users type in a query keyword character-by-character [3, 21–23]. They allow users to browse the results during typing characters, which can immensely improve user experience. [19, 32, 47] investigate the instant queries for spatial databases (in this paper, we refer to this type of instant queries as instant spatial keyword queries). Specifically, with every character of keyword being typed in, the instant spatial keyword queries return geo-textual objects that are relevant to the query inputs in terms of text and location. Sometimes, typing a complete keyword is cumbersome and also susceptible to errors. The instant spatial keyword queries save users from typing in complete keywords and thus are helpful for users in real-world applications. As an example, assume that a user (i.e., the red circle in Figure 1) wants to go to a restaurant called "Mama seafood" for dinner, and uses the location based service (LBS) to find the location. As shown in Figure 1, as soon as the user types in "Ma", the queries can return the results containing "Mama seafood" for her/him.

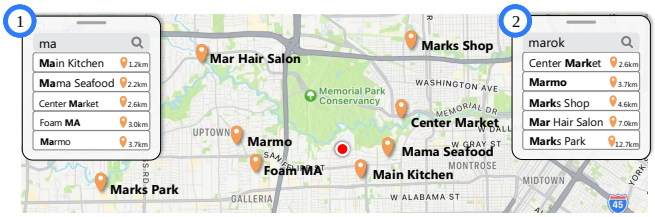


Figure 1: A motivating example

However, the state-of-the-art methods of instant spatial keyword queries [19, 32, 47] suffer from several limitations.

- The existing methods are mainly designed for the Euclidean space. For example, the bound-materialized trie proposed in [32] employs grid to index the space. Both the filtering-effective hybrid index [19] and prefix-region tree [47] leverage R-tree for the space indexing. All those techniques cannot be applied to road networks since they use different metrics to compute the distance between two objects. In real applications, it is straightforward and more practical to query the geo-textual data on road networks [1, 17, 20, 29, 31, 42]. Hence, it is necessary to develop techniques for instant spatial keyword queries on road networks.
- The existing methods do not consider the typographical errors of input keywords. Sometimes, typing accurately is a tedious task, and the users' inputs tend to contain typographical errors, especially for mobile devices. Consequently, it is critical for the instant spatial keyword queries to tolerate typos [30, 33, 35, 37, 48], which can help users query in a friendly way. Thus, it motivates us to consider the error tolerance for the instant spatial keyword queries.
- The traditional instant spatial keyword queries mostly focus on the cases where users type in queries character-by-character. Nevertheless, some common yet important cases

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Table 1: Taxonomy of representative related work and our work

Category	Index	Error tolerant	Multiple keywords	Search as you type	Character deletion	Non-tail operations	Road network
Instant spatial keyword queries	Bound-Materialized Trie [32]	✘	✘	✔	✘	✘	✘
	Filtering-effective hybrid index (FEH) [19]	✘	✘	✔	✘	✘	✘
	Prefix-region tree (PR-Tree) [47]	✘	✔	✔	✘	✘	✘
Spatial keywords queries over road networks	Map B-tree and inverted file [31]	✘	✘	✘	✘	✘	✔
	Compact tree [29]	✘	✘	✘	✘	✘	✔
	LB Index and KT Index [20]	✘	✔	✘	✘	✘	✔
	Keyword separated index (K-SPIN) [1]	✘	✔	✘	✘	✘	✔
Instant spatial keywords queries over road networks	**Reverse 2-hop label based trie (Our work)**	✔	✔	✔	✔	✔	✔

are ignored. For instance, users may (i) delete characters during the query, and (ii) add/delete characters anywhere in the query. Considering these cases can greatly enrich the instant spatial keyword queries.

To overcome these limitations, in this paper, we study a new problem, i.e., instant error-tolerant spatial keyword queries on road networks. Specifically, given a road network including geo-textual objects, a query location, and a query string that may have typographical errors, the instant error-tolerant spatial keyword queries return top-k geo-textual objects that are the most relevant to query location and query string. When the query string updates, e.g., the users input new characters to the query string or delete some characters from the query string, the queries should update the top-k geo-textual objects instantly. Our studied problem has wide applications in LBS. For example, in Figure 1, assume that the user wants to query "Marks shop", and starts typing in a string "Ma". Then, five geo-textual objects containing the prefix "Ma" are quickly returned. However, the desired "Marks shop" is not in the results. Next, the user proceeds to type in the string "Marok", where a typo is contained. The queries tolerate the typo, and return new results containing "Marks shop". It is worth mentioning that the traditional instant queries only deal with the cases of typing keywords character-by-character. Our work also considers the deletion of characters, which is beneficial in real applications. As an example, if a user finds a typo in the input string, she/he can delete the typo, and input correct characters for more accurate queries.

In the literature, many indexes have been proposed to handle the instant spatial keyword queries and spatial keyword queries on road networks, as summarized in Table 1. Since those indexes do not take all the requirements of our problem into consideration, they cannot be applied to tackle our problem. To this end, we present a novel index called reverse 2-hop label based trie (R2T) to answer the instant error-tolerant spatial keyword queries on road networks. R2T consists of two parts, i.e., *reverse 2-hop label* and *trie*. Specifically, the reverse 2-hop label is based on 2-hop labeling techniques [10], which enables efficient calculation of the distance between two vertices. With the help of the reverse 2-hop label, R2T is capable of efficient distance computation during the query processing. For the textual information, we employ trie that can efficiently support characters matching. The complete trie is complex and large, which is not efficient for queries since we need to traverse it multiple times. Thus, we design a novel structure called *node array* to store partial trie for each vertex with respect to the reverse 2-hop label. We demonstrate that after getting 2-hop label, R2T can be constructed in $O(\log^2 |V|)$ time, and the size of R2T is bounded by $O(\log |V|)$, where $|V|$ is the road network size. Moreover, we have discussed the index maintenance for dynamic road networks.

Based on R2T, we devise efficient algorithms to support instant error-tolerant spatial keyword queries on road networks, including *instant query algorithm* and *instant update algorithm*. The instant query algorithm aims to return the results when users type in characters for the first time. It first traverses the complete trie to get the active nodes, which contains the candidate results, and then, it visits the R2T to progressively find the geo-textual objects that are the most relevant to the query location and query string. The instant update algorithm is to update the query results according to the updated query string. To avoid querying from the scratch, we design *query information inheritance mechanism* to make full use of previous query processing information, which can dramatically improve the query performance. Furthermore, we extend our algorithms to support multiple query strings.

Our key contributions are summarized as follows:

- We identify and formalize the problem of instant error-tolerant spatial keyword queries on road networks, which is rooted in real-world applications. To the best of our knowledge, it is the first attempt to investigate this problem.
- We design a novel index called R2T, which seamlessly integrates the spatial and textual information for each vertex of the road network to facilitate queries. Efficient algorithms are also proposed to construct and maintain R2T.
- We present efficient algorithms to answer queries using R2T, which can return the results in a progressive way. In particular, the first type of algorithms focus on how to retrieve results when users type in a query string for the first time. The second type of algorithms handle how to efficiently update results for the updated query string.
- We conduct extensive experiments on real-world road networks to demonstrate the efficiency of our proposed index and algorithms.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 formally defines the problem. Section 4 introduces the framework Task. Section 5 presents the structure, construction, and maintenance of R2T index. Section 6 proposes algorithms for queries. Experimental results are reported in Section 7. Finally, Section 8 concludes the paper.

2 RELATED WORK

Instant spatial keyword queries. Existing studies proposed different indexes to support instant spatial keyword queries [19, 32, 47]. Basu Roy and Chakrabarti [32] first introduced the instant queries to spatial database, and developed the index called materialized trie (MT). MT uses trie as the main index structure, and incorporates spatial information into the node of trie. Ji et al. [19] proposed an R-tree based method called Filtering-Effective Hybrid Indexing (FEH) for instant spatial keyword queries. FEH utilizes R-tree as the key index structure. In each R-tree node, FEH incorporates textual filters according to the geo-textual objects contained in this node. Zhong et al. [47] proposed the Prefix-region tree (PR-Tree), which considers spatial information and textual information in a balanced manner. In addition, as surveyed in [5], many indexes, such as IR-tree [26], have been proposed for spatial keyword queries. Nonetheless, all these indexes are designed for the Euclidean space, and cannot be directly used in our work.

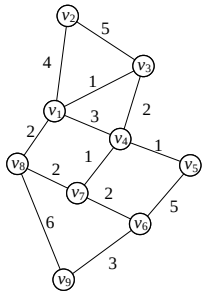
It is worth mentioning that the instant spatial keyword queries are also related to location-aware autocompletion [18, 33]. Specifically, as users type in queries, the location-aware autocompletion returns the possible completion of the queries, which can also help users reduce the amount of typing. However, the location-aware autocompletion cannot be applied to solve our problem due to three reasons. First, the location-aware autocompletion utilizes the Euclidean space, which cannot be applied to road networks. Second, the location-aware autocompletion takes spatial similarity and textual similarity separately, while our problem considers them comprehensively. Third, whenever the users input characters, the location-aware autocompletion needs to query from the scratch, which is time consuming.

Spatial keyword queries on road networks. Rocha-Junior and Nørvåg [31] first explored the spatial keyword queries on road networks. An index consisting of map tree and inverted file is proposed to answer the queries. Then, Qiao et al. [29] devised an index structure based on distance oracles and compact trees of keywords. Jiang et al. [20] presented 2-hop label backward index (LB) and keyword-lookup tree index (KT). Abeywickrama et al. [1] designed the keyword separated index (K-SPIN), which includes a set of *on-demand inverted heap* of a keyword. Besides traditional spatial keyword queries on road networks, many variants have also been studied, such as aggregate queries [9], time-aware queries [42], continuous queries [46], why-not queries [43], diversified queries [38], and reverse queries [44]. All these studies need users to type in complete queries and hence cannot be applied in our work.

3 PROBLEM FORMULATION

In this section, we formalize the problem of instant error-tolerant spatial keyword queries on road networks.

The road network is denoted as a connected undirected weighted graph $G = (V, E)$. V and E are the sets of vertices and edges of G, which represent the road junctions and segments, respectively. The weight of an edge $e = (u, v)$, denoted by $\mathrm{w}(e)$ or $\mathrm{w}(u, v)$, is a positive integer, which denotes a metric, such as distance or travel time, between u and v. A path p is a sequence of vertices $p = (v_1, v_2, ..., v_j)$, where $\forall 1 \leq i \leq j$, $(v_i, v_{i+1}) \in E$. The length of a path is the sum of weights of edges along the path. Given



(a) Road network

Vertex	Keywords
v_1	baber shop
v_2	starbucks
v_3	stadium
v_4	book store
v_5	starline club
v_6	cloud school
v_7	bus station
v_8	cinema
v_9	baby center

(b) Textual information

Figure 2: A running example

two vertices u and v of a road network G, the distance between u and v, denoted by $\mathrm{dis}(u, v)$, is the shortest path between u and v in G. For example, in Figure 2, the length of path $p = (v_5, v_6, v_7)$ is $\mathrm{w}(v_5, v_6) + \mathrm{w}(v_6, v_7) = 5 + 2 = 7$; and the distance between v_5 and v_7 is $\mathrm{dis}(v_5, v_7) = \mathrm{w}(v_5, v_4) + \mathrm{w}(v_4, v_7) = 2$.

The geo-textual objects and query location may appear on any point of the road network G. Given a geo-textual object and a query location on G, we can compute the distance between them by mapping each of them to an adjacent vertex with an offset. To make exposition simpler, we assume that the geo-textual objects and query location all appear on vertices, which follows previous studies such as [1, 20]. Under this assumption, in the road network G, a vertex v contains a set of keywords, which is denoted by $\mathrm{doc}(v)$. We use the notation $kw \in \mathrm{doc}(v)$ to denote that the vertex v includes the keyword kw. If a string str is the prefix of a keyword kw, we denote it by $str \preceq kw$. For instance, in Figure 2, "bus" $\in \mathrm{doc}(v_7)$ and "bu" $\preceq$ "bus". Next, we introduce the concepts of edit distance and prefix edit distance to measure the textual similarity.

Definition 3.1. **(Edit Distance, Prefix Edit Distance)** *Given a keyword kw, two strings str_1 and str_2.*

(1) *The edit distance between str_1 and str_2, denoted by $\mathrm{ED}(str_1, str_2)$, is the minimum number of single-character edit operations, including insertion, deletion, and substitution, needed to transform str_1 to str_2.*

(2) *The prefix edit distance between kw and str_1, denoted by $\mathrm{PED}(kw, str_1)$, is the minimum edit distance between kw's prefix and str_1, i.e., $\mathrm{PED}(kw, str_1) = \min_{\forall str' \preceq kw} \mathrm{ED}(str', str_1)$.*

For example, ED("school", "scholar") = 3, PED("school", "sco") = ED("sch", "sco") = ED("sc", "sco") = ED("scho", "sco") = 1. Based on the metric of textual similarity presented above, we formally define the error-tolerant spatial keyword queries on road networks.

Definition 3.2. **(Error-tolerant Spatial Keyword Queries on Road Networks)** *Given a road network $G = (V, E)$, a parameter k, an error threshold τ, a query $q = (q.loc, q.str)$, where $q.loc \in V$ and $q.str$ are the query location and query string, respectively. The error-tolerant spatial keyword queries on road networks return a set $\mathcal{R} \subseteq V$ such that:*

(1) $|\mathcal{R}| = k$;

(2) $\forall v \in \mathcal{R}, \exists kw \in \mathrm{doc}(v), \mathrm{PED}(kw, q.str) \leq \tau$;

(3) $\forall v \in \mathcal{R}$ *and* $\forall v' \in V - \mathcal{R}, \mathrm{score}(q, v) \leq \mathrm{score}(q, v')$, *in which* $\mathrm{score}(q, v)$ *computes the spatial and textual similarity between q and v. Specifically,*

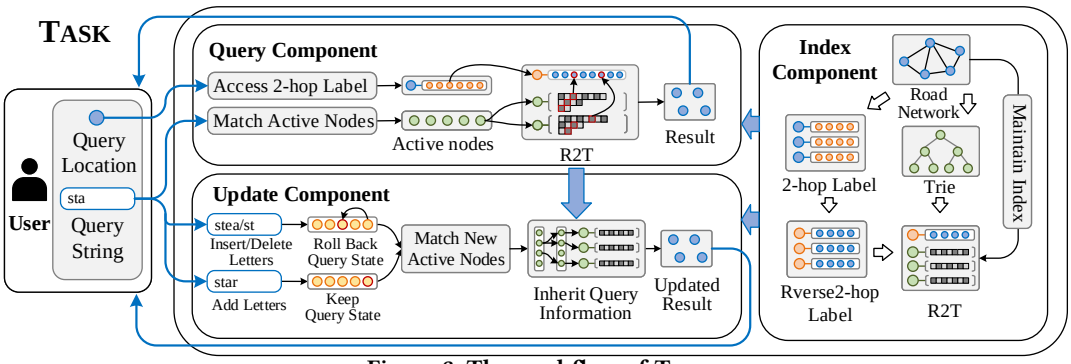


Figure 3: The workflow of TASK

$$\texttt{score}(q,v) = \alpha \cdot \frac{\texttt{dis}(q.loc, v)}{\max_{\forall u,u' \in V} \texttt{dis}(u,u')} + (1-\alpha) \cdot \frac{\min_{\forall kw \in \texttt{doc}(v)} \texttt{PED}(kw, q.str)}{\tau} \quad (1)$$

In Definition 3.2, (1) the first condition ensures that the returned results have at most k geo-textual objects; (2) the second condition gives the upper bound of the typographical error for the query string such that the keywords of each returned geo-textual object can match the query string; and (3) the third condition employs the normalized aggregated score of spatial similarity and textual similarity to rank the geo-textual objects such that the returned k geo-textual objects are optimal. Specifically, Equation 1 consists of two parts. The first part, i.e., $\frac{\texttt{dis}(q.loc,v)}{\max_{\forall u,u' \in V} \texttt{dis}(u,u')}$, employs the shortest distance to measure the spatial similarity between the query location $q.loc$ and the geo-textual object v. The closer v is to $q.loc$, the more preferable v is to $q.loc$. The second part, i.e., $\frac{\min_{\forall kw \in \texttt{doc}(v)} \texttt{PED}(kw, q.str)}{\tau}$, uses the prefix edit distance to measure the textual similarity between the query string $q.str$ and the keywords of v. The smaller prefix edit distance between $q.str$ and the keywords of v, the more similar they are. For equality, we employ the maximum shortest distance of the road network and the threshold τ to normalize these two parts, respectively. Moreover, the parameter $\alpha \in [0,1]$ in Equation 1 is introduced to adjust the importance of spatial and textual similarities. In brief, the error-tolerant spatial keyword queries on road networks tolerate the input error, and return the most relevant geo-textual objects for users. Based on this, we formalize our studied problem below.

PROBLEM 1. *Given a road network $G = (V, E)$, a parameter k, an error threshold τ, and a query $q = (q.loc, q.str)$, the problem of instant error-tolerant spatial keyword queries on road networks should* (1) *return the set $\mathcal{R}$ satisfying the three conditions in Definition 3.2; and* (2) *update $\mathcal{R}$ whenever the query string $q.str$ changes.*

In a word, the goal of Problem 1 is two-folded. First, when a user types in a string, we should return the top-k geo-textual objects that best match user's location and the typed string. Second, when the typed string changes, e.g., a new character is inserted into the string or a character is deleted from the string, we should update the results instantly. For instance, in Figure 2, assume that $q.loc = v_1$, $k = 3$, $\tau = 1$, and $\alpha = 0.5$. When a user types in a query string "st", the results $\{v_3, v_4, v_2\}$ are returned. When the user proceeds to type in "a" after "st", i.e., the current query string is "sta", the results are updated to $\{v_3, v_2, v_5\}$ immediately.

4 FRAMEWORK OVERVIEW

Problem 1 returns k geo-textual objects with the minimum score, and the typo should be not larger than τ. A naive method is to traverse the road network from $q.loc$ in a breadth first manner. For the visited vertex, we check whether the prefix edit distance of the vertex's keywords is no larger than τ. If yes, we compute its score using Equation 1, and add it to the candidate set. Finally, k vertices with the minimum score are returned. When the query string changes, we can use the above naive method to compute the results from the scratch. Obviously, the simple combination of road network traversal and text examination leads to poor performance of the naive method. First, at the worst, it may traverse the whole road network with time complexity $O(|E| + |V|)$, which is costly for online search. Second, when the query string changes, it is time consuming to query from the scratch.

Motivated by this, we propose an efficient framework, termed as TASK, to tackle the instant error-tolerant spatial keyword queries on road networks. As illustrated in Figure 3, TASK consists of an index component, a query component, and an update component. The index component is responsible for (1) constructing the R2T index for the road network and (2) maintaining R2T when the road network changes, e.g., the insertion/deletion of vertices/keywords and the change of edge's weight. The index component provides support for queries, and is the cornerstone of the framework. When a user types in a query string at the first time (i.e., when the current query string is empty), TASK uses the query component to return results to the user. Whenever the user makes change for the query string, TASK calls the update component, which employs the information of previous query processing to quickly update the results for user. The user proceeds to type in the query string until the desirable results are found. In the following two sections, we will detail these three components. Note that, due to the space limitation, some details and proofs are moved to Appendix.

5 R2T INDEX

In this section, we propose a novel index called reverse 2-hop label based trie (R2T for short) for Problem 1. We first introduce the structure of R2T, and then present the construction and maintenance algorithms of R2T.

Table 2: 2-hop label and reverse 2-hop label of the road network in Figure 2

Vertex	2-hop label	Reverse 2-hop label
v_1	$(v_1, 0)$	$(v_1, 0)$, $(v_3, 1)$, $(v_8, 2)$, $(v_4, 3)$, $(v_2, 4)$, $(v_5, 4)$, $(v_7, 4)$, $(v_6, 6)$, $(v_9, 8)$
v_2	$(v_2, 0)$, $(v_1, 4)$	$(v_2, 0)$
v_3	$(v_3, 0)$, $(v_1, 1)$, $(v_4, 2)$	$(v_3, 0)$
v_4	$(v_4, 0)$, $(v_1, 3)$	$(v_4, 0)$, $(v_5, 1)$, $(v_7, 1)$, $(v_3, 2)$, $(v_6, 3)$, $(v_8, 3)$, $(v_9, 6)$
v_5	$(v_5, 0)$, $(v_4, 1)$, $(v_1, 4)$	$(v_5, 0)$
v_6	$(v_6, 0)$, $(v_4, 3)$, $(v_1, 6)$	$(v_6, 0)$, $(v_7, 2)$, $(v_9, 3)$, $(v_8, 4)$
v_7	$(v_7, 0)$, $(v_4, 1)$, $(v_6, 2)$, $(v_1, 4)$	$(v_7, 0)$, $(v_8, 2)$
v_8	$(v_8, 0)$, $(v_1, 2)$, $(v_7, 2)$, $(v_4, 3)$, $(v_6, 4)$	$(v_8, 0)$, $(v_9, 1)$
v_9	$(v_9, 0)$, $(v_6, 3)$, $(v_4, 6)$, $(v_8, 6)$, $(v_1, 8)$	$(v_9, 0)$

5.1 R2T Structure

R2T integrates both road network information and textual information. Specifically, we employ the reverse 2-hop label and trie techniques to store spatial and textual information, respectively. First, we introduce the concept of 2-hop label [2, 8, 10, 15, 28].

Given a road network G, the 2-hop labeling technique assigns each vertex $v \in V$ a label $\mathsf{L}(v)$ containing a set of pairs $(u, \mathrm{dis}(u, v))$, i.e., $\mathsf{L}(v) = \{(u, \mathrm{dis}(u, v)) | u \in V\}$. The 2-hop labels of all vertices have the following property. Given any two vertices v and v' of the road network G, the distance between v and v' is $\mathrm{dis}(v, v') = \min_{u \in \mathsf{L}(v) \cap \mathsf{L}(v')} \mathrm{dis}(v, u) + \mathrm{dis}(v', u)$. In other words, the distance between any two vertices can be computed via an intermediate hop. For example, Table 2 shows the 2-hop labels for all vertices of the road network G in Figure 2. We can observe that $\mathsf{L}(v_2) = \{(v_2, 0), (v_1, 4)\}$ and $\mathsf{L}(v_7) = \{(v_7, 0), (v_4, 1), (v_6, 2), (v_1, 4)\}$. The vertex v_1 is the only common label vertex in $\mathsf{L}(v_2)$ and $\mathsf{L}(v_7)$. Thus, $\mathrm{dis}(v_2, v_7) = \mathrm{dis}(v_2, v_1) + \mathrm{dis}(v_1, v_7) = 4 + 4 = 8$. Based on the 2-hop label, the reverse 2-hop label is defined as follows.

Definition 5.1. **(Reverse 2-hop Label)** *Given a road network G and 2-hop labels of all vertices in G, the reverse 2-hop label of a vertex v, denoted by $\widetilde{\mathsf{L}}(v)$, consists of pairs $(u, \mathrm{dis}(u, v))$ with $(v, \mathrm{dis}(u, v)) \in \mathsf{L}(u)$, i.e., $\widetilde{\mathsf{L}}(v) = \{(u, \mathrm{dis}(u, v)) | \forall u,\ (v, \mathrm{dis}(u, v)) \in \mathsf{L}(u)\}$.*

If a vertex v is included in the 2-hop label of u, the reverse 2-hop label of v contains u. For instance, in Table 2, $(v_1, 4) \in \mathsf{L}(v_2)$ and $(v_2, 4) \in \widetilde{\mathsf{L}}(v_1)$. The reverse 2-hop label is mainly used for the distance computation, which can benefit Problem 1 a lot. The 2-hop label only supports distance computation between two vertices. Given 2-hop labels of a road network, if we want to find the k nearest neighbors of a vertex v, we should compute the common 2-hop label vertices of v and every other vertex, which is time consuming. With the help of reverse 2-hop label, we only need to compute the common 2-hop label vertices of v and v's k nearest neighbors, and thus improving the query efficiency. Note that, we assume that the pairs $(v, \mathrm{dis}(u, v))$ in both 2-hop label and reverse 2-hop label are in *ascending* order w.r.t., $\mathrm{dis}(u, v)$. Besides, when the context is clear, we use the notations $v \in \mathsf{L}(u)$ and $v \in \widetilde{\mathsf{L}}(u)$ to denote $(v, \mathrm{dis}(u, v)) \in \mathsf{L}(u)$ and $(v, \mathrm{dis}(u, v)) \in \widetilde{\mathsf{L}}(u)$ for simplicity.

Next, we introduce another technique used in R2T, i.e., trie. The trie is an ordered tree to represent a set of keywords. Its root node is an empty node. Each non-leaf node is labeled with one character, and each leaf node contains the last character of a keyword. The path from the root node to a leaf/intermediate node represents a keyword/prefix in the trie. As an example, Figure 4 depicts a trie representing the keywords set in Figure 2(b). The nodes n_4 and n_7 denote the prefix "bab" and keyword "baby", respectively.

Based on the reverse 2-hop label and trie, we formally define our proposed index as follows.

Definition 5.2. **(Reverse 2-hop Label Based Trie)** *Given a road network G and a vertex $v \in V$, a reverse 2-hop label based trie of v, denoted by $\mathsf{R2T}(v)$, consists of the reverse 2-hop label of v and the trie of v w.r.t. $\widetilde{\mathsf{L}}(v)$, i.e., $\mathsf{R2T}(v) = (\widetilde{\mathsf{L}}(v), \mathsf{T}(v))$. In particular, $\mathsf{T}(v)$ is a trie of v to represent all the keywords contained in the vertices in $\widetilde{\mathsf{L}}(v)$, i.e., $\forall v' \in \widetilde{\mathsf{L}}(v), \forall kw \in \mathsf{doc}(v')$.*

For example, in Table 2, $\widetilde{\mathsf{L}}(v_6) = \{(v_6, 0), (v_7, 2), (v_9, 3), (v_8, 4)\}$ and the keywords set w.r.t. $\widetilde{\mathsf{L}}(v_6)$ is {"cloud", "school", "bus", "station", "cinema", "baby", "center"}. Correspondingly, Figure 5 depicts $\mathsf{T}(v_6)$.

Combining Figures 4 and 5, we can find that each $\mathsf{T}(v)$ is a part of the complete trie. If we directly keep the entire $\mathsf{T}(v)$ for $\mathsf{R2T}(v)$, it has two drawbacks: (1) redundant storage and (2) repeated trie traversal. To this end, we use a node array to store $\mathsf{T}(v)$. Specifically, the node array consists of a set of $(\mathtt{ID}(n), \mathsf{B}(n))$ for every leaf node n of $\mathsf{T}(v)$, where $\mathtt{ID}(n)$ is the ID of n in the complete trie of the road network and $\mathsf{B}(n)$ consists of bitmaps of trie nodes along the path from root node to n. Note that, the bitmaps of root node, leaf nodes, and the nodes whose bitmaps are the same as leaf nodes are not stored. In $\mathsf{T}(v)$, a node n's bitmap is defined below: (1) Each bitmap has $|\widetilde{\mathsf{L}}(v)|$ bits, and each bit represents a vertex in $\widetilde{\mathsf{L}}(v)$. (2) A bit is "1" if the represented vertex has a keyword containing the prefix denoted by n; otherwise the bit is "0"; and all bits of the root node are "1". We take the node n_2' of $\mathsf{T}(v_6)$ in Figure 5 as an example. Since $|\widetilde{\mathsf{L}}(v_6)| = 4$, each bitmap of $\mathsf{T}(v_6)$'s node has 4 bits, representing v_6, v_7, v_9, and v_8, respectively. n_2' denotes the prefix "b". According to Figure 2(b), "b" $\preceq$ "bus" $\in \mathsf{doc}(v_7)$ and "b" $\preceq$ "baby" $\in \mathsf{doc}(v_9)$. Thus, the bitmap of n_2' is "0110". Figure 5 shows the bitmaps for every node of $\mathsf{T}(v_6)$.

In Figure 5, we can observe that there are a lot of "0" bits in bitmaps, which can be further compressed. Let b_1 and b_2 be the bitmaps of nodes n_1 and n_2, respectively, and n_2 be the child node of n_1. For the same bit of b_1 and b_2, if both bits are "0", we delete the corresponding "0" bit from b_2. Back to Figure 5, the bitmaps of n_2' and n_3' are "0110" and "0010", respectively. Since the first and last bits of n_2' and n_3' are all "0", we can delete the first and last bits of n_3', and the compressed bitmap of n_3' is "01". The compressed bitmap for every node of $\mathsf{T}(v_6)$ is listed below the original bitmap in Figure 5. After getting the compressed bitmaps, we can construct the $\mathsf{B}(n)$ for each leaf node of $\mathsf{T}(v)$. Take the leaf node n_5' of $\mathsf{T}(v_6)$ in Figure 5 as an example. $\mathtt{ID}(n_5') = n_7$, and the $\mathsf{B}(n_5')$ consists of the compressed bitmaps from n_2' to n_3', i.e., $\mathsf{B}(n_5') = \{0110, 01\}$. In the same way, we can construct $(\mathtt{ID}(n), \mathsf{B}(n))$ for every leaf node of $\mathsf{T}(v_6)$. Figure 6 illustrates the $\mathsf{R2T}(v_6)$ for vertex v_6. Next, we analyze the size of R2T.

Theorem 5.1. *The space complexity of R2T is $O(w \cdot |V| \cdot \log |V|)$, where w is the shortest path tree width of the road network.*

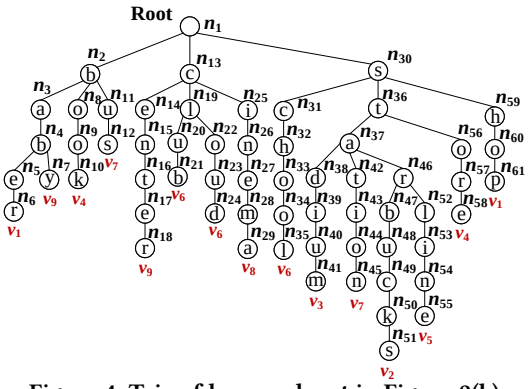


Figure 4: Trie of keywords set in Figure 2(b)

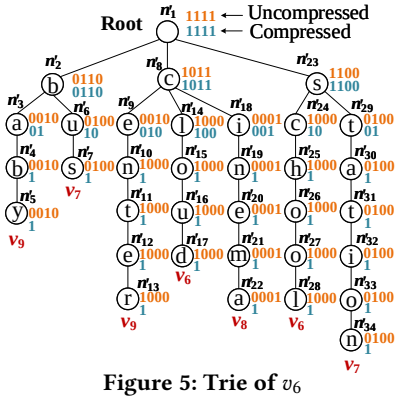


Figure 5: Trie of v_6

R2T(v_6)			
$\widetilde{\mathsf{L}}(v_6)$ = {(v_6, 0), (v_7, 2), (v_9, 3), (v_8, 4)}			
T(v_6)	Leaf Node	ID(n)	B(n)
	n'_5	n_7	{0110, 01}
	n'_7	n_{12}	{0110, 10}
	n'_{13}	n_{18}	{1011, 010}
	n'_{17}	n_{24}	{1011, 100}
	n'_{22}	n_{29}	{1011, 001}
	n'_{28}	n_{35}	{1100, 10}
	n'_{34}	n_{45}	{1100, 01}

Figure 6: R2T(v_6)

Proof. R2T consists of the reverse 2-hop label and the node array. It has been proved that the space complexity of the 2-hop label is $O(w \cdot |V| \cdot \log |V|)$ [41]. Thus, R2T also requires $O(w \cdot |V| \cdot \log |V|)$ space to store the reverse 2-hop label. After compressing the bitmaps, the space complexity of node array is $O(|V| \cdot (w \cdot \log |V| + \sum_{i=1}^{n-1} x^i \cdot \frac{w \cdot \log |V|}{x^{i-1}})) = O(w \cdot |V| \cdot \log |V|)$. Thus, the space complexity of R2T is $O(w \cdot |V| \cdot \log |V|)$. □

5.2 R2T Construction

Given a road network G, the construction of R2T(v) for every vertex $v \in V$ includes two parts, i.e., the constructions of $\widetilde{\mathsf{L}}(v)$ and $\mathsf{T}(v)$.

The construction of $\widetilde{\mathsf{L}}(v)$. First, we use existing 2-hop labeling algorithms [24] to compute the 2-hop label for every vertex. Then, we traverse the 2-hop label to get the reverse 2-hop label. Specifically, if $(v, \mathrm{dis}(u, v)) \in \mathsf{L}(u)$, we add the pair $(u, \mathrm{dis}(u, v))$ to $\widetilde{\mathsf{L}}(v)$. Finally, we sort the pairs in $\widetilde{\mathsf{L}}(v)$ in ascending order of the distances $\mathrm{dis}(u, v)$. Note that we do not store vertices without keywords since it cannot contribute to the final results.

The construction of T(v). $\mathsf{T}(v)$ is stored in the form of node array. Hence, this step focuses on how to construct the node array. First, we construct (1) a complete trie for all the keywords contained in the given road network, and (2) a trie for all the keywords contained in the vertices of $\widetilde{\mathsf{L}}(v)$ (we call it the trie of v for short). Then, we compute the bitmaps for each node of the trie of v. Specifically, we traverse the trie in a depth-first manner. For each visited trie node n, a bitmap with $|\widetilde{\mathsf{L}}(v)|$ bits is created for n. To be more specific, for each $(u, \mathrm{dis}(u, v)) \in \widetilde{\mathsf{L}}(v)$, if the keywords of u contain the prefix represented by n, the bit representing u in the bitmap is set to "1". Otherwise, the bit is set to "0". In addition, if the visited node n is a leaf node, we need to perform the following operations: (1) add n to the node array; (2) find the corresponding ID(n) in the complete trie; (3) backtrack to the root node and compress the bitmap in a bottom-up manner; and (4) add the compressed bitmap to B(n).

Theorem 5.2. *The time and space complexities of* R2T *construction are* $O(w \cdot |E| \cdot \log |V| + w^2 \cdot |V| \cdot \log^3 |V|)$ *and* $O(w^2 \cdot |V| \cdot \log^2 |V|)$, *respectively.*

Proof. R2T construction has two stages, i.e., the constructions of the reverse 2-hop label and the node array. The reverse 2-hop label construction should compute the 2-hop label, whose time complexity is $O(w \cdot |E| \cdot \log |V| + w^2 \cdot |V| \cdot \log^3 |V|)$ [41]. For a vertex $v \in V$, v's node array construction includes bitmaps calculation and compression. Calculating bitmaps takes $O(y \cdot \frac{|\mathrm{doc}(V)|}{|V|} \cdot w \cdot \log |V|) = O(w \cdot \log |V|)$ time. Here, y denotes the average word length. Compressing bitmaps takes $O(w \cdot \log |V|) \cdot O(y \cdot \frac{|\mathrm{doc}(V)|}{|V|} \cdot w \cdot \log |V|) = O(\log^2 |V| \cdot w^2)$ time. Thus, the construction of the node array for all vertices takes $O(w^2 \cdot |V| \cdot \log^2 |V|)$ time. Since $O(w^2 \cdot |V| \cdot \log^2 |V|) \ll O(w \cdot |E| \cdot \log |V| + w^2 \cdot |V| \cdot \log^3 |V|))$, the time complexity of R2T construction is $O(w \cdot |E| \cdot \log |V| + w^2 \cdot |V| \cdot \log^3 |V|)$.

The space overhead of R2T construction is the storage of uncompressed bitmaps. Thus, the space complexity of R2T construction is $O(|V| \cdot y \cdot \frac{|\mathrm{doc}(V)|}{|V|} \cdot \log |V| \cdot \log |V| \cdot w^2) = O(w^2 \cdot |V| \cdot \log^2 |V|)$. □

5.3 R2T Maintenance

In this section, we present the maintenance of R2T. In the following, we describe in detail the index maintenance process for two categories: (1) the keywords update and (2) the road network structure update.

Keywords Update. The first category is the keywords update, which includes inserting/deleting a keyword into/from doc(v) of v. Recall that R2T(v) consists of $\widetilde{\mathsf{L}}(v)$ and $\mathsf{T}(v)$, where only $\mathsf{T}(v)$ stores the keyword information. Therefore, inserting or deleting keywords only affects $\mathsf{T}(v)$. Specifically, assume that we insert/delete a keyword into/from doc(v) of vertex v, all $\mathsf{T}(v')$ with $v \in \widetilde{\mathsf{L}}(v')$ need to be updated.

Assume that $v \in \widetilde{\mathsf{L}}(v')$. We discuss insertion and deletion of keywords separately. (1) Inserting a keyword kw into doc(v). First, we find the leaf node n representing the keyword kw in the trie of v'. If the trie does not contain such node, we should insert a new leaf node n into the trie to represent kw. Second, we should update the bitmaps of the nodes along the path from the root node to n. Specifically, for existing nodes that denote the prefix of kw, we should set the bit representing v to "1". The bitmaps of their children nodes have to be checked if they need to add a "0" bit. For new added nodes, we should create new bitmaps and corresponding compressed bitmaps. Finally, in $\mathsf{T}(v')$, if the leaf node n is newly added, we add it to $\mathsf{T}(v')$. Thereafter, we only need to update the B(n') of leaf nodes n', whose representing keywords have the same prefix as kw. (2) Deleting a keyword kw from doc(v). In the trie of v', for the nodes that denote the prefix of kw, we should set the bit representing v to "0" or delete it if there is no other keywords in doc(v) that have the same prefix. Next, in $\mathsf{T}(v')$, we delete the leaf

node and its $(\texttt{ID}(n), \texttt{B}(n))$ if other vertices in $\widetilde{\texttt{L}}(v)$ do not contain kw. Also, for the leaf nodes n' whose representing keywords have the same prefix as kw, we update the $\texttt{B}(n')$ as the first step.

Road network structure update. Another category of update is the update of road network structure, including the change of edge's weight and inserting/deleting edges/vertices. When the road network structure updates, first, we can use the existing 2-hop label maintenance algorithms [4, 39–41] to compute the updated 2-hop label. Second, based on the updated 2-hop label, we update the reverse 2-hop label. Specifically, if a pair $(v, \text{dis}(u, v))$ is added/deleted to/from $\texttt{L}(u)$, we should update $\widetilde{\texttt{L}}(v)$ by inserting/deleting $(u, \text{dis}(u, v))$. Based on the change of the reverse 2-hop label, we should maintain R2T index. We discuss the maintenance in two cases. (1) A new pair $(u, \text{dis}(u, v))$ is added to $\widetilde{\texttt{L}}(v)$. For the first case, we first insert the keywords of u into the trie of v one by one. Correspondingly, we should insert the nodes representing the inserted keyword into $\texttt{T}(v)$. Then, we update the bitmaps of $\texttt{T}(v)$ by traversing the trie of v in a depth-first manner. If the visited trie node is newly inserted, we should compute its bitmap and insert it into $\texttt{T}(v)$. Otherwise, we only need to update the existing bitmap as follows. (i) If u's keywords contain the prefix represented by the visited trie node n, we should add a "1" bit into n's bitmap to represent u, and then visit n's child node. (ii) If u's keywords do not contain the prefix represented by the visited trie node n, we should insert a "0" bit into n's bitmap to represent u, and stop visiting n's child node. After traversing the trie, all the bitmaps can be updated. (2) A pair $(u, \text{dis}(u, v))$ is deleted from $\widetilde{\texttt{L}}(v)$. For the second case, we first delete all the nodes, which denote the unique keywords of u, and its bitmaps from $\texttt{T}(v)$. Similarly, by traversing the trie of v in a depth-first manner, we update the bitmaps of $\texttt{T}(v)$. (i) If the prefix represented by the visited trie node n is contained in u's keywords, we should delete the "1" bit of u from n's bitmap, and then visit n's child node. (ii) If the prefix represented by the visited trie node n is not contained in u's keywords, we should delete the "0" bit of u from n's bitmap, and stop visiting n's child node.

6 QUERY PROCESSING ALGORITHMS

Using R2T index, we propose the query processing algorithms.

6.1 Instant Query Algorithm

First, in this section, we present an algorithm, called Instant Query Algorithm (IQA), to handle the first case of Problem 1, i.e., when a user types in a query string for the first time, the query returns the top-k geo-textual objects with the minimal scores. With the help of the R2T index introduced in the previous section, the query can be efficiently handled. First, we propose some lemmas, which establish the solid base to design IQA.

Lemma 6.1. *Given a vertex $v \in G$,*

$$\bigcup_{v' \in \texttt{L}(v)} \widetilde{\texttt{L}}(v') = V \tag{2}$$

Lemma 6.1 shows that, for a vertex v, all the reverse 2-hop labels $\widetilde{\texttt{L}}(v')$ of $v' \in \texttt{L}(v)$ constitute the vertex set of the road network. For example, in Table 2, $\texttt{L}(v_2) = \{(v_2, 0), (v_1, 4)\}$ and $\widetilde{\texttt{L}}(v_2) \cup \widetilde{\texttt{L}}(v_1) = V$. Based on Lemma 6.1, we can compute the distance from a vertex to all the other vertices by using the reverse 2-hop labels. Moreover,

Algorithm 1: Instant Query Algorithm (IQA)

Input: a query $q = (q.loc, q.str)$, parameters k and τ, a trie T of G
Output: a set $\mathcal{R}$ of k geo-textual objects

```
1  R ← ∅; C ← ∅;
2  AN ← find active nodes of T for q.str [13];
3  if AN = ∅ then
4      return R;
5  for ∀v ∈ L(q) do
6      if T(v) contains the leaf nods of AN then
7          v' ← MinL̃(v);  // Computing the vertex with
               the minimal score using Algorithm 2
8          C ← C ∪ (v', v, score(q, v'));
9  while |R| < k ∧ C ≠ ∅ do
10     (v', v, score(q, v')) ←
         arg min_{(v',v,score(q,v'))∈C} score(q, v');
11     C ← C − (v', v, score(q, v'));
12     if v' ∉ R then
13         R ← R ∪ v';
14     v'' ← MinL̃(v);  // Computing the next vertex
           with the minimal score using Algorithm 2
15     C ← C ∪ (v'', v, score(q, v''));
16 return R;
```

the reverse 2-hop label also enable the computation of the nearest neighbors in a progressive way as shown in the following lemma.

Lemma 6.2. *Given a vertex $v \in G$, for a vertex $v' \in \texttt{L}(v)$, let $v'' = \arg\min_{v'' \in \widetilde{\texttt{L}}(v')} \text{dis}(v', v'')$. Note that $v'' \neq v$. Then, the nearest neighbor of v is*

$$v'' = \arg\min_{v'' \in \widetilde{\texttt{L}}(v'), v' \in \texttt{L}(v)} \text{dis}(v, v') + \text{dis}(v', v'') \tag{3}$$

According to Lemma 6.2, in each reverse 2-hop label of $v' \in \texttt{L}(v)$, we can find the nearest neighbor of v'. Then, among all reverse 2-hop labels, the vertex with the minimal distance to v is the nearest neighbor of v. For instance, in Table 2, $\texttt{L}(v_4) = \{v_4, v_1\}$. In $\widetilde{\texttt{L}}(v_4)$, the closest vertex to v_4 is v_5 with $\text{dis}(v_4, v_5) = 1$. In $\widetilde{\texttt{L}}(v_1)$, the nearest vertex to v_1 is v_1 with $\text{dis}(v_1, v_1) = 0$. Since $\text{dis}(v_4, v_5) = 1$ and $\text{dis}(v_4, v_1) = 3$, v_5 is the nearest neighbor of v_4. Lemma 6.2 can also be extended to our problem as follows.

Corollary 6.1. *Given a query $q = (q.loc, q.str)$, for a vertex $v \in \texttt{L}(q)$, let $v' = \arg\min_{v' \in \widetilde{\texttt{L}}(v)} \texttt{score}(q, v')$. Then, the vertex with the minimal score w.r.t. q is*

$$v' = \arg\min_{v' \in \widetilde{\texttt{L}}(v), v \in \texttt{L}(q)} \texttt{score}(q, v') \tag{4}$$

In other words, the minimal score vertex is among the vertices, which have the minimal score in the reverse 2-hop label of $v' \in \texttt{L}(v)$. Motivated by Corollary 6.1, we develop IQA to find the k vertices with the minimal score. The basic idea of IQA is to progressively return the vertices with the minimal score among all reverse 2-hop labels of $v \in \texttt{L}(q)$. To this end, we should find the vertex with the minimal score for each reverse 2-hop label of $v \in \texttt{L}(q)$. Then, in each round, we select the vertex with the minimal score among all reverse 2-hop labels of $v \in \texttt{L}(q)$, and add it to the results. Assume the vertex with the minimal score is in $\widetilde{\texttt{L}}(v)$. After this, we should compute the next vertex with the minimal score in $\widetilde{\texttt{L}}(v)$ for the next

Algorithm 2: Min$\widetilde{\mathsf{L}}(v)$

Input: a set AN of active nodes, R2T(v) index of a vertex v
Output: a vertex v' with the minimal score in $\widetilde{\mathsf{L}}(v)$

```
1  v' ← ∅; MinScore ← +∞;
2  if BTag[v] = ∅ then
3      initialize BTag[v];
4  for each active node an ∈ AN do
5      find a leaf node n of an using binary search;
6      for i ← an.depth to 1 do
7          if i = an.depth then
8              BTag[v][n][i].y = BTag[v][n][i].y + 1;
9          else
10             BTag[v][n][i].y = BTag[v][n][i + 1].x;
11         BTag[v][n][i].x ← the position of
             BTag[v][n][i].y-th "1" bit in bitmap B[v][n][i];
12         if BTag[v][n][i].x = null then
13             break;
14         i − −;
15     v'' ← (BTag[v][n][i].x)-th vertex in L̃(v);
16     if score(q, v'') < MinScore then
17         v' ← v'';
18         MinScore ← score(q, v'');
19 return v';
```

round processing. IQA repeats the selection of vertices having the minimal scores until all results are found.

Algorithm 1 shows the pseudo-code of IQA. First, IQA computes the active nodes in the complete trie of the road network (line 2). Here, the active node is the node whose represented string/keyword's edit distance to $q.str$ is not larger than τ. It is worth mentioning that, the active nodes are mainly used to compute the score of the vertex, which will be illustrated later. If there is no such active node, it means that all vertices do not match the $q.str$. IQA returns an empty result (lines 3-4). Otherwise, IQA finds the vertex with the minimal score for each reverse 2-hop label of $v \in \mathsf{L}(q)$ (lines 5-8). Then, IQA repeatedly selects the vertex with the minimal score among all reverse 2-hop labels until all results are found (lines 9-15). Finally, IQA returns the top-k geo-textual objects (line 16).

Next, we introduce how to find the vertex with the minimal score in a reverse 2-hop label using R2T index. Note that, we have to keep in mind that the vertex with the minimal score should satisfy the prefix edit distance constraint in Definition 3.2. Recall that, at the beginning of Algorithm 1, we have computed the active nodes in the complete trie. The edit distances between the prefixes represented by those active nodes and $q.str$ are no larger than τ. Hence, if a vertex's keyword contains the prefix denoted by an active node, the vertex is a candidate of the minimal score vertex. Thus, a naive method is to compute the score for all candidates in the reverse 2-hop label. However, this naive method is not progressive. In the sequel, we devise a method to progressively return the minimal score vertex in a reverse 2-hop label.

In $\mathsf{T}(v)$ of R2T(v), we store the bitmap for the nodes of trie. Each bit of the bitmap indicates whether the keyword of corresponding vertex contains the prefix represented by the trie node. (1) For a node of trie, the vertices, whose keyword includes the prefix denoted by the node, have the same textual similarity. (2) In the reverse 2-hop label, the vertices are in ascending order of the distances. Based on the above two facts, for an active node, the vertex with the minimal score is just the vertex represented by the first "1" bit in the corresponding bitmap of the active node. In the same way, the vertex denoted by the second "1" bit in the bitmap of the active node is the second minimal score vertex. Using this property of the active node, for a reverse 2-hop label, we can compute the minimal score vertex for each active node. The vertex having the smallest score among all active nodes is the minimal score vertex in the reverse 2-hop label.

Next, we introduce how to find the minimal score vertex for an active node. Finding the minimal score vertex for an active node is only to find the first "1" bit in the bitmap of the active node. A straightforward way is to convert the compressed bitmap to the full bitmap, and we can quickly find the vertex of the first "1" bit. However, this method is inefficient since it has to convert the compressed bitmaps from the active to the root node. In view of this, we devise a novel method via traversing the compressed bitmaps from the active node to the root node with the help of an auxiliary structure BTag, which is defined as follows.

Definition 6.1. *A* BTag *of a bitmap is a pair of* $\langle x, y \rangle$*, which denotes that the position of* y*-th "1" bit is* x*.*

$\langle x, y \rangle$ of a bitmap means that from the first bit to the x-th bit, there are y "1" bits. For example, let a bitmap be "110011", the BTag $\langle 5, 3 \rangle$ indicates that the fifth bit is the 3-th "1" bit in "110011". The BTag has the following property.

Lemma 6.3. *Given two trie nodes* n_1 *and* n_2*, two pairs* $\langle x_1, y_1 \rangle$ *and* $\langle x_2, y_2 \rangle$*, which are the* BTag*s of the bitmaps of* n_1 *and* n_2*, respectively. Assume that* n_1 *is a child node of* n_2*. If* $y_2 = x_1$*, the vertex represented by the* x_1*-th bit in the bitmap of* n_1 *and the vertex denoted by the* x_2*-th bit in the bitmap of* n_2 *are the same.*

For instance, in Figure 5, the compressed bitmaps of n'_2 and n'_6 are "0110" and "10", respectively. Both $\langle 1, 1 \rangle$ of "10" and $\langle 2, 1 \rangle$ of "0110" represent the second vertex in $\widetilde{\mathsf{L}}(v_6)$, i.e., v_7. Lemma 6.3 not only enables us to traverse the compressed bitmaps in a bottom-up manner to get the minimal score vertex for an active node, but also allows us to get the next minimal score vertex in the same way.

Based on the above discussion, we propose an algorithm to find the vertex with the minimal score in a reverse 2-hop label, whose pseudo-code is depicted in Algorithm 2. Initially, when Algorithm 2 computes the minimal score vertex for the first time, it initializes BTag (lines 2-3), i.e., to set BTag as $\langle 0, 0 \rangle$. Then, Algorithm 2 computes the minimal score vertex for each active node (lines 4-18). For each active node, it first gets a leaf node containing the bitmap of the active node using binary search (line 5). Note that, there may be multiple such leaf nodes. We only select any one of them. Next, Algorithm 2 traverses the bitmaps from the active node to the child of the root node for getting the position of the minimal score vertex of the active node (lines 6-14). Finally, the minimal score vertices of all active nodes are found, and the one having the smallest score among all active nodes is returned (lines 15-19).

Optimizations. We present two optimizations to speed up Algorithms 1 and 2.

Optimizations 1. Algorithm 1 computes the minimal score vertex for each reverse 2-hop label (lines 5-8) for initialization. To improve

the efficiency, we can first compute the lower bound of the score for a reverse 2-hop label. If the lower bound is larger than the current minimal score, the reverse 2-hop label definitely does not exist the minimum score vertex among all reverse 2-hop labels. Hence, we can skip the minimal score vertex computation for this reverse 2-hop label. Specifically, we can use Equation 1 to compute the lower bound of the score for the reverse 2-hop label $\widetilde{\mathsf{L}}(v)$. The spatial similarity is the minimal distance between q and the vertex in $\widetilde{\mathsf{L}}(v)$, and the textual similarity is the minimal edit distance between $q.str$ and the strings represented by active nodes.

Optimizations 2. Algorithm 2 needs to compute the minimal score vertex for every active node (lines 4-18). If the number of active nodes is large, it is costly. Actually, many active nodes have a parent-child relationship, and the vertices denoted by the bitmap of a child active node is a subset of that of its father active node. Thus, we need to only compute the minimal score vertex for father active nodes.

The time and space complexities of IQA are analyzed below.

Theorem 6.4. *The time and space complexities of IQA are* $O((k + w \cdot \log|V|) \cdot |AN| \cdot \sqrt{w \cdot \log|V|})$ *and* $O(w \cdot \log|V| \cdot |AN| \cdot |q.str|)$, *respectively.* $|AN|$ *denotes the number of active nodes.*

Proof. The time complexity of IQA mainly determined by two parts. First, IQA finds the vertex with the minimal score for each reverse 2-hop label of $v \in \mathsf{L}(q)$, which takes $O(w \cdot \log|V| \cdot |AN| \cdot \sqrt{w \cdot \log|V|})$ time. Then, IQA repeatedly selects the vertex with the minimal score among all reverse 2-hop labels until k vertices are found, which takes $O(k \cdot |AN| \cdot \sqrt{w \cdot \log|V|})$ time. Totally, IQA takes $O((k + w \cdot \log|V|) \cdot |AN| \cdot \sqrt{w \cdot \log|V|})$ time. The space overhead of IQA is determined by BTags, which needs $O(w \cdot \log|V| \cdot |AN| \cdot |q.str|)$ space. Thus, the space complexity of IQA is $O(w \cdot \log|V| \cdot |AN| \cdot |q.str|)$. □

6.2 Instant Update Algorithms

In this section, we present how to update the query results when the query string changes. A straightforward method is to query from the scratch. However, this method is not efficient, especially when updates are frequent. To this end, we extend Algorithms 1 and 2 to update the query results. We only need to modify three places in Algorithms 1 and 2. First, at the initialization step, Algorithm 1 should compute the first minimal score vertex for each reverse 2-hop label (lines 5-8). For the update algorithm, we can reuse the minimal score vertex found in the previous query, and the initialization can be omitted. Second, Algorithm 2 computes the minimal score vertex by traversing the bitmaps of active nodes and their ancestor nodes from the first bit, during which BTags are used. For the update algorithm, we can reuse the visited vertices, and traverse the bitmap from the current bit. Third, we have to recompute the score of previous query results based on the new query string to prune the unqualified vertices. Next, we detail the second modification for three cases as follows.

Case I: Inserting character(s) at the end of the query string. If the query string changes, the active nodes change accordingly. Let AN and AN' be the active nodes before and after inserting character(s) at the end of the query string. Then, it satisfies that $\forall an' \in AN'$, we can find an active $an \in AN$ such that $an = an'$ or an' is a descender node of an. If $an = an'$, we can still use the bitmap and corresponding BTag of an for an' to find the minimal score vertex. If an' is a child node of an, we should initialize the BTag of an' through an. Specifically, let $\langle x', y\rangle$ and $\langle x, y\rangle$ be the BTag of an' and an, respectively. Then, x' can be set to y. In the same way, we can iteratively derive the BTag for the bitmap of a descender node.

Case II: Deleting character(s) at the end of the query string. In order to achieve a fast update of deleting character(s) at the end, BTags for each insertion (Case I) need to be saved. BTags will be rolled back to the state corresponding to the string after the letter is deleted, and AN will also be rolled back. Note that if the BTags needed are not saved, then roll back to further back. The letters that pass at the end can be treated as Case I.

Case III: Inserting/deleting character(s) at random position of the query string. Let AN and AN' be the active nodes before and after deleting character(s) at random position of the query string. We rolled back BTags and AN like Case II to the position. Then, for each active nodes in AN', it satisfies that $\forall an' \in AN'$, we can find an active $an \in AN$ such that $an = an'$ or an' is a descender node of an. This relationship is consistent with that in Case I, so we can update BTags with the method in Case I. For example, let q =(v_9, "sto"), $k = 3$, $\tau = 1$, and $\alpha = 0.5$. After the initial query finishes, we have $AN = \{n_{36}, n_{56}, n_{60}\}$. If we insert a character "h" after the character "t" in the string "sto", IUA first rolls back the query state to q=(v_9, "st") and updates $AN = \{n_{30}, n_{36}\}$. Then, IUA gets the active nodes AN' for "stho", i.e., $AN' = \{n_{32}, n_{56}, n_{60}\}$. Finally, for each active node of AN', IUA initializes its BTags by inheriting the BTags from its ancestor node in AN.

6.3 Multiple Query Strings

Sections 6.1 and 6.2 mainly aim at a single query string. In this section, we discuss how to handle multiple query strings. After the user types in a query string, if the result does not meet his/her expectation, he/she may proceed to type in more query strings. The methods proposed in Sections 6.1 and 6.2 can also be extended to tackle multiple query strings. In particular, we discuss the extension of Algorithms 1 and 2 for the case of multiple query strings.

When a new query string $q.str'$ is added, Algorithm 1 should find new active nodes for $q.str'$ (line 2). Then, it iteratively computes the vertex with the minimal score to find the top-k results by using Algorithm 2 (lines 5-15), which is the same as the case of a single query string. Specifically, Algorithm 2 computes the vertex with the minimal score for each active node (lines 4-18). For a single query string, the first "1" bit in the bitmap of active node is the minimal score vertex. But, for multiple query strings, we should compute the scores of vertices denoted by "1" bits in the bitmap of active node until the lower bound of vertex's score is larger than the current minimum score. The lower bound of vertex's score can be computed using Equation 1, where the textual similarity is the sum of the minimal prefix edit distance for all query strings. In addition, if a vertex's prefix edit distance w.r.t. previously entered query strings is larger than τ, it cannot become the final result and thus can be skipped the score computation.

7 PERFORMANCE STUDY

This section evaluates the performance of our proposed index and algorithms. All algorithms were implemented in C++, and compiled

Table 3: Statistics of road networks

Dataset	Region	$\|V\|$	$\|E\|$	$\|\mathsf{doc}(V)\|$	$\|W\|$
NY	New York City	264,346	733,846	157,100	6,556
FLA	Florida	1,070,376	2,712,798	343,452	16,656
CAL	California	1,890,815	4,657,742	401,258	20,319
LKS	Great Lakes	2,758,119	6,885,658	615,168	25,807
EU	Eastern USA	3,598,623	8,778,114	780,749	33,522
WU	Western USA	6,262,104	15,248,146	1,580,430	52,316
CTR	Central USA	14,081,816	34,292,496	2,782,249	78,658
USA	Full USA	23,947,347	58,333,344	4,118,452	112,353

by GCC 7.5.0 with -O3 optimization. The experiments were conducted on a machine running on Ubuntu server 18.04.5 LTS version with two Intel Xeon 2.40GHZ processors and 512G main memory.

Datasets: We employ eight real-world road networks in experiments. The road networks are obtained from DIMACS[1], which do not contain POIs. For each road network, we get POIs in the corresponding area from OpenStreeMap (OSM)[2] and map it to the road network. Here, OSM is an open-source mapping web site that provides POIs for different countries and cities. After getting the road network and POIs, the keywords of each POI are mapped to the closest vertex on the road network. Table 3 lists the statistics of the road networks, where $|\mathsf{doc}(V)| = \sum_{v \in V} |\mathsf{doc}(v)|$, and $|W|$ represents the number of distinct keywords in the road network.

Compared Methods: In experiments, the competitors include the naive method, the keyword query method, the instant query algorithm (IQA), and the instant update algorithm (IUA). Recall that, the first step of the naive method is to traverse the road network from $q.loc$. We implemented three versions of the naive method by using different techniques to traverse the road network, including Dijkstra, G*-tree, and 2-hop label. Note that, for G*-tree, we employ the default parameter settings as reported in [25]. Also, we extend two state-of-the-art spatial keywords query methods, i.e., K-SPIN [1] and KT [20], to handle our problem. Specifically, first, we find all the complete keywords, whose prefix edit distance is not larger than τ. These keywords are used as candidate keywords. Then, we use K-SPIN and KT to find the k geo-textual objects with the minimum score.

Metrics: We report the query time, index construction/update time, and index size in our experiments. For the evaluation of query efficiency, we randomly generate 5000 queries, and finally report the average query time. Note that, we only report empirical results of partial datasets in this paper due to the space limitation and the similar trends in the results of different datasets. Please refer to Appendix for the complete empirical results.

7.1 Evaluation of Instant Query Algorithm

In this section, we study the efficiency of instant query algorithm.

Exp-1: Effect of $|q.str|$. We first verify the effect of the query string length $|q.str|$. We vary $|q.str|$ from 1 to 7, and fix the other parameters to default values. Figure 7(a) shows the experimental results. We can observe that, with the growth of $|q.str|$, the query time of K-SPIN and KT drops; the query time of G*-tree, 2-hop, and Dijkstra increases; and the query time of IQA ascends as $|q.str| \leq 3$. When $|q.str| > 3$, the query time of IQA almost does not change. For K-SPIN and KT, the longer $q.str$ is, the less the candidate keywords, resulting less query time. For G*-tree, 2-hop, and Dijkstra, if $q.str$ becomes longer, there are less vertices satisfying the textual constraints. Thus, the search spaces of G*-tree, 2-hop, and Dijkstra become larger, incurring more query time. For IQA, the number of active nodes increases with the growth of $|q.str|$ when $|q.str| \leq 3$. Since we set the default value of error threshold to 2, the number of active nodes goes down gradually when $|q.str| > 3$. Hence, the impact of the increased search space is offset, the running time of IQA almost keep the same as $|q.str| > 3$. Moreover, IQA outperforms other algorithms by 1-2 orders of magnitude.

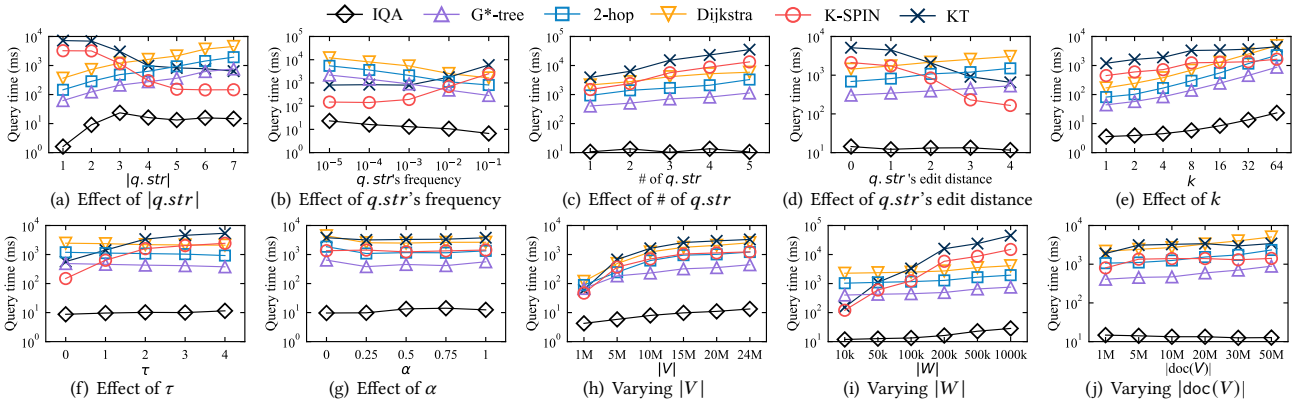
Exp-2: Effect of $q.str$'s frequency. Next, we evaluate the effect of $q.str$'s frequency. Here, the $q.str$'s frequency is $\frac{|V(q.str)|}{|\mathsf{doc}(V)|}$, where $V(q.str) = \{v | \forall v \in V, \exists kw \in \mathsf{doc}(v), q.str \preceq kw\}$. The query time are shown in Figure 7(b). The query time of all algorithms decrease with the increase of $q.str$'s frequency. The reason behind is that if $q.str$ is frequent in the road network, there are more vertices containing $q.str$. Thus, the results are easier to be found, resulting in less query time.

Exp-3: Effect of # of $q.str$. In this experiment, we explore the effect of the number of query strings, whose value is varied from 1 to 5. The query time of all algorithms are depicted in Figure 7(c). Specifically, the query time of IQA almost remains stable while the query time of other algorithms increase with the growth of the number of query strings. This is because IQA finds the minimal score vertices by traversing the bitmaps of active nodes. Although the number of active nodes grows when the number of query strings increases, IQA usually can find the results by accessing a small number of active nodes. Therefore, the query time of IQA does not change much. For the other algorithms, if the number of query strings increases, fewer vertices will satisfy the textual constraint. Thus, these algorithms need to traverse more vertices to find the results, incurring more query time.

Exp-4: Effect of $q.str$'s edit distance. Recall that our proposed algorithms can tolerate the typos in $q.str$. Thus, we study the effect of $q.str$'s edit distance. To this end, we assume that $q.str$ includes several typos. The $q.str$'s edit distance is the edit distance between $q.str$ and the correct string. Note that, to ensure non-empty query result, we set $\tau = 4$ in this experiment. Figure 7(d) shows the empirical results. We can observe that when the $q.str$'s edit distance increases, the query time of IQA, K-SPIN, and KT gradually drops while the query time of other algorithms gradually grows. This is because the larger the $q.str$'s edit distance, the less vertices in the road network satisfy the prefix edit distance constraint, meaning that G*-tree, 2-hop, and Dijkstra need to search more vertices. For K-SPIN and KT, the larger the editing distance, the fewer candidate keywords, and therefore the shorter the query time. IQA can quickly skip invalid vertices since the larger $q.str$'s edit distance leads to less active nodes.

Exp-5: Effect of k. We investigate the effect of k by varying k from 1 to 64. Figure 7(e) plots the query time of four algorithms. As k becomes larger, the query time of all algorithms increase. The reason behind is that the larger k indicates more results. Hence, all algorithms take more time to query. However, the performance of IQA is still much better than that of other algorithms.

[1] http://www.diag.uniroma1.it//~challenge9/download.shtml
[2] https://www.openstreetmap.org/



(a) Effect of $|q.str|$ (b) Effect of $q.str$'s frequency (c) Effect of # of $q.str$ (d) Effect of $q.str$'s edit distance (e) Effect of k

(f) Effect of τ (g) Effect of α (h) Varying $|V|$ (i) Varying $|W|$ (j) Varying $|\mathsf{doc}(V)|$

Figure 7: Evaluation of instant query algorithm on USA

Exp-6: Effect of τ. Then, we evaluate the effect of error threshold τ on algorithms. The empirical results are reported in Figure 7(f). We can observe that the query time of G*-tree, 2-hop, and Dijkstra decrease while the query time of IQA, K-SPIN, and KT increase with the growth of τ. If τ becomes larger, (1) more vertices satisfy the textual constraint, and (2) more active nodes and candidate keywords will be found. Thus, the G*-tree, 2-hop, and Dijkstra spend less time while IQA, K-SPIN, and KT take more time.

Exp-7: Effect of α. α represents the user preference for score computation in Equation 1. This set of experiment test the effect of α on algorithms and the results are shown in Figure 7(g). We can observe that when $\alpha = 0$, other algorithms take more time. This is because the text pruning abilities of other algorithms are weak. For other values of α, the performance of all algorithms are stable, meaning that α has little effect on algorithms.

Exp-8: Scalability. In this set of experiments, we verify the scalability of the algorithms. In view of this, we vary the vertex cardinality $|V|$, the number of distinct keywords $|W|$, and the occurrences of keywords $|\mathsf{doc}(V)|$. Figures 7(h), 7(i), and 7(j) plot the query time by changing $|V|$, $|W|$, and $|\mathsf{doc}(V)|$, respectively. In Figure 7(h), with the growth of vertex cardinality, the performance of all algorithms degrade since the larger road network needs more time to find the results. In Figure 7(i), as the number of distinct keywords increases, the performance of four algorithms degrade as well. This is because when the number of distinct keywords grows, the keywords will become less frequent. Hence, all algorithms take more time with the growth of $|W|$, which is consistent with the empirical results of $q.str$'s frequency depicted in Figure 7(b). In Figure 7(j), as the occurrences of keywords increase, the performance of IQA keeps stable while that of other algorithms all degrade. If the occurrences of keywords grow, the vertices contain more keywords. Thus, the search spaces of other algorithms become larger, incurring more query time. On the other hand, the growth of the occurrences of keywords does not affect the complete trie of the road network. Hence, the active nodes found by IQA do not change as well. Therefore, the query time of IQA keeps stable.

7.2 Evaluation of Instant Update Algorithm

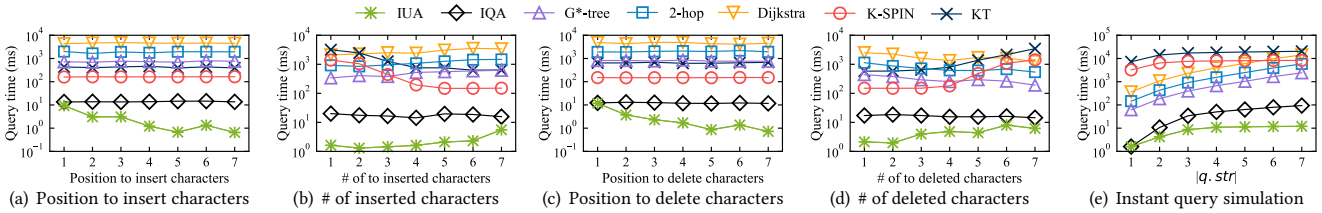
This section evaluates the performance of instant update algorithm. IQA, G*-tree, Dijkstra, 2-hop label, K-SPIN, and KT are instant query algorithms used for results update, i.e., to query from the scratch. IUA is the instant update algorithm presented in Section 6.2.

Exp-9: Effect of the position to insert characters. First, we explore the effect of inserting characters into different positions of $q.str$. Here, if we insert the characters into the position i of $q.str$, it means that we insert the characters after the i-th character of $q.str$. Note that, the $|q.str| \geq 7$, and we vary the position from 1 to 7. We insert one character in this experiment. Figure 8(a) depicts the empirical results. Specifically, the query time of all algorithms almost remain the same, except for IUA. Moreover, we can observe that if the insertion position is closer to the end of $q.str$, IUA has better performance. This is because the closer to the end of the query string, the less BTags will be updated. Thus, IUA needs less time to update. In addition, IUA has better performance compared with other algorithms. Specifically, IUA returns results in an average time of 2.1ms while IQA takes 10ms.

Exp-10: Effect of # of inserted characters. Next, we investigate the effect of inserting different number of characters into $q.str$. To this end, we extract characters from a complete keyword, and take the remaining string as $q.str$. Then, in experiments, we insert the extracted characters into $q.str$. In this way, we ensure the query result is non-empty. The query time of all algorithms are shown in Figure 8(b). When the number of inserted characters increases, IQA, G*-tree, Dijkstra, and 2-hop label take more query time. This is because when we insert more characters, $q.str$ becomes more accurate. They should traverse more vertices to find the results, and thus need more time to query. For IUA, if we insert more characters, the difference between the original query string and new query string is greater. Hence, IUA has to need more time to update BTags. Although IUA becomes less efficient when more characters are inserted, it is still better than IQA, and is 2 orders of magnitude faster than other algorithms.

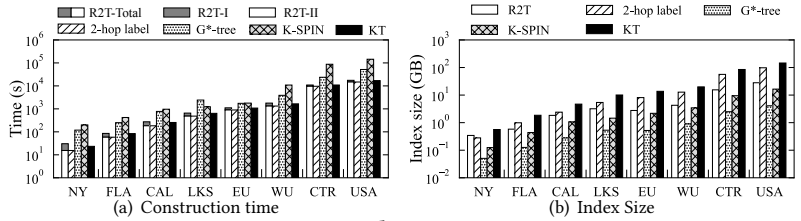
Exp-11: Effect of the position to delete characters. In this experiment, we study the instant update algorithm by deleting characters from different positions of the query string. We vary the deletion position from 1 to 7. Here, the deletion position i means that the i-th character of $q.str$ is deleted. Figure 8(c) depicts the empirical results. As observed, the closer the deletion position is to the end of $q.str$, the better performance of IUA. The reason behind is similar with that of Exp-9, i.e., when deleting characters near the end of $q.str$, less BTags need to be updated.

Exp-12: Effect of # of deleted characters. Then, we verify the effect of deleting different number of characters from $q.str$. In view of this, we randomly choose a deletion position in $q.str$, and delete



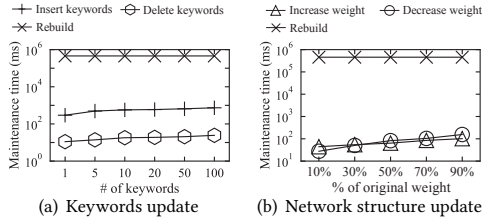
(a) Position to insert characters (b) # of inserted characters (c) Position to delete characters (d) # of deleted characters (e) Instant query simulation

Figure 8: Evaluation of instant update algorithm on USA



(a) Construction time (b) Index Size

Figure 9: Index construction



(a) Keywords update (b) Network structure update

Figure 10: Index maintenance (WU)

a certain number of characters, which varies from 1 to 7. Empirical results are plotted in Figure 8(d). We have the following observations. With the growth of the number of deleted characters, (1) the query time of G*-tree, Dijkstra, and 2-hop label drops while that of K-SPIN and KT increases, (2) the query time of IQA remains unchanged, and (3) the query time of IUA increases slowly. However, IUA is still able to handle the deletion of characters efficiently, and outperforms other algorithms by 1-2 orders of magnitude.

Exp-13: Instant query simulation. This experiment simulates the users' instant queries, i.e., to simulate users type in the query string character-by-character. As soon as a character is typed in, we perform the query/update algorithms for the new query string, and return the results. Finally, after the users type in the complete query string, we report the total query/update time, as depicted in Figure 8(e). Note that, the length of query string for this experiment changes from 1 to 7. As expected, the total query time of all algorithms increase if the length of query string becomes larger. Nevertheless, the total query time of other algorithms ascends much faster than that of both IUA and IQA. This is because the query information inheritance mechanism of IUA makes it be able to perform update incrementally. In addition, the performance of IUA is much better than other algorithms. This is because IUA leverages a query information inheritance mechanism that enables the efficient utilization of information from previous queries. As a result, IUA can incrementally update the results, and significantly reduce query time compared with other algorithms.

7.3 Index Evaluation

In this section, we evaluate the performance of R2T index, including the index construction and maintenance.

Exp-14: Index Construction. First, we investigate the performance of index construction. In this experiment, we take G*-tree, 2-hop label, K-SPIN, and KT as competitors. Figures 9(a) and 9(b) show the index construction time and index size, respectively. Note that, in Figure 9(a), we split the construction time of R2T into two parts, i.e., R2T-I and R2T-II. Specifically, R2T-I represents the time of computing 2-hop label for every vertex, and R2T-II denotes the time of constructing $\widetilde{\mathsf{L}}(v)$ and $\mathsf{T}(v)$ for every vertex after getting 2-hop labels. In Figure 9(a), we can observe that R2T-I takes up most of the construction time. For example, on the road network *CTR*, the time of R2T-I and R2T-II are 1289s and 9626.5s, respectively. R2T-I is 11.8% of the total construction time. Overall, the construction time of R2T is less than or comparable with other indexes except the 2-hop label. It is obvious since the construction time of R2T including the 2-hop label computation time and the $\widetilde{\mathsf{L}}(v)$ and $\mathsf{T}(v)$ construction time. In Figure 9(b), G*-tree has the smallest size and KT has the largest size. The size of R2T is smaller than that of the 2-hop label in most cases. The reason is tow-fold. (1) In the road network, the vertices without keywords cannot contribute to the final results. We do not store these vertices in R2T. (2) The compression technique significantly reduces the bitmap size.

Exp-15: Index maintenance. Next, we explore the performance of index maintenance. We mainly consider two categories of the road network update, including keywords update and road network structure update. The keywords update is to insert/delete a certain number of keywords into/from $\mathsf{doc}(v)$ of a vertex v. Figure 10(a) plots the index maintenance time for the keywords update. The road network structure update contains the update of edges' weight and the insertion/deletion of edges/vertices. Since the insertion/deletion of edges/vertices can be reduced to the update of edges' weight [41], we mainly consider the update of edges' weight in this experiment, which include the cases of increasing and decreasing the weight of an edge. For each road network, we select 1000 edges at random and change their weights. The maintenance time of road network structure update is shown in Figure 10(b).

8 CONCLUSIONS

In this paper, we study the problem of instant error-tolerant spatial keyword queries on road networks. To efficiently answer the queries, we propose a new index called R2T. R2T employs reverse 2-hop label and trie to seamlessly integrate the spatial and textual information for each vertex of the road network. Based on R2T, we present a suite of algorithms to answer the queries. Both theoretical analysis and empirical evaluation demonstrate the efficiency of our proposed index and algorithms. This work is our first step towards the studied problem. In the future, we would like to investigate the instant error-tolerant spatial keyword queries for moving query object or in a distributed environment.

REFERENCES

[1] Tenindra Abeywickrama, Muhammad Aamir Cheema, and Arijit Khan. 2020. K-SPIN: Efficiently Processing Spatial Keyword Queries on Road Networks. *IEEE Trans. Knowl. Data Eng.* 32, 5 (2020), 983–997.

[2] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast Exact Shortest-path Distance Queries on Large Networks by Pruned Landmark Labeling. In *SIGMOD*. 349–360.

[3] Hannah Bast and Ingmar Weber. 2006. Type Less, Find More: Fast Autocompletion Search with A Succinct Index. In *SIGIR*. 364–371.

[4] Ramadhana Bramandia, Byron Choi, and Wee Keong Ng. 2010. Incremental Maintenance of 2-Hop Labeling of Large Graphs. *IEEE Trans. Knowl. Data Eng.* 22, 5 (2010), 682–698.

[5] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. 2013. Spatial Keyword Query Processing: An Experimental Evaluation. *Proc. VLDB Endow.* 6, 3 (2013), 217–228.

[6] Lei Chen, Jianliang Xu, Xin Lin, Christian S. Jensen, and Haibo Hu. 2016. Answering Why-not Spatial Keyword Top-k Queries via Keyword Adaption. In *ICDE*. 697–708.

[7] Zhida Chen, Lisi Chen, Gao Cong, and Christian S. Jensen. 2021. Location- and Keyword-based Querying of Geo-textual Data: A Survey. *VLDB J.* 30, 4 (2021), 603–640.

[8] Zitong Chen, Ada Wai-Chee Fu, Minhao Jiang, Eric Lo, and Pengfei Zhang. 2021. P2H: Efficient Distance Querying on Road Networks by Projected Vertex Separators. In *SIGMOD*. 313–325.

[9] Zhongpu Chen, Bin Yao, Zhi-Jie Wang, Xiaofeng Gao, Shuo Shang, Shuai Ma, and Minyi Guo. 2021. Flexible Aggregate Nearest Neighbor Queries and its Keyword-aware Variant on Road Networks. *IEEE Trans. Knowl. Data Eng.* 33, 12 (2021), 3701–3715.

[10] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2002. Reachability and Distance Queries via 2-Hop Labels. In *SODA*. 937–946.

[11] Gao Cong and Christian S. Jensen. 2016. Querying Geo-textual Data: Spatial Keyword Queries and Beyond. In *SIGMOD*. 2207–2212.

[12] Gao Cong, Christian S. Jensen, and Dingming Wu. 2009. Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects. *Proc. VLDB Endow.* 2, 1 (2009), 337–348.

[13] Dong Deng, Guoliang Li, He Wen, H. V. Jagadish, and Jianhua Feng. 2016. META: An Efficient Matching-based Method for Error-tolerant Autocompletion. *Proc. VLDB Endow.* 9, 10 (2016), 828–839.

[14] Yuyang Dong, Chuan Xiao, Hanxiong Chen, Jeffrey Xu Yu, Kunihiro Takeoka, Masafumi Oyamada, and Hiroyuki Kitagawa. 2021. Continuous Top-k Spatial-keyword Search on Dynamic Objects. *VLDB J.* 30, 2 (2021), 141–161.

[15] Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. 2013. IS-LABEL: An Independent-set Based Labeling Scheme for Point-to-point Distance Querying. *Proc. VLDB Endow.* 6, 6 (2013), 457–468.

[16] Yunjun Gao, Xu Qin, Baihua Zheng, and Gang Chen. 2015. Efficient Reverse Top-k Boolean Spatial Keyword Queries on Road Networks. *IEEE Trans. Knowl. Data Eng.* 27, 5 (2015), 1205–1218.

[17] Yunjun Gao, Jingwen Zhao, Baihua Zheng, and Gang Chen. 2016. Efficient Collective Spatial Keyword Query Processing on Road Networks. *IEEE Trans. Intell. Transp. Syst.* 17, 2 (2016), 469–480.

[18] Sheng Hu, Chuan Xiao, and Yoshiharu Ishikawa. 2018. An Efficient Algorithm for Location-aware Query Autocompletion. *IEICE Trans. Inf. Syst.* 101-D, 1 (2018), 181–192.

[19] Shengyue Ji and Chen Li. 2011. Location-based Instant Search. In *SSDBM*, Vol. 6809. 17–36.

[20] Minhao Jiang, Ada Wai-Chee Fu, and Raymond Chi-Wing Wong. 2015. Exact Top-k Nearest Keyword Search in Large Networks. In *SIGMOD*. 393–404.

[21] Guoliang Li, Jianhua Feng, and Chen Li. 2013. Supporting Search-As-You-Type Using SQL in Databases. *IEEE Trans. Knowl. Data Eng.* 25, 2 (2013), 461–475.

[22] Guoliang Li, Shengyue Ji, Chen Li, and Jianhua Feng. 2009. Efficient Type-ahead Search on Relational Data: A TASTIER approach. In *SIGMOD*. 695–706.

[23] Guoliang Li, Shengyue Ji, Chen Li, and Jianhua Feng. 2011. Efficient Fuzzy Full-text Type-ahead Search. *VLDB J.* 20, 4 (2011), 617–640.

[24] Ye Li, Leong Hou U, Man Lung Yiu, and Ngai Meng Kou. 2017. An Experimental Study on Hub Labeling based Shortest Path Algorithms. *Proc. VLDB Endow.* 11, 4 (2017), 445–457.

[25] Zijian Li, Lei Chen, and Yue Wang. 2019. G*-Tree: An Efficient Spatial Index on Road Networks. In *ICDE*. 268–279.

[26] Zhisheng Li, Ken C. K. Lee, Baihua Zheng, Wang-Chien Lee, Dik Lun Lee, and Xufa Wang. 2011. IR-Tree: An Efficient Index for Geographic Document Search. *IEEE Trans. Knowl. Data Eng.* 23, 4 (2011), 585–599.

[27] Ying Lu, Jiaheng Lu, Gao Cong, Wei Wu, and Cyrus Shahabi. 2014. Efficient Algorithms and Cost Models for Reverse Spatial-keyword k-nearest Neighbor Search. *ACM Trans. Database Syst.* 39, 2 (2014), 13:1–13:46.

[28] Dian Ouyang, Lu Qin, Lijun Chang, Xuemin Lin, Ying Zhang, and Qing Zhu. 2018. When Hierarchy Meets 2-Hop-labeling: Efficient Shortest Distance Queries on Road Networks. In *SIGMOD*. 709–724.

[29] Miao Qiao, Lu Qin, Hong Cheng, Jeffrey Xu Yu, and Wentao Tian. 2013. Top-k Nearest Keyword Search on Large Graphs. *Proc. VLDB Endow.* 6, 10 (2013), 901–912.

[30] Jianbin Qin, Chuan Xiao, Sheng Hu, Jie Zhang, Wei Wang, Yoshiharu Ishikawa, Koji Tsuda, and Kunihiko Sadakane. 2020. Efficient Query Autocompletion with Edit Distance-based Error Tolerance. *VLDB J.* 29, 4 (2020), 919–943.

[31] João B. Rocha-Junior and Kjetil Nørvåg. 2012. Top-k Spatial Keyword Queries on Road Networks. In *EDBT*. 168–179.

[32] Senjuti Basu Roy and Kaushik Chakrabarti. 2011. Location-aware Type Ahead Search on Spatial Databases: Semantics and Efficiency. In *SIGMOD*. 361–372.

[33] Jin Wang and Chunbin Lin. 2020. Fast Error-tolerant Location-aware Query Autocompletion. In *ICDE*. 1998–2001.

[34] Xiang Wang, Wenjie Zhang, Ying Zhang, Xuemin Lin, and Zengfeng Huang. 2017. Top-k Spatial-keyword Publish/Subscribe over Sliding Window. *VLDB J.* 26, 3 (2017), 301–326.

[35] Chuan Xiao, Jianbin Qin, Wei Wang, Yoshiharu Ishikawa, Koji Tsuda, and Kunihiko Sadakane. 2013. Efficient Error-tolerant Query Autocompletion. *Proc. VLDB Endow.* 6, 6 (2013), 373–384.

[36] Hongfei Xu, Yu Gu, Yu Sun, Jianzhong Qi, Ge Yu, and Rui Zhang. 2020. Efficient Processing of Moving Collective Spatial Keyword Queries. *VLDB J.* 29, 4 (2020), 841–865.

[37] Junye Yang, Yong Zhang, Xiaofang Zhou, Jin Wang, Huiqi Hu, and Chunxiao Xing. 2019. A Hierarchical Framework for Top-k Location-aware Error-tolerant Keyword Search. In *ICDE*. 986–997.

[38] Chengyuan Zhang, Ying Zhang, Wenjie Zhang, Xuemin Lin, Muhammad Aamir Cheema, and Xiaoyang Wang. 2014. Diversified Spatial Keyword Search on Road Networks. In *EDBT*. 367–378.

[39] Mengxuan Zhang, Lei Li, Wen Hua, Rui Mao, Pingfu Chao, and Xiaofang Zhou. 2021. Dynamic Hub Labeling for Road Networks. In *ICDE*. 336–347.

[40] Mengxuan Zhang, Lei Li, Wen Hua, and Xiaofang Zhou. 2021. Efficient 2-Hop Labeling Maintenance in Dynamic Small-world Networks. In *ICDE*. 133–144.

[41] Mengxuan Zhang, Lei Li, and Xiaofang Zhou. 2021. An Experimental Evaluation and Guideline for Path Finding in Weighted Dynamic Network. *Proc. VLDB Endow.* 14, 11 (2021), 2127–2140.

[42] Jingwen Zhao, Yunjun Gao, Gang Chen, and Rui Chen. 2018. Towards Efficient Framework for Time-aware Spatial Keyword Queries on Road Networks. *ACM Trans. Inf. Syst.* 36, 3 (2018), 24:1–24:48.

[43] Jingwen Zhao, Yunjun Gao, Gang Chen, and Rui Chen. 2018. Why-not Questions on Top-k Geo-social Keyword Queries in Road Networks. In *ICDE*. 965–976.

[44] Jingwen Zhao, Yunjun Gao, Gang Chen, Christian S. Jensen, Rui Chen, and Deng Cai. 2017. Reverse Top-k Geo-social Keyword Queries in Road Networks. In *ICDE*. 387–398.

[45] Bolong Zheng, Kai Zheng, Christian S. Jensen, Nguyen Quoc Viet Hung, Han Su, Guohui Li, and Xiaofang Zhou. 2020. Answering Why-not Group Spatial Keyword Queries. *IEEE Trans. Knowl. Data Eng.* 32, 1 (2020), 26–39.

[46] Bolong Zheng, Kai Zheng, Xiaokui Xiao, Han Su, Hongzhi Yin, Xiaofang Zhou, and GuoHui Li. 2016. Keyword-aware Continuous kNN Query on Road Networks. In *ICDE*. 871–882.

[47] Ruicheng Zhong, Ju Fan, Guoliang Li, Kian-Lee Tan, and Lizhu Zhou. 2012. Location-aware Instant Search. In *CIKM*. 385–394.

[48] Xiaoling Zhou, Jianbin Qin, Chuan Xiao, Wei Wang, Xuemin Lin, and Yoshiharu Ishikawa. 2016. BEVA: An Efficient Query Processing Algorithm for Error-tolerant Autocompletion. *ACM Trans. Database Syst.* 41, 1 (2016), 5:1–5:44.

Algorithm 3: R2T Construction Algorithm

Input: a road network $G = (V, E)$
Output: R2T of the road network

1 compute the 2-hop label for each vertex $v \in V$ [24];
2 **for** *each* $u \in V$ **do**
3 **for** *each* $(v, dis(u, v)) \in \mathsf{L}(u)$ **do**
4 add$(u, dis(u, v))$ into $\mathsf{R2T}(v).\widetilde{\mathsf{L}}(v)$;
5 sort labels in each $\widetilde{\mathsf{L}}(v)$ in ascending order.
6 build a complete trie CT for G;
7 **for** *each* $v \in V$ **do**
8 build a trie T of v;
9 **for** *each node* $n \in T$ *traversed in depth-first* **do**
10 create a bitmap with $|\widetilde{\mathsf{L}}(v)|$ bits for n;
11 **for** *each* $u \in \widetilde{\mathsf{L}}(v)$ **do**
12 **if** u *containes the prefix of* n **then**
13 set the bit representing u in $n.bitmap$ to 1;
14 **else**
15 set the bit representing u in $n.bitmap$ to 0;
16 **if** n *is a leaf node* **then**
17 add n to $\mathsf{R2T}(v).\mathsf{T}(v)$;
18 find the correspoding $\mathtt{ID}(n)$ in CT;
19 backtrack n to the root and compress bitmaps;
20 add compressed bitmaps to $\mathsf{R2T}(v).\mathsf{T}(v).\mathsf{B}(n)$;
21 **return** R2T;

Algorithm 4: R2T maintenance Algorithm For Keyword Insertion

Input: a vertex v, a keyword kw, the 2-hop Label and R2T of the road network
Output: updated R2T

1 $\mathsf{doc}(v).insert(kw)$;
2 **for** *each* $(v', d) \in \mathsf{L}(v)$ **do**
3 $T \leftarrow$ the trie of v';
4 **if** kw *is a new keyword in* T **then**
5 create new nodes for kw in T;
6 $n \leftarrow$ the leaf node of kw in T;
7 $path \leftarrow T.match(kw)$;
8 $pos \leftarrow$ the position of v in $\widetilde{\mathsf{L}}(v')$;
9 **for** *each node* $n' \in path$ **do**
10 **if** n' *is not newly added* **then**
11 **if** $n'.at(pos) == 0$ **then**
12 $n'.bitmap.setOne(pos)$;
13 **for** *each node* $n'' \in n'.childNodes$ **do**
14 insert a "0" bit into $n''.bitmap$;
15 $pos = n'.bitmap.countOne(pos)$;
16 **else**
17 create a new bitmap for n';
18 **if** n *is newly added* **then**
19 add node n and $\mathsf{B}(n)$ into $\mathsf{R2T}(v').\mathsf{T}(v')$;
20 update other bitmaps affected;
21 **return** R2T;

APPENDIX

A R2T CONSTRUCTION ALGORITHM

Algorithm 3 shows the pseudocode of R2T construction. Given a road network G, the construction of $\mathsf{R2T}(v)$ for every vertex $v \in V$ includes two parts, i.e., the constructions of $\widetilde{\mathsf{L}}(v)$ and $\mathsf{T}(v)$. First, Algorithm 3 uses existing 2-hop labeling algorithms [24] to compute the 2-hop label for every vertex (line 1). Then, Algorithm 3 traverses the 2-hop label to get the reverse 2-hop label. Specifically, if $(v, \mathrm{dis}(u, v)) \in \mathsf{L}(u)$, Algorithm 3 adds the pair $(u, \mathrm{dis}(u, v))$ to $\widetilde{\mathsf{L}}(v)$ (lines 2-4). Finally, Algorithm 3 sorts the resulting pairs in $\widetilde{\mathsf{L}}(v)$ in ascending order of the distances $\mathrm{dis}(u, v)$ (line 5). The reverse 2-hop label $\widetilde{\mathsf{L}}(v)$ for each vertex v is created. Next, Algorithm 3 constructs a complete trie for all the keywords contained in the given road network (line 6). For each vertex v, $\mathsf{T}(v)$ is stored in the form of a node array. Therefore, this step focuses on how to construct the node array. First, Algorithm 3 constructs a trie for all the keywords contained in the vertices of $\widetilde{\mathsf{L}}(v)$ (we call it the trie of v for short) (line 8). Then, Algorithm 3 computes the bitmaps for each node of the trie of v. Specifically, Algorithm 3 traverses the trie in a depth-first manner (line 9). For each visited trie node n, a bitmap with $|\widetilde{\mathsf{L}}(v)|$ bits is created for n (lines 10). To be specific, for each $(u, \mathrm{dis}(u, v)) \in \widetilde{\mathsf{L}}(v)$, if the keywords of u contain the prefix represented by n, the bit representing u in the bitmap is set to "1" (line 13). Otherwise, the bit is set to "0" (line 15). In addition, if the visited node n is a leaf node, Algorithm 3 needs to perform the following operations (lines 16-20): (1) add n to the node array; (2) find the corresponding $\mathtt{ID}(n)$ in the complete trie; (3) backtrack to the root node and compress the bitmap in a bottom-up manner (Section 5.1); and (4) add the compressed bitmap to $\mathsf{B}(n)$.

B R2T MAINTENANCE ALGORITHMS

Keyword Insertion. Algorithm 4 shows the pseudocode for R2T maintenance for keyword insertion. First, a new keyword kw is inserted into $\mathsf{doc}(v)$ (line 1). Next, Algorithm 4 finds the leaf node n representing the keyword kw in the trie of v' (line 6). If the trie does not contain such a node, Algorithm 4 inserts a new leaf node n into the trie to represent kw (lines 4-5). Then, the path of the keyword and the position of v are recorded (lines 7-8). Second, Algorithm 4 updates the bitmaps of the nodes along the path from the root node to n. Specifically, for the existing nodes that denote the prefix of kw, Algorithm 4 sets the bit representing v to "1" (lines 10-12). The bitmaps of their children nodes need to be checked to see if they need to add a "0" bit (lines 13-14). This is because when the bitmap of the father node has one more "1", the bitmaps of all its children will also have one more bit. Then, the position of v in the bitmap is updated (line 15). For newly added nodes, Algorithm 4 creates new bitmaps and the corresponding compressed bitmaps according to its father node, following the compression rule in Section 5.1 (line 17). Finally, in $\mathsf{T}(v')$, if the leaf node n is newly added, Algorithm 4 adds it to $\mathsf{T}(v')$ (lines 18-19). Thereafter, Algorithm 4 only needs to update the $\mathsf{B}(n')$ of leaf nodes n' whose representing keywords have the same prefix as kw (line 20).

Keyword Deletion. Algorithm 5 shows the pseudocode for R2T maintenance for keyword deletion. First, the keyword kw is deleted from $\mathsf{doc}(v)$ (line 1). For the nodes in the trie of v' that denote the prefix of kw, if there are other keywords in $\mathsf{doc}(v)$ with the same prefix, the bitmaps of such nodes do not need to be modified (lines 8-10). Algorithm 5 identifies the node with the smallest depth for

Algorithm 5: R2T maintenance Algorithm For Keyword Deletion

Input: a vertex v, a keyword kw, the 2-hop Label and R2T of the road network
Output: updated R2T

```
1  doc(v).delete(kw);
2  for each (v', d) ∈ L(v) do
3      T ← the trie of v';
4      n ← the leaf node of kw in T;
5      path ← T.match(kw);
6      pos ← the position of v in L̃(v');
7      isSet = false;
8      for each node n' ∈ path do
9          if ∃kw' ∈ doc(v) ∧ kw' ≠ kw ∧ kw' has the prefix
             of n' then
10             pos = n'.bitmap.countOne(pos);
11             continue;
12         if isSet==false then
13             n'.bitmap.setZero(pos);
14             isSet = true;
15         pos = n'.bitmap.countOne(pos);
16         for each node n'' ∈ n'.childNodes do
17             n''.bitmap.delete(pos);
18     if n.bitmap == 0 then
19         delete node n and B(n) from R2T(v').T(v');
20     update other bitmaps affected;
21 return R2T;
```

which no other keyword in $\mathtt{doc}(v)$ contains the prefix represented by the node. Then, Algorithm 5 sets the bit in its bitmap representing v to "0" (lines 12-14). Bits representing v in the descendants of the node are deleted (lines 16-17). Next, in $\mathtt{T}(v')$, Algorithm 5 deletes the leaf node and its $(\mathtt{ID}(n), \mathtt{B}(n))$ if no other vertices in $\widetilde{\mathtt{L}}(v')$ contain kw (lines 18-19). Additionally, for the leaf nodes n' whose representing keywords have the same prefix as kw, we update their $\mathtt{B}(n')$ as the first step (line 20).

Label Insertion. Algorithm 6 shows the pseudocode for label insertion maintenance. A new pair $(u, \mathrm{dis}(u, v))$ is added to $\widetilde{\mathtt{L}}(v)$. Algorithm 6 first inserts the keywords of u into the trie of v one by one (lines 4-6). Correspondingly, Algorithm 6 inserts the nodes representing the inserted keyword into $\mathtt{T}(v)$ (lines 8-9). Then, Algorithm 6 updates the bitmaps of $\mathtt{T}(v)$ by traversing the trie of v in a depth-first manner. If the visited trie node is newly inserted, Algorithm 6 computes its bitmap and inserts it into $\mathtt{T}(v)$ (lines 15-19). Otherwise, Algorithm 6 just needs to update the existing bitmap as follows. (i) If u's keywords contain the prefix represented by the visited trie node n, Algorithm 6 adds a "1" bit into n's bitmap to represent u, and then visits n's child node (lines 21-24). (ii) If u's keywords do not contain the prefix represented by the visited trie node n, Algorithm 6 inserts a "0" bit into n's bitmap to represent u, and stops visiting n's child node (line 26). After traversing the trie, all the bitmaps can be updated.

Label Deletion. Algorithm 7 shows the pseudocode for the maintenance of label deletion. A pair $(u, \mathrm{dis}(u, v))$ is deleted from $\widetilde{\mathtt{L}}(v)$. Algorithm 7 first deletes all the nodes that represent the unique keywords of u and their bitmaps from $\mathtt{T}(v)$ (lines 4-7). Similarly, by

Algorithm 6: R2T maintenance Algorithm For Label Insertion

Input: a label (u, d), $\mathtt{R2T}(v)$
Output: updated $\mathtt{R2T}(v)$

```
1  R2T(v).insert(u, d);
2  doc(u).insert(kw);
3  T ← the trie of v;
4  for each kw ∈ doc(u) do
5      if kw is a new keyword in T then
6          create new nodes for kw in T;
7      n ← the leaf node of kw in T;
8      if n is newly added then
9          add n into R2T(v').T(v);
10 initialize a stack st = ∅;
11 st.push(T.root);
12 while st is not empty do
13     n = st.top();
14     st.pop();
15     if n is newly added then
16         create a new bitmap for n;
17         add the bitmap into R2T(v).T(v).B(n);
18         for each node n' ∈ n.childNodes do
19             st.push(n');
20     else
21         if u contains the prefix of n then
22             insert a "1" bit into n.bitmap;
23             for each node n' ∈ n.childNodes do
24                 st.push(n');
25         else
26             insert a "0" bit into n.bitmap;
27 return R2T(v);
```

traversing the trie of v in a depth-first manner, Algorithm 7 updates the bitmaps of $\mathtt{T}(v)$. (i) If the prefix represented by the visited trie node n is contained in u's keywords, Algorithm 7 should delete the "1" bit of u from n's bitmap and then visit n's child node (lines 13-16). (ii) If the prefix represented by the visited trie node n is not contained in u's keywords, Algorithm 7 should delete the "0" bit of u from n's bitmap and stop visiting n's child node (line 18).

C INSTANT UPDATE ALGORITHM

IUA processes the query based on previous query information, rather than querying from scratch. Algorithm 8 shows the pseudocode of IUA. Specifically, Algorithm 8 first identifies the first character that differs between the old and new query strings (line 1). Next, Algorithm 8 rolls back the query state by several steps, which means that $\mathcal{R}$, C and BTags will change to a state where the query string is the first pos of $q.str$ (lines 2). Then, Algorithm 8 gets the active nodes of $q.str'$ and updates $q.str$ to $q.str'$ (line 3). For each element in C, if it has a keyword with a PED less than tau for $q.str'$, Algorithm 8 need recomputed its score; otherwise, this element should be removed, and the next vertex with minimal score in the source $\widetilde{\mathtt{L}}(v)$ of this element should be added into C (lines 4-10). For each vertex in R, if it has a keyword with a PED less than τ for $q.str'$, its score will be recomputed and it will be added into C again (lines 11-13). For each node in AN', it will inherit BTags of

Algorithm 7: R2T maintenance Algorithm For Label Deletion

Input: a label (u, d), $\texttt{R2T}(v)$
Output: updated $\texttt{R2T}(v)$

```
1  R2T(v).delete(u, d);
2  doc(u).delete(kw);
3  T ← the trie of v;
4  for each kw ∈ doc(u) do
5      n ← the leaf node of kw in T;
6      if not ∃u′ ∈ L̃(v) ∧ u′ contains the prefix of n then
7          delete n and its bitmaps from R2T(v).T(v′);
8  initialize a stack st = ∅;
9  st.push(T.root);
10 while st is not empty do
11     n = st.top();
12     st.pop();
13     if u contains the prefix of n then
14         delete the "1" bit from n.bitmap;
15         for each node n′ ∈ n.childNodes do
16             st.push(n′);
17     else
18         delete the "0" bit from n.bitmap;
19 return R2T(v);
```

Algorithm 8: Instant Update Algorithm (IUA)

Input: query $q = (q.loc, q.str)$, the new query string $q.str'$, parameters k and τ, a trie T of G
Output: a set $\mathcal{R}$ of k geo-textual objects

```
1  pos ← the position first distinct character between q.str
     and q.str′;
2  Rollback the R,C and BTags (|q.str| − pos) steps;
3  AN′ ← find active nodes of T for q.str′ [13]; q.str = q.str′;
4  for each (v′, v, score(q, v′)) ∈ C do
5      if v′ conforms to the constraints of q.str′. then
6          recompute score(q, v′);
7      else
8          C ← C − (v′, v, score(q, v′));
9          v″ ← MinL̃(v);  // Computing the next vertex
               with minimal score using Algorithm 2
10         C ← C ∪ (v″, v, score(q, v″));
11 for each v′ ∈ R do
12     if v′ conforms to the constraints of q.str′. then
13         C ← C ∪ (v′, −, score(q, v′));
14 for each an′ ∈ AN′ do
15     an ← the ancestor node of an′ in AN;
16     Let an′ inherit BTags of an;
17 R = ∅; AN = AN′;
18 Algorithm 1 lines 9-15;
19 return R;
```

its ancestor node in AN (lines 14-16). Then, Algorithm 8 clears $\mathcal{R}$ and updates the active node set AN (line 17). Finally, Algorithm 8 can continue the query with lines 9-15 in Algorithm 1.

Algorithm 9: MinL̃(v) for multiple query strings

Input: a vector AN of active nodes, $q.str$, $\texttt{R2T}(v)$ index of a vertex v
Output: a vertex v' with the minimal score in $\widetilde{\texttt{L}}(v)$

```
1  v′ ← ∅; MinScore ← +∞;
2  if BTag[v] = ∅ then
3      initialize BTag[v];
4  lastAN = AN[|AN| − 1];
5  for each active node an ∈ lastAN do
6      find a leaf node n of an using binary search;
7      v″ = -1;
8      while v″ does not conform to the constraints of q.str do
9          Algorithm 2 lines 6-14;
10     if score(q, v″) < MinScore then
11         v′ ← v″;
12         MinScore ← score(q, v″);
13 return v′;
```

D MULTIPLE QUERY STRINGS

For query strings with more than one prefix, our approach uses the same main query process as with single query strings (as described in Algorithm 1). However, we need to modify Algorithm 2 to find the minimum vertex score in $\widetilde{\texttt{L}}(v)$. Algorithm 9 shows the pseudocode of modified Algorithm 2. To be specific, the input AN is a vector that stores active nodes for each string in $q.str$. Algorithm 9 initializes BTag and retrieves the set of active nodes corresponding to the last string in $q.str$ (lines 2-4). For each active node, Algorithm 9 finds the leaf node containing the bitmap of the active node using binary search (line 6) and initializes a vertex whose ID is -1 (line 7). Then, Algorithm 9 continues to find the next vertex until v'' meets the constraints of all query strings in $q.str$ (lines 8-9). Specifically, for each string in $q.str$, there should be a keyword of v'' whose PED with the string is less than or equal to τ. Finally, Algorithm 9 finds the minimal score vertices of all active nodes and returns the one with the smallest score among all active nodes (lines 11-13). It is worth noting that when calculating the score $\texttt{score}(q, v'')$, the textual similarity is determined as the sum of the minimal prefix edit distance for all query strings.

E RUNNING EXAMPLE OF INDEX CONSTRUCTION

Take the construction of $\texttt{R2T}(v_6)$ in Figure 6 as an example. We first traverse the root node n'_1 of trie in Figure 5, and compute n'_1's bitmap and compressed bitmap, which are "1111" and "1111", respectively. In the same manner, we visit the nodes n'_2, n'_3, n'_4, and n'_5, and compute the corresponding bitmaps and compressed bitmaps. Since n'_5 is a leaf node, we backtrack to the root node, and get $\texttt{B}(n'_5) = \{0110, 01\}$. The bitmap of n'_4 is not added to $\texttt{B}(n'_5)$ as it is the same as that of n'_5. Then, we add n'_5, $\texttt{ID}(n'_5) = n_7$, and $\texttt{B}(n'_5) = \{0110, 01\}$ to the node array. In the same way, we can traverse the whole trie, and get the complete $\texttt{T}(v_6)$ as depicted in Figure 6.

F RUNNING EXAMPLE OF ALGORITHM 1

Example F.1. *We illustrate Algorithm 1 using the road network in Figure 2. Let* $q = (v_9, \text{"}sto\text{"})$, $k = 3$, $\tau = 1$, *and* $\alpha = 0.5$. *Table 4 shows*

the details of the query processing. In Table 4, it is worth mentioning that (1) "−" means that the reverse 2-hop label does not have vertices matching "sto"; (2) the number next to vertex indicates the vertex's score; and (3) in each row, we mark the vertex with the minimal score in red color. In the Initialization *row, we find the vertices with the minimal score for $\widetilde{\mathsf{L}}(v_6)$, $\widetilde{\mathsf{L}}(v_4)$, and $\widetilde{\mathsf{L}}(v_1)$, which are v_7, v_4, and v_1, respectively. In the first round, v_4 of $\widetilde{\mathsf{L}}(v_4)$ has the minimum score among all reverse 2-hop labels, and it is added to the result set. Then, we should find the next vertex having the minimal score in $\widetilde{\mathsf{L}}(v_4)$, which is v_5. In the second round, v_4 of $\widetilde{\mathsf{L}}(v_1)$ has the minimal score. But, v_4 has been in the result set. We only need to find the next vertex with the minimal score in $\widetilde{\mathsf{L}}(v_1)$, which is v_1. In the same way, we find the vertices v_7 and v_5 in the following two rounds. In the end, we get the final result set $\{v_4, v_7, v_5\}$.*

G RUNNING EXAMPLE OF ALGORITHM 2

Example G.1. *We employ the computation of the vertex with the minimal score in $\widetilde{\mathsf{L}}(v_6)$ for the active node n_{36} (i.e., the column $\widetilde{\mathsf{L}}(v_6)$ in Table 4) to illustrate Algorithm 2. In Table 4, Algorithm 2 finds the vertex with the minimal score twice, i.e., in the rows of* initialization *and* Select v_7 of $\widetilde{\mathsf{L}}(v_6)$. *Table 5 shows the details of* BTag *for the whole processing. For the active node n_{36}, we need to traverse two bitmaps, i.e., "1100" and "01". At the initialization step, the* BTag*s of "1100" and "01" are both $\langle 0, 0\rangle$. When finding the first minimal score vertex, the* BTag*s of "1100" and "01" are $\langle 2, 2\rangle$ and $\langle 2, 1\rangle$, respectively, meaning that the second vertex in $\widetilde{\mathsf{L}}(v_6)$ is the result. Thus, v_7 is found. When finding the second minimal score vertex, the* BTag*s of "1100" and "01" are $\langle -, -\rangle$ and $\langle -, 2\rangle$, respectively, indicating that no qualified vertex exists. Therefore, the result is empty.*

H THE PROOF OF THEOREM 5.1

Theorem 5.1. The space complexity of R2T is $O(w \cdot |V| \cdot \log|V|)$, where w is the shortest path tree width of the road network..

Proof. R2T consists of the reverse 2-hop label and the node array. The reverse 2-hop label is constructed from the 2-hop label. It has been proved that the space complexity of the 2-hop label is $O(w \cdot |V| \cdot \log|V|)$ [42]. Thus, R2T also requires $O(w \cdot |V| \cdot \log|V|)$ space to store the reverse 2-hop label.

Next, we analyze the space complexity of the node array. Let y be the average word length. For each node $v \in V$, the total number of nodes in the trie of v is $O(y \cdot \frac{|\mathrm{doc}(V)|}{|V|} \cdot w \cdot \log|V|)$. Each node has a bitmap. The number of bits in the bitmap is equal to the number of labels in the reverse 2-hop label. Before compressing bitmaps, it takes $O(y \cdot \frac{|\mathrm{doc}(V)|}{|V|} \cdot w \cdot \log|V| + y \cdot \frac{|\mathrm{doc}(V)|}{|V|} \cdot \log|V| \cdot \log|V| \cdot w^2) = O(w^2 \cdot \log^2|V|)$ space to store the trie and all bitmaps. Let x be the average number of child of a node in the trie. Assumed that the words of vertices are evenly distributed on the prefix tree. The bitmap of node in level i has $O(\frac{w \cdot \log|V|}{x^{i-1}})$ bits. There are x^i nodes in the i-th layer. After compressing the bitmaps, the space cost is $O(|V| \cdot (w \cdot \log|V| + \sum_{i=1}^{n-1} x^i \cdot \frac{w \cdot \log|V|}{x^{i-1}})) = O(w \cdot |V| \cdot \log|V|)$.

Totally, the space complexity of R2T is $O(w \cdot |V| \cdot \log|V|)$. □

I THE PROOF OF THEOREM 5.2

Theorem 5.2. The time and space complexities of R2T construction are $O(w \cdot |E| \cdot \log|V| + w^2 \cdot |V| \cdot \log^3|V|)$ and $O(w^2 \cdot |V| \cdot \log^2|V|)$, respectively.

Proof. R2T construction has two stages, i.e., the constructions of the reverse 2-hop label and the node array. The reverse 2-hop label construction should compute the 2-hop label, whose time complexity is $O(w \cdot |E| \cdot \log|V| + w^2 \cdot |V| \cdot \log^3|V|)$ [42]. For a vertex $v \in V$, v's node array $\mathsf{T}(v)$ construction includes bitmaps calculation and compression. When calculating a bitmap, we should set a certain bit of the bitmap to "1", which takes $O(1)$ time. In the proof of Theorem 5.1, we have showed that there are $O(y \cdot \frac{|\mathrm{doc}(V)|}{|V|} \cdot w \cdot \log|V|)$ bitmaps. Hence, it needs $O(y \cdot \frac{|\mathrm{doc}(V)|}{|V|} \cdot w \cdot \log|V|) = O(w \cdot \log|V|)$ time to calculate all bitmaps. When compressing a bitmap, we should check each bit and the bitmap of its father node. Since the number of bits is equal to the number of labels in reverse 2-hop label, it takes $O(w \cdot \log|V|) \cdot O(y \cdot \frac{|\mathrm{doc}(V)|}{|V|} \cdot w \cdot \log|V|) = O(w^2 \cdot \log^2|V|)$ time to compress all bitmaps. Therefore, the total time complexity of R2T construction is $O(|V| \cdot (w \cdot \log|V| + w^2 \cdot \log^2|V|)) = O(w^2 \cdot |V| \cdot \log^2|V|)$. Since $O(w^2 \cdot |V| \cdot \log^2|V|) \ll O(w \cdot |E| \cdot \log|V| + w^2 \cdot |V| \cdot \log^3|V|)$, the time complexity of R2T construction is $O(w \cdot |E| \cdot \log|V| + w^2 \cdot |V| \cdot \log^3|V|)$.

The space overhead of R2T construction is the storage of uncompressed bitmaps. Thus, the space complexity of R2T construction is $O(|V| \cdot y \cdot \frac{|\mathrm{doc}(V)|}{|V|} \cdot \log|V| \cdot \log|V| \cdot w^2) = O(w^2 \cdot |V| \cdot \log^2|V|)$. □

J THE PROOF OF LEMMA 6.1

Lemma 6.1. Given a vertex $v \in G$,

$$\bigcup_{v' \in \mathsf{L}(v)} \widetilde{\mathsf{L}}(v') = V \tag{2}$$

Proof. We prove it by contradiction. Given a vertex $v \in G$. Assume that $\bigcup_{v' \in \mathsf{L}(v)} \widetilde{\mathsf{L}}(v') \neq V$. Then, there exists a vertex $u \in V$ and $u \notin \bigcup_{v' \in \mathsf{L}(v)} \widetilde{\mathsf{L}}(v')$. According to the definition of reverse 2-hop label, for any vertex $p \in \mathsf{L}(u)$, $u \in \widetilde{\mathsf{L}}(p)$. Since $u \notin \bigcup_{v' \in \mathsf{L}(v)} \widetilde{\mathsf{L}}(v')$, $p \notin \mathsf{L}(v)$, i.e. $\mathsf{L}(v) \cap \mathsf{L}(u) = \varnothing$. This contradicts the property of 2-hop label and the assumption does not hold. The proof is completed. □

K THE PROOF OF LEMMA 6.2

Lemma 6.2. Given a vertex $v \in G$, for a vertex $v' \in \mathsf{L}(v)$, let $v'' = \arg\min_{v'' \in \widetilde{\mathsf{L}}(v')} dis(v', v'')$. Note that $v'' \neq v$. Then, the nearest neighbor of v is

$$v'' = \arg\min_{v'' \in \widetilde{\mathsf{L}}(v'), v' \in \mathsf{L}(v)} dis(v, v') + dis(v', v'') \tag{3}$$

Proof. Let v'' be the nearest neighbor of v and $u \in G$. According to Lemma 6.1 and the definition of 2-hop label, $\mathrm{dis}(v, u) = \min_{u \in \widetilde{\mathsf{L}}(p), p \in \mathsf{L}(v)} dis(v, p) + dis(p, u)$. Since v'' is the nearest neighbor of v, $v'' = \arg\min_{v'' \in \widetilde{\mathsf{L}}(v'), v' \in \mathsf{L}(v)} dis(v, v') + dis(v', v'')$.

Next, we prove $v'' = \arg\min_{v'' \in \widetilde{\mathsf{L}}(v')} dis(v', v'')$ by contradiction. Assume that $v'' \neq \arg\min_{v'' \in \widetilde{\mathsf{L}}(v')} dis(v', v'')$, meaning that there exists a vertex u such that $\mathrm{dis}(v', u) < \mathrm{dis}(v', v'')$. We have $\mathrm{dis}(v, v') + \mathrm{dis}(v', u) < \mathrm{dis}(v, v') + \mathrm{dis}(v', v'') = dis(v, v'')$. It contradicts the fact that v'' is the nearest neighbor of v. Hence, the assumption does not hold. The proof is completed. □

Table 4: Illustration of Algorithm 1

Operation	The vertex with minimum score in $\widetilde{\mathsf{L}}(v)$					Result
	$\widetilde{\mathsf{L}}(v_9)$	$\widetilde{\mathsf{L}}(v_6)$	$\widetilde{\mathsf{L}}(v_4)$	$\widetilde{\mathsf{L}}(v_8)$	$\widetilde{\mathsf{L}}(v_1)$	
Initialization	–	v_7 (0.71)	v_4 (0.25)	–	v_4 (0.46)	$\varnothing$
Select v_4 of $\widetilde{\mathsf{L}}(v_4)$	–	v_7 (0.71)	v_5 (0.79)	–	v_4 (0.46)	$\{v_4\}$
Select v_4 of $\widetilde{\mathsf{L}}(v_1)$	–	v_7 (0.71)	v_5 (0.79)	–	v_1 (0.83)	$\{v_4\}$
Select v_7 of $\widetilde{\mathsf{L}}(v_6)$	–	–	v_5 (0.79)	–	v_1 (0.83)	$\{v_4, v_7\}$
Select v_5 of $\widetilde{\mathsf{L}}(v_4)$	–	–	v_7 (0.79)	–	v_1 (0.83)	$\{v_4, v_7, v_5\}$

Table 5: Illustration of Algorithm 2

Bitmap	$\mathsf{BTag}[v_6][n_{45}]$		
	Initialization	Compute the vertex with the minimum score	Compute the second vertex with the minimum score
1100	$\langle 0, 0\rangle$	$\langle 2, 2\rangle$	$\langle -, -\rangle$
01	$\langle 0, 0\rangle$	$\langle 2, 1\rangle$	$\langle -, 2\rangle$

L THE PROOF OF COROLLARY 6.1

Corollary 6.1. Given a query $q = (q.loc, q.str)$, for a vertex $v \in \mathsf{L}(q)$, let $v' = \arg\min_{v' \in \widetilde{\mathsf{L}}(v)} \mathsf{score}(q, v')$. Then, the vertex with the minimal score w.r.t. q is

$$v' = \arg\min_{v' \in \widetilde{\mathsf{L}}(v), v \in \mathsf{L}(q)} \mathsf{score}(q, v') \quad (4)$$

Proof. Let $q \in G$ and v' be the vertex with the minimal score w.r.t q. For any $u \in G$ and $p \in \mathsf{L}(q)$, according to Lemma 6.2 and Equation(1), $\mathsf{score}(q, u) = \min_{u \in \widetilde{\mathsf{L}}(p), p \in \mathsf{L}(v)} \mathsf{score}(q, u)$. Since v' has the smallest score, $v' = \arg\min_{v' \in \widetilde{\mathsf{L}}(v), v \in \mathsf{L}(q)} \mathsf{score}(q, v')$. We prove $v' = \arg\min_{v' \in \widetilde{\mathsf{L}}(v)} \mathsf{score}(q, v')$ by contradiction. Assume that $v' \neq \arg\min_{v' \in \widetilde{\mathsf{L}}(v)} \mathsf{score}(q, v')$. There exists a vertex that has smaller score than v'. It contradicts the fact that v' is the vertex with the minimal score w.r.t q. Therefore, the assumption does not hold. The proof is completed. □

M THE PROOF OF LEMMA 6.3

Lemma 6.3. Given two trie nodes n_1 and n_2, two pairs $\langle x_1, y_1\rangle$ and $\langle x_2, y_2\rangle$, which are the BTags of the bitmaps of n_1 and n_2, respectively. Assume that n_1 is the child node of n_2. If $y_2 = x_1$, the vertex represented by the x_1-th bit in the bitmap of n_1 and the vertex represented by the x_2-th bit in the bitmap of n_2 are the same.

Proof. Let B_1 be the bitmap of n_1 and B_2 be the bitmap of n_2. According to the rule of compression, the number of bits in B_1 is equal to the number of "1" bits in B_2, i.e. the number of vertices represented by B_2 are the same as that of the vertices denoted by all "1" bits in B_1. Thus, the x_1-th bit in B_1 is the x_1-th "1" bit in B_2. As the x_2-th bit in B_2 is the y_2-th "1" bit in $B2$ and $y_2 = x_1$, the x_1-th bit in B_1 and x_2-th bit in B_2 represent the same vertex. □

N THE PROOF OF THEOREM 6.4

Theorem 6.4. The time and space complexities of IQA are $O((k + w \cdot \log|V|) \cdot |AN| \cdot \sqrt{w \cdot \log|V|})$ and $O(w \cdot \log|V| \cdot |AN| \cdot |q.str|)$, respectively. $|AN|$ denotes the number of active nodes. w is the shortest path tree width of the road network.

Proof. The time complexity of IQA consists of two parts. First, IQA finds the vertex with the minimal score for each reverse 2-hop label of $v \in \mathsf{L}(q)$ (lines 5-8 of Algorithm 1), which takes $O(w \cdot \log|V| \cdot |AN| \cdot \sqrt{w \cdot \log|V|})$ time. Specifically, IQA takes $O(|AN| \cdot \log(w \cdot \log|V|))$ time to match leaf nodes in each node array. When searching the first candidate label in a reverse 2-hop label of $v \in \mathsf{L}(q)$, IQA checks a part of these bitmaps, which takes $O(|AN| \cdot (w \cdot \log|V|)^{\frac{|q.str|}{y}})(|q.str| \leq y)$ time, where y is the average word length. Considering the average case, i.e. $|q.str| = y/2$, the expected time complexity for the first part is $O(w \cdot \log|V| \cdot |AN| \cdot \sqrt{w \cdot \log|V|})$. Second, IQA repeatedly selects the vertex with the minimal score among all reverse 2-hop labels until k vertices are found (lines 9-15 of Algorithm 1), which takes $O(k \cdot |AN| \cdot \sqrt{w \cdot \log|V|})$ time. As analyzed in the first part, searching the vertex with the minimal score for a reverse 2-hop label of $v \in \mathsf{L}(q)$ takes $O(|AN| \cdot \sqrt{w \cdot \log|V|})$ time. Since k vertices should be found, the time complexity of the second part is $O(k \cdot |AN| \cdot \sqrt{w \cdot \log|V|})$. Totally, IQA takes $O((k + w \cdot \log|V|) \cdot |AN| \cdot \sqrt{w \cdot \log|V|})$ time.

The space overhead of IQA is determined by BTags, which needs $O(w \cdot \log|V| \cdot |AN| \cdot |q.str|)$ space. Thus, the space complexity of IQA is $O(w \cdot \log|V| \cdot |AN| \cdot |q.str|)$. □

O EMPIRICAL RESULTS

This section shows the complete empirical results.

O.1 Evaluation of Instant Query Algorithm

Exp-1: Effect of $|q.str|$. We first verify the effect of the query string length $|q.str|$. We vary $|q.str|$ from 1 to 7, and fix the other parameters to default values. Figure 7(a) shows the experimental results. We can observe that the query time of K-SPIN and KT is decreased with the growth of $|q.str|$, since the longer the query string, the fewer candidate keywords. The query time of native methods (G*-tree, 2-hop, and Dijkstra) increases with the growth of $|q.str|$ while the query time of IQA ascends when $|q.str| \leq 3$. As $|q.str| > 3$, the query time of IQA almost does not change. This is because, for K-SPIN and KT, the longer $q.str$ is, the fewer the candidate keywords, and the less time it takes to query. When $|q.str| < 3$, they take a very long time to query because all the keywords in the dataset are candidate keywords at this time. For *nativemethods*, the longer $q.str$ is, the less vertices satisfy the textual constraints. Thus, the search spaces of other algorithms become larger, incurring more query time. For IQA, the number of active nodes increases with the growth of $|q.str|$ when $|q.str| \leq 3$. Since we set the default value of error threshold to 2, the number of active nodes goes down gradually when $|q.str| > 3$. Hence, the impact of the increased search space is offset, the running time of IQA almost keep the same as $|q.str| > 3$. Moreover, IQA outperforms other algorithms by 1-2 orders of magnitude.

Exp-2: Effect of $q.str$'s frequency. Next, we evaluate the effect of $q.str$'s frequency. Here, the $q.str$'s frequency is $\frac{|V(q.str)|}{|\mathsf{doc}(V)|}$, where $V(q.str) = \{v | \forall v \in V, \exists kw \in \mathsf{doc}(v), q.str \preceq kw\}$. The query time are shown in Figure 12. The query time of IQA and naive methods decrease with the increase of $q.str$'s frequency. The reason behind is that if $q.str$ is frequent in the road network, there are more vertices containing $q.str$. Thus, the results are easier to be found, resulting in less query time. Again, IQA is still much faster than these three methods by more than two orders of magnitude. However, the changing trends of K-SPIN and KT are opposite to those of other methods. This is because, in general, the higher the frequency of a

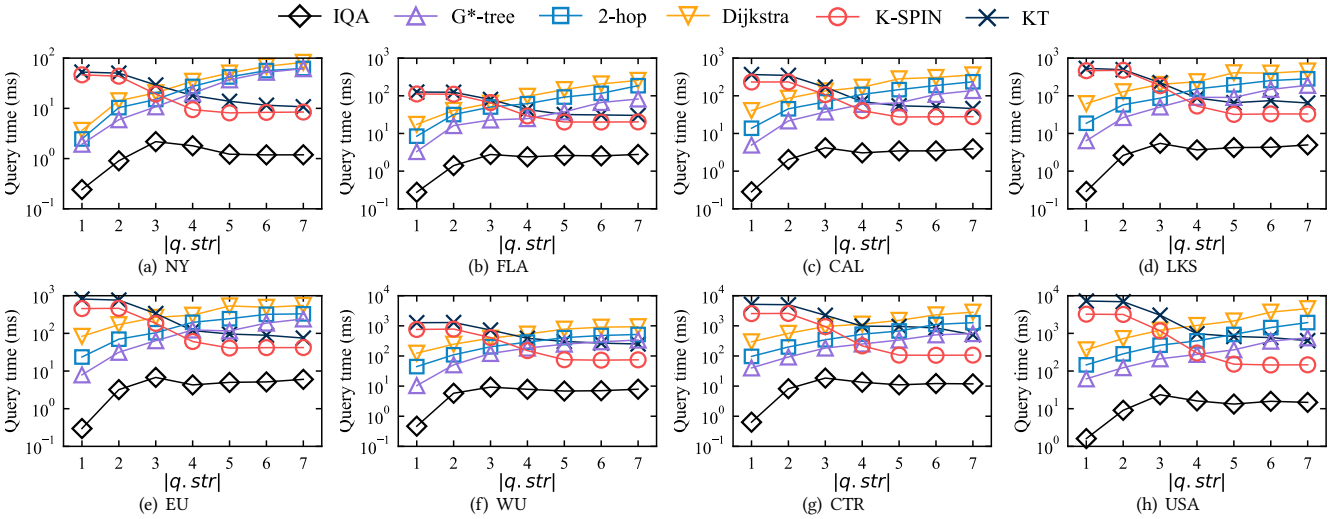


Figure 11: Effect of $|q.str|$

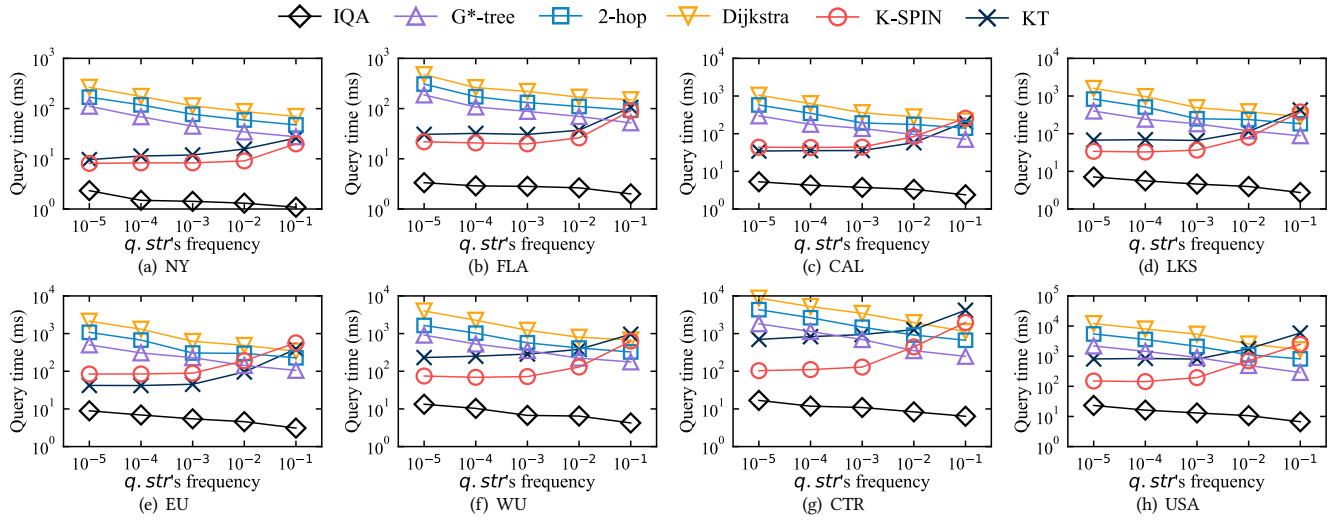


Figure 12: Effect of $q.str$'s frequency

prefix, the more candidate keywords it corresponds to. This means that K-SPIN and KT need to perform more keyword queries when $q.str$ is frequent.

Exp-3: Effect of # of $q.str$. In this experiment, we explore the effect of the number of query strings, whose value is varied from 1 to 5. The query time of six algorithms are depicted in Figure 13. Specifically, the query time of other algorithms grow with the increase of the number of query strings while IQA almost keeps the same performance. For other algorithms, if the number of query strings increases, less vertices will satisfy the textual constraint. Hence, other algorithms have to traverse more vertices to find the results. IQA finds the minimal score vertices by traversing the bitmaps of active nodes. For multiple query strings, we only need to traverse the bitmaps of active nodes for a single query string, other query strings are used to help to calculate the score. Therefore, the query time of IQA does not change much.

Exp-4: Effect of $q.str$'s edit distance. Recall that our proposed algorithms can tolerate the typos in $q.str$. Thus, we study the effect of $q.str$'s edit distance. To this end, we assume that $q.str$ includes several typos. The $q.str$'s edit distance is the edit distance between $q.str$ and the correct string. Note that, to ensure non-empty query result, we set $\tau = 4$ in this experiment. Figure 14 shows the empirical results. We can observe that when the $q.str$'s edit distance increases, the query time of IQA, K-SPIN and KT gradually drop while the query time of other algorithms gradually grows. This is because the larger the $q.str$'s edit distance, the less vertices in the road network satisfy the prefix edit distance constraint, meaning that naive methods need to search more vertices. K-SPIN and KT have

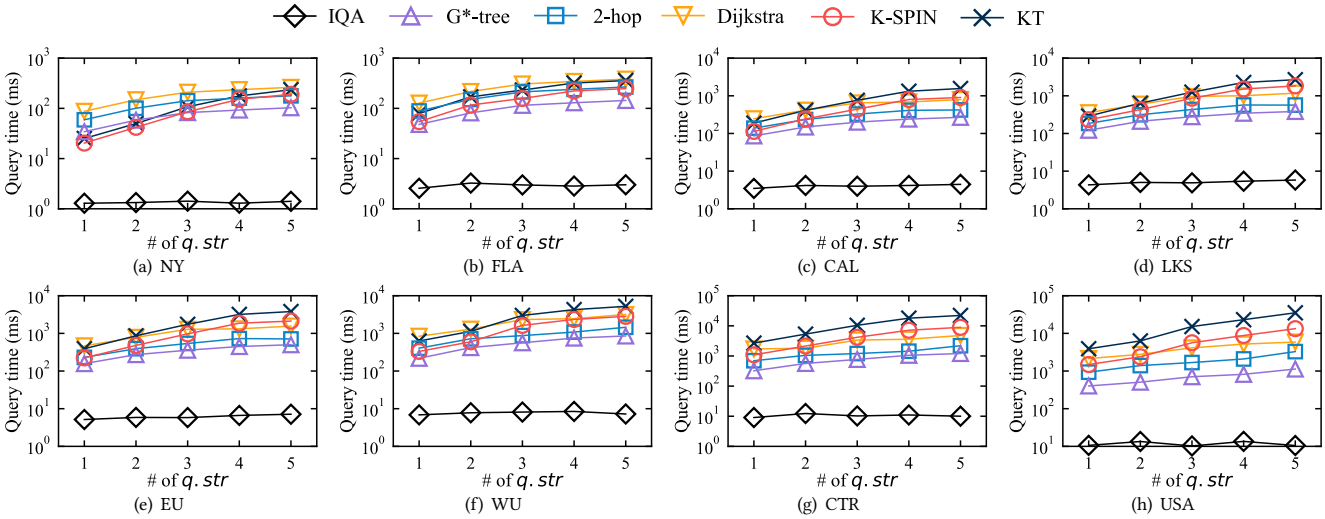


Figure 13: Effect of # of $q.str$

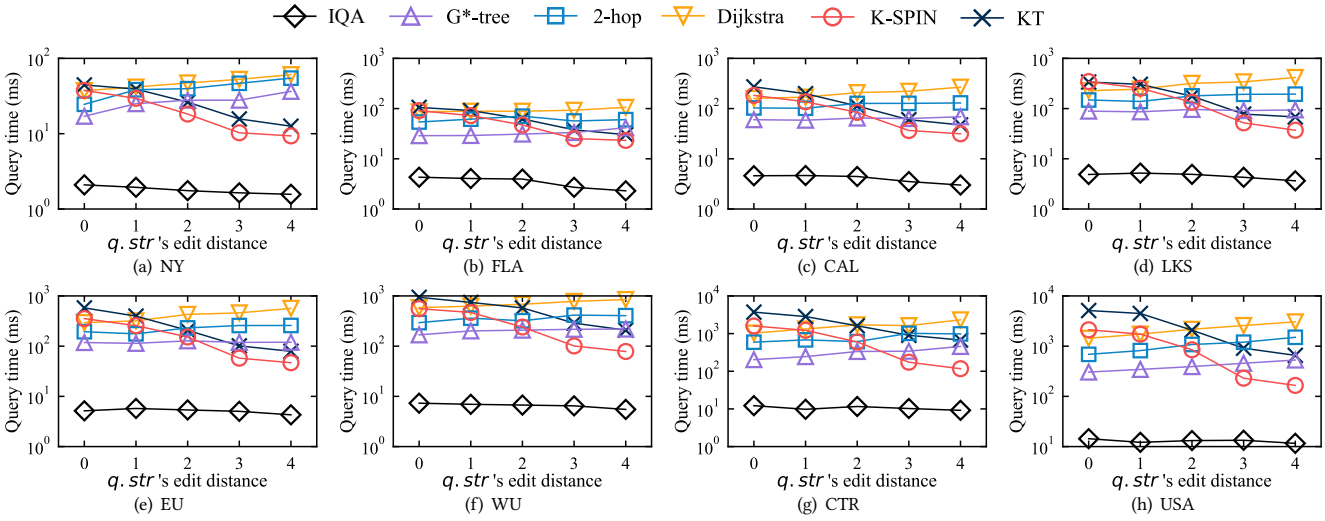


Figure 14: Effect of $q.str$'s edit distance

this opposite trend, because the larger the edit distance, the fewer the candidate keywords and the faster the query time. IQA can quickly skip invalid vertices since larger $q.str$'s edit distance leads to less active nodes.

Exp-5: Effect of k. We investigate the effect of k by varying k from 1 to 64. Figure 15 plots the query time of four algorithms. As k becomes larger, the query time of six algorithms all increase. The reason behind is that the larger k indicates more results. Hence, all algorithms take more time to query. In contrast, the rising trend of query time for K-SPIN and KT is slower. This is because they take more time to get the first node corresponding to each candidate keyword, while the subsequent search for k results takes less query time. However, the performance of IQA is still much better than that of other algorithms.

Exp-6: Effect of τ. Then, we evaluate the effect of error threshold τ on algorithms. The empirical results are reported in Figure 16. We can observe that the query time of naive methods decrease while the query time of IQA ,K-SPIN and KT increases with the growth of τ. If τ becomes larger, (1) more vertices satisfy the textual constraint, and (2) more active nodes and candidate keywords will be found. Thus, the naive methods spend less time while IQA ,K-SPIN and KT take more time. Nonetheless, IQA is still 1-2 orders of magnitude faster than other algorithms.

Exp-7: Effect of α. α represents the user preference for score computation in Equation 1. A large α value indicates that users prefer to geo-textual objects with short distances. Otherwise, users prefer to geo-textual objects that match query string better. This set of experiment test the effect of α on algorithms and the results

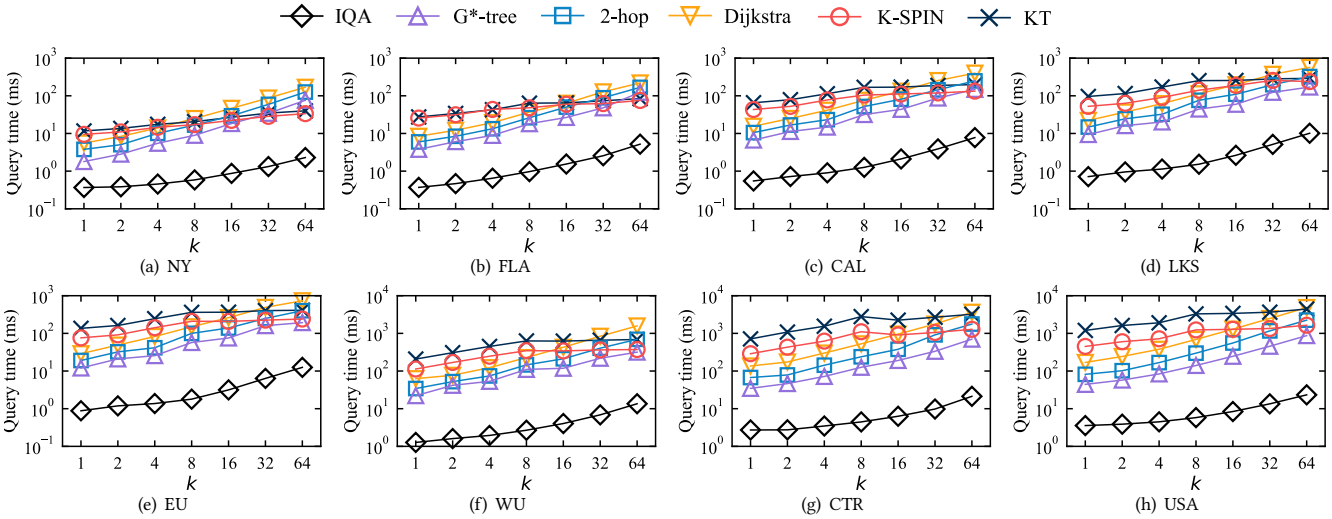


Figure 15: Effect of k

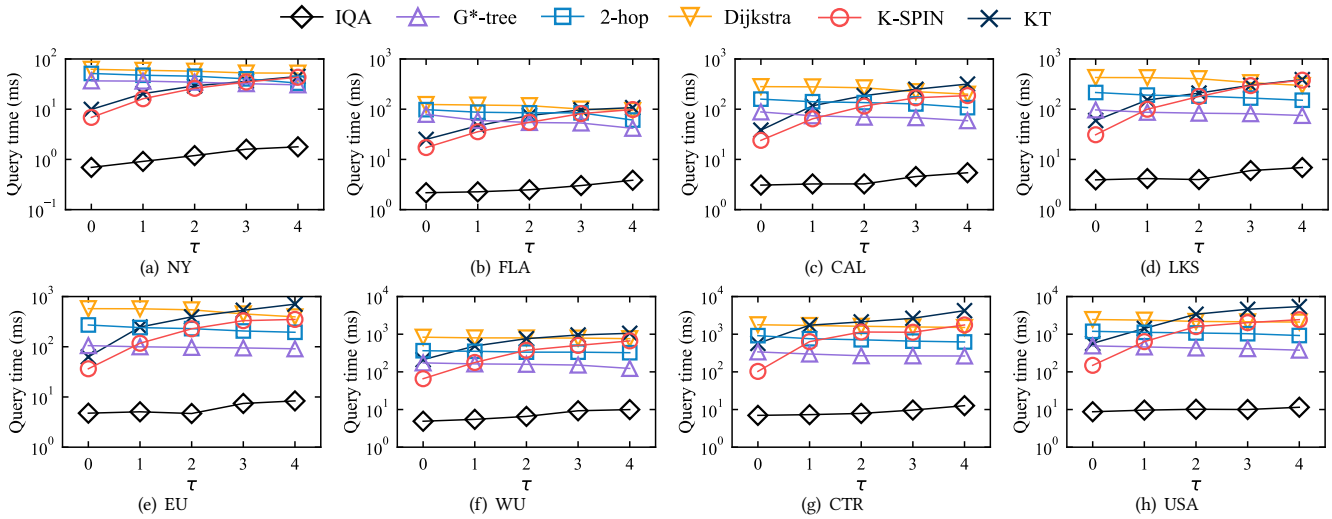


Figure 16: Effect of τ

are shown in Figure 17. We can observe that when $\alpha = 0$, other algorithms take more time. This is because the text pruning abilities of other algorithms are weak. For other values of α, the performance of all algorithms are stable, meaning that α has little effect on algorithms.

Exp-8: Scalability. In this set of experiments, we verify the scalability of the algorithms. In view of this, we vary the vertex cardinality $|V|$, the edge cardinality $|E|$, the number of distinct keywords $|W|$, and the occurrences of keywords $|\mathsf{doc}(V)|$. Figures 18(a), 18(b), 18(c), and 18(d) plot the query time by changing $|V|$, $|E|$, $|W|$, and $|\mathsf{doc}(V)|$, respectively. In Figure 18(a), with the growth of vertex cardinality, the performance of all algorithms degrade since the larger road network needs more time to find the results. In Figure 18(b), the trend for each method is similar to Figure 18(a), which is because more edges also imply a larger network size, so the query time also increases. In Figure 18(c), as the number of distinct keywords increases, the performance of four algorithms degrade as well. This is because when the number of distinct keywords grows, the keywords will become less frequent. Hence, IQA and naive methods take more time with the growth of $|W|$, which is consistent with the empirical results of $q.str$'s frequency depicted in Figure 7(b).For K-SPIN and KT, their performance is highly influenced by the number of candidate keywords. As $|W|$ increases, more candidate keywords will be found, which can have a significant impact on their performance. In Figure 18(d), as the occurrences of keywords increase, the performance of other algorithms degrade while that of IQA keeps stable. If the occurrences of keywords grow, the vertices contain more keywords. Thus, the search spaces of other algorithms become larger, incurring more query time. On the other hand, the growth of the occurrences of keywords does not affect the complete trie of the road network. Hence, the active nodes found by IQA do not change as well. Therefore, the query time of IQA keeps stable.

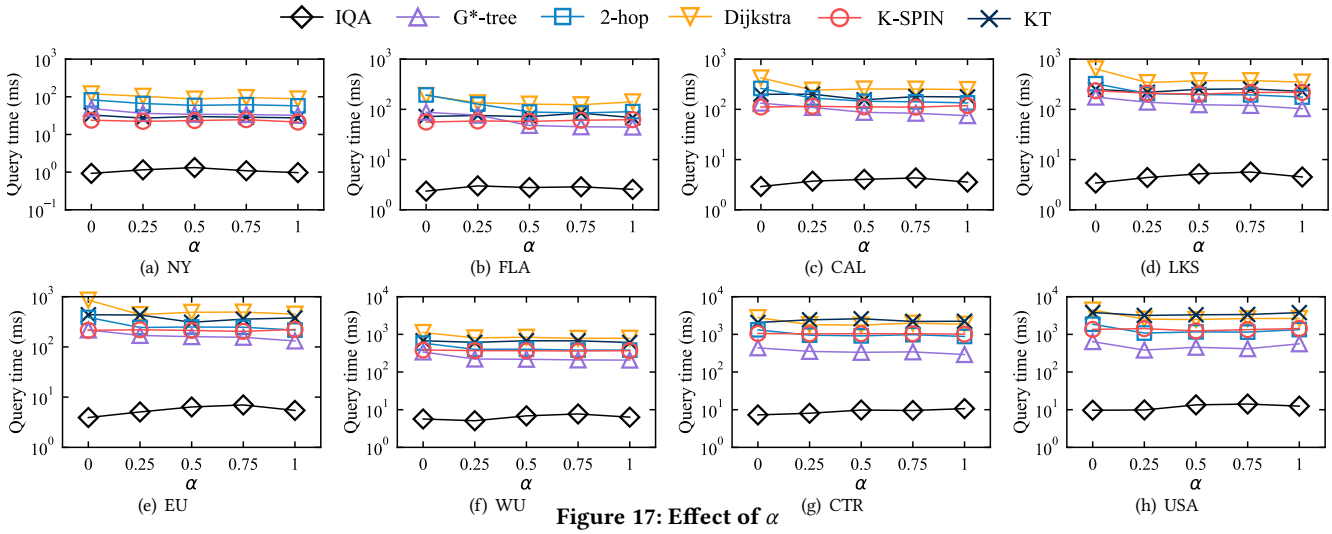


Figure 17: Effect of α

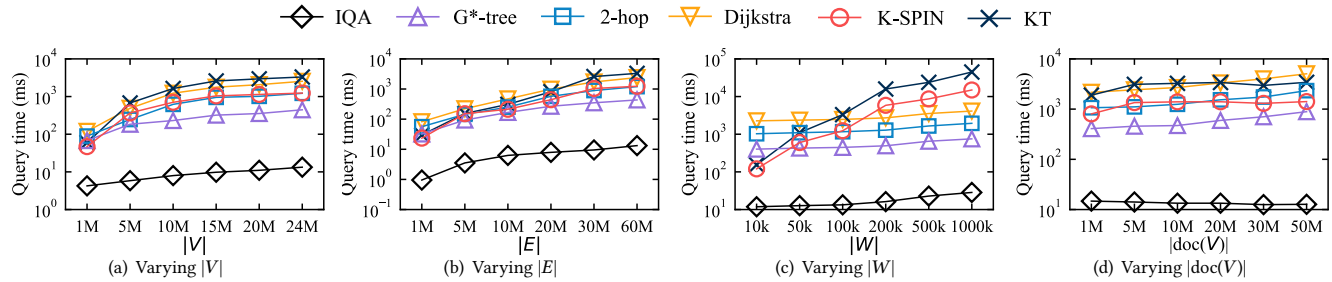


Figure 18: Effect of scalability

O.2 Evaluation of Instant Update Algorithm

In this section, we evaluate the performance of instant update algorithm. IQA, G*-tree, Dijkstra, 2-hop label, K-SPIN, and KT are instant query algorithms used for results update, i.e., to query from the scratch. IUA is the instant update algorithm presented in Section 6.2.

Exp-9: Effect of the position to insert characters. First, we explore the effect of inserting characters into different positions of $q.str$. Here, if we insert the characters into the position i of $q.str$, it means that we insert the characters after the i-th character of $q.str$. Note that, the $|q.str| \geq 7$, and we vary the position from 1 to 7. We insert one character in this experiment. Figure 19 depicts the empirical results. Specifically, the query time of all algorithms almost remain the same, except for IUA. This is because once $q.str$ changes, each of these other methods needs to start the query from scratch. Moreover, we can observe that if the insertion position is closer to the end of $q.str$, IUA has better performance. This is because the closer to the end of the query string, the less BTags will be updated. Thus, IUA needs less time to update. In addition, IUA has better performance compared with other algorithms. Specifically, IUA returns results in an average time of 2.1ms while IQA takes 10ms.

Exp-10: Effect of # of inserted characters. Next, we investigate the effect of inserting different number of characters into $q.str$. To this end, we extract characters from a complete keyword, and take the remaining string as $q.str$. Then, in experiments, we insert the extracted characters into $q.str$. In this way, we ensure the query result is non-empty. The query time of all algorithms are shown in Figure 20. When the number of inserted characters increases, IQA and naive methods take more query time. This is because when we insert more characters, $q.str$ becomes more accurate. They should traverse more vertices to find the results, and thus need more time to query. The more characters inserted, the fewer the candidate keywords, so the performance of K-SPIN and KT can be improved. For IUA, if we insert more characters, the difference between the original query string and new query string is greater. Hence, IUA has to need more time to update BTags. Although IUA becomes less efficient when more characters are inserted, it is still better than IQA, and is 2 orders of magnitude faster than other algorithms.

Exp-11: Effect of the position to delete characters. In this experiment, we study the instant update algorithm by deleting characters from different positions of the query string. We vary the deletion position from 1 to 7. Here, the deletion position i means that the i-th character of $q.str$ is deleted. Figure 21 depicts the empirical results. Specifically, the query time of all algorithms almost remain the same, except for IUA. As observed, the closer the deletion position is to the end of $q.str$, the better performance of IUA. The reason behind is similar with that of Exp-9, i.e., when

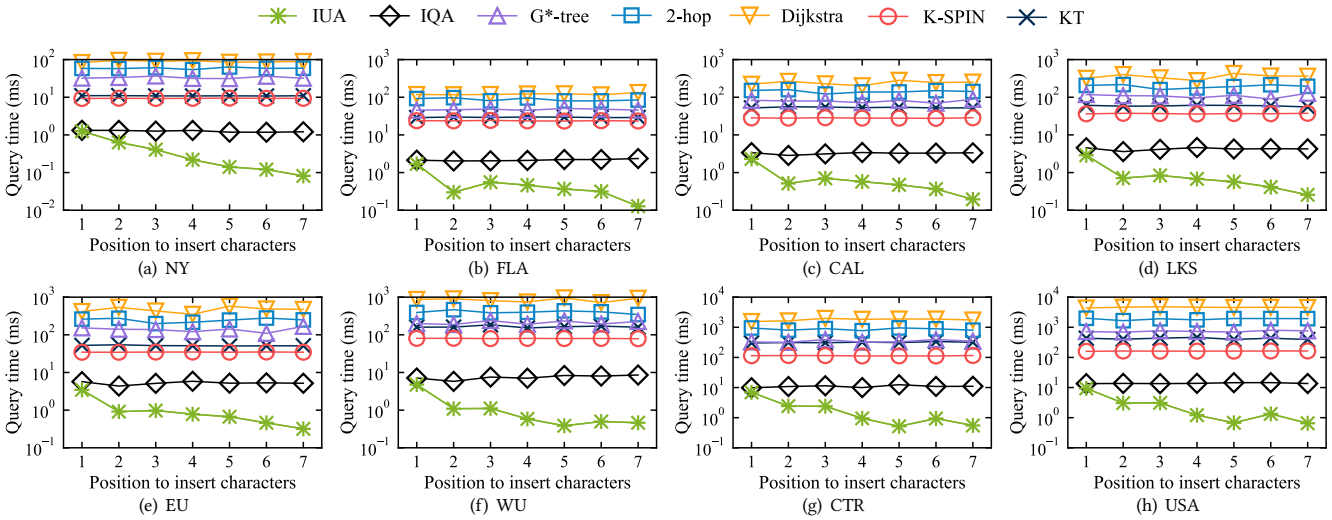


Figure 19: Position to insert characters

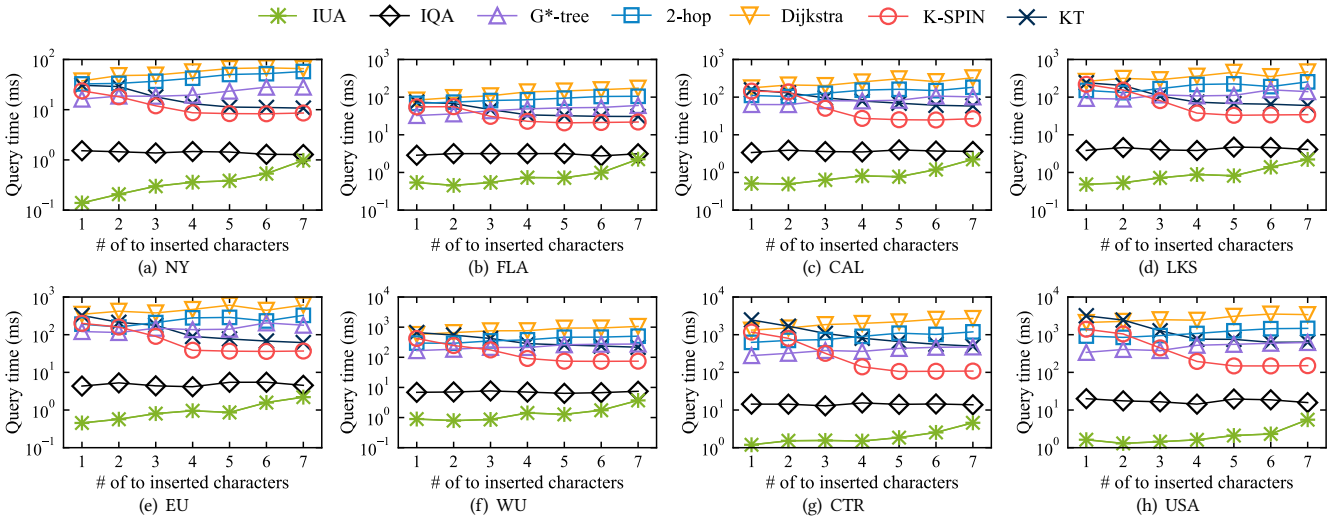


Figure 20: # of inserted characters

deleting characters near the end of $q.str$, less BTags need to be updated.

Exp-12: Effect of # of deleted characters. Then, we verify the effect of deleting different number of characters from $q.str$. In view of this, we randomly choose a deletion position in $q.str$, and delete a certain number of characters, which varies from 1 to 7. Empirical results are plotted in Figure 22. We have the following observations. With the growth of the number of deleted characters, (1) the query time of naive methods drops while that of K-SPIN and KT increases, (2) the query time of IQA remains unchanged, and (3) the query time of IUA increases slowly. However, IUA is still able to handle the deletion of characters efficiently, and outperforms other algorithms by 1-2 orders of magnitude.

Exp-13: Instant query simulation. This experiment simulates the users' instant queries, i.e., to simulate users type in the query string character-by-character. As soon as a character is typed in, we perform the query/update algorithms for the new query string, and return the results. Finally, after the users type in the complete query string, we report the total query/update time, as depicted in Figure 23. Note that, the length of query string for this experiment changes from 1 to 7. As expected, the total query time of all algorithms increase if the length of query string becomes larger. Nevertheless, the total query time of other algorithms ascends much faster than that of both IUA and IQA. This is because the query information inheritance mechanism of IUA makes it be able to perform update incrementally. In addition, the performance of IUA is much better than other algorithms. For example, as $|q.str| = 7$, IUA

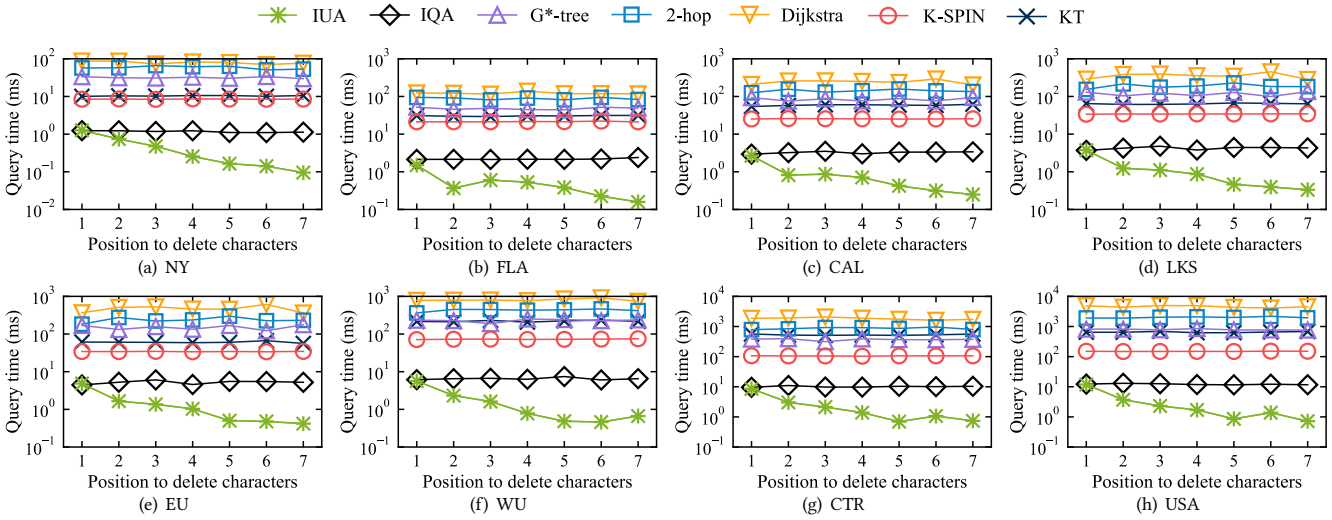


Figure 21: Position to delete characters

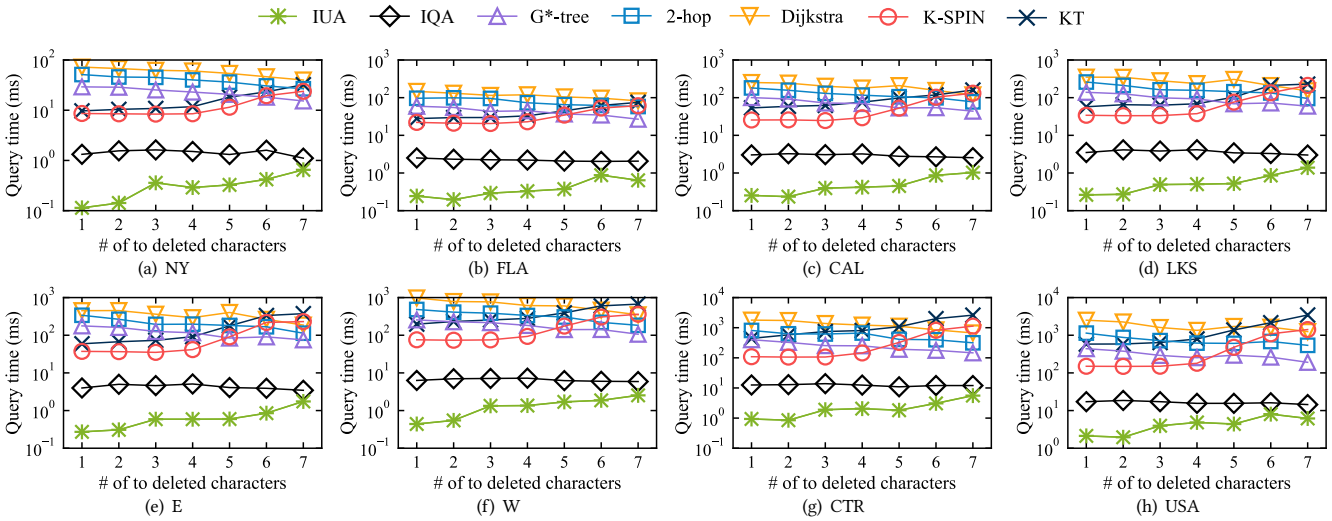


Figure 22: # of deleted characters

can finish the query within 7ms while the fastest algorithm (i.e., the G*-tree) takes 2500ms, confirming the superiority of IUA.

O.3 Index Evaluation

In this section, we evaluate the performance of R2T index, including the index construction and maintenance.

Exp-14: Index Construction. First, we investigate the performance of index construction. In this experiment, we take G*-tree, 2-hop label, K-SPIN, and KT, which are the indexes of other algorithms, as competitors. Figures 24 and 25 show the index construction time and index size, respectively. Note that, in Figure 24, we split the construction time of R2T into two parts, i.e., R2T-I and R2T-II. Specifically, R2T-I represents the time of computing 2-hop label for every vertex, and R2T-II denotes the time of constructing $\widetilde{\mathsf{L}}(v)$ and $\mathsf{T}(v)$ for every vertex after getting 2-hop labels. In Figure 24, we can observe that K-SPIN needs the longest time to construct. For R2T, R2T-I takes up most of the construction time. For example, on the road network CTR, the time of R2T-I and R2T-II are 1289s and 9626.5s , respectively. R2T-I is 11.8% of the total construction time. In Figure 25, G*-tree has the smallest size and colorKT has the largest size. The size of R2T is smaller than that of 2-hop label. This is because we remove some vertices without keywords from the reverse 2-hop label, and the bitmap compression strategy is effective.

We tested the multi-threaded construction capabilities of our proposed index, R2T, across eight datasets using 1, 4, 8, 16, and 32 threads. Figure 26 shows the acceleration ratio, which is the ratio of single-threaded time to multi-threaded time, displayed on each bar. Our results demonstrate a significant speedup in the index-building process with the use of multiple threads. For instance, with 32 threads, the index-building time can be reduced by nearly ten times across datasets. In addition, we experimented with the

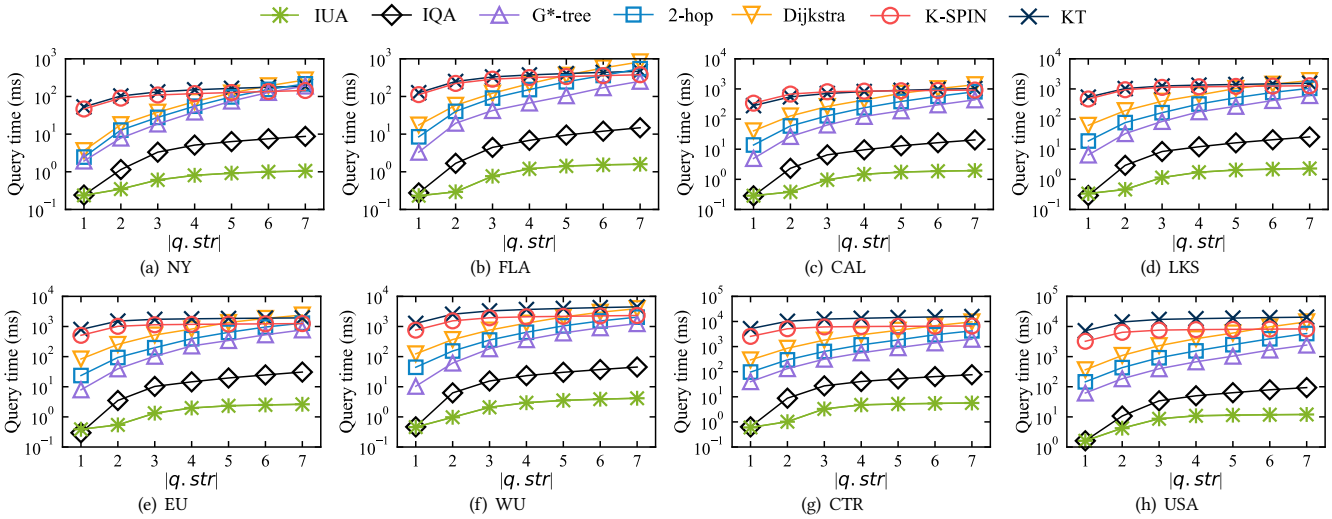


Figure 23: Intant query simulation

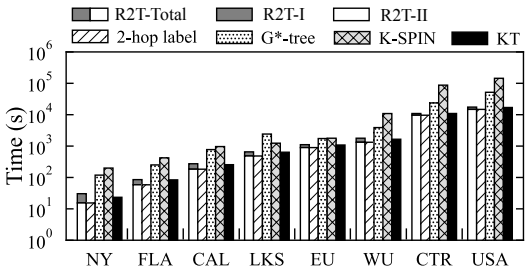


Figure 24: Construction time

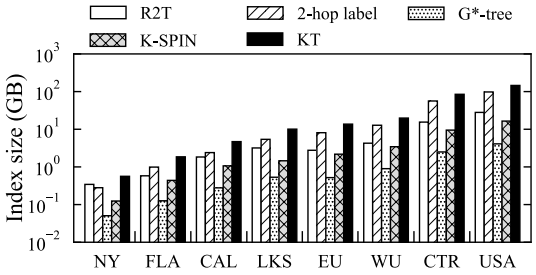


Figure 25: Index size

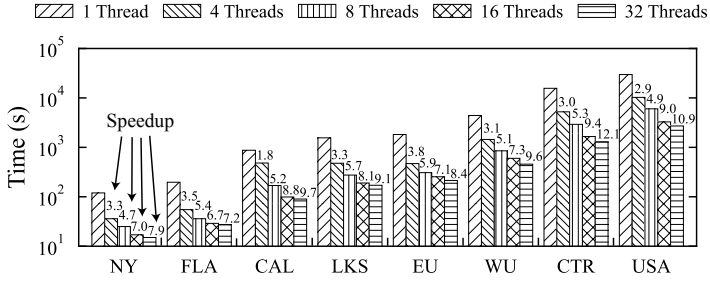


Figure 26: Construction time of multi-threads

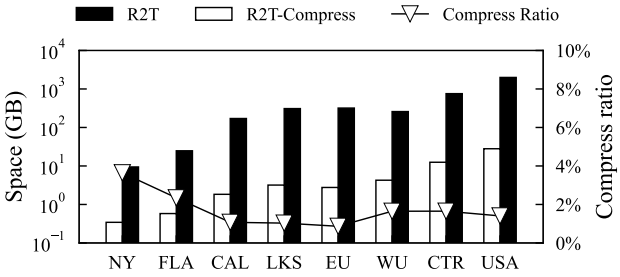


Figure 27: Index size of compression

effectiveness of our compression method for bitmap in Section 5.1 using real datasets. Figure 27 shows the index size before and after compression, along with the compression ratio achieved, which was around 2% across different datasets. This method removes redundant information from the bitmap, greatly reducing the size of the R2T index and improving its efficiency and scalability in practice. The compression ratio achieved by our method makes our R2T index particularly suitable for large road networks, significantly reducing the index size without sacrificing query performance.

Exp-15: Index maintenance. Next, we explore the performance of index maintenance. We mainly consider two categories of the road network update, including keywords update and road network structure update. The keywords update is to insert/delete a certain number of keywords into/from $\mathtt{doc}(v)$ of a vertex v. Figure 28 plots the index maintenance time for the keywords update. The road network structure update contains the update of edges' weight and the insertion/deletion of edges/vertices. Since the insertion/deletion of edges/vertices can be reduced to the update of edges' weight [41], we mainly consider the update of edges' weight in this experiment. In particular, the update of edges' weight include the cases of increasing and decreasing the weight of an edge. For each road network, we select 1000 edges at random and change their weights. The maintenance time of road network structure update is shown in Figure 29. Overall, in both Figures 28 and 29, the maintenance time is much smaller than the time of rebuilding the index, demonstrating the efficiency of our proposed algorithms.

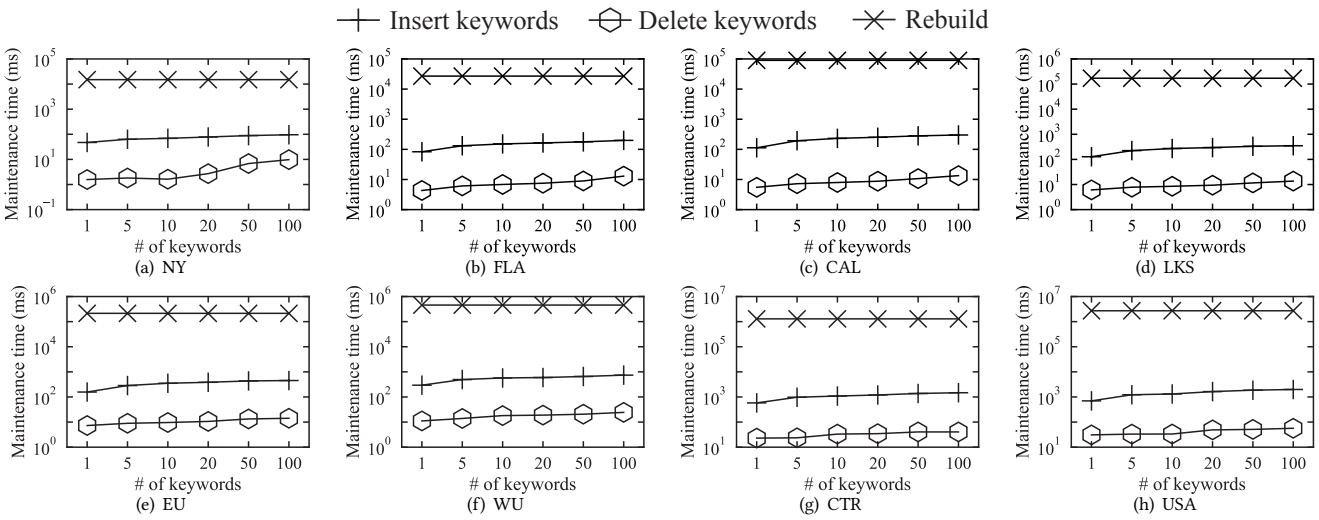


Figure 28: Keywords update

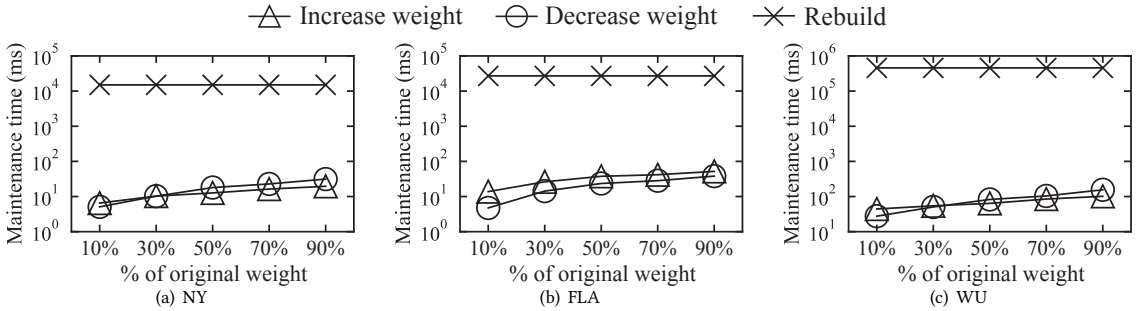


Figure 29: Road network structure update